R: A Language and Environment for Statistical Computing

Reference Index

The R Core Team

Version 4.4.1 (2024-06-14)

Chapter 1

The base package

---

base-package *The R Base Package*

Description

Base R functions

Details

This package contains the basic functions which let R function as a language: arithmetic, input/output, basic programming support, etc. Its contents are available through inheritance from any environment.

For a complete list of functions, use `library(help = "base")`.

.bincode *Bin a Numeric Vector*

Description

Bin a numeric vector and return integer codes for the binning.

Usage

```
.bincode(x, breaks, right = TRUE, include.lowest = FALSE)
```

Arguments

x	a numeric vector which is to be converted to integer codes by binning.
breaks	a numeric vector of two or more cut points, sorted in increasing order.
right	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
include.lowest	logical, indicating if an 'x[i]' equal to the lowest (or highest, for right = FALSE) 'breaks' value should be included in the first (or last) bin.

Details

This is a 'barebones' version of cut.default(labels = FALSE) intended for use in other functions which have checked the arguments passed. (Note the different order of the arguments they have in common.)

Unlike cut, the breaks do not need to be unique. An input can only fall into a zero-length interval if it is closed at both ends, so only if include.lowest = TRUE and it is the first (or last for right = FALSE) interval.

Value

An integer vector of the same length as x indicating which bin each element falls into (the leftmost bin being bin 1). NaN and NA elements of x are mapped to NA codes, as are values outside range of breaks.

See Also

cut, tabulate

Examples

```
## An example with non-unique breaks:
x <- c(0, 0.01, 0.5, 0.99, 1)
b <- c(0, 0, 1, 1)
.bincode(x, b, TRUE)
.bincode(x, b, FALSE)
.bincode(x, b, TRUE, TRUE)
.bincode(x, b, FALSE, TRUE)
```

.Device *Lists of Open/Active Graphics Devices*

Description

A pairlist of the names of open graphics devices is stored in .Devices. The name of the active device (see dev.cur) is stored in .Device. Both are symbols and so appear in the base namespace.

Usage

```
.Device
.Devices
```

Details

`.Device` is a length-one character vector.

`.Devices` is a pairlist of length-one character vectors. The first entry is always `"null device"`, and there are as many entries as the maximal number of graphics devices which have been simultaneously active. If a device has been removed, its entry will be `""` until the device number is reused.

Devices may add attributes to the character vector: for example devices which write to a file may record its path in attribute `"filepath"`.

.Machine *Numerical Characteristics of the Machine*

Description

`.Machine` is a variable holding information on the numerical characteristics of the machine R is running on, such as the largest double or integer and the machine's precision.

Usage

```
.Machine
```

Details

The algorithm is based on Cody's (1988) subroutine `MACHAR`. As all current implementations of R use 32-bit integers and use IEC 60559 floating-point (double precision) arithmetic, the `"integer"` and `"double"` related values are the same for almost all R builds.

Note that on most platforms smaller positive values than `.Machine$double.xmin` can occur. On a typical R platform the smallest positive double is about `5e-324`.

Value

A list with components

`double.eps` the smallest positive floating-point number `x` such that `1 + x != 1`. It equals `double.base ^ ulp.digits` if either `double.base` is 2 or `double.rounding` is 0; otherwise, it is `(double.base ^ double.ulp.digits) / 2`. Normally `2.220446e-16`.

`double.neg.eps` a small positive floating-point number `x` such that `1 - x != 1`. It equals `double.base ^ double.neg.ulp.digits` if `double.base` is 2 or `double.rounding` is 0; otherwise, it is `(double.base ^ double.neg.ulp.digits) / 2`. Normally `1.110223e-16`. As `double.neg.ulp.digits` is bounded below by `-(double.digits + 3)`, `double.neg.eps` may not be the smallest number that can alter 1 by subtraction.

`double.xmin` the smallest non-zero normalized floating-point number, a power of the radix, i.e., `double.base ^ double.min.exp`. Normally `2.225074e-308`.

`double.xmax` the largest normalized floating-point number. Typically, it is equal to `(1 - double.neg.eps) * double.base ^ double.max.exp`, but on some machines it is only the second or third largest such number, being too small by 1 or 2 units in the last digit of the significand. Normally `1.797693e+308`. Note that larger unnormalized numbers can occur.

`double.base` the radix for the floating-point representation: normally `2`.

`double.digits` the number of base digits in the floating-point significand: normally `53`.

`double.rounding`
the rounding action, one of
0 if floating-point addition chops;
1 if floating-point addition rounds, but not in the IEEE style;
2 if floating-point addition rounds in the IEEE style;
3 if floating-point addition chops, and there is partial underflow;
4 if floating-point addition rounds, but not in the IEEE style, and there is partial underflow;
5 if floating-point addition rounds in the IEEE style, and there is partial underflow.
Normally `5`.

`double.guard` the number of guard digits for multiplication with truncating arithmetic. It is 1 if floating-point arithmetic truncates and more than `double digits` base-`double.base` digits participate in the post-normalization shift of the floating-point significand in multiplication, and 0 otherwise.
Normally `0`.

`double.ulp.digits`
the largest negative integer `i` such that `1 + double.base ^ i != 1`, except that it is bounded below by `-(double.digits + 3)`. Normally `-52`.

`double.neg.ulp.digits`
the largest negative integer `i` such that `1 - double.base ^ i != 1`, except that it is bounded below by `-(double.digits + 3)`. Normally `-53`.

`double.exponent`
the number of bits (decimal places if `double.base` is 10) reserved for the representation of the exponent (including the bias or sign) of a floating-point number. Normally `11`.

`double.min.exp` the largest in magnitude negative integer `i` such that `double.base ^ i` is positive and normalized. Normally `-1022`.

`double.max.exp` the smallest positive power of `double.base` that overflows. Normally `1024`.

`integer.max` the largest integer which can be represented. Always $2^{31} - 1 = 2147483647$.

`sizeof.long` the number of bytes in a C '`long`' type: `4` or `8` (most 64-bit systems, but not Windows).

`sizeof.longlong`
the number of bytes in a C '`long long`' type. Will be zero if there is no such type, otherwise usually `8`.

`sizeof.longdouble`
the number of bytes in a C '`long double`' type. Will be zero if there is no such type (or its use was disabled when R was built), otherwise possibly `12` (most 32-bit builds), `16` (most 64-bit builds) or 8 (CPUs such as ARM where for most compilers '`long double`' is identical to `double`).

`sizeof.pointer` the number of bytes in the C `SEXP` type. Will be `4` on 32-bit builds and `8` on 64-bit builds of R.

`sizeof.time_t` the number of *bytes* in the C `time_t` type: a 64-bit `time_t` (value `8`) is much preferred these days. Note that this is the type used by code in R itself, not necessarily the *system* type if R was configured with '`--with-internal-tzcode`' as also used on Windows.

`longdouble.eps`, `longdouble.neg.eps`, `longdouble.digits`, `...`
introduced in R 4.0.0. When `capabilities("long.double")` is true, there are 10 such `"longdouble.`*kind*`"` values, specifying the '`long double`' property corresponding to its `"double.*"` counterpart. See also 'Note'.

Note

In the (typical) case where `capabilities("long.double")` is true, R uses the '`long double`' C type in quite a few places internally for accumulators in e.g. `sum`, reading non-integer numeric constants into (binary) double precision numbers, or arithmetic such as `x %% y`; also, '`long double`' can be read by `readBin`.
For this reason, in that case, `.Machine` contains ten further components, `longdouble.eps`, `*.neg.eps`, `*.digits`, `*.rounding` `*.guard`, `*.ulp.digits`, `*.neg.ulp.digits`, `*.exponent`, `*.min.exp`, and `*.max.exp`, computed entirely analogously to their `double.*` counterparts, see there.

`sizeof.longdouble` only tells you the amount of storage allocated for a long double. Often what is stored is the 80-bit extended double type of IEC 60559, padded to the double alignment used on the platform — this seems to be the case for the common R platforms using ix86 and x86_64 chips. There are other implementation of long double, usually in software for example on Sparc Solaris and AIX.

Note that it is legal for a platform to have a '`long double`' C type which is identical to the '`double`' type — this happens on ARM CPUs. In that case `capabilities("long.double")` will be false but on versions of R prior to 4.0.4, `.Machine` may contain `"longdouble.`*kind*`"` elements.

Source

Uses a C translation of Fortran code in the reference, modified by the R Core Team to defeat over-optimization in modern compilers.

References

Cody, W. J. (1988). MACHAR: A subroutine to dynamically determine machine parameters. *Transactions on Mathematical Software*, **14**(4), 303–311. doi:10.1145/50063.51907.

See Also

`.Platform` for details of the platform.

Examples

```
.Machine
## or for a neat printout
noquote(unlist(format(.Machine)))
```

`.Platform` *Platform Specific Variables*

Description

`.Platform` is a list with some details of the platform under which R was built. This provides means to write OS-portable R code.

Usage

```
.Platform
```

Value

A list with at least the following components:

`OS.type`	character string, giving the **O**perating **S**ystem (family) of the computer. One of `"unix"` or `"windows"`.
`file.sep`	character string, giving the **file sep**arator used on your platform: `"/"` on both Unix-alikes *and* on Windows (but not on the former port to Classic Mac OS).
`dynlib.ext`	character string, giving the file name **ext**ension of **dyn**amically loadable **lib**raries, e.g., `".dll"` on Windows and `".so"` or `".sl"` on Unix-alikes. (Note for macOS users: these are shared objects as loaded by `dyn.load` and not dylibs: see `dyn.load`.)
`GUI`	character string, giving the type of GUI in use, or `"unknown"` if no GUI can be assumed. Possible values are for Unix-alikes the values given via the '`-g`' command-line flag (`"X11"`, `"Tk"`), `"AQUA"` (running under `R.app` on macOS), `"Rgui"` and `"RTerm"` (Windows) and perhaps others under alternative front-ends or embedded R.
`endian`	character string, `"big"` or `"little"`, giving the 'endianness' of the processor in use. This is relevant when it is necessary to know the order to read/write bytes of e.g. an integer or double from/to a connection: see `readBin`.

pkgType — character string, the preferred setting for `options("pkgType")`. Values `"source"`, `"mac.binary"` and `"win.binary"` are currently in use.

This should **not** be used to identify the OS.

path.sep — character string, giving the **path sep**arator, used on your platform, e.g., `":"` on Unix-alikes and `";"` on Windows. Used to separate paths in environment variables such as `PATH` and `TEXINPUTS`.

r_arch — character string, possibly `""`. The name of an architecture-specific directory used in this build of R.

AQUA

`.Platform$GUI` is set to `"AQUA"` under the macOS GUI, `R.app`. This has a number of consequences:

- '`/usr/local/bin`' is *appended* to the `PATH` environment variable.
- the default graphics device is set to `quartz`.
- selects native (rather than Tk) widgets for the `graphics = TRUE` options of `menu` and `select.list`.
- HTML help is displayed in the internal browser.
- the spreadsheet-like data editor/viewer uses a Quartz version rather than the X11 one.

See Also

`R.version` and `Sys.info` give more details about the OS. In particular, `R.version$platform` is the canonical name of the platform under which R was compiled. `osVersion` may give more details about the platform R is running on.

`.Machine` for details of the arithmetic used, and `system` for invoking platform-specific system commands.

`capabilities` and `extSoftVersion` (and links there) for availability of capabilities partly *external* to R but used from R functions.

Examples

```
## Note: this can be done in a system-independent way by dir.exists()
if(.Platform$OS.type == "unix") {
   system.test <- function(...) system(paste("test", ...)) == 0L
   dir.exists2 <- function(dir)
       sapply(dir, function(d) system.test("-d", d))
   dir.exists2(c(R.home(), "/tmp", "~", "/NO")) # > T T T F
}
```

abbreviate *Abbreviate Strings*

Description

Abbreviate strings to at least `minlength` characters, such that they remain *unique* (if they were), unless `strict = TRUE`.

Usage

```
abbreviate(names.arg, minlength = 4, use.classes = TRUE,
           dot = FALSE, strict = FALSE,
           method = c("left.kept", "both.sides"), named = TRUE)
```

Arguments

`names.arg`	a character vector of names to be abbreviated, or an object to be coerced to a character vector by `as.character`.
`minlength`	the minimum length of the abbreviations.
`use.classes`	logical: should lowercase characters be removed first?
`dot`	logical: should a dot (`"."`) be appended?
`strict`	logical: should `minlength` be observed strictly? Note that setting `strict = TRUE` may return *non*-unique strings.
`method`	a character string specifying the method used with default `"left.kept"`, see 'Details' below. Partial matches allowed.
`named`	logical: should `names` (with original vector) be returned.

Details

The default algorithm (`method = "left.kept"`) used is similar to that of S. For a single string it works as follows. First spaces at the ends of the string are stripped. Then (if necessary) any other spaces are stripped. Next, lower case vowels are removed followed by lower case consonants. Finally if the abbreviation is still longer than `minlength` upper case letters and symbols are stripped.

Characters are always stripped from the end of the strings first. If an element of `names.arg` contains more than one word (words are separated by spaces) then at least one letter from each word will be retained.

Missing (`NA`) values are unaltered.

If `use.classes` is `FALSE` then the only distinction is to be between letters and space.

Value

A character vector containing abbreviations for the character strings in its first argument. Duplicates in the original `names.arg` will be given identical abbreviations. If any non-duplicated elements have the same `minlength` abbreviations then, if `method = "both.sides"` the basic internal `abbreviate()` algorithm is applied to the characterwise *reversed* strings; if there are still duplicated abbreviations and if `strict = FALSE` as by default, `minlength` is incremented by one and new abbreviations are found for those elements only. This process is repeated until all unique elements of `names.arg` have unique abbreviations.

If `names` is true, the character version of `names.arg` is attached to the returned value as a `names` attribute: no other attributes are retained.

If a input element contains non-ASCII characters, the corresponding value will be in UTF-8 and marked as such (see `Encoding`).

Warning

If `use.classes` is true (the default), this is really only suitable for English, and prior to R 3.3.0 did not work correctly with non-ASCII characters in multibyte locales. It will warn if used with non-ASCII characters (and required to reduce the length). It is unlikely to work well with inputs not in the Unicode Basic Multilingual Plane nor on (rare) platforms where wide characters are not encoded in Unicode.

As from R 3.3.0 the concept of 'vowel' is extended from English vowels by including characters which are accented versions of lower-case English vowels (including 'o with stroke'). Of course, there are languages (even Western European languages such as Welsh) with other vowels.

See Also

`substr`.

Examples

```
x <- c("abcd", "efgh", "abce")
abbreviate(x, 2)
abbreviate(x, 2, strict = TRUE) # >> 1st and 3rd are == "ab"

(st.abb <- abbreviate(state.name, 2))
stopifnot(identical(unname(st.abb),
           abbreviate(state.name, 2, named=FALSE)))
table(nchar(st.abb)) # out of 50, 3 need 4 letters :
as <- abbreviate(state.name, 3, strict = TRUE)
as[which(as == "Mss")]

## and without distinguishing vowels:
st.abb2 <- abbreviate(state.name, 2, FALSE)
cbind(st.abb, st.abb2)[st.abb2 != st.abb, ]

## method = "both.sides" helps:  no 4-letters, and only 4 3-letters:
st.ab2 <- abbreviate(state.name, 2, method = "both")
table(nchar(st.ab2))
## Compare the two methods:
```

```
cbind(st.abb, st.ab2)
```

---

agrep *Approximate String Matching (Fuzzy Matching)*

Description

Searches for approximate matches to `pattern` (the first argument) within each element of the string `x` (the second argument) using the generalized Levenshtein edit distance (the minimal possibly weighted number of insertions, deletions and substitutions needed to transform one string into another).

Usage

```
agrep(pattern, x, max.distance = 0.1, costs = NULL,
      ignore.case = FALSE, value = FALSE, fixed = TRUE,
      useBytes = FALSE)

agrepl(pattern, x, max.distance = 0.1, costs = NULL,
       ignore.case = FALSE, fixed = TRUE, useBytes = FALSE)
```

Arguments

`pattern`	a non-empty character string to be matched. For `fixed = FALSE` this should contain an extended regular expression. Coerced by `as.character` to a string if possible.
`x`	character vector where matches are sought. Coerced by `as.character` to a character vector if possible.
`max.distance`	maximum distance allowed for a match. Expressed either as integer, or as a fraction of the *pattern* length times the maximal transformation cost (will be replaced by the smallest integer not less than the corresponding fraction), or a list with possible components `cost`: maximum number/fraction of match cost (generalized Levenshtein distance) `all`: maximal number/fraction of *all* transformations (insertions, deletions and substitutions) `insertions`: maximum number/fraction of insertions `deletions`: maximum number/fraction of deletions `substitutions`: maximum number/fraction of substitutions If `cost` is not given, `all` defaults to 10%, and the other transformation number bounds default to `all`. The component names can be abbreviated.
`costs`	a numeric vector or list with names partially matching 'insertions', 'deletions' and 'substitutions' giving the respective costs for computing the generalized Levenshtein distance, or `NULL` (default) indicating using unit cost for all three possible transformations. Coerced to integer via `as.integer` if possible.

ignore.case	if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
value	if FALSE, a vector containing the (integer) indices of the matches determined is returned and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE (default), the pattern is matched literally (as is). Otherwise, it is matched as a regular expression.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See 'Details'.

Details

The Levenshtein edit distance is used as measure of approximateness: it is the (possibly cost-weighted) total number of insertions, deletions and substitutions required to transform one string into another.

This uses the tre code by Ville Laurikari (https://github.com/laurikari/tre), which supports MBCS character matching.

The main effect of useBytes = TRUE is to avoid errors/warnings about invalid inputs and spurious matches in multibyte locales. It inhibits the conversion of inputs with marked encodings, and is forced if any input is found which is marked as "bytes" (see Encoding).

Value

agrep returns a vector giving the indices of the elements that yielded a match, or, if value is TRUE, the matched elements (after coercion, preserving names but no other attributes).

agrepl returns a logical vector.

Note

Since someone who read the description carelessly even filed a bug report on it, do note that this matches substrings of each element of x (just as grep does) and **not** whole elements. See also adist in package **utils**, which optionally returns the offsets of the matched substrings.

Author(s)

Original version in R < 2.10.0 by David Meyer. Current version by Brian Ripley and Kurt Hornik.

See Also

grep, adist. A different interface to approximate string matching is provided by aregexec().

Examples

```
agrep("lasy", "1 lazy 2")
agrep("lasy", c(" 1 lazy 2", "1 lasy 2"), max.distance = list(sub = 0))
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max.distance = 2)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max.distance = 2, value = TRUE)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max.distance = 2, ignore.case = TRUE)
```

`all` *Are All Values True?*

Description

Given a set of logical vectors, are all of the values true?

Usage

```
all(..., na.rm = FALSE)
```

Arguments

`...`	zero or more logical vectors. Other objects of zero length are ignored, and the rest are coerced to logical ignoring any class.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

Coercion of types other than integer (raw, double, complex, character, list) gives a warning as this is often unintentional.

This is a primitive function.

Value

The value is a logical vector of length one.

Let `x` denote the concatenation of all the logical vectors in `...` (after coercion), after removing `NA`s if requested by `na.rm = TRUE`.

The value returned is `TRUE` if all of the values in `x` are `TRUE` (including if there are no values), and `FALSE` if at least one of the values in `x` is `FALSE`. Otherwise the value is `NA` (which can only occur if `na.rm = FALSE` and `...` contains no `FALSE` values and at least one `NA` value).

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

Note

That `all(logical(0))` is true is a useful convention: it ensures that

```
all(all(x), all(y)) == all(x, y)
```

even if `x` has length zero.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

any, the 'complement' of all, and stopifnot(*) which is an all(*) 'insurance'.

Examples

```
range(x <- sort(round(stats::rnorm(10) - 1.2, 1)))
if(all(x < 0)) cat("all x values are negative\n")

all(logical(0))  # true, as all zero of the elements are true.
```

---

all.equal *Test if Two Objects are (Nearly) Equal*

Description

all.equal(x, y) is a utility to compare R objects x and y testing 'near equality'. If they are different, comparison is still made to some extent, and a report of the differences is returned. Do not use all.equal directly in if expressions—either use isTRUE(all.equal(....)) or identical if appropriate.

Usage

```
all.equal(target, current, ...)

## Default S3 method:
all.equal(target, current, ..., check.class = TRUE)

## S3 method for class 'numeric'
all.equal(target, current,
          tolerance = sqrt(.Machine$double.eps), scale = NULL,
          countEQ = FALSE,
          formatFUN = function(err, what) format(err),
          ..., check.attributes = TRUE, check.class = TRUE, giveErr = FALSE)

## S3 method for class 'list'
all.equal(target, current, ...,
          check.attributes = TRUE, use.names = TRUE)

## S3 method for class 'environment'
all.equal(target, current, all.names = TRUE,
          evaluate = TRUE, ...)
```

```
## S3 method for class 'function'
all.equal(target, current, check.environment=TRUE, ...)

## S3 method for class 'POSIXt'
all.equal(target, current, ..., tolerance = 1e-3, scale,
          check.tzone = TRUE)


attr.all.equal(target, current, ...,
               check.attributes = TRUE, check.names = TRUE)
```

Arguments

`target`	R object.
`current`	other R object, to be compared with `target`.
`...`	further arguments for different methods, notably the following two, for numerical comparison:
`tolerance`	numeric ≥ 0. Differences smaller than `tolerance` are not reported. The default value is close to `1.5e-8`.
`scale`	`NULL` or numeric > 0, typically of length 1 or `length(target)`. See 'Details'.
`countEQ`	logical indicating if the `target == current` cases should be counted when computing the mean (absolute or relative) differences. The default, `FALSE` may seem misleading in cases where `target` and `current` only differ in a few places; see the extensive example.
`formatFUN`	a `function` of two arguments, `err`, the relative, absolute or scaled error, and `what`, a character string indicating the *kind* of error; may be used, e.g., to format relative and absolute errors differently.
`check.attributes`	logical indicating if the `attributes` of `target` and `current` (other than the names) should be compared.
`check.class`	logical indicating if the `data.class()` of `target` and `current` should be compared.
`giveErr`	`logical` indicating if the result should contain the numerical error as an `"err"` attribute.
`use.names`	logical indicating if `list` comparison should report differing components by name (if matching) instead of integer index. Note that this comes after `...` and so must be specified by its full name.
`all.names`	logical passed to `ls` indicating if "hidden" objects should also be considered in the environments.
`evaluate`	for the `environment` method: `logical` indicating if "promises should be forced", i.e., typically formal function arguments be evaluated for comparison. If false, only the `names` of the objects in the two environments are checked for equality.

`check.environment`
: logical requiring that the `environment()`s of functions should be compared, too. You may need to set `check.environment=FALSE` in unexpected cases, such as when comparing two `nls()` fits.

`check.tzone`
: logical indicating if the `"tzone"` attributes of `target` and `current` should be compared.

`check.names`
: logical indicating if the `names(.)` of `target` and `current` should be compared.

Details

`all.equal` is a generic function, dispatching methods on the `target` argument. To see the available methods, use `methods("all.equal")`, but note that the default method also does some dispatching, e.g. using the raw method for logical targets.

Remember that arguments which follow `...` must be specified by (unabbreviated) name. It is inadvisable to pass unnamed arguments in `...` as these will match different arguments in different methods.

Numerical comparisons for `scale = NULL` (the default) are typically on a *relative difference* scale unless the `target` values are close to zero or infinite. Specifically, the scale is computed as the mean absolute value of `target`. If this scale is finite and exceeds `tolerance`, differences are expressed relative to it; otherwise, absolute differences are used. Note that this scale and all further steps are computed only for those vector elements where `target` is not `NA` and differs from `current`. If `countEQ` is true, the equal and `NA` cases are *counted* in determining the "sample" size.

If `scale` is numeric (and positive), absolute comparisons are made after scaling (dividing) by `scale`. Note that if all of scale is close to 1 (specifically, within 1e-7), the difference is still reported as being on an absolute scale.

For complex `target`, the modulus (`Mod`) of the difference is used: `all.equal.numeric` is called so arguments `tolerance` and `scale` are available.

The `list` method compares components of `target` and `current` recursively, passing all other arguments, as long as both are "list-like", i.e., fulfill either `is.vector` or `is.list`.

The `environment` method works via the `list` method, and is also used for reference classes (unless a specific `all.equal` method is defined).

The method for date-time objects uses `all.equal.numeric` to compare times (in `"POSIXct"` representation) with a default `tolerance` of 0.001 seconds, ignoring `scale`. A time zone mismatch between `target` and `current` is reported unless `check.tzone = FALSE`.

`attr.all.equal` is used for comparing `attributes`, returning `NULL` or a `character` vector.

Value

Either `TRUE` (`NULL` for `attr.all.equal`) or a vector of `mode` `"character"` describing the differences between `target` and `current`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

identical, isTRUE, ==, and all for exact equality testing.

Examples

```
all.equal(pi, 355/113)
# not precise enough (default tol) > relative error

quarts <- 1/4 + 1:10 # exact
d45 <- pi*quarts ; one <- rep(1, 10)
tan(d45) == one  # mostly FALSE, as typically exact; embarrassingly,
tanpi(quarts) == one # (is always FALSE (Fedora 34; gcc 11.2.1))
stopifnot(all.equal(
          tan(d45), one)) # TRUE, but not if we are picky:
all.equal(tan(d45), one, tolerance = 0)  # to see difference
all.equal(tan(d45), one, tolerance = 0, scale = 1)# "absolute diff.."
all.equal(tan(d45), one, tolerance = 0, scale = 1+(-2:2)/1e9) # "absolute"
all.equal(tan(d45), one, tolerance = 0, scale = 1+(-2:2)/1e6) # "scaled"

## advanced: equality of environments
ae <- all.equal(as.environment("package:stats"),
                asNamespace("stats"))
stopifnot(is.character(ae), length(ae) > 10,
          ## were incorrectly "considered equal" in R <= 3.1.1
          all.equal(asNamespace("stats"), asNamespace("stats")))

## A situation where  'countEQ = TRUE' makes sense:
x1 <- x2 <- (1:100)/10;  x2[2] <- 1.1*x1[2]
## 99 out of 100 pairs (x1[i], x2[i]) are equal:
plot(x1,x2, main = "all.equal.numeric() -- not counting equal parts")
all.equal(x1,x2) ## "Mean relative difference: 0.1"
mtext(paste("all.equal(x1,x2) :", all.equal(x1,x2)), line= -2)
##' extract the 'Mean relative difference' as number:
all.eqNum <- function(...) as.numeric(sub(".*:", '', all.equal(...)))
set.seed(17)
## When x2 is jittered, typically all pairs (x1[i],x2[i]) do differ:
summary(r <- replicate(100, all.eqNum(x1, x2*(1+rnorm(x1)*1e-7))))
mtext(paste("mean(all.equal(x1, x2*(1 + eps_k))) {100 x} Mean rel.diff.=",
            signif(mean(r), 3)), line = -4, adj=0)
## With argument  countEQ=TRUE, get "the same" (w/o need for jittering):
mtext(paste("all.equal(x1,x2, countEQ=TRUE) :",
          signif(all.eqNum(x1,x2, countEQ=TRUE), 3)), line= -6, col=2)

## Using giveErr=TRUE :
x1. <- x1 * (1+ 1e-9*rnorm(x1))
str(all.equal(x1, x1., giveErr=TRUE))
## logi TRUE
## - attr(*,  "err")= num 8.66e-10
## - attr(*, "what")= chr "relative"

## Used with stopifnot(), still *showing* diff:
all.equalShow <- function (...) {
```

```
   r <- all.equal(..., giveErr=TRUE)
   cat(attr(r,"what"), "err:", attr(r,"err"), "\n")
   c(r) # can drop attributes, as not used anymore
}
# checks, showing error in any case:
stopifnot(all.equalShow(x1, x1.)) # -> relative err: 8.66002e-10
tryCatch(error=identity, stopifnot(all.equalShow(x1, 2*x1))) -> eAe
stopifnot(inherits(eAe, "error"))
# stopifnot(all.equal....()) giving smart msg:
cat(conditionMessage(eAe), "\n")

two <- structure(2, foo = 1, class = "bar")
all.equal(two^20, 2^20) # lots of diff
all.equal(two^20, 2^20, check.attributes = FALSE)# "target is bar, current is numeric"
all.equal(two^20, 2^20, check.attributes = FALSE, check.class = FALSE) # TRUE

## comparison of date-time objects
now <- Sys.time()
stopifnot(
all.equal(now, now + 1e-4)  # TRUE (default tolerance = 0.001 seconds)
)
all.equal(now, now + 0.2)
all.equal(now, as.POSIXlt(now, "UTC"))
stopifnot(
all.equal(now, as.POSIXlt(now, "UTC"), check.tzone = FALSE)  # TRUE
)
```

`all.names` *Find All Names in an Expression*

Description

Return a character vector containing all the names which occur in an expression or call.

Usage

```
all.names(expr, functions = TRUE, max.names = -1L, unique = FALSE)

all.vars(expr, functions = FALSE, max.names = -1L, unique = TRUE)
```

Arguments

`expr`	an expression or call from which the names are to be extracted.
`functions`	a logical value indicating whether function names should be included in the result.
`max.names`	the maximum number of names to be returned. `-1` indicates no limit (other than vector size limits).
`unique`	a logical value which indicates whether duplicate names should be removed from the value.

Details

These functions differ only in the default values for their arguments.

Value

A character vector with the extracted names.

See Also

`substitute` to replace symbols with values in an expression.

Examples

```
all.names(expression(sin(x+y)))
all.names(quote(sin(x+y))) # or a call
all.vars(expression(sin(x+y)))
```

---

any *Are Some Values True?*

Description

Given a set of logical vectors, is at least one of the values true?

Usage

```
any(..., na.rm = FALSE)
```

Arguments

`...`	zero or more logical vectors. Other objects of zero length are ignored, and the rest are coerced to logical ignoring any class.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

Coercion of types other than integer (raw, double, complex, character, list) gives a warning as this is often unintentional.

This is a primitive function.

Value

The value is a logical vector of length one.

Let x denote the concatenation of all the logical vectors in ... (after coercion), after removing NAs if requested by na.rm = TRUE.

The value returned is TRUE if at least one of the values in x is TRUE, and FALSE if all of the values in x are FALSE (including if there are no values). Otherwise the value is NA (which can only occur if na.rm = FALSE and ... contains no TRUE values and at least one NA value).

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

all, the 'complement' of any.

Examples

```
range(x <- sort(round(stats::rnorm(10) - 1.2, 1)))
if(any(x < 0)) cat("x contains negative values\n")
```

---

aperm *Array Transposition*

Description

Transpose an array by permuting its dimensions and optionally resizing it.

Usage

```
aperm(a, perm, ...)
## Default S3 method:
aperm(a, perm = NULL, resize = TRUE, ...)
## S3 method for class 'table'
aperm(a, perm = NULL, resize = TRUE, keep.class = TRUE, ...)
```

Arguments

a	the array to be transposed.
perm	the subscript permutation vector, usually a permutation of the integers 1:n, where n is the number of dimensions of a. When a has named dimnames, it can be a character vector of length n giving a permutation of those names. The default (used whenever perm has zero length) is to reverse the order of the dimensions.
resize	a flag indicating whether the vector should be resized as well as having its elements reordered (default TRUE).
keep.class	logical indicating if the result should be of the same class as a.
...	potential further arguments of methods.

Value

A transposed version of array a, with subscripts permuted as indicated by the array perm. If resize is TRUE, the array is reshaped as well as having its elements permuted, the dimnames are also permuted; if resize = FALSE then the returned object has the same dimensions as a, and the dimnames are dropped. In each case other attributes are copied from a.

The function t provides a faster and more convenient way of transposing matrices.

Author(s)

Jonathan Rougier, <J.C.Rougier@durham.ac.uk> did the faster C implementation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

t, to transpose matrices.

Examples

```
# interchange the first two subscripts on a 3-way array x
x  <- array(1:24, 2:4)
xt <- aperm(x, c(2,1,3))
stopifnot(t(xt[,,2]) == x[,,2],
          t(xt[,,3]) == x[,,3],
          t(xt[,,4]) == x[,,4])

UCB <- aperm(UCBAdmissions, c(2,1,3))
UCB[1,,]
summary(UCB) # UCB is still a contingency table
```

append *Vector Merging*

Description

Add elements to a vector.

Usage

```
append(x, values, after = length(x))
```

Arguments

x	the vector the values are to be appended to.
values	to be included in the modified vector.
after	a subscript, after which the values are to be appended.

Value

A vector containing the values in x with the elements of values appended after the specified element of x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
append(1:5, 0:1, after = 3)
```

apply *Apply Functions Over Array Margins*

Description

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Usage

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

Arguments

X	an array, including a matrix.
MARGIN	a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
FUN	the function to be applied: see 'Details'. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.
...	optional arguments to FUN.
simplify	a logical indicating whether results should be simplified if possible.

Details

If X is not an array but an object of a class with a non-null dim value (such as a data frame), apply attempts to coerce it to an array via as.matrix if it is two-dimensional (e.g., a data frame) or via as.array.

FUN is found by a call to match.fun and typically is either a function or a symbol (e.g., a backquoted name) or a character string specifying a function to be searched for from the environment of the call to apply.

Arguments in ... cannot have the same name as any of the other arguments, and care may be needed to avoid partial matching to MARGIN or FUN. In general-purpose code it is good practice to name the first three arguments if ... is passed through: this both avoids partial matching to MARGIN or FUN and ensures that a sensible error message is given if arguments named X, MARGIN or FUN are passed through ....

Value

If each call to FUN returns a vector of length n, and simplify is TRUE, then apply returns an array of dimension c(n, dim(X)[MARGIN]) if n > 1. If n equals 1, apply returns a vector if MARGIN has length 1 and an array of dimension dim(X)[MARGIN] otherwise. If n is 0, the result has length 0 but not necessarily the 'correct' dimension.

If the calls to FUN return vectors of different lengths, or if simplify is FALSE, apply returns a list of length prod(dim(X)[MARGIN]) with dim set to MARGIN if this has length greater than one.

In all cases the result is coerced by as.vector to one of the basic vector types before the dimensions are set, so that (for example) factor results will be coerced to a character array.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

lapply and there, simplify2array; tapply, and convenience functions sweep and aggregate.

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x, 2, is.vector))

## Sort the columns of a matrix
apply(x, 2, sort)

## keeping named dimnames
names(dimnames(x)) <- c("row", "col")
x3 <- array(x, dim = c(dim(x),3),
     dimnames = c(dimnames(x), list(C = paste0("cop.",1:3))))
identical(x,  apply( x,  2,  identity))
identical(x3, apply(x3, 2:3, identity))

##- function with extra args:
cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x, 1, cave,  c1 = "x1", c2 = c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nrow = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, stats::quantile) # 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))

## Example with different lengths for each call
z <- array(1:24, dim = 2:4)
zseq <- apply(z, 1:2, function(x) seq_len(max(x)))
zseq         ## a 2 x 3 matrix
typeof(zseq) ## list
dim(zseq) ## 2 3
zseq[1,]
apply(z, 3, function(x) seq_len(max(x)))
# a list without a dim attribute
```

`args` *Argument List of a Function*

Description

Displays the argument names and corresponding default values of a (non-primitive or primitive) function.

Usage

```
args(name)
```

Arguments

name	a function (a primitive or a closure, i.e., "non-primitive"). If name is a character string then the function with that name is found and used.

Details

This function is mainly used interactively to print the argument list of a function. For programming, consider using formals instead.

Value

For a closure, a closure with identical formal argument list but an empty (NULL) body.

For a primitive (function), a closure with the documented usage and NULL body. Note that some primitives do not make use of named arguments and match by position rather than name.

NULL in case of a non-function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

formals, help; str also prints the argument list of a function.

Examples

```
## "regular" (non-primitive) functions "print their arguments"
## (by returning another function with NULL body which you also see):
args(ls)
args(graphics::plot.default)
utils::str(ls) # (just "prints": does not show a NULL)

## You can also pass a string naming a function.
args("scan")
## ...but :: package specification doesn't work in this case.
tryCatch(args("graphics::plot.default"), error = print)

## As explained above, args() gives a function with empty body:
list(is.f = is.function(args(scan)), body = body(args(scan)))

## Primitive functions mostly behave like non-primitive functions.
args(c)
args(`+`)
## primitive functions without well-defined argument list return NULL:
args(`if`)
```

`Arithmetic` *Arithmetic Operators*

Description

These unary and binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

Usage

```
+ x
- x
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

Arguments

`x, y`	numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written.

Details

The unary and binary arithmetic operators are generic functions: methods can be written for them individually or via the `Ops` group generic function. (See `Ops` for how dispatch is computed.)

If applied to arrays the result will be an array if this is sensible (for example it will not if the recycling rule has been invoked).

Logical vectors will be coerced to integer or numeric vectors, `FALSE` having value zero and `TRUE` having value one.

`1 ^ y` and `y ^ 0` are `1`, *always*. `x ^ y` should also give the proper limit result when either (numeric) argument is infinite (one of `Inf` or `-Inf`).

Objects such as arrays or time-series can be operated on this way provided they are conformable.

For double arguments, `%%` can be subject to catastrophic loss of accuracy if `x` is much larger than `y`, and a warning is given if this is detected.

`%%` and `x %/% y` can be used for non-integer `y`, e.g. `1 %/% 0.2`, but the results are subject to representation error and so may be platform-dependent. Mathematically, the answer to `1 %/% 0.2` should be `5`, but because the IEC 60559 representation of `0.2` is a binary fraction slightly larger than `0.2` most platforms give `4`.

Users are sometimes surprised by the value returned, for example why `(-8)^(1/3)` is `NaN`. For double inputs, R makes use of IEC 60559 arithmetic on all platforms, together with the C system function 'pow' for the `^` operator. The relevant standards define the result in many corner cases. In

particular, the result in the example above is mandated by the C99 standard. On many Unix-alike systems the command `man pow` gives details of the values in a large number of corner cases.

Arithmetic on type double in R is supposed to be done in 'round to nearest, ties to even' mode, but this does depend on the compiler and FPU being set up correctly.

Value

Unary `+` and unary `-` return a numeric or complex vector. All attributes (including class) are preserved if there is no coercion: logical `x` is coerced to integer and names, dims and dimnames are preserved.

The binary operators return vectors containing the result of the element by element operations. If involving a zero-length vector the result has length zero. Otherwise, the elements of shorter vectors are recycled as necessary (with a `warning` when they are recycled only *fractionally*). The operators are `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division and `^` for exponentiation.

`%%` indicates `x mod y` ("x modulo y"), i.e., computes the 'remainder' `r <- x %% y`, and `%/%` indicates integer division, where R uses "floored" integer division, i.e., `q <- x %/% y` := `floor(x/y)`, as promoted by Donald Knuth, see the Wikipedia page on 'Modulo operation', and hence `sign(r) == sign(y)`. It is guaranteed that

```
x == (x %% y) + y * (x %/% y)     (up to rounding error)
```

unless `y == 0` where the result of `%%` is `NA_integer_` or `NaN` (depending on the `typeof` of the arguments) or for some non-finite arguments, e.g., when the RHS of the identity above amounts to `Inf - Inf`.

If either argument is complex the result will be complex, otherwise if one or both arguments are numeric, the result will be numeric. If both arguments are of type integer, the type of the result of `/` and `^` is numeric and for the other operators it is integer (with overflow, which occurs at $\pm(2^{31}-1)$, returned as `NA_integer_` with a warning).

The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. If the arguments are the same length, attributes will be copied from both, with those of the first argument taking precedence when the same attribute is present in both arguments. For time series, these operations are allowed only if the series are compatible, when the class and `tsp` attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

S4 methods

These operators are members of the S4 `Arith` group generic, and so methods can be written for them individually as well as for the group generic (or the `Ops` group generic), with arguments `c(e1, e2)` (with `e2` missing for a unary operator).

Implementation limits

R is dependent on OS services (and they on FPUs) for floating-point arithmetic. On all current R platforms IEC 60559 (also known as IEEE 754) arithmetic is used, but some things in those standards are optional. In particular, the support for *denormal* aka *subnormal* numbers (those outside

the range given by `.Machine`) may differ between platforms and even between calculations on a single platform.

Another potential issue is signed zeroes: on IEC 60559 platforms there are two zeroes with internal representations differing by sign. Where possible R treats them as the same, but for example direct output from C code often does not do so and may output '`-0.0`' (and on Windows whether it does so or not depends on the version of Windows). One place in R where the difference might be seen is in division by zero: `1/x` is `Inf` or `-Inf` depending on the sign of zero `x`. Another place is `identical(0, -0, num.eq = FALSE)`.

Note

All logical operations involving a zero-length vector have a zero-length result.

The binary operators are sometimes called as functions as e.g. `` `&`(x, y) ``: see the description of how argument-matching is done in `Ops`.

`**` is translated in the parser to `^`, but this was undocumented for many years. It appears as an index entry in Becker et al. (1988), pointing to the help for `Deprecated` but is not actually mentioned on that page. Even though it had been deprecated in S for 20 years, it was still accepted in R in 2008.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

D. Goldberg (1991). What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Computing Surveys*, **23**(1), 5–48. doi:10.1145/103162.103163.
Also available at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

For the IEC 60559 (aka IEEE 754) standard: https://www.iso.org/standard/57469.html and https://en.wikipedia.org/wiki/IEEE_754.

On the integer division and remainder (modulo) computations, `%%` and `%/%`: https://en.wikipedia.org/wiki/Modulo_operation, and Donald Knuth (1972) *The Art of Computer Programming*, Vol.1.

See Also

`sqrt` for miscellaneous and `Special` for special mathematical functions.

`Syntax` for operator precedence.

`%*%` for matrix multiplication.

Examples

```
x <- -1:12
x + 1
2 * x + 3
x %%  3 # is periodic  2 0  1  2 0  1 ...
x %% -3 #  (ditto)    -1 0 -2 -1 0 -2 ...
x %/% 5
x %% Inf # now is defined by limit (gave NaN in earlier versions of R)

## Illustrating PR#18677, see above
```

```
1 %/% print(0.2, digits=19)
```

`array` *Multi-way Arrays*

Description

Creates or tests for arrays.

Usage

```
array(data = NA, dim = length(data), dimnames = NULL)
as.array(x, ...)
is.array(x)
```

Arguments

`data`	a vector (including a list or `expression` vector) giving data to fill the array. Non-atomic classed objects are coerced by `as.vector`.
`dim`	the dim attribute for the array to be created, that is an integer vector of length one or more giving the maximal indices in each dimension.
`dimnames`	either `NULL` or the names for the dimensions. This must be a list (or it will be ignored) with one component for each dimension, either `NULL` or a character vector of the length given by `dim` for that dimension. The list can be named, and the list names will be used as names for the dimensions. If the list is shorter than the number of dimensions, it is extended by `NULL`s to the length required.
`x`	an R object.
`...`	additional arguments to be passed to or from methods.

Details

An array in R can have one, two or more dimensions. It is simply a vector which is stored with additional attributes giving the dimensions (attribute `"dim"`) and optionally names for those dimensions (attribute `"dimnames"`).

A two-dimensional array is the same thing as a `matrix`.

One-dimensional arrays often look like vectors, but may be handled differently by some functions: `str` does distinguish them in recent versions of R.

The `"dim"` attribute is an integer vector of length one or more containing non-negative values: the product of the values must match the length of the array.

The `"dimnames"` attribute is optional: if present it is a list with one component for each dimension, either `NULL` or a character vector of the length given by the element of the `"dim"` attribute for that dimension.

`is.array` is a primitive function.

For a list array, the `print` methods prints entries of length not one in the form '`integer,7`' indicating the type and length.

Value

array returns an array with the extents specified in dim and naming information in dimnames. The values in data are taken to be those in the array with the leftmost subscript moving fastest. If there are too few elements in data to fill the array, then the elements in data are recycled. If data has length zero, NA of an appropriate type is used for atomic vectors (0 for raw vectors) and NULL for lists.

Unlike matrix, array does not currently remove any attributes left by as.vector from a classed list data, so can return a list array with a class attribute.

as.array is a generic function for coercing to arrays. The default method does so by attaching a dim attribute to it. It also attaches dimnames if x has names. The sole purpose of this is to make it possible to access the dim[names] attribute at a later time.

is.array returns TRUE or FALSE depending on whether its argument is an array (i.e., has a dim attribute of positive length) or not. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Note

is.array is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

aperm, matrix, dim, dimnames.

Examples

```
dim(as.array(letters))
array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
#     [,1] [,2] [,3] [,4]
#[1,]    1    3    2    1
#[2,]    2    1    3    2
```

---

array2DF *Convert array to data frame*

Description

array2DF converts an array, including list arrays commonly returned by tapply, into data frames for use in further analysis or plotting functions.

Usage

```
array2DF(x, responseName = "Value",
         sep = "", base = list(LETTERS),
         simplify = TRUE, allowLong = TRUE)
```

Arguments

x	an array object.
responseName	character string, used for creating column name(s) in the result, if required.
sep	character string, used as separator when creating new names, if required.
base	character vector, giving an initial set of names to create dimnames of x, if missing.
simplify	logical, whether to attempt simplification of the result.
allowLong	logical, specifying whether a long format data frame should be returned if x is a list array and all elements of x are unnamed atomic vectors. Ignored unless simplify = TRUE.

Details

The main use of array2DF is to convert an array, as typically returned by tapply, into a data frame.

When simplify = FALSE, this is similar to as.data.frame.table, except that it works for list arrays as well as atomic arrays. Specifically, the resulting data frame has one row for each element of the array, with one column for each dimension of the array giving the corresponding dimnames. The contents of the array are placed in a column whose name is given by the responseName argument. The mode of this column is the same as that of x, usually an atomic vector or a list.

If x does not have dimnames, they are automatically created using base and sep.

In the default case, when simplify = TRUE, some common cases are handled specially.

If all components of x are data frames with identical column names (with possibly different numbers of rows), they are rbind-ed to form the response. The additional columns giving dimnames are repeated according to the number of rows, and responseName is ignored in this case.

If all components of x are *unnamed* atomic vectors *and* allowLong = TRUE, each component is treated as a single-column data frame with column name given by responseName, and processed as above.

In all other cases, an attempt to simplify is made by simplify2array. If this results in multiple unnamed columns, names are constructed using responseName and sep.

Value

A data frame with at least length(dim(x)) + 1 columns. The first length(dim(x)) columns each represent one dimension of x and gives the corresponding values of dimnames, which are implicitly created if necessary. The remaining columns contain the contents of x, after attempted simplification if requested.

See Also

tapply, as.data.frame.table, split, aggregate.

Examples

```
s1 <- with(ToothGrowth,
           tapply(len, list(dose, supp), mean, simplify = TRUE))

s2 <- with(ToothGrowth,
           tapply(len, list(dose, supp), mean, simplify = FALSE))

str(s1) # atomic array
str(s2) # list array

str(array2DF(s1, simplify = FALSE)) # Value column is vector
str(array2DF(s2, simplify = FALSE)) # Value column is list
str(array2DF(s2, simplify = TRUE))  # simplified to vector

### The remaining examples use the default 'simplify = TRUE'

## List array with list components: columns are lists (no simplification)

with(ToothGrowth,
     tapply(len, list(dose, supp),
     function(x) t.test(x)[c("p.value", "alternative")])) |>
  array2DF() |> str()

## List array with data frame components: columns are atomic (simplified)

with(ToothGrowth,
     tapply(len, list(dose, supp),
     function(x) with(t.test(x), data.frame(p.value, alternative)))) |>
  array2DF() |> str()

## named vectors

with(ToothGrowth,
     tapply(len, list(dose, supp),
            quantile)) |> array2DF()

## unnamed vectors: long format

with(ToothGrowth,
     tapply(len, list(dose, supp),
            sample, size = 5)) |> array2DF()

## unnamed vectors: wide format

with(ToothGrowth,
     tapply(len, list(dose, supp),
            sample, size = 5)) |> array2DF(allowLong = FALSE)

## unnamed vectors of unequal length

with(ToothGrowth[-1, ],
     tapply(len, list(dose, supp),
```

```
            sample, replace = TRUE)) |>
  array2DF(allowLong = FALSE)

## unnamed vectors of unequal length with allowLong = TRUE
## (within-group bootstrap)

with(ToothGrowth[-1, ],
     tapply(len, list(dose, supp), sample, replace = TRUE)) |>
  array2DF() |> str()

## data frame input

tapply(ToothGrowth, ~ dose + supp, FUN = with,
       data.frame(n = length(len), mean = mean(len), sd = sd(len))) |>
  array2DF()
```

---

as.data.frame *Coerce to a Data Frame*

Description

Functions to check if an object is a data frame, or coerce it if possible.

Usage

```
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'character'
as.data.frame(x, ...,
              stringsAsFactors = FALSE)

## S3 method for class 'list'
as.data.frame(x, row.names = NULL, optional = FALSE, ...,
              cut.names = FALSE, col.names = names(x), fix.empty.names = TRUE,
              check.names = !optional,
              stringsAsFactors = FALSE)

## S3 method for class 'matrix'
as.data.frame(x, row.names = NULL, optional = FALSE,
              make.names = TRUE, ...,
              stringsAsFactors = FALSE)

as.data.frame.vector(x, row.names = NULL, optional = FALSE, ...,
                     nm = deparse1(substitute(x)))

is.data.frame(x)
```

Arguments

x	any R object.
row.names	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
optional	logical. If TRUE, setting row names and converting column names (to syntactic names: see make.names) is optional. Note that all of R's **base** package as.data.frame() methods use optional only for column names treatment, basically with the meaning of data.frame(*, check.names = !optional). See also the make.names argument of the matrix method.
...	additional arguments to be passed to or from methods.
stringsAsFactors	logical: should the character vector be converted to a factor?
cut.names	logical or integer; indicating if column names with more than 256 (or cut.names if that is numeric) characters should be shortened (and the last 6 characters replaced by " ...").
col.names	(optional) character vector of column names.
fix.empty.names	logical indicating if empty column names, i.e., "" should be fixed up (in data.frame) or not.
check.names	logical; passed to the data.frame() call.
make.names	a logical, i.e., one of FALSE, NA, TRUE, indicating what should happen if the row names (of the matrix x) are invalid. If they are invalid, the default, TRUE, calls make.names(*, unique=TRUE); make.names=NA will use "automatic" row names and a FALSE value will signal an error for invalid row names.
nm	a character string to be used as column name.

Details

as.data.frame is a generic function with many methods, and users and packages can supply further methods. For classes that act as vectors, often a copy of as.data.frame.vector will work as the method.

Since R 4.3.0, the *default* method will call as.data.frame.vector for atomic (as by is.atomic) x.

Direct calls of as.data.frame.*class* are still possible (base package!), for 12 atomic base classes, but are deprecated where calling as.data.frame.vector instead is recommended.

If a list is supplied, each element is converted to a column in the data frame. Similarly, each column of a matrix is converted separately. This can be overridden if the object has a class which has a method for as.data.frame: two examples are matrices of class "model.matrix" (which are included as a single column) and list objects of class "POSIXlt" which are coerced to class "POSIXct".

Arrays can be converted to data frames. One-dimensional arrays are treated like vectors and two-dimensional arrays like matrices. Arrays with more than two dimensions are converted to matrices by 'flattening' all dimensions after the first and creating suitable column labels.

Character variables are converted to factor columns unless protected by I.

If a data frame is supplied, all classes preceding "data.frame" are stripped, and the row names are changed if that argument is supplied.

If row.names = NULL, row names are constructed from the names or dimnames of x, otherwise are the integer sequence starting at one. Few of the methods check for duplicated row names. Names are removed from vector columns unless I.

Value

as.data.frame returns a data frame, normally with all row names "" if optional = TRUE.

is.data.frame returns TRUE if its argument is a data frame (that is, has "data.frame" amongst its classes) and FALSE otherwise.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, as.data.frame.table for the table method (which has additional arguments if called directly).

---

as.Date *Date Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of class "Date" representing calendar dates.

Usage

```
as.Date(x, ...)
## S3 method for class 'character'
as.Date(x, format, tryFormats = c("%Y-%m-%d", "%Y/%m/%d"),
        optional = FALSE, ...)
## S3 method for class 'numeric'
as.Date(x, origin, ...)
## S3 method for class 'POSIXct'
as.Date(x, tz = "UTC", ...)

## S3 method for class 'Date'
format(x, format = "%Y-%m-%d", ...)

## S3 method for class 'Date'
as.character(x, ...)
```

Arguments

x	an object to be converted.
format	a character string. If not specified when converting from a character representation, it will try tryFormats one by one on the first non-NA element, and give an error if none works. Otherwise, the processing is via strptime() whose help page describes available conversion specifications.
tryFormats	character vector of format strings to try if format is not specified.
optional	logical indicating to return NA (instead of signalling an error) if the format guessing does not succeed.
origin	a Date object, or something which can be coerced by as.Date(origin, ...) to such an object or missing. In that case, "1970-01-01" is used.
tz	a time zone name.
...	further arguments to be passed from or to other methods.

Details

The usual vector re-cycling rules are applied to x and format so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months.

The as.Date methods accept character strings, factors, logical NA and objects of classes "POSIXlt" and "POSIXct". (The last is converted to days by ignoring the time after midnight in the representation of the time in specified time zone, default UTC.) Also objects of class "date" (from package **date**) and "dates" (from package **chron**). Character strings are processed as far as necessary for the format specified: any trailing characters are ignored.

as.Date will accept numeric data (the number of days since an epoch), since R 4.3.0 also when origin is not supplied.

The format and as.character methods ignore any fractional part of the date.

Value

The format and as.character methods return a character vector representing the date. NA dates are returned as NA_character_.

The as.Date methods return an object of class "Date".

Conversion from other Systems

Most systems record dates internally as the number of days since some origin, but this is fraught with problems, including

- Is the origin day 0 or day 1? As the 'Examples' show, Excel manages to use both choices for its two date systems.
- If the origin is far enough back, the designers may show their ignorance of calendar systems. For example, Excel's designer thought 1900 was a leap year (claiming to copy the error from earlier DOS spreadsheets), and Matlab's designer chose the non-existent date of 'January

0, 0000' (there is no such day), not specifying the calendar. (There is such a year in the 'Gregorian' calendar as used in ISO 8601:2004, but that does say that it is only to be used for years before 1582 with the agreement of the parties in information exchange.)

The only safe procedure is to check the other systems values for known dates: reports on the Internet (including R-help) are more often wrong than right.

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as `"2001-02-03"`.

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as `NA`. Unfortunately some common implementations (such as '`glibc`') are unreliable and guess at the intended meaning.

Years before 1CE (aka 1AD) will probably not be handled correctly.

References

International Organization for Standardization (2004, 1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* For links to versions available on-line see (at the time of writing) `https://www.qsl.net/g1smd/isopdf.htm`.

See Also

Date for details of the date class; `locales` to query or set a locale.

Your system's help pages on `strftime` and `strptime` to see how to specify their formats. Windows users will find no help page for `strptime`: code based on '`glibc`' is used (with corrections), so all the format specifiers described here are supported, but with no alternative number representation nor era available in any locale.

Examples

```
## locale-specific version of the date
format(Sys.Date(), "%a %b %d")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- as.Date(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
as.Date(dates, "%m/%d/%y")
```

```
## date given as number of days since 1900-01-01 (a date in 1989)
as.Date(32768, origin = "1900-01-01")
## Excel is said to use 1900-01-01 as day 1 (Windows default) or
## 1904-01-01 as day 0 (Mac default), but this is complicated by Excel
## incorrectly treating 1900 as a leap year.
## So for dates (post-1901) from Windows Excel
as.Date(35981, origin = "1899-12-30") # 1998-07-05
## and Mac Excel
as.Date(34519, origin = "1904-01-01") # 1998-07-05
## (these values come from http://support.microsoft.com/kb/214330)

## Experiment shows that Matlab's origin is 719529 days before ours,
## (it takes the non-existent 0000-01-01 as day 1)
## so Matlab day 734373 can be imported as
as.Date(734373) - 719529 # 2010-08-23
## (value from
## http://www.mathworks.de/de/help/matlab/matlab_prog/represent-date-and-times-in-MATLAB.html)

## Time zone effect
z <- ISOdate(2010, 04, 13, c(0,12)) # midnight and midday UTC
as.Date(z) # in UTC
## these time zone names are common
as.Date(z, tz = "NZ")
as.Date(z, tz = "HST") # Hawaii
```

`as.environment` *Coerce to an Environment Object*

Description

A generic function coercing an R object to an `environment`. A number or a character string is converted to the corresponding environment on the search path.

Usage

```
as.environment(x)
```

Arguments

x	an R object to convert. If it is already an environment, just return it. If it is a positive number, return the environment corresponding to that position on the search list. If it is -1, the environment it is called from. If it is a character string, match the string to the names on the search list. If it is a list, the equivalent of `list2env(x, parent = emptyenv())` is returned. If `is.object(x)` is true and it has a `class` for which an `as.environment` method is found, that is used.

Details

This is a primitive generic function: you can write methods to handle specific classes of objects, see InternalMethods.

Value

The corresponding environment object.

Author(s)

John Chambers

See Also

`environment` for creation and manipulation, `search`; `list2env`.

Examples

```
as.environment(1) ## the global environment
identical(globalenv(), as.environment(1)) ## is TRUE
try( ## <<- stats need not be attached
    as.environment("package:stats"))
ee <- as.environment(list(a = "A", b = pi, ch = letters[1:8]))
ls(ee) # names of objects in ee
utils::ls.str(ee)
```

---

as.function *Convert Object to Function*

Description

`as.function` is a generic function which is used to convert objects to functions.

`as.function.default` works on a list `x`, which should contain the concatenation of a formal argument list and an expression or an object of mode `"call"` which will become the function body. The function will be defined in a specified environment, by default that of the caller.

Usage

```
as.function(x, ...)

## Default S3 method:
as.function(x, envir = parent.frame(), ...)
```

Arguments

`x`	object to convert, a list for the default method.
`...`	additional arguments to be passed to or from methods.
`envir`	environment in which the function should be defined.

Value

The desired function.

Author(s)

Peter Dalgaard

See Also

`function`; `alist` which is handy for the construction of argument lists, etc.

Examples

```
as.function(alist(a = , b = 2, a+b))
as.function(alist(a = , b = 2, a+b))(3)
```

`as.POSIX*` *Date-time Conversion Functions*

Description

Functions to manipulate objects of classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times.

Usage

```
as.POSIXct(x, tz = "", ...)
as.POSIXlt(x, tz = "", ...)

## S3 method for class 'character'
as.POSIXlt(x, tz = "", format,
           tryFormats = c("%Y-%m-%d %H:%M:%OS",
                          "%Y/%m/%d %H:%M:%OS",
                          "%Y-%m-%d %H:%M",
                          "%Y/%m/%d %H:%M",
                          "%Y-%m-%d",
                          "%Y/%m/%d"),
           optional = FALSE, ...)
## Default S3 method:
as.POSIXlt(x, tz = "",
           optional = FALSE, ...)
## S3 method for class 'numeric'
as.POSIXlt(x, tz = "", origin, ...)

## S3 method for class 'Date'
as.POSIXct(x, tz = "UTC", ...)
## S3 method for class 'Date'
```

```
as.POSIXlt(x, tz = "UTC", ...)
## S3 method for class 'numeric'
as.POSIXct(x, tz = "", origin, ...)

## S3 method for class 'POSIXlt'
as.double(x, ...)
```

Arguments

x	R object to be converted.
tz	a character string. The time zone specification to be used for the conversion, *if one is required*. System-specific (see time zones), but "" is the current time zone, and "GMT" is UTC (Universal Time, Coordinated). Invalid values are most commonly treated as UTC, on some platforms with a warning.
...	further arguments to be passed to or from other methods.
format	character string giving a date-time format as used by strptime.
tryFormats	character vector of format strings to try if format is not specified.
optional	logical indicating to return NA (instead of signalling an error) if the format guessing does not succeed.
origin	a date-time object, or something which can be coerced by as.POSIXct(tz = "GMT") to such an object. Optional since R 4.3.0, where the equivalent of "1970-01-01" is used.

Details

The as.POSIX* functions convert an object to one of the two classes used to represent date/times (calendar dates plus time to the nearest second). They can convert objects of the other class and of class "Date" to these classes. Dates without times are treated as being at midnight UTC.

They can also convert character strings of the formats "2001-02-03" and "2001/02/03" optionally followed by white space and a time in the format "14:52" or "14:52:03". (Formats such as "01/02/03" are ambiguous but can be converted via a format specification by strptime.) Fractional seconds are allowed. Alternatively, format can be specified for character vectors or factors: if it is not specified and no standard format works for all non-NA inputs an error is thrown.

If format is specified, remember that some of the format specifications are locale-specific, and you may need to set the LC_TIME category appropriately *via* Sys.setlocale. This most often affects the use of %a, %A (weekday names), %b, %B (month names) and %p (AM/PM).

Logical NAs can be converted to either of the classes, but no other logical vectors can be.

If you are given a numeric time as the number of seconds since an epoch, see the examples.

Character input is first converted to class "POSIXlt" by strptime: numeric input is first converted to "POSIXct". Any conversion that needs to go between the two date-time classes requires a time zone: conversion from "POSIXlt" to "POSIXct" will validate times in the selected time zone. One issue is what happens at transitions to and from DST, for example in the UK

```
as.POSIXct(strptime("2011-03-27 01:30:00", "%Y-%m-%d %H:%M:%S"))
as.POSIXct(strptime("2010-10-31 01:30:00", "%Y-%m-%d %H:%M:%S"))
```

are respectively invalid (the clocks went forward at 1:00 GMT to 2:00 BST) and ambiguous (the clocks went back at 2:00 BST to 1:00 GMT). What happens in such cases is OS-specific: one should expect the first to be NA, but the second could be interpreted as either BST or GMT (and common OSes give both possible values). Note too (see strftime) that OS facilities may not format invalid times correctly.

Value

as.POSIXct and as.POSIXlt return an object of the appropriate class. If tz was specified, as.POSIXlt will give an appropriate "tzone" attribute. Date-times known to be invalid will be returned as NA.

Note

Some of the concepts used have to be extended backwards in time (the usage is said to be 'proleptic'). For example, the origin of time for the "POSIXct" class, '1970-01-01 00:00.00 UTC', is before UTC was defined. More importantly, conversion is done assuming the Gregorian calendar which was introduced in 1582 and not used near-universally until the 20th century. One of the reinterpretations assumed by ISO 8601:2004 is that there was a year zero, even though current year numbering (and zero) is a much later concept (525 CE for year numbers from 1 CE).

Conversions between "POSIXlt" and "POSIXct" of future times are speculative except in UTC. The main uncertainty is in the use of and transitions to/from DST (most systems will assume the continuation of current rules but these can be changed at short notice).

If you want to extract specific aspects of a time (such as the day of the week) just convert it to class "POSIXlt" and extract the relevant component(s) of the list, or if you want a character representation (such as a named day of the week) use the format method.

If a time zone is needed and that specified is invalid on your system, what happens is system-specific but attempts to set it will probably be ignored.

Conversion from character needs to find a suitable format unless one is supplied (by trying common formats in turn): this can be slow for long inputs.

See Also

DateTimeClasses for details of the classes; strptime for conversion to and from character representations.

Sys.timezone for details of the (system-specific) naming of time zones.

locales for locale-specific aspects.

Examples

```
(z <- Sys.time())             # the current datetime, as class "POSIXct"
unclass(z)                    # a large integer
floor(unclass(z)/86400)       # the number of days since 1970-01-01 (UTC)
(now <- as.POSIXlt(Sys.time())) # the current datetime, as class "POSIXlt"
str(unclass(now))             # the internal list ; use now$hour, etc :
now$year + 1900               # see ?DateTimeClasses
months(now); weekdays(now)    # see ?months; using LC_TIME locale
```

```
## suppose we have a time in seconds since 1960-01-01 00:00:00 GMT
## (the origin used by SAS)
z <- 1472562988
# ways to convert this
as.POSIXct(z, origin = "1960-01-01")               # local
as.POSIXct(z, origin = "1960-01-01", tz = "GMT")    # in UTC

## SPSS dates (R-help 2006-02-16)
z <- c(10485849600, 10477641600, 10561104000, 10562745600)
as.Date(as.POSIXct(z, origin = "1582-10-14", tz = "GMT"))

## Stata date-times: milliseconds since 1960-01-01 00:00:00 GMT
## format %tc excludes leap-seconds, assumed here
## For format %tC including leap seconds, see foreign::read.dta()
z <- 1579598122120
op <- options(digits.secs = 3)
# avoid rounding down: milliseconds are not exactly representable
as.POSIXct((z+0.1)/1000, origin = "1960-01-01")
options(op)

## Matlab 'serial day number' (days and fractional days)
z <- 7.343736909722223e5 # 2010-08-23 16:35:00
as.POSIXct((z - 719529)*86400, origin = "1970-01-01", tz = "UTC")

as.POSIXlt(Sys.time(), "GMT") # the current time in UTC

## These may not be correct names on your system
as.POSIXlt(Sys.time(), "America/New_York")  # in New York
as.POSIXlt(Sys.time(), "EST5EDT")           # alternative.
as.POSIXlt(Sys.time(), "EST" )   # somewhere in Eastern Canada
as.POSIXlt(Sys.time(), "HST")    # in Hawaii
as.POSIXlt(Sys.time(), "Australia/Darwin")


tab <- file.path(R.home("share"), "zoneinfo", "zone1970.tab")
if(file.exists(tab)) { # typically on Windows; *not* on Linux
  cols <- c("code", "coordinates", "TZ", "comments")
  tmp <- read.delim(tab,
                    header = FALSE, comment.char = "#", col.names = cols)
  if(interactive()) View(tmp)
  head(tmp, 10)
}
```

`AsIs` *Inhibit Interpretation/Conversion of Objects*

Description

Change the class of an object to indicate that it should be treated 'as is'.

Usage

```
I(x)
```

Arguments

x	an object

Details

Function I has two main uses.

- In function data.frame. Protecting an object by enclosing it in I() in a call to data.frame inhibits the conversion of character vectors to factors and the dropping of names, and ensures that matrices are inserted as single columns. I can also be used to protect objects which are to be added to a data frame, or converted to a data frame *via* as.data.frame.

  It achieves this by prepending the class "AsIs" to the object's classes. Class "AsIs" has a few of its own methods, including for [, as.data.frame, print and format.
- In function formula. There it is used to inhibit the interpretation of operators such as "+", "-", "*" and "^" as formula operators, so they are used as arithmetical operators. This is interpreted as a symbol by terms.formula.

Value

A copy of the object with class "AsIs" prepended to the class(es).

References

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, formula

---

asplit *Split Array/Matrix By Its Margins*

Description

Split an array or matrix by its margins.

Usage

```
asplit(x, MARGIN)
```

Arguments

x	an array, including a matrix.
MARGIN	a vector giving the margins to split by. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where x has named dimnames, it can be a character vector selecting dimension names.

Details

Since R 4.1.0, one can also obtain the splits (less efficiently) using apply(x, MARGIN, identity, simplify = FALSE). The values of the splits can also be obtained (less efficiently) by split(x, slice.index(x, MARGIN)).

Value

A "list array" with dimension dv and each element an array of dimension de and dimnames preserved as available, where dv and de are, respectively, the dimensions of x included and not included in MARGIN.

Examples

```
## A 3-dimensional array of dimension 2 x 3 x 4:
d <- 2 : 4
x <- array(seq_len(prod(d)), d)
x
## Splitting by margin 2 gives a 1-d list array of length 3
## consisting of 2 x 4 arrays:
asplit(x, 2)
## Splitting by margins 1 and 2 gives a 2 x 3 list array
## consisting of 1-d arrays of length 4:
asplit(x, c(1, 2))
## Compare to
split(x, slice.index(x, c(1, 2)))

## A 2 x 3 matrix:
(x <- matrix(1 : 6, 2, 3))
## To split x by its rows, one can use
asplit(x, 1)
## or less efficiently
split(x, slice.index(x, 1))
split(x, row(x))
```

---

assign *Assign a Value to a Name*

Description

Assign a value to a name in an environment.

Usage

```
assign(x, value, pos = -1, envir = as.environment(pos),
       inherits = FALSE, immediate = TRUE)
```

Arguments

x	a variable name, given as a character string. No coercion is done, and the first element of a character vector of length greater than one will be used, with a warning.
value	a value to be assigned to x.
pos	where to do the assignment. By default, assigns into the current environment. See 'Details' for other possibilities.
envir	the environment to use. See 'Details'.
inherits	should the enclosing frames of the environment be inspected?
immediate	an ignored compatibility feature.

Details

There are no restrictions on the name given as x: it can be a non-syntactic name (see make.names).

The pos argument can specify the environment in which to assign the object in any of several ways: as -1 (the default), as a positive integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The envir argument is an alternative way to specify an environment, but is primarily for back compatibility.

assign does not dispatch assignment methods, so it cannot be used to set elements of vectors, names, attributes, etc.

Note that assignment to an attached list or data frame changes the attached copy and not the original object: see attach and with.

Value

This function is invoked for its side effect, which is assigning value to the variable x. If no envir is specified, then the assignment takes place in the currently active environment.

If inherits is TRUE, enclosing environments of the supplied environment are searched until the variable x is encountered. The value is then assigned in the environment in which the variable is encountered (provided that the binding is not locked: see lockBinding: if it is, an error is signaled). If the symbol is not encountered then assignment takes place in the user's workspace (the global environment).

If inherits is FALSE, assignment takes place in the initial frame of envir, unless an existing binding is locked or there is no existing binding and the environment is locked (when an error is signaled).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

<-, get, the inverse of assign(), exists, environment.

Examples

```
for(i in 1:6) { #-- Create objects  'r.1', 'r.2', ... 'r.6' --
    nam <- paste("r", i, sep = ".")
    assign(nam, 1:i)
}
ls(pattern = "^r..$")

##-- Global assignment within a function:
myf <- function(x) {
    innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
    innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2          # FALSE
get("a[1]") == 2   # TRUE
```

---

assignOps *Assignment Operators*

Description

Assign a value to a name.

Usage

```
x <- value
x <<- value
value -> x
value ->> x

x = value
```

Arguments

x	a variable name (possibly quoted).
value	a value to be assigned to x.

Details

There are three different assignment operators: two of them have leftwards and rightwards forms.

The operators <- and = assign into the environment in which they are evaluated. The operator <- can be used anywhere, whereas the operator = is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions.

The operators <<- and ->> are normally only used in functions, and cause a search to be made through parent environments for an existing definition of the variable being assigned. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment. Note that their semantics differ from that in the S language, but are useful in conjunction with the scoping rules of R. See 'The R Language Definition' manual for further details and examples.

In all the assignment operator expressions, x can be a name or an expression defining a part of an object to be replaced (e.g., z[[1]]). A syntactic name does not need to be quoted, though it can be (preferably by backticks).

The leftwards forms of assignment <- = <<- group right to left, the other from left to right.

Value

value. Thus one can use a <- b <- c <- 6.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

assign (and its inverse get), for "subassignment" such as x[i] <- v, see [<-; further, environment.

---

attach *Attach Set of R Objects to Search Path*

Description

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

Usage

```
attach(what, pos = 2L, name = deparse1(substitute(what), backtick=FALSE),
       warn.conflicts = TRUE)
```

Arguments

what	'database'. This can be a data.frame or a list or a R data file created with save or NULL or an environment. See also 'Details'.
pos	integer specifying position in search() where to attach.
name	name to use for the attached database. Names starting with package: are reserved for library.
warn.conflicts	logical. If TRUE, message()s are printed about conflicts from attaching the database, unless that database contains an object .conflicts.OK. A conflict is a function masking a function, or a non-function masking a non-function. NB: Even though the name is warn.conflicts for historical reasons, the messages about conflicts are *not* warning()s but message()s.

Details

When evaluating a variable or function name R searches for that name in the databases listed by search. The first name of the appropriate type is used.

By attaching a data frame (or list) to the search path it is possible to refer to the variables in the data frame by their names alone, rather than as components of the data frame (e.g., in the example below, height rather than women$height).

By default the database is attached in position 2 in the search path, immediately after the user's workspace and before all previously attached packages and previously attached databases. This can be altered to attach later in the search path with the pos option, but you cannot attach at pos = 1.

The database is not actually attached. Rather, a new environment is created on the search path and the elements of a list (including columns of a data frame) or objects in a save file or an environment are *copied* into the new environment. If you use <<- or assign to assign to an attached database, you only alter the attached copy, not the original object. (Normal assignment will place a modified version in the user's workspace: see the examples.) For this reason attach can lead to confusion.

One useful 'trick' is to use what = NULL (or equivalently a length-zero list) to create a new environment on the search path into which objects can be assigned by assign or load or sys.source.

Names starting "package:" are reserved for library and should not be used by end users. Attached files are by default given the name file:*what*. The name argument given for the attached environment will be used by search and can be used as the argument to as.environment.

Value

The environment is returned invisibly with a "name" attribute.

Good practice

attach has the side effect of altering the search path and this can easily lead to the wrong object of a particular name being found. People do often forget to detach databases.

In interactive use, with is usually preferable to the use of attach/detach, unless what is a save()-produced file in which case attach() is a (safety) wrapper for load().

In programming, functions should not change the search path unless that is their purpose. Often with can be used within a function. If not, good practice is to

- Always use a distinctive name argument, and
- To immediately follow the attach call by an on.exit call to detach using the distinctive name.

This ensures that the search path is left unchanged even if the function is interrupted or if code after the attach call changes the search path.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

library, detach, search, objects, environment, with.

Examples

```
require(utils)

summary(women$height)   # refers to variable 'height' in the data frame
attach(women)
summary(height)         # The same variable now available by name
height <- height*2.54   # Don't do this. It creates a new variable
                        # in the user's workspace
find("height")
summary(height)         # The new variable in the workspace
rm(height)
summary(height)         # The original variable.
height <<- height*25.4  # Change the copy in the attached environment
find("height")
summary(height)         # The changed copy
detach("women")
summary(women$height)   # unchanged

## Not run: ## create an environment on the search path and populate it
sys.source("myfuns.R", envir = attach(NULL, name = "myfuns"))

## End(Not run)
```

---

attr *Object Attributes*

Description

Get or set specific attributes of an object.

Usage

```
attr(x, which, exact = FALSE)
attr(x, which) <- value
```

Arguments

x	an object whose attributes are to be accessed.
which	a non-empty character string specifying which attribute is to be accessed.
exact	logical: should which be matched exactly?
value	an object, the new value of the attribute, or NULL to remove the attribute.

Details

These functions provide access to a single attribute of an object. The replacement form causes the named attribute to take the value specified (or create a new attribute with the value given).

The extraction function first looks for an exact match to which amongst the attributes of x, then (unless exact = TRUE) a unique partial match. (Setting options(warnPartialMatchAttr = TRUE) causes partial matches to give warnings.)

The replacement function only uses exact matches.

Note that some attributes (namely class, comment, dim, dimnames, names, row.names and tsp) are treated specially and have restrictions on the values which can be set. (Note that this is not true of levels which should be set for factors via the levels replacement function.)

The extractor function allows (and does not match) empty and missing values of which: the replacement function does not.

NULL objects cannot have attributes and attempting to assign one by attr gives an error.

Both are primitive functions.

Value

For the extractor, the value of the attribute matched, or NULL if no exact match is found and no or more than one partial match is found.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attributes

Examples

```
# create a 2 by 5 matrix
x <- 1:10
attr(x,"dim") <- c(2, 5)
```

attributes *Object Attribute Lists*

Description

These functions access an object's attributes. The first form below returns the object's attribute list. The replacement forms uses the list on the right-hand side of the assignment as the object's attributes (if appropriate).

Usage

```
attributes(x)
attributes(x) <- value
mostattributes(x) <- value
```

Arguments

x	any R object.
value	an appropriate named list of attributes, or NULL.

Details

Unlike attr it is not an error to set attributes on a NULL object: it will first be coerced to an empty list.

Note that some attributes (namely class, comment, dim, dimnames, names, row.names and tsp) are treated specially and have restrictions on the values which can be set. (Note that this is not true of levels which should be set for factors via the levels replacement function.)

Attributes are not stored internally as a list and should be thought of as a set and not a vector, i.e, the *order* of the elements of attributes() does not matter. This is also reflected by identical()'s behaviour with the default argument attrib.as.set = TRUE. Attributes must have unique names (and NA is taken as "NA", not a missing value).

Assigning attributes first removes all attributes, then sets any dim attribute and then the remaining attributes in the order given: this ensures that setting a dim attribute always precedes the dimnames attribute.

The mostattributes assignment takes special care for the dim, names and dimnames attributes, and assigns them only when known to be valid whereas an attributes assignment would give an error if any are not. It is principally intended for arrays, and should be used with care on classed objects. For example, it does not check that row.names are assigned correctly for data frames.

The names of a pairlist are not stored as attributes, but are reported as if they were (and can be set by the replacement form of attributes).

NULL objects cannot have attributes and attempts to assign them will promote the object to an empty list.

Both assignment and replacement forms of attributes are primitive functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attr, structure.

Examples

```
x <- cbind(a = 1:3, pi = pi) # simple matrix with dimnames
attributes(x)

## strip an object's attributes:
attributes(x) <- NULL
x # now just a vector of length 6

mostattributes(x) <- list(mycomment = "really special", dim = 3:2,
   dimnames = list(LETTERS[1:3], letters[1:5]), names = paste(1:6))
x # dim(), but not {dim}names
```

---

autoload *On-demand Loading of Packages*

Description

autoload creates a promise-to-evaluate autoloader and stores it with name name in .AutoloadEnv environment. When R attempts to evaluate name, autoloader is run, the package is loaded and name is re-evaluated in the new package's environment. The result is that R behaves as if package was loaded but it does not occupy memory.

.Autoloaded contains the names of the packages for which autoloading has been promised.

Usage

```
autoload(name, package, reset = FALSE, ...)
autoloader(name, package, ...)

.AutoloadEnv
.Autoloaded
```

Arguments

name	string giving the name of an object.
package	string giving the name of a package containing the object.
reset	logical: for internal use by autoloader.
...	other arguments to library.

Value

This function is invoked for its side-effect. It has no return value.

See Also

delayedAssign, library

Examples

```
require(stats)
autoload("interpSpline", "splines")
search()
ls("Autoloads")
.Autoloaded

x <- sort(stats::rnorm(12))
y <- x^2
is <- interpSpline(x, y)
search() ## now has splines
detach("package:splines")
search()
is2 <- interpSpline(x, y+x)
search() ## and again
detach("package:splines")
```

---

backsolve *Solve an Upper or Lower Triangular System*

Description

Solves a triangular system of linear equations.

Usage

```
   backsolve(r, x, k = ncol(r), upper.tri = TRUE,
             transpose = FALSE)
forwardsolve(l, x, k = ncol(l), upper.tri = FALSE,
             transpose = FALSE)
```

Arguments

r, l	an upper (or lower) triangular matrix giving the coefficients for the system to be solved. Values below (above) the diagonal are ignored.
x	a matrix whose columns give the right-hand sides for the equations.
k	the number of columns of r and rows of x to use.
upper.tri	logical; if TRUE (default), the *upper tri*angular part of r is used. Otherwise, the lower one.
transpose	logical; if TRUE, solve $r' * y = x$ for y, i.e., t(r) %*% y == x.

Details

Solves a system of linear equations where the coefficient matrix is upper (or 'right', 'R') or lower ('left', 'L') triangular.

`x <- backsolve(R, b)` solves $Rx = b$, and
`x <- forwardsolve(L, b)` solves $Lx = b$, respectively.

The `r`/`l` must have at least `k` rows and columns, and `x` must have at least `k` rows.

This is a wrapper for the level-3 BLAS routine `dtrsm`.

Value

The solution of the triangular system. The result will be a vector if `x` is a vector and a matrix if `x` is a matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

See Also

`chol`, `qr`, `solve`.

Examples

```
## upper triangular matrix 'r':
r <- rbind(c(1,2,3),
           c(0,1,1),
           c(0,0,2))
( y <- backsolve(r, x <- c(8,4,2)) ) # -1 3 1
r %*% y # == x = (8,4,2)
backsolve(r, x, transpose = TRUE) # 8 -12 -5
```

`balancePOSIXlt` *Balancing "Ragged" and Out-of-range POSIXlt Date-Times*

Description

Utilities to 'balance' objects of class `"POSIXlt"`.

`unCfillPOSIXlt(x)` is a fast primitive version of `balancePOSIXlt(x, fill.only=TRUE, classed=FALSE)` or equivalently, `unclass(balancePOSIXlt(x, fill.only=TRUE))` from where it is named.

Usage

```
balancePOSIXlt(x, fill.only = FALSE, classed = TRUE)
unCfillPOSIXlt(x)
```

Arguments

x	an R object inheriting from "POSIXlt", see POSIXlt.
fill.only	a logical specifying if balancePOSIXlt(x, ..) should only "fill up" by recycling, but not re-check validity nor recompute, e.g., x$wday and x$yday.
classed	a logical specifying if the result should be classed, true by default. Using balancePOSIXlt(x, classed = FALSE) is equivalent to but faster than unclass(balancePOSIXlt(x)).

"Ragged" and Out-of-range *vs* "Balanced" POSIXlt

Note that "POSIXlt" objects x may have their (9 to 11) list components of different lengths, by simply recycling them to full length. Prior to R 4.3.0, this has worked in printing, formatting, and conversion to "POSIXct", but often not for length(), conversion to "Date" or indexing, i.e., subsetting, [, or subassigning, [<-.

Relatedly, components sec, min, hour, mday and mon could have been out of their designated range (say, 0–23 for hours) and still work correctly, e.g. in conversions and printing. This is supported as well, since R 4.3.0, at least when the values are not extreme.

Function balancePOSIXlt(x) will now return a version of the "POSIXlt" object x which by default is balanced in both ways: All the internal list components are of full length, and their values are inside their ranges as specified in as.POSIXlt's 'Details on POSIXlt'. Setting fill.only = TRUE will only recycle the list components to full length, but not check them at all. This is particularly faster when all components of x are already of full length.

Experimentally, balancePOSIXlt() and other functions returning POSIXlt objects now set a logical attribute "balanced" with NA meaning "filled-in", i.e., not "ragged" and TRUE means (fully) balanced.

See Also

For more details about many aspects of valid POSIXlt objects, notably their internal list components, see 'DateTimeClasses', e.g., as.POSIXlt, notably the section 'Details on POSIXlt'.

Examples

```
## FIXME: this should also work for regular (non-UTC) time zones.
TZ <-"UTC"
# Could be
# d1 <- as.POSIXlt("2000-01-02 3:45", tz = TZ)
# on systems (almost all) which have tm_zone.
oldTZ <- Sys.getenv('TZ', unset = "unset")
Sys.setenv(TZ = "UTC")
d1 <- as.POSIXlt("2000-01-02 3:45")
d1$min <- d1$min + (0:16)*20L
(f1 <- format(d1))
```

```
str(unclass(d1))      # only $min is of length > 1
df <- balancePOSIXlt(d1, fill.only = TRUE) # a "POSIXlt" object
str(unclass(df))      # all of length 17; 'min' unchanged
db <- balancePOSIXlt(d1, classed = FALSE)  # a list
stopifnot(identical(
    unCfillPOSIXlt(d1),
    balancePOSIXlt(d1, fill.only = TRUE, classed = FALSE)))
str(db) # of length 17 *and* in range

if(oldTZ == "unset") Sys.unsetenv('TZ') else Sys.setenv(TZ = oldTZ)
```

---

basename *Manipulate File Paths*

Description

basename removes all of the path up to and including the last path separator (if any).

dirname returns the part of the path up to but excluding the last path separator, or "." if there is no path separator.

Usage

```
basename(path)
dirname(path)
```

Arguments

path	character vector, containing path names.

Details

tilde expansion of the path will be performed.

Trailing path separators are removed before dissecting the path, and for dirname any trailing file separators are removed from the result.

Value

A character vector of the same length as path. A zero-length input will give a zero-length output with no error.

Paths not containing any separators are taken to be in the current directory, so dirname returns ".".

If an element of path is NA, so is the result.

"" is not a valid pathname, but is returned unchanged.

Behaviour on Windows

On Windows this will accept either `\` or `/` as the path separator, but `dirname` will return a path using `/` (except if on a network share, when the leading `\\` will be preserved). Expect these only to be able to handle complete paths, and not for example just a network share or a drive.

UTF-8-encoded path names not valid in the current locale can be used.

Note

These are not wrappers for the POSIX system functions of the same names: in particular they do **not** have the special handling of the path `"/"` and of returning `"."` for empty strings.

See Also

`file.path`, `path.expand`.

Examples

```
basename(file.path("","p1","p2","p3", c("file1", "file2")))
dirname (file.path("","p1","p2","p3", "filename"))
```

`Bessel` *Bessel Functions*

Description

Bessel Functions of integer and fractional order, of first and second kind, J_ν and Y_ν, and Modified Bessel functions (of first and third kind), I_ν and K_ν.

Usage

```
besselI(x, nu, expon.scaled = FALSE)
besselK(x, nu, expon.scaled = FALSE)
besselJ(x, nu)
besselY(x, nu)
```

Arguments

`x`	numeric, ≥ 0.
`nu`	numeric; the *order* (maybe fractional and negative) of the corresponding Bessel function.
`expon.scaled`	logical; if `TRUE`, the results are exponentially scaled in order to avoid overflow (I_ν) or underflow (K_ν), respectively.

Details

If `expon.scaled = TRUE`, $e^{-x} I_\nu(x)$, or $e^x K_\nu(x)$ are returned.

For $\nu < 0$, formulae 9.1.2 and 9.6.2 from Abramowitz & Stegun are applied (which is probably suboptimal), except for `besselK` which is symmetric in `nu`.

The current algorithms will give warnings about accuracy loss for large arguments. In some cases, these warnings are exaggerated, and the precision is perfect. For large `nu`, say in the order of millions, the current algorithms are rarely useful.

Value

Numeric vector with the (scaled, if `expon.scaled = TRUE`) values of the corresponding Bessel function.

The length of the result is the maximum of the lengths of the parameters. All parameters are recycled to that length.

Author(s)

Original Fortran code: W. J. Cody, Argonne National Laboratory
Translation to C and adaptation to R: Martin Maechler <maechler@stat.math.ethz.ch>.

Source

The C code is a translation of Fortran routines from https://netlib.org/specfun/ribesl, '../rjbesl', etc. The four source code files for bessel[IJKY] each contain a paragraph "Acknowledgement" and "References", a short summary of which is

besselI based on (code) by David J. Sookne, see Sookne (1973). . . Modifications. . . An earlier version was published in Cody (1983).

besselJ as `besselI`

besselK based on (code) by J. B. Campbell (1980). . . Modifications. . .

besselY draws heavily on Temme's Algol program for Y. . . and on Campbell's programs for $Y_\nu(x)$ . . . . . heavily modified.

References

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions*. Dover, New York; Chapter 9: Bessel Functions of Integer Order.

In order of "Source" citation above:

Sookne, David J. (1973). Bessel Functions of Real Argument and Integer Order. *Journal of Research of the National Bureau of Standards*, **77B**, 125–132. doi:10.6028/jres.077B.012.

Cody, William J. (1983). Algorithm 597: Sequence of modified Bessel functions of the first kind. *ACM Transactions on Mathematical Software*, **9**(2), 242–245. doi:10.1145/357456.357462.

Campbell, J.B. (1980). On Temme's algorithm for the modified Bessel function of the third kind. *ACM Transactions on Mathematical Software*, **6**(4), 581–586. doi:10.1145/355921.355928.

Campbell, J.B. (1979). Bessel functions J_nu(x) and Y_nu(x) of float order and float argument. *Computer Physics Communications*, **18**, 133–142. doi:10.1016/00104655(79)900304.

Temme, Nico M. (1976). On the numerical evaluation of the ordinary Bessel function of the second kind. *Journal of Computational Physics*, **21**, 343–350. doi:10.1016/00219991(76)900322.

See Also

Other special mathematical functions, such as `gamma`, $\Gamma(x)$, and `beta`, $B(x)$.

Examples

```
require(graphics)

nus <- c(0:5, 10, 20)

x <- seq(0, 4, length.out = 501)
plot(x, x, ylim = c(0, 6), ylab = "", type = "n",
     main = "Bessel Functions  I_nu(x)")
for(nu in nus) lines(x, besselI(x, nu = nu), col = nu + 2)
legend(0, 6, legend = paste("nu=", nus), col = nus + 2, lwd = 1)

x <- seq(0, 40, length.out = 801); yl <- c(-.5, 1)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  J_nu(x)")
abline(h=0, v=0, lty=3)
for(nu in nus) lines(x, besselJ(x, nu = nu), col = nu + 2)
legend("topright", legend = paste("nu=", nus), col = nus + 2, lwd = 1, bty="n")

## Negative nu's ---------------------------------------------------
xx <- 2:7
nu <- seq(-10, 9, length.out = 2001)
## --- I() --- --- --- ---
matplot(nu, t(outer(xx, nu, besselI)), type = "l", ylim = c(-50, 200),
        main = expression(paste("Bessel ", I[nu](x), " for fixed ", x,
                                ",  as ", f(nu))),
        xlab = expression(nu))
abline(v = 0, col = "light gray", lty = 3)
legend(5, 200, legend = paste("x=", xx), col=seq(xx), lty=1:5)

## --- J() --- --- --- ---
bJ <- t(outer(xx, nu, besselJ))
matplot(nu, bJ, type = "l", ylim = c(-500, 200),
        xlab = quote(nu), ylab = quote(J[nu](x)),
        main = expression(paste("Bessel ", J[nu](x), " for fixed ", x)))
abline(v = 0, col = "light gray", lty = 3)
legend("topright", legend = paste("x=", xx), col=seq(xx), lty=1:5)

## ZOOM into right part:
matplot(nu[nu > -2], bJ[nu > -2,], type = "l",
        xlab = quote(nu), ylab = quote(J[nu](x)),
        main = expression(paste("Bessel ", J[nu](x), " for fixed ", x)))
abline(h=0, v = 0, col = "gray60", lty = 3)
legend("topright", legend = paste("x=", xx), col=seq(xx), lty=1:5)
```

```
##---------------  x --> 0  -----------------------------
x0 <- 2^seq(-16, 5, length.out=256)
plot(range(x0), c(1e-40, 1), log = "xy", xlab = "x", ylab = "", type = "n",
     main = "Bessel Functions  J_nu(x)  near 0\n log - log  scale") ; axis(2, at=1)
for(nu in sort(c(nus, nus+0.5)))
    lines(x0, besselJ(x0, nu = nu), col = nu + 2, lty= 1+ (nu%%1 > 0))
legend("right", legend = paste("nu=", paste(nus, nus+0.5, sep=", ")),
       col = nus + 2, lwd = 1, bty="n")

x0 <- 2^seq(-10, 8, length.out=256)
plot(range(x0), 10^c(-100, 80), log = "xy", xlab = "x", ylab = "", type = "n",
     main = "Bessel Functions  K_nu(x)  near 0\n log - log  scale") ; axis(2, at=1)
for(nu in sort(c(nus, nus+0.5)))
    lines(x0, besselK(x0, nu = nu), col = nu + 2, lty= 1+ (nu%%1 > 0))
legend("topright", legend = paste("nu=", paste(nus, nus + 0.5, sep = ", ")),
       col = nus + 2, lwd = 1, bty="n")

x <- x[x > 0]
plot(x, x, ylim = c(1e-18, 1e11), log = "y", ylab = "", type = "n",
     main = "Bessel Functions  K_nu(x)"); axis(2, at=1)
for(nu in nus) lines(x, besselK(x, nu = nu), col = nu + 2)
legend(0, 1e-5, legend=paste("nu=", nus), col = nus + 2, lwd = 1)

yl <- c(-1.6, .6)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  Y_nu(x)")
for(nu in nus){
    xx <- x[x > .6*nu]
    lines(xx, besselY(xx, nu=nu), col = nu+2)
}
legend(25, -.5, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## negative nu in bessel_Y -- was bogus for a long time
curve(besselY(x, -0.1), 0, 10, ylim = c(-3,1), ylab = "")
for(nu in c(seq(-0.2, -2, by = -0.1)))
  curve(besselY(x, nu), add = TRUE)
title(expression(besselY(x, nu) * "   " *
                 {nu == list(-0.1, -0.2, ..., -2)}))
```

---

bindenv *Binding and Environment Locking, Active Bindings*

Description

These functions represent an interface for adjustments to environments and bindings within environments. They allow for locking environments as well as individual bindings, and for linking a variable to a function.

Usage

```
lockEnvironment(env, bindings = FALSE)
environmentIsLocked(env)
lockBinding(sym, env)
unlockBinding(sym, env)
bindingIsLocked(sym, env)

makeActiveBinding(sym, fun, env)
bindingIsActive(sym, env)
activeBindingFunction(sym, env)
```

Arguments

env	an environment.
bindings	logical specifying whether bindings should be locked.
sym	a name object or character string.
fun	a function taking zero or one arguments.

Details

The function `lockEnvironment` locks its environment argument. Locking the environment prevents adding or removing variable bindings from the environment. Changing the value of a variable is still possible unless the binding has been locked. The namespace environments of packages with namespaces are locked when loaded.

`lockBinding` locks individual bindings in the specified environment. The value of a locked binding cannot be changed. Locked bindings may be removed from an environment unless the environment is locked.

`makeActiveBinding` installs `fun` in environment `env` so that getting the value of `sym` calls `fun` with no arguments, and assigning to `sym` calls `fun` with one argument, the value to be assigned. This allows the implementation of things like C variables linked to R variables and variables linked to databases, and is used to implement `setRefClass`. It may also be useful for making thread-safe versions of some system globals. Currently active bindings are not preserved during package installation, but they can be created in `.onLoad`.

Value

The `bindingIsLocked` and `environmentIsLocked` return a length-one logical vector. The remaining functions return `NULL`, invisibly.

Author(s)

Luke Tierney

Examples

```
# locking environments
e <- new.env()
assign("x", 1, envir = e)
```

```
get("x", envir = e)
lockEnvironment(e)
get("x", envir = e)
assign("x", 2, envir = e)
try(assign("y", 2, envir = e)) # error

# locking bindings
e <- new.env()
assign("x", 1, envir = e)
get("x", envir = e)
lockBinding("x", e)
try(assign("x", 2, envir = e)) # error
unlockBinding("x", e)
assign("x", 2, envir = e)
get("x", envir = e)

# active bindings
f <- local( {
    x <- 1
    function(v) {
       if (missing(v))
           cat("get\n")
       else {
           cat("set\n")
           x <<- v
       }
       x
    }
})
makeActiveBinding("fred", f, .GlobalEnv)
bindingIsActive("fred", .GlobalEnv)
fred
fred <- 2
fred
```

`bitwise` *Bitwise Logical Operations*

Description

Logical operations on integer vectors with elements viewed as sets of bits.

Usage

```
bitwNot(a)
bitwAnd(a, b)
bitwOr(a, b)
bitwXor(a, b)

bitwShiftL(a, n)
bitwShiftR(a, n)
```

Arguments

a, b	integer vectors; numeric vectors are coerced to integer vectors.
n	non-negative integer vector of values up to 31.

Details

Each element of an integer vector has 32 bits.

Pairwise operations can result in integer NA.

Shifting is done assuming the values represent unsigned integers.

Value

An integer vector of length the longer of the arguments, or zero length if one is zero-length.

The output element is NA if an input is NA (after coercion) or an invalid shift.

See Also

The logical operators, !, &, |, xor. Notably these *do* work bitwise for raw arguments.

The classes "octmode" and "hexmode" whose implementation of the standard logical operators is based on these functions.

Package **bitops** has similar functions for numeric vectors which differ in the way they treat integers 2^{31} or larger.

Examples

```
bitwNot(0:12) # -1 -2  ... -13
bitwAnd(15L, 7L) #  7
bitwOr (15L, 7L) # 15
bitwXor(15L, 7L) #  8
bitwXor(-1L, 1L) # -2

## The "same" for 'raw' instead of integer :
rr12 <- as.raw(0:12) ; rbind(rr12, !rr12)
c(r15 <- as.raw(15), r7 <- as.raw(7)) #  0f 07
r15 & r7    # 07
r15 | r7    # 0f
xor(r15, r7)# 08

bitwShiftR(-1, 1:31) # shifts of 2^32-1 = 4294967295
```

body *Access to and Manipulation of the Body of a Function*

Description

Get or set the *body* of a function which is basically all of the function definition but its formal arguments (formals), see the 'Details'.

Usage

```
body(fun = sys.function(sys.parent()))
body(fun, envir = environment(fun)) <- value
```

Arguments

fun	a function object, or see 'Details'.
envir	environment in which the function should be defined.
value	an object, usually a language object: see section 'Value'.

Details

For the first form, fun can be a character string naming the function to be manipulated, which is searched for from the parent frame. If it is not specified, the function calling body is used.

The bodies of all but the simplest are braced expressions, that is calls to {: see the 'Examples' section for how to create such a call.

Value

body returns the body of the function specified. This is normally a language object, most often a call to {, but it can also be a symbol such as pi or a constant (e.g., 3 or "R") to be the return value of the function.

The replacement form sets the body of a function to the object on the right hand side, and (potentially) resets the environment of the function, and drops attributes. If value is of class "expression" the first element is used as the body: any additional elements are ignored, with a warning.

See Also

The three parts of a (non-primitive) function are its formals, body, and environment.

Further, see alist, args, function.

Examples

```
body(body)
f <- function(x) x^5
body(f) <- quote(5^x)
## or equivalently  body(f) <- expression(5^x)
f(3) # = 125
body(f)

## creating a multi-expression body
e <- expression(y <- x^2, return(y)) # or a list
body(f) <- as.call(c(as.name("{"), e))
f
f(8)
## Using substitute() may be simpler than 'as.call(c(as.name("{",..)))':
stopifnot(identical(body(f), substitute({ y <- x^2; return(y) })))
```

---

bquote *Partial substitution in expressions*

Description

An analogue of the LISP backquote macro. `bquote` quotes its argument except that terms wrapped in `.()` are evaluated in the specified `where` environment. If `splice = TRUE` then terms wrapped in `..()` are evaluated and spliced into a call.

Usage

```
bquote(expr, where = parent.frame(), splice = FALSE)
```

Arguments

`expr`	A language object.
`where`	An environment.
`splice`	Logical; if `TRUE` splicing is enabled.

Value

A language object.

See Also

`quote`, `substitute`

Examples

```
require(graphics)

a <- 2

bquote(a == a)
quote(a == a)

bquote(a == .(a))
substitute(a == A, list(A = a))

plot(1:10, a*(1:10), main = bquote(a == .(a)))

## to set a function default arg
default <- 1
bquote( function(x, y = .(default)) x+y )

exprs <- expression(x <- 1, y <- 2, x + y)
bquote(function() {..(exprs)}, splice = TRUE)
```

`browser` *Environment Browser*

Description

Interrupt the execution of an expression and allow the inspection of the environment where `browser` was called from.

Usage

```
browser(text = "", condition = NULL, expr = TRUE, skipCalls = 0L)
```

Arguments

`text`	a text string that can be retrieved once the browser is invoked.
`condition`	a condition that can be retrieved once the browser is invoked.
`expr`	a "condition". By default, and whenever not false after being coerced to `logical`, the debugger will be invoked, otherwise control is returned directly.
`skipCalls`	how many previous calls to skip when reporting the calling context.

Details

A call to `browser` can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter.

The purpose of the `text` and `condition` arguments are to allow helper programs (e.g., external debuggers) to insert specific values here, so that the specific call to browser (perhaps its location in

a source file) can be identified and special processing can be achieved. The values can be retrieved by calling `browserText` and `browserCondition`.

The purpose of the `expr` argument is to allow for the illusion of conditional debugging. It is an illusion, because execution is always paused at the call to browser, but control is only passed to the evaluator described below if `expr` is not `FALSE` after coercion to logical. In most cases it is going to be more efficient to use an `if` statement in the calling program, but in some cases using this argument will be simpler.

The `skipCalls` argument should be used when the `browser()` call is nested within another debugging function: it will look further up the call stack to report its location.

At the browser prompt the user can enter commands or R expressions, followed by a newline. The commands are

`c` exit the browser and continue execution at the next statement.

`cont` synonym for `c`.

`f` finish execution of the current loop or function.

`help` print this list of commands.

`n` evaluate the next statement, stepping over function calls. For byte compiled functions interrupted by `browser` calls, `n` is equivalent to `c`.

`s` evaluate the next statement, stepping into function calls. Again, byte compiled functions make `s` equivalent to `c`.

`where` print a stack trace of all active function calls.

`r` invoke a `"resume"` restart if one is available; interpreted as an R expression otherwise. Typically `"resume"` restarts are established for continuing from user interrupts.

`Q` exit the browser and the current evaluation and return to the top-level prompt.

Leading and trailing whitespace is ignored, except for an empty line. Handling of empty lines depends on the `"browserNLdisabled"` option; if it is `TRUE`, empty lines are ignored. If not, an empty line is the same as `n` (or `s`, if it was used most recently).

Anything else entered at the browser prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly, or use autoprint via `(n)`.

The number of lines printed for the deparsed call can be limited by setting `options(deparse.max.lines)`.

The browser prompt is of the form `Browse[`*n*`]>`: here *n* indicates the 'browser level'. The browser can be called when browsing (and often is when debug is in use), and each recursive call increases the number. (The actual number is the number of 'contexts' on the context stack: this is usually `2` for the outer level of browsing and `1` when examining dumps in `debugger`.)

This is a primitive function but does argument matching in the standard way.

Interaction with Condition Handling

Because the browser prompt is implemented using the restart and condition handling mechanism, it prevents error handlers set up before the breakpoint from being called or invoked. The implementation follows this model:

```
repeat withRestarts(
    withCallingHandlers(
        readEvalPrint(),
        error = function(cnd) {
            cat("Error:", conditionMessage(cnd), "\n")
            invokeRestart("browser")
        }
    ),
    browser = function(...) NULL
)

readEvalPrint <- function(env = parent.frame()) {
    print(eval(parse(prompt = "Browse[n]> "), env))
}
```

The restart invocation interrupts the lookup for condition handlers and transfers control to the next iteration of the debugger REPL.

Note that condition handlers for other classes (such as `"warning"`) are still called and may cause a non-local transfer of control out of the debugger.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

debug, and traceback for the stack on error. browserText for how to retrieve the text and condition.

---

browserText *Functions to Retrieve Values Supplied by Calls to the Browser*

Description

A call to browser can provide context by supplying either a text argument or a condition argument. These functions can be used to retrieve either of these arguments.

Usage

```
browserText(n = 1)
browserCondition(n = 1)
browserSetDebug(n = 1)
```

Arguments

n	The number of contexts to skip over, it must be non-negative.

Details

Each call to `browser` can supply either a text string or a condition. The functions `browserText` and `browserCondition` provide ways to retrieve those values. Since there can be multiple browser contexts active at any time we also support retrieving values from the different contexts. The innermost (most recently initiated) browser context is numbered 1: other contexts are numbered sequentially.

`browserSetDebug` provides a mechanism for initiating the browser in one of the calling functions. See `sys.frame` for a more complete discussion of the calling stack. To use `browserSetDebug` you select some calling function, determine how far back it is in the call stack and call `browserSetDebug` with `n` set to that value. Then, by typing `c` at the browser prompt you will cause evaluation to continue, and provided there are no intervening calls to browser or other interrupts, control will halt again once evaluation has returned to the closure specified. This is similar to the up functionality in GDB or the "step out" functionality in other debuggers.

Value

`browserText` returns the text, while `browserCondition` returns the condition from the specified browser context.

`browserSetDebug` returns NULL, invisibly.

Note

It may be of interest to allow for querying further up the set of browser contexts and this functionality may be added at a later date.

Author(s)

R. Gentleman

See Also

`browser`

---

builtins *Returns the Names of All Built-in Objects*

Description

Return the names of all the built-in objects. These are fetched directly from the symbol table of the R interpreter.

Usage

```
builtins(internal = FALSE)
```

Arguments

`internal`	a logical indicating whether only 'internal' functions (which can be called via `.Internal`) should be returned.

Details

`builtins()` returns an unsorted list of the objects in the symbol table, that is all the objects in the base environment. These are the built-in objects plus any that have been added subsequently when the base package was loaded. It is less confusing to use `ls(baseenv(), all.names = TRUE)`.

`builtins(TRUE)` returns an unsorted list of the names of internal functions, that is those which can be accessed as `.Internal(`*foo*`(args ...))` for *foo* in the list.

Value

A character vector.

---

by *Apply a Function to a Data Frame Split by Factors*

Description

Function by is an object-oriented wrapper for `tapply` applied to data frames.

Usage

```
by(data, INDICES, FUN, ..., simplify = TRUE)
```

Arguments

`data`	an R object, normally a data frame, possibly a matrix.
`INDICES`	a factor or a list of factors, each of length `nrow(data)`. For the data frame method, `INDICES` can also be a formula as in the `f` argument of the `split` method for data frames.
`FUN`	a function to be applied to (usually data-frame) subsets of `data`.
`...`	further arguments to `FUN`.
`simplify`	logical: see `tapply`.

Details

A data frame is split by row into data frames subsetted by the values of one or more factors, and function `FUN` is applied to each subset in turn.

For the default method, an object with dimensions (e.g., a matrix) is coerced to a data frame and the data frame method applied. Other objects are also coerced to a data frame, but `FUN` is applied separately to (subsets of) each column of the data frame.

Value

An object of class `"by"`, giving the results for each subset. This is always a list if `simplify` is false, otherwise a list or array (see `tapply`).

See Also

tapply, simplify2array. array2DF to convert result to a data frame. ave also applies a function block-wise.

Examples

```
require(stats)
by(warpbreaks[, 1:2], warpbreaks[,"tension"], summary)
by(warpbreaks[, 1],   warpbreaks[, -1],       summary)
by(warpbreaks, warpbreaks[,"tension"],
   function(x) lm(breaks ~ wool, data = x))

## now suppose we want to extract the coefficients by group
tmp1 <- with(warpbreaks,
            by(warpbreaks, tension,
               function(x) lm(breaks ~ wool, data = x)))
sapply(tmp1, coef)

## another way
tmp2 <- by(warpbreaks, ~ tension,
           with, coef(lm(breaks ~ wool)))
array2DF(tmp2, simplify = TRUE)
```

---

c *Combine Values into a Vector or List*

Description

This is a generic function which combines its arguments.

The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value, and all attributes except names are removed.

Usage

```
## S3 Generic function
c(...)

## Default S3 method:
c(..., recursive = FALSE, use.names = TRUE)
```

Arguments

...	objects to be concatenated. All NULL entries are dropped before method dispatch unless at the very beginning of the argument list.
recursive	logical. If recursive = TRUE, the function recursively descends through lists (and pairlists) combining all their elements into a vector.
use.names	logical indicating if names should be preserved.

Details

The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression. Pairlists are treated as lists, whereas non-vector components (such as names / symbols and calls) are treated as one-element lists which cannot be unlisted even if `recursive = TRUE`.

If the output type is complex, logical, integer, and double NAs keep their imaginary parts zero when coerced, and hence will *not* become `NA_complex_` (with imaginary part NA).

There is a `c.factor` method which combines factors into a factor.

`c` is sometimes used for its side effect of removing attributes except names, for example to turn an array into a vector. `as.vector` is a more intuitive way to do this, but also drops names. Note that `c` methods other than the default are not required to remove attributes (and they will almost certainly preserve a class attribute).

This is a primitive function.

Value

NULL or an expression or a vector of an appropriate mode. (With no arguments the value is NULL.)

S4 methods

This function is S4 generic, but with argument list `(x, ...)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

unlist and as.vector to produce attribute-free vectors.

Examples

```
c(1, 7:9)
c(1:5, 10.5, "next")

## uses with a single argument to drop attributes
x <- 1:4
names(x) <- letters[1:4]
x
c(x)          # has names
as.vector(x)  # no names
dim(x) <- c(2,2)
x
c(x)
as.vector(x)

## append to a list:
ll <- list(A = 1, c = "C")
```

```
## do *not* use
c(ll, d = 1:3) # which is == c(ll, as.list(c(d = 1:3)))
## but rather
c(ll, d = list(1:3))  # c() combining two lists

## descend through lists:
c(list(A = c(B = 1)), recursive = TRUE)
c(list(A = c(B = 1, C = 2), B = c(E = 7)), recursive = TRUE)
```

`call` *Function Calls*

Description

Create or test for objects of mode `"call"` (or `"("`, see Details).

Usage

```
call(name, ...)
is.call(x)
as.call(x)
```

Arguments

`name`	a non-empty character string naming the function to be called.
`...`	arguments to be part of the call.
`x`	an arbitrary R object.

Details

`call` returns an unevaluated function call, that is, an unevaluated expression which consists of the named function applied to the given arguments (`name` must be a string which gives the name of a function to be called). Note that although the call is unevaluated, the arguments `...` are evaluated.

`call` is a primitive, so the first argument is taken as `name` and the remaining arguments as arguments for the constructed call: if the first argument is named the name must partially match `name`.

`is.call` is used to determine whether `x` is a call (i.e., of mode `"call"` or `"("`). Note that

- `is.call(x)` is strictly equivalent to `typeof(x) == "language"`.
- `is.language()` is also true for calls (but also for `symbol`s and `expression`s where `is.call()` is false).
- When `is.call(cl)` is true, `class(cl)` typically returns `"call"`, except when `cl` is one of `if`, `for`, `while`, `(`, `{`, `<-`, `=`, which each has its own `class(cl)` (equal to the "function" name), see the 'Special calls' example.

`as.call(x)`: Objects of mode `"list"` can be coerced to mode `"call"`. The first element of the list becomes the function part of the call, so should be a function or the name of one (as a symbol; a character string will not do).

If you think of using `as.call(`*string*`)`, consider using `str2lang(`*string*`)` which is an efficient version of `parse(text=`*string*`)`. Note that `call()` and `as.call()`, when applicable, are much preferable to these `parse()` based approaches.

All three are primitive functions.

`as.call` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Warning

`call` should not be used to attempt to evade restrictions on the use of `.Internal` and other non-API calls.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`do.call` for calling a function by name and argument list; `Recall` for recursive calling of functions; further `is.language`, `expression`, `function`.

Producing `call`s etc from character: `str2lang` and `parse`.

Examples

```
is.call(call) #-> FALSE: Functions are NOT calls

## set up a function call to round with argument 10.5
cl <- call("round", 10.5)
is.call(cl) # TRUE
cl
identical(quote(round(10.5)), # <- less functional, but the same
          cl) # TRUE
## such a call can also be evaluated.
eval(cl) # [1] 10

class(cl) # "call"
typeof(cl)# "language"
is.call(cl) && is.language(cl) # always TRUE for "call"s

A <- 10.5
call("round", A)        # round(10.5)
call("round", quote(A)) # round(A)
f <- "round"
call(f, quote(A))       # round(A)
## if we want to supply a function we need to use as.call or similar
f <- round
```

```
## Not run: call(f, quote(A))  # error: first arg must be character
(g <- as.call(list(f, quote(A))))
eval(g)
## alternatively but less transparently
g <- list(f, quote(A))
mode(g) <- "call"
g
eval(g)

## Special calls (and some regular ones):
L <- as.list(E <- setNames( , c("if", "for", "while", "repeat", "function",
                                "(", "{", "[",  "<-", "<<-", "->", "=")))
for(i in seq_along(L)) L[[i]] <- call(E[[i]]) # instead of lapply(E, call) ..
list_ <- function (...) `names<-`(list(...), vapply(sys.call()[-1L], as.character, ""))
(Tab <- noquote(sapply(list_(is.call, typeof, class, mode), \(F) sapply(L, F))))
## The 7 exceptions:
Tab[ Tab[,"class"] != "call" , c(3:4, 1:2)]

## see also the examples in the help for do.call
```

`callCC` *Call With Current Continuation*

Description

A downward-only version of Scheme's call with current continuation.

Usage

```
callCC(fun)
```

Arguments

`fun`	function of one argument, the exit procedure.

Details

`callCC` provides a non-local exit mechanism that can be useful for early termination of a computation. `callCC` calls `fun` with one argument, an *exit function*. The exit function takes a single argument, the intended return value. If the body of `fun` calls the exit function then the call to `callCC` immediately returns, with the value supplied to the exit function as the value returned by `callCC`.

Author(s)

Luke Tierney

Examples

```
# The following all return the value 1
callCC(function(k) 1)
callCC(function(k) k(1))
callCC(function(k) {k(1); 2})
callCC(function(k) repeat k(1))
```

`CallExternal` *Modern Interfaces to C/C++ code*

Description

Functions to pass R objects to compiled C/C++ code that has been loaded into R.

Usage

```
    .Call(.NAME, ..., PACKAGE)
.External(.NAME, ..., PACKAGE)
```

Arguments

`.NAME`	a character string giving the name of a C function, or an object of class "NativeSymbolInfo", "RegisteredNativeSymbol" or "NativeSymbol" referring to such a name.
`...`	arguments to be passed to the compiled code. Up to 65 for `.Call`.
`PACKAGE`	if supplied, confine the search for a character string `.NAME` to the DLL given by this argument (plus the conventional extension, '`.so`', '`.dll`', ...). This argument follows `...` and so its name cannot be abbreviated. This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols, and also speeds up the search (see 'Note').

Details

The functions are used to call compiled code which makes use of internal R objects, passing the arguments to the code as a sequence of R objects. They assume C calling conventions, so can usually also be used for C++ code.

For details about how to write code to use with these functions see the chapter on 'System and foreign language interfaces' in the 'Writing R Extensions' manual. They differ in the way the arguments are passed to the C code: `.External` allows for a variable or unlimited number of arguments.

These functions are primitive, and `.NAME` is always matched to the first argument supplied (which should not be named). For clarity, avoid using names in the arguments passed to `...` that match or partially match `.NAME`.

Value

An R object constructed in the compiled code.

Header files for external code

Writing code for use with these functions will need to use internal R structures defined in 'Rinternals.h' and/or the macros in 'Rdefines.h'.

Note

If one of these functions is to be used frequently, do specify PACKAGE (to confine the search to a single DLL) or pass .NAME as one of the native symbol objects. Searching for symbols can take a long time, especially when many namespaces are loaded.

You may see PACKAGE = "base" for symbols linked into R. Do not use this in your own code: such symbols are not part of the API and may be changed without warning.

PACKAGE = "" used to be accepted (but was undocumented): it is now an error.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (.Call.)

See Also

dyn.load, .C, .Fortran.

The 'Writing R Extensions' manual.

---

capabilities *Report Capabilities of this Build of R*

Description

Report on the optional features which have been compiled into this build of R.

Usage

```
capabilities(what = NULL,
             Xchk = any(nas %in% c("X11", "jpeg", "png", "tiff")))
```

Arguments

what	character vector or NULL, specifying required components. NULL implies that all are required.
Xchk	logical with a smart default, indicating if X11-related capabilities should be fully checked, notably on macOS. If set to false, may avoid a warning "No protocol specified" and e.g., the "X11" capability may be returned as NA.

Value

A named logical vector. Current components are

jpeg	is the jpeg function operational?
png	is the png function operational?
tiff	is the tiff function operational?
tcltk	is the **tcltk** package operational? Note that to make use of Tk you will almost always need to check that "X11" is also available.
X11	are the X11 graphics device and the X11-based data editor available? This loads the X11 module if not already loaded, and checks that the default display can be contacted unless a X11 device has already been used.
aqua	is the quartz function operational? Only on some macOS builds, including CRAN binary distributions of R. Note that this is distinct from .Platform$GUI == "AQUA", which is true only when using the Mac R.app GUI console.
http/ftp	does the default method for url and download.file support 'http://' and 'ftp://' URLs? Always TRUE as from R 3.3.0. However, in recent versions the default method is "libcurl" which depends on an external library and it is conceivable that library might not support 'ftp://' in future.
sockets	are make.socket and related functions available? Always TRUE as from R 3.3.0.
libxml	is there support for integrating libxml with the R event loop? TRUE as from R 3.3.0, FALSE as from R 4.2.0.
fifo	are FIFO connections supported?
cledit	is command-line editing available in the current R session? This is false in non-interactive sessions. It will be true for the command-line interface if readline support has been compiled in and '--no-readline' was *not* used when R was invoked. (If '--interactive' was used, command-line editing will not actually be available.)
iconv	is internationalization conversion via iconv supported? Always true in current R.
NLS	is there Natural Language Support (for message translations)?
Rprof	is there support for Rprof() profiling? This is true if R was configured (before compilation) with default settings which include --enable-R-profiling.
profmem	is there support for memory profiling? See tracemem.
cairo	is there support for the svg, cairo_pdf and cairo_ps devices, and for type = "cairo" in the bmp, jpeg, png and tiff devices? Prior to R 4.1.0 this also indicated Cairo support in the X11 device, but it is now possible to build R with Cairo support for the bitmap devices without support for the X11 device (usually when that is not supported at all).
ICU	is ICU available for collation? See the help on Comparison and icuSetCollate: it is never used for a C locale.

`long.double` does this build use a `C` `long double` type which is longer than `double`? Some platforms do not have such a type, and on others its use can be suppressed by the configure option '`--disable-long-double`'.

Although not guaranteed, it is a reasonable assumption that if present long doubles will have at least as much range and accuracy as the ISO/IEC 60559 80-bit 'extended precision' format. Since R 4.0.0 `.Machine` gives information on the long-double type (if present).

`libcurl` is `libcurl` available in this build? Used by function `curlGetHeaders` and optionally by `download.file` and `url`. As from R 3.3.0 always true for Unix-alikes, and as from R 4.2.0 true on Windows.

Note to macOS users

Capabilities `"jpeg"`, `"png"` and `"tiff"` refer to the X11-based versions of these devices. If `capabilities("aqua")` is true, then these devices with `type = "quartz"` will be available, and out-of-the-box will be the default type. Thus for example the `tiff` device will be available if `capabilities("aqua") || capabilities("tiff")` if the defaults are unchanged.

See Also

`.Platform`, `extSoftVersion`, and `grSoftVersion` (and links there) for availability of capabilities *external* to R but used from R functions.

Examples

```
capabilities()

if(!capabilities("ICU"))
   warning("ICU is not available")

## Does not call the internal X11-checking function:
capabilities(Xchk = FALSE)

## See also the examples for 'connections'.
```

---

cat *Concatenate and Print*

Description

Outputs the objects, concatenating the representations. `cat` performs much less conversion than `print`.

Usage

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Arguments

...	R objects (see 'Details' for the types of objects allowed).
file	a connection, or a character string naming the file to print to. If "" (the default), cat prints to the standard output connection, the console unless redirected by sink. If it is "|cmd", the output is piped to the command given by 'cmd', by opening a pipe connection.
sep	a character vector of strings to append after each element.
fill	a logical or (positive) numeric controlling how the output is broken into successive lines. If FALSE (default), only newlines created explicitly by '"\n"' are printed. Otherwise, the output is broken into lines with print width equal to the option width if fill is TRUE, or the value of fill if this is numeric. Linefeeds are only inserted *between* elements, strings wider than fill are not wrapped. Non-positive fill values are ignored, with a warning.
labels	character vector of labels for the lines printed. Ignored if fill is FALSE.
append	logical. Only used if the argument file is the name of file (and not a connection or "|cmd"). If TRUE output will be appended to file; otherwise, it will overwrite the contents of file.

Details

cat is useful for producing output in user-defined functions. It converts its arguments to character vectors, concatenates them to a single character vector, appends the given sep = string(s) to each element and then outputs them.

No line feeds (aka "newline"s) are output unless explicitly requested by '"\n"' or if generated by filling (if argument fill is TRUE or numeric).

If file is a connection and open for writing it is written from its current position. If it is not open, it is opened for the duration of the call in "wt" mode and then closed again.

Currently only atomic vectors and names are handled, together with NULL and other zero-length objects (which produce no output). Character strings are output 'as is' (unlike print.default which escapes non-printable characters and backslash — use encodeString if you want to output encoded strings using cat). Other types of R object should be converted (e.g., by as.character or format) before being passed to cat. That includes factors, which are output as integer vectors.

cat converts numeric/complex elements in the same way as print (and not in the same way as as.character which is used by the S equivalent), so options "digits" and "scipen" are relevant. However, it uses the minimum field width necessary for each element, rather than the same field width for all elements.

Value

None (invisible NULL).

Note

If any element of sep contains a newline character, it is treated as a vector of terminators rather than separators, an element being output after every vector element *and* a newline after the last. Entries are recycled as needed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`print`, `format`, and `paste` which concatenates into a string.

Examples

```
iter <- stats::rpois(1, lambda = 10)
## print an informative message
cat("iteration = ", iter <- iter + 1, "\n")

## 'fill' and label lines:
cat(paste(letters, 100* 1:26), fill = TRUE, labels = paste0("{", 1:10, "}:"))
```

---

cbind *Combine R Objects by Rows or Columns*

Description

Take a sequence of vector, matrix or data-frame arguments and combine by *c*olumns or *r*ows, respectively. These are generic functions with methods for other R classes.

Usage

```
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
## S3 method for class 'data.frame'
rbind(..., deparse.level = 1, make.row.names = TRUE,
      stringsAsFactors = FALSE, factor.exclude = TRUE)
```

Arguments

`...`	(generalized) vectors or matrices. These can be given as named arguments. Other R objects may be coerced as appropriate, or S4 methods may be used: see sections 'Details' and 'Value'. (For the `"data.frame"` method of `cbind` these can be further arguments to `data.frame` such as `stringsAsFactors`.)
`deparse.level`	integer controlling the construction of labels in the case of non-matrix-like arguments (for the default method): `deparse.level = 0` constructs no labels; the default `deparse.level = 1` typically and `deparse.level = 2` always construct labels from the argument names, see the 'Value' section below.
`make.row.names`	(only for data frame method:) logical indicating if unique and valid `row.names` should be constructed from the arguments.

`stringsAsFactors`
: logical, passed to `as.data.frame`; only has an effect when the `...` arguments contain a (non-`data.frame`) `character`.

`factor.exclude`
: if the data frames contain factors, the default `TRUE` ensures that `NA` levels of factors are kept, see PR#17562 and the 'Data frame methods'. In R versions up to 3.6.x, `factor.exclude = NA` has been implicitly hardcoded (R <= 3.6.0) or the default (R = 3.6.x, x >= 1).

Details

The functions `cbind` and `rbind` are S3 generic, with methods for data frames. The data frame method will be used if at least one argument is a data frame and the rest are vectors or matrices. There can be other methods; in particular, there is one for time series objects. See the section on 'Dispatch' for how the method to be used is selected. If some of the arguments are of an S4 class, i.e., `isS4(.)` is true, S4 methods are sought also, and the hidden `cbind` / `rbind` functions from package **methods** maybe called, which in turn build on `cbind2` or `rbind2`, respectively. In that case, `deparse.level` is obeyed, similarly to the default method.

In the default method, all the vectors/matrices must be atomic (see `vector`) or lists. Expressions are not allowed. Language objects (such as formulae and calls) and pairlists will be coerced to lists: other objects (such as names and external pointers) will be included as elements in a list result. Any classes the inputs might have are discarded (in particular, factors are replaced by their internal codes).

If there are several matrix arguments, they must all have the same number of columns (or rows) and this will be the number of columns (or rows) of the result. If all the arguments are vectors, the number of columns (rows) in the result is equal to the length of the longest vector. Values in shorter arguments are recycled to achieve this length (with a `warning` if they are recycled only *fractionally*).

When the arguments consist of a mix of matrices and vectors the number of columns (rows) of the result is determined by the number of columns (rows) of the matrix arguments. Any vectors have their values recycled or subsetted to achieve this length.

For `cbind` (`rbind`), vectors of zero length (including `NULL`) are ignored unless the result would have zero rows (columns), for S compatibility. (Zero-extent matrices do not occur in S3 and are not ignored in R.)

Matrices are restricted to less than 2^{31} rows and columns even on 64-bit systems. So input vectors have the same length restriction: as from R 3.2.0 input matrices with more elements (but meeting the row and column restrictions) are allowed.

Value

For the default method, a matrix combining the `...` arguments column-wise or row-wise. (Exception: if there are no inputs or all the inputs are `NULL`, the value is `NULL`.)

The type of a matrix result determined from the highest type of any of the inputs in the hierarchy raw < logical < integer < double < complex < character < list .

For `cbind` (`rbind`) the column (row) names are taken from the `colnames` (`rownames`) of the arguments if these are matrix-like. Otherwise from the names of the arguments or where those are not supplied and `deparse.level > 0`, by deparsing the expressions given, for `deparse.level = 1` only if that gives a sensible name (a 'symbol', see `is.symbol`).

For cbind row names are taken from the first argument with appropriate names: rownames for a matrix, or names for a vector of length the number of rows of the result.

For rbind column names are taken from the first argument with appropriate names: colnames for a matrix, or names for a vector of length the number of columns of the result.

Data frame methods

The cbind data frame method is just a wrapper for data.frame(..., check.names = FALSE). This means that it will split matrix columns in data frame arguments, and convert character columns to factors unless stringsAsFactors = FALSE is specified.

The rbind data frame method first drops all zero-column and zero-row arguments. (If that leaves none, it returns the first argument with columns otherwise a zero-column zero-row data frame.) It then takes the classes of the columns from the first data frame, and matches columns by name (rather than by position). Factors have their levels expanded as necessary (in the order of the levels of the level sets of the factors encountered) and the result is an ordered factor if and only if all the components were ordered factors. Old-style categories (integer vectors with levels) are promoted to factors.

Note that for result column j, factor(., exclude = X(j)) is applied, where

```
X(j) := if(isTRUE(factor.exclude)) {
           if(!NA.lev[j]) NA # else NULL
        } else factor.exclude
```

where NA.lev[j] is true iff any contributing data frame has had a factor in column j with an explicit NA level.

Dispatch

The method dispatching is *not* done via UseMethod(), but by C-internal dispatching. Therefore there is no need for, e.g., rbind.default.

The dispatch algorithm is described in the source file ('.../src/main/bind.c') as

1. For each argument we get the list of possible class memberships from the class attribute.
2. We inspect each class in turn to see if there is an applicable method.
3. If we find a method, we use it. Otherwise, if there was an S4 object among the arguments, we try S4 dispatch; otherwise, we use the default code.

If you want to combine other objects with data frames, it may be necessary to coerce them to data frames first. (Note that this algorithm can result in calling the data frame method if all the arguments are either data frames or vectors, and this will result in the coercion of character vectors to factors.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c to combine vectors (and lists) as vectors, data.frame to combine vectors and matrices as a data frame.

Examples

```
m <- cbind(1, 1:7) # the '1' (= shorter vector) is recycled
m
m <- cbind(m, 8:14)[, c(1, 3, 2)] # insert a column
m
cbind(1:7, diag(3)) # vector is subset -> warning

cbind(0, rbind(1, 1:3))
cbind(I = 0, X = rbind(a = 1, b = 1:3))  # use some names
xx <- data.frame(I = rep(0,2))
cbind(xx, X = rbind(a = 1, b = 1:3))   # named differently

cbind(0, matrix(1, nrow = 0, ncol = 4)) #> Warning (making sense)
dim(cbind(0, matrix(1, nrow = 2, ncol = 0))) #-> 2 x 1

## deparse.level
dd <- 10
rbind(1:4, c = 2, "a++" = 10, dd, deparse.level = 0) # middle 2 rownames
rbind(1:4, c = 2, "a++" = 10, dd, deparse.level = 1) # 3 rownames (default)
rbind(1:4, c = 2, "a++" = 10, dd, deparse.level = 2) # 4 rownames

## cheap row names:
b0 <- gl(3,4, labels=letters[1:3])
bf <- setNames(b0, paste0("o", seq_along(b0)))
df  <- data.frame(a = 1, B = b0, f = gl(4,3))
df. <- data.frame(a = 1, B = bf, f = gl(4,3))
new <- data.frame(a = 8, B ="B", f = "1")
(df1  <- rbind(df , new))
(df.1 <- rbind(df., new))
stopifnot(identical(df1, rbind(df,  new, make.row.names=FALSE)),
          identical(df1, rbind(df., new, make.row.names=FALSE)))
```

---

char.expand *Expand a String with Respect to a Target Table*

Description

Seeks a unique match of its first argument among the elements of its second. If successful, it returns this element; otherwise, it performs an action specified by the third argument.

Usage

```
char.expand(input, target, nomatch = stop("no match"))
```

Arguments

input	a character string to be expanded.
target	a character vector with the values to be matched against.
nomatch	an R expression to be evaluated in case expansion was not possible.

Details

This function is particularly useful when abbreviations are allowed in function arguments, and need to be uniquely expanded with respect to a target table of possible values.

Value

A length-one character vector, one of the elements of target (unless nomatch is changed to be a non-error, when it can be a zero-length character string).

See Also

charmatch and pmatch for performing partial string matching.

Examples

```
locPars <- c("mean", "median", "mode")
char.expand("me", locPars, warning("Could not expand!"))
char.expand("mo", locPars)
```

---

character *Character Vectors*

Description

Create or test for objects of type "character".

Usage

```
character(length = 0)
as.character(x, ...)
is.character(x)
```

Arguments

length	a non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

as.character and is.character are generic: you can write methods to handle specific classes of objects, see InternalMethods. Further, for as.character the default method calls as.vector, so, only if(is.object(x)) is true, dispatch is first on methods for as.character and then for methods for as.vector.

as.character represents real and complex numbers to 15 significant digits (technically the compiler's setting of the ISO C constant DBL_DIG, which will be 15 on machines supporting IEC 60559 arithmetic according to the C99 standard). This ensures that all the digits in the result will be reliable (and not the result of representation error), but does mean that conversion to character and back to numeric may change the number. If you want to convert numbers to character with the maximum possible precision, use format.

Value

character creates a character vector of the specified length. The elements of the vector are all equal to "".

as.character attempts to coerce its argument to character type; like as.vector it strips attributes including names. For lists and pairlists (including language objects such as calls) it deparses the elements individually, except that it extracts the first element of length-one character vectors, see the Abc example.

is.character returns TRUE or FALSE depending on whether its argument is of character type or not.

Note

as.character breaks lines in language objects at 500 characters, and inserts newlines. Prior to 2.15.0 lines were truncated.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

options: options scipen and OutDec affect the conversion of numbers.

paste, substr and strsplit for character concatenation and splitting, chartr for character translation and case folding (e.g., upper to lower case) and sub, grep etc for string matching and substitutions. Note that help.search(keyword = "character") gives even more links.

deparse, which is normally preferable to as.character for language objects.

Quotes on how to specify character / string constants, including *raw* ones.

Examples

```
form <- y ~ a + b + c
as.character(form)  ## length 3
deparse(form)       ## like the input
```

```
a0 <- 11/999          # has a repeating decimal representation
(a1 <- as.character(a0))
format(a0, digits = 16) # shows 1 to 2 more digit(s)
a2 <- as.numeric(a1)
a2 - a0               # normally around -1e-17
as.character(a2)      # possibly different from a1
print(c(a0, a2), digits = 16)

as.character(list(A = "Abc", xy = c("x", "y"))) # "Abc"  "c(\"x\", \"y\")"
## i.e., "Abc" directly instead of deparsing to "\"Abc\""
```

---

charmatch *Partial String Matching*

Description

charmatch seeks matches for the elements of its first argument among those of its second.

Usage

```
charmatch(x, table, nomatch = NA_integer_)
```

Arguments

x	the values to be matched: converted to a character vector by as.character. Long vectors are supported.
table	the values to be matched against: converted to a character vector. Long vectors are not supported.
nomatch	the (integer) value to be returned at non-matching positions.

Details

Exact matches are preferred to partial matches (those where the value to be matched has an exact match to the initial part of the target, but the target is longer).

If there is a single exact match or no exact match and a unique partial match then the index of the matching value is returned; if multiple exact or multiple partial matches are found then 0 is returned and if no match is found then nomatch is returned.

NA values are treated as the string constant "NA".

Value

An integer vector of the same length as x, giving the indices of the elements in table which matched, or nomatch.

Author(s)

This function is based on a C function written by Terry Therneau.

See Also

pmatch, match.

startsWith for another matching of initial parts of strings; grep or regexpr for more general (regexp) matching of strings.

Examples

```
charmatch("", "")                          # returns 1
charmatch("m",   c("mean", "median", "mode")) # returns 0
charmatch("med", c("mean", "median", "mode")) # returns 2
```

---

chartr *Character Translation and Case Folding*

Description

Translate characters in character vectors, in particular from upper to lower case or vice versa.

Usage

```
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
```

Arguments

x	a character vector, or an object that can be coerced to character by as.character.
old	a character string specifying the characters to be translated. If a character vector of length 2 or more is supplied, the first element is used with a warning.
new	a character string specifying the translations. If a character vector of length 2 or more is supplied, the first element is used with a warning.
upper	logical: translate to upper or lower case?

Details

chartr translates each character in x that is specified in old to the corresponding character specified in new. Ranges are supported in the specifications, but character classes and repeated characters are not. If old contains more characters than new, an error is signaled; if it contains fewer characters, the extra characters at the end of new are ignored.

tolower and toupper convert upper-case characters in a character vector to lower-case, or vice versa. Non-alphabetic characters are left unchanged. More than one character can be mapped to a single upper-case character.

casefold is a wrapper for tolower and toupper originally written for compatibility with S-PLUS.

Value

A character vector of the same length and with the same attributes as x (after possible coercion).

Elements of the result will be have the encoding declared as that of the current locale (see Encoding) if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8. The result will be in the current locale's encoding unless the corresponding input was in UTF-8 or Latin-1, when it will be in UTF-8.

Note

These functions are platform-dependent, usually using OS services. The latter can be quite deficient, for example only covering ASCII characters in 8-bit locales. The definition of 'alphabetic' is platform-dependent and liable to change over time as most platforms are based on the frequently-updated Unicode tables.

See Also

sub and gsub for other substitutions in strings.

Examples

```
x <- "MiXeD cAsE 123"
chartr("iXs", "why", x)
chartr("a-cX", "D-Fw", x)
tolower(x)
toupper(x)

## "Mixed Case" Capitalizing - toupper( every first letter of a word ) :

.simpleCap <- function(x) {
    s <- strsplit(x, " ")[[1]]
    paste(toupper(substring(s, 1, 1)), substring(s, 2),
          sep = "", collapse = " ")
}
.simpleCap("the quick red fox jumps over the lazy brown dog")
## ->  [1] "The Quick Red Fox Jumps Over The Lazy Brown Dog"

## and the better, more sophisticated version:
capwords <- function(s, strict = FALSE) {
    cap <- function(s) paste(toupper(substring(s, 1, 1)),
                  {s <- substring(s, 2); if(strict) tolower(s) else s},
                             sep = "", collapse = " " )
    sapply(strsplit(s, split = " "), cap, USE.NAMES = !is.null(names(s)))
}
capwords(c("using AIC for model selection"))
## ->  [1] "Using AIC For Model Selection"
capwords(c("using AIC", "for MODEL selection"), strict = TRUE)
## ->  [1] "Using Aic"  "For Model Selection"
##                ^^^        ^^^^^
##               'bad'       'good'

## -- Very simple insecure crypto --
```

```
rot <- function(ch, k = 13) {
   p0 <- function(...) paste(c(...), collapse = "")
   A <- c(letters, LETTERS, " '")
   I <- seq_len(k); chartr(p0(A), p0(c(A[-I], A[I])), ch)
}

pw <- "my secret pass phrase"
(crypw <- rot(pw, 13)) #-> you can send this off

## now ``decrypt'' :
rot(crypw, 54 - 13) # -> the original:
stopifnot(identical(pw, rot(crypw, 54 - 13)))
```

---

chkDots *Warn About Extraneous Arguments in the "..." of Its Caller*

Description

Warn about extraneous arguments in the `...` of its caller. A utility to be used e.g., in S3 methods which need a formal `...` argument but do not make any use of it. This helps catching user errors in calling the function in question (which is the caller of `chkDots()`).

Usage

```
chkDots(..., which.call = -1, allowed = character(0))
```

Arguments

`...`	"the dots", as passed from the caller.
`which.call`	passed to `sys.call()`. A caller may use -2 if the message should mention *its* caller.
`allowed`	not yet implemented: character vector of *named* elements in `...` which are "allowed" and hence not warned about.

Author(s)

Martin Maechler, first version outside base, June 2012.

See Also

`warning`, `....`

Examples

```
seq.default ## <- you will see  ' chkDots(...) '

seq(1,5, foo = "bar") # gives warning via chkDots()

## warning with more than one ...-entry:
density.f <- function(x, ...) NextMethod("density")
x <- density(structure(rnorm(10), class="f"), bar=TRUE, baz=TRUE)
```

---

chol *The Cholesky Decomposition*

Description

Compute the Cholesky factorization of a real symmetric positive-definite square matrix.

Usage

```
chol(x, ...)

## Default S3 method:
chol(x, pivot = FALSE,  LINPACK = FALSE, tol = -1, ...)
```

Arguments

x	an object for which a method exists. The default method applies to numeric (or logical) symmetric, positive-definite matrices.
...	arguments to be passed to or from methods.
pivot	logical: should pivoting be used?
LINPACK	logical. Defunct and gives an error.
tol	a numeric tolerance for use with pivot = TRUE.

Details

chol is generic: the description here applies to the default method.

Note that only the upper triangular part of x is used, so that $R'R = x$ when x is symmetric.

If pivot = FALSE and x is not non-negative definite an error occurs. If x is positive semi-definite (i.e., some zero eigenvalues) an error will also occur as a numerical tolerance is used.

If pivot = TRUE, then the Cholesky decomposition of a positive semi-definite x can be computed. The rank of x is returned as attr(Q, "rank"), subject to numerical errors. The pivot is returned as attr(Q, "pivot"). It is no longer the case that t(Q) %*% Q equals x. However, setting pivot <- attr(Q, "pivot") and oo <- order(pivot), it is true that t(Q[, oo]) %*% Q[, oo] equals x, or, alternatively, t(Q) %*% Q equals x[pivot, pivot]. See the examples.

The value of `tol` is passed to LAPACK, with negative values selecting the default tolerance of (usually) `nrow(x) * .Machine$double.neg.eps * max(diag(x))`. The algorithm terminates once the pivot is less than `tol`.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The upper triangular factor of the Cholesky decomposition, i.e., the matrix R such that $R'R = x$ (see example).

If pivoting is used, then two additional attributes `"pivot"` and `"rank"` are also returned.

Warning

The code does not check for symmetry.

If `pivot = TRUE` and `x` is not non-negative definite then there will be a warning message but a meaningless result will occur. So only use `pivot = TRUE` when `x` is non-negative definite by construction.

Source

This is an interface to the LAPACK routines `DPOTRF` and `DPSTRF`,

LAPACK is from https://netlib.org/lapack/ and its guide is listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`chol2inv` for its *inverse* (without pivoting), `backsolve` for solving linear systems with upper triangular left sides.

`qr`, `svd` for related matrix factorizations.

Examples

```
( m <- matrix(c(5,1,1,3),2,2) )
( cm <- chol(m) )
t(cm) %*% cm  #-- = 'm'
crossprod(cm)  #-- = 'm'

# now for something positive semi-definite
x <- matrix(c(1:5, (1:5)^2), 5, 2)
x <- cbind(x, x[, 1] + 3*x[, 2])
colnames(x) <- letters[20:22]
m <- crossprod(x)
qr(m)$rank # is 2, as it should be
```

```
# chol() may fail, depending on numerical rounding:
# chol() unlike qr() does not use a tolerance.
try(chol(m))

(Q <- chol(m, pivot = TRUE))
## we can use this by
pivot <- attr(Q, "pivot")
crossprod(Q[, order(pivot)]) # recover m

## now for a non-positive-definite matrix
( m <- matrix(c(5,-5,-5,3), 2, 2) )
try(chol(m))  # fails
(Q <- chol(m, pivot = TRUE)) # warning
crossprod(Q)  # not equal to m
```

---

chol2inv *Inverse from Cholesky (or QR) Decomposition*

Description

Invert a symmetric, positive definite square matrix from its Cholesky decomposition. Equivalently, compute $(X'X)^{-1}$ from the (R part) of the QR decomposition of X.

Usage

```
chol2inv(x, size = NCOL(x), LINPACK = FALSE)
```

Arguments

x	a matrix. The first size columns of the upper triangle contain the Cholesky decomposition of the matrix to be inverted.
size	the number of columns of x containing the Cholesky decomposition.
LINPACK	logical. Defunct and gives an error.

Value

The inverse of the matrix whose Cholesky decomposition was given.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Source

This is an interface to the LAPACK routine DPOTRI. LAPACK is from https://netlib.org/lapack/ and its guide is listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM. Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

See Also

chol, solve.

Examples

```
cma <- chol(ma  <- cbind(1, 1:3, c(1,3,7)))
ma %*% chol2inv(cma)
```

---

chooseOpsMethod *Choose the Appropriate Method for Ops*

Description

chooseOpsMethod is a function called by the Ops Group Generic when two suitable methods are found for a given call. It determines which method to use for the operation based on the objects being dispatched.

The function is first called with reverse = FALSE, where x corresponds to the first argument and y to the second argument of the group generic call. If chooseOpsMethod() returns FALSE for x, then chooseOpsMethod is called again, with x and y swapped, mx and my swapped, and reverse = TRUE.

Usage

```
chooseOpsMethod(x, y, mx, my, cl, reverse)
```

Arguments

x, y	the objects being dispatched on by the group generic.
mx, my	the methods found for objects x and y.
cl	the call to the group generic.
reverse	logical value indicating whether x and y are reversed from the way they were supplied to the generic.

Value

This function must return either TRUE or FALSE. A value of TRUE indicates that method mx should be used.

See Also

Ops

Examples

```
# Create two objects with custom Ops methods
foo_obj <- structure(1, class = "foo")
bar_obj <- structure(1, class = "bar")

`+.foo` <- function(e1, e2) "foo"
Ops.bar <- function(e1, e2) "bar"

invisible(foo_obj + bar_obj) # Warning: Incompatible methods

chooseOpsMethod.bar <- function(x, y, mx, my, cl, reverse) TRUE

stopifnot(exprs = {
  identical(foo_obj + bar_obj, "bar")
  identical(bar_obj + foo_obj, "bar")
})

# cleanup
rm(foo_obj, bar_obj, `+.foo`, Ops.bar, chooseOpsMethod.bar)
```

---

class *Object Classes*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class of the first argument to the generic function.

Usage

```
class(x)
class(x) <- value
unclass(x)
inherits(x, what, which = FALSE)
nameOfClass(x)
isa(x, what)

oldClass(x)
oldClass(x) <- value
.class2(x)
```

Arguments

x	an R object.
what, value	a character vector naming classes. value can also be NULL. what can also be a non-character R object with a nameOfClass() method.
which	logical affecting return value: see 'Details'.

Details

Here, we describe the so called "S3" classes (and methods). For "S4" classes (and methods), see 'Formal classes' below.

Many R objects have a `class` attribute, a character vector giving the names of the classes from which the object *inherits*. (Functions `oldClass` and `oldClass<-` get and set the attribute, which can also be done directly.)

If the object does not have a class attribute, it has an implicit class, notably `"matrix"`, `"array"`, `"function"` or `"numeric"` or the result of `typeof(x)` (which is similar to `mode(x)`), but for type `"language"` and `mode` `"call"`, where the following extra classes exist for the corresponding function `call`s: `if`, `for`, `while`, `(`, `{`, `<-`, `=`.

Note that for objects `x` of an implicit (or an S4) class, when a (S3) generic function `foo(x)` is called, method dispatch may use more classes than are returned by `class(x)`, e.g., for a numeric matrix, the `foo.numeric()` method may apply. The exact full `character` vector of the classes which `UseMethod()` uses, is available as `.class2(x)` since R version 4.0.0. (This also applies to S4 objects when S3 dispatch is considered, see below.)

Beware that using `.class2()` for other reasons than didactical, diagnostical or for debugging may rather be a misuse than smart.

`NULL` objects (of implicit class `"NULL"`) cannot have attributes (hence no `class` attribute) and attempting to assign a class is an error.

When a generic function `fun` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found, a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

The function `class` prints the vector of names of classes an object inherits from. Correspondingly, `class<-` sets the classes an object inherits from. Assigning an empty character vector or `NULL` removes the class attribute, as for `oldClass<-` or direct attribute setting. Whereas it is clearer to explicitly assign `NULL` to remove the class, using an empty vector is more natural in e.g., `class(x) <- setdiff(class(x), "ts")`.

`unclass` returns (a copy of) its argument with its class attribute removed. (It is not allowed for objects which cannot be copied, namely environments and external pointers.)

`inherits` indicates whether its first argument inherits from any of the classes specified in the `what` argument. If `which` is `TRUE` then an integer vector of the same length as `what` is returned. Each element indicates the position in the `class(x)` matched by the element of `what`; zero indicates no match. If `which` is `FALSE` then `TRUE` is returned by `inherits` if any of the names in `what` match with any `class`.

`nameOfClass` is an S3 generic. It is called by `inherits` to get the class name for `what`, allowing for `what` to be values other than a character vector. `nameOfClass` methods are expected to return a character vector of length 1.

`isa` tests whether `x` is an object of class(es) as given in `what` by using `is` if `x` is an S4 object, and otherwise giving `TRUE` iff *all* elements of `class(x)` are contained in `what`.

All but `inherits` and `isa` are primitive functions.

Formal classes

An additional mechanism of *formal* classes, nicknamed "S4", is available in package **methods** which is attached by default. For objects which have a formal class, its name is returned by `class` as a character vector of length one and method dispatch can happen on *several* arguments, instead of only the first. However, S3 method selection attempts to treat objects from an S4 class as if they had the appropriate S3 class attribute, as does `inherits`. Therefore, S3 methods can be defined for S4 classes. See the 'Introduction' and 'Methods_for_S3' help pages for basic information on S4 methods and for the relation between these and S3 methods.

The replacement version of the function sets the class to the value provided. For classes that have a formal definition, directly replacing the class this way is strongly deprecated. The expression `as(object, value)` is the way to coerce an object to a particular class.

The analogue of `inherits` for formal classes is `is`. The two functions behave consistently with one exception: S4 classes can have conditional inheritance, with an explicit test. In this case, `is` will test the condition, but `inherits` ignores all conditional superclasses.

Note

`UseMethod` dispatches on the class as returned by `class` (with some interpolated classes: see the link) rather than `oldClass`. *However*, group generics dispatch on the `oldClass` for efficiency, and internal generics only dispatch on objects for which `is.object` is true.

See Also

`UseMethod`, `NextMethod`, 'group generic', 'internal generic'

Examples

```
x <- 10
class(x) # "numeric"
oldClass(x) # NULL
inherits(x, "a") #FALSE
class(x) <- c("a", "b")
inherits(x,"a") #TRUE
inherits(x, "a", TRUE) # 1
inherits(x, c("a", "b", "c"), TRUE) # 1 2 0

class( quote(pi) )           # "name"
## regular calls
class( quote(sin(pi*x)) )    # "call"
## special calls
class( quote(x <- 1) )       # "<-"
class( quote((1 < 2)) )      # "("
class( quote( if(8<3) pi ) ) # "if"

.class2(pi)               # "double" "numeric"
.class2(matrix(1:6, 2,3)) # "matrix" "array" "integer" "numeric"
```

col *Column Indexes*

Description

Returns a matrix of integers indicating their column number in a matrix-like object, or a factor of column labels.

Usage

```
col(x, as.factor = FALSE)
.col(dim)
```

Arguments

x	a matrix-like object, that is one with a two-dimensional dim.
dim	a matrix dimension, i.e., an integer valued numeric vector of length two (with non-negative entries).
as.factor	a logical value indicating whether the value should be returned as a factor of column labels (created if necessary) rather than as numbers.

Value

An integer (or factor) matrix with the same dimensions as x and whose ij-th element is equal to j (or the j-th column label).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

row to get rows; slice.index for a general way to get slice indices in an array.

Examples

```
# extract an off-diagonal of a matrix
ma <- matrix(1:12, 3, 4)
ma[row(ma) == col(ma) + 1]

# create an identity 5-by-5 matrix more slowly than diag(n = 5):
x <- matrix(0, nrow = 5, ncol = 5)
x[row(x) == col(x)] <- 1

(i34 <- .col(3:4))
stopifnot(identical(i34, .col(c(3,4)))) # 'dim' maybe "double"
```

`Colon` *Colon Operator*

Description

Generate regular sequences.

Usage

```
from:to
   a:b
```

Arguments

`from`	starting value of sequence.
`to`	(maximal) end value of the sequence.
`a, b`	`factor`s of the same length.

Details

The binary operator `:` has two meanings: for factors `a:b` is equivalent to `interaction(a, b)` (but the levels are ordered and labelled differently).

For other arguments `from:to` is equivalent to `seq(from, to)`, and generates a sequence from `from` to `to` in steps of `1` or `-1`. Value `to` will be included if it differs from `from` by an integer up to a numeric fuzz of about `1e-7`. Non-numeric arguments are coerced internally (hence without dispatching methods) to numeric—complex values will have their imaginary parts discarded with a warning.

Value

For numeric arguments, a numeric vector. This will be of type `integer` if `from` is integer-valued and the result is representable in the R integer type, otherwise of type `"double"` (aka `mode` `"numeric"`).

For factors, an unordered factor with levels labelled as `la:lb` and ordered lexicographically (that is, `lb` varies fastest).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
(for numeric arguments: S does not have `:` for factors.)

See Also

`seq` (a *generalization* of `from:to`).

As an alternative to using `:` for factors, `interaction`.

For `:` used in the formal representation of an interaction, see `formula`.

Examples

```
1:4
pi:6 # real
6:pi # integer

f1 <- gl(2, 3); f1
f2 <- gl(3, 2); f2
f1:f2 # a factor, the "cross"  f1 x f2
```

---

colSums *Form Row and Column Sums and Means*

Description

Form row and column sums and means for numeric arrays (or data frames).

Usage

```
colSums (x, na.rm = FALSE, dims = 1)
rowSums (x, na.rm = FALSE, dims = 1)
colMeans(x, na.rm = FALSE, dims = 1)
rowMeans(x, na.rm = FALSE, dims = 1)

.colSums(x, m, n, na.rm = FALSE)
.rowSums(x, m, n, na.rm = FALSE)
.colMeans(x, m, n, na.rm = FALSE)
.rowMeans(x, m, n, na.rm = FALSE)
```

Arguments

x	an array of two or more dimensions, containing numeric, complex, integer or logical values, or a numeric data frame. For `.colSums()` etc, a numeric, integer or logical matrix (or vector of length `m * n`).
na.rm	logical. Should missing values (including NaN) be omitted from the calculations?
dims	integer: Which dimensions are regarded as 'rows' or 'columns' to sum over. For `row*`, the sum or mean is over dimensions `dims+1, ...`; for `col*` it is over dimensions `1:dims`.
m, n	the dimensions of the matrix x for `.colSums()` etc.

Details

These functions are equivalent to use of apply with FUN = mean or FUN = sum with appropriate margins, but are a lot faster. As they are written for speed, they blur over some of the subtleties of NaN and NA. If na.rm = FALSE and either NaN or NA appears in a sum, the result will be one of NaN or NA, but which might be platform-dependent.

Notice that omission of missing values is done on a per-column or per-row basis, so column means may not be over the same set of rows, and vice versa. To use only complete rows or columns, first select them with `na.omit` or `complete.cases` (possibly on the transpose of `x`).

The versions with an initial dot in the name (`.colSums()` etc) are 'bare-bones' versions for use in programming: they apply only to numeric (like) matrices and do not name the result.

Value

A numeric or complex array of suitable size, or a vector if the result is one-dimensional. For the first four functions the `dimnames` (or `names` for a vector result) are taken from the original array.

If there are no values in a range to be summed over (after removing missing values with `na.rm = TRUE`), that component of the output is set to `0` (`*Sums`) or `NaN` (`*Means`), consistent with `sum` and `mean`.

See Also

`apply`, `rowsum`

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
rowSums(x); colSums(x)
dimnames(x)[[1]] <- letters[1:8]
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
x[] <- as.integer(x)
rowSums(x); colSums(x)
x[] <- x < 3
rowSums(x); colSums(x)
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)

## an array
dim(UCBAdmissions)
rowSums(UCBAdmissions); rowSums(UCBAdmissions, dims = 2)
colSums(UCBAdmissions); colSums(UCBAdmissions, dims = 2)

## complex case
x <- cbind(x1 = 3 + 2i, x2 = c(4:1, 2:5) - 5i)
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)
```

commandArgs *Extract Command Line Arguments*

Description

Provides access to a copy of the command line arguments supplied when this R session was invoked.

Usage

```
commandArgs(trailingOnly = FALSE)
```

Arguments

trailingOnly	logical. Should only arguments after '--args' be returned?

Details

These arguments are captured before the standard R command line processing takes place. This means that they are the unmodified values. This is especially useful with the '--args' command-line flag to R, as all of the command line after that flag is skipped.

Value

A character vector containing the name of the executable and the user-supplied command line arguments. The first element is the name of the executable by which R was invoked. The exact form of this element is platform dependent: it may be the fully qualified name, or simply the last component (or basename) of the application, or for an embedded R it can be anything the programmer supplied.

If trailingOnly = TRUE, a character vector of those arguments (if any) supplied after '--args'.

See Also

R.home(), Startup and BATCH

Examples

```
commandArgs()
## Spawn a copy of this application as it was invoked,
## subject to shell quoting issues
## system(paste(commandArgs(), collapse = " "))
```

comment *Query or Set a "comment" Attribute*

Description

These functions set and query a *comment* attribute for any R objects. This is typically useful for `data.frame`s or model fits.

Contrary to other `attributes`, the `comment` is not printed (by `print` or `print.default`).

Assigning `NULL` or a zero-length character vector removes the comment.

Usage

```
comment(x)
comment(x) <- value
```

Arguments

`x`	any R object.
`value`	a `character` vector, or `NULL`.

See Also

`attributes` and `attr` for other attributes.

Examples

```
x <- matrix(1:12, 3, 4)
comment(x) <- c("This is my very important data from experiment #0234",
                "Jun 5, 1998")
x
comment(x)
```

Comparison *Relational Operators*

Description

Binary operators which allow the comparison of values in atomic vectors.

Usage

```
x < y
x > y
x <= y
x >= y
x == y
x != y
```

Arguments

x, y	atomic vectors, symbols, calls, or other objects for which methods have been written.

Details

The binary comparison operators are generic functions: methods can be written for them individually or via the Ops group generic function. (See Ops for how dispatch is computed.)

Comparison of strings in character vectors is lexicographic within the strings using the collating sequence of the locale in use: see locales. The collating sequence of locales such as 'en_US' is normally different from 'C' (which should use ASCII) and can be surprising. Beware of making *any* assumptions about the collation order: e.g. in Estonian Z comes between S and T, and collation is not necessarily character-by-character – in Danish aa sorts as a single letter, after z. In Welsh ng may or may not be a single sorting unit: if it is it follows g. Some platforms may not respect the locale and always sort in numerical order of the bytes in an 8-bit locale, or in Unicode code-point order for a UTF-8 locale (and may not sort in the same order for the same language in different character sets). Collation of non-letters (spaces, punctuation signs, hyphens, fractions and so on) is even more problematic.

Character strings can be compared with different marked encodings (see Encoding): they are translated to UTF-8 before comparison.

Raw vectors should not really be considered to have an order, but the numeric order of the byte representation is used.

At least one of x and y must be an atomic vector, but if the other is a list R attempts to coerce it to the type of the atomic vector: this will succeed if the list is made up of elements of length one that can be coerced to the correct type.

If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw.

Missing values (NA) and NaN values are regarded as non-comparable even to themselves, so comparisons involving them will always result in NA. Missing values can also result when character strings are compared and one is not valid in the current collation locale.

Language objects such as symbols and calls can only be used as operands for == and !=; the other comparisons signal an error when one of the operands is a language object. Currently language objects are deparsed to character strings before comparison. This can be inefficient and may not be what is really wanted. For equality comparisons identical is usually a better choice.

Value

A logical vector indicating the result of the element by element comparison. The elements of shorter vectors are recycled as necessary.

Objects such as arrays or time-series can be compared this way provided they are conformable.

S4 methods

These operators are members of the S4 Compare group generic, and so methods can be written for them individually as well as for the group generic (or the Ops group generic), with arguments c(e1, e2).

Note

Do not use == and != for tests, such as in `if` expressions, where you must get a single `TRUE` or `FALSE`. Unless you are absolutely sure that nothing unusual can happen, you should use the `identical` function instead.

For numerical and complex values, remember == and != do not allow for the finite representation of fractions, nor for rounding error. Using `all.equal` with `identical` or `isTRUE` is almost always preferable; see the examples. (This also applies to the other comparison operators.)

These operators are sometimes called as functions as e.g. `` `<`(x, y) ``: see the description of how argument-matching is done in `Ops`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Collation of character strings is a complex topic. For an introduction see https://en.wikipedia.org/wiki/Collating_sequence. The *Unicode Collation Algorithm* (https://unicode.org/reports/tr10/) is likely to be increasingly influential. Where available R by default makes use of ICU (https://icu.unicode.org/) for collation (except in a C locale).

See Also

`Logic` on how to *combine* results of comparisons, i.e., logical vectors.

`factor` for the behaviour with factor arguments.

`Syntax` for operator precedence.

`capabilities` for whether ICU is available, and `icuSetCollate` to tune the string collation algorithm when it is.

Examples

```
x <- stats::rnorm(20)
x < 1
x[x > 0]

x1 <- 0.5 - 0.3
x2 <- 0.3 - 0.1
x1 == x2                   # FALSE on most machines
isTRUE(all.equal(x1, x2))  # TRUE everywhere


# range of most 8-bit charsets, as well as of Latin-1 in Unicode
z <- c(32:126, 160:255)
x <- if(l10n_info()$MBCS) {
    intToUtf8(z, multiple = TRUE)
} else rawToChar(as.raw(z), multiple = TRUE)
## by number
writeLines(strwrap(paste(x, collapse=" "), width = 60))
## by locale collation
writeLines(strwrap(paste(sort(x), collapse=" "), width = 60))
```

complex *Complex Numbers and Basic Functionality*

Description

Basic functions which support complex arithmetic in R, in addition to the arithmetic operators +, -, *, /, and ^.

Usage

```
complex(length.out = 0, real = numeric(), imaginary = numeric(),
        modulus = 1, argument = 0)
as.complex(x, ...)
is.complex(x)

Re(z)
Im(z)
Mod(z)
Arg(z)
Conj(z)
```

Arguments

length.out	numeric. Desired length of the output vector, inputs being recycled as needed.
real	numeric vector.
imaginary	numeric vector.
modulus	numeric vector.
argument	numeric vector.
x	an object, probably of mode complex.
z	an object of mode complex, or one of a class for which a methods has been defined.
...	further arguments passed to or from other methods.

Details

Complex vectors can be created with complex. The vector can be specified either by giving its length, its real and imaginary parts, or modulus and argument. (Giving just the length generates a vector of complex zeroes.)

as.complex attempts to coerce its argument to be of complex type: like as.vector it strips attributes including names. Since R version 4.4.0, as.complex(x) for "number-like" x, i.e., types "logical", "integer", and "double", will always keep imaginary part zero, now also for NA's. Up to R versions 3.2.x, all forms of NA and NaN were coerced to a complex NA, i.e., the NA_complex_ constant, for which both the real and imaginary parts are NA. Since R 3.3.0, typically only objects which are NA in parts are coerced to complex NA, but others with NaN parts, are *not*. As a consequence, complex arithmetic where only NaN's (but no NA's) are involved typically will *not* give

complex NA but complex numbers with real or imaginary parts of NaN. All of these many different complex numbers fulfill is.na(.) but only one of them is identical to NA_complex_.

Note that is.complex and is.numeric are never both TRUE.

The functions Re, Im, Mod, Arg and Conj have their usual interpretation as returning the real part, imaginary part, modulus, argument and complex conjugate for complex values. The modulus and argument are also called the *polar coordinates*. If $z = x + iy$ with real x and y, for $r = Mod(z) = \sqrt{x^2 + y^2}$, and $\phi = Arg(z)$, $x = r\cos(\phi)$ and $y = r\sin(\phi)$. They are all internal generic primitive functions: methods can be defined for them individually or *via* the Complex group generic.

In addition to the arithmetic operators (see Arithmetic) +, -, *, /, and ^, the elementary trigonometric, logarithmic, exponential, square root and hyperbolic functions are implemented for complex values.

Matrix multiplications (%*%, crossprod, tcrossprod) are also defined for complex matrices (matrix), and so are solve, eigen or svd.

Internally, complex numbers are stored as a pair of double precision numbers, either or both of which can be NaN (including NA, see NA_complex_ and above) or plus or minus infinity.

S4 methods

as.complex is primitive and can have S4 methods set.

Re, Im, Mod, Arg and Conj constitute the S4 group generic Complex and so S4 methods can be set for them individually or via the group generic.

Note

Operations and functions involving complex NaN mostly rely on the C library's handling of 'double complex' arithmetic, which typically returns complex(re=NaN, im=NaN) (but we have not seen a guarantee for that). For + and -, R's own handling works strictly "coordinate wise".

Operations involving complex NA, i.e., NA_complex_, return NA_complex_.

Only since R version 4.4.0, as.complex("1i") gives 1i, it returned NA_complex_ with a warning, previously.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

Arithmetic; polyroot finds all n complex roots of a polynomial of degree n.

Examples

```
require(graphics)

0i ^ (-3:3)

matrix(1i^ (-6:5), nrow = 4) #- all columns are the same
```

```
0 ^ 1i # a complex NaN

## create a complex normal vector
z <- complex(real = stats::rnorm(100), imaginary = stats::rnorm(100))
## or also (less efficiently):
z2 <- 1:2 + 1i*(8:9)

## The Arg(.) is an angle:
zz <- (rep(1:4, length.out = 9) + 1i*(9:1))/10
zz.shift <- complex(modulus = Mod(zz), argument = Arg(zz) + pi)
plot(zz, xlim = c(-1,1), ylim = c(-1,1), col = "red", asp = 1,
     main = expression(paste("Rotation by ","  ", pi == 180^o)))
abline(h = 0, v = 0, col = "blue", lty = 3)
points(zz.shift, col = "orange")

## as.complex(<some NA>): numbers keep Im = 0:
stopifnot(identical(as.complex(NA_real_), NA_real_ + 0i)) # has always been true
NAs <- vapply(list(NA, NA_integer_, NA_real_, NA_character_, NA_complex_),
              as.complex, 0+0i)
stopifnot(is.na(NAs), is.na(Re(NAs))) # has always been true
showC <- function(z) noquote(paste0("(", Re(z), ",", Im(z), ")"))
showC(NAs)
Im(NAs) # [0 0 0 NA NA]  \ in R <= 4.3.x was [NA NA 0 NA NA]
stopifnot(Im(NAs)[1:3] == 0)


## The exact result of this *depends* on the platform, compiler, math-library:
(NpNA <- NaN + NA_complex_) ; str(NpNA) # *behaves* as 'cplx NA' ..
stopifnot(is.na(NpNA), is.na(NA_complex_), is.na(Re(NA_complex_)), is.na(Im(NA_complex_)))
showC(NpNA)# but does not always show '(NaN,NA)'
## and this is not TRUE everywhere:
identical(NpNA, NA_complex_)
showC(NA_complex_) # always == (NA,NA)
```

---

conditions *Condition Handling and Recovery*

Description

These functions provide a mechanism for handling unusual conditions, including errors and warnings.

Usage

```
tryCatch(expr, ..., finally)
withCallingHandlers(expr, ...)
globalCallingHandlers(...)

signalCondition(cond)
```

```
simpleCondition(message, call = NULL)
simpleError    (message, call = NULL)
simpleWarning  (message, call = NULL)
simpleMessage  (message, call = NULL)

errorCondition(message, ..., class = NULL, call = NULL)
warningCondition(message, ..., class = NULL, call = NULL)

## S3 method for class 'condition'
as.character(x, ...)
## S3 method for class 'error'
as.character(x, ...)
## S3 method for class 'condition'
print(x, ...)
## S3 method for class 'restart'
print(x, ...)

conditionCall(c)
## S3 method for class 'condition'
conditionCall(c)
conditionMessage(c)
## S3 method for class 'condition'
conditionMessage(c)

withRestarts(expr, ...)

computeRestarts(cond = NULL)
findRestart(name, cond = NULL)
invokeRestart(r, ...)
tryInvokeRestart(r, ...)
invokeRestartInteractively(r)

isRestart(x)
restartDescription(r)
restartFormals(r)

suspendInterrupts(expr)
allowInterrupts(expr)

.signalSimpleWarning(msg, call)
.handleSimpleError(h, msg, call)
.tryResumeInterrupt()
```

Arguments

c	a condition object.
call	call expression.
cond	a condition object.

expr	expression to be evaluated.
finally	expression to be evaluated before returning or exiting.
h	function.
message	character string.
msg	character string.
name	character string naming a restart.
r	restart object.
x	object.
class	character string naming a condition class.
...	additional arguments; see details below.

Details

The condition system provides a mechanism for signaling and handling unusual conditions, including errors and warnings. Conditions are represented as objects that contain information about the condition that occurred, such as a message and the call in which the condition occurred. Currently conditions are S3-style objects, though this may eventually change.

Conditions are objects inheriting from the abstract class `condition`. Errors and warnings are objects inheriting from the abstract subclasses `error` and `warning`. The class `simpleError` is the class used by `stop` and all internal error signals. Similarly, `simpleWarning` is used by `warning`, and `simpleMessage` is used by `message`. The constructors by the same names take a string describing the condition as argument and an optional call. The functions `conditionMessage` and `conditionCall` are generic functions that return the message and call of a condition.

The function `errorCondition` can be used to construct error conditions of a particular class with additional fields specified as the `...` argument. `warningCondition` is analogous for warnings.

Conditions are signaled by `signalCondition`. In addition, the `stop` and `warning` functions have been modified to also accept condition arguments.

The function `tryCatch` evaluates its expression argument in a context where the handlers provided in the `...` argument are available. The `finally` expression is then evaluated in the context in which `tryCatch` was called; that is, the handlers supplied to the current `tryCatch` call are not active when the `finally` expression is evaluated.

Handlers provided in the `...` argument to `tryCatch` are established for the duration of the evaluation of `expr`. If no condition is signaled when evaluating `expr` then `tryCatch` returns the value of the expression.

If a condition is signaled while evaluating `expr` then established handlers are checked, starting with the most recently established ones, for one matching the class of the condition. When several handlers are supplied in a single `tryCatch` then the first one is considered more recent than the second. If a handler is found then control is transferred to the `tryCatch` call that established the handler, the handler found and all more recent handlers are disestablished, the handler is called with the condition as its argument, and the result returned by the handler is returned as the value of the `tryCatch` call.

Calling handlers are established by `withCallingHandlers`. If a condition is signaled and the applicable handler is a calling handler, then the handler is called by `signalCondition` in the context

where the condition was signaled but with the available handlers restricted to those below the handler called in the handler stack. If the handler returns, then the next handler is tried; once the last handler has been tried, `signalCondition` returns `NULL`.

`globalCallingHandlers` establishes calling handlers globally. These handlers are only called as a last resort, after the other handlers dynamically registered with `withCallingHandlers` have been invoked. They are called before the `error` global option (which is the legacy interface for global handling of errors). Registering the same handler multiple times moves that handler on top of the stack, which ensures that it is called first. Global handlers are a good place to define a general purpose logger (for instance saving the last error object in the global workspace) or a general recovery strategy (e.g. installing missing packages via the `retry_loadNamespace` restart).

Like `withCallingHandlers` and `tryCatch`, `globalCallingHandlers` takes named handlers. Unlike these functions, it also has an `options`-like interface: you can establish handlers by passing a single list of named handlers. To unregister all global handlers, supply a single 'NULL'. The list of deleted handlers is returned invisibly. Finally, calling `globalCallingHandlers` without arguments returns the list of currently established handlers, visibly.

User interrupts signal a condition of class `interrupt` that inherits directly from class `condition` before executing the default interrupt action.

Restarts are used for establishing recovery protocols. They can be established using `withRestarts`. One pre-established restart is an `abort` restart that represents a jump to top level.

`findRestart` and `computeRestarts` find the available restarts. `findRestart` returns the most recently established restart of the specified name. `computeRestarts` returns a list of all restarts. Both can be given a condition argument and will then ignore restarts that do not apply to the condition.

`invokeRestart` transfers control to the point where the specified restart was established and calls the restart's handler with the arguments, if any, given as additional arguments to `invokeRestart`. The restart argument to `invokeRestart` can be a character string, in which case `findRestart` is used to find the restart. If no restart is found, an error is thrown.

`tryInvokeRestart` is a variant of `invokeRestart` that returns silently when the restart cannot be found with `findRestart`. Because a condition of a given class might be signalled with arbitrary protocols (error, warning, etc), it is recommended to use this permissive variant whenever you are handling conditions signalled from a foreign context. For instance, invocation of a `"muffleWarning"` restart should be optional because the warning might have been signalled by the user or from a different package with the `stop` or `message` protocols. Only use `invokeRestart` when you have control of the signalling context, or when it is a logical error if the restart is not available.

New restarts for `withRestarts` can be specified in several ways. The simplest is in `name = function` form where the function is the handler to call when the restart is invoked. Another simple variant is as `name = string` where the string is stored in the `description` field of the restart object returned by `findRestart`; in this case the handler ignores its arguments and returns `NULL`. The most flexible form of a restart specification is as a list that can include several fields, including `handler`, `description`, and `test`. The `test` field should contain a function of one argument, a condition, that returns `TRUE` if the restart applies to the condition and `FALSE` if it does not; the default function returns `TRUE` for all conditions.

One additional field that can be specified for a restart is `interactive`. This should be a function of no arguments that returns a list of arguments to pass to the restart handler. The list could be obtained by interacting with the user if necessary. The function `invokeRestartInteractively` calls this

function to obtain the arguments to use when invoking the restart. The default `interactive` method queries the user for values for the formal arguments of the handler function.

Interrupts can be suspended while evaluating an expression using `suspendInterrupts`. Subexpression can be evaluated with interrupts enabled using `allowInterrupts`. These functions can be used to make sure cleanup handlers cannot be interrupted.

`.signalSimpleWarning`, `.handleSimpleError`, and `.tryResumeInterrupt` are used internally and should not be called directly.

References

The `tryCatch` mechanism is similar to Java error handling. Calling handlers are based on Common Lisp and Dylan. Restarts are based on the Common Lisp restart mechanism.

See Also

`stop` and `warning` signal conditions, and `try` is essentially a simplified version of `tryCatch`. `assertCondition` in package **tools** *tests* that conditions are signalled and works with several of the above handlers.

Examples

```
tryCatch(1, finally = print("Hello"))
e <- simpleError("test error")
## Not run:
 stop(e)
 tryCatch(stop(e), finally = print("Hello"))
 tryCatch(stop("fred"), finally = print("Hello"))

## End(Not run)
tryCatch(stop(e), error = function(e) e, finally = print("Hello"))
tryCatch(stop("fred"),  error = function(e) e, finally = print("Hello"))
withCallingHandlers({ warning("A"); 1+2 }, warning = function(w) {})
## Not run:
 { withRestarts(stop("A"), abort = function() {}); 1 }

## End(Not run)
withRestarts(invokeRestart("foo", 1, 2), foo = function(x, y) {x + y})

##--> More examples are part of
##-->   demo(error.catching)
```

---

conflicts *Search for Masked Objects on the Search Path*

Description

`conflicts` reports on objects that exist with the same name in two or more places on the `search` path, usually because an object in the user's workspace or a package is masking a system object of the same name. This helps discover unintentional masking.

Usage

```
conflicts(where = search(), detail = FALSE)
```

Arguments

where	A subset of the search path, by default the whole search path.
detail	If TRUE, give the masked or masking functions for all members of the search path.

Value

If detail = FALSE, a character vector of masked objects. If detail = TRUE, a list of character vectors giving the masked or masking objects in that member of the search path. Empty vectors are omitted.

Examples

```
lm <- 1:3
conflicts(, TRUE)
## gives something like
# $.GlobalEnv
# [1] "lm"
#
# $package:base
# [1] "lm"

## Remove things from your "workspace" that mask others:
remove(list = conflicts(detail = TRUE)$.GlobalEnv)
```

---

connections *Functions to Manipulate Connections (Files, URLs, ...)*

Description

Functions to create, open and close connections, i.e., "generalized files", such as possibly compressed files, URLs, pipes, etc.

Usage

```
file(description = "", open = "", blocking = TRUE,
     encoding = getOption("encoding"), raw = FALSE,
     method = getOption("url.method", "default"))

url(description, open = "", blocking = TRUE,
    encoding = getOption("encoding"),
    method = getOption("url.method", "default"),
    headers = NULL)
```

```
gzfile(description, open = "", encoding = getOption("encoding"),
       compression = 6)

bzfile(description, open = "", encoding = getOption("encoding"),
       compression = 9)

xzfile(description, open = "", encoding = getOption("encoding"),
       compression = 6)

unz(description, filename, open = "", encoding = getOption("encoding"))

pipe(description, open = "", encoding = getOption("encoding"))

fifo(description, open = "", blocking = FALSE,
     encoding = getOption("encoding"))

socketConnection(host = "localhost", port, server = FALSE,
                 blocking = FALSE, open = "a+",
                 encoding = getOption("encoding"),
                 timeout = getOption("timeout"),
                 options = getOption("socketOptions"))

serverSocket(port)

socketAccept(socket, blocking = FALSE, open = "a+",
             encoding = getOption("encoding"),
             timeout = getOption("timeout"),
             options = getOption("socketOptions"))

open(con, ...)
## S3 method for class 'connection'
open(con, open = "r", blocking = TRUE, ...)

close(con, ...)
## S3 method for class 'connection'
close(con, type = "rw", ...)

flush(con)

isOpen(con, rw = "")
isIncomplete(con)

socketTimeout(socket, timeout = -1)
```

Arguments

description	character string. A description of the connection: see 'Details'.
open	character string. A description of how to open the connection (if it should be

	opened initially). See section 'Modes' for possible values.
`blocking`	logical. See the 'Blocking' section.
`encoding`	the name of the encoding to be assumed. See the 'Encoding' section.
`raw`	logical. If true, a 'raw' interface is used which will be more suitable for arguments which are not regular files, e.g. character devices. This suppresses the check for a compressed file when opening for text-mode reading, and asserts that the 'file' may not be seekable.
`method`	character string, partially matched to `c("default", "internal", "wininet", "libcurl")`: see 'Details'.
`headers`	named character vector of HTTP headers to use in HTTP requests. It is ignored for non-HTTP URLs. The `User-Agent` header, coming from the `HTTPUserAgent` option (see `options`) is used as the first header, automatically.
`compression`	integer in 0–9. The amount of compression to be applied when writing, from none to maximal available. For `xzfile` can also be negative: see the 'Compression' section.
`timeout`	numeric: the timeout (in seconds) to be used for this connection. Beware that some OSes may treat very large values as zero: however the POSIX standard requires values up to 31 days to be supported.
`options`	optional character vector with options. Currently only `"no-delay"` is supported on TCP sockets.
`filename`	a filename within a zip file.
`host`	character string. Host name for the port.
`port`	integer. The TCP port number.
`server`	logical. Should the socket be a client or a server?
`socket`	a server socket listening for connections.
`con`	a connection.
`type`	character string. Currently ignored.
`rw`	character string. Empty or `"read"` or `"write"`, partial matches allowed.
`...`	arguments passed to or from other methods.

Details

The first eleven functions create connections. By default the connection is not opened (except for a socket connection created by `socketConnection` or `socketAccept` and for server socket connection created by `serverSocket`), but may be opened by setting a non-empty value of argument `open`.

For `file` the description is a path to the file to be opened (when tilde expansion is done) or a complete URL (when it is the same as calling `url`), or `""` (the default) or `"clipboard"` (see the 'Clipboard' section). Use `"stdin"` to refer to the C-level 'standard input' of the process (which need not be connected to anything in a console or embedded version of R, and is not in `RGui` on Windows). See also `stdin()` for the subtly different R-level concept of `stdin`. See `nullfile()` for a platform-independent way to get filename of the null device.

For `url` the description is a complete URL including scheme (such as '`http://`', '`https://`', '`ftp://`' or '`file://`'). Method `"internal"` is that available since connections were introduced but now mainly defunct. Method `"wininet"` is only available on Windows (it uses the WinINet functions of that OS) and method `"libcurl"` (using the library of that name: https://curl.se/libcurl/) is nowadays required but was optional on Windows before R 4.2.0. Method `"default"` currently uses method `"internal"` for '`file://`' URLs and `"libcurl"` for all others. Which methods support which schemes has varied by R version – currently `"internal"` supports only '`file://`'; `"wininet"` supports '`file://`', '`http://`' and '`https://`'. Proxies can be specified: see `download.file`.

For `gzfile` the description is the path to a file compressed by `gzip`: it can also open for reading uncompressed files and those compressed by `bzip2`, `xz` or `lzma`.

For `bzfile` the description is the path to a file compressed by `bzip2`.

For `xzfile` the description is the path to a file compressed by `xz` (https://en.wikipedia.org/wiki/Xz) or (for reading only) `lzma` (https://en.wikipedia.org/wiki/LZMA).

`unz` reads (only) single files within zip files, in binary mode. The description is the full path to the zip file, with '`.zip`' extension if required.

For `pipe` the description is the command line to be piped to or from. This is run in a shell, on Windows that specified by the `COMSPEC` environment variable.

For `fifo` the description is the path of the fifo. (Support for `fifo` connections is optional but they are available on most Unix platforms and on Windows.)

The intention is that `file` and `gzfile` can be used generally for text input (from files, '`http://`' and '`https://`' URLs) and binary input respectively.

`open`, `close` and `seek` are generic functions: the following applies to the methods relevant to connections.

`open` opens a connection. In general functions using connections will open them if they are not open, but then close them again, so to leave a connection open call `open` explicitly.

`close` closes and destroys a connection. This will happen automatically in due course (with a warning) if there is no longer an R object referring to the connection.

`flush` flushes the output stream of a connection open for write/append (where implemented, currently for file and clipboard connections, `stdout` and `stderr`).

If for a `file` or (on most platforms) a `fifo` connection the description is `""`, the file/fifo is immediately opened (in `"w+"` mode unless `open = "w+b"` is specified) and unlinked from the file system. This provides a temporary file/fifo to write to and then read from.

`socketConnection(server=TRUE)` creates a new temporary server socket listening on the given port. As soon as a new socket connection is accepted on that port, the server socket is automatically closed. `serverSocket` creates a listening server socket which can be used for accepting multiple socket connections by `socketAccept`. To stop listening for new connections, a server socket needs to be closed explicitly by `close`.

`socketConnection` and `socketAccept` support setting of socket-specific options. Currently only `"no-delay"` is implemented which enables the `TCP_NODELAY` socket option, causing the socket to flush send buffers immediately (instead of waiting to collect all output before sending). This option is useful for protocols that need fast request/response turn-around times.

`socketTimeout` sets connection timeout of a socket connection. A negative `timeout` can be given to query the old value.

Value

file, pipe, fifo, url, gzfile, bzfile, xzfile, unz, socketConnection, socketAccept and serverSocket return a connection object which inherits from class "connection" and has a first more specific class.

open and flush return NULL, invisibly.

close returns either NULL or an integer status, invisibly. The status is from when the connection was last closed and is available only for some types of connections (e.g., pipes, files and fifos): typically zero values indicate success. Negative values will result in a warning; if writing, these may indicate write failures and should not be ignored. Connections should be closed explicitly when finished with to avoid wasting resources and to reduce the risk that some buffered data in output connections would be lost (see on.exit() for how to run code also in case of error).

isOpen returns a logical value, whether the connection is currently open.

isIncomplete returns a logical value, whether the last read attempt from a non-blocking connection provided no data (currently no data from a socket or an unterminated line in readLines), or for an output text connection whether there is unflushed output. See example below.

socketTimeout returns the old timeout value of a socket connection.

URLs

url and file support URL schemes 'file://', 'http://', 'https://' and 'ftp://'.

method = "libcurl" allows more schemes: exactly which schemes is platform-dependent (see libcurlVersion), but all platforms will support 'https://' and most platforms will support 'ftps://'.

Support for the 'ftp://' scheme by the "internal" method was deprecated in R 4.1.1 and removed in R 4.2.0.

Most methods do not percent-encode special characters such as spaces in 'http://' URLs (see URLencode), but it seems the "wininet" method does.

A note on 'file://' URLs (which are handled by the same internal code irrespective of argument method). The most general form (from RFC1738) is 'file://host/path/to/file', but R only accepts the form with an empty host field referring to the local machine.

On a Unix-alike, this is then 'file:///path/to/file', where 'path/to/file' is relative to '/'. So although the third slash is strictly part of the specification not part of the path, this can be regarded as a way to specify the file '/path/to/file'. It is not possible to specify a relative path using a file URL.

In this form the path is relative to the root of the filesystem, not a Windows concept. The standard form on Windows is 'file:///d:/R/repos': for compatibility with earlier versions of R and Unix versions, any other form is parsed as R as 'file://' plus path_to_file. Also, backslashes are accepted within the path even though RFC1738 does not allow them.

No attempt is made to decode a percent-encoded 'file:' URL: call URLdecode if necessary.

All the methods attempt to follow redirected HTTP and HTTPS URLs.

Server-side cached data is always accepted.

Function download.file and several contributed packages provide more comprehensive facilities to download from URLs.

Modes

Possible values for the argument open are

"r" **or** "rt" Open for reading in text mode.

"w" **or** "wt" Open for writing in text mode.

"a" **or** "at" Open for appending in text mode.

"rb" Open for reading in binary mode.

"wb" Open for writing in binary mode.

"ab" Open for appending in binary mode.

"r+", "r+b" Open for reading and writing.

"w+", "w+b" Open for reading and writing, truncating file initially.

"a+", "a+b" Open for reading and appending.

Not all modes are applicable to all connections: for example URLs can only be opened for reading. Only file and socket connections can be opened for both reading and writing. An unsupported mode is usually silently substituted.

If a file or fifo is created on a Unix-alike, its permissions will be the maximal allowed by the current setting of umask (see Sys.umask).

For many connections there is little or no difference between text and binary modes. For file-like connections on Windows, translation of line endings (between LF and CRLF) is done in text mode only (but text read operations on connections such as readLines, scan and source work for any form of line ending). Various R operations are possible in only one of the modes: for example pushBack is text-oriented and is only allowed on connections open for reading in text mode, and binary operations such as readBin, load and save can only be done on binary-mode connections.

The mode of a connection is determined when actually opened, which is deferred if open = "" is given (the default for all but socket connections). An explicit call to open can specify the mode, but otherwise the mode will be "r". (gzfile, bzfile and xzfile connections are exceptions, as the compressed file always has to be opened in binary mode and no conversion of line-endings is done even on Windows, so the default mode is interpreted as "rb".) Most operations that need write access or text-only or binary-only mode will override the default mode of a non-yet-open connection.

Append modes need to be considered carefully for compressed-file connections. They do **not** produce a single compressed stream on the file, but rather append a new compressed stream to the file. Readers may or may not read beyond end of the first stream: currently R does so for gzfile, bzfile and xzfile connections.

Compression

R supports gzip, bzip2 and xz compression (also read-only support for its precursor, lzma compression).

For reading, the type of compression (if any) can be determined from the first few bytes of the file. Thus for file(raw = FALSE) connections, if open is "", "r" or "rt" the connection can read any of the compressed file types as well as uncompressed files. (Using "rb" will allow compressed files to be read byte-by-byte.) Similarly, gzfile connections can read any of the forms of compression and uncompressed files in any read mode.

(The type of compression is determined when the connection is created if open is unspecified and a file of that name exists. If the intention is to open the connection to write a file with a *different* form of compression under that name, specify open = "w" when the connection is created or unlink the file before creating the connection.)

For write-mode connections, compress specifies how hard the compressor works to minimize the file size, and higher values need more CPU time and more working memory (up to ca 800Mb for xzfile(compress = 9)). For xzfile negative values of compress correspond to adding the xz argument '-e': this takes more time (double?) to compress but may achieve (slightly) better compression. The default (6) has good compression and modest (100Mb memory) usage: but if you are using xz compression you are probably looking for high compression.

Choosing the type of compression involves tradeoffs: gzip, bzip2 and xz are successively less widely supported, need more resources for both compression and decompression, and achieve more compression (although individual files may buck the general trend). Typical experience is that bzip2 compression is 15% better on text files than gzip compression, and xz with maximal compression 30% better. The experience with R save files is similar, but on some large '.rda' files xz compression is much better than the other two. With current computers decompression times even with compress = 9 are typically modest and reading compressed files is usually faster than uncompressed ones because of the reduction in disc activity.

Encoding

The encoding of the input/output stream of a connection can be specified by name in the same way as it would be given to iconv: see that help page for how to find out what encoding names are recognized on your platform. Additionally, "" and "native.enc" both mean the 'native' encoding, that is the internal encoding of the current locale and hence no translation is done.

When writing to a text connection, the connections code always assumes its input is in native encoding, so e.g. writeLines has to convert text to native encoding. The native encoding is UTF-8 on most systems (since R 4.2 also on recent Windows) and can represent all characters. writeLines does not do the conversion when useBytes=TRUE (for expert use only, only useful on systems with native encoding other than UTF-8), but the connections code still behaves as if the text was in native encoding, so any attempt to convert encoding (encoding argument other than "" and "native.enc") in connections will produce incorrect results.

When reading from a text connection, the connections code re-encodes the input to native encoding (from the encoding given by the encoding argument). On systems where UTF-8 is not the native encoding, one can read text not representable in the native encoding using readLines and scan by providing them with an unopened connection that has been created with the encoding argument specifying the input encoding. readLines and scan would then instruct the connections code to convert the text to UTF-8 (instead of native encoding) and they will return it marked (aka declared, see Encoding) as "UTF-8". Finally and for expert use only, one may disable re-encoding of input by specifying "" or "native.enc" as encoding for the connection, but then mark the text as being "UTF-8" or "latin1" via the encoding argument of readLines and scan.

Re-encoding only works for connections in text mode: reading from a connection with re-encoding specified in binary mode will read the stream of bytes, but mixing text and binary mode reads (e.g., mixing calls to readLines and readChar) is likely to lead to incorrect results.

The encodings "UCS-2LE" and "UTF-16LE" are treated specially, as they are appropriate values for Windows 'Unicode' text files. If the first two bytes are the Byte Order Mark 0xFEFF then these are removed as some implementations of iconv do not accept BOMs. Note that whereas

most implementations will handle BOMs using encoding "UCS-2" and choose the appropriate byte order, some (including earlier versions of glibc) will not. There is a subtle distinction between "UTF-16" and "UCS-2" (see https://en.wikipedia.org/wiki/UTF-16): the use of characters in the 'Supplementary Planes' which need surrogate pairs is very rare so "UCS-2LE" is an appropriate first choice (as it is more widely implemented).

The encoding "UTF-8-BOM" is accepted for reading and will remove a Byte Order Mark if present (which it often is for files and webpages generated by Microsoft applications). If a BOM is required (it is not recommended) when writing it should be written explicitly, e.g. by `writeChar("\ufeff", con, eos = NULL)` or `writeBin(as.raw(c(0xef, 0xbb, 0xbf)), binary_con)`

Encoding names "utf8", "mac" and "macroman" are not portable, and not supported on all current R platforms. "UTF-8" is portable and "macintosh" is the official (and most widely supported) name for 'Mac Roman'. (R maps "utf8" to "UTF-8" internally.)

Requesting a conversion that is not supported is an error, reported when the connection is opened. Exactly what happens when the requested translation cannot be done for invalid input is in general undocumented. On output the result is likely to be that up to the error, with a warning. On input, it will most likely be all or some of the input up to the error.

It may be possible to deduce the current native encoding from `Sys.getlocale("LC_CTYPE")`, but not all OSes record it.

Blocking

Whether or not the connection blocks can be specified for file, url (default yes), fifo and socket connections (default not).

In blocking mode, functions using the connection do not return to the R evaluator until the read/write is complete. In non-blocking mode, operations return as soon as possible, so on input they will return with whatever input is available (possibly none) and for output they will return whether or not the write succeeded.

The function `readLines` behaves differently in respect of incomplete last lines in the two modes: see its help page.

Even when a connection is in blocking mode, attempts are made to ensure that it does not block the event loop and hence the operation of GUI parts of R. These do not always succeed, and the whole R process will be blocked during a DNS lookup on Unix, for example.

Most blocking operations on HTTP/FTP URLs and on sockets are subject to the timeout set by `options("timeout")`. Note that this is a timeout for no response, not for the whole operation. The timeout is set at the time the connection is opened (more precisely, when the last connection of that type – 'http:', 'ftp:' or socket – was opened).

Fifos

Fifos default to non-blocking. That follows S version 4 and is probably most natural, but it does have some implications. In particular, opening a non-blocking fifo connection for writing (only) will fail unless some other process is reading on the fifo.

Opening a fifo for both reading and writing (in any mode: one can only append to fifos) connects both sides of the fifo to the R process, and provides an similar facility to `file()`.

Clipboard

`file` can be used with `description = "clipboard"` in mode `"r"` only. This reads the X11 primary selection (see https://specifications.freedesktop.org/clipboards-spec/clipboards-latest.txt), which can also be specified as `"X11_primary"` and the secondary selection as `"X11_secondary"`. On most systems the clipboard selection (that used by 'Copy' from an 'Edit' menu) can be specified as `"X11_clipboard"`.

When a clipboard is opened for reading, the contents are immediately copied to internal storage in the connection.

Unix users wishing to *write* to one of the X11 selections may be able to do so via `xclip` (https://github.com/astrand/xclip) or `xsel` (https://www.vergenet.net/~conrad/software/xsel/), for example by `pipe("xclip -i", "w")` for the primary selection.

macOS users can use `pipe("pbpaste")` and `pipe("pbcopy", "w")` to read from and write to that system's clipboard.

File paths

In most cases these are translated to the native encoding.

The exceptions are `file` and `pipe` on Windows, where a `description` which is marked as being in UTF-8 is passed to Windows as a 'wide' character string. This allows files with names not in the native encoding to be opened on file systems which use Unicode file names (such as NTFS but not FAT32).

'ftp://' URLs

Most modern browsers do not support such URLs, and 'https://' ones are much preferred for use in R.

It is intended that R will continue to allow such URLs for as long as `libcurl` does, but as they become rarer this is increasingly untested. What 'protocols' the version of `libcurl` being used supports can be seen by calling `libcurlVersion()`.

Number of connections

There is a limit on the number of connections which can be allocated (not necessarily open) at any one time. It is good practice to close connections when finished with, but if necessary garbage-collection will be invoked to close those connections without any R object referring to them.

The default limit is 128 (including the three terminal connections, `stdin`, `stdout` and `stderr`). This can be increased when R is started using the option '`--max-connections=N`', where the maximum allowed value is 4096.

However, many types of connections use other resources which are themselves limited. Notably on Unix, 'file descriptors' which by default are per-process limited: this limits the number of connections using files, pipes and fifos. (The default limit is 256 on macOS (and Solaris) but 1024 on Linux. The limit can be raised in the shell used to launch R, for example by `ulimit -n`.) File descriptors are used for many other purposes including dynamically loading DSO/DLLs (see `dyn.load`) which may use up to 60% of the limit.

Windows has a default limit of 512 open C file streams: these are used by at least `file`, `gzfile`, `bzfile`, `xzfile`, `pipe`, `url` and `unz` connections applied to files (rather than URLs).

Package **parallel**'s `makeCluster` uses socket connections to communicate with the worker processes, one per worker.

Note

R's connections are modelled on those in S version 4 (see Chambers, 1998). However R goes well beyond the S model, for example in output text connections and URL, compressed and socket connections. The default open mode in R is `"r"` except for socket connections. This differs from S, where it is the equivalent of `"r+"`, known as `"*"`.

On (historic) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 byte output limit on the length of a line for text output on `fifo`, `gzfile`, `bzfile` and `xzfile` connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.

Ripley, B. D. (2001). "Connections." *R News*, **1**(1), 16–7. https://www.r-project.org/doc/Rnews/Rnews_2001-1.pdf.

See Also

`textConnection`, `seek`, `showConnections`, `pushBack`.

Functions making direct use of connections are (text-mode) `readLines`, `writeLines`, `cat`, `sink`, `scan`, `parse`, `read.dcf`, `dput`, `dump` and (binary-mode) `readBin`, `readChar`, `writeBin`, `writeChar`, `load` and `save`.

`capabilities` to see if `fifo` connections are supported by this build of R.

`gzcon` to wrap `gzip` (de)compression around a connection.

`options` `HTTPUserAgent`, `internet.info` and `timeout` are used by some of the methods for URL connections.

`memCompress` for more ways to (de)compress and references on data compression.

`extSoftVersion` for the versions of the `zlib` (for `gzfile`), `bzip2` and `xz` libraries in use.

To flush output to the Windows and macOS consoles, see `flush.console`.

Examples

```
zzfil <- tempfile(fileext=".data")
zz <- file(zzfil, "w")  # open an output file connection
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
cat("One more line\n", file = zz)
close(zz)
readLines(zzfil)
unlink(zzfil)

zzfil <- tempfile(fileext=".gz")
zz <- gzfile(zzfil, "w")  # compressed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile(zzfil))
```

```
close(zz)
unlink(zzfil)
zz # an invalid connection

zzfil <- tempfile(fileext=".bz2")
zz <- bzfile(zzfil, "w")  # bzip2-ed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
zz # print() method: invalid connection
print(readLines(zz <- bzfile(zzfil)))
close(zz)
unlink(zzfil)

## An example of a file open for reading and writing
Tpath <- tempfile("test")
Tfile <- file(Tpath, "w+")
c(isOpen(Tfile, "r"), isOpen(Tfile, "w")) # both TRUE
cat("abc\ndef\n", file = Tfile)
readLines(Tfile)
seek(Tfile, 0, rw = "r") # reset to beginning
readLines(Tfile)
cat("ghi\n", file = Tfile)
readLines(Tfile)

Tfile # -> print() :  "valid" connection
close(Tfile)
Tfile # -> print() :  "invalid" connection
unlink(Tpath)

## We can do the same thing with an anonymous file.
Tfile <- file()
cat("abc\ndef\n", file = Tfile)
readLines(Tfile)
close(Tfile)

## Not run: ## fifo example -- may hang even with OS support for fifos
if(capabilities("fifo")) {
  zzfil <- tempfile(fileext="-fifo")
  zz <- fifo(zzfil, "w+")
  writeLines("abc", zz)
  print(readLines(zz))
  close(zz)
  unlink(zzfil)
}
## End(Not run)

## Unix examples of use of pipes

# read listing of current directory
readLines(pipe("ls -1"))

# remove trailing commas.  Suppose
```

```
## Not run: % cat data2_
450, 390, 467, 654,  30, 542, 334, 432, 421,
357, 497, 493, 550, 549, 467, 575, 578, 342,
446, 547, 534, 495, 979, 479
## End(Not run)
# Then read this by
scan(pipe("sed -e s/,$// data2_"), sep = ",")


# convert decimal point to comma in output: see also write.table
# both R strings and (probably) the shell need \ doubled
zzfil <- tempfile("outfile")
zz <- pipe(paste("sed s/\\\\./,/ >", zzfil), "w")
cat(format(round(stats::rnorm(48), 4)), fill = 70, file = zz)
close(zz)
file.show(zzfil, delete.file = TRUE)

## Not run:
## example for a machine running a finger daemon

con <- socketConnection(port = 79, blocking = TRUE)
writeLines(paste0(system("whoami", intern = TRUE), "\r"), con)
gsub(" *$", "", readLines(con))
close(con)

## End(Not run)

## Not run:
## Two R processes communicating via non-blocking sockets
# R process 1
con1 <- socketConnection(port = 6011, server = TRUE)
writeLines(LETTERS, con1)
close(con1)

# R process 2
con2 <- socketConnection(Sys.info()["nodename"], port = 6011)
# as non-blocking, may need to loop for input
readLines(con2)
while(isIncomplete(con2)) {
   Sys.sleep(1)
   z <- readLines(con2)
   if(length(z)) print(z)
}
close(con2)

## examples of use of encodings
# write a file in UTF-8
cat(x, file = (con <- file("foo", "w", encoding = "UTF-8"))); close(con)
# read a 'Windows Unicode' file
A <- read.table(con <- file("students", encoding = "UCS-2LE")); close(con)

## End(Not run)
```

`Constants` *Built-in Constants*

Description

Constants built into R.

Usage

```
LETTERS
letters
month.abb
month.name
pi
```

Details

R has a small number of built-in constants.

The following constants are available:

- `LETTERS`: the 26 upper-case letters of the Roman alphabet;
- `letters`: the 26 lower-case letters of the Roman alphabet;
- `month.abb`: the three-letter abbreviations for the English month names;
- `month.name`: the English names for the months of the year;
- `pi`: the ratio of the circumference of a circle to its diameter.

These are implemented as variables in the base namespace taking appropriate values.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`data`, `DateTimeClasses`.

`Quotes` for the parsing of character constants, `NumericConstants` for numeric constants.

Examples

```
## John Machin (ca 1706) computed pi to over 100 decimal places
## using the Taylor series expansion of the second term of
pi - 4*(4*atan(1/5) - atan(1/239))

## months in English
month.name
## months in your current locale
format(ISOdate(2000, 1:12, 1), "%B")
format(ISOdate(2000, 1:12, 1), "%b")
```

contributors *R Project Contributors*

Description

The R Who-is-who, describing who made significant contributions to the development of R.

Usage

```
contributors()
```

Control *Control Flow*

Description

These are the basic control-flow constructs of the R language. They function in much the same way as control statements in any Algol-like language. They are all reserved words.

Usage

```
if(cond) expr
if(cond) cons.expr  else  alt.expr

for(var in seq) expr
while(cond) expr
repeat expr
break
next

x %||% y
```

Arguments

`cond`	A length-one logical vector that is not `NA`. Other types are coerced to logical if possible, ignoring any class. (Conditions of length greater than one are an error.)
`var`	A syntactical name for a variable.
`seq`	An expression evaluating to a vector (including a list and an expression) or to a pairlist or `NULL`. A factor value will be coerced to a character vector. This can be a long vector.
`expr`, `cons.expr`, `alt.expr`, `x`, `y`	An *expression* in a formal sense. This is either a simple expression or a so-called *compound expression*, usually of the form `{ expr1 ; expr2 }`.

Details

`break` breaks out of a `for`, `while` or `repeat` loop; control is transferred to the first statement outside the inner-most loop. `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

Note that it is a common mistake to forget to put braces (`{ .. }`) around your statements, e.g., after `if(..)` or `for(....)`. In particular, you should not have a newline between `}` and `else` to avoid a syntax error in entering a `if ... else` construct at the keyboard or via `source`. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for `if` clauses.

The `seq` in a `for` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop. If `seq` has length zero the body of the loop is skipped. Otherwise the variable `var` is assigned in turn the value of each element of `seq`. You can assign to `var` within the body of the loop, but this will not affect the next iteration. When the loop terminates, `var` remains as a variable containing its latest value.

The null coalescing operator `%||%` is a simple 1-line function: `x %||% y` is an idiomatic way to call

```
if (is.null(x)) y else x
                          # or equivalently, of course,
if(!is.null(x)) x else y
```

Inspired by Ruby, it was first proposed by Hadley Wickham.

Value

`if` returns the value of the expression evaluated, or `NULL` invisibly if none was (which may happen if there is no `else`).

`for`, `while` and `repeat` return `NULL` invisibly. `for` sets `var` to the last used element of `seq`, or to `NULL` if it was of length zero.

`break` and `next` do not return a value as they transfer control within the loop.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Syntax` for the basic R syntax and operators, `Paren` for parentheses and braces.

`ifelse`, `switch` for other ways to control flow.

Examples

```
for(i in 1:5) print(1:i)
for(n in c(2,5,10,20,50)) {
   x <- stats::rnorm(n)
   cat(n, ": ", sum(x^2), "\n", sep = "")
}
f <- factor(sample(letters[1:5], 10, replace = TRUE))
for(i in unique(f)) print(i)
```

```
res <- {}
res %||% "alternative result"
x <- head(x) %||% stop("parsed, but *not* evaluated..")

res <- if(sum(x) > 7.5) mean(x) # may be NULL
res %||% "sum(x) <= 7.5"
```

---

copyright *Copyrights of Files Used to Build R*

Description

R is released under the 'GNU Public License': see `license` for details. The license describes your right to use R. Copyright is concerned with ownership of intellectual rights, and some of the software used has conditions that the copyright must be explicitly stated: see the 'Details' section. We are grateful to these people and other contributors (see `contributors`) for the ability to use their work.

Details

The file '*R_HOME*`/COPYRIGHTS`' lists the copyrights in full detail.

---

crossprod *Matrix Cross-Product*

Description

Given matrices `x` and `y` as arguments, return a matrix cross-product. This is formally equivalent to (but faster than) the call `t(x) %*% y` (`crossprod`) or `x %*% t(y)` (`tcrossprod`).

These are generic functions since R 4.4.0: methods can be written individually or via the `matOps` group generic function; it dispatches to S3 and S4 methods.

Usage

```
 crossprod(x, y = NULL, ...)
tcrossprod(x, y = NULL, ...)
```

Arguments

`x, y`	numeric or complex matrices (or vectors): `y = NULL` is taken to be the same matrix as `x`. Vectors are promoted to single-column or single-row matrices, depending on the context.
`...`	potential further arguments for methods.

Value

A double or complex matrix, with appropriate `dimnames` taken from `x` and `y`.

Note

When `x` or `y` are not matrices, they are treated as column or row matrices, but their `names` are usually **not** promoted to `dimnames`. Hence, currently, the last example has empty dimnames.

In the same situation, these matrix products (also `%*%`) are more flexible in promotion of vectors to row or column matrices, such that more cases are allowed, since R 3.2.0.

The propagation of `NaN`/`Inf` values, precision, and performance of matrix products can be controlled by `options("matprod")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`%*%` and outer product `%o%`.

Examples

```
(z <- crossprod(1:4))    # = sum(1 + 2^2 + 3^2 + 4^2)
drop(z)                  # scalar
x <- 1:4; names(x) <- letters[1:4]; x
tcrossprod(as.matrix(x)) # is
identical(tcrossprod(as.matrix(x)),
          crossprod(t(x)))
tcrossprod(x)            # no dimnames

m <- matrix(1:6, 2,3) ; v <- 1:3; v2 <- 2:1
stopifnot(identical(tcrossprod(v, m), v %*% t(m)),
          identical(tcrossprod(v, m), crossprod(v, t(m))),
          identical(crossprod(m, v2), t(m) %*% v2))
```

Cstack_info *Report Information on C Stack Size and Usage*

Description

Report information on the C stack size and usage (if available).

Usage

```
Cstack_info()
```

Details

On most platforms, C stack information is recorded when R is initialized and used for stack-checking. If this information is unavailable, the `size` will be returned as `NA`, and stack-checking is not performed.

The information on the stack base address is thought to be accurate on Windows, Linux (using `glibc`), macOS and FreeBSD but a heuristic is used on other platforms. Because this might be slightly inaccurate, the current usage could be estimated as negative. (The heuristic is not used on embedded uses of R on platforms where the stack base information is not thought to be accurate.)

The 'evaluation depth' is the number of nested R expressions currently under evaluation: this has a limit controlled by `options("expressions")`.

Value

An integer vector. This has named elements

`size`	The size of the stack (in bytes), or `NA` if unknown.
`current`	The estimated current usage (in bytes), possibly `NA`.
`direction`	`1` (stack grows down, the usual case) or `-1` (stack grows up).
`eval_depth`	The current evaluation depth (including two calls for the call to `Cstack_info`).

Examples

```
Cstack_info()
```

`cumsum` *Cumulative Sums, Products, and Extremes*

Description

Returns a vector whose elements are the cumulative sums, products, minima or maxima of the elements of the argument.

Usage

```
cumsum(x)
cumprod(x)
cummax(x)
cummin(x)
```

Arguments

`x`	a numeric or complex (not `cummin` or `cummax`) object, or an object that can be coerced to one of these.

Details

These are generic functions: methods can be defined for them individually or via the Math group generic.

Value

A vector of the same length and type as x (after coercion), except that cumprod returns a numeric vector for integer input (for consistency with *). Names are preserved.

An NA value in x causes the corresponding and following elements of the return value to be NA, as does integer overflow in cumsum (with a warning). In the complex case with NAs, these NA elements may have finite real or imaginary parts, notably for cumsum(), fulfilling the identity Im(cumsum(x)) ≡ cumsum(Im(x)).

S4 methods

cumsum and cumprod are S4 generic functions: methods can be defined for them individually or via the Math group generic. cummax and cummin are individually S4 generic functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (cumsum only.)

Examples

```
cumsum(1:10)
cumprod(1:10)
cummin(c(3:1, 2:0, 4:2))
cummax(c(3:1, 2:0, 4:2))
```

---

curlGetHeaders *Retrieve Headers from URLs*

Description

Retrieve the headers for a URL for a supported protocol such as 'http://', 'ftp://', 'https://' and 'ftps://'.

Usage

```
curlGetHeaders(url, redirect = TRUE, verify = TRUE,
               timeout = 0L, TLS = "")
```

Arguments

`url`	character string specifying the URL.
`redirect`	logical: should redirections be followed?
`verify`	logical: should certificates be verified as valid and applying to that host?
`timeout`	integer: the maximum time in seconds the request is allowed to take. Non-positive and invalid values are ignored (including the default). (Added in R 4.1.0.)
`TLS`	character: the minimum version of the TLS protocol to be used for '`https://`' URLs: the default (`""`) is no restriction beyond that of the underlying `libcurl` (usually 1.0). Other valid values are `"1.1"`, `"1.2"` (both for `libcurl` 7.34.0 and later) and `"1.3"` (7.52.0 and later), if supported by the underlying version of `libcurl` and the SSL library it uses.

Details

This reports what `curl -I -L` or `curl -I` would report. For a '`ftp://`' URL the 'headers' are a record of the conversation between client and server before data transfer.

Only 500 header lines will be reported: there is a limit of 20 redirections so this should suffice (and even 20 would indicate problems).

If argument `timeout` is not set to a positive integer this uses `getOption("timeout")` which defaults to 60 seconds. As the request cannot be interrupted you may want to consider a shorter value.

To see all the details of the interaction with the server(s) set `options(internet.info = 1)`.

HTTP[S] servers are allowed to refuse requests to read the headers and some do: this will result in a `status` of `405`.

For possible issues with secure URLs (especially on Windows) see `download.file`.

There is a security risk in not verifying certificates, but as only the headers are captured it is slight. Usually looking at the URL in a browser will reveal what the problem is (and it may well be machine-specific).

Value

A character vector with integer attribute `"status"` (the last-received 'status' code). If redirection occurs this will include the headers for all the URLs visited.

For the interpretation of 'status' codes see https://en.wikipedia.org/wiki/List_of_HTTP_status_codes and https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes. A successful FTP connection will usually have status 250, 257 or 350.

See Also

`capabilities("libcurl")` to see if this is supported. `libcurlVersion` for the version of `libcurl` in use.

`options` `HTTPUserAgent` and `timeout` are used.

Examples

```
## needs Internet access, results vary
curlGetHeaders("http://bugs.r-project.org")   ## this redirects to https://
## 2023-04: replaces slow and unreliable https://httpbin.org/status/404
curlGetHeaders("https://developer.R-project.org/inet-tests/not-found")
## returns status
```

---

cut *Convert Numeric to Factor*

Description

cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.

Usage

```
cut(x, ...)

## Default S3 method:
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, dig.lab = 3,
    ordered_result = FALSE, ...)
```

Arguments

x	a numeric vector which is to be converted to a factor by cutting.
breaks	either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut.
labels	labels for the levels of the resulting category. By default, labels are constructed using "(a,b]" interval notation. If labels = FALSE, simple integer codes are returned instead of a factor.
include.lowest	logical, indicating if an 'x[i]' equal to the lowest (or highest, for right = FALSE) 'breaks' value should be included.
right	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
dig.lab	integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers.
ordered_result	logical: should the result be an ordered factor?
...	further arguments passed to or from other methods.

Details

When `breaks` is specified as a single number, the range of the data is divided into `breaks` pieces of equal length, and then the outer limits are moved away by 0.1% of the range to ensure that the extreme values both fall within the break intervals. (If `x` is a constant vector, equal-length intervals are created, one of which includes the single value.)

If a `labels` parameter is specified, its values are used to name the factor levels. If none is specified, the factor level labels are constructed as `"(b1, b2]"`, `"(b2, b3]"` etc. for `right = TRUE` and as `"[b1, b2)"`, ...if `right = FALSE`. In this case, `dig.lab` indicates the minimum number of digits should be used in formatting the numbers `b1`, `b2`, ... A larger value (up to 12) will be used if needed to distinguish between any pair of endpoints: if this fails labels such as `"Range3"` will be used. Formatting is done by `formatC`.

The default method will sort a numeric vector of `breaks`, but other methods are not required to and `labels` will correspond to the intervals after sorting.

As from R 3.2.0, `getOption("OutDec")` is consulted when labels are constructed for `labels = NULL`.

Value

A `factor` is returned, unless `labels = FALSE` which results in an integer vector of level codes.

Values which fall outside the range of `breaks` are coded as `NA`, as are `NaN` and `NA` values.

Note

Instead of `table(cut(x, br))`, `hist(x, br, plot = FALSE)` is more efficient and less memory hungry. Instead of `cut(*, labels = FALSE)`, `findInterval()` is more efficient.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`split` for splitting a variable according to a group factor; `factor`, `tabulate`, `table`, `findInterval`.

`quantile` for ways of choosing breaks of roughly equal content (rather than length).

`.bincode` for a bare-bones version.

Examples

```
Z <- stats::rnorm(10000)
table(cut(Z, breaks = -6:6))
sum(table(cut(Z, breaks = -6:6, labels = FALSE)))
sum(graphics::hist(Z, breaks = -6:6, plot = FALSE)$counts)

cut(rep(1,5), 4) #-- dummy
tx0 <- c(9, 4, 6, 5, 3, 10, 5, 3, 5)
x <- rep(0:8, tx0)
```

```
stopifnot(table(x) == tx0)

table( cut(x, breaks = 8))
table( cut(x, breaks = 3*(-2:5)))
table( cut(x, breaks = 3*(-2:5), right = FALSE))

##--- some values OUTSIDE the breaks :
table(cx  <- cut(x, breaks = 2*(0:4)))
table(cxl <- cut(x, breaks = 2*(0:4), right = FALSE))
which(is.na(cx));  x[is.na(cx)]  #-- the first 9  values  0
which(is.na(cxl)); x[is.na(cxl)] #-- the last  5  values  8


## Label construction:
y <- stats::rnorm(100)
table(cut(y, breaks = pi/3*(-3:3)))
table(cut(y, breaks = pi/3*(-3:3), dig.lab = 4))

table(cut(y, breaks =  1*(-3:3), dig.lab = 4))
# extra digits don't "harm" here
table(cut(y, breaks =  1*(-3:3), right = FALSE))
#- the same, since no exact INT!

## sometimes the default dig.lab is not enough to be avoid confusion:
aaa <- c(1,2,3,4,5,2,3,4,5,6,7)
cut(aaa, 3)
cut(aaa, 3, dig.lab = 4, ordered_result = TRUE)

## one way to extract the breakpoints
labs <- levels(cut(aaa, 3))
cbind(lower = as.numeric( sub("\\((.+),.*", "\\1", labs) ),
      upper = as.numeric( sub("[^,]*,([^]]*)\\]", "\\1", labs) ))
```

---

cut.POSIXt *Convert a Date or Date-Time Object to a Factor*

Description

Method for cut applied to date-time objects.

Usage

```
## S3 method for class 'POSIXt'
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)

## S3 method for class 'Date'
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
breaks	a vector of cut points *or* number giving the number of intervals which x is to be cut into *or* an interval specification, one of "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year", optionally preceded by an integer and a space, or followed by "s". (For "Date" objects only interval specifications using "day", "week", "month", "quarter" and "year" are allowed.)
labels	labels for the levels of the resulting category. By default, labels are constructed from the left-hand end of the intervals (which are included for the default value of right). If labels = FALSE, simple integer codes are returned instead of a factor.
start.on.monday	logical. If breaks = "weeks", should the week start on Mondays or Sundays?
right, ...	arguments to be passed to or from other methods.

Details

Note that the default for right differs from the default method. Using include.lowest = TRUE will include both ends of the range of dates.

Using breaks = "quarter" will create intervals of 3 calendar months, with the intervals beginning on January 1, April 1, July 1 or October 1 (based upon min(x)) as appropriate.

A vector of breaks will be sorted before use: labels should correspond to the sorted vector.

Value

A factor is returned, unless labels = FALSE which returns the integer level codes.

Values which fall outside the range of breaks are coded as NA, as are and NA values.

See Also

seq.POSIXt, seq.Date, cut

Examples

```
## random dates in a 10-week period
cut(ISOdate(2001, 1, 1) + 70*86400*stats::runif(100), "weeks")
cut(as.Date("2001/1/1") + 70*stats::runif(100), "weeks")

# The standards all have midnight as the start of the day, but some
# people incorrectly interpret it at the end of the previous day ...
tm <- seq(as.POSIXct("2012-06-01 06:00"), by = "6 hours", length.out = 24)
aggregate(1:24, list(day = cut(tm, "days")), mean)
# and a version with midnight included in the previous day:
aggregate(1:24, list(day = cut(tm, "days", right = TRUE)), mean)
```

`data.class` *Object Classes*

Description

Determine the class of an arbitrary R object.

Usage

```
data.class(x)
```

Arguments

x	an R object.

Value

character string giving the *class* of x.

The class is the (first element) of the `class` attribute if this is non-NULL, or inferred from the object's `dim` attribute if this is non-NULL, or `mode(x)`.

Simply speaking, `data.class(x)` returns what is typically useful for method dispatching. (Or, what the basic creator functions already and maybe eventually all will attach as a class attribute.)

Note

For compatibility reasons, there is one exception to the rule above: When x is `integer`, the result of `data.class(x)` is "numeric" even when x is classed.

See Also

`class`

Examples

```
x <- LETTERS
data.class(factor(x))                # has a class attribute
data.class(matrix(x, ncol = 13))     # has a dim attribute
data.class(list(x))                  # the same as mode(x)
data.class(x)                        # the same as mode(x)

stopifnot(data.class(1:2) == "numeric") # compatibility "rule"
```

`data.frame` *Data Frames*

Description

The function `data.frame()` creates data frames, tightly coupled collections of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software.

Usage

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE, fix.empty.names = TRUE,
           stringsAsFactors = FALSE)
```

Arguments

`...`	these arguments are of either the form `value` or `tag = value`. Component names are created based on the tag (if present) or the deparsed argument itself.
`row.names`	`NULL` or a single integer or character string specifying a column to be used as row names, or a character or integer vector giving the row names for the data frame.
`check.rows`	if `TRUE` then the rows are checked for consistency of length and names.
`check.names`	logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted (by `make.names`) so that they are.
`fix.empty.names`	logical indicating if arguments which are "unnamed" (in the sense of not being formally called as `someName = arg`) get an automatically constructed name or rather name `""`. Needs to be set to `FALSE` even when `check.names` is false if `""` names should be kept.
`stringsAsFactors`	logical: should character vectors be converted to factors? The 'factory-fresh' default has been `TRUE` previously but has been changed to `FALSE` for R 4.0.0.

Details

A data frame is a list of variables of the same number of rows with unique row names, given class `"data.frame"`. If no variables are included, the row names determine the number of rows.

The column names should be non-empty, and attempts to use empty names will have unsupported results. Duplicate column names are allowed, but you need to use `check.names = FALSE` for `data.frame` to generate such a data frame. However, not all operations on data frames will preserve duplicated column names: for example matrix-like subsetting will force column names in the result to be unique.

`data.frame` converts each of its arguments to a data frame by calling `as.data.frame(optional = TRUE)`. As that is a generic function, methods can be written to change the behaviour of arguments according to their classes: R comes with many such methods. Character variables passed to `data.frame` are converted to factor columns if not protected by `I` and argument `stringsAsFactors` is true. If a list or data frame or matrix is passed to `data.frame` it is as if each component or column had been passed as a separate argument (except for matrices protected by `I`).

Objects passed to `data.frame` should have the same number of rows, but atomic vectors (see `is.vector`), factors and character vectors protected by `I` will be recycled a whole number of times if necessary (including as elements of list arguments).

If row names are not supplied in the call to `data.frame`, the row names are taken from the first component that has suitable names, for example a named vector or a matrix with rownames or a data frame. (If that component is subsequently recycled, the names are discarded with a warning.) If `row.names` was supplied as `NULL` or no suitable component was found the row names are the integer sequence starting at one (and such row names are considered to be 'automatic', and not preserved by `as.matrix`).

If row names are supplied of length one and the data frame has a single row, the `row.names` is taken to specify the row names and not a column (by name or number).

Names are removed from vector inputs not protected by `I`.

Value

A data frame, a matrix-like structure whose columns may be of differing types (numeric, logical, factor and character and so on).

How the names of the data frame are created is complex, and the rest of this paragraph is only the basic story. If the arguments are all named and simple objects (not lists, matrices of data frames) then the argument names give the column names. For an unnamed simple argument, a deparsed version of the argument is used as the name (with an enclosing `I(...)` removed). For a named matrix/list/data frame argument with more than one named column, the names of the columns are the name of the argument followed by a dot and the column name inside the argument: if the argument is unnamed, the argument's column names are used. For a named or unnamed matrix/list/data frame argument that contains a single column, the column name in the result is the column name in the argument. Finally, the names are adjusted to be unique and syntactically valid unless `check.names = FALSE`.

Note

In versions of R prior to 2.4.0 `row.names` had to be character: to ensure compatibility with such versions of R, supply a character vector as the `row.names` argument.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`I`, `plot.data.frame`, `print.data.frame`, `row.names`, `names` (for the column names), `[.data.frame` for subsetting methods and `I(matrix(..))` examples; `Math.data.frame` etc, about

Group methods for data.frames; read.table, make.names, list2DF for creating data frames from lists of variables.

Examples

```
L3 <- LETTERS[1:3]
char <- sample(L3, 10, replace = TRUE)
(d <- data.frame(x = 1, y = 1:10, char = char))
## The "same" with automatic column names:
data.frame(1, 1:10, sample(L3, 10, replace = TRUE))

is.data.frame(d)

## enable automatic conversion of character arguments to factor columns:
(dd <- data.frame(d, fac = letters[1:10], stringsAsFactors = TRUE))
rbind(class = sapply(dd, class), mode = sapply(dd, mode))

stopifnot(1:10 == row.names(d))  # {coercion}

(d0  <- d[, FALSE])   # data frame with 0 columns and 10 rows
(d.0 <- d[FALSE, ])   # <0 rows> data frame  (3 named cols)
(d00 <- d0[FALSE, ])  # data frame with 0 columns and 0 rows
```

---

data.matrix *Convert a Data Frame to a Numeric Matrix*

Description

Return the matrix obtained by converting all the variables in a data frame to numeric mode and then binding them together as the columns of a matrix. Factors and ordered factors are replaced by their internal codes.

Usage

```
data.matrix(frame, rownames.force = NA)
```

Arguments

frame	a data frame whose components are logical vectors, factors or numeric or character vectors.
rownames.force	logical indicating if the resulting matrix should have character (rather than NULL) rownames. The default, NA, uses NULL rownames if the data frame has 'automatic' row.names or for a zero-row data frame.

Details

Logical and factor columns are converted to integers. Character columns are first converted to factors and then to integers. Any other column which is not numeric (according to is.numeric) is converted by as.numeric or, for S4 objects, as(, "numeric"). If all columns are integer (after conversion) the result is an integer matrix, otherwise a numeric (double) matrix.

Value

If `frame` inherits from class `"data.frame"`, an integer or numeric matrix of the same dimensions as `frame`, with dimnames taken from the `row.names` (or `NULL`, depending on `rownames.force`) and `names`.

Otherwise, the result of `as.matrix`.

Note

The default behaviour for data frames differs from R < 2.5.0 which always gave the result character rownames.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`as.matrix`, `data.frame`, `matrix`.

Examples

```
DF <- data.frame(a = 1:3, b = letters[10:12],
                 c = seq(as.Date("2004-01-01"), by = "week", length.out = 3),
                 stringsAsFactors = TRUE)
data.matrix(DF[1:2])
data.matrix(DF)
```

`date` *System Date and Time*

Description

Returns a character string of the current system date and time.

Usage

```
date()
```

Value

The string has the form `"Fri Aug 20 11:11:00 1999"`, i.e., length 24, since it relies on POSIX's `ctime` ensuring the above fixed format. Timezone and Daylight Saving Time are taken account of, but *not* indicated in the result.

The day and month abbreviations are always in English, irrespective of locale.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Sys.Date` and `Sys.time`; `Date` and `DateTimeClasses` for objects representing date and time.

Examples

```
(d <- date())
nchar(d) == 24

## something similar in the current locale
##   depending on ctime; e.g. %e could be %d:
format(Sys.time(), "%a %b %e %H:%M:%S %Y")
```

`Dates` *Date Class*

Description

Description of the class `"Date"` representing calendar dates.

Usage

```
## S3 method for class 'Date'
summary(object, digits = 12, ...)

## S3 method for class 'Date'
print(x, max = NULL, ...)
```

Arguments

`object`, `x`	a `Date` object to be summarized or printed.
`digits`	number of significant digits for the computations.
`max`	numeric or `NULL`, specifying the maximal number of entries to be printed. By default, when `NULL`, `getOption("max.print")` used.
`...`	further arguments to be passed from or to other methods.

Details

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. They are always printed following the rules of the current Gregorian calendar, even though that calendar was not in use long ago (it was adopted in 1752 in Great Britain and its colonies). When printing there is assumed to be a year zero.

It is intended that the date should be an integer value, but this is not enforced in the internal representation. Fractional days will be ignored when printing. It is possible to produce fractional days via the mean method or by adding or subtracting (see Ops.Date).

When a date is converted to a date-time (for example by as.POSIXct or as.POSIXlt its time is taken as midnight in UTC.

Printing dates involves conversion to class "POSIXlt" which treats dates of more than about 780 million years from present as NA.

For the many methods see methods(class = "Date"). Several are documented separately, see below.

See Also

Sys.Date for the current date.

weekdays for convenience extraction functions.

Methods with extra arguments and documentation:

Ops.Date for operators on "Date" objects.

format.Date for conversion to and from character strings.

axis.Date and hist.Date for plotting.

seq.Date , cut.Date, and round.Date for utility operations.

DateTimeClasses for date-time classes.

Examples

```
(today <- Sys.Date())
format(today, "%d %b %Y")  # with month as a word
(tenweeks <- seq(today, length.out=10, by="1 week")) # next ten weeks
weekdays(today)
months(tenweeks)

(Dls <- as.Date(.leap.seconds))

## Show use of year zero:
(z <- as.Date("01-01-01")) # how it is printed depends on the OS
z - 365 # so year zero was a leap year.
as.Date("00-02-29")
# if you want a different format, consider something like (if supported)
## Not run: format(z, "%04Y-%m-%d") # "0001-01-01"
format(z, "%_4Y-%m-%d") # "   1-01-01"
format(z, "%_Y-%m-%d")  # "1-01-01"

## End(Not run)

##  length(<Date>) <- n   now works
ls <- Dls; length(ls) <- 12
l2 <- Dls; length(l2) <- 5 + length(Dls)
stopifnot(exprs = {
  ## length(.) <- * is compatible to subsetting/indexing:
```

```
  identical(ls, Dls[seq_along(ls)])
  identical(l2, Dls[seq_along(l2)])
  ## has filled with NA's
  is.na(l2[(length(Dls)+1):length(l2)])
})
```

DateTimeClasses *Date-Time Classes*

Description

Description of the classes "POSIXlt" and "POSIXct" representing calendar dates and times.

Usage

```
## S3 method for class 'POSIXct'
print(x, tz = "", usetz = TRUE, max = NULL, ...)

## S3 method for class 'POSIXct'
summary(object, digits = 15, ...)

time + z
z + time
time - z
time1 lop time2
```

Arguments

x, object	an object to be printed or summarized from one of the date-time classes.
tz, usetz	for timezone formatting, passed to format.POSIXct.
max	numeric or NULL, specifying the maximal number of entries to be printed. By default, when NULL, getOption("max.print") used.
digits	number of significant digits for the computations: should be high enough to represent the least important time unit exactly.
...	further arguments to be passed from or to other methods.
time	date-time objects.
time1, time2	date-time objects or character vectors. (Character vectors are converted by as.POSIXct.)
z	a numeric vector (in seconds).
lop	one of ==, !=, <, <=, > or >=.

Details

There are two basic classes of date/times. Class "POSIXct" represents the (signed) number of seconds since the beginning of 1970 (in the UTC time zone) as a numeric vector. Class "POSIXlt" is internally a list of vectors with components named sec, min, hour for the time, mday, mon, and year, for the date, wday, yday for the day of the week and day of the year, isdst, a Daylight Saving Time flag, and sometimes (both *optional*) zone, a string for the time zone, and gmtoff, offset in seconds from GMT, see the section 'Details on POSIXlt' below for more details.

The classes correspond to the POSIX/C99 constructs of 'calendar time' (the time_t data type, "ct"), and 'local time' (or broken-down time, the 'struct tm' data type, "lt"), from which they also inherit their names.

"POSIXct" is more convenient for including in data frames, and "POSIXlt" is closer to human-readable forms. A virtual class "POSIXt" exists from which both of the classes inherit: it is used to allow operations such as subtraction to mix the two classes.

Logical comparisons and some arithmetic operations are available for both classes. One can add or subtract a number of seconds from a date-time object, but not add two date-time objects. Subtraction of two date-time objects is equivalent to using difftime. Be aware that "POSIXlt" objects will be interpreted as being in the current time zone for these operations unless a time zone has been specified.

Both classes may have an attribute "tzone", specifying the time zone. Note however that their meaning differ, see the section 'Time Zones' below for more details.

Unfortunately, the conversion is complicated by the operation of time zones and leap seconds (according to this version of R's data, 27 days have been 86401 seconds long so far, the last being on (actually, immediately before) 2017-01-01: the times of the extra seconds are in the object .leap.seconds). The details of this are entrusted to the OS services where possible. It seems that some rare systems used to use leap seconds, but all known current platforms ignore them (as required by POSIX). This is detected and corrected for at build time, so "POSIXct" times used by R do not include leap seconds on any platform.

Using c on "POSIXlt" objects converts them to the current time zone, and on "POSIXct" objects drops "tzone" attributes if they are not all the same.

A few times have specific issues. First, the leap seconds are ignored, and real times such as "2005-12-31 23:59:60" are (probably) treated as the next second. However, they will never be generated by R, and are unlikely to arise as input. Second, on some OSes there is a problem in the POSIX/C99 standard with "1969-12-31 23:59:59 UTC", which is -1 in calendar time and that value is on those OSes also used as an error code. Thus as.POSIXct("1969-12-31 23:59:59", format = "%Y-%m-%d %H:%M:%S", tz = "UTC") may give NA, and hence as.POSIXct("1969-12-31 23:59:59", tz = "UTC") will give "1969-12-31 23:59:00". Other OSes (including the code used by R on Windows) report errors separately and so are able to handle that time as valid.

The print methods respect options("max.print").

Time zones

"POSIXlt" objects will often have an attribute "tzone", a character vector of length 3 giving the time zone name (from the TZ environment variable or argument tz of functions creating "POSIXlt" objects; "" marks the current time zone) and the names of the base time zone and the alternate (daylight-saving) time zone. Sometimes this may just be of length one, giving the time zone name.

"POSIXct" objects may also have an attribute "tzone", a character vector of length one. If set to a non-empty value, it will determine how the object is converted to class "POSIXlt" and in particular how it is printed. This is usually desirable, but if you want to specify an object in a particular time zone but to be printed in the current time zone you may want to remove the "tzone" attribute.

Details on POSIXlt

Class "POSIXlt" is internally a named list of vectors representing date-times, with the following list components

sec 0–61: seconds, allowing for leap seconds.

min 0–59: minutes.

hour 0–23: hours.

mday 1–31: day of the month.

mon 0–11: months after the first of the year.

year years since 1900.

wday 0–6 day of the week, starting on Sunday.

yday 0–365: day of the year (365 only in leap years).

isdst Daylight Saving Time flag. Positive if in force, zero if not, negative if unknown.

zone (Optional.) The abbreviation for the time zone in force at that time: "" if unknown (but "" might also be used for UTC).

gmtoff (Optional.) The offset in seconds from GMT: positive values are East of the meridian. Usually NA if unknown, but 0 could mean unknown.

The components must be in this order: that was only minimally checked prior to R 4.3.0. All objects created in R 4.3.0 have the optional components. From earlier versions of R, he last two components will not be present for times in UTC and are platform-dependent. Currently gmtoff is set on almost all current platforms: those based on BSD or glibc (including Linux and macOS) and those using the tzcode implementation shipped with R (including Windows and by default macOS).

Note that the internal list structure is somewhat hidden, as many methods (including length(x), print() and str()) apply to the abstract date-time vector, as for "POSIXct". One can extract and replace *single* components via [indexing with *two* indices (see the examples).

The components of "POSIXlt" are integer vectors, except sec (double) and zone (character). However most users will coerce numeric values for the first to real and the rest bar zone to integer.

Components wday and yday are for information, and are not used in the conversion to calendar time nor for printing, format(), or in as.character().

However, component isdst is needed to distinguish times at the end of DST: typically 1am to 2am occurs twice, first in DST and then in standard time. At all other times isdst can be deduced from the first six values, but the behaviour if it is set incorrectly is platform-dependent. For example Linux/glibc when checked fixed up incorrect values in time zones which support DST but gave an error on value 1 in those without DST.

For "ragged" and out-of-range vs "balanced" "POSIXlt" objects, see balancePOSIXlt().

Sub-second Accuracy

Classes "POSIXct" and "POSIXlt" are able to express fractions of a second where the latter allows for higher accuracy. Consequently, conversion of fractions between the two forms may not be exact, but will have better than microsecond accuracy.

Fractional seconds are printed only if options("digits.secs") is set: see strftime.

Valid ranges for times

The "POSIXlt" class can represent a very wide range of times (up to billions of years), but such times can only be interpreted with reference to a time zone.

The concept of time zones was first adopted in the nineteenth century, and the Gregorian calendar was introduced in 1582 but not universally adopted until 1927. OS services almost invariably assume the Gregorian calendar and may assume that the time zone that was first enacted for the location was in force before that date. (The earliest legislated time zone seems to have been London on 1847-12-01.) Some OSes assume the previous use of 'local time' based on the longitude of a location within the time zone.

Most operating systems represent POSIXct times as C type long. This means that on 32-bit OSes this covers the period 1902 to 2037. On all known 64-bit platforms and for the code we use on 32-bit Windows, the range of representable times is billions of years: however, not all can convert correctly times before 1902 or after 2037. A few benighted OSes used a unsigned type and so cannot represent times before 1970.

Where possible the platform limits are detected, and outside the limits we use our own C code. This uses the offset from GMT in use either for 1902 (when there was no DST) or that predicted for one of 2030 to 2037 (chosen so that the likely DST transition days are Sundays), and uses the alternate (daylight-saving) time zone only if isdst is positive or (if -1) if DST was predicted to be in operation in the 2030s on that day.

Note that there are places (e.g., Rome) whose offset from UTC varied in the years prior to 1902, and these will be handled correctly only where there is OS support.

There is no reason to assume that the DST rules will remain the same in the future: the US legislated in 2005 to change its rules as from 2007, with a possible future reversion. So conversions for times more than a year or two ahead are speculative. Other countries have changed their rules (and indeed, if DST is used at all) at a few days' notice. So representations and conversion of future dates are tentative. This also applies to dates after the in-use version of the time-zone database – not all platforms keep it up to date, which includes that shipped with older versions of R where used (which it is by default on Windows and macOS).

Warnings

Some Unix-like systems (especially Linux ones) do not have environment variable TZ set, yet have internal code that expects it (as does POSIX). We have tried to work around this, but if you get unexpected results try setting TZ. See Sys.timezone for valid settings.

Great care is needed when comparing objects of class "POSIXlt". Not only are components and attributes optional; several components may have values meaning 'not yet determined' and the same time represented in different time zones will look quite different.

The *order* of the list components of "POSIXlt" objects must not be changed, as several C-based conversion methods rely on the order for efficiency.

References

Ripley, B. D. and Hornik, K. (2001). "Date-time classes." *R News*, **1**(2), 8–11. https://www.r-project.org/doc/Rnews/Rnews_2001-2.pdf.

See Also

Dates for dates without times.

as.POSIXct and as.POSIXlt for conversion between the classes.

strptime for conversion to and from character representations.

Sys.time for clock time as a "POSIXct" object.

difftime for time intervals.

balancePOSIXlt() for balancing or filling "ragged" POSIXlt objects.

cut.POSIXt, seq.POSIXt, round.POSIXt and trunc.POSIXt for methods for these classes.

weekdays for convenience extraction functions.

Examples

```
(z <- Sys.time())             # the current date, as class "POSIXct"

Sys.time() - 3600             # an hour ago

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
format(.leap.seconds)         # the leap seconds in your time zone
print(.leap.seconds, tz = "America/Los_Angeles")  # and in Seattle's


## look at *internal* representation of "POSIXlt" :
leapS <- as.POSIXlt(.leap.seconds)
names(unclass(leapS)) ; is.list(leapS)
## str() on inner structure needs unclass(.):
utils::str(unclass(leapS), vec.len = 7)
## show all (apart from "tzone" attr):
data.frame(unclass(leapS))

## Extracting *single* components of POSIXlt objects:
leapS[1 : 5, "year"]
leapS[17:22, "mon" ]

##  length(.) <- n   now works for "POSIXct" and "POSIXlt" :
for(lpS in list(.leap.seconds, leapS)) {
    ls <- lpS; length(ls) <- 12
    l2 <- lpS; length(l2) <- 5 + length(lpS)
    stopifnot(exprs = {
      ## length(.) <- * is compatible to subsetting/indexing:
      identical(ls, lpS[seq_along(ls)])
      identical(l2, lpS[seq_along(l2)])
      ## has filled with NA's
      is.na(l2[(length(lpS)+1):length(l2)])
    })
```

```
}
```

---

dcf *Read and Write Data in DCF Format*

Description

Reads or writes an R object from/to a file in Debian Control File format.

Usage

```
read.dcf(file, fields = NULL, all = FALSE, keep.white = NULL)

write.dcf(x, file = "", append = FALSE, useBytes = FALSE,
          indent = 0.1 * getOption("width"),
          width = 0.9 * getOption("width"),
          keep.white = NULL)
```

Arguments

`file`	either a character string naming a file or a connection. `""` indicates output to the console. For `read.dcf` this can name a compressed file (see `gzfile`).
`fields`	a character vector with the names of the fields to read from the DCF file. Default is to read all fields.
`all`	a logical indicating whether in case of multiple occurrences of a field in a record, all these should be gathered. If `all` is false (default), only the last such occurrence is used.
`keep.white`	a character vector with the names of the fields for which whitespace should be kept as is, or `NULL` (default) indicating that there are no such fields. Coerced to character if possible. For fields where whitespace is not to be kept as is, `read.dcf` removes leading and trailing whitespace, and `write.dcf` folds using `strwrap`.
`x`	the object to be written, typically a data frame. If not, it is attempted to coerce `x` to a data frame.
`append`	logical. If `TRUE`, the output is appended to the file. If `FALSE`, any existing file of the name is destroyed.
`useBytes`	logical to be passed to `writeLines()`, see there: "for expert use".
`indent`	a positive integer specifying the indentation for continuation lines in output entries.
`width`	a positive integer giving the target column for wrapping lines in the output.

Details

DCF is a simple format for storing databases in plain text files that can easily be directly read and written by humans. DCF is used in various places to store R system information, like descriptions and contents of packages.

The DCF rules as implemented in R are:

1. A database consists of one or more records, each with one or more named fields. Not every record must contain each field. Fields may appear more than once in a record.
2. Regular lines start with a non-whitespace character.
3. Regular lines are of form `tag:value`, i.e., have a name tag and a value for the field, separated by `:` (only the first `:` counts). The value can be empty (i.e., whitespace only).
4. Lines starting with whitespace are continuation lines (to the preceding field) if at least one character in the line is non-whitespace. Continuation lines where the only non-whitespace character is a '`.`' are taken as blank lines (allowing for multi-paragraph field values).
5. Records are separated by one or more empty (i.e., whitespace only) lines.
6. Individual lines may not be arbitrarily long; prior to R 3.0.2 the length limit was approximately 8191 bytes per line.

Note that `read.dcf(all = FALSE)` reads the file byte-by-byte. This allows a '`DESCRIPTION`' file to be read and only its ASCII fields used, or its '`Encoding`' field used to re-encode the remaining fields.

`write.dcf` does not write `NA` fields.

Value

The default `read.dcf(all = FALSE)` returns a character matrix with one row per record and one column per field. Leading and trailing whitespace of field values is ignored unless a field is listed in `keep.white`. If a tag name is specified in the file, but the corresponding value is empty, then an empty string is returned. If the tag name of a field is specified in `fields` but never used in a record, then the corresponding value is `NA`. If fields are repeated within a record, the last one encountered is returned. Malformed lines lead to an error.

For `read.dcf(all = TRUE)` a data frame is returned, again with one row per record and one column per field. The columns are lists of character vectors for fields with multiple occurrences, and character vectors otherwise.

Note that an empty `file` is a valid DCF file, and `read.dcf` will return a zero-row matrix or data frame.

For `write.dcf`, invisible `NULL`.

Note

As from R 3.4.0, 'whitespace' in all cases includes newlines.

References

https://www.debian.org/doc/debian-policy/ch-controlfields.html.

Note that R does not require encoding in UTF-8, which is a recent Debian requirement. Nor does it use the Debian-specific sub-format which allows comment lines starting with '#'.

See Also

write.table.

available.packages, which uses read.dcf to read the indices of package repositories.

Examples

```
## Create a reduced version of the DESCRIPTION file in package 'splines'
x <- read.dcf(file = system.file("DESCRIPTION", package = "splines"),
              fields = c("Package", "Version", "Title"))
write.dcf(x)

## An online DCF file with multiple records
con <- url("https://cran.r-project.org/src/contrib/PACKAGES")
y <- read.dcf(con, all = TRUE)
close(con)
utils::str(y)
```

---

debug *Debug a Function*

Description

Set, unset or query the debugging flag on a function. The text and condition arguments are the same as those that can be supplied via a call to browser. They can be retrieved by the user once the browser has been entered, and provide a mechanism to allow users to identify which breakpoint has been activated.

Usage

```
debug(fun, text = "", condition = NULL, signature = NULL)
debugonce(fun, text = "", condition = NULL, signature = NULL)
undebug(fun, signature = NULL)
isdebugged(fun, signature = NULL)
debuggingState(on = NULL)
```

Arguments

fun	any interpreted R function.
text	a text string that can be retrieved when the browser is entered.
condition	a condition that can be retrieved when the browser is entered.
signature	an optional method signature. If specified, the method is debugged, rather than its generic.
on	logical; a call to the support function debuggingState returns TRUE if debugging is globally turned on, FALSE otherwise. An argument of one or the other of those values sets the state. If the debugging state is FALSE, none of the debugging actions will occur (but explicit browser calls in functions will continue to work).

Details

When a function flagged for debugging is entered, normal execution is suspended and the body of function is executed one statement at a time. A new `browser` context is initiated for each step (and the previous one destroyed).

At the debug prompt the user can enter commands or R expressions, followed by a newline. The commands are described in the `browser` help topic.

To debug a function which is defined inside another function, single-step through to the end of its definition, and then call debug on its name.

If you want to debug a function not starting at the very beginning, use `trace(..., at = *)` or `setBreakpoint`.

Using debug is persistent, and unless debugging is turned off the debugger will be entered on every invocation (note that if the function is removed and replaced the debug state is not preserved). Use `debugonce()` to enter the debugger only the next time the function is invoked.

To debug an S4 method by explicit signature, use `signature`. When specified, signature indicates the method of `fun` to be debugged. Note that debugging is implemented slightly differently for this case, as it uses the trace machinery, rather than the debugging bit. As such, `text` and `condition` cannot be specified in combination with a non-null `signature`. For methods which implement the `.local` rematching mechanism, the `.local` closure itself is the one that will be ultimately debugged (see `isRematched`).

`isdebugged` returns `TRUE` if a) `signature` is `NULL` and the closure `fun` has been debugged, or b) `signature` is not `NULL`, `fun` is an S4 generic, and the method of `fun` for that signature has been debugged. In all other cases, it returns `FALSE`.

The number of lines printed for the deparsed call when a function is entered for debugging can be limited by setting `options(deparse.max.lines)`.

When debugging is enabled on a byte compiled function then the interpreted version of the function will be used until debugging is disabled.

Value

debug and undebug invisibly return `NULL`.

`isdebugged` returns `TRUE` if the function or method is

marked for debugging, and `FALSE` otherwise.

See Also

`debugcall` for conveniently debugging methods, `browser` notably for its '*commands*', `trace`; `traceback` to see the stack after an `Error: ...` message; `recover` for another debugging approach.

Examples

```
## Not run:
debug(library)
library(methods)

## End(Not run)
## Not run:
```

```
debugonce(sample)
## only the first call will be debugged
sampe(10, 1)
sample(10, 1)

## End(Not run)
```

`declare` *Declarations*

Description

A framework for specifying information about R code for use by the interpreter, compiler, and code analysis tools.

Usage

```
declare(...)
```

Arguments

`...`	declaration expressions.

Details

A syntax for declaration expressions is still being developed.

Value

Evaluating a `declare()` call ignores the arguments and returns `NULL` invisibly.

`Defunct` *Marking Objects as Defunct*

Description

When a function is removed from R it should be replaced by a function which calls `.Defunct`.

Usage

```
.Defunct(new, package = NULL, msg)
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the defunct function might be listed.
`msg`	character string: A message to be printed, if missing a default message is used.

Details

`.Defunct` is called from defunct functions. Functions should be listed in `help("pkg-defunct")` for an appropriate `pkg`, including `base` (with the alias added to the respective Rd file).

`.Defunct` signals an error of class `defunctError` with fields `old`, `new`, and `package`.

See Also

`Deprecated`.

`base-defunct` and so on which list the defunct functions in the packages.

`delayedAssign` *Delay Evaluation and Promises*

Description

`delayedAssign` creates a *promise* to evaluate the given expression if its value is requested. This provides direct access to the *lazy evaluation* mechanism used by R for the evaluation of (interpreted) functions.

Usage

```
delayedAssign(x, value, eval.env = parent.frame(1),
              assign.env = parent.frame(1))
```

Arguments

`x`	a variable name (given as a quoted string in the function call)
`value`	an expression to be assigned to `x`
`eval.env`	an environment in which to evaluate `value`
`assign.env`	an environment in which to assign `x`

Details

Both `eval.env` and `assign.env` default to the currently active environment.

The expression assigned to a promise by `delayedAssign` will not be evaluated until it is eventually 'forced'. This happens when the variable is first accessed.

When the promise is eventually forced, it is evaluated within the environment specified by `eval.env` (whose contents may have changed in the meantime). After that, the value is fixed and the expression will not be evaluated again, where the promise still keeps its expression.

Value

This function is invoked for its side effect, which is assigning a promise to evaluate `value` to the variable `x`.

See Also

substitute, to see the expression associated with a promise, if assign.env is not the .GlobalEnv.

Examples

```
msg <- "old"
delayedAssign("x", msg)
substitute(x) # shows only 'x', as it is in the global env.
msg <- "new!"
x # new!

delayedAssign("x", {
    for(i in 1:3)
        cat("yippee!\n")
    10
})

x^2 #- yippee
x^2 #- simple number

ne <- new.env()
delayedAssign("x", pi + 2, assign.env = ne)
## See the promise {without "forcing" (i.e. evaluating) it}:
substitute(x, ne) #  'pi + 2'


### Promises in an environment [for advanced users]:  ----------------------

e <- (function(x, y = 1, z) environment())(cos, "y", {cat(" HO!\n"); pi+2})
## How can we look at all promises in an env (w/o forcing them)?
gete <- function(e_) {
   ne <- names(e_)
   names(ne) <- ne
   lapply(lapply(ne, as.name),
          function(n) eval(substitute(substitute(X, e_), list(X=n))))
}
(exps <- gete(e))
sapply(exps, typeof)

(le <- as.list(e)) # evaluates ("force"s) the promises
stopifnot(identical(le, lapply(exps, eval))) # and another "Ho!"
```

---

deparse *Expression Deparsing*

Description

Turn unevaluated expressions into character strings.

Usage

```
deparse(expr, width.cutoff = 60L,
        backtick = mode(expr) %in% c("call", "expression", "(", "function"),
        control = c("keepNA", "keepInteger", "niceNames", "showAttributes"),
        nlines = -1L)

deparse1(expr, collapse = " ", width.cutoff = 500L, ...)
```

Arguments

`expr`	any R expression.
`width.cutoff`	integer in $[20, 500]$ determining the cutoff (in bytes) at which line-breaking is tried.
`backtick`	logical indicating whether symbolic names should be enclosed in backticks if they do not follow the standard syntax.
`control`	character vector (or `NULL`) of deparsing options. `control = "all"` is thorough, see `.deparseOpts`.
`nlines`	integer: the maximum number of lines to produce. Negative values indicate no limit.
`collapse`	a string, passed to `paste()`.
`...`	further arguments passed to `deparse()`.

Details

These functions turn unevaluated expressions (where 'expression' is taken in a wider sense than the strict concept of a vector of `mode` and type (`typeof`) `"expression"` used in `expression`) into character strings (a kind of inverse to `parse`).

A typical use of this is to create informative labels for data sets and plots. The example shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

The default for the `backtick` option is not to quote single symbols but only composite expressions. This is a compromise to avoid breaking existing code.

`width.cutoff` is a lower bound for the line lengths: deparsing a line proceeds until at least `width.cutoff` *bytes* have been output and e.g. `arg = value` expressions will not be split across lines.

`deparse1()` is a simple utility added in R 4.0.0 to ensure a string result (`character` vector of length one), typically used in name construction, as `deparse1(substitute(.))`.

Note

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be deparsed as an attribute.

Deparsing internal structures may not be accurate: for example the graphics display list recorded by `recordPlot` is not intended to be deparsed and `.Internal` calls will be shown as primitive calls.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.deparseOpts` for available `control` settings; `dput()` and `dump()` for related functions using identical internal deparsing functionality.

`substitute`, `parse`, `expression`.

`Quotes` for quoting conventions, including backticks.

Examples

```
require(stats); require(graphics)

deparse(args(lm))
deparse(args(lm), width.cutoff = 500)

myplot <- function(x, y) {
    plot(x, y, xlab = deparse1(substitute(x)),
               ylab = deparse1(substitute(y)))
}

e <- quote(`foo bar`)
deparse(e)
deparse(e, backtick = TRUE)
e <- quote(`foo bar`+1)
deparse(e)
deparse(e, control = "all") # wraps it w/ quote( . )
```

---

deparseOpts *Options for Expression Deparsing*

Description

Process the deparsing options for `deparse`, `dput` and `dump`.

Usage

```
.deparseOpts(control)

..deparseOpts
```

Arguments

`control`	character vector of deparsing options.

Details

`..deparseOpts` is the `character` vector of possible deparsing options used by `.deparseOpts()`.

`.deparseOpts()` is called by `deparse`, `dput` and `dump` to process their `control` argument.

The `control` argument is a vector containing zero or more of the following strings (exactly those in `..deparseOpts`). Partial string matching is used.

`"keepInteger"`: Either surround integer vectors by `as.integer()` or use suffix `L`, so they are not converted to type double when parsed. This includes making sure that integer `NA`s are preserved (via `NA_integer_` if there are no non-`NA` values in the vector, unless `"S_compatible"` is set).

`"quoteExpressions"`: Surround unevaluated expressions, but not `formula`s, with `quote()`, so they are not evaluated when re-parsed.

`"showAttributes"`: If the object has `attributes` (other than a `source` attribute, see `srcref`), use `structure()` to display them as well as the object value unless the only such attribute is `names` and the `"niceNames"` option is set. This (`"showAttributes"`) is the default for `deparse` and `dput`.

`"useSource"`: If the object has a `source` attribute (`srcref`), display that instead of deparsing the object. Currently only applies to function definitions.

`"warnIncomplete"`: Some exotic objects such as environments, external pointers, etc. can not be deparsed properly. This option causes a warning to be issued if the deparser recognizes one of these situations.

Also, the parser in R < 2.7.0 would only accept strings of up to 8192 bytes, and this option gives a warning for longer strings.

`"keepNA"`: Integer, real and character `NA`s are surrounded by coercion functions where necessary to ensure that they are parsed to the same type. Since e.g. `NA_real_` can be output in R, this is mainly used in connection with `S_compatible`.

`"niceNames"`: If true, `list`s and atomic vectors with non-`NA` names (see `names`) are deparsed as e.g., `c(A = 1)` instead of `structure(1, names = "A")`, independently of the `"showAttributes"` setting.

`"all"`: An abbreviated way to specify all of the options listed above *plus* `"digits17"`. This is the default for `dump`, and, without `"digits17"`, the options used by `edit` (which are fixed).

`"delayPromises"`: Deparse promises in the form <promise: expression> rather than evaluating them. The value and the environment of the promise will not be shown and the deparsed code cannot be sourced.

`"S_compatible"`: Make deparsing as far as possible compatible with S and R < 2.5.0. For compatibility with S, integer values of double vectors are deparsed with a trailing decimal point. Backticks are not used.

`"hexNumeric"`: Real and finite complex numbers are output in ‘`"%a"`’ format as binary fractions (coded as hexadecimal: see `sprintf`) with maximal opportunity to be recorded exactly to full precision. Complex numbers with one or both non-finite components are output as if this option were not set.

(This relies on that format being correctly supported: known problems on Windows are worked around as from R 3.1.2.)

"digits17": Real and finite complex numbers are output using format '"%.17g"' which may give more precision than the default (but the output will depend on the platform and there may be loss of precision when read back). Complex numbers with one or both non-finite components are output as if this option were not set.

"exact": An abbreviated way to specify control = c("all", "hexNumeric") which is guaranteed to be exact for numbers, see also below.

For the most readable (but perhaps incomplete) display, use control = NULL. This displays the object's value, but not its attributes. The default in deparse is to display the attributes as well, but not to use any of the other options to make the result parseable. (dump uses more default options via control = "all", and printing of functions without sources uses c("keepInteger", "keepNA") to which one may add "warnIncomplete".)

Using control = "exact" (short for control = c("all", "hexNumeric")) comes closest to making deparse() an inverse of parse() (but we have not yet seen an example where "all", now including "digits17", would not have been as good). However, not all objects are deparse-able even with these options, and a warning will be issued if the function recognizes that it is being asked to do the impossible.

Only one of "hexNumeric" and "digits17" can be specified.

Value

An integer value corresponding to the control options selected.

Examples

```
stopifnot(.deparseOpts("exact") == .deparseOpts(c("all", "hexNumeric")))
(iOpt.all <- .deparseOpts("all")) # a four digit integer

## one integer --> vector binary bits
int2bits <- function(x, base = 2L,
                     ndigits = 1 + floor(1e-9 + log(max(x,1), base))) {
    r <- numeric(ndigits)
    for (i in ndigits:1) {
        r[i] <- x%%base
        if (i > 1L)
            x <- x%/%base
    }
    rev(r) # smallest bit at left
}
int2bits(iOpt.all)
## What options does  "all" contain ? =========
(depO.indiv <- setdiff(..deparseOpts, c("all", "exact")))
(oa <- depO.indiv[int2bits(iOpt.all) == 1])# 8 strings
stopifnot(identical(iOpt.all, .deparseOpts(oa)))

## ditto for "exact" instead of "all":
(iOpt.X <- .deparseOpts("exact"))
data.frame(opts = depO.indiv,
           all  = int2bits(iOpt.all),
           exact= int2bits(iOpt.X))
(oX <- depO.indiv[int2bits(iOpt.X) == 1]) # 8 strings, too
```

```
diffXall <- oa != oX
stopifnot(identical(iOpt.X, .deparseOpts(oX)),
          identical(oX[diffXall], "hexNumeric"),
          identical(oa[diffXall], "digits17"))
```

Deprecated *Marking Objects as Deprecated*

Description

When an object is about to be removed from R it is first deprecated and should include a call to `.Deprecated`.

Usage

```
.Deprecated(new, package = NULL, msg,
            old = as.character(sys.call(sys.parent()))[1L])
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the deprecated function might be listed.
`msg`	character string: A message to be printed, if missing a default message is used.
`old`	character string specifying the function (default) or usage which is being deprecated.

Details

`.Deprecated("new name")` is called from deprecated functions. The original help page for these functions is often available at `help("old-deprecated")` (note the quotes). Deprecated functions should be listed in `help("pkg-deprecated")` for an appropriate *pkg*, including **base**.

`.Deprecated` signals a warning of class `"deprecatedWarning"` with fields `old`, `new`, and `package`.

See Also

`Defunct`

`help("base-deprecated")` and so on which list the deprecated functions in the packages.

det *Calculate the Determinant of a Matrix*

Description

det calculates the determinant of a matrix. determinant is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant.

Usage

```
det(x, ...)
determinant(x, logarithm = TRUE, ...)
```

Arguments

x	numeric matrix: logical matrices are coerced to numeric.
logarithm	logical; if TRUE (default) return the logarithm of the modulus of the determinant.
...	optional arguments, currently unused.

Details

The determinant function uses an LU decomposition and the det function is simply a wrapper around a call to determinant.

Often, computing the determinant is *not* what you should be doing to solve a given problem.

Value

For det, the determinant of x. For determinant, a list with components

modulus	a numeric value. The modulus (absolute value) of the determinant if logarithm is FALSE; otherwise the logarithm of the modulus.
sign	integer; either $+1$ or -1 according to whether the determinant is positive or negative.

Examples

```
(x <- matrix(1:4, ncol = 2))
unlist(determinant(x))
det(x)

det(print(cbind(1, 1:3, c(2,0,1))))
```

detach *Detach Objects from the Search Path*

Description

Detach a database, i.e., remove it from the search() path of available R objects. Usually this is either a data.frame which has been attached or a package which was attached by library.

Usage

```
detach(name, pos = 2L, unload = FALSE, character.only = FALSE,
       force = FALSE)
```

Arguments

name	the object to detach. Defaults to search()[pos]. This can be an unquoted name or a character string but *not* a character vector. If a number is supplied this is taken as pos.
pos	index position in search() of the database to detach. When name is a number, pos = name is used.
unload	a logical value indicating whether or not to attempt to unload the namespace when a package is being detached. If the package has a namespace and unload is TRUE, then detach will attempt to unload the namespace *via* unloadNamespace: if the namespace is imported by another namespace or unload is FALSE, no unloading will occur.
character.only	a logical indicating whether name can be assumed to be a character string.
force	logical: should a package be detached even though other attached packages depend on it?

Details

This is most commonly used with a single number argument referring to a position on the search list, and can also be used with a unquoted or quoted name of an item on the search list such as package:tools.

If a package has a namespace, detaching it does not by default unload the namespace (and may not even with unload = TRUE), and detaching will not in general unload any dynamically loaded compiled code (DLLs); see getLoadedDLLs and library.dynam.unload. Further, registered S3 methods from the namespace will not be removed, and because S3 methods are not tagged to their source on registration, it is in general not possible to safely un-register the methods associated with a given package. If you use library on a package whose namespace is loaded, it attaches the exports of the already loaded namespace. So detaching and re-attaching a package may not refresh some or all components of the package, and is inadvisable. The most reliable way to completely detach a package is to restart R.

Value

The return value is invisible. It is NULL when a package is detached, otherwise the environment which was returned by attach when the object was attached (incorporating any changes since it was attached).

Good practice

detach() without an argument removes the first item on the search path after the workspace. It is all too easy to call it too many or too few times, or to not notice that the search path has changed since an attach call.

Use of attach/detach is best avoided in functions (see the help for attach) and in interactive use and scripts it is prudent to detach by name.

Note

You cannot detach either the workspace (position 1) nor the **base** package (the last item in the search list), and attempting to do so will throw an error.

Unloading some namespaces has undesirable side effects: e.g. unloading **grid** closes all graphics devices, and on some systems **tcltk** cannot be reloaded once it has been unloaded and may crash R if this is attempted.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attach, library, search, objects, unloadNamespace, library.dynam.unload .

Examples

```
require(splines) # package
detach(package:splines)
## or also
library(splines)
pkg <- "package:splines"

detach(pkg, character.only = TRUE)

## careful: do not do this unless 'splines' is not already attached.
library(splines)
detach(2) # 'pos' used for 'name'

## an example of the name argument to attach
## and of detaching a database named by a character vector
attach_and_detach <- function(db, pos = 2)
{
   name <- deparse1(substitute(db))
   attach(db, pos = pos, name = name)
```

```
    print(search()[pos])
    detach(name, character.only = TRUE)
}
attach_and_detach(women, pos = 3)
```

---

diag *Matrix Diagonals*

Description

Extract or replace the diagonal of a matrix, or construct a diagonal matrix.

Usage

```
diag(x = 1, nrow, ncol, names = TRUE)
diag(x) <- value
```

Arguments

x	a matrix, vector or 1D array, or missing.
nrow, ncol	optional dimensions for the result when x is not a matrix.
names	(when x is a matrix) logical indicating if the resulting vector, the diagonal of x, should inherit names from dimnames(x) if available.
value	either a single value or a vector of length equal to that of the current diagonal. Should be of a mode which can be coerced to that of x.

Details

diag has four distinct usages:

1. x is a matrix, when it extracts the diagonal.
2. x is missing and nrow is specified, it returns an identity matrix.
3. x is a scalar (length-one vector) and the only argument, it returns a square identity matrix of size given by the scalar.
4. x is a 'numeric' (complex, numeric, integer, logical, or raw) vector, either of length at least 2 or there were further arguments. This returns a matrix with the given diagonal and zero off-diagonal entries.

It is an error to specify nrow or ncol in the first case.

Value

If `x` is a matrix then `diag(x)` returns the diagonal of `x`. The resulting vector will have `names` if `names` is true and if the matrix `x` has matching column and rownames.

The replacement form sets the diagonal of the matrix `x` to the given value(s).

In all other cases the value is a diagonal matrix with `nrow` rows and `ncol` columns (if `ncol` is not given the matrix is square). Here `nrow` is taken from the argument if specified, otherwise inferred from `x`: if that is a vector (or 1D array) of length two or more, then its length is the number of rows, but if it is of length one and neither `nrow` nor `ncol` is specified, `nrow = as.integer(x)`.

When a diagonal matrix is returned, the diagonal elements are one except in the fourth case, when `x` gives the diagonal elements: it will be recycled or truncated as needed, but fractional recycling and truncation will give a warning.

Note

Using `diag(x)` can have unexpected effects if `x` is a vector that could be of length one. Use `diag(x, nrow = length(x))` for consistent behaviour.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`upper.tri`, `lower.tri`, `matrix`.

Examples

```
dim(diag(3))
diag(10, 3, 4) # guess what?
all(diag(1:3) == {m <- matrix(0,3,3); diag(m) <- 1:3; m})

## other "numeric"-like diagonal matrices :
diag(c(1i,2i))    # complex
diag(TRUE, 3)     # logical
diag(as.raw(1:3)) # raw
(D2 <- diag(2:1, 4)); typeof(D2) # "integer"

require(stats)
## diag(<var-cov-matrix>) = variances
diag(var(M <- cbind(X = 1:5, Y = rnorm(5))))
#-> vector with names "X" and "Y"
rownames(M) <- c(colnames(M), rep("", 3))
M; diag(M) #  named as well
diag(M, names = FALSE) # w/o names
```

`diff` *Lagged Differences*

Description

Returns suitably lagged and iterated differences.

Usage

```
diff(x, ...)

## Default S3 method:
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'POSIXt'
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'Date'
diff(x, lag = 1, differences = 1, ...)
```

Arguments

`x`	a numeric vector or matrix containing the values to be differenced.
`lag`	an integer indicating which lag to use.
`differences`	an integer indicating the order of the difference.
`...`	further arguments to be passed to or from methods.

Details

`diff` is a generic function with a default method and ones for classes `"ts"`, `"POSIXt"` and `"Date"`.

`NA`'s propagate.

Value

If `x` is a vector of length `n` and `differences = 1`, then the computed result is equal to the successive differences `x[(1+lag):n] - x[1:(n-lag)]`.

If `difference` is larger than one this algorithm is applied recursively to `x`. Note that the returned value is a vector which is shorter than `x`.

If `x` is a matrix then the difference operations are carried out on each column separately.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

diff.ts, diffinv.

Examples

```
diff(1:10, 2)
diff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
diff(x, lag = 2)
diff(x, differences = 2)

diff(.leap.seconds)
## allows to pass units via ... to difftime()
diff(.leap.seconds, units = "weeks")
diff(as.Date(.leap.seconds), units = "weeks")
```

---

difftime *Time Intervals / Differences*

Description

Time intervals creation, printing, and some arithmetic. The print() method calls these "time differences".

Usage

```
time1 - time2

difftime(time1, time2, tz,
         units = c("auto", "secs", "mins", "hours",
                   "days", "weeks"))

as.difftime(tim, format = "%X", units = "auto", tz = "UTC")

## S3 method for class 'difftime'
format(x, ...)
## S3 method for class 'difftime'
units(x)
## S3 replacement method for class 'difftime'
units(x) <- value
## S3 method for class 'difftime'
as.double(x, units = "auto", ...)

## Group methods, notably for round(), signif(), floor(),
## ceiling(), trunc(), abs(); called directly, *not* as Math():
## S3 method for class 'difftime'
Math(x, ...)
```

Arguments

time1, time2	date-time or date objects.
tz	an optional time zone specification to be used for the conversion, mainly for "POSIXlt" objects.
units	character string. Units in which the results are desired. Can be abbreviated.
value	character string. Like units, except that abbreviations are not allowed.
tim	character string or numeric value specifying a time interval.
format	character specifying the format of tim: see strptime. The default is a locale-specific time format.
x	an object inheriting from class "difftime".
...	arguments to be passed to or from other methods.

Details

Function difftime calculates a difference of two date/time objects and returns an object of class "difftime" with an attribute indicating the units. The Math group method provides round, signif, floor, ceiling, trunc, abs, and sign methods for objects of this class, and there are methods for the group-generic (see Ops) logical and arithmetic operations.

If units = "auto", a suitable set of units is chosen, the largest possible (excluding "weeks") in which all the absolute differences are greater than one.

Subtraction of date-time objects gives an object of this class, by calling difftime with units = "auto". Alternatively, as.difftime() works on character-coded or numeric time intervals; in the latter case, units must be specified, and format has no effect.

Limited arithmetic is available on "difftime" objects: they can be added or subtracted, and multiplied or divided by a numeric vector. In addition, adding or subtracting a numeric vector by a "difftime" object implicitly converts the numeric vector to a "difftime" object with the same units as the "difftime" object. There are methods for mean and sum (via the Summary group generic), and diff via diff.default building on the "difftime" method for arithmetic, notably -.

The units of a "difftime" object can be extracted by the units function, which also has a replacement form. If the units are changed, the numerical value is scaled accordingly. The replacement version keeps attributes such as names and dimensions.

Note that units = "days" means a period of 24 hours, hence takes no account of Daylight Savings Time. Differences in objects of class "Date" are computed as if in the UTC time zone.

The as.double method returns the numeric value expressed in the specified units. Using units = "auto" means the units of the object.

The format method simply formats the numeric value and appends the units as a text string.

Warning

Because R follows POSIX (and almost all computer clocks) in ignoring leap seconds, so do time differences. So in a UTC time zone

```
z <- as.POSIXct(c("2016-12-31 23:59:59", "2017-01-01 00:00:01"))
z[2] - z[1]
```

reports 'Time difference of 2 secs' but 3 seconds elapsed while the computer clock advanced by 2 seconds.

If you want the elapsed time interval, you need to add in any leap seconds for yourself.

Note

Units such as "months" are not possible as they are not of constant length. To create intervals of months, quarters or years use seq.Date or seq.POSIXt.

See Also

DateTimeClasses.

Examples

```
(z <- Sys.time() - 3600)
Sys.time() - z               # just over 3600 seconds.

## time interval between release days of R 1.2.2 and 1.2.3.
ISOdate(2001, 4, 26) - ISOdate(2001, 2, 26)

as.difftime(c("0:3:20", "11:23:15"))
as.difftime(c("3:20", "23:15", "2:"), format = "%H:%M") # 3rd gives NA
(z <- as.difftime(c(0,30,60), units = "mins"))
as.numeric(z, units = "secs")
as.numeric(z, units = "hours")
format(z)
```

---

dim *Dimensions of an Object*

Description

Retrieve or set the dimension of an object.

Usage

```
dim(x)
dim(x) <- value
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	for the default method, either NULL or a numeric vector, which is coerced to integer (by truncation).

Details

The functions dim and dim<- are internal generic primitive functions.

dim has a method for data.frames, which returns the lengths of the row.names attribute of x and of x (as the numbers of rows and columns respectively).

Value

For an array (and hence in particular, for a matrix) dim retrieves the dim attribute of the object. It is NULL or a vector of mode integer.

The replacement method changes the "dim" attribute (provided the new value is compatible) and removes any "dimnames" *and* "names" attributes.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ncol, nrow and dimnames.

Examples

```
x <- 1:12 ; dim(x) <- c(3,4)
x

# simple versions of nrow and ncol could be defined as follows
nrow0 <- function(x) dim(x)[1]
ncol0 <- function(x) dim(x)[2]
```

---

dimnames *Dimnames of an Object*

Description

Retrieve or set the dimnames of an object.

Usage

```
dimnames(x)
dimnames(x) <- value

provideDimnames(x, sep = "", base = list(LETTERS), unique = TRUE)
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	a possible value for dimnames(x): see the 'Value' section.
sep	a character string, used to separate base symbols and digits in the constructed dimnames.
base	a non-empty list of character vectors. The list components are used in turn (and recycled when needed) to construct replacements for empty dimnames components. See also the examples.
unique	logical indicating that the dimnames constructed are unique within each dimension in the sense of make.unique.

Details

The functions dimnames and dimnames<- are generic.

For an array (and hence in particular, for a matrix), they retrieve or set the dimnames attribute (see attributes) of the object. A list value can have names, and these will be used to label the dimensions of the array where appropriate.

The replacement method for arrays/matrices coerces vector and factor elements of value to character, but does not dispatch methods for as.character. It coerces zero-length elements to NULL, and a zero-length list to NULL. If value is a list shorter than the number of dimensions, it is extended with NULLs to the needed length.

Both have methods for data frames. The dimnames of a data frame are its row.names and its names. For the replacement method each component of value will be coerced by as.character.

For a 1D matrix the names are the same thing as the (only) component of the dimnames.

Both are primitive functions.

provideDimnames(x) provides dimnames where "missing", such that its result has character dimnames for each component. If unique is true as by default, they are unique within each component via make.unique(*, sep=sep).

Value

The dimnames of a matrix or array can be NULL (which is not stored) or a list of the same length as dim(x). If a list, its components are either NULL or a character vector with positive length of the appropriate dimension of x. The list can have names. It is possible that all components are NULL: such dimnames may get converted to NULL.

For the "data.frame" method both dimnames are character vectors, and the rownames must contain no duplicates nor missing values.

provideDimnames(x) returns x, with "NULL - free" dimnames, i.e. each component a character vector of correct length.

Note

Setting components of the dimnames, e.g., dimnames(A)[[1]] <- value is a common paradigm, but note that it will not work if the value assigned is NULL. Use rownames instead, or (as it does) manipulate the whole dimnames list.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

rownames, colnames; array, matrix, data.frame.

Examples

```
## simple versions of rownames and colnames
## could be defined as follows
rownames0 <- function(x) dimnames(x)[[1]]
colnames0 <- function(x) dimnames(x)[[2]]

(dn <- dimnames(A <- provideDimnames(N <- array(1:24, dim = 2:4))))
A0 <- A; dimnames(A)[2:3] <- list(NULL)
stopifnot(identical(A0, provideDimnames(A)))
strd <- function(x) utils::str(dimnames(x))
strd(provideDimnames(A, base= list(letters[-(1:9)], tail(LETTERS))))
strd(provideDimnames(N, base= list(letters[-(1:9)], tail(LETTERS)))) # recycling
strd(provideDimnames(A, base= list(c("AA","BB")))) # recycling on both levels
## set "empty dimnames":
provideDimnames(rbind(1, 2:3), base = list(""), unique=FALSE)
```

---

do.call *Execute a Function Call*

Description

do.call constructs and executes a function call from a name or a function and a list of arguments to be passed to it.

Usage

```
do.call(what, args, quote = FALSE, envir = parent.frame())
```

Arguments

what	either a function or a non-empty character string naming the function to be called.
args	a *list* of arguments to the function call. The names attribute of args gives the argument names.
quote	a logical value indicating whether to quote the arguments.
envir	an environment within which to evaluate the call. This will be most useful if what is a character string and the arguments are symbols or quoted expressions.

Details

If quote is FALSE, the default, then the arguments are evaluated (in the calling environment, not in envir). If quote is TRUE then each argument is quoted (see quote) so that the effect of argument evaluation is to remove the quotes – leaving the original arguments unevaluated when the call is constructed.

The behavior of some functions, such as substitute, will not be the same for functions evaluated using do.call as if they were evaluated from the interpreter. The precise semantics are currently undefined and subject to change.

Value

The result of the (evaluated) function call.

Warning

This should not be used to attempt to evade restrictions on the use of .Internal and other non-API calls.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

call which creates an unevaluated call.

Examples

```
do.call("complex", list(imaginary = 1:3))

## if we already have a list (e.g., a data frame)
## we need c() to add further arguments
tmp <- expand.grid(letters[1:2], 1:3, c("+", "-"))
do.call("paste", c(tmp, sep = ""))

do.call(paste, list(as.name("A"), as.name("B")), quote = TRUE)

## examples of where objects will be found.
A <- 2
f <- function(x) print(x^2)
env <- new.env()
assign("A", 10, envir = env)
assign("f", f, envir = env)
f <- function(x) print(x)
f(A)                                     # 2
do.call("f", list(A))                    # 2
do.call("f", list(A), envir = env)       # 4
do.call( f,  list(A), envir = env)       # 2
do.call("f", list(quote(A)), envir = env) # 100
```

```
do.call( f,  list(quote(A)), envir = env) # 10
do.call("f", list(as.name("A")), envir = env) # 100

eval(call("f", A))                      # 2
eval(call("f", quote(A)))               # 2
eval(call("f", A), envir = env)         # 4
eval(call("f", quote(A)), envir = env)  # 100
```

---

dontCheck *Identity Function to Suppress Checking*

Description

The dontCheck function is the same as identity, but is interpreted by R CMD check code analysis as a directive to suppress checking of x. Currently this is only used by checkFF(registration = TRUE) when checking the .NAME argument of foreign function calls.

Usage

```
dontCheck(x)
```

Arguments

x	an R object.

See Also

suppressForeignCheck which explains why that and dontCheck are undesirable and should be avoided if at all possible.

---

dots ..., ..1, *etc used in Functions*

Description

... and ..1, ..2 etc are used to refer to arguments passed down from a calling function. These (and the following) can only be used *inside* a function which has ... among its formal arguments.

...elt(n) is a functional way to get ..*n* and basically the same as eval(paste0("..", n)), just more elegant and efficient. Note that switch(n, ...) is very close, differing by returning NULL invisibly instead of an error when n is zero or too large.

...length() returns the number of expressions in ..., and ...names() the names. These are the same as length(list(...)) or names(list(...)) but without evaluating the expressions in ... (which happens with list(...)).

Evaluating elements of ... with ..1, ..2, ...elt(n), etc. propagates visibility. This is consistent with the evaluation of named arguments which also propagates visibility.

Usage

```
...length()
...names()
...elt(n)
```

Arguments

n	a positive integer, not larger than the number of expressions in ..., which is the same as `...length()` which is the same as `length(list(...))`, but the latter evaluates all expressions in ....

See Also

`...` and `..1`, `..2` are *reserved* words in R, see `Reserved`.

For more, see the 'Introduction to R' manual for usage of these syntactic elements, and dotsMethods for their use in formal (S4) methods.

Examples

```
tst <- function(n, ...) ...elt(n)
tst(1, pi=pi*0:1, 2:4) ## [1] 0.000000 3.141593
tst(2, pi=pi*0:1, 2:4) ## [1] 2 3 4
try(tst(1)) # -> Error about '...' not containing an element.

tst.dl  <- function(x, ...) ...length()
tst.dns <- function(x, ...) ...names()
tst.dl(1:10)    # 0  (because the first argument is 'x')
tst.dl(4, 5)    # 1
tst.dl(4, 5, 6) # 2  namely '5, 6'
tst.dl(4, 5, 6, 7, sin(1:10), "foo"/"bar") # 5.    Note: no evaluation!
tst.dns(4, foo=5, 6, bar=7, sini = sin(1:10), "foo"/"bar")
##        "foo"  "" "bar"  "sini"               ""

## From R 4.1.0 to 4.1.2, ...names() sometimes did not match names(list(...));
## check and show (these examples all would've failed):
chk.n2 <- function(...) stopifnot(identical(print(...names()), names(list(...))))
chk.n2(4, foo=5, 6, bar=7, sini = sin(1:10), "bar")
chk.n2()
chk.n2(1,2)
```

---

double *Double-Precision Vectors*

Description

Create, coerce to or test for a double-precision vector.

Usage

```
double(length = 0)
as.double(x, ...)
is.double(x)

single(length = 0)
as.single(x, ...)
```

Arguments

length	a non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

`double` creates a double-precision vector of the specified length. The elements of the vector are all equal to `0`. It is identical to `numeric`.

`as.double` is a generic function. It is identical to `as.numeric`. Methods should return an object of base type `"double"`.

`is.double` is a test of double type.

R *has no single precision data type. All real numbers are stored in double precision format.* The functions `as.single` and `single` are identical to `as.double` and `double` except they set the attribute `Csingle` that is used in the `.C` and `.Fortran` interface, and they are intended only to be used in that context.

Value

`double` creates a double-precision vector of the specified length. The elements of the vector are all equal to `0`.

`as.double` attempts to coerce its argument to be of double type: like `as.vector` it strips attributes including names. (To ensure that an object is of double type without stripping attributes, use `storage.mode`.) Character strings containing optional whitespace followed by either a decimal representation or a hexadecimal representation (starting with `0x` or `0X`) can be converted, as can special values such as `"NA"`, `"NaN"`, `"Inf"` and `"infinity"`, irrespective of case.

`as.double` for factors yields the codes underlying the factor levels, not the numeric representation of the labels, see also `factor`.

`is.double` returns `TRUE` or `FALSE` depending on whether its argument is of double type or not.

Double-precision values

All R platforms are required to work with values conforming to the IEC 60559 (also known as IEEE 754) standard. This basically works with a precision of 53 bits, and represents to that precision a range of absolute values from about 2×10^{-308} to 2×10^{308}. It also has special values `NaN` (many of them), plus and minus infinity and plus and minus zero (although R acts as if these are the same).

There are also *denormal(ized)* (or *subnormal*) numbers with values below the range given above but represented to less precision.

See `.Machine` for precise information on these limits. Note that ultimately how double precision numbers are handled is down to the CPU/FPU and compiler.

In IEEE 754-2008/IEC60559:2011 this is called 'binary64' format.

Note on names

It is a historical anomaly that R has two names for its floating-point vectors, `double` and `numeric` (and formerly had `real`).

`double` is the name of the type. `numeric` is the name of the mode and also of the implicit class. As an S4 formal class, use `"numeric"`.

The potential confusion is that R has used *mode* `"numeric"` to mean 'double or integer', which conflicts with the S4 usage. Thus `is.numeric` tests the mode, not the class, but `as.numeric` (which is identical to `as.double`) coerces to the class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

https://en.wikipedia.org/wiki/IEEE_754-1985, https://en.wikipedia.org/wiki/IEEE_754-2008, https://en.wikipedia.org/wiki/IEEE_754-2019, https://en.wikipedia.org/wiki/Double_precision, https://en.wikipedia.org/wiki/Denormal_number.

See Also

`integer`, `numeric`, `storage.mode`.

Examples

```
is.double(1)
all(double(3) == 0)
```

---

dput *Write an Object to a File or Recreate it*

Description

Writes an ASCII text representation of an R object to a file, the R console, or a connection, or uses one to recreate the object.

Usage

```
dput(x, file = "",
     control = c("keepNA", "keepInteger", "niceNames", "showAttributes"))

dget(file, keep.source = FALSE)
```

Arguments

x	an object.
file	either a character string naming a file or a connection. "" indicates output to the console.
control	character vector (or NULL) of deparsing options. control = "all" is thorough, see .deparseOpts.
keep.source	logical: should the source formatting be retained when parsing functions, if possible?

Details

dput opens file and deparses the object x into that file. The object name is not written (unlike dump). If x is a function the associated environment is stripped. Hence scoping information can be lost.

Deparsing an object is difficult, and not always possible. With the default control, dput() attempts to deparse in a way that is readable, but for more complex or unusual objects (see dump), not likely to be parsed as identical to the original. Use control = "all" for the most complete deparsing; use control = NULL for the simplest deparsing, not even including attributes.

dput will warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

To display saved source rather than deparsing the internal representation include "useSource" in control. R currently saves source only for function definitions. If you do not care about source representation (e.g., for a data object), for speed set options(keep.source = FALSE) when calling source.

Value

For dput, the first argument invisibly.

For dget, the object created.

Note

This is **not** a good way to transfer objects between R sessions. dump is better, but the functions save and saveRDS are designed to be used for transporting R data, and will work with R objects that dput does not handle correctly as well as being much faster.

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be written as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

deparse, .deparseOpts, dump, write.

Examples

```
fil <- tempfile()
## Write an ASCII version of the 'base' function mean() to our temp file, ..
dput(base::mean, fil)
## ... read it back into 'bar' and confirm it is the same
bar <- dget(fil)
stopifnot(all.equal(bar, base::mean, check.environment = FALSE))

## Create a function with comments
baz <- function(x) {
  # Subtract from one
  1-x
}
## and display it
dput(baz)
## and now display the saved source
dput(baz, control = "useSource")

## Numeric values:
xx <- pi^(1:3)
dput(xx)
dput(xx, control = "digits17")
dput(xx, control = "hexNumeric")
dput(xx, fil); dget(fil) - xx # slight rounding on all platforms
dput(xx, fil, control = "digits17")
dget(fil) - xx # slight rounding on some platforms
dput(xx, fil, control = "hexNumeric"); dget(fil) - xx
unlink(fil)

xn <- setNames(xx, paste0("pi^",1:3))
dput(xn) # nicer, now "niceNames" being part of default 'control'
dput(xn, control = "S_compat") # no names
## explicitly asking for output as in R < 3.5.0:
dput(xn, control = c("keepNA", "keepInteger", "showAttributes"))
```

`drop` *Drop Redundant Extent Information*

Description

Delete the dimensions of an array which have only one level.

Usage

```
drop(x)
```

Arguments

`x`	an array (including a matrix).

Value

If x is an object with a dim attribute (e.g., a matrix or array), then drop returns an object like x, but with any extents of length one removed. Any accompanying dimnames attribute is adjusted and returned with x: if the result is a vector the names are taken from the dimnames (if any). If the result is a length-one vector, the names are taken from the first dimension with a dimname.

Array subsetting ([) performs this reduction unless used with drop = FALSE, but sometimes it is useful to invoke drop directly.

See Also

drop1 which is used for dropping terms in models, and droplevels used for dropping unused levels from a factor.

Examples

```
dim(drop(array(1:12, dim = c(1,3,1,1,2,1,2)))) # = 3 2 2
drop(1:3 %*% 2:4)  # scalar product
```

---

droplevels *Drop Unused Levels from Factors*

Description

The function droplevels is used to drop unused levels from a factor or, more commonly, from factors in a data frame.

Usage

```
droplevels(x, ...)
## S3 method for class 'factor'
droplevels(x, exclude = if(anyNA(levels(x))) NULL else NA, ...)
## S3 method for class 'data.frame'
droplevels(x, except, exclude, ...)
```

Arguments

x	an object from which to drop unused factor levels.
exclude	passed to factor(); factor levels which should be excluded from the result even if present. Note that this was *implicitly* NA in R <= 3.3.1 which did drop NA levels even when present in x, contrary to the documentation. The current default is compatible with x[, drop=TRUE].
...	further arguments passed to methods.
except	indices of columns from which *not* to drop levels.

Details

The method for class "factor" is currently equivalent to factor(x, exclude=exclude). For the data frame method, you should rarely specify exclude "globally" for all factor columns; rather the default uses the same factor-specific exclude as the factor method itself.

The except argument follows the usual indexing rules.

Value

droplevels returns an object of the same class as x

Note

This function was introduced in R 2.12.0. It is primarily intended for cases where one or more factors in a data frame contains only elements from a reduced level set after subsetting. (Notice that subsetting does *not* in general drop unused levels). By default, levels are dropped from all factors in a data frame, but the except argument allows you to specify columns for which this is not wanted.

See Also

subset for subsetting data frames. factor for definition of factors. drop for dropping array dimensions. drop1 for dropping terms from a model. [.factor for subsetting of factors.

Examples

```
aq <- transform(airquality, Month = factor(Month, labels = month.abb[5:9]))
aq <- subset(aq, Month != "Jul")
table(           aq $Month)
table(droplevels(aq)$Month)
```

---

dump *Text Representations of R Objects*

Description

This function takes a vector of names of R objects and produces text representations of the objects on a file or connection. A dump file can usually be sourced into another R session.

Usage

```
dump(list, file = "dumpdata.R", append = FALSE,
     control = "all", envir = parent.frame(), evaluate = TRUE)
```

Arguments

list	character vector (or NULL). The names of R objects to be dumped.
file	either a character string naming a file or a connection. "" indicates output to the console.
append	if TRUE and file is a character string, output will be appended to file; otherwise, it will overwrite the contents of file.
control	character vector (or NULL) indicating deparsing options. See .deparseOpts for their description.
envir	the environment to search for objects.
evaluate	logical. Should promises be evaluated?

Details

If some of the objects named do not exist (in scope), they are omitted, with a warning. If file is a file and no objects exist then no file is created.

sourceing may not produce an identical copy of dumped objects. A warning is issued if it is likely that problems will arise, for example when dumping exotic or complex objects (see the Note).

dump will also warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

A dump file can be sourced into another R (or perhaps S) session, but the functions save and saveRDS are designed to be used for transporting R data, and will work with R objects that dump does not handle. For maximal reproducibility use control = "exact".

To produce a more readable representation of an object, use control = NULL. This will skip attributes, and will make other simplifications that make source less likely to produce an identical copy. See .deparseOpts for details.

To deparse the internal representation of a function rather than displaying the saved source, use control = c("keepInteger", "warnIncomplete", "keepNA"). This will lose all formatting and comments, but may be useful in those cases where the saved source is no longer correct.

Promises will normally only be encountered by users as a result of lazy-loading (when the default evaluate = TRUE is essential) and after the use of delayedAssign, when evaluate = FALSE might be intended.

Value

An invisible character vector containing the names of the objects which were dumped.

Note

As dump is defined in the base namespace, the **base** package will be searched *before* the global environment unless dump is called from the top level prompt or the envir argument is given explicitly.

To avoid the risk of a source attribute becoming out of sync with the actual function definition, the source attribute of a function will never be dumped as an attribute.

Currently environments, external pointers, weak references and objects of type S4 are not deparsed in a way that can be sourced. In addition, language objects are deparsed in a simple way whatever the value of control, and this includes not dumping their attributes (which will result in a warning).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.deparseOpts` for available `control` settings; `dput()`, `dget()` and `deparse()` for related functions using identical internal deparsing functionality.

`write`, `write.table`, etc for "dumping" data to (text) files.

`save` and `saveRDS` for a more reliable way to save R objects.

Examples

```
x <- 1; y <- 1:10
fil <- tempfile(fileext=".Rdmped")
dump(ls(pattern = '^[xyz]'), fil)
print(.Last.value)
unlink(fil)
```

---

duplicated *Determine Duplicate Elements*

Description

`duplicated()` determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

`anyDuplicated(.)` is a "generalized" more efficient version `any(duplicated(.))`, returning positive integer indices instead of just `TRUE`.

Usage

```
duplicated(x, incomparables = FALSE, ...)

## Default S3 method:
duplicated(x, incomparables = FALSE,
           fromLast = FALSE, nmax = NA, ...)

## S3 method for class 'array'
duplicated(x, incomparables = FALSE, MARGIN = 1,
           fromLast = FALSE, ...)

anyDuplicated(x, incomparables = FALSE, ...)
## Default S3 method:
anyDuplicated(x, incomparables = FALSE,
           fromLast = FALSE, ...)
## S3 method for class 'array'
anyDuplicated(x, incomparables = FALSE,
           MARGIN = 1, fromLast = FALSE, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	a vector of values that cannot be compared. FALSE is a special value, meaning that all values can be compared, and may be the only value accepted for methods other than the default. It will be coerced internally to the same type as x.
fromLast	logical indicating if duplication should be considered from the reverse side, i.e., the last (or rightmost) of identical elements would correspond to duplicated = FALSE.
nmax	the maximum number of unique items expected (greater than one).
...	arguments for particular methods.
MARGIN	the array margin to be held fixed: see apply, and note that MARGIN = 0 may be useful.

Details

These are generic functions with methods for vectors (including lists), data frames and arrays (including matrices).

For the default methods, and whenever there are equivalent method definitions for duplicated and anyDuplicated, anyDuplicated(x, ...) is a "generalized" shortcut for any(duplicated(x, ...)), in the sense that it returns the *index* i of the first duplicated entry x[i] if there is one, and 0 otherwise. Their behaviours may be different when at least one of duplicated and anyDuplicated has a relevant method.

duplicated(x, fromLast = TRUE) is equivalent to but faster than rev(duplicated(rev(x))).

The array method calculates for each element of the sub-array specified by MARGIN if the remaining dimensions are identical to those for an earlier (or later, when fromLast = TRUE) element (in row-major order). This would most commonly be used to find duplicated rows (the default) or columns (with MARGIN = 2). Note that MARGIN = 0 returns an array of the same dimensionality attributes as x.

Missing values ("NA") are regarded as equal, numeric and complex ones differing from NaN; character strings will be compared in a "common encoding"; for details, see match (and unique) which use the same concept.

Values in incomparables will never be marked as duplicated. This is intended to be used for a fairly small set of values and will not be efficient for a very large set.

Except for factors, logical and raw vectors the default nmax = NA is equivalent to nmax = length(x). Since a hash table of size 8*nmax bytes is allocated, setting nmax suitably can save large amounts of memory. For factors it is automatically set to the smaller of length(x) and the number of levels plus one (for NA). If nmax is set too small there is liable to be an error: nmax = 1 is silently ignored.

Long vectors are supported for the default method of duplicated, but may only be usable if nmax is supplied.

Value

duplicated(): For a vector input, a logical vector of the same length as x. For a data frame, a logical vector with one element for each row. For a matrix or array, and when MARGIN = 0, a logical array with the same dimensions and dimnames.

anyDuplicated(): an integer or real vector of length one with value the 1-based index of the first duplicate if any, otherwise 0.

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see vector) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

unique.

Examples

```
x <- c(9:20, 1:5, 3:7, 0:8)
## extract unique elements
(xu <- x[!duplicated(x)])
## similar, same elements but different order:
(xu2 <- x[!duplicated(x, fromLast = TRUE)])

## xu == unique(x) but unique(x) is more efficient
stopifnot(identical(xu,  unique(x)),
          identical(xu2, unique(x, fromLast = TRUE)))

duplicated(iris)[140:143]

duplicated(iris3, MARGIN = c(1, 3))
anyDuplicated(iris) ## 143

anyDuplicated(x)
anyDuplicated(x, fromLast = TRUE)
```

---

dyn.load *Foreign Function Interface*

Description

Load or unload DLLs (also known as shared objects), and test whether a C function or Fortran subroutine is available.

Usage

```
dyn.load(x, local = TRUE, now = TRUE, ...)
dyn.unload(x)

is.loaded(symbol, PACKAGE = "", type = "")
```

Arguments

x	a character string giving the pathname to a DLL, also known as a dynamic shared object. (See 'Details' for what these terms mean.)
local	a logical value controlling whether the symbols in the DLL are stored in their own local table and not shared across DLLs, or added to the global symbol table. Whether this has any effect is system-dependent.
now	a logical controlling whether all symbols are resolved (and relocated) immediately when the library is loaded or deferred until they are used. This control is useful for developers testing whether a library is complete and has all the necessary symbols, and for users to ignore missing symbols. Whether this has any effect is system-dependent.
...	other arguments for future expansion.
symbol	a character string giving a symbol name.
PACKAGE	if supplied, confine the search for the name to the DLL given by this argument (plus the conventional extension, '.so', '.sl', '.dll', ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. This is used in the same way as in the .C, .Call, .Fortran and .External functions.
type	the type of symbol to look for: can be any ("", the default), "Fortran", "Call" or "External".

Details

The objects dyn.load loads are called 'dynamically loadable libraries' (abbreviated to 'DLL') on all platforms except macOS, which uses the term for a different sort of object. On Unix-alikes they are also called 'dynamic shared objects' ('DSO'), or 'shared objects' for short. (The POSIX standards use 'executable object file', but no one else does.)

See 'See Also' and the 'Writing R Extensions' and 'R Installation and Administration' manuals for how to create and install a suitable DLL.

Unfortunately some rare platforms (e.g., Compaq Tru64) do not handle the PACKAGE argument correctly, and may incorrectly find symbols linked into R.

The additional arguments to dyn.load mirror the different aspects of the mode argument to the dlopen() routine on POSIX systems. They are available so that users can exercise greater control over the loading process for an individual library. In general, the default values are appropriate and you should override them only if there is good reason and you understand the implications.

The local argument allows one to control whether the symbols in the DLL being attached are visible to other DLLs. While maintaining the symbols in their own namespace is good practice, the ability to share symbols across related 'chapters' is useful in many cases. Additionally, on certain platforms and versions of an operating system, certain libraries must have their symbols loaded globally to successfully resolve all symbols.

One should be careful of one potential side-effect of using lazy loading via now = FALSE: if a routine is called that has a missing symbol, the process will terminate immediately. The intended use is for library developers to call this with value TRUE to check that all symbols are actually resolved and for regular users to call it with FALSE so that missing symbols can be ignored and the available ones can be called.

The initial motivation for adding these was to avoid such termination in the `_init()` routines of the Java virtual machine library. However, symbols loaded locally may not be (read: probably) available to other DLLs. Those added to the global table are available to all other elements of the application and so can be shared across two different DLLs.

Some (very old) systems do not provide (explicit) support for local/global and lazy/eager symbol resolution. This can be the source of subtle bugs. One can arrange to have warning messages emitted when unsupported options are used. This is done by setting either of the options `verbose` or `warn` to be non-zero via the `options` function.

There is a short discussion of these additional arguments with some example code available at https://www.stat.ucdavis.edu/~duncan/R/dynload/.

Value

The function `dyn.load` is used for its side effect which links the specified DLL to the executing R image. Calls to `.C`, `.Call`, `.Fortran` and `.External` can then be used to execute compiled C functions or Fortran subroutines contained in the library. The return value of `dyn.load` is an object of class `DLLInfo`. See `getLoadedDLLs` for information about this class.

The function `dyn.unload` unlinks the DLL. Note that unloading a DLL and then re-loading a DLL of the same name may or may not work: on Solaris it used the first version loaded. Note also that some DLLs cannot be safely unloaded at all: unloading a DLL which implements C finalizers but does not unregister them on unload causes R to crash.

`is.loaded` checks if the symbol name is loaded *and searchable* and hence available for use as a character string value for argument `.NAME` in `.C`, `.Fortran`, `.Call`, or `.External`. It will succeed if any one of the four calling functions would succeed in using the entry point unless `type` is specified. (See `.Fortran` for how Fortran symbols are mapped.) Note that symbols in base packages are not searchable, and other packages can be so marked.

Warning

Do not use `dyn.unload` on a DLL loaded by `library.dynam`: use `library.dynam.unload`. This is needed for system housekeeping.

Note

`is.loaded` requires the name you would give to `.C` etc. It must be a character string and so cannot be an R object as used for registered native symbols (see "Writing R Extensions" section 5.4.). Some registered symbols are available by name but most are not, including those in the examples below.

By default, the maximum number of DLLs that can be loaded is now 614 when the OS limit on the number of open files allows or can be increased, but less otherwise (but it will be at least 100). A specific maximum can be requested *via* the environment variable `R_MAX_NUM_DLLS`, which has to be set (to a value between 100 and 1000 inclusive) before starting an R session. If the OS limit on the number of open files does not allow using this maximum and cannot be increased, R will fail to start with an error. The maximum is not allowed to be greater than 60% of the OS limit on the number of open files (essentially unlimited on Windows, on Unix typically 1024, but 256 on macOS). The limit can sometimes (including on macOS) be modified using command `ulimit -n` (`sh`, `bash`) or `limit descriptors` (`csh`) in the shell used to launch R. Increasing `R_MAX_NUM_DLLS` comes with some memory overhead, and be aware that many types of connections also use file descriptors.

If the OS limit on the number of open files cannot be determined, the DLL limit is 100 and cannot be changed *via* R_MAX_NUM_DLLS.

The creation of DLLs and the runtime linking of them into executing programs is very platform dependent. In recent years there has been some simplification in the process because the C subroutine call dlopen has become the POSIX standard for doing this. Under Unix-alikes dyn.load uses the dlopen mechanism and should work on all platforms which support it. On Windows it uses the standard mechanism (LoadLibrary) for loading DLLs.

The original code for loading DLLs in Unix-alikes was provided by Heiner Schwarte.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

library.dynam to be used inside a package's .onLoad initialization.

SHLIB for how to create suitable DLLs.

.C, .Fortran, .External, .Call.

Examples

```
## expect all of these to be false in R >= 3.0.0 as these can only be
## used via registered symbols.
is.loaded("supsmu") # Fortran entry point in stats
is.loaded("supsmu", "stats", "Fortran")
is.loaded("PDF", type = "External") # pdf() device in grDevices
```

---

eapply *Apply a Function Over Values in an Environment*

Description

eapply applies FUN to the named values from an environment and returns the results as a list. The user can request that all named objects are used (normally names that begin with a dot are not). The output is not sorted and no enclosing environments are searched.

Usage

```
eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)
```

Arguments

env	environment to be used.
FUN	the function to be applied, found *via* match.fun. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.
...	optional arguments to FUN.
all.names	a logical indicating whether to apply the function to all values.
USE.NAMES	logical indicating whether the resulting list should have names.

Value

A named (unless USE.NAMES = FALSE) list. Note that the order of the components is arbitrary for hashed environments.

See Also

environment, lapply.

Examples

```
require(stats)

env <- new.env(hash = FALSE) # so the order is fixed
env$a <- 1:10
env$beta <- exp(-3:3)
env$logic <- c(TRUE, FALSE, FALSE, TRUE)
# what have we there?
utils::ls.str(env)

# compute the mean for each list element
       eapply(env, mean)
unlist(eapply(env, mean, USE.NAMES = FALSE))

# median and quartiles for each element (making use of "..." passing):
eapply(env, quantile, probs = 1:3/4)
eapply(env, quantile)
```

---

eigen *Spectral Decomposition of a Matrix*

Description

Computes eigenvalues and eigenvectors of numeric (double, integer, logical) or complex matrices.

Usage

```
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)
```

Arguments

x	a numeric or complex matrix whose spectral decomposition is to be computed. Logical matrices are coerced to numeric.
symmetric	if TRUE, the matrix is assumed to be symmetric (or Hermitian if complex) and only its lower triangle (diagonal included) is used. If symmetric is not specified, isSymmetric(x) is used.
only.values	if TRUE, only the eigenvalues are computed and returned, otherwise both eigenvalues and eigenvectors are returned.
EISPACK	logical. Defunct and ignored.

Details

If symmetric is unspecified, isSymmetric(x) determines if the matrix is symmetric up to plausible numerical inaccuracies. It is surer and typically much faster to set the value yourself.

Computing the eigenvectors is the slow part for large matrices.

Computing the eigendecomposition of a matrix is subject to errors on a real-world computer: the definitive analysis is Wilkinson (1965). All you can hope for is a solution to a problem suitably close to x. So even though a real asymmetric x may have an algebraic solution with repeated real eigenvalues, the computed solution may be of a similar matrix with complex conjugate pairs of eigenvalues.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code (most often 1): these can only be interpreted by detailed study of the FORTRAN code.

Missing, NaN or infinite values in x will given an error.

Value

The spectral decomposition of x is returned as a list with components

values	a vector containing the p eigenvalues of x, sorted in *decreasing* order, according to Mod(values) in the asymmetric case when they might be complex (even for real matrices). For real asymmetric matrices the vector will be complex only if complex conjugate pairs of eigenvalues are detected.
vectors	either a $p \times p$ matrix whose columns contain the eigenvectors of x, or NULL if only.values is TRUE. The vectors are normalized to unit length. Recall that the eigenvectors are only defined up to a constant: even when the length is specified they are still only defined up to a scalar of modulus one (the sign for real matrices).

When only.values is not true, as by default, the result is of S3 class "eigen".

If r <- eigen(A), and V <- r$vectors; lam <- r$values, then

$$A = V \Lambda V^{-1}$$

(up to numerical fuzz), where Λ =diag(lam).

Source

eigen uses the LAPACK routines DSYEVR, DGEEV, ZHEEV and ZGEEV.

LAPACK is from https://netlib.org/lapack/ and its guide is listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Wilkinson, J. H. (1965) *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford.

See Also

svd, a generalization of eigen; qr, and chol for related decompositions.

To compute the determinant of a matrix, the qr decomposition is much more efficient: det.

Examples

```
eigen(cbind(c(1,-1), c(-1,1)))
eigen(cbind(c(1,-1), c(-1,1)), symmetric = FALSE)
# same (different algorithm).

eigen(cbind(1, c(1,-1)), only.values = TRUE)
eigen(cbind(-1, 2:1)) # complex values
eigen(print(cbind(c(0, 1i), c(-1i, 0)))) # Hermite ==> real Eigenvalues
## 3 x 3:
eigen(cbind( 1, 3:1, 1:3))
eigen(cbind(-1, c(1:2,0), 0:2)) # complex values
```

---

encodeString *Encode Character Vector as for Printing*

Description

encodeString escapes the strings in a character vector in the same way print.default does, and optionally fits the encoded strings within a field width.

Usage

```
encodeString(x, width = 0, quote = "", na.encode = TRUE,
             justify = c("left", "right", "centre", "none"))
```

Arguments

x	a character vector, or an object that can be coerced to one by as.character.
width	integer: the minimum field width. If NULL or NA, this is taken to be the largest field width needed for any element of x.
quote	character: quoting character, if any.
na.encode	logical: should NA strings be encoded?
justify	character: partial matches are allowed. If padding to the minimum field width is needed, how should spaces be inserted? justify == "none" is equivalent to width = 0, for consistency with format.default.

Details

This escapes backslash and the control characters '\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (line feed, aka "newline"), '\r' (carriage return), '\t' (tab) and '\v' (vertical tab) as well as any non-printable characters in a single-byte locale, which are printed in octal notation ('\xyz' with leading zeroes).

Which characters are non-printable depends on the current locale. Windows' reporting of printable characters is unreliable, so there all other control characters are regarded as non-printable, and all characters with codes 32–255 as printable in a single-byte locale. See print.default for how non-printable characters are handled in multi-byte locales.

If quote is a single or double quote any embedded quote of the same type is escaped. Note that justification is of the quoted string, hence spaces are added outside the quotes.

Value

A character vector of the same length as x, with the same attributes (including names and dimensions) but with no class set.

Marked UTF-8 encodings are preserved.

Note

The default for width is different from format.default, which does similar things for character vectors but without encoding using escapes.

See Also

print.default

Examples

```
x <- "ab\bc\ndef"
print(x)
cat(x) # interprets escapes
cat(encodeString(x), "\n", sep = "") # similar to print()

factor(x) # makes use of this to print the levels
```

```
x <- c("a", "ab", "abcde")
encodeString(x) # width = 0: use as little as possible
encodeString(x, 2) # use two or more (left justified)
encodeString(x, width = NA) # left justification
encodeString(x, width = NA, justify = "c")
encodeString(x, width = NA, justify = "r")
encodeString(x, width = NA, quote = "'", justify = "r")
```

`Encoding` *Read or Set the Declared Encodings for a Character Vector*

Description

Read or set the declared encodings for a character vector.

Usage

```
Encoding(x)

Encoding(x) <- value

enc2native(x)
enc2utf8(x)
```

Arguments

`x`	A character vector.
`value`	A character vector of positive length.

Details

Character strings in R can be declared to be encoded in `"latin1"` or `"UTF-8"` or as `"bytes"`. These declarations can be read by `Encoding`, which will return a character vector of values `"latin1"`, `"UTF-8"` `"bytes"` or `"unknown"`, or set, when `value` is recycled as needed and other values are silently treated as `"unknown"`. ASCII strings will never be marked with a declared encoding, since their representation is the same in all supported encodings. Strings marked as `"bytes"` are intended to be non-ASCII strings which should be manipulated as bytes, and never converted to a character encoding (so writing them to a text file is supported only by `writeLines(useBytes = TRUE)`).

`enc2native` and `enc2utf8` convert elements of character vectors to the native encoding or UTF-8 respectively, taking any marked encoding into account. They are primitive functions, designed to do minimal copying.

There are other ways for character strings to acquire a declared encoding apart from explicitly setting it (and these have changed as R has evolved). The parser marks strings containing '\u' or '\U' escapes. Functions `scan`, `read.table`, `readLines`, and `parse` have an `encoding` argument that is used to declare encodings, `iconv` declares encodings from its `to` argument, and console input in suitable locales is also declared. `intToUtf8` declares its output as `"UTF-8"`, and output text connections (see `textConnection`) are marked if running in a suitable locale. Under some

circumstances (see its help page) `source(encoding=)` will mark encodings of character strings it outputs.

Most character manipulation functions will set the encoding on output strings if it was declared on the corresponding input. These include `chartr`, `strsplit(useBytes = FALSE)`, `tolower` and `toupper` as well as `sub(useBytes = FALSE)` and `gsub(useBytes = FALSE)`. Note that such functions do not *preserve* the encoding, but if they know the input encoding and that the string has been successfully re-encoded (to the current encoding or UTF-8), they mark the output.

`substr` does preserve the encoding, and `chartr`, `tolower` and `toupper` preserve UTF-8 encoding on systems with Unicode wide characters. With their `fixed` and `perl` options, `strsplit`, `sub` and `gsub` will give a marked UTF-8 result if any of the inputs are UTF-8.

`paste` and `sprintf` return elements marked as bytes if any of the corresponding inputs is marked as bytes, and otherwise marked as UTF-8 if any of the inputs is marked as UTF-8.

`match`, `pmatch`, `charmatch`, `duplicated` and `unique` all match in UTF-8 if any of the elements are marked as UTF-8.

Changing the current encoding from a running R session may lead to confusion (see `Sys.setlocale`).

There is some ambiguity as to what is meant by a 'Latin-1' locale, since some OSes (notably Windows) make use of character positions undefined (or used for control characters) in the ISO 8859-1 character set. How such characters are interpreted is system-dependent but as from R 3.5.0 they are if possible interpreted as per Windows codepage 1252 (which Microsoft calls 'Windows Latin 1 (ANSI)') when converting to e.g. UTF-8.

Value

A character vector.

For `enc2utf8` encodings are always marked: they are for `enc2native` in UTF-8 and Latin-1 locales.

Examples

```
## x is intended to be in latin1
x. <- x <- "fran\xE7ais"
Encoding(x.) # "unknown" (UTF-8 loc.) | "latin1" (8859-1/CP-1252 loc.) | ....
Encoding(x) <- "latin1"
x
xx <- iconv(x, "latin1", "UTF-8")
Encoding(c(x., x, xx))
c(x, xx)
xb <- xx; Encoding(xb) <- "bytes"
xb # will be encoded in hex
cat("x = ", x, ", xx = ", xx, ", xb = ", xb, "\n", sep = "")
(Ex <- Encoding(c(x.,x,xx,xb)))
stopifnot(identical(Ex, c(Encoding(x.), Encoding(x),
                          Encoding(xx), Encoding(xb))))
```

environment *Environment Access*

Description

Get, set, test for and create environments.

Usage

```
environment(fun = NULL)
environment(fun) <- value

is.environment(x)

.GlobalEnv
globalenv()
.BaseNamespaceEnv

emptyenv()
baseenv()

new.env(hash = TRUE, parent = parent.frame(), size = 29L)

parent.env(env)
parent.env(env) <- value

environmentName(env)

env.profile(env)
```

Arguments

fun	a function, a formula, or NULL, which is the default.
value	an environment to associate with the function.
x	an arbitrary R object.
hash	a logical, if TRUE the environment will use a hash table.
parent	an environment to be used as the enclosure of the environment created.
env	an environment.
size	an integer specifying the initial size for a hashed environment. An internal default value will be used if size is NA or zero. This argument is ignored if hash is FALSE.

Details

Environments consist of a *frame*, or collection of named objects, and a pointer to an *enclosing environment*. The most common example is the frame of variables local to a function call; its *enclosure* is the environment where the function was defined (unless changed subsequently). The enclosing environment is distinguished from the *parent frame*: the latter (returned by `parent.frame`) refers to the environment of the caller of a function. Since confusion is so easy, it is best never to use 'parent' in connection with an environment (despite the presence of the function `parent.env`).

When `get` or `exists` search an environment with the default `inherits = TRUE`, they look for the variable in the frame, then in the enclosing frame, and so on.

The global environment `.GlobalEnv`, more often known as the user's workspace, is the first item on the search path. It can also be accessed by `globalenv()`. On the search path, each item's enclosure is the next item.

The object `.BaseNamespaceEnv` is the namespace environment for the base package. The environment of the base package itself is available as `baseenv()`.

If one follows the chain of enclosures found by repeatedly calling `parent.env` from any environment, eventually one reaches the empty environment `emptyenv()`, into which nothing may be assigned.

The replacement function `parent.env<-` is extremely dangerous as it can be used to destructively change environments in ways that violate assumptions made by the internal C code. It may be removed in the near future.

The replacement form of `environment`, `is.environment`, `baseenv`, `emptyenv` and `globalenv` are primitive functions.

System environments, such as the base, global and empty environments, have names as do the package and namespace environments and those generated by `attach()`. Other environments can be named by giving a `"name"` attribute, but this needs to be done with care as environments have unusual copying semantics.

Value

If `fun` is a function or a formula then `environment(fun)` returns the environment associated with that function or formula. If `fun` is `NULL` then the current evaluation environment is returned.

The replacement form sets the environment of the function or formula `fun` to the `value` given.

`is.environment(obj)` returns `TRUE` if and only if `obj` is an `environment`.

`new.env` returns a new (empty) environment with (by default) enclosure the parent frame.

`parent.env` returns the enclosing environment of its argument.

`parent.env<-` sets the enclosing environment of its first argument.

`environmentName` returns a character string, that given when the environment is printed or `""` if it is not a named environment.

`env.profile` returns a list with the following components: `size` the number of chains that can be stored in the hash table, `nchains` the number of non-empty chains in the table (as reported by `HASHPRI`), and `counts` an integer vector giving the length of each chain (zero for empty chains). This function is intended to assess the performance of hashed environments. When `env` is a non-hashed environment, `NULL` is returned.

See Also

For the performance implications of hashing or not, see https://en.wikipedia.org/wiki/Hash_table.

The `envir` argument of `eval`, `get`, and `exists`.

`ls` may be used to view the objects in an environment, and hence `ls.str` may be useful for an overview.

`sys.source` can be used to populate an environment.

Examples

```
f <- function() "top level function"

##-- all three give the same:
environment()
environment(f)
.GlobalEnv

ls(envir = environment(stats::approxfun(1:2, 1:2, method = "const")))

is.environment(.GlobalEnv) # TRUE

e1 <- new.env(parent = baseenv())  # this one has enclosure package:base.
e2 <- new.env(parent = e1)
assign("a", 3, envir = e1)
ls(e1)
ls(e2)
exists("a", envir = e2)   # this succeeds by inheritance
exists("a", envir = e2, inherits = FALSE)
exists("+", envir = e2)   # this succeeds by inheritance

eh <- new.env(hash = TRUE, size = NA)
with(env.profile(eh), stopifnot(size == length(counts)))
```

EnvVar *Environment Variables*

Description

Details of some of the environment variables which affect an R session.

Details

It is impossible to list all the environment variables which can affect an R session: some affect the OS system functions which R uses, and others will affect add-on packages. But here are notes on some of the more important ones. Those that set the defaults for options are consulted only at startup (as are some of the others).

`HOME`: The user's 'home' directory.

LANGUAGE: Optional. The language(s) to be used for message translations. This is consulted when needed.

LC_ALL: (etc) Optional. Use to set various aspects of the locale – see Sys.getlocale. Consulted at startup.

MAKEINDEX: The path to makeindex. If unset to a value determined when R was built. Used by the emulation mode of texi2dvi and texi2pdf.

R_BATCH: Optional – set in a batch session, that is one started by R CMD BATCH. Most often set to "", so test by something like !is.na(Sys.getenv("R_BATCH", NA)).

R_BROWSER: The path to the default browser. Used to set the default value of options("browser").

R_COMPLETION: Optional. If set to FALSE, command-line completion is not used. (Not used by the macOS GUI.)

R_DEFAULT_PACKAGES: A comma-separated list of packages which are to be attached in every session. See options.

R_DOC_DIR: The location of the R 'doc' directory. Set by R.

R_ENVIRON: Optional. The path to the site environment file: see Startup. Consulted at startup.

R_GSCMD: Optional. The path to Ghostscript, used by dev2bitmap, bitmap and embedFonts. Consulted when those functions are invoked. Since it will be treated as if passed to system, spaces and shell metacharacters should be escaped.

R_HISTFILE: Optional. The path of the history file: see Startup. Consulted at startup and when the history is saved.

R_HISTSIZE: Optional. The maximum size of the history file, in lines. Exactly how this is used depends on the interface.

On Unix-alikes, for the readline command-line interface it takes effect when the history is saved (by savehistory or at the end of a session).

On Windows, for Rgui it controls the number of lines saved to the history file: the size of the history used in the session is controlled by the console customization: see Rconsole.

R_HOME: The top-level directory of the R installation: see R.home. Set by R.

R_INCLUDE_DIR: The location of the R 'include' directory. Set by R.

R_LIBS: Optional. Used for initial setting of .libPaths.

R_LIBS_SITE: Optional. Used for initial setting of .libPaths.

R_LIBS_USER: Optional. Used for initial setting of .libPaths.

R_PAPERSIZE: Optional. Used to set the default for options("papersize"), e.g. used by pdf and postscript.

R_PCRE_JIT_STACK_MAXSIZE: Optional. Consulted when PCRE's JIT pattern compiler is first used. See grep.

R_PDFVIEWER: The path to the default PDF viewer. Used by R CMD Rd2pdf.

R_PLATFORM: The platform – a string of the form *"cpu-vendor-os"*, see R.Version.

R_PROFILE: Optional. The path to the site profile file: see Startup. Consulted at startup.

R_RD4PDF: Options for pdflatex processing of Rd files. Used by R CMD Rd2pdf.

R_SHARE_DIR: The location of the R 'share' directory. Set by R.

R_TEXI2DVICMD: The path to texi2dvi. Defaults to the value of TEXI2DVI, and if that is unset to a value determined when R was built.

Only on Unix-alikes:
Consulted at startup to set the default for options("texi2dvi"), used by texi2dvi and texi2pdf in package **tools**.

R_TIDYCMD: The path to HTML tidy. Used by R CMD check if _R_CHECK_RD_VALIDATE_RD2HTML_ is set to a true value (as it is by '--as-cran'.

R_UNZIPCMD: The path to unzip. Sets the initial value for options("unzip") on a Unix-alike when namespace **utils** is loaded.

R_ZIPCMD: The path to zip. Used by zip and by R CMD INSTALL --build on Windows.

TMPDIR, TMP, TEMP: Consulted (in that order) when setting the temporary directory for the session: see tempdir. TMPDIR is also used by some of the utilities: see the help for build.

TZ: Optional. The current time zone. See Sys.timezone for the system-specific formats. Consulted as needed.

TZDIR: Optional. The top-level directory of the time-zone database. See Sys.timezone.

no_proxy, http_proxy, ftp_proxy: (and more). Optional. Settings for download.file: see its help for further details.

Unix-specific

Some variables set on Unix-alikes, and not (in general) on Windows.

DISPLAY: Optional: used by X11, Tk (in package **tcltk**), the data editor and various packages.

EDITOR: The path to the default editor: sets the default for options("editor") when namespace **utils** is loaded.

PAGER: The path to the pager with the default setting of options("pager"). The default value is chosen at configuration, usually as the path to less.

R_PRINTCMD: Sets the default for options("printcmd"), which sets the default print command to be used by postscript.

R_SUPPORT_OLD_TARS logical. Sets the default for the support_old_tars argument of untar. Should be set to TRUE if an old system tar command is used which does not support either xz compression or automagically detecting compression type.

Windows-specific

Some Windows-specific variables are

GSC: Optional: the path to Ghostscript, used if R_GSCMD is not set.

R_USER: The user's 'home' directory. Set by R. (HOME will be set to the same value if not already set.)

See Also

Sys.getenv and Sys.setenv to read and set environmental variables in an R session.

gctorture for environment variables controlling garbage collection.

`eval` *Evaluate an (Unevaluated) Expression*

Description

Evaluate an R expression in a specified environment.

Usage

```
eval(expr, envir = parent.frame(),
           enclos = if(is.list(envir) || is.pairlist(envir))
                       parent.frame() else baseenv())
evalq(expr, envir, enclos)
eval.parent(expr, n = 1)
local(expr, envir = new.env())
```

Arguments

`expr`	an object to be evaluated. See 'Details'.
`envir`	the `environment` in which `expr` is to be evaluated. May also be `NULL`, a list, a data frame, a pairlist or an integer as specified to `sys.call`.
`enclos`	relevant when `envir` is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in `envir`. This can be `NULL` (interpreted as the base package environment, `baseenv()`) or an environment.
`n`	number of parent generations to go back.

Details

`eval` evaluates the `expr` argument in the environment specified by `envir` and returns the computed value. If `envir` is not specified, then the default is `parent.frame()` (the environment where the call to `eval` was made).

Objects to be evaluated can be of types `call` or `expression` or `name` (when the name is looked up in the current scope and its binding is evaluated), a promise or any of the basic types such as vectors, functions and environments (which are returned unchanged).

The `evalq` form is equivalent to `eval(quote(expr), ...)`. `eval` evaluates its first argument in the current scope before passing it to the evaluator: `evalq` avoids this.

`eval.parent(expr, n)` is a shorthand for `eval(expr, parent.frame(n))`.

If `envir` is a list (such as a data frame) or pairlist, it is copied into a temporary environment (with enclosure `enclos`), and the temporary environment is used for evaluation. So if `expr` changes any of the components named in the (pair)list, the changes are lost.

If `envir` is `NULL` it is interpreted as an empty list so no values could be found in `envir` and look-up goes directly to `enclos`.

`local` evaluates an expression in a local environment. It is equivalent to `evalq` except that its default argument creates a new, empty environment. This is useful to create anonymous recursive functions and as a kind of limited namespace feature since variables defined in the environment are not visible from the outside.

Value

The result of evaluating the object: for an expression vector this is the result of evaluating the last element.

Note

Due to the difference in scoping rules, there are some differences between R and S in this area. In particular, the default enclosure in S is the global environment.

When evaluating expressions in a data frame that has been passed as an argument to a function, the relevant enclosure is often the caller's environment, i.e., one needs `eval(x, data, parent.frame())`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`eval` only.)

See Also

`expression`, `quote`, `sys.frame`, `parent.frame`, `environment`.

Further, `force` to *force* evaluation, typically of function arguments.

Examples

```
eval(2 ^ 2 ^ 3)
mEx <- expression(2^2^3); mEx; 1 + eval(mEx)
eval({ xx <- pi; xx^2}) ; xx

a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, list(a = 1)), list(b = 5)) # == 10
a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, -1), list(b = 5))        # == 12

ev <- function() {
   e1 <- parent.frame()
   ## Evaluate a in e1
   aa <- eval(expression(a), e1)
   ## evaluate the expression bound to a in e1
   a <- expression(x+y)
   list(aa = aa, eval = eval(a, e1))
}
tst.ev <- function(a = 7) { x <- pi; y <- 1; ev() }
tst.ev()  #-> aa : 7,  eval : 4.14

a <- list(a = 3, b = 4)
with(a, a <- 5) # alters the copy of a from the list, discarded.

##
## Example of evalq()
##

N <- 3
```

```
env <- new.env()
assign("N", 27, envir = env)
## this version changes the visible copy of N only, since the argument
## passed to eval is '4'.
eval(N <- 4, env)
N
get("N", envir = env)
## this version does the assignment in env, and changes N only there.
evalq(N <- 5, env)
N
get("N", envir = env)


##
## Uses of local()
##

# Mutually recursive.
# gg gets value of last assignment, an anonymous version of f.

gg <- local({
    k <- function(y)f(y)
    f <- function(x) if(x) x*k(x-1) else 1
})
gg(10)
sapply(1:5, gg)

# Nesting locals: a is private storage accessible to k
gg <- local({
    k <- local({
        a <- 1
        function(y){print(a <<- a+1);f(y)}
    })
    f <- function(x) if(x) x*k(x-1) else 1
})
sapply(1:5, gg)

ls(envir = environment(gg))
ls(envir = environment(get("k", envir = environment(gg))))
```

`exists` *Is an Object Defined?*

Description

Look for an R object of the given name and possibly return it

Usage

```
exists(x, where = -1, envir = , frame, mode = "any",
```

```
        inherits = TRUE)

get0(x, envir = pos.to.env(-1L), mode = "any", inherits = TRUE,
     ifnotfound = NULL)
```

Arguments

x	a variable name (given as a character string or a symbol).
where	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.
envir	an alternative way to specify an environment to look in, but it is usually simpler to just use the where argument.
frame	a frame in the calling list. Equivalent to giving where as sys.frame(frame).
mode	the mode or type of object sought: see the 'Details' section.
inherits	should the enclosing frames of the environment be searched?
ifnotfound	the return value of get0(x, *) when x does not exist.

Details

The where argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The envir argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name x has a value bound to it in the specified environment. If inherits is TRUE and a value is not found for x in the specified environment, the enclosing frames of the environment are searched until the name x is encountered. See environment and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

Warning: inherits = TRUE is the default behaviour for R but not for S.

If mode is specified then only objects of that type are sought. The mode may specify one of the collections "numeric" and "function" (see mode): any member of the collection will suffice. (This is true even if a member of a collection is specified, so for example mode = "special" will seek any type of function.)

Value

exists(): Logical, true if and only if an object of the correct name and mode is found.

get0(): The object—as from get(x, *)— if exists(x, *) is true, otherwise ifnotfound.

Note

With get0(), instead of the easy to read but somewhat inefficient

```
if (exists(myVarName, envir = myEnvir)) {
  r <- get(myVarName, envir = myEnvir)
  ## ... deal with r ...
}
```

you now can use the more efficient (and slightly harder to read)

```
if (!is.null(r <- get0(myVarName, envir = myEnvir))) {
  ## ... deal with r ...
}
```

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`get` and `hasName`. For quite a different kind of "existence" checking, namely if function arguments were specified, `missing`; and for yet a different kind, namely if a file exists, `file.exists`.

Examples

```
##  Define a substitute function if necessary:
if(!exists("some.fun", mode = "function"))
  some.fun <- function(x) { cat("some.fun(x)\n"); x }
search()
exists("ls", 2) # true even though ls is in pos = 3
exists("ls", 2, inherits = FALSE) # false

## These are true (in most circumstances):
identical(ls,   get0("ls"))
identical(NULL, get0(".foo.bar.")) # default ifnotfound = NULL (!)
```

---

expand.grid *Create a Data Frame from All Combinations of Factor Variables*

Description

Create a data frame from all combinations of the supplied vectors or factors. See the description of the return value for precise details of the way this is done.

Usage

```
expand.grid(..., KEEP.OUT.ATTRS = TRUE, stringsAsFactors = TRUE)
```

Arguments

`...`	vectors, factors or a list containing these.
`KEEP.OUT.ATTRS`	a logical indicating the `"out.attrs"` attribute (see below) should be computed and returned.
`stringsAsFactors`	logical specifying if character vectors are converted to factors.

Value

A data frame containing one row for each combination of the supplied factors. The first factors vary fastest. The columns are labelled by the factors if these are supplied as named arguments or named components of a list. The row names are 'automatic'.

Attribute `"out.attrs"` is a list which gives the dimension and dimnames for use by `predict` methods.

Note

Conversion to a factor is done with levels in the order they occur in the character vectors (and not alphabetically, as is most common when converting to factors).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`combn` (package `utils`) for the generation of all combinations of n elements, taken m at a time.

Examples

```
require(utils)

expand.grid(height = seq(60, 80, 5), weight = seq(100, 300, 50),
            sex = c("Male","Female"))

x <- seq(0, 10, length.out = 100)
y <- seq(-1, 1, length.out = 20)
d1 <- expand.grid(x = x, y = y)
d2 <- expand.grid(x = x, y = y, KEEP.OUT.ATTRS = FALSE)
object.size(d1) - object.size(d2)
##-> 5992 or 8832 (on 32- / 64-bit platform)
```

`expression` *Unevaluated Expressions*

Description

Creates or tests for objects of mode and class `"expression"`.

Usage

```
expression(...)

is.expression(x)
as.expression(x, ...)
```

Arguments

...	expression: R objects, typically calls, symbols or constants. as.expression: arguments to be passed to methods.
x	an arbitrary R object.

Details

'Expression' here is not being used in its colloquial sense, that of mathematical expressions. Those are calls (see call) in R, and an R expression vector is a list of calls, symbols etc, for example as returned by parse.

As an object of mode "expression" is a list, it can be subsetted by [, [[or $, the latter two extracting individual calls etc. The replacement forms of these operators can be used to replace or delete elements.

expression and is.expression are primitive functions. expression is 'special': it does not evaluate its arguments.

Value

expression returns a vector of type "expression" containing its arguments (unevaluated).

is.expression returns TRUE if expr is an expression object and FALSE otherwise.

as.expression attempts to coerce its argument into an expression object. It is generic, and only the default method is described here. (The default method calls as.vector(type = "expression") and so may dispatch methods for as.vector.) NULL, calls, symbols (see as.symbol) and pairlists are returned as the element of a length-one expression vector. Atomic vectors are placed element-by-element into an expression vector (without using any names): lists have their type (typeof) changed to an expression vector (keeping all attributes). Other types are not currently supported.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

call, eval, function. Further, text, legend, and plotmath for plotting mathematical expressions.

Examples

```
length(ex1 <- expression(1 + 0:9)) # 1
ex1
eval(ex1) # 1:10

length(ex3 <- expression(u, 2, u + 0:9)) # 3
mode(ex3 [3])   # expression
mode(ex3[[3]])  # call
## but not all components are 'call's :
sapply(ex3, mode  ) #  name  numeric  call
```

```
sapply(ex3, typeof) # symbol  double  language
rm(ex3)
```

`Extract` *Extract or Replace Parts of an Object*

Description

Operators acting on vectors, matrices, arrays and lists to extract or replace parts.

Usage

```
x[i]
x[i, j, ... , drop = TRUE]
x[[i, exact = TRUE]]
x[[i, j, ..., exact = TRUE]]
x$name
getElement(object, name)

x[i] <- value
x[i, j, ...] <- value
x[[i]] <- value
x$name <- value
```

Arguments

`x`, `object`	object from which to extract element(s) or in which to replace element(s).
`i`, `j`, ...	indices specifying elements to extract or replace. Indices are `numeric` or `character` vectors or empty (missing) or `NULL`. Numeric values are coerced to integer or whole numbers as by `as.integer` or for large values by `trunc` (and hence truncated towards zero). Character vectors will be matched to the `names` of the object (or for matrices/arrays, the `dimnames`): see 'Character indices' below for further details. For `[`-indexing only: `i`, `j`, ... can be logical vectors, indicating elements/slices to select. Such vectors are recycled if necessary to match the corresponding extent. `i`, `j`, ... can also be negative integers, indicating elements/slices to leave out of the selection. When indexing arrays by `[` a single argument `i` can be a matrix with as many columns as there are dimensions of `x`; the result is then a vector with elements corresponding to the sets of indices in each row of `i`. An index value of `NULL` is treated as if it were `integer(0)`.
`name`	a literal character string or a name (possibly backtick quoted). For extraction, this is normally (see under 'Environments') partially matched to the `names` of the object.
`drop`	relevant for matrices and arrays. If `TRUE` the result is coerced to the lowest possible dimension (see the examples). This only works for extracting elements, not for the replacement. See `drop` for further details.

`exact` controls possible partial matching of `[[` when extracting by a character vector (for most objects, but see under 'Environments'). The default is no partial matching. Value `NA` allows partial matching but issues a warning when it occurs. Value `FALSE` allows partial matching without any warning.

`value` typically an array-like R object of a similar class as `x`.

Details

These operators are generic. You can write methods to handle indexing of specific classes of objects, see InternalMethods as well as `[.data.frame` and `[.factor`. The descriptions here apply only to the default methods. Note that separate methods are required for the replacement functions `[<-`, `[[<-` and `$<-` for use when indexing occurs on the assignment side of an expression.

The most important distinction between `[`, `[[` and `$` is that the `[` can select more than one element whereas the other two select a single element.

Note that `x[[]]` is always erroneous.

The default methods work somewhat differently for atomic vectors, matrices/arrays and for recursive (list-like, see `is.recursive`) objects. `$` is only valid for recursive objects (and `NULL`), and is only discussed in the section below on recursive objects.

Subsetting (except by an empty index) will drop all attributes except `names`, `dim` and `dimnames`.

Indexing can occur on the right-hand-side of an expression for extraction, or on the left-hand-side for replacement. When an index expression appears on the left side of an assignment (known as *subassignment*) then that part of `x` is set to the value of the right hand side of the assignment. In this case no partial matching of character indices is done, and the left-hand-side is coerced as needed to accept the values. For vectors, the answer will be of the higher of the types of `x` and `value` in the hierarchy raw < logical < integer < double < complex < character < list < expression. Attributes are preserved (although `names`, `dim` and `dimnames` will be adjusted suitably). Subassignment is done sequentially, so if an index is specified more than once the latest assigned value for an index will result.

It is an error to apply any of these operators to an object which is not subsettable (e.g., a function).

Atomic vectors

The usual form of indexing is `[`. `[[` can be used to select a single element *dropping* `names`, whereas `[` keeps them, e.g., in `c(abc = 123)[1]`.

The index object `i` can be numeric, logical, character or empty. Indexing by factors is allowed and is equivalent to indexing by the numeric codes (see `factor`) and not by the character values which are printed (for which use `[as.character(i)]`).

An empty index selects all values: this is most often used to replace all the entries but keep the `attributes`.

Matrices and arrays

Matrices and arrays are vectors with a dimension attribute and so all the vector forms of indexing can be used with a single index. The result will be an unnamed vector unless `x` is one-dimensional when it will be a one-dimensional array.

The most common form of indexing a k-dimensional array is to specify k indices to `[`. As for vector indexing, the indices can be numeric, logical, character, empty or even factor. And again, indexing by factors is equivalent to indexing by the numeric codes, see 'Atomic vectors' above.

An empty index (a comma separated blank) indicates that all entries in that dimension are selected. The argument `drop` applies to this form of indexing.

A third form of indexing is via a numeric matrix with the one column for each dimension: each row of the index matrix then selects a single element of the array, and the result is a vector. Negative indices are not allowed in the index matrix. `NA` and zero values are allowed: rows of an index matrix containing a zero are ignored, whereas rows containing an `NA` produce an `NA` in the result.

Indexing via a character matrix with one column per dimensions is also supported if the array has dimension names. As with numeric matrix indexing, each row of the index matrix selects a single element of the array. Indices are matched against the appropriate dimension names. `NA` is allowed and will produce an `NA` in the result. Unmatched indices as well as the empty string (`""`) are not allowed and will result in an error.

A vector obtained by matrix indexing will be unnamed unless `x` is one-dimensional when the row names (if any) will be indexed to provide names for the result.

Recursive (list-like) objects

Indexing by `[` is similar to atomic vectors and selects a list of the specified element(s).

Both `[[` and `$` select a single element of the list. The main difference is that `$` does not allow computed indices, whereas `[[` does. `x$name` is equivalent to `x[["name", exact = FALSE]]`. Also, the partial matching behavior of `[[` can be controlled using the `exact` argument.

`getElement(x, name)` is a version of `x[[name, exact = TRUE]]` which for formally classed (S4) objects returns `slot(x, name)`, hence providing access to even more general list-like objects.

`[` and `[[` are sometimes applied to other recursive objects such as calls and expressions. Pairlists (such as calls) are coerced to lists for extraction by `[`, but all three operators can be used for replacement.

`[[` can be applied recursively to lists, so that if the single index `i` is a vector of length `p`, `alist[[i]]` is equivalent to `alist[[i1]]...[[ip]]` providing all but the final indexing results in a list.

Note that in all three kinds of replacement, a value of `NULL` deletes the corresponding item of the list. To set entries to `NULL`, you need `x[i] <- list(NULL)`.

When `$<-` is applied to a `NULL` `x`, it first coerces `x` to `list()`. This is what also happens with `[[<-` where in R versions less than 4.y.z, a length one value resulted in a length one (atomic) *vector*.

Environments

Both `$` and `[[` can be applied to environments. Only character indices are allowed and no partial matching is done. The semantics of these operations are those of `get(i, env = x, inherits = FALSE)`. If no match is found then `NULL` is returned. The replacement versions, `$<-` and `[[<-`, can also be used. Again, only character arguments are allowed. The semantics in this case are those of `assign(i, value, env = x, inherits = FALSE)`. Such an assignment will either create a new binding or change the existing binding in `x`.

NAs in indexing

When extracting, a numerical, logical or character `NA` index picks an unknown element and so returns `NA` in the corresponding element of a logical, integer, numeric, complex or character result, and `NULL` for a list. (It returns `00` for a raw result.)

When replacing (that is using indexing on the lhs of an assignment) `NA` does not select any element to be replaced. As there is ambiguity as to whether an element of the rhs should be used or not, this is only allowed if the rhs value is of length one (so the two interpretations would have the same outcome). (The documented behaviour of S was that an `NA` replacement index 'goes nowhere' but uses up an element of `value`: Becker et al. p. 359. However, that has not been true of other implementations.)

Argument matching

Note that these operations do not match their index arguments in the standard way: argument names are ignored and positional matching only is used. So `m[j = 2, i = 1]` is equivalent to `m[2, 1]` and **not** to `m[1, 2]`.

This may not be true for methods defined for them; for example it is not true for the `data.frame` methods described in `[.data.frame` which warn if `i` or `j` is named and have undocumented behaviour in that case.

To avoid confusion, do not name index arguments (but `drop` and `exact` must be named).

S4 methods

These operators are also implicit S4 generics, but as primitives, S4 methods will be dispatched only on S4 objects `x`.

The implicit generics for the `$` and `$<-` operators do not have `name` in their signature because the grammar only allows symbols or string constants for the `name` argument.

Character indices

Character indices can in some circumstances be partially matched (see `pmatch`) to the names or dimnames of the object being subsetted (but never for subassignment). Unlike S (Becker et al. p. 358), R never uses partial matching when extracting by `[`, and partial matching is not by default used by `[[` (see argument `exact`).

Thus the default behaviour is to use partial matching only when extracting from recursive objects (except environments) by `$`. Even in that case, warnings can be switched on by `options(warnPartialMatchDollar = TRUE)`.

Neither empty (`""`) nor `NA` indices match any names, not even empty nor missing names. If any object has no names or appropriate dimnames, they are taken as all `""` and so match nothing.

Error conditions

Attempting to apply a subsetting operation to objects for which this is not possible signals an error of class `notSubsettableError`. The `object` component of the error condition contains the non-subsettable object.

Subscript out of bounds errors are signaled as errors of class `subscriptOutOfBoundsError`. The `object` component of the error condition contains the object being subsetted. The integer

`subscript` component is zero for vector subscripting, and for multiple subscripts indicates which subscript was out of bounds. The `index` component contains the erroneous index.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`names` for details of matching to names, and `pmatch` for partial matching.

`list`, `array`, `matrix`.

`[.data.frame` and `[.factor` for the behaviour when applied to data.frame and factors.

`Syntax` for operator precedence, and the 'R Language Definition' manual about indexing details.

`NULL` for details of indexing null objects.

Examples

```
x <- 1:12
m <- matrix(1:6, nrow = 2, dimnames = list(c("a", "b"), LETTERS[1:3]))
li <- list(pi = pi, e = exp(1))
x[10]                 # the tenth element of x
x <- x[-1]            # delete the 1st element of x
m[1,]                 # the first row of matrix m
m[1, , drop = FALSE]  # is a 1-row matrix
m[,c(TRUE,FALSE,TRUE)]# logical indexing
m[cbind(c(1,2,1),3:1)]# matrix numeric index
ci <- cbind(c("a", "b", "a"), c("A", "C", "B"))
m[ci]                 # matrix character index
m <- m[,-1]           # delete the first column of m
li[[1]]               # the first element of list li
y <- list(1, 2, a = 4, 5)
y[c(3, 4)]            # a list containing elements 3 and 4 of y
y$a                   # the element of y named a

## non-integer indices are truncated:
(i <- 3.999999999) # "4" is printed
(1:5)[i]  # 3

## named atomic vectors, compare "[" and "[[" :
nx <- c(Abc = 123, pi = pi)
nx[1] ; nx["pi"] # keeps names, whereas "[[" does not:
nx[[1]] ; nx[["pi"]]

## recursive indexing into lists
z <- list(a = list(b = 9, c = "hello"), d = 1:5)
unlist(z)
z[[c(1, 2)]]
z[[c(1, 2, 1)]]  # both "hello"
z[[c("a", "b")]] <- "new"
unlist(z)
```

```
## check $ and [[ for environments
e1 <- new.env()
e1$a <- 10
e1[["a"]]
e1[["b"]] <- 20
e1$b
ls(e1)

## partial matching - possibly with warning :
stopifnot(identical(li$p, pi))
op <- options(warnPartialMatchDollar = TRUE)
stopifnot( identical(li$p, pi), #-- a warning
  inherits(tryCatch (li$p, warning = identity), "warning"))
## revert the warning option:
options(op)
```

`Extract.data.frame` *Extract or Replace Parts of a Data Frame*

Description

Extract or replace subsets of data frames.

Usage

```
## S3 method for class 'data.frame'
x[i, j, drop = ]
## S3 replacement method for class 'data.frame'
x[i, j] <- value
## S3 method for class 'data.frame'
x[[..., exact = TRUE]]
## S3 replacement method for class 'data.frame'
x[[i, j]] <- value
## S3 replacement method for class 'data.frame'
x$name <- value
```

Arguments

`x`	data frame.
`i, j, ...`	elements to extract or replace. For `[` and `[[`, these are `numeric` or `character` or, for `[` only, empty or `logical`. Numeric values are coerced to integer as if by `as.integer`. For replacement by `[`, a logical matrix is allowed.
`name`	a literal character string or a name (possibly backtick quoted).
`drop`	logical. If `TRUE` the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but **not** to drop if only one row is left.

value	a suitable replacement value: it will be repeated a whole number of times if necessary and it may be coerced: see the Coercion section. If NULL, deletes the column if a single column is selected.
exact	logical: see [, and applies to column names.

Details

Data frames can be indexed in several modes. When [and [[are used with a single vector index (x[i] or x[[i]]), they index the data frame as if it were a list. In this usage a drop argument is ignored, with a warning.

There is no data.frame method for $, so x$name uses the default method which treats x as a list (with partial matching of column names if the match is unique, see Extract). The replacement method (for $) checks value for the correct number of rows, and replicates it if necessary.

When [and [[are used with two indices (x[i, j] and x[[i, j]]) they act like indexing a matrix: [[can only be used to select one element. Note that for each selected column, xj say, typically (if it is not matrix-like), the resulting column will be xj[i], and hence rely on the corresponding [method, see the examples section.

If [returns a data frame it will have unique (and non-missing) row names, if necessary transforming the row names using make.unique. Similarly, if columns are selected column names will be transformed to be unique if necessary (e.g., if columns are selected more than once, or if more than one column of a given name is selected if the data frame has duplicate column names).

When drop = TRUE, this is applied to the subsetting of any matrices contained in the data frame as well as to the data frame itself.

The replacement methods can be used to add whole column(s) by specifying non-existent column(s), in which case the column(s) are added at the right-hand edge of the data frame and numerical indices must be contiguous to existing indices. On the other hand, rows can be added at any row after the current last row, and the columns will be in-filled with missing values. Missing values in the indices are not allowed for replacement.

For [the replacement value can be a list: each element of the list is used to replace (part of) one column, recycling the list as necessary. If columns specified by number are created, the names (if any) of the corresponding list elements are used to name the columns. If the replacement is not selecting rows, list values can contain NULL elements which will cause the corresponding columns to be deleted. (See the Examples.)

Matrix indexing (x[i] with a logical or a 2-column integer matrix i) using [is not recommended. For extraction, x is first coerced to a matrix. For replacement, logical matrix indices must be of the same dimension as x. Replacements are done one column at a time, with multiple type coercions possibly taking place.

Both [and [[extraction methods partially match row names. By default neither partially match column names, but [[will if exact = FALSE (and with a warning if exact = NA). If you want to exact matching on row names use match, as in the examples.

Value

For [a data frame, list or a single column (the latter two only when dimensions have been dropped). If matrix indexing is used for extraction a vector results. If the result would be a data frame an error

results if undefined columns are selected (as there is no general concept of a 'missing' column in a data frame). Otherwise if a single column is selected and this is undefined the result is NULL.

For [[a column of the data frame or NULL (extraction with one index) or a length-one vector (extraction with two indices).

For $, a column of the data frame (or NULL).

For [<-, [[<- and $<-, a data frame.

Coercion

The story over when replacement values are coerced is a complicated one, and one that has changed during R's development. This section is a guide only.

When [and [[are used to add or replace a whole column, no coercion takes place but value will be replicated (by calling the generic function rep) to the right length if an exact number of repeats can be used.

When [is used with a logical matrix, each value is coerced to the type of the column into which it is to be placed.

When [and [[are used with two indices, the column will be coerced as necessary to accommodate the value.

Note that when the replacement value is an array (including a matrix) it is *not* treated as a series of columns (as data.frame and as.data.frame do) but inserted as a single column.

Warning

The default behaviour when only one *row* is left is equivalent to specifying drop = FALSE. To drop from a data frame to a list, drop = TRUE has to be specified explicitly.

Arguments other than drop and exact should not be named: there is a warning if they are and the behaviour differs from the description here.

See Also

subset which is often easier for extraction, data.frame, Extract.

Examples

```
sw <- swiss[1:5, 1:4]  # select a manageable subset

sw[1:3]      # select columns
sw[, 1:3]    # same
sw[4:5, 1:3] # select rows and columns
sw[1]        # a one-column data frame
sw[, 1, drop = FALSE]  # the same
sw[, 1]      # a (unnamed) vector
sw[[1]]      # the same
sw$Fert      # the same (possibly w/ warning, see ?Extract)

sw[1,]       # a one-row data frame
sw[1,, drop = TRUE]  # a list
```

```
sw["C", ] # partially matches
sw[match("C", row.names(sw)), ] # no exact match
try(sw[, "Ferti"]) # column names must match exactly


sw[sw$Fertility > 90,] # logical indexing, see also ?subset
sw[c(1, 1:2), ]        # duplicate row, unique row names are created

sw[sw <= 6] <- 6  # logical matrix indexing
sw

## adding a column
sw["new1"] <- LETTERS[1:5]   # adds a character column
sw[["new2"]] <- letters[1:5] # ditto
sw[, "new3"] <- LETTERS[1:5] # ditto
sw$new4 <- 1:5
sapply(sw, class)
sw$new  # -> NULL: no unique partial match
sw$new4 <- NULL              # delete the column
sw
sw[6:8] <- list(letters[10:14], NULL, aa = 1:5)
# update col. 6, delete 7, append
sw

## matrices in a data frame
A <- data.frame(x = 1:3, y = I(matrix(4:9, 3, 2)),
                         z = I(matrix(letters[1:9], 3, 3)))
A[1:3, "y"] # a matrix
A[1:3, "z"] # a matrix
A[, "y"]    # a matrix
stopifnot(identical(colnames(A), c("x", "y", "z")), ncol(A) == 3L,
          identical(A[,"y"], A[1:3, "y"]),
          inherits (A[,"y"], "AsIs"))

## keeping special attributes: use a class with a
## "as.data.frame" and "[" method;
## "avector" := vector that keeps attributes.   Could provide a constructor
##  avector <- function(x) { class(x) <- c("avector", class(x)); x }
as.data.frame.avector <- as.data.frame.vector

`[.avector` <- function(x,i,...) {
  r <- NextMethod("[")
  mostattributes(r) <- attributes(x)
  r
}

d <- data.frame(i = 0:7, f = gl(2,4),
                u = structure(11:18, unit = "kg", class = "avector"))
str(d[2:4, -1]) # 'u' keeps its "unit"
```

`Extract.factor` *Extract or Replace Parts of a Factor*

Description

Extract or replace subsets of factors.

Usage

```
## S3 method for class 'factor'
x[..., drop = FALSE]
## S3 method for class 'factor'
x[[...]]
## S3 replacement method for class 'factor'
x[...] <- value
## S3 replacement method for class 'factor'
x[[...]] <- value
```

Arguments

`x`	a factor.
`...`	a specification of indices – see `Extract`.
`drop`	logical. If true, unused levels are dropped.
`value`	character: a set of levels. Factor values are coerced to character.

Details

When unused levels are dropped the ordering of the remaining levels is preserved.

If `value` is not in `levels(x)`, a missing value is assigned with a warning.

Any `contrasts` assigned to the factor are preserved unless `drop = TRUE`.

The `[[` method supports argument `exact`.

Value

A factor with the same set of levels as `x` unless `drop = TRUE`.

See Also

`factor`, `Extract`.

Examples

```
## following example(factor)
(ff <- factor(substring("statistics", 1:10, 1:10), levels = letters))
ff[, drop = TRUE]
factor(letters[7:10])[2:3, drop = TRUE]
```

`Extremes` *Maxima and Minima*

Description

Returns the (regular or **p**arallel) maxima and minima of the input values.

`pmax*()` and `pmin*()` take one or more vectors as arguments, recycle them to common length and return a single vector giving the *'parallel'* maxima (or minima) of the argument vectors.

Usage

```
max(..., na.rm = FALSE)
min(..., na.rm = FALSE)

pmax(..., na.rm = FALSE)
pmin(..., na.rm = FALSE)

pmax.int(..., na.rm = FALSE)
pmin.int(..., na.rm = FALSE)
```

Arguments

`...`	numeric or character arguments (see Note).
`na.rm`	a logical indicating whether missing values should be removed.

Details

`max` and `min` return the maximum or minimum of *all* the values present in their arguments, as `integer` if all are `logical` or `integer`, as `double` if all are numeric, and character otherwise.

If `na.rm` is `FALSE` an `NA` value in any of the arguments will cause a value of `NA` to be returned, otherwise `NA` values are ignored.

The minimum and maximum of a numeric empty set are `+Inf` and `-Inf` (in this order!) which ensures *transitivity*, e.g., `min(x1, min(x2)) == min(x1, x2)`. For numeric `x` `max(x) == -Inf` and `min(x) == +Inf` whenever `length(x) == 0` (after removing missing values if requested). However, `pmax` and `pmin` return `NA` if all the parallel elements are `NA` even for `na.rm = TRUE`.

`pmax` and `pmin` take one or more vectors (or matrices) as arguments and return a single vector giving the 'parallel' maxima (or minima) of the vectors. The first element of the result is the maximum (minimum) of the first elements of all the arguments, the second element of the result is the maximum (minimum) of the second elements of all the arguments and so on. Shorter inputs (of non-zero length) are recycled if necessary. Attributes (see `attributes`: such as `names` or `dim`) are copied from the first argument (if applicable, e.g., *not* for an `S4` object).

`pmax.int` and `pmin.int` are faster internal versions only used when all arguments are atomic vectors and there are no classes: they drop all attributes. (Note that all versions fail for raw and complex vectors since these have no ordering.)

`max` and `min` are generic functions: methods can be defined for them individually or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

By definition the min/max of a numeric vector containing an `NaN` is `NaN`, except that the min/max of any vector containing an `NA` is `NA` even if it also contains an `NaN`. Note that `max(NA, Inf) == NA` even though the maximum would be `Inf` whatever the missing value actually is.

Character versions are sorted lexicographically, and this depends on the collating sequence of the locale in use: the help for 'Comparison' gives details. The max/min of an empty character vector is defined to be character `NA`. (One could argue that as `""` is the smallest character element, the maximum should be `""`, but there is no obvious candidate for the minimum.)

Value

For `min` or `max`, a length-one vector. For `pmin` or `pmax`, a vector of length the longest of the input vectors, or length zero if one of the inputs had zero length.

The type of the result will be that of the highest of the inputs in the hierarchy integer < double < character.

For `min` and `max` if there are only numeric inputs and all are empty (after possible removal of `NA`s), the result is double (`Inf` or `-Inf`).

S4 methods

`max` and `min` are part of the S4 `Summary` group generic. Methods for them must use the signature `x, ..., na.rm`.

Note

'Numeric' arguments are vectors of type integer and numeric, and logical (coerced to integer). For historical reasons, `NULL` is accepted as equivalent to `integer(0)`.

`pmax` and `pmin` will also work on classed S3 or S4 objects with appropriate methods for comparison, `is.na` and `rep` (if recycling of arguments is needed).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`range` (*both* min and max) and `which.min` (`which.max`) for the *arg min*, i.e., the location where an extreme value occurs.

'plotmath' for the use of `min` in plot annotation.

Examples

```
require(stats); require(graphics)
 min(5:1, pi) #-> one number
pmin(5:1, pi) #->  5  numbers
```

```
x <- sort(rnorm(100));  cH <- 1.35
pmin(cH, quantile(x)) # no names
pmin(quantile(x), cH) # has names
plot(x, pmin(cH, pmax(-cH, x)), type = "b", main =  "Huber's function")

cut01 <- function(x) pmax(pmin(x, 1), 0)
curve(      x^2 - 1/4, -1.4, 1.5, col = 2)
curve(cut01(x^2 - 1/4), col = "blue", add = TRUE, n = 500)
## pmax(), pmin() preserve attributes of *first* argument
D <- diag(x = (3:1)/4) ; n0 <- numeric()
stopifnot(identical(D,  cut01(D) ),
          identical(n0, cut01(n0)),
          identical(n0, cut01(NULL)),
          identical(n0, pmax(3:1, n0, 2)),
          identical(n0, pmax(n0, 4)))
```

`extSoftVersion` *Report Versions of Third-Party Software*

Description

Report versions of (external) third-party software used.

Usage

```
extSoftVersion()
```

Details

The reports the versions of third-party software libraries in use. These are often external but might have been compiled into R when it was installed.

With dynamic linking, these are the versions of the libraries linked to in this session: with static linking, of those compiled in.

Value

A named character vector, currently with components

`zlib`	The version of `zlib` in use.
`bzlib`	The version of `bzlib` (from `bzip2`) in use.
`xz`	The version of `liblzma` (from `xz`) in use.
`libdeflate`	The version of `libdeflate` (if any otherwise `""`) used when R was built.
`PCRE`	The version of `PCRE` in use. PCRE1 has versions < 10.00, PCRE2 has versions >= 10.00.
`ICU`	The version of `ICU` in use (if any, otherwise `""`).
`TRE`	The version of `libtre` in use.

iconv	The implementation and version of the iconv library in use (if known).
readline	The version of readline in use (if any, otherwise ""). If using the emulation by libedit aka editline this will be "EditLine wrapper" preceded by the readline version it emulates: that is most likely to be seen on macOS.
BLAS	Name of the binary/executable file with the implementation of BLAS in use (if known, otherwise "").

Note that the values for bzlib and pcre normally contain a date as well as the version number, and that for tre includes several items separated by spaces, the version number being the second.

For iconv this will give the implementation as well as the version, for example "GNU libiconv 1.14", "glibc 2.18" or "win_iconv" (which has no version number).

The name of the binary/executable file for BLAS can be used as an indication of which implementation is in use. Typically, the R version of BLAS will appear as libR.so (libR.dylib), R or libRblas.so (libRblas.dylib), depending on how R was built. Note that libRblas.so (libRblas.dylib) may also be shown for an external BLAS implementation that had been copied, hard-linked or renamed by the system administrator. For an external BLAS, a shared object file will be given and its path/name may indicate the vendor/version. The detection does not work on Windows nor for some uses of the Accelerate framework on macOS.

See Also

libcurlVersion for the version of libCurl.

La_version for the version of LAPACK in use.

La_library for binary/executable file with LAPACK in use.

grSoftVersion for third-party graphics software.

tclVersion in package **tcltk** for the version of Tcl/Tk.

pcre_config for PCRE configuration options.

Examples

```
extSoftVersion()
## the PCRE version
sub(" .*", "", extSoftVersion()["PCRE"])
```

---

factor *Factors*

Description

The function factor is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered. For compatibility with S there is also a function ordered.

is.factor, is.ordered, as.factor and as.ordered are the membership and coercion functions for these classes.

Usage

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax = NA)

ordered(x = character(), ...)

is.factor(x)
is.ordered(x)

as.factor(x)
as.ordered(x)

addNA(x, ifany = FALSE)

.valid.factor(object)
```

Arguments

x	a vector of data, usually taking a small number of distinct values.
levels	an optional vector of the unique values (as character strings) that x might have taken. The default is the unique set of values taken by as.character(x), sorted into increasing order *of* x. Note that this set can be specified as smaller than sort(unique(x)).
labels	*either* an optional character vector of labels for the levels (in the same order as levels after removing those in exclude), *or* a character string of length 1. Duplicated values in labels can be used to map different values of x to the same factor level.
exclude	a vector of values to be excluded when forming the set of levels. This may be factor with the same level set as x or should be a character.
ordered	logical flag to determine if the levels should be regarded as ordered (in the order given).
nmax	an upper bound on the number of levels; see 'Details'.
...	(in ordered(.)): any of the above, apart from ordered itself.
ifany	only add an NA level if it is used, i.e. if any(is.na(x)).
object	an R object.

Details

The type of the vector x is not restricted; it only must have an as.character method and be sortable (by order).

Ordered factors differ from factors only in their class, but methods and model-fitting functions may treat the two classes quite differently, see options("contrasts").

The encoding of the vector happens as follows. First all the values in exclude are removed from levels. If x[i] equals levels[j], then the i-th element of the result is j. If no match is found for x[i] in levels (which will happen for excluded values) then the i-th element of the result is set to NA.

Normally the 'levels' used as an attribute of the result are the reduced set of levels after removing those in exclude, but this can be altered by supplying labels. This should either be a set of new labels for the levels, or a character string, in which case the levels are that character string with a sequence number appended.

factor(x, exclude = NULL) applied to a factor without NAs is a no-operation unless there are unused levels: in that case, a factor with the reduced level set is returned. If exclude is used, since R version 3.4.0, excluding non-existing character levels is equivalent to excluding nothing, and when exclude is a character vector, that *is* applied to the levels of x. Alternatively, exclude can be factor with the same level set as x and will exclude the levels present in exclude.

The codes of a factor may contain NA. For a numeric x, set exclude = NULL to make NA an extra level (prints as '<NA>'); by default, this is the last level.

If NA is a level, the way to set a code to be missing (as opposed to the code of the missing level) is to use is.na on the left-hand-side of an assignment (as in is.na(f)[i] <- TRUE; indexing inside is.na does not work). Under those circumstances missing values are currently printed as '<NA>', i.e., identical to entries of level NA.

is.factor is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Where levels is not supplied, unique is called. Since factors typically have quite a small number of levels, for large vectors x it is helpful to supply nmax as an upper bound on the number of unique values.

When using c to combine a (possibly ordered) factor with other objects, if all objects are (possibly ordered) factors, the result will be a factor with levels the union of the level sets of the elements, in the order the levels occur in the level sets of the elements (which means that if all the elements have the same level set, that is the level set of the result), equivalent to how unlist operates on a list of factor objects.

Value

factor returns an object of class "factor" which has a set of integer codes the length of x with a "levels" attribute of mode character and unique (!anyDuplicated(.)) entries. If argument ordered is true (or ordered() is used) the result has class c("ordered", "factor"). Undocumentedly for a long time, factor(x) loses all attributes(x) but "names", and resets "levels" and "class".

Applying factor to an ordered or unordered factor returns a factor (of the same type) with just the levels which occur: see also [.factor for a more transparent way to achieve this.

is.factor returns TRUE or FALSE depending on whether its argument is of type factor or not. Correspondingly, is.ordered returns TRUE when its argument is an ordered factor and FALSE otherwise.

as.factor coerces its argument to a factor. It is an abbreviated (sometimes faster) form of factor.

as.ordered(x) returns x if this is ordered, and ordered(x) otherwise.

addNA modifies a factor by turning NA into an extra level (so that NA values are counted in tables, for instance).

.valid.factor(object) checks the validity of a factor, currently only levels(object), and returns TRUE if it is valid, otherwise a string describing the validity problem. This function is used for validObject(<factor>).

Warning

The interpretation of a factor depends on both the codes and the "levels" attribute. Be careful only to compare factors with the same set of levels (in the same order). In particular, as.numeric applied to a factor is meaningless, and may happen by implicit coercion. To transform a factor f to approximately its original numeric values, as.numeric(levels(f))[f] is recommended and slightly more efficient than as.numeric(as.character(f)).

The levels of a factor are by default sorted, but the sort order may well depend on the locale at the time of creation, and should not be assumed to be ASCII.

There are some anomalies associated with factors that have NA as a level. It is suggested to use them sparingly, e.g., only for tabulation purposes.

Comparison operators and group generic methods

There are "factor" and "ordered" methods for the group generic Ops which provide methods for the Comparison operators, and for the min, max, and range generics in Summary of "ordered". (The rest of the groups and the Math group generate an error as they are not meaningful for factors.)

Only == and != can be used for factors: a factor can only be compared to another factor with an identical set of levels (not necessarily in the same ordering) or to a character vector. Ordered factors are compared in the same way, but the general dispatch mechanism precludes comparing ordered and unordered factors.

All the comparison operators are available for ordered factors. Collation is done by the levels of the operands: if both operands are ordered factors they must have the same level set.

Note

In earlier versions of R, storing character data as a factor was more space efficient if there is even a small proportion of repeats. However, identical character strings now share storage, so the difference is small in most cases. (Integer values are stored in 4 bytes whereas each reference to a character string needs a pointer of 4 or 8 bytes.)

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

[.factor for subsetting of factors.

gl for construction of balanced factors and C for factors with specified contrasts. levels and nlevels for accessing the levels, and unclass to get integer codes.

Examples

```
(ff <- factor(substring("statistics", 1:10, 1:10), levels = letters))
as.integer(ff)      # the internal codes
(f. <- factor(ff))  # drops the levels that do not occur
ff[, drop = TRUE]   # the same, more transparently

factor(letters[1:20], labels = "letter")
```

```
class(ordered(4:1)) # "ordered", inheriting from "factor"
z <- factor(LETTERS[3:1], ordered = TRUE)
## and "relational" methods work:
stopifnot(sort(z)[c(1,3)] == range(z), min(z) < max(z))


## suppose you want "NA" as a level, and to allow missing values.
(x <- factor(c(1, 2, NA), exclude = NULL))
is.na(x)[2] <- TRUE
x  # [1] 1    <NA> <NA>
is.na(x)
# [1] FALSE  TRUE FALSE

## More rational, since R 3.4.0 :
factor(c(1:2, NA), exclude =  "" ) # keeps <NA> , as
factor(c(1:2, NA), exclude = NULL) # always did
## exclude = <character>
z # ordered levels 'A < B < C'
factor(z, exclude = "C") # does exclude
factor(z, exclude = "B") # ditto

## Now, labels maybe duplicated:
## factor() with duplicated labels allowing to "merge levels"
x <- c("Man", "Male", "Man", "Lady", "Female")
## Map from 4 different values to only two levels:
(xf <- factor(x, levels = c("Male", "Man" , "Lady",   "Female"),
                 labels = c("Male", "Male", "Female", "Female")))
#> [1] Male   Male   Male   Female Female
#> Levels: Male Female

## Using addNA()
Month <- airquality$Month
table(addNA(Month))
table(addNA(Month, ifany = TRUE))
```

`file.access` *Ascertain File Accessibility*

Description

Utility function to access information about files on the user's file systems.

Usage

```
file.access(names, mode = 0)
```

Arguments

names	character vector containing file names. Tilde-expansion will be done: see path.expand.
mode	integer specifying access mode required: see 'Details'.

Details

The mode value can be the exclusive or (xor), i.e., a partial sum of the following values, and hence must be in 0:7,

0 test for existence.

1 test for execute permission.

2 test for write permission.

4 test for read permission.

Permission will be computed for real user ID and real group ID (rather than the effective IDs).

Please note that it is not a good idea to use this function to test before trying to open a file. On a multi-tasking system, it is possible that the accessibility of a file will change between the time you call file.access() and the time you try to open the file. It is better to wrap file open attempts in try.

Value

An integer vector with values 0 for success and -1 for failure.

Note

This was written as a replacement for the S-PLUS function access, a wrapper for the C function of the same name, which explains the return value encoding. Note that the return value is **false** for **success**.

See Also

file.info for more details on permissions, Sys.chmod to change permissions, and try for a 'test it and see' approach.

file_test for shell-style file tests.

Examples

```
fa <- file.access(dir("."))

table(fa) # count successes & failures
```

file.choose *Choose a File Interactively*

Description

Choose a file interactively.

Usage

```
file.choose(new = FALSE)
```

Arguments

new	Logical: choose the style of dialog box presented to the user: at present only new = FALSE is used.

Value

A character vector of length one giving the file path.

See Also

list.files for non-interactive selection.

file.info *Extract File Information*

Description

Utility function to extract information about files on the user's file systems.

Usage

```
file.info(..., extra_cols = TRUE)

file.mode(...)
file.mtime(...)
file.size(...)
```

Arguments

...	character vectors containing file paths. Tilde-expansion is done: see path.expand.
extra_cols	logical: return all cols rather than just the first six.

Details

What constitutes a 'file' is OS-dependent but includes directories. (However, directory names must not include a trailing backslash or slash on Windows.) See also the section in the help for `file.exists` on case-insensitive file systems.

The file 'mode' follows POSIX conventions, giving three octal digits summarizing the permissions for the file owner, the owner's group and for anyone respectively. Each digit is the logical *or* of read (4), write (2) and execute/search (1) permissions.

See files for how file paths with marked encodings are interpreted.

On unix alikes: On most systems symbolic links are followed, so information is given about the file to which the link points rather than about the link.

On Windows: File modes are probably only useful on NTFS file systems, and it seems all three digits refer to the file's owner. The execute/search bits are set for directories, and for files based on their extensions (e.g., '`.exe`', '`.com`', '`.cmd`' and '`.bat`' files). `file.access` will give a more reliable view of read/write access availability to the R process.

UTF-8-encoded file names not valid in the current locale can be used.

Junction points and symbolic links are followed, so information is given about the file/directory to which the link points rather than about the link.

Value

For `file.info()`, data frame with row names the file names and columns

`size`	double: File size in bytes.
`isdir`	logical: Is the file a directory?
`mode`	integer of class `"octmode"`. The file permissions, printed in octal, for example `644`.
`mtime`, `ctime`, `atime`	object of class `"POSIXct"`: file modification, 'last status change' and last access times.

On unix alikes: **uid:** integer, the user ID of the file's owner.

gid: integer, the group ID of the file's group.

uname: character, `uid` interpreted as a user name.

grname: character, `gid` interpreted as a group name. Unknown user and group names will be `NA`.

On Windows only: **exe:** character indicating the sort of executable. Possible values are `"no"`, `"msdos"`, `"win16"`, `"win32"`, `"win64"` and `"unknown"`. Note that a file (e.g., a script file) can be executable according to the mode bits but not executable in this sense.

If `extra_cols` is false, only the first six columns are returned: as these can all be found from a single C system call this can be faster. (However, properly configured systems will use a 'name service cache daemon' to speed up the name lookups.)

Entries for non-existent or non-readable files will be `NA`.

The `uid`, `gid`, `uname` and `grname` columns may not be supplied on a non-POSIX Unix-alike system, and will not be on Windows.

What is meant by the three file times depends on the OS and file system. On Windows native file systems `ctime` is the file creation time (something which is not recorded on most Unix-alike file systems). What is meant by 'file access' and hence the 'last access time' is system-dependent.

The resolution of the file times depends on both the OS and the type of the file system. Modern file systems typically record times to an accuracy of a microsecond or better: notable exceptions are HFS+ on macOS (recorded in seconds) and modification time on older FAT systems (recorded in increments of 2 seconds). Note that `"POSIXct"` times are by default printed in whole seconds: to change that see `strftime`.

`file.mode()`, `file.mtime()` and `file.size()` are fast convenience wrappers returning just one of the columns.

Note

Some (now old) unix alike systems allow files of more than 2Gb to be created but not accessed by the `stat` system call. Such files may show up as non-readable (and very likely not be readable by any of R's input functions).

See Also

`Sys.readlink` to find out about symbolic links, `files`, `file.access`, `list.files`, and `DateTimeClasses` for the date formats.

`Sys.chmod` to change permissions.

Examples

```
ncol(finf <- file.info(dir()))  # at least six
finf # the whole list
## Those that are more than 100 days old :
finf <- file.info(dir(), extra_cols = FALSE)
finf[difftime(Sys.time(), finf[,"mtime"], units = "days") > 100 , 1:4]

file.info("no-such-file-exists")

## E.g., for R-core, in a R-devel version:
if(Sys.info()[["sysname"]] == "Linux")
    sort(file.mtime(file.path(R.home("bin"),
                              c("",
                                file.path(c("", "exec"), "R"))))
         ))
```

---

file.path *Construct Path to File*

Description

Construct the path to a file from components in a platform-independent way.

Usage

```
file.path(..., fsep = .Platform$file.sep)
```

Arguments

...	character vectors. Long vectors are not supported.
fsep	the path separator to use (assumed to be ASCII).

Details

The implementation is designed to be fast (faster than paste) as this function is used extensively in R itself.

It can also be used for environment paths such as PATH and R_LIBS with fsep = .Platform$path.sep.

Trailing path separators are invalid for Windows file paths apart from '/' and 'd:/' (although some functions/utilities do accept them), so a trailing / or \ is removed there.

Value

A character vector of the arguments concatenated term-by-term and separated by fsep if all arguments have positive length; otherwise, an empty character vector (unlike paste).

An element of the result will be marked (see Encoding) as UTF-8 if run in a UTF-8 locale (when marked inputs are converted to UTF-8) or if a component of the result is marked as UTF-8, or as Latin-1 in a non-Latin-1 locale.

Note

The components are by default separated by / (not \) on Windows.

See Also

basename, normalizePath, path.expand.

---

file.show *Display One or More Text Files*

Description

Display one or more (plain) text files, in a platform specific way, typically via a 'pager'.

Usage

```
file.show(..., header = rep("", nfiles),
          title = "R Information",
          delete.file = FALSE, pager = getOption("pager"),
          encoding = "")
```

Arguments

...	one or more character vectors containing the names of the files to be displayed. Paths with have tilde expansion.
header	character vector (of the same length as the number of files specified in ...) giving a header for each file being displayed. Defaults to empty strings.
title	an overall title for the display. If a single separate window is used for the display, title will be used as the window title. If multiple windows are used, their titles should combine the title and the file-specific header.
delete.file	should the files be deleted after display? Used for temporary files.
pager	the pager to be used, see 'Details'.
encoding	character string giving the encoding to be assumed for the file(s).

Details

This function provides the core of the R help system, but it can be used for other purposes as well, such as page.

How the pager is implemented is highly system-dependent.

The basic Unix version concatenates the files (using the headers) to a temporary file, and displays it in the pager selected by the pager argument, which is a character vector specifying a system command (a full path or a command found on the PATH) to run on the set of files. The 'factory-fresh' default is to use 'R_HOME/bin/pager', which is a shell script running the command-line specified by the environment variable PAGER whose default is set at configuration, usually to less. On a Unix-alike more is used if pager is empty.

Most GUI systems will use a separate pager window for each file, and let the user leave it up while R continues running. The selection of such pagers could either be done using special pager names being intercepted by lower-level code (such as "internal" and "console" on Windows), or by letting pager be an R function which will be called with arguments (files, header, title, delete.file) corresponding to the first four arguments of file.show and take care of interfacing to the GUI.

The R.app GUI on macOS uses its internal pager irrespective of the setting of pager.

Not all implementations will honour delete.file. In particular, using an external pager on Windows does not, as there is no way to know when the external application has finished with the file.

Author(s)

Ross Ihaka, Brian Ripley.

See Also

file.exists, list.files.

Text-type help and RShowDoc call file.show.

Consider getOption("pdfviewer") and, e.g., system for displaying pdf files.

file.edit.

Examples

```
file.show(file.path(R.home("doc"), "COPYRIGHTS"))
```

`files` *File Manipulation*

Description

These functions provide a low-level interface to the computer's file system.

Usage

```
file.create(..., showWarnings = TRUE)
file.exists(...)
file.remove(...)
file.rename(from, to)
file.append(file1, file2)
file.copy(from, to, overwrite = recursive, recursive = FALSE,
          copy.mode = TRUE, copy.date = FALSE)
file.symlink(from, to)
file.link(from, to)
```

Arguments

`...`, `file1`, `file2`	character vectors, containing file names or paths.
`from`, `to`	character vectors, containing file names or paths. For `file.copy` and `file.symlink` `to` can alternatively be the path to a single existing directory.
`overwrite`	logical; should existing destination files be overwritten?
`showWarnings`	logical; should the warnings on failure be shown?
`recursive`	logical. If `to` is a directory, should directories in `from` be copied (and their contents)? (Like `cp -R` on POSIX OSes.)
`copy.mode`	logical: should file permission bits be copied where possible?
`copy.date`	logical: should file dates be preserved where possible? See `Sys.setFileTime`.

Details

The `...` arguments are concatenated to form one character string: you can specify the files separately or as one vector. All of these functions expand path names: see `path.expand`. (`file.exists` silently reports false for paths that would be too long after expansion: the rest will give a warning.)

`file.create` creates files with the given names if they do not already exist and truncates them if they do. They are created with the maximal read/write permissions allowed by the 'umask' setting (where relevant). By default a warning is given (with the reason) if the operation fails.

`file.exists` returns a logical vector indicating whether the files named by its argument exist. (Here 'exists' is in the sense of the system's `stat` call: a file will be reported as existing only if

you have the permissions needed by `stat`. Existence can also be checked by `file.access`, which might use different permissions and so obtain a different result. Note that the existence of a file does not imply that it is readable: for that use `file.access`.) What constitutes a 'file' is system-dependent, but should include directories. (However, directory names must not include a trailing backslash or slash on Windows.) Note that if the file is a symbolic link on a Unix-alike, the result indicates if the link points to an actual file, not just if the link exists. On Windows, the result is unreliable for a broken symbolic link (junction). Lastly, note the *different* function `exists` which checks for existence of R objects.

`file.remove` attempts to remove the files named in its argument. On most Unix platforms 'file' includes *empty* directories, symbolic links, fifos and sockets. On Windows, 'file' means a regular file and not, say, an empty directory.

`file.rename` attempts to rename files (and `from` and `to` must be of the same length). Where file permissions allow this will overwrite an existing element of `to`. This is subject to the limitations of the OS's corresponding system call (see something like `man 2 rename` on a Unix-alike): in particular in the interpretation of 'file': most platforms will not rename files from one file system to another. **NB:** This means that renaming a file from a temporary directory to the user's filespace or during package installation will often fail. (On Windows, `file.rename` can rename files but not directories across volumes.) On platforms which allow directories to be renamed, typically neither or both of `from` and `to` must a directory, and if `to` exists it must be an empty directory.

`file.append` attempts to append the files named by its second argument to those named by its first. The R subscript recycling rule is used to align names given in vectors of different lengths.

`file.copy` works in a similar way to `file.append` but with the arguments in the natural order for copying. Copying to existing destination files is skipped unless `overwrite = TRUE`. The `to` argument can specify a single existing directory. If `copy.mode = TRUE` file read/write/execute permissions are copied where possible, restricted by 'umask'. (On Windows this applies only to files.) Other security attributes such as ACLs are not copied. On a POSIX filesystem the targets of symbolic links will be copied rather than the links themselves, and hard links are copied separately. Using `copy.date = TRUE` may or may not copy the timestamp exactly (for example, fractional seconds may be omitted), but is more likely to do so as from R 3.4.0.

`file.symlink` and `file.link` make symbolic and hard links on those file systems which support them. For `file.symlink` the `to` argument can specify a single existing directory. (Unix and macOS native filesystems support both. Windows has hard links to files on NTFS file systems and concepts related to symbolic links on recent versions: see the section below on the Windows version of this help page. What happens on a FAT or SMB-mounted file system is OS-specific.)

File arguments with a marked encoding (see `Encoding` are if possible translated to the native encoding, except on Windows where Unicode file operations are used (so marking as UTF-8 can be used to access file paths not in the native encoding on suitable file systems).

Value

These functions return a logical vector indicating which operation succeeded for each of the files attempted. Using a missing value for a file or path name will always be regarded as a failure.

If `showWarnings = TRUE`, `file.create` will give a warning for an unexpected failure.

Case-insensitive file systems

Case-insensitive file systems are the norm on Windows and macOS, but can be found on all OSes (for example a FAT-formatted USB drive is probably case-insensitive).

These functions will most likely match existing files regardless of case on such file systems: however this is an OS function and it is possible that file names might be mapped to upper or lower case.

Warning

Always check the return value of these functions when used in package code. This is especially important for `file.rename`, which has OS-specific restrictions (and note that the session temporary directory is commonly on a different file system from the working directory): it is only portable to use `file.rename` to change file name(s) within a single directory.

Author(s)

Ross Ihaka, Brian Ripley

See Also

`file.info`, `file.access`, `file.path`, `file.show`, `list.files`, `unlink`, `basename`, `path.expand`.

`dir.create`.

`Sys.glob` to expand wildcards in file specifications.

`file_test`, `Sys.readlink` (for 'symlink's).

https://en.wikipedia.org/wiki/Hard_link and https://en.wikipedia.org/wiki/Symbolic_link for the concepts of links and their limitations.

Examples

```
cat("file A\n", file = "A")
cat("file B\n", file = "B")
file.append("A", "B")
file.create("A") # (trashing previous)
file.append("A", rep("B", 10))
if(interactive()) file.show("A") # -> the 10 lines from 'B'
file.copy("A", "C")
dir.create("tmp")
file.copy(c("A", "B"), "tmp")
list.files("tmp") # -> "A" and "B"
setwd("tmp")
file.remove("A") # the tmp/A file
file.symlink(file.path("..", c("A", "B")), ".")
                    # |--> (TRUE,FALSE) : ok for A but not B as it exists already
setwd("..")
unlink("tmp", recursive = TRUE)
file.remove("A", "B", "C")
```

files2 *Manipulation of Directories and File Permissions*

Description

These functions provide a low-level interface to the computer's file system.

Usage

```
dir.exists(paths)
dir.create(path, showWarnings = TRUE, recursive = FALSE, mode = "0777")
Sys.chmod(paths, mode = "0777", use_umask = TRUE)
Sys.umask(mode = NA)
```

Arguments

path	a character vector containing a single path name. Tilde expansion (see path.expand) is done.
paths	character vectors containing file or directory paths. Tilde expansion (see path.expand) is done.
showWarnings	logical; should the warnings on failure be shown?
recursive	logical. Should elements of the path other than the last be created? If true, like the Unix command mkdir -p.
mode	the mode to be used on Unix-alikes: it will be coerced by as.octmode. For Sys.chmod it is recycled along paths.
use_umask	logical: should the mode be restricted by the umask setting?

Details

dir.exists checks that the paths exist (in the same sense as file.exists) and are directories.

dir.create creates the last element of the path, unless recursive = TRUE. Trailing path separators are discarded. The mode will be modified by the umask setting in the same way as for the system function mkdir. What modes can be set is OS-dependent, and it is unsafe to assume that more than three octal digits will be used. For more details see your OS's documentation on the system call mkdir, e.g. man 2 mkdir (and not that on the command-line utility of that name).

One of the idiosyncrasies of Windows is that directory creation may report success but create a directory with a different name, for example dir.create("G.S.") creates '"G.S"'. This is undocumented, and what are the precise circumstances is unknown (and might depend on the version of Windows). Also avoid directory names with a trailing space.

Sys.chmod sets the file permissions of one or more files. It may not be supported on a system (when a warning is issued). See the comments for dir.create for how modes are interpreted. Changing mode on a symbolic link is unlikely to work (nor be necessary). For more details see your OS's documentation on the system call chmod, e.g. man 2 chmod (and not that on the command-line utility of that name). Whether this changes the permission of a symbolic link or its target is OS-dependent

(although to change the target is more common, and POSIX does not support modes for symbolic links: BSD-based Unixes do, though).

`Sys.umask` sets the `umask` and returns the previous value: as a special case `mode = NA` just returns the current value. It may not be supported (when a warning is issued and `"0"` is returned). For more details see your OS's documentation on the system call `umask`, e.g. `man 2 umask`.

How modes are handled depends on the file system, even on Unix-alikes (although their documentation is often written assuming a POSIX file system). So treat documentation cautiously if you are using, say, a FAT/FAT32 or network-mounted file system.

See files for how file paths with marked encodings are interpreted.

Value

`dir.exists` returns a logical vector of `TRUE` or `FALSE` values (without names).

`dir.create` and `Sys.chmod` return invisibly a logical vector indicating if the operation succeeded for each of the files attempted. Using a missing value for a path name will always be regarded as a failure. `dir.create` indicates failure if the directory already exists. If `showWarnings = TRUE`, `dir.create` will give a warning for an unexpected failure (e.g., not for a missing value nor for an already existing component for `recursive = TRUE`).

`Sys.umask` returns the previous value of the `umask`, as a length-one object of class `"octmode"`: the visibility flag is off unless `mode` is `NA`.

See also the section in the help for `file.exists` on case-insensitive file systems for the interpretation of `path` and `paths`.

Author(s)

Ross Ihaka, Brian Ripley

See Also

`file.info`, `file.exists`, `file.path`, `list.files`, `unlink`, `basename`, `path.expand`.

Examples

```
## Not run:
## Fix up maximal allowed permissions in a file tree
Sys.chmod(list.dirs("."), "777")
f <- list.files(".", all.files = TRUE, full.names = TRUE, recursive = TRUE)
Sys.chmod(f, (file.mode(f) | "664"))

## End(Not run)
```

`find.package` *Find Packages*

Description

Find the paths to one or more packages.

Usage

```
find.package(package, lib.loc = NULL, quiet = FALSE,
             verbose = getOption("verbose"))

path.package(package, quiet = FALSE)

packageNotFoundError(package, lib.loc, call = NULL)
```

Arguments

`package`	character vector: the names of packages.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to checking the loaded namespace, then all libraries currently known in `.libPaths()`.
`quiet`	logical. Should this not give warnings or an error if the package is not found?
`verbose`	a logical. If `TRUE`, additional diagnostics are printed, notably when a package is found more than once.
`call`	call expression.

Details

`find.package` returns path to the locations where the given packages are found. If `lib.loc` is `NULL`, then loaded namespaces are searched before the libraries. If a package is found more than once, the first match is used. Unless `quiet = TRUE` a warning will be given about the named packages which are not found, and an error if none are. If `verbose` is true, warnings about packages found more than once are given. For a package to be returned it must contain a either a '`Meta`' subdirectory or a '`DESCRIPTION`' file containing a valid `version` field, but it need not be installed (it could be a source package if `lib.loc` was set suitably).

`find.package` is not usually the right tool to find out if a package is available for use: the only way to do that is to use `require` to try to load it. It need not be installed for the correct platform, it might have a version requirement not met by the running version of R, there might be dependencies which are not available, ...

`path.package` returns the paths from which the named packages were loaded, or if none were named, for all currently attached packages. Unless `quiet = TRUE` it will warn if some of the packages named are not attached, and given an error if none are.

`packageNotFoundError` creates an error condition object of class `packageNotFoundError` for signaling errors. The condition object contains the fields `package` and `lib.loc`.

Value

A character vector of paths of package directories.

See Also

`path.expand` and `normalizePath` for path standardization.

Examples

```
try(find.package("knitr"))
## will not give an error, maybe a warning about *all* locations it is found:
find.package("kitty", quiet=TRUE, verbose=TRUE)

## Find all .libPaths() entries a package is found:
findPkgAll <- function(pkg)
  unlist(lapply(.libPaths(), function(lib)
           find.package(pkg, lib, quiet=TRUE, verbose=FALSE)))

findPkgAll("MASS")
findPkgAll("knitr")
```

---

findInterval *Find Interval Numbers or Indices*

Description

Given a vector of non-decreasing breakpoints in `vec`, find the interval containing each element of `x`; i.e., if `i <- findInterval(x,v)`, for each index `j` in `x` $v_{i_j} \le x_j < v_{i_j+1}$ where $v_0 := -\infty$, $v_{N+1} := +\infty$, and `N <- length(v)`. At the two boundaries, the returned index may differ by 1, depending on the optional arguments `rightmost.closed` and `all.inside`.

Usage

```
findInterval(x, vec, rightmost.closed = FALSE, all.inside = FALSE,
             left.open = FALSE)
```

Arguments

`x`	numeric.
`vec`	numeric, sorted (weakly) increasingly, of length `N`, say.
`rightmost.closed`	logical; if true, the rightmost interval, `vec[N-1] .. vec[N]` is treated as *closed*, see below.
`all.inside`	logical; if true, the returned indices are coerced into `1, . . . ,N-1`, i.e., `0` is mapped to `1` and `N` to `N-1`.
`left.open`	logical; if true all the intervals are open at left and closed at right; in the formulas below, $\le$ should be swapped with $<$ (and $>$ with $\ge$), and `rightmost.closed` means 'leftmost is closed'. This may be useful, e.g., in survival analysis computations.

Details

The function `findInterval` finds the index of one vector `x` in another, `vec`, where the latter must be non-decreasing. Where this is trivial, equivalent to `apply( outer(x, vec, `>=`), 1, sum)`, as a matter of fact, the internal algorithm uses interval search ensuring $O(n \log N)$ complexity where `n <- length(x)` (and `N <- length(vec)`). For (almost) sorted `x`, it will be even faster, basically $O(n)$.

This is the same computation as for the empirical distribution function, and indeed, `findInterval(t, sort(X))` is *identical* to $nF_n(t; X_1, \ldots, X_n)$ where F_n is the empirical distribution function of $X_1, \ldots, X_n$.

When `rightmost.closed = TRUE`, the result for `x[j] = vec[N]` ($= \max vec$), is `N - 1` as for all other values in the last interval.

`left.open = TRUE` is occasionally useful, e.g., for survival data. For (anti-)symmetry reasons, it is equivalent to using "mirrored" data, i.e., the following is always true:

```
identical(
      findInterval( x,  v,      left.open= TRUE, ...) ,
  N - findInterval(-x, -v[N:1], left.open=FALSE, ...) )
```

where `N <- length(vec)` as above.

Value

vector of length `length(x)` with values in `0:N` (and `NA`) where `N <- length(vec)`, or values coerced to `1:(N-1)` if and only if `all.inside = TRUE` (equivalently coercing all `x` values *inside* the intervals). Note that `NA`s are propagated from `x`, and `Inf` values are allowed in both `x` and `vec`.

Author(s)

Martin Maechler

See Also

`approx(*, method = "constant")` which is a generalization of `findInterval()`, `ecdf` for computing the empirical distribution function which is (up to a factor of n) also basically the same as `findInterval(.)`.

Examples

```
x <- 2:18
v <- c(5, 10, 15) # create two bins [5,10) and [10,15)
cbind(x, findInterval(x, v))

N <- 100
X <- sort(round(stats::rt(N, df = 2), 2))
tt <- c(-100, seq(-2, 2, length.out = 201), +100)
it <- findInterval(tt, X)
tt[it < 1 | it >= N] # only first and last are outside range(X)
```

```
##  'left.open = TRUE' means  "mirroring" :
N <- length(v)
stopifnot(identical(
                  findInterval( x,  v,  left.open=TRUE) ,
             N - findInterval(-x, -v[N:1])))
```

`force` *Force Evaluation of an Argument*

Description

Forces the evaluation of a function argument.

Usage

```
force(x)
```

Arguments

`x`	a formal argument of the enclosing function.

Details

`force` forces the evaluation of a formal argument. This can be useful if the argument will be captured in a closure by the lexical scoping rules and will later be altered by an explicit assignment or an implicit assignment in a loop or an apply function.

Note

This is semantic sugar: just evaluating the symbol will do the same thing (see the examples).

`force` does not force the evaluation of other promises. (It works by forcing the promise that is created when the actual arguments of a call are matched to the formal arguments of a closure, the mechanism which implements *lazy evaluation*.)

Examples

```
f <- function(y) function() y
lf <- vector("list", 5)
for (i in seq_along(lf)) lf[[i]] <- f(i)
lf[[1]]()  # returns 5

g <- function(y) { force(y); function() y }
lg <- vector("list", 5)
for (i in seq_along(lg)) lg[[i]] <- g(i)
lg[[1]]()  # returns 1

## This is identical to
g <- function(y) { y; function() y }
```

forceAndCall *Call a function with Some Arguments Forced*

Description

Call a function with a specified number of leading arguments forced before the call if the function is a closure.

Usage

```
forceAndCall(n, FUN, ...)
```

Arguments

n	number of leading arguments to force.
FUN	function to call.
...	arguments to FUN.

Details

forceAndCall calls the function FUN with arguments specified in .... If the value of FUN is a closure then the first n arguments to the function are evaluated (i.e. their delayed evaluation promises are forced) before executing the function body. If the value of FUN is a primitive then the call FUN(...) is evaluated in the usual way.

forceAndCall is intended to help defining higher order functions like apply to behave more reasonably when the result returned by the function applied is a closure that captured its arguments.

See Also

force, promise, closure.

Foreign *Foreign Function Interface*

Description

Functions to make calls to compiled code that has been loaded into R.

Usage

```
      .C(.NAME, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING)
.Fortran(.NAME, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING)
```

Arguments

`.NAME`	a character string giving the name of a C function or Fortran subroutine, or an object of class `"NativeSymbolInfo"`, `"RegisteredNativeSymbol"` or `"NativeSymbol"` referring to such a name.
`...`	arguments to be passed to the foreign function. Up to 65.
`NAOK`	if `TRUE` then any `NA` or `NaN` or `Inf` values in the arguments are passed on to the foreign function. If `FALSE`, the presence of `NA` or `NaN` or `Inf` values is regarded as an error.
`PACKAGE`	if supplied, confine the search for a character string `.NAME` to the DLL given by this argument (plus the conventional extension, '`.so`', '`.dll`', ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols, and also speeds up the search (see 'Note').
`DUP, ENCODING`	For back-compatibility, accepted but ignored.

Details

These functions can be used to make calls to compiled C and Fortran code. Later interfaces are `.Call` and `.External` which are more flexible and have better performance.

These functions are primitive, and `.NAME` is always matched to the first argument supplied (which should not be named). The other named arguments follow `...` and so cannot be abbreviated. For clarity, should avoid using names in the arguments passed to `...` that match or partially match `.NAME`.

Value

A list similar to the `...` list of arguments passed in (including any names given to the arguments), but reflecting any changes made by the C or Fortran code.

Argument types

The mapping of the types of R arguments to C or Fortran arguments is

R	C	Fortran
`integer`	`int *`	`integer`
`numeric`	`double *`	`double precision`
– or –	`float *`	`real`
`complex`	`Rcomplex *`	`double complex`
`logical`	`int *`	`integer`
`character`	`char **`	[see below]
`raw`	`unsigned char *`	not allowed
`list`	`SEXP *`	not allowed
other	`SEXP`	not allowed

Note: The C types corresponding to `integer` and `logical` are `int`, not `long` as in S. This difference matters on most 64-bit platforms, where `int` is 32-bit and `long` is 64-bit (but not on 64-bit Windows).

Note: The Fortran type corresponding to `logical` is `integer`, not `logical`: the difference matters on some Fortran compilers.

Numeric vectors in R will be passed as type `double *` to C (and as `double precision` to Fortran) unless the argument has attribute `Csingle` set to `TRUE` (use `as.single` or `single`). This mechanism is only intended to be used to facilitate the interfacing of existing C and Fortran code.

The C type `Rcomplex` is defined in 'Complex.h' as a `typedef struct {double r; double i;}`. It may or may not be equivalent to the C99 `double complex` type, depending on the compiler used.

Logical values are sent as `0` (`FALSE`), `1` (`TRUE`) or `INT_MIN` = `-2147483648` (`NA`, but only if `NAOK` = `TRUE`), and the compiled code should return one of these three values: however non-zero values other than `INT_MIN` are mapped to `TRUE`.

Missing (`NA`) string values are passed to `.C` as the string "NA". As the C `char` type can represent all possible bit patterns there appears to be no way to distinguish missing strings from the string `"NA"`. If this distinction is important use `.Call`.

Using a character string with `.Fortran` is deprecated and will give a warning. It passes the first (only) character string of a character vector as a C character array to Fortran: that may be usable as `character*255` if its true length is passed separately. Only up to 255 characters of the string are passed back. (How well this works, and even if it works at all, depends on the C and Fortran compilers and the platform.)

Lists, functions or other R objects can (for historical reasons) be passed to `.C`, but the `.Call` interface is much preferred. All inputs apart from atomic vectors should be regarded as read-only, and all apart from vectors (including lists), functions and environments are now deprecated.

Fortran symbol names

All Fortran compilers known to be usable to compile R map symbol names to lower case, and so does `.Fortran`.

Symbol names containing underscores are not valid Fortran 77 (although they are valid in Fortran 9x). Many Fortran 77 compilers will allow them but may translate them in a different way to names not containing underscores. Such names will often work with `.Fortran` (since how they are translated is detected when R is built and the information used by `.Fortran`), but portable code should not use Fortran names containing underscores.

Use `.Fortran` with care for compiled Fortran 9x code: it may not work if the Fortran 9x compiler used differs from the Fortran compiler used when configuring R, especially if the subroutine name is not lower-case or includes an underscore. The most portable way to call Fortran 9x code from R is to use `.C` and the Fortran 2003 module `iso_c_binding` to provide a C interface to the Fortran code.

Copying of arguments

Character vectors are copied before calling the compiled code and to collect the results. For other atomic vectors the argument is copied before calling the compiled code if it is otherwise used in the calling code.

Non-atomic-vector objects are read-only to the C code and are never copied.

This behaviour can be changed by setting `options(CBoundsCheck = TRUE)`. In that case raw, logical, integer, double and complex vector arguments are copied both before and after calling the

compiled code. The first copy made is extended at each end by guard bytes, and on return it is checked that these are unaltered. For .C, each element of a character vector uses guard bytes.

Note

If one of these functions is to be used frequently, do specify PACKAGE (to confine the search to a single DLL) or pass .NAME as one of the native symbol objects. Searching for symbols can take a long time, especially when many namespaces are loaded.

You may see PACKAGE = "base" for symbols linked into R. Do not use this in your own code: such symbols are not part of the API and may be changed without warning.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

dyn.load, .Call.

The 'Writing R Extensions' manual.

---

formals *Access to and Manipulation of the Formal Arguments*

Description

Get or set the formal arguments of a function.

Usage

```
formals(fun = sys.function(sys.parent()), envir = parent.frame())
formals(fun, envir = environment(fun)) <- value
```

Arguments

fun	a function, or see 'Details'.
envir	environment in which the function should be defined (or found via get() in the first case and when fun a character string).
value	a list (or pairlist, hence possibly NULL) of R expressions.

Details

For the first form, fun can also be a character string naming the function to be manipulated, which is searched for in envir, by default from the parent frame. If it is not specified, the function calling formals is used.

Only *closures*, i.e., non-primitive functions, have formals, not primitive functions.
Note that formals(args(f)) gives a formal argument list for all functions f, primitive or not.

Value

`formals` returns the formal argument list of the function specified, as a `pairlist`, or `NULL` for a non-function or primitive.

The replacement form sets the formals of a function to the list/pairlist on the right hand side, and (potentially) resets the environment of the function, dropping `attributes`.

See Also

`formalArgs` (from **methods**), a shortcut for `names(formals(.))`. `args` for a human-readable version, *and* as intermediary to get formals of a primitive function.
`alist` to *construct* a typical formals value, see the examples.

The three parts of a (non-primitive) `function` are its `formals`, `body`, and `environment`.

Examples

```
require(stats)
formals(lm)

## If you just want the names of the arguments, use formalArgs instead.
names(formals(lm))
methods:: formalArgs(lm)     # same

## formals returns a pairlist. Arguments with no default have type symbol (aka name).
str(formals(lm))

## formals returns NULL for primitive functions.  Use it in combination with
## args for this case.
is.primitive(`+`)
formals(`+`)
formals(args(`+`))

## You can overwrite the formal arguments of a function (though this is
## advanced, dangerous coding).
f <- function(x) a + b
formals(f) <- alist(a = , b = 3)
f    # function(a, b = 3) a + b
f(2) # result = 5
```

---

format *Encode in a Common Format*

Description

Format an R object for pretty printing.

Usage

```
format(x, ...)

## Default S3 method:
format(x, trim = FALSE, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark   = "",   big.interval = 3L,
       small.mark = "", small.interval = 5L,
       decimal.mark = getOption("OutDec"),
       zero.print = NULL, drop0trailing = FALSE, ...)

## S3 method for class 'data.frame'
format(x, ..., justify = "none")

## S3 method for class 'factor'
format(x, ...)

## S3 method for class 'AsIs'
format(x, width = 12, ...)
```

Arguments

x	any R object (conceptually); typically numeric.
trim	logical; if FALSE, logical, numeric and complex values are right-justified to a common width: if TRUE the leading blanks for justification are suppressed.
digits	a positive integer indicating how many significant digits are to be used for numeric and complex x. The default, NULL, uses getOption("digits"). This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits, and also to satisfy nsmall. (For more, notably the interpretation for complex numbers see signif.)
nsmall	the minimum number of digits to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values are 0 <= nsmall <= 20.
justify	should a *character* vector be left-justified (the default), right-justified, centred or left alone. Can be abbreviated.
width	default method: the *minimum* field width or NULL or 0 for no restriction. AsIs method: the *maximum* field width for non-character objects. NULL corresponds to the default 12.
na.encode	logical: should NA strings be encoded? Note this only applies to elements of character vectors, not to numerical, complex nor logical NAs, which are always encoded as "NA".
scientific	either a logical specifying whether elements of a real or complex vector should be encoded in scientific format, or an integer penalty (see options("scipen")). Missing values correspond to the current default penalty.
...	further arguments passed to or from other methods.

`big.mark`, `big.interval`, `small.mark`, `small.interval`, `decimal.mark`, `zero.print`, `drop0trailing`
: used for prettying (longish) numerical and complex sequences. Passed to `prettyNum`: that help page explains the details.

Details

`format` is a generic function. Apart from the methods described here there are methods for dates (see `format.Date`), date-times (see `format.POSIXct`) and for other classes such as `format.octmode` and `format.dist`.

`format.data.frame` formats the data frame column by column, applying the appropriate method of `format` for each column. Methods for columns are often similar to `as.character` but offer more control. Matrix and data-frame columns will be converted to separate columns in the result, and character columns (normally all) will be given class `"AsIs"`.

`format.factor` converts the factor to a character vector and then calls the default method (and so `justify` applies).

`format.AsIs` deals with columns of complicated objects that have been extracted from a data frame. Character objects and (atomic) matrices are passed to the default method (and so `width` does not apply). Otherwise it calls `toString` to convert the object to character (if a vector or list, element by element) and then right-justifies the result.

Justification for character vectors (and objects converted to character vectors by their methods) is done on display width (see `nchar`), taking double-width characters and the rendering of special characters (as escape sequences, including escaping backslash but not double quote: see `print.default`) into account. Thus the width is as displayed by `print(quote = FALSE)` and not as displayed by `cat`. Character strings are padded with blanks to the display width of the widest. (If `na.encode = FALSE` missing character strings are not included in the width computations and are not encoded.)

Numeric vectors are encoded with the minimum number of decimal places needed to display all the elements to at least the `digits` significant digits. However, if all the elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit; see also the argument documentation for `big.*`, `small.*` etc, above. See the note in `print.default` about `digits >= 16`.

Raw vectors are converted to their 2-digit hexadecimal representation by `as.character`.

`format.default(x)` now provides a "minimal" string when `isS4(x)` is true.

While the internal code respects the option `getOption("OutDec")` for the 'decimal mark' in general, `decimal.mark` takes precedence over that option. Similarly, `scientific` takes precedence over `getOption("scipen")`.

Value

An object of similar structure to `x` containing character representations of the elements of the first argument `x` in a common format, and in the current locale's encoding.

For character, numeric, complex or factor `x`, dims and dimnames are preserved on matrices/arrays and names on vectors: no other attributes are copied.

If `x` is a list, the result is a character vector obtained by applying `format.default(x, ...)` to each element of the list (after `unlist`ing elements which are themselves lists), and then collapsing the

result for each element with `paste(collapse = ", ")`. The defaults in this case are `trim = TRUE`, `justify = "none"` since one does not usually want alignment in the collapsed strings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`format.info` indicates how an atomic vector would be formatted.

`formatC`, `paste`, `as.character`, `sprintf`, `print`, `prettyNum`, `toString`, `encodeString`.

Examples

```
format(1:10)
format(1:10, trim = TRUE)

zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names = FALSE)
format(zz)
format(zz, justify = "left")

## use of nsmall
format(13.7)
format(13.7, nsmall = 3)
format(c(6.0, 13.1), digits = 2)
format(c(6.0, 13.1), digits = 2, nsmall = 1)

## use of scientific
format(2^31-1)
format(2^31-1, scientific = TRUE)
## scientific = numeric scipen (= {sci}entific notation {pen}alty) :
x <- c(1e5, 1000, 10, 0.1, .001, .123)
t(sapply(setNames(,-4:1),
         \(sci) sapply(x, format, scientific=sci)))


## a list
z <- list(a = letters[1:3], b = (-pi+0i)^((-2:2)/2), c = c(1,10,100,1000),
          d = c("a", "longer", "character", "string"),
          q = quote( a + b ), e = expression(1+x))
## can you find the "2" small differences?
(f1 <- format(z, digits = 2))
(f2 <- format(z, digits = 2, justify = "left", trim = FALSE))
f1 == f2 ## 2 FALSE, 4 TRUE

## A "minimal" format() for S4 objects without their own format() method:
cc <- methods::getClassDef("standardGeneric")
format(cc) ## "<S4 class ......>"
```

`format.info` *format(.) Information*

Description

Information is returned on how `format(x, digits, nsmall)` would be formatted.

Usage

```
format.info(x, digits = NULL, nsmall = 0)
```

Arguments

`x`	an atomic vector; a potential argument of `format(x, ...)`.
`digits`	how many significant digits are to be used for numeric and complex `x`. The default, `NULL`, uses `getOption("digits")`.
`nsmall`	(see `format(..., nsmall)`).

Value

An `integer vector` of length 1, 3 or 6, say `r`.

For logical, integer and character vectors a single element, the width which would be used by `format` if `width = NULL`.

For numeric vectors:

`r[1]`	width (in characters) used by `format(x)`
`r[2]`	number of digits after decimal point.
`r[3]`	in `0:2`; if ≥ 1, *exponential* representation would be used, with exponent length of `r[3]+1`.

For a complex vector the first three elements refer to the real parts, and there are three further elements corresponding to the imaginary parts.

See Also

`format` (notably about `digits >= 16`), `formatC`.

Examples

```
dd <- options("digits") ; options(digits = 7) #-- for the following
format.info(123)   # 3 0 0
format.info(pi)    # 8 6 0
format.info(1e8)   # 5 0 1 - exponential "1e+08"
format.info(1e222) # 6 0 2 - exponential "1e+222"

x <- pi*10^c(-10,-2,0:2,8,20)
names(x) <- formatC(x, width = 1, digits = 3, format = "g")
```

```
cbind(sapply(x, format))
t(sapply(x, format.info))

## using at least 8 digits right of "."
t(sapply(x, format.info, nsmall = 8))

# Reset old options:
options(dd)
```

`format.pval` *Format P Values*

Description

`format.pval` is intended for formatting p-values.

Usage

```
format.pval(pv, digits = max(1, getOption("digits") - 2),
            eps = .Machine$double.eps, na.form = "NA", ...)
```

Arguments

`pv`	a numeric vector.
`digits`	how many significant digits are to be used.
`eps`	a numerical tolerance: see 'Details'.
`na.form`	character representation of `NA`s.
`...`	further arguments to be passed to `format` such as `nsmall`.

Details

`format.pval` is mainly an auxiliary function for `print.summary.lm` etc., and does separate formatting for fixed, floating point and very small values; those less than `eps` are formatted as `"< [eps]"` (where '[eps]' stands for `format(eps, digits)`).

Value

A character vector.

Examples

```
format.pval(c(stats::runif(5), pi^-100, NA))
format.pval(c(0.1, 0.0001, 1e-27))
```

formatC *Formatting Using C-style Formats*

Description

`formatC()` formats numbers individually and flexibly using `C` style format specifications.

`prettyNum()` is used for "prettifying" (possibly formatted) numbers, also in `format.default`.

`.format.zeros(x)`, an auxiliary function of `prettyNum()`, re-formats the zeros in a vector `x` of formatted numbers.

Usage

```
formatC(x, digits = NULL, width = NULL,
        format = NULL, flag = "", mode = NULL,
        big.mark = "", big.interval = 3L,
        small.mark = "", small.interval = 5L,
        decimal.mark = getOption("OutDec"),
        preserve.width = "individual",
        zero.print = NULL, replace.zero = TRUE,
        drop0trailing = FALSE)

prettyNum(x, big.mark = "",   big.interval = 3L,
          small.mark  = "", small.interval = 5L,
          decimal.mark = getOption("OutDec"), input.d.mark = decimal.mark,
          preserve.width = c("common", "individual", "none"),
          zero.print = NULL, replace.zero = FALSE,
          drop0trailing = FALSE, is.cmplx = NA,
          ...)

.format.zeros(x, zero.print, nx = suppressWarnings(as.numeric(x)),
              replace = FALSE, warn.non.fitting = TRUE)
```

Arguments

`x`	an atomic numerical or character object, possibly `complex` only for `prettyNum()`, typically a vector of real numbers. Any class is discarded, with a warning.
`digits`	the desired number of digits after the decimal point (`format = "f"`) or *significant* digits (`format = "g"`, `= "e"` or `= "fg"`). Default: 2 for integer, 4 for real numbers. If less than 0, the C default of 6 digits is used. If specified as more than 50, 50 will be used with a warning unless `format = "f"` where it is limited to typically 324. (Not more than 15–21 digits need be accurate, depending on the OS and compiler used. This limit is just a precaution against segfaults in the underlying C runtime.)

`width` the total field width; if both `digits` and `width` are unspecified, `width` defaults to 1, otherwise to `digits + 1`. `width = 0` will use `width = digits`, `width < 0` means left justify the number in this field (equivalent to `flag = "-"`). If necessary, the result will have more characters than `width`. For character data this is interpreted in characters (not bytes nor display width).

`format` equal to `"d"` (for integers), `"f"`, `"e"`, `"E"`, `"g"`, `"G"`, `"fg"` (for reals), or `"s"` (for strings). Default is `"d"` for integers, `"g"` for reals.

`"f"` gives numbers in the usual `xxx.xxx` format; `"e"` and `"E"` give `n.ddde+nn` or `n.dddE+nn` (scientific format); `"g"` and `"G"` put `x[i]` into scientific format only if it saves space to do so *and* drop trailing zeros and decimal point - unless `flag` contains `"#"` which keeps trailing zeros for the `"g"`, `"G"` formats.

`"fg"` (our own hybrid format) uses fixed format as `"f"`, but `digits` as the minimum number of *significant* digits. This can lead to quite long result strings, see examples below. Note that unlike `signif` this prints large numbers with more significant digits than `digits`. Trailing zeros are *dropped* in this format, unless `flag` contains `"#"`.

`flag` for `formatC`, a character string giving a format modifier as in Kernighan and Ritchie (1988, page 243) or the C+99 standard.

`"0"` pads leading zeros;

`"-"` does left adjustment,

`"+"` ensures a sign in all cases, i.e., `"+"` for positive numbers ,

`" "` if the first character is not a sign, the space character `" "` will be used instead.

`"#"` specifies "an alternative output form", specifically depending on `format`.

`"'"` on some platform–locale combination, activates "thousands' grouping" for decimal conversion,

`"I"` in some versions of 'glibc' allow for integer conversion to use the locale's alternative output digits, if any.

There can be more than one of these flags, in any order. Other characters used to have no effect for `character` formatting, but signal an error since R 3.4.0.

`mode` `"double"` (or `"real"`), `"integer"` or `"character"`. Default: Determined from the storage mode of `x`.

`big.mark` character; if not empty used as mark between every `big.interval` decimals *before* (hence `big`) the decimal point.

`big.interval` see `big.mark` above; defaults to 3.

`small.mark` character; if not empty used as mark between every `small.interval` decimals *after* (hence `small`) the decimal point.

`small.interval` see `small.mark` above; defaults to 5.

`decimal.mark` the character to be used to indicate the numeric decimal point.

`input.d.mark` if `x` is `character`, the character known to have been used as the numeric decimal point in `x`.

`preserve.width` string specifying if the string widths should be preserved where possible in those cases where marks (`big.mark` or `small.mark`) are added. `"common"`, the default, corresponds to `format`-like behavior whereas `"individual"` is the default in `formatC()`. Value can be abbreviated.

`zero.print` logical, character string or `NULL` specifying if and how *zeros* should be formatted specially. Useful for pretty printing 'sparse' objects.

`replace.zero, replace`
logical; if `zero.print` is a character string, indicates if the exact zero entries in `x` should be simply replaced by `zero.print`. Otherwise, depending on the widths of the respective strings, the (formatted) zeroes are *partly* replaced by `zero.print` and then padded with `" "` to the right were applicable. In that case (false `replace[.zero]`), if the `zero.print` string does not fit, a warning is produced (if `warn.non.fitting` is true).

This works via `prettyNum()`, which calls `.format.zeros(*, replace=replace.zero)` three times in this case, see the 'Details'.

`warn.non.fitting`
logical; if it is true, `replace[.zero]` is false and the `zero.print` string does not fit, a `warning` is signalled.

`drop0trailing` logical, indicating if trailing zeros, i.e., `"0"` *after* the decimal mark, should be removed; also drops `"e+00"` in exponential formats. This is simply passed to `prettyNum()`, see the 'Details'.

`is.cmplx` optional logical, to be used when `x` is `"character"` to indicate if it stems from `complex` vector or not. By default (`NA`), `x` is checked to 'look like' complex.

`...` arguments passed to `format`.

`nx` numeric vector of the same length as `x`, typically the numbers of which the character vector `x` is the pre-format.

Details

For numbers, `formatC()` calls `prettyNum()` when needed which itself calls `.format.zeros(*, replace=replace.zero)`. (*"when needed"*: when `zero.print` is not `NULL`, `drop0trailing` is true, or one of `big.mark`, `small.mark`, or `decimal.mark` is not at default.)

If you set `format` it overrides the setting of `mode`, so `formatC(123.45, mode = "double", format = "d")` gives `123`.

The rendering of scientific format is platform-dependent: some systems use `n.ddde+nnn` or `n.dddenn` rather than `n.ddde+nn`.

`formatC` does not necessarily align the numbers on the decimal point, so `formatC(c(6.11, 13.1), digits = 2, format = "fg")` gives `c("6.1", " 13")`. If you want common formatting for several numbers, use `format`.

`prettyNum` is the utility function for prettifying `x`. `x` can be complex (or `format(<complex>)`), here. If `x` is not a character, `format(x[i], ...)` is applied to each element, and then it is left unchanged if all the other arguments are at their defaults. Use the `input.d.mark` argument for `prettyNum(x)` when `x` is a `character` vector not resulting from something like `format(<number>)` with a period as decimal mark.

Because `gsub` is used to insert the `big.mark` and `small.mark`, special characters need escaping. In particular, to insert a single backslash, use `"\\\\"`.

The C doubles used for R numerical vectors have signed zeros, which `formatC` may output as `-0`, `-0.000` ....

There is a warning if `big.mark` and `decimal.mark` are the same: that would be confusing to those reading the output.

Value

A character object of same size and attributes as x (after discarding any class), in the current locale's encoding.

Unlike format, each number is formatted individually. Looping over each element of x, the C function sprintf(...) is called for numeric inputs (inside the C function str_signif).

formatC: for character x, do simple (left or right) padding with white space.

Note

The default for decimal.mark in formatC() was changed in R 3.2.0: for use within print methods in packages which might be used with earlier versions: use decimal.mark = getOption("OutDec") explicitly.

Author(s)

formatC was originally written by Bill Dunlap for S-PLUS, later much improved by Martin Maechler.

It was first adapted for R by Friedrich Leisch and since much improved by the R Core team.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition. Prentice Hall.

See Also

format.

sprintf for more general C-like formatting.

Examples

```
xx <- pi * 10^(-5:4)
cbind(format(xx, digits = 4), formatC(xx))
cbind(formatC(xx, width = 9, flag = "-"))
cbind(formatC(xx, digits = 5, width = 8, format = "f", flag = "0"))
cbind(format(xx, digits = 4), formatC(xx, digits = 4, format = "fg"))

f <- (-2:4); f <- f*16^f
# Default ("g") format:
formatC(pi*f)
# Fixed ("f") format, more than one flag ('width' partly "enlarged"):
cbind(formatC(pi*f, digits = 3, width=9, format = "f", flag = "0+"))

formatC(    c("a", "Abc", "no way"), width = -7)  # <=> flag = "-"
formatC(c((-1:1)/0,c(1,100)*pi), width = 8, digits = 1)

## note that some of the results here depend on the implementation
## of long-double arithmetic, which is platform-specific.
xx <- c(1e-12,-3.98765e-10,1.45645e-69,1e-70,pi*1e37,3.44e4)
```

```
##       1        2             3        4      5       6
formatC(xx)
formatC(xx, format = "fg")       # special "fixed" format.
formatC(xx[1:4], format = "f", digits = 75) #>> even longer strings

formatC(c(3.24, 2.3e-6), format = "f", digits = 11)
formatC(c(3.24, 2.3e-6), format = "f", digits = 11, drop0trailing = TRUE)

r <- c("76491283764.97430", "29.12345678901", "-7.1234", "-100.1","1123")
## American:
prettyNum(r, big.mark = ",")
## Some Europeans:
prettyNum(r, big.mark = "'", decimal.mark = ",")

(dd <- sapply(1:10, function(i) paste((9:0)[1:i], collapse = "")))
prettyNum(dd, big.mark = "'")

## examples of 'small.mark'
pN <- stats::pnorm(1:7, lower.tail = FALSE)
cbind(format (pN, small.mark = " ", digits = 15))
cbind(formatC(pN, small.mark = " ", digits = 17, format = "f"))

cbind(ff <- format(1.2345 + 10^(0:5), width = 11, big.mark = "'"))
## all with same width (one more than the specified minimum)

## individual formatting to common width:
fc <- formatC(1.234 + 10^(0:8), format = "fg", width = 11, big.mark = "'")
cbind(fc)
## Powers of two, stored exactly, formatted individually:
pow.2 <- formatC(2^-(1:32), digits = 24, width = 1, format = "fg")
## nicely printed (the last line showing 5^32 exactly):
noquote(cbind(pow.2))

## complex numbers:
r <- 10.0000001; rv <- (r/10)^(1:10)
(zv <- (rv + 1i*rv))
op <- options(digits = 7) ## (system default)
(pnv <- prettyNum(zv))
stopifnot(pnv == "1+1i", pnv == format(zv),
          pnv == prettyNum(zv, drop0trailing = TRUE))
## more digits change the picture:
options(digits = 8)
head(fv <- format(zv), 3)
prettyNum(fv)
prettyNum(fv, drop0trailing = TRUE) # a bit nicer
options(op)

## The  '  flag :
doLC <- FALSE # <= R warns, so change to TRUE manually if you want see the effect
if(doLC) {
  oldLC <- Sys.getlocale("LC_NUMERIC")
           Sys.setlocale("LC_NUMERIC", "de_CH.UTF-8")
}
```

```
formatC(1.234 + 10^(0:4), format = "fg", width = 11, flag = "'")
## -->  .....  "      1'001" "     10'001"   on supported platforms
if(doLC) ## revert, typically to  "C"  :
  Sys.setlocale("LC_NUMERIC", oldLC)
```

`formatDL` *Format Description Lists*

Description

Format vectors of items and their descriptions as 2-column tables or LaTeX-style description lists.

Usage

```
formatDL(x, y, style = c("table", "list"),
         width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

`x`	a vector giving the items to be described, or a list of length 2 or a matrix with 2 columns giving both items and descriptions.
`y`	a vector of the same length as `x` with the corresponding descriptions. Only used if `x` does not already give the descriptions.
`style`	a character string specifying the rendering style of the description information. Can be abbreviated. If `"table"`, a two-column table with items and descriptions as columns is produced (similar to Texinfo's `@table` environment). If `"list"`, a LaTeX-style tagged description list is obtained.
`width`	a positive integer giving the target column for wrapping lines in the output.
`indent`	a positive integer specifying the indentation of the second column in table style, and the indentation of continuation lines in list style. Must not be greater than `width/2`, and defaults to `width/3` for table style and `width/9` for list style.

Details

After extracting the vectors of items and corresponding descriptions from the arguments, both are coerced to character vectors.

In table style, items with more than `indent - 3` characters are displayed on a line of their own.

Value

a character vector with the formatted entries.

Examples

```
## Provide a nice summary of the numerical characteristics of the
## machine R is running on:
writeLines(formatDL(unlist(.Machine)))
## Inspect Sys.getenv() results in "list" style (by default, these are
## printed in "table" style):
writeLines(formatDL(Sys.getenv(), style = "list"))
```

---

function *Function Definition*

Description

These functions provide the base mechanisms for defining new functions in the R language.

Usage

```
function( arglist ) expr
\( arglist ) expr
return(value)
```

Arguments

arglist	empty or one or more (comma-separated) 'name' or 'name = expression' terms and/or the special token ....
expr	an expression.
value	an expression.

Details

The names in an argument list can be back-quoted non-standard names (see 'backquote').

If value is missing, NULL is returned. If it is a single expression, the value of the evaluated expression is returned. (The expression is evaluated as soon as return is called, in the evaluation frame of the function and before any on.exit expression is evaluated.)

If the end of a function is reached without calling return, the value of the last evaluated expression is returned.

The shorthand form \(x) x + 1 is parsed as function(x) x + 1. It may be helpful in making code containing simple function expressions more readable.

Technical details

This type of function is not the only type in R: they are called *closures* (a name with origins in LISP) to distinguish them from primitive functions.

A closure has three components, its formals (its argument list), its body (expr in the 'Usage' section) and its environment which provides the enclosure of the evaluation frame when the closure is used.

There is an optional further component if the closure has been byte-compiled. This is not normally user-visible, but is indicated when functions are printed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`args`.

`formals`, `body` and `environment` for accessing the component parts of a function.

`debug` for debugging; using `invisible` inside `return(.)` for returning *invisibly*.

Examples

```
norm <- function(x) sqrt(x%*%x)
norm(1:4)

## An anonymous function:
(function(x, y){ z <- x^2 + y^2; x+y+z })(0:7, 1)
```

`funprog` *Common Higher-Order Functions in Functional Programming Languages*

Description

`Reduce` uses a binary function to successively combine the elements of a given vector and a possibly given initial value.

`Filter` extracts the elements of a vector for which a predicate (logical) function gives true.

`Find` **and** `Position` give the first or last such element and its position in the vector, respectively.

`Map` applies a function to the corresponding elements of given vectors.

`Negate` creates the negation of a given function.

Usage

```
Reduce(f, x, init, right = FALSE, accumulate = FALSE, simplify = TRUE)
Filter(f, x)
Find(f, x, right = FALSE, nomatch = NULL)
Map(f, ...)
Negate(f)
Position(f, x, right = FALSE, nomatch = NA_integer_)
```

Arguments

f	a function of the appropriate arity (binary for Reduce, unary for Filter, Find and Position, k-ary for Map if this is called with k arguments). An arbitrary predicate function for Negate.
x	a vector.
init	an R object of the same kind as the elements of x.
right	a logical indicating whether to proceed from left to right (default) or from right to left.
accumulate	a logical indicating whether the successive reduce combinations should be accumulated. By default, only the final combination is used.
simplify	a logical indicating whether accumulated results should be simplified (by unlisting) in case they all are length one.
nomatch	the value to be returned in the case when "no match" (no element satisfying the predicate) is found.
...	vectors to which the function is Map()ped, and other arguments of mapply passed to it, e.g., MoreArgs.

Details

If init is given, Reduce logically adds it to the start (when proceeding left to right) or the end of x, respectively. If this possibly augmented vector v has $n > 1$ elements, Reduce successively applies f to the elements of v from left to right or right to left, respectively. I.e., a left reduce computes $l_1 = f(v_1, v_2)$, $l_2 = f(l_1, v_3)$, etc., and returns $l_{n-1} = f(l_{n-2}, v_n)$, and a right reduce does $r_{n-1} = f(v_{n-1}, v_n)$, $r_{n-2} = f(v_{n-2}, r_{n-1})$ and returns $r_1 = f(v_1, r_2)$. (E.g., if v is the sequence (2, 3, 4) and f is division, left and right reduce give $(2/3)/4 = 1/6$ and $2/(3/4) = 8/3$, respectively.) If v has only a single element, this is returned; if there are no elements, NULL is returned. Thus, it is ensured that f is always called with 2 arguments.

The current implementation is non-recursive to ensure stability and scalability.

Reduce is patterned after Common Lisp's reduce. A reduce is also known as a fold (e.g., in Haskell) or an accumulate (e.g., in the C++ Standard Template Library). The accumulative version corresponds to Haskell's scan functions.

Filter applies the unary predicate function f to each element of x, coercing to logical if necessary, and returns the subset of x for which this gives true. Note that possible NA values are currently always taken as false; control over NA handling may be added in the future. Filter corresponds to filter in Haskell or 'remove-if-not' in Common Lisp.

Find and Position are patterned after Common Lisp's 'find-if' and 'position-if', respectively. If there is an element for which the predicate function gives true, then the first or last such element or its position is returned depending on whether right is false (default) or true, respectively. If there is no such element, the value specified by nomatch is returned. The current implementation is not optimized for performance.

Map is a simple wrapper to mapply which does not attempt to simplify the result, similar to Common Lisp's mapcar (with arguments being recycled, however). Future versions may allow some control of the result type.

Negate corresponds to Common Lisp's complement. Given a (predicate) function f, it creates a function which returns the logical negation of what f returns.

See Also

Function `clusterMap` and `mcmapply` (not Windows) in package **parallel** provide parallel versions of `Map`.

Examples

```
## A general-purpose adder:
add <- function(x) Reduce(`+`, x)
add(list(1, 2, 3))
## Like sum(), but can also used for adding matrices etc., as it will
## use the appropriate '+' method in each reduction step.
## More generally, many generics meant to work on arbitrarily many
## arguments can be defined via reduction:
FOO <- function(...) Reduce(FOO2, list(...))
FOO2 <- function(x, y) UseMethod("FOO2")
## FOO() methods can then be provided via FOO2() methods.

## A general-purpose cumulative adder:
cadd <- function(x) Reduce(`+`, x, accumulate = TRUE)
cadd(seq_len(7))

## A simple function to compute continued fractions:
cfrac <- function(x) Reduce(function(u, v) u + 1 / v, x, right = TRUE)
## Continued fraction approximation for pi:
cfrac(c(3, 7, 15, 1, 292))
## Continued fraction approximation for Euler's number (e):
cfrac(c(2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8))

## Map() now recycles similar to basic Ops:
Map(`+`, 1,         1 : 3) ;         1 + 1:3
Map(`+`, numeric(), 1 : 3) ; numeric() + 1:3

## Iterative function application:
Funcall <- function(f, ...) f(...)
## Compute log(exp(acos(cos(0))))
Reduce(Funcall, list(log, exp, acos, cos), 0, right = TRUE)
## n-fold iterate of a function, functional style:
Iterate <- function(f, n = 1)
    function(x) Reduce(Funcall, rep.int(list(f), n), x, right = TRUE)
## Continued fraction approximation to the golden ratio:
Iterate(function(x) 1 + 1 / x, 30)(1)
## which is the same as
cfrac(rep.int(1, 31))
## Computing square root approximations for x as fixed points of the
## function t |-> (t + x / t) / 2, as a function of the initial value:
asqrt <- function(x, n) Iterate(function(t) (t + x / t) / 2, n)
asqrt(2, 30)(10) # Starting from a positive value => +sqrt(2)
asqrt(2, 30)(-1) # Starting from a negative value => -sqrt(2)

## A list of all functions in the base environment:
funs <- Filter(is.function, sapply(ls(baseenv()), get, baseenv()))
## Functions in base with more than 10 arguments:
```

```
names(Filter(function(f) length(formals(f)) > 10, funs))
## Number of functions in base with a '...' argument:
length(Filter(function(f)
              any(names(formals(f)) %in% "..."),
              funs))

## Find all objects in the base environment which are *not* functions:
Filter(Negate(is.function),  sapply(ls(baseenv()), get, baseenv()))
```

---

gc *Garbage Collection*

Description

A call of gc causes a garbage collection to take place. gcinfo sets a flag so that automatic collection is either silent (verbose = FALSE) or prints memory usage statistics (verbose = TRUE).

Usage

```
gc(verbose = getOption("verbose"), reset = FALSE, full = TRUE)
gcinfo(verbose)
```

Arguments

verbose	logical; if TRUE, the garbage collection prints statistics about cons cells and the space allocated for vectors.
reset	logical; if TRUE the values for maximum space used are reset to the current values.
full	logical; if TRUE a full collection is performed; otherwise only more recently allocated objects may be collected.

Details

A call of gc causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling gc is for the report on memory usage. For an accurate report full = TRUE should be used.

It can be useful to call gc after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of "Vcells", a relic of an earlier allocator (that used a vector heap).

When gcinfo(TRUE) is in force, messages are sent to the message connection at each garbage collection of the form

```
Garbage collection 12 = 10+0+2 (level 0) ...
6.4 Mbytes of cons cells used (58%)
2.0 Mbytes of vectors used (32%)
```

Here the last two lines give the current memory usage rounded up to the next 0.1Mb and as a percentage of the current trigger value. The first line gives a breakdown of the number of garbage collections at various levels (for an explanation see the 'R Internals' manual).

Value

gc returns a matrix with rows "Ncells" (*cons cells*), usually 28 bytes each on 32-bit systems and 56 bytes on 64-bit systems, and "Vcells" (*vector cells*, 8 bytes each), and columns "used" and "gc trigger", each also interpreted in megabytes (rounded up to the next 0.1Mb).

If maxima have been set for either "Ncells" or "Vcells", a fifth column is printed giving the current limits in Mb (with NA denoting no limit).

The final two columns show the maximum space used since the last call to gc(reset = TRUE) (or since R started).

gcinfo returns the previous value of the flag.

See Also

The 'R Internals' manual.

Memory on R's memory management, and gctorture if you are an R developer.

gc.time() reports *time* used for garbage collection.

reg.finalizer for actions to happen at garbage collection.

Examples

```
gc() #- do it now
gcinfo(TRUE) #-- in the future, show when R does it
##            vvvvv use larger to *show* something
x <- integer(100000); for(i in 1:18) x <- c(x, i)
gcinfo(verbose = FALSE) #-- don't show it anymore

gc(TRUE)

gc(reset = TRUE)
```

---

gc.time *Report Time Spent in Garbage Collection*

Description

This function reports the time spent in garbage collection so far in the R session while GC timing was enabled.

Usage

```
gc.time(on = TRUE)
```

Arguments

on	logical; if TRUE, GC timing is enabled.

Details

Due to timer resolution this may be under-estimate.

This is a primitive.

Value

A numerical vector of length 5 giving the user CPU time, the system CPU time, the elapsed time and children's user and system CPU times (normally both zero), of time spent doing garbage collection whilst GC timing was enabled.

Times of child processes are not available on Windows and will always be given as NA.

See Also

gc, proc.time for the timings for the session.

Examples

```
gc.time()
```

---

gctorture *Torture Garbage Collector*

Description

Provokes garbage collection on (nearly) every memory allocation. Intended to ferret out memory protection bugs. Also makes R run *very* slowly, unfortunately.

Usage

```
gctorture(on = TRUE)
gctorture2(step, wait = step, inhibit_release = FALSE)
```

Arguments

on	logical; turning it on/off.
step	integer; run GC every step allocations; step = 0 turns the GC torture off.
wait	integer; number of allocations to wait before starting GC torture.
inhibit_release	logical; do not release free objects for re-use: use with caution.

Details

Calling `gctorture(TRUE)` instructs the memory manager to force a full GC on every allocation. `gctorture2` provides a more refined interface that allows the start of the GC torture to be deferred and also gives the option of running a GC only every `step` allocations.

The third argument to `gctorture2` is only used if R has been configured with a strict write barrier enabled. When this is the case all garbage collections are full collections, and the memory manager marks free nodes and enables checks in many situations that signal an error when a free node is used. This can help greatly in isolating unprotected values in C code. It does not detect the case where a node becomes free and is reallocated. The `inhibit_release` argument can be used to prevent such reallocation. This will cause memory to grow and should be used with caution and in conjunction with operating system facilities to monitor and limit process memory use.

`gctorture2` can also be invoked via environment variables at the start of the R session. `R_GCTORTURE` corresponds to the `step` argument, `R_GCTORTURE_WAIT` to `wait`, and `R_GCTORTURE_INHIBIT_RELEASE` to `inhibit_release`.

Value

Previous value of first argument.

Author(s)

Peter Dalgaard and Luke Tierney

`get` *Return the Value of a Named Object*

Description

Search by name for an object (`get`) or zero or more objects (`mget`).

Usage

```
get(x, pos = -1, envir = as.environment(pos), mode = "any",
    inherits = TRUE)

mget(x, envir = as.environment(-1), mode = "any", ifnotfound,
     inherits = FALSE)

dynGet(x, ifnotfound = , minframe = 1L, inherits = FALSE)
```

Arguments

`x`	For `get`, an object name (given as a character string or a symbol). For `mget`, a character vector of object names.
`pos`, `envir`	where to look for the object (see 'Details'); if omitted search as if the name of the object appeared unquoted in an expression.

mode	the mode or type of object sought: see the 'Details' section.
inherits	should the enclosing frames of the environment be searched?
ifnotfound	For mget, a list of values to be used if the item is not found: it will be coerced to a list if necessary. For dynGet any R object, e.g., a call to stop().
minframe	integer specifying the minimal frame number to look into.

Details

The pos argument can specify the environment in which to look for the object in any of several ways: as a positive integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The default of -1 indicates the current environment of the call to get. The envir argument is an alternative way to specify an environment.

These functions look to see if each of the name(s) x have a value bound to it in the specified environment. If inherits is TRUE and a value is not found for x in the specified environment, the enclosing frames of the environment are searched until the name x is encountered. See environment and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

If mode is specified then only objects of that type are sought. mode here is a mixture of the meanings of typeof and mode: "function" covers primitive functions and operators, "numeric", "integer" and "double" all refer to any numeric type, "symbol" and "name" are equivalent *but* "language" must be used (and not "call" or "("). Currently, mode = "S4" and mode = "object" are equivalent.

For mget, the values of mode and ifnotfound can be either the same length as x or of length 1. The argument ifnotfound must be a list containing either the value to use if the requested item is not found or a function of one argument which will be called if the item is not found, with argument the name of the item being requested.

dynGet() is somewhat experimental and to be used *inside* another function. It looks for an object in the callers, i.e., the sys.frame()s of the function. Use with caution.

Value

For get, the object found. If no object is found an error results.

For mget, a named list of objects (found or specified *via* ifnotfound).

Note

The reverse (or "inverse") of a <- get(nam) is assign(nam, a), assigning a to name nam.

inherits = TRUE is the default for get in R but not for S where it had a different meaning.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

exists for checking whether an object exists; get0 for an efficient way of both checking existence and getting an object.

assign, the inverse of get(), see above.

Use getAnywhere for searching for an object anywhere, including in other namespaces, and getFromNamespace to find an object in a specific namespace.

Examples

```
get("%o%")

## test mget
e1 <- new.env()
mget(letters, e1, ifnotfound = as.list(LETTERS))
```

---

getDLLRegisteredRoutines

Reflectance Information for C/Fortran routines in a DLL

Description

This function allows us to query the set of routines in a DLL that are registered with R to enhance dynamic lookup, error handling when calling native routines, and potentially security in the future. This function provides a description of each of the registered routines in the DLL for the different interfaces, i.e. .C, .Call, .Fortran and .External.

Usage

```
getDLLRegisteredRoutines(dll, addNames = TRUE)
```

Arguments

dll	a character string or DLLInfo object. The character string specifies the file name of the DLL of interest, and is given without the file name extension (e.g., the '.dll' or '.so') and with no directory/path information. So a file 'MyPackage/libs/MyPackage.so' would be specified as 'MyPackage'. The DLLInfo objects can be obtained directly in calls to dyn.load and library.dynam, or can be found after the DLL has been loaded using getLoadedDLLs, which returns a list of DLLInfo objects (index-able by DLL file name). The DLLInfo approach avoids any ambiguities related to two DLLs having the same name but corresponding to files in different directories.
addNames	a logical value. If this is TRUE, the elements of the returned lists are named using the names of the routines (as seen by R via registration or raw name). If FALSE, these names are not computed and assigned to the lists. As a result, the call should be quicker. The name information is also available in the NativeSymbolInfo objects in the lists.

Details

This takes the registration information after it has been registered and processed by the R internals. In other words, it uses the extended information.

There is a `print` method for the class, which prints only the types which have registered routines.

Value

A list of class `"DLLRegisteredRoutines"` with four elements corresponding to the routines registered for the `.C`, `.Call`, `.Fortran` and `.External` interfaces. Each is a list (of class `"NativeRoutineList"`) with as many elements as there were routines registered for that interface.

Each element identifies a routine and is an object of class `"NativeSymbolInfo"`. An object of this class has the following fields:

`name`	the registered name of the routine (not necessarily the name in the C code).
`address`	the memory address of the routine as resolved in the loaded DLL. This may be `NULL` if the symbol has not yet been resolved.
`dll`	an object of class `DLLInfo` describing the DLL. This is same for all elements returned.
`numParameters`	the number of arguments the native routine is to be called with.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

'Writing R Extensions' manual for symbol registration.

Duncan Temple Lang (2001). "In Search of C/C++ & FORTRAN Routines". *R News*, **1**(3), 20–23. https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

See Also

`getLoadedDLLs`, `getNativeSymbolInfo` for information on the entry points listed.

Examples

```
dlls <- getLoadedDLLs()
getDLLRegisteredRoutines(dlls[["base"]])

getDLLRegisteredRoutines("stats")
```

getLoadedDLLs *Get DLLs Loaded in Current Session*

Description

This function provides a way to get a list of all the DLLs (see dyn.load) that are currently loaded in the R session.

Usage

```
getLoadedDLLs()
```

Details

This queries the internal table that manages the DLLs.

Value

An object of class "DLLInfoList" which is a list with an element corresponding to each DLL that is currently loaded in the session. Each element is an object of class "DLLInfo" which has the following entries.

name	the abbreviated name.
path	the fully qualified name of the loaded DLL.
dynamicLookup	a logical value indicating whether R uses only the registration information to resolve symbols or whether it searches the entire symbol table of the DLL.
handle	a reference to the C-level data structure that provides access to the contents of the DLL. This is an object of class "DLLHandle".

Note that the class DLLInfo has a method for $ which can be used to resolve native symbols within that DLL. Therefore, one must access the R-level elements described above using [[, e.g. x[["name"]] or x[["handle"]].

Note

We are starting to use the handle elements in the DLL object to resolve symbols more directly in R.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>.

See Also

getDLLRegisteredRoutines, getNativeSymbolInfo

Examples

```
getLoadedDLLs()

utils::tail(getLoadedDLLs(), 2) # the last 2 loaded ones, still a DLLInfoList
```

`getNativeSymbolInfo` *Obtain a Description of one or more Native (C/Fortran) Symbols*

Description

This finds and returns a description of one or more dynamically loaded or 'exported' built-in native symbols. For each name, it returns information about the name of the symbol, the library in which it is located and, if available, the number of arguments it expects and by which interface it should be called (i.e `.Call`, `.C`, `.Fortran`, or `.External`). Additionally, it returns the address of the symbol and this can be passed to other C routines. Specifically, this provides a way to explicitly share symbols between different dynamically loaded package libraries. Also, it provides a way to query where symbols were resolved, and aids diagnosing strange behavior associated with dynamic resolution.

Usage

```
getNativeSymbolInfo(name, PACKAGE, unlist = TRUE,
                    withRegistrationInfo = FALSE)
```

Arguments

`name`	the name(s) of the native symbol(s).
`PACKAGE`	an optional argument that specifies to which DLL to restrict the search for this symbol. If this is `"base"`, we search in the R executable itself.
`unlist`	a logical value which controls how the result is returned if the function is called with the name of a single symbol. If `unlist` is `TRUE` and the number of symbol names in `name` is one, then the `NativeSymbolInfo` object is returned. If it is `FALSE`, then a list of `NativeSymbolInfo` objects is returned. This is ignored if the number of symbols passed in `name` is more than one. To be compatible with earlier versions of this function, this defaults to `TRUE`.
`withRegistrationInfo`	a logical value indicating whether, if `TRUE`, to return information that was registered with R about the symbol and its parameter types if such information is available, or if `FALSE` to return just the address of the symbol.

Details

This uses the same mechanism for resolving symbols as is used in all the native interfaces (`.Call`, etc.). If the symbol has been explicitly registered by the DLL in which it is contained, information about the number of arguments and the interface by which it should be called will be returned. Otherwise, a generic native symbol object is returned.

Value

Generally, a list of NativeSymbolInfo elements whose elements can be indexed by the elements of name in the call. Each NativeSymbolInfo object is a list containing the following elements:

name — the name of the symbol, as given by the name argument.

address — if withRegistrationInfo is FALSE, this is the native memory address of the symbol which can be used to invoke the routine, and also to compare with other symbol addresses. This is an external pointer object and of class NativeSymbol. If withRegistrationInfo is TRUE and registration information is available for the symbol, then this is an object of class RegisteredNativeSymbol and is a reference to an internal data type that has access to the routine pointer and registration information. This too can be used in calls to .Call, .C, .Fortran and .External.

dll — a list containing 3 elements:

- **name** the short form of the library name which can be used as the value of the PACKAGE argument in the different native interface functions.
- **path** the fully qualified name of the DLL.
- **dynamicLookup** a logical value indicating whether dynamic resolution is used when looking for symbols in this library, or only registered routines can be located.

If the routine was explicitly registered by the dynamically loaded library, the list contains a fourth field

numParameters — the number of arguments that should be passed in a call to this routine.

Additionally, the list will have an additional class, being CRoutine, CallRoutine, FortranRoutine or ExternalRoutine corresponding to the R interface by which it should be invoked.

If any of the symbols is not found, an error is raised.

If name contains only one symbol name and unlist is TRUE, then the single NativeSymbolInfo is returned rather than the list containing that one element.

Note

The third element of the NativeSymbolInfo objects was renamed from package to dll in R version 3.6.0, for consistency with the names of the NativeSymbolInfo objects returned by getDLLRegisteredRoutines().

Note

One motivation for accessing this reflectance information is to be able to pass native routines to C routines as function pointers in C. This allows us to treat native routines and R functions in a similar manner, such as when passing an R function to C code that makes callbacks to that function at different points in its computation (e.g., nls). Additionally, we can resolve the symbol just once and avoid resolving it repeatedly or using the internal cache.

Author(s)

Duncan Temple Lang

References

For information about registering native routines, see "In Search of C/C++ & FORTRAN Routines", R-News, volume 1, number 3, 2001, p20–23 (https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf).

See Also

getDLLRegisteredRoutines, is.loaded, .C, .Fortran, .External, .Call, dyn.load.

---

gettext *Translate Text Messages*

Description

Translation of text messages typically from calls to stop(), warning(), or message() happens when Native Language Support (NLS) was enabled in this build of R as it is almost always, see also the bindtextdomain() example.

The functions documented here are the low level building blocks used explicitly or implicitly in almost all such message producing calls and they attempt to translate character vectors or set where the translations are to be found.

Usage

```
gettext(..., domain = NULL, trim = TRUE)

ngettext(n, msg1, msg2, domain = NULL)

bindtextdomain(domain, dirname = NULL)

Sys.setLanguage(lang, unset = "en")
```

Arguments

...	one or more character vectors.
trim	logical indicating if the white space trimming in gettext() should happen. trim = FALSE may be needed for compiled code (C / C++) messages which often end with \n.
domain	the 'domain' for the translation, a character string, or NULL; see 'Details'.
n	a non-negative integer.
msg1	the message to be used in English for n = 1.
msg2	the message to be used in English for n = 0, 2, 3, ...

dirname	the directory in which to find translated message catalogs for the domain.
lang	a character string specifying a language for which translations should be sought.
unset	a string, specifying the default language assumed to be current in the case Sys.getenv("LANGUAGE") is unset or empty.

Details

If domain is NULL (the default) in gettext or ngettext, the domain is inferred. If gettext or ngettext is called from a function in the namespace of package **pkg** including called via stop(), warning(), or message() from the function, or, say, evaluated as if called from that namespace, see the evalq() example, the domain is set to "R-pkg". Otherwise there is no default domain and messages are not translated.

Setting domain = NA in gettext or ngettext suppresses any translation.

"" does not match any domain. In gettext or ngettext, domain = "" is effectively the same as domain = NA.

If the domain is found, each character string is offered for translation, and replaced by its translation into the current language if one is found.

The *language* to be used for message translation is determined by your OS default and/or the locale setting at R's startup, see Sys.getlocale(), and notably the LANGUAGE environment variable, and also Sys.setLanguage() here.

Conventionally the domain for R warning/error messages in package **pkg** is "R-pkg", and that for C-level messages is "pkg".

For gettext, when trim is true as by default, leading and trailing whitespace is ignored ("trimmed") when looking for the translation.

ngettext is used where the message needs to vary by a single integer. Translating such messages is subject to very specific rules for different languages: see the GNU Gettext Manual. The string will often contain a single instance of %d to be used in sprintf. If English is used, msg1 is returned if n == 1 and msg2 in all other cases.

bindtextdomain is typically wrapper for the C function of the same name: your system may have a man page for it. With a non-NULL dirname it specifies where to look for message catalogues: with dirname = NULL it returns the current location. If NLS is not enabled, bindtextdomain(*,*) returns NULL. The special case bindtextdomain(NULL) calls C level textdomain(textdomain(NULL)) for the purpose of flushing (i.e., emptying) the cache of already translated strings; it returns TRUE when NLS is enabled.

The utility Sys.setLanguage(lang) combines setting the LANGUAGE environment variable with flushing the translation cache by bindtextdomain(NULL).

Value

For gettext, a character vector, one element per string in .... If translation is not enabled or no domain is found or no translation is found in that domain, the original strings are returned.

For ngettext, a character string.

For bindtextdomain, a character string giving the current base directory, or NULL if setting it failed.

For `Sys.setLanguage()`, the previous `LANGUAGE` setting with attribute `attr(*, "ok")`, a `logical` indicating success. Note that currently, using a non-existing language `lang` is still set and no translation will happen, without any `message`.

See Also

`stop` and `warning` make use of `gettext` to translate messages.

`xgettext` (package **tools**) for extracting translatable strings from R source files.

Examples

```
bindtextdomain("R")  # non-null if and only if NLS is enabled

for(n in 0:3)
    print(sprintf(ngettext(n, "%d variable has missing values",
                              "%d variables have missing values"),
                  n))

## Not run: ## for translation, those strings should appear in R-pkg.pot as
msgid        "%d variable has missing values"
msgid_plural "%d variables have missing values"
msgstr[0] ""
msgstr[1] ""

## End(Not run)

miss <- "One only" # this line, or the next for the ngettext() below
miss <- c("one", "or", "another")
cat(ngettext(length(miss), "variable", "variables"),
    paste(sQuote(miss), collapse = ", "),
    ngettext(length(miss), "contains", "contain"), "missing values\n")

## better for translators would be to use
cat(sprintf(ngettext(length(miss),
                     "variable %s contains missing values\n",
                     "variables %s contain missing values\n"),
            paste(sQuote(miss), collapse = ", ")))

thisLang <- Sys.getenv("LANGUAGE", unset = NA) # so we can reset it
if(is.na(thisLang) || !nzchar(thisLang)) thisLang <- "en" # "factory" default
enT <- "empty model supplied"
Sys.setenv(LANGUAGE = "de") # may not always 'work'
gettext(enT, domain="R-stats")# "leeres Modell angegeben" (if translation works)
tget <- function() gettext(enT)
tget() # not translated as fn tget() is not from "stats" pkg/namespace
evalq(function() gettext(enT), asNamespace("stats"))() # *is* translated

## Sys.setLanguage()  -- typical usage --
Sys.setLanguage("en") -> oldSet # does set LANGUAGE env.var
errMsg <- function(expr) tryCatch(expr, error=conditionMessage)
(errMsg(1 + "2") -> err)
Sys.setLanguage("fr")
```

```
errMsg(1 + "2")
Sys.setLanguage("de")
errMsg(1 + "2")
## Usually, you would reset the language to "previous" via
Sys.setLanguage(oldSet)

## A show off of translations -- platform (font etc) dependent:
## The translation languages available for "base" R in this version of R:
if(capabilities("NLS")) withAutoprint({
  langs <- list.files(bindtextdomain("R"),
        pattern = "^[a-z]{2}(_[A-Z]{2}|@quot)?$")
  langs
  txts <- sapply(setNames(,langs),
   function(lang) { Sys.setLanguage(lang)
     gettext("incompatible dimensions", domain="R-stats") })
  cbind(txts)
  (nTrans <- length(unique(txts)))
  (not_translated <- names(txts[txts == txts[["en"]]]))
})

## Here, we reset to the *original* setting before the full example started:
if(nzchar(thisLang)) { ## reset to previous and check
  Sys.setLanguage(thisLang)
  stopifnot(identical(errMsg(1 + "2"), err))
} # else staying at 'de' ..
```

`getwd` *Get or Set Working Directory*

Description

`getwd` returns an absolute filepath representing the current working directory of the R process; `setwd(dir)` is used to set the working directory to `dir`.

Usage

```
getwd()
setwd(dir)
```

Arguments

`dir`	A character string: tilde expansion will be done.

Details

See files for how file paths with marked encodings are interpreted.

Value

getwd returns a character string or NULL if the working directory is not available. On Windows the path returned will use / as the path separator and be encoded in UTF-8. The path will not have a trailing / unless it is the root directory (of a drive or share on Windows).

setwd returns the current directory before the change, invisibly and with the same conventions as getwd. It will give an error if it does not succeed (including if it is not implemented).

Note

Note that the return value is said to be **an** absolute filepath: there can be more than one representation of the path to a directory and on some OSes the value returned can differ after changing directories and changing back to the same directory (for example if symbolic links have been traversed).

See Also

list.files for the *contents* of a directory.

normalizePath for a 'canonical' path name.

Examples

```
(WD <- getwd())
if (!is.null(WD)) setwd(WD)
```

---

gl *Generate Factor Levels*

Description

Generate factors by specifying the pattern of their levels.

Usage

```
gl(n, k, length = n*k, labels = seq_len(n), ordered = FALSE)
```

Arguments

n	an integer giving the number of levels.
k	an integer giving the number of replications.
length	an integer giving the length of the result.
labels	an optional vector of labels for the resulting factor levels.
ordered	a logical indicating whether the result should be ordered or not.

Value

The result has levels from 1 to n with each value replicated in groups of length k out to a total length of length.

gl is modelled on the *GLIM* function of the same name.

See Also

The underlying factor().

Examples

```
## First control, then treatment:
gl(2, 8, labels = c("Control", "Treat"))
## 20 alternating 1s and 2s
gl(2, 1, 20)
## alternating pairs of 1s and 2s
gl(2, 2, 20)
```

---

grep *Pattern Matching and Replacement*

Description

grep, grepl, regexpr, gregexpr, regexec and gregexec search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

sub and gsub perform replacement of the first and all matches respectively.

Usage

```
grep(pattern, x, ignore.case = FALSE, perl = FALSE, value = FALSE,
     fixed = FALSE, useBytes = FALSE, invert = FALSE)

grepl(pattern, x, ignore.case = FALSE, perl = FALSE,
      fixed = FALSE, useBytes = FALSE)

sub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE,
    fixed = FALSE, useBytes = FALSE)

gsub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE,
     fixed = FALSE, useBytes = FALSE)

regexpr(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)

gregexpr(pattern, text, ignore.case = FALSE, perl = FALSE,
         fixed = FALSE, useBytes = FALSE)
```

```
regexec(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)

gregexec(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)
```

Arguments

pattern	character string containing a regular expression (or character string for fixed = TRUE) to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed except for regexpr, gregexpr and regexec.
x, text	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
ignore.case	if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
perl	logical. Should Perl-compatible regexps be used?
value	if FALSE, a vector containing the (integer) indices of the matches determined by grep is returned, and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See 'Details'.
invert	logical. If TRUE return indices or values for elements that do *not* match.
replacement	a replacement for matched pattern in sub and gsub. Coerced to character if possible. For fixed = FALSE this can include backreferences "\1" to "\9" to parenthesized subexpressions of pattern. For perl = TRUE only, it can also contain "\U" or "\L" to convert the rest of the replacement to upper or lower case and "\E" to end case conversion. If a character vector of length 2 or more is supplied, the first element is used with a warning. If NA, all elements in the result corresponding to matches will be set to NA.

Details

Arguments which should be character strings or character vectors are coerced to character if possible.

Each of these functions operates in one of three modes:

1. fixed = TRUE: use exact matching.
2. perl = TRUE: use Perl-style regular expressions.
3. fixed = FALSE, perl = FALSE: use POSIX 1003.2 extended regular expressions (the default).

See the help pages on regular expression for details of the different types of regular expressions.

The two *sub functions differ only in that sub replaces only the first occurrence of a pattern whereas gsub replaces all occurrences. If replacement contains backreferences which are not defined in pattern the result is undefined (but most often the backreference is taken to be "").

For regexpr, gregexpr, regexec and gregexec it is an error for pattern to be NA, otherwise NA is permitted and gives an NA match.

Both grep and grepl take missing values in x as not matching a non-missing pattern.

The main effect of useBytes = TRUE is to avoid errors/warnings about invalid inputs and spurious matches in multibyte locales, but for regexpr it changes the interpretation of the output. It inhibits the conversion of inputs with marked encodings, and is forced if any input is found which is marked as "bytes" (see Encoding).

Caseless matching does not make much sense for bytes in a multibyte locale, and you should expect it only to work for ASCII characters if useBytes = TRUE.

regexpr and gregexpr with perl = TRUE allow Python-style named captures, but not for *long vector* inputs.

Invalid inputs in the current locale are warned about up to 5 times.

Caseless matching with perl = TRUE for non-ASCII characters depends on the PCRE library being compiled with 'Unicode property support', which PCRE2 is by default.

Value

grep(value = FALSE) returns a vector of the indices of the elements of x that yielded a match (or not, for invert = TRUE). This will be an integer vector unless the input is a *long vector*, when it will be a double vector.

grep(value = TRUE) returns a character vector containing the selected elements of x (after coercion, preserving names but no other attributes).

grepl returns a logical vector (match or not for each element of x).

sub and gsub return a character vector of the same length and with the same attributes as x (after possible coercion to character). Elements of character vectors x which are not substituted will be returned unchanged (including any declared encoding if useBytes = FALSE). If useBytes = FALSE a non-ASCII substituted result will often be in UTF-8 with a marked encoding (e.g., if there is a UTF-8 input, and in a multibyte locale unless fixed = TRUE). Such strings can be re-encoded by enc2native. If any of the inputs is marked as "bytes", elements of character vectors x which are substituted will be returned marked as "bytes", but the encoding flag on elements not substituted is unspecified (it may be the original or "bytes"). If none of the inputs is marked as "bytes", but useBytes = TRUE is given explicitly, the encoding flag is unspecified even on the substituted elements (it may be "bytes" or "unknown", possibly invalid in the current encoding). Mixed use of "bytes" and other marked encodings is discouraged, but if still desired one may use iconv to re-encode the result e.g. to UTF-8 with suitably substituted invalid bytes.

regexpr returns an integer vector of the same length as text giving the starting position of the first match or -1 if there is none, with attribute "match.length", an integer vector giving the length of the matched text (or -1 for no match). The match positions and lengths are in characters unless useBytes = TRUE is used, when they are in bytes (as they are for ASCII-only matching: in either case an attribute useBytes with value TRUE is set on the result). If named capture is used there are further attributes "capture.start", "capture.length" and "capture.names".

gregexpr returns a list of the same length as text each element of which is of the same form as the return value for regexpr, except that the starting positions of every (disjoint) match are given.

regexec returns a list of the same length as text each element of which is either -1 if there is no match, or a sequence of integers with the starting positions of the match and all substrings corresponding to parenthesized subexpressions of pattern, with attribute "match.length" a vector giving the lengths of the matches (or -1 for no match). The interpretation of positions and length and the attributes follows regexpr.

gregexec returns the same as regexec, except that to accommodate multiple matches per element of text, the integer sequences for each match are made into columns of a matrix, with one matrix per element of text with matches.

Where matching failed because of resource limits (especially for perl = TRUE) this is regarded as a non-match, usually with a warning.

Warning

The POSIX 1003.2 mode of gsub and gregexpr does not work correctly with repeated word-boundaries (e.g., pattern = "\b"). Use perl = TRUE for such matches (but that may not work as expected with non-ASCII inputs, as the meaning of 'word' is system-dependent).

Performance considerations

If you are doing a lot of regular expression matching, including on very long strings, you will want to consider the options used. Generally perl = TRUE will be faster than the default regular expression engine, and fixed = TRUE faster still (especially when each pattern is matched only a few times).

If you are working with texts with non-ASCII characters, which can be easily turned into ASCII (e.g. by substituting fancy quotes), doing so is likely to improve performance.

If you are working in a single-byte locale (though not common since R 4.2) and have marked UTF-8 strings that are representable in that locale, convert them first as just one UTF-8 string will force all the matching to be done in Unicode, which attracts a penalty of around $3\times$ for the default POSIX 1003.2 mode.

While useBytes = TRUE will improve performance further, because the strings will not be checked before matching and the actual matching will be faster, it can produce unexpected results so is best avoided. With fixed = TRUE and useBytes = FALSE, optimizations are in place that take advantage of byte-based matching working for such patterns in UTF-8. With useBytes = TRUE, character ranges, wildcards, and other regular expression patterns may produce unexpected results.

PCRE-based matching by default used to put additional effort into 'studying' the compiled pattern when x/text has length 10 or more. That study may use the PCRE JIT compiler on platforms where it is available (see pcre_config). As from PCRE2 (PCRE version >= 10.00 as reported by extSoftVersion), there is no study phase, but the patterns are optimized automatically when possible, and PCRE JIT is used when enabled. The details are controlled by options PCRE_study and PCRE_use_JIT. (Some timing comparisons can be seen by running file 'tests/PCRE.R' in the R sources (and perhaps installed).) People working with PCRE and very long strings can adjust the maximum size of the JIT stack by setting environment variable R_PCRE_JIT_STACK_MAXSIZE before JIT is used to a value between 1 and 1000 in MB: the default is 64. When JIT is not used with PCRE version < 10.30 (that is with PCRE1 and old versions of PCRE2), it might also be wise to set the option PCRE_limit_recursion.

Note

Aspects will be platform-dependent as well as locale-dependent: for example the implementation of character classes (except `[:digit:]` and `[:xdigit:]`). One can expect results to be consistent for ASCII inputs and when working in UTF-8 mode (when most platforms will use Unicode character tables, although those are updated frequently and subject to some degree of interpretation – is a circled capital letter alphabetic or a symbol?). However, results in 8-bit encodings can differ considerably between platforms, modes and from the UTF-8 versions.

Source

The C code for POSIX-style regular expression matching has changed over the years. As from R 2.10.0 (Oct 2009) the TRE library of Ville Laurikari (https://github.com/laurikari/tre) is used. The POSIX standard does give some room for interpretation, especially in the handling of invalid regular expressions and the collation of character ranges, so the results will have changed slightly over the years.

For Perl-style matching PCRE2 or PCRE (https://www.pcre.org) is used: again the results may depend (slightly) on the version of PCRE in use.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (grep)

See Also

regular expression (aka `regexp`) for the details of the pattern specification.

`regmatches` for extracting matched substrings based on the results of `regexpr`, `gregexpr` and `regexec`.

`glob2rx` to turn wildcard matches into regular expressions.

`agrep` for approximate matching.

`charmatch`, `pmatch` for partial matching, `match` for matching to whole strings, `startsWith` for matching of initial parts of strings.

`tolower`, `toupper` and `chartr` for character translations.

`apropos` uses regexps and has more examples.

`grepRaw` for matching raw vectors.

Options `PCRE_limit_recursion`, `PCRE_study` and `PCRE_use_JIT`.

`extSoftVersion` for the versions of regex and PCRE libraries in use, `pcre_config` for more details for PCRE.

Examples

```
grep("[a-z]", letters)

txt <- c("arm","foot","lefroo", "bafoobar")
if(length(i <- grep("foo", txt)))
   cat("'foo' appears at least once in\n\t", txt, "\n")
```

```
i # 2 and 4
txt[i]

## Double all 'a' or 'b's;  "\" must be escaped, i.e., 'doubled'
gsub("([ab])", "\\1_\\1_", "abc and ABC")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
  "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
  "is", "intended", "to", "guarantee", "your", "freedom", "to",
  "share", "and", "change", "free", "software", "--",
  "to", "make", "sure", "the", "software", "is",
  "free", "for", "all", "its", "users")
( i <- grep("[gu]", txt) ) # indices
stopifnot( txt[i] == grep("[gu]", txt, value = TRUE) )

## Note that for some implementations character ranges are
## locale-dependent (but not currently).  Then [b-e] in locales such as
## en_US may include B as the collation order is aAbBcCdDe ...
(ot <- sub("[b-e]",".", txt))
txt[ot != gsub("[b-e]",".", txt)]#- gsub does "global" substitution
## In caseless matching, ranges include both cases:
a <- grep("[b-e]", txt, value = TRUE)
b <- grep("[b-e]", txt, ignore.case = TRUE, value = TRUE)
setdiff(b, a)

txt[gsub("g","#", txt) !=
    gsub("g","#", txt, ignore.case = TRUE)] # the "G" words

regexpr("en", txt)

gregexpr("e", txt)

## Using grepl() for filtering
## Find functions with argument names matching "warn":
findArgs <- function(env, pattern) {
  nms <- ls(envir = as.environment(env))
  nms <- nms[is.na(match(nms, c("F","T")))] # <-- work around "checking hack"
  aa <- sapply(nms, function(.) { o <- get(.)
               if(is.function(o)) names(formals(o)) })
  iw <- sapply(aa, function(a) any(grepl(pattern, a, ignore.case=TRUE)))
  aa[iw]
}
findArgs("package:base", "warn")

## trim trailing white space
str <- "Now is the time      "
sub(" +$", "", str)  ## spaces only
## what is considered 'white space' depends on the locale.
sub("[[:space:]]+$", "", str) ## white space, POSIX-style
## what PCRE considered white space changed in version 8.34: see ?regex
sub("\\s+$", "", str, perl = TRUE) ## PCRE-style white space
```

```
## capitalizing
txt <- "a test of capitalizing"
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", txt, perl=TRUE)
gsub("\\b(\\w)",    "\\U\\1",       txt, perl=TRUE)

txt2 <- "useRs may fly into JFK or laGuardia"
gsub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", txt2, perl=TRUE)
 sub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", txt2, perl=TRUE)

## named capture
notables <- c("  Ben Franklin and Jefferson Davis",
              "\tMillard Fillmore")
# name groups 'first' and 'last'
name.rex <- "(?<first>[[:upper:]][[:lower:]]+) (?<last>[[:upper:]][[:lower:]]+)"
(parsed <- regexpr(name.rex, notables, perl = TRUE))
gregexpr(name.rex, notables, perl = TRUE)[[2]]
parse.one <- function(res, result) {
  m <- do.call(rbind, lapply(seq_along(res), function(i) {
    if(result[i] == -1) return("")
    st <- attr(result, "capture.start")[i, ]
    substring(res[i], st, st + attr(result, "capture.length")[i, ] - 1)
  }))
  colnames(m) <- attr(result, "capture.names")
  m
}
parse.one(notables, parsed)

## Decompose a URL into its components.
## Example by LT (http://www.cs.uiowa.edu/~luke/R/regexp.html).
x <- "http://stat.umn.edu:80/xyz"
m <- regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)", x)
m
regmatches(x, m)
## Element 3 is the protocol, 4 is the host, 6 is the port, and 7
## is the path.  We can use this to make a function for extracting the
## parts of a URL:
URL_parts <- function(x) {
    m <- regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)", x)
    parts <- do.call(rbind,
                     lapply(regmatches(x, m), `[`, c(3L, 4L, 6L, 7L)))
    colnames(parts) <- c("protocol","host","port","path")
    parts
}
URL_parts(x)

## gregexec() may match multiple times within a single string.
pattern <- "([[:alpha:]]+)([[:digit:]]+)"
s <- "Test: A1 BC23 DEF456"
m <- gregexec(pattern, s)
m
regmatches(s, m)
```

```
## Before gregexec() was implemented, one could emulate it by running
## regexec() on the regmatches obtained via gregexpr().  E.g.:
lapply(regmatches(s, gregexpr(pattern, s)),
       function(e) regmatches(e, regexec(pattern, e)))
```

---

grepRaw *Pattern Matching for Raw Vectors*

Description

grepRaw searches for substring pattern matches within a raw vector x.

Usage

```
grepRaw(pattern, x, offset = 1L, ignore.case = FALSE,
        value = FALSE, fixed = FALSE, all = FALSE, invert = FALSE)
```

Arguments

pattern	raw vector containing a regular expression (or fixed pattern for fixed = TRUE) to be matched in the given raw vector. Coerced by charToRaw to a character string if possible.
x	a raw vector where matches are sought, or an object which can be coerced by charToRaw to a raw vector. Long vectors are not supported.
ignore.case	if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
offset	an integer specifying the offset from which the search should start. Must be positive. The beginning of line is defined to be at that offset so "^" will match there.
value	logical. Determines the return value: see 'Value'.
fixed	logical. If TRUE, pattern is a pattern to be matched as is.
all	logical. If TRUE all matches are returned, otherwise just the first one.
invert	logical. If TRUE return indices or values for elements that do *not* match. Ignored (with a warning) unless value = TRUE.

Details

Unlike grep, seeks matching patterns within the raw vector x . This has implications especially in the all = TRUE case, e.g., patterns matching empty strings are inherently infinite and thus may lead to unexpected results.

The argument invert is interpreted as asking to return the complement of the match, which is only meaningful for value = TRUE. Argument offset determines the start of the search, not of the complement. Note that invert = TRUE with all = TRUE will split x into pieces delimited by the pattern including leading and trailing empty strings (consequently the use of regular expressions with "^" or "$" in that case may lead to less intuitive results).

Some combinations of arguments such as fixed = TRUE with value = TRUE are supported but are less meaningful.

Value

grepRaw(value = FALSE) returns an integer vector of the offsets at which matches have occurred. If all = FALSE then it will be either of length zero (no match) or length one (first matching position).

grepRaw(value = TRUE, all = FALSE) returns a raw vector which is either empty (no match) or the matched part of x.

grepRaw(value = TRUE, all = TRUE) returns a (potentially empty) list of raw vectors corresponding to the matched parts.

Source

The TRE library of Ville Laurikari (https://github.com/laurikari/tre/) is used except for fixed = TRUE.

See Also

regular expression (aka regexp) for the details of the pattern specification.

grep for matching character vectors.

Examples

```
grepRaw("no match", "textText")  # integer(0): no match
grepRaw("adf", "adadfadfdfadadf") # 3 - the first match
grepRaw("adf", "adadfadfdfadadf", all=TRUE, fixed=TRUE)
## [1]  3  6 13 -- three matches
```

---

groupGeneric *S3 Group Generic Functions*

Description

Group generic methods can be defined for the following pre-specified groups of functions, Math, Ops, matrixOps, Summary and Complex. (There are no objects of these names in base R, but there are in the **methods** package, not yet for matrixOps.)

A method defined for an individual member of the group takes precedence over a method defined for the group as a whole.

Usage

```
## S3 methods for group generics have prototypes:
Math(x, ...)
Ops(e1, e2)
Complex(z)
Summary(..., na.rm = FALSE)
matrixOps(x, y)
```

Arguments

x, y, z, e1, e2	objects.
...	further arguments passed to methods.
na.rm	logical: should missing values be removed?

Details

There are five *groups* for which S3 methods can be written, namely the "Math", "Ops", "Summary", "matrixOps", and "Complex" groups. These are not R objects in base R, but methods can be supplied for them and base R contains factor, data.frame and difftime methods for the first three groups. (There is also a ordered method for Ops, POSIXt and Date methods for Math and Ops, package_version methods for Ops and Summary, as well as a ts method for Ops in package **stats**.)

1. Group "Math":
   - abs, sign, sqrt,
     floor, ceiling, trunc,
     round, signif
   - exp, log, expm1, log1p,
     cos, sin, tan,
     cospi, sinpi, tanpi,
     acos, asin, atan
     cosh, sinh, tanh,
     acosh, asinh, atanh
   - lgamma, gamma, digamma, trigamma
   - cumsum, cumprod, cummax, cummin

   Members of this group dispatch on x. Most members accept only one argument, but members log, round and signif accept one or two arguments, and trunc accepts one or more.
2. Group "Ops":
   - "+", "-", "*", "/", "^", "%%", "%/%"
   - "&", "|", "!"
   - "==", "!=", "<", "<=", ">=", ">"

   This group contains both binary and unary operators (+, - and !): when a unary operator is encountered the Ops method is called with one argument and e2 is missing.

   The classes of both arguments are considered in dispatching any member of this group. For each argument its vector of classes is examined to see if there is a matching specific (preferred) or Ops method. If a method is found for just one argument or the same method is found for both, it is used. If different methods are found, then the generic chooseOpsMethod() is called to pick the appropriate method. (See ?chooseOpsMethod for details). If chooseOpsMethod() does not resolve the method, then there is a warning about 'incompatible methods': in that case or if no method is found for either argument the internal method is used.

   Note that the data.frame methods for the comparison ("Compare": ==, <, ...) and logic ("Logic": & | and !) operators return a logical matrix instead of a data frame, for convenience and back compatibility.

   If the members of this group are called as functions, any argument names are removed to ensure that positional matching is always used.

3. Group "matrixOps":
   - "%*%"

   This group currently contains the matrix multiply %*% binary operator only, where at least crossprod() and tcrossprod() are meant to follow. Members of the group have the same dispatch semantics (using *both* arguments) as the Ops group.
4. Group "Summary":
   - all, any
   - sum, prod
   - min, max
   - range

   Members of this group dispatch on the first argument supplied.

   Note that the data.frame methods for the "Summary" and "Math" groups require "numeric-alike" columns x, i.e., fulfilling

   ```
   is.numeric(x) || is.logical(x) || is.complex(x)
   ```
5. Group "Complex":
   - Arg, Conj, Im, Mod, Re

   Members of this group dispatch on z.

Note that a method will be used for one of these groups or one of its members *only* if it corresponds to a "class" attribute, as the internal code dispatches on oldClass and not on class. This is for efficiency: having to dispatch on, say, Ops.integer would be too slow.

The number of arguments supplied for primitive members of the "Math" group generic methods is not checked prior to dispatch.

There is no lazy evaluation of arguments for group-generic functions.

Technical Details

These functions are all primitive and internal generic.

The details of method dispatch and variables such as .Generic are discussed in the help for UseMethod. There are a few small differences:

- For the operators of group Ops, the object .Method is a length-two character vector with elements the methods selected for the left and right arguments respectively. (If no method was selected, the corresponding element is "".)
- Object .Group records the group used for dispatch (if a specific method is used this is "").

Note

Package **methods** does contain objects with these names, which it has re-used in confusing similar (but different) ways. See the help for that package.

References

Appendix A, *Classes and Methods* of
Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

methods for methods of non-internal generic functions.

S4groupGeneric for group generics for S4 methods.

Examples

```
require(utils)

d.fr <- data.frame(x = 1:9, y = stats::rnorm(9))
class(1 + d.fr) == "data.frame" ##-- add to d.f. ...

methods("Math")
methods("Ops")
methods("Summary")
methods("Complex")  # none in base R
```

---

grouping *Grouping Permutation*

Description

grouping returns a permutation which rearranges its first argument such that identical values are adjacent to each other. Also returned as attributes are the group-wise partitioning and the maximum group size.

Usage

```
grouping(...)
```

Arguments

...	a sequence of numeric, character or logical vectors, all of the same length, or a classed R object.

Details

The function partially sorts the elements so that identical values are adjacent. NA values come last. This is guaranteed to be stable, so ties are preserved, and if the data are already grouped/sorted, the grouping is unchanged. This is useful for aggregation and is particularly fast for character vectors.

Under the covers, the "radix" method of order is used, and the same caveats apply, including restrictions on character encodings and lack of support for long vectors (those with 2^{31} or more elements). Real-valued numbers are slightly rounded to account for numerical imprecision.

Like order, for a classed R object the grouping is based on the result of xtfrm.

Value

An object of class "grouping", the representation of which should be considered experimental and subject to change. It is an integer vector with two attributes:

ends	subscripts in the result corresponding to the last member of each group
maxgrpn	the maximum group size

See Also

order, xtfrm.

Examples

```
(ii <- grouping(x <- c(1, 1, 3:1, 1:4, 3), y <- c(9, 9:1), z <- c(2, 1:9)))
## 6  5  2  1  7  4 10  8  3  9
rbind(x, y, z)[, ii]
```

---

gzcon *(De)compress I/O Through Connections*

Description

gzcon provides a modified connection that wraps an existing connection, and decompresses reads or compresses writes through that connection. Standard gzip headers are assumed.

Usage

```
gzcon(con, level = 6, allowNonCompressed = TRUE, text = FALSE)
```

Arguments

con	a connection.
level	integer between 0 and 9, the compression level when writing.
allowNonCompressed	logical. When reading, should non-compressed input be allowed?
text	logical. Should the connection be text-oriented? This is distinct from the mode of the connection (must always be binary). If TRUE, pushBack works on the connection, otherwise readBin and friends apply.

Details

If con is open then the modified connection is opened. Closing the wrapper connection will also close the underlying connection.

Reading from a connection which does not supply a gzip magic header is equivalent to reading from the original connection if allowNonCompressed is true, otherwise an error.

Compressed output will contain embedded NUL bytes, and so con is not permitted to be a textConnection opened with open = "w". Use a writable rawConnection to compress data into a variable.

The original connection becomes unusable: any object pointing to it will now refer to the modified connection. For this reason, the new connection needs to be closed explicitly.

Value

An object inheriting from class "connection". This is the same connection *number* as supplied, but with a modified internal structure. It has binary mode.

See Also

gzfile

Examples

```
## Uncompress a data file from a URL
z <- gzcon(url("https://www.stats.ox.ac.uk/pub/datasets/csb/ch12.dat.gz"))
# read.table can only read from a text-mode connection.
raw <- textConnection(readLines(z))
close(z)
dat <- read.table(raw)
close(raw)
dat[1:4, ]


## gzfile and gzcon can inter-work.
## Of course here one would use gzfile, but file() can be replaced by
## any other connection generator.
zzfil <- tempfile(fileext = ".gz")
zz <- gzfile(zzfil, "w")
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzcon(file(zzfil, "rb")))
close(zz)
unlink(zzfil)

zzfil2 <- tempfile(fileext = ".gz")
zz <- gzcon(file(zzfil2, "wb"))
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile(zzfil2))
close(zz)
unlink(zzfil2)
```

hexmode *Integer Numbers Displayed in Hexadecimal*

Description

Integers which are displayed in hexadecimal (short 'hex') format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Arithmetic works as for integers, and non-integer valued mathematical functions typically work by truncating the result to integer.

Usage

```
as.hexmode(x)

## S3 method for class 'hexmode'
as.character(x, keepStr = FALSE, ...)

## S3 method for class 'hexmode'
format(x, width = NULL, upper.case = FALSE, ...)

## S3 method for class 'hexmode'
print(x, ...)
```

Arguments

x	an object, for the methods inheriting from class "hexmode".
keepStr	a logical indicating that names and dimensions should be kept; set TRUE for back compatibility, if needed.
width	NULL or a positive integer specifying the minimum field width to be used, with padding by leading zeroes.
upper.case	a logical indicating whether to use upper-case letters or lower-case letters (default).
...	further arguments passed to or from other methods.

Details

Class "hexmode" consists of integer vectors with that class attribute, used primarily to ensure that they are printed in hex. Subsetting ([) works too, as do arithmetic or other mathematical operations, albeit truncated to integer.

as.character(x) drops all attributes (unless when keepStr=TRUE where it keeps, dim, dimnames and names for back compatibility) and converts each entry individually, hence with no leading zeroes, whereas in format(), when width = NULL (the default), the output is padded with leading zeroes to the smallest width needed for all the non-missing elements.

as.hexmode can convert integers (of type "integer" or "double") and character vectors whose elements contain only 0-9, a-f, A-F (or are NA) to class "hexmode".

There is a ! method and methods for | and &: these recycle their arguments to the length of the longer and then apply the operators bitwise to each element.

See Also

octmode, sprintf for other options in converting integers to hex, strtoi to convert hex strings to integers.

Examples

```
i <- as.hexmode("7fffffff")
i; class(i)
identical(as.integer(i), .Machine$integer.max)

hm <- as.hexmode(c(NA, 1)); hm
as.integer(hm)

Xm <- as.hexmode(1:16)
Xm # print()s via format()
stopifnot(nchar(format(Xm)) == 2)
Xm[-16] # *no* leading zeroes!
stopifnot(format(Xm[-16]) == c(1:9, letters[1:6]))

## Integer arithmetic (remaining "hexmode"):
16*Xm
Xm^2
-Xm
(fac <- factorial(Xm[1:12])) # !1, !2, !3, !4 .. in hexadecimals
as.integer(fac) # indeed the same as  factorial(1:12)
```

Hyperbolic *Hyperbolic Functions*

Description

These functions give the obvious hyperbolic functions. They respectively compute the hyperbolic cosine, sine, tangent, and their inverses, arc-cosine, arc-sine, arc-tangent (or '*area cosine*', etc).

Usage

```
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
```

Arguments

x	a numeric or complex vector

Details

These are internal generic primitive functions: methods can be defined for them individually or via the Math group generic.

Branch cuts are consistent with the inverse trigonometric functions asin *et seq*, and agree with those defined in Abramowitz & Stegun, figure 4.7, page 86. The behaviour actually on the cuts follows the C99 standard which requires continuity coming round the endpoint in a counter-clockwise direction.

S4 methods

All are S4 generic functions: methods can be defined for them individually or via the Math group generic.

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

See Also

The trigonometric functions, cos, sin, tan, and their inverses acos, asin, atan.

The logistic distribution function plogis is a shifted version of tanh() for numeric x.

---

iconv *Convert Character Vector between Encodings*

Description

This uses system facilities to convert a character vector between encodings: the 'i' stands for 'internationalization'.

Usage

```
iconv(x, from = "", to = "", sub = NA, mark = TRUE, toRaw = FALSE)

iconvlist()
```

Arguments

x	a character vector, or an object to be converted to a character vector by as.character, or a list with NULL and raw elements as returned by iconv(toRaw = TRUE).
from	a character string describing the current encoding.
to	a character string describing the target encoding.

sub	character string. If not NA it is used to replace any non-convertible bytes in the input. (This would normally be a single character, but can be more.) If "byte", the indication is "<xx>" with the hex code of the byte. If "Unicode" and converting from UTF-8, the Unicode point in the form "<U+xxxx>", or if c99, a C99-style escape "\uxxxx". (For points in a 'supplementary plane', "\Uxxxxxxxx" is used, with zero-padding)
mark	logical, for expert use. Should encodings be marked?
toRaw	logical. Should a list of raw vectors be returned rather than a character vector?

Details

The names of encodings and which ones are available are platform-dependent. All R platforms support "" (for the encoding of the current locale), "latin1" and "UTF-8". Generally case is ignored when specifying an encoding.

On most platforms iconvlist provides an alphabetical list of the supported encodings. On others, the information is on the man page for iconv(5) or elsewhere in the man pages (but beware that the system command iconv may not support the same set of encodings as the C functions R calls). Unfortunately, the names are rarely supported across all platforms.

Elements of x which cannot be converted (perhaps because they are invalid or because they cannot be represented in the target encoding) will be returned as NA (or NULL for toRaw = TRUE) unless sub is specified.

Most versions of iconv will allow transliteration by appending '//TRANSLIT' to the to encoding: see the examples.

Encoding "ASCII" is accepted, and on most systems "C" and "POSIX" are synonyms for ASCII. Where "ASCII/TRANSLIT" is unsupported by the OS, "ASCII" is used with sub = "c99" if from UTF-8, else sub = "?". (However, musl's version of "ASCII" substitutes *.)

Elements of x with a declared encoding (UTF-8 or latin1, see Encoding) are converted from that encoding if from = "", otherwise they are taken as being in the encoding specified by from.

Note that implementations of iconv typically do not do much validity checking and will often mis-convert inputs which are invalid in encoding from.

If sub = "Unicode" or sub = "c99" is used for a non-UTF-8 input it is the same as sub = "byte".

Value

If toRaw = FALSE (the default), the value is a character vector of the same length and the same attributes as x (after conversion to a character vector). If conversion fails for an element that element of the result is set to NA_character_. (NB: whether conversion fails is implementation-specific.) NA_character_ inputs give NA_character_ outputs.

If mark = TRUE (the default) the elements of the result have a declared encoding if to is "latin1" or "UTF-8", or if to = "" and the current locale's encoding is detected as Latin-1 (or its superset CP1252 on Windows) or UTF-8.

If toRaw = TRUE, the value is a list of the same length and the same attributes as x whose elements are either NULL (if conversion fails or the input was NA_character_) or a raw vector.

For iconvlist(), a character vector (typically of a few hundred elements) of known encoding names.

Implementation Details

There are three main implementations of `iconv` in use. Linux's most common C runtime, 'glibc', contains one. Several platforms supply versions or emulations of GNU 'libiconv', including previous versions of macOS and FreeBSD, in some cases with additional encodings. On Windows we use a version of Yukihiro Nakadaira's 'win_iconv', which is based on Windows' codepages. (We have added many encoding names for compatibility with other systems.) All three have `iconvlist`, ignore case in encoding names and support '`//TRANSLIT`' (but with different results, and for 'win_iconv' currently a 'best fit' strategy is used except for `to = "ASCII"`).

The macOS 14 implementation is attributed to the 'Citrus Project': the Apple headers declare it as 'compatible' with GNU 'libiconv' 1.11 from 2006. However, it differs in significant ways including using transliteration for conversions which cannot be represented exactly in the target encoding. (It seems this implementation is also used in recent versions of FreeBSD. Earlier versions of macOS used GNU 'libiconv' 1.11 and some CRAN builds still do.) For a failing conversion macOS 14 generally translated character(s) to ? but 14.1 gives an error (so an `NA` result in R).

Most commercial Unixes contain an implementation of `iconv` but none we have encountered have supported the encoding names we need: the 'R Installation and Administration' manual recommended installing GNU 'libiconv' on Solaris and AIX.

Some Linux distributions use 'musl' as their C runtime. This is less comprehensive than 'glibc': it does not support '`//TRANSLIT`' but does inexact conversions (currently using '`*`').

There are other implementations, e.g. NetBSD has used one from the Citrus project (which does not support '`//TRANSLIT`') and there is an older FreeBSD port.

Note that you cannot rely on invalid inputs being detected, especially for `to = "ASCII"` where some implementations allow 8-bit characters and pass them through unchanged or with transliteration or substitution.

Some of the implementations have interesting extra encodings: for example GNU 'libiconv' and macOS 14 allow `to = "C99"` to use '`\uxxxx`' escapes (or if needed '`\Uuxxxxxxxx`') for non-ASCII characters.

Byte Order Marks

most commonly known as 'BOMs'.

Encodings using character units which are more than one byte in size can be written on a file in either big-endian or little-endian order: this applies most commonly to UCS-2, UTF-16 and UTF-32/UCS-4 encodings. Some systems will write the Unicode character `U+FEFF` at the beginning of a file in these encodings and perhaps also in UTF-8. In that usage the character is known as a BOM, and should be handled during input (see the 'Encodings' section under `connection`: re-encoded connections have some special handling of BOMs). The rest of this section applies when this has not been done so `x` starts with a BOM.

Implementations will generally interpret a BOM for `from` given as one of `"UCS-2"`, `"UTF-16"` and `"UTF-32"`. Implementations differ in how they treat BOMs in `x` in other `from` encodings: they may be discarded, returned as character `U+FEFF` or regarded as invalid.

Note

The most portable name for the ISO 8859-15 encoding, commonly known as 'Latin 9', is `"iso885915"`: most platforms support both `"latin-9"` and`"latin9"` but GNU 'libiconv' does

not support the latter. 'musl' (as used by Alpine Linux and other lightweight Linux distributions) supports neither, but R remaps there to "iso885915".

Encoding names "utf8", "mac" and "macroman" are not portable. "utf8" is converted to "UTF-8" for from and to by iconv, but not for e.g. fileEncoding arguments. "macintosh" is the official (and most widely supported) name for 'Mac Roman' (https://en.wikipedia.org/wiki/Mac_OS_Roman).

Using sub substitutes each non-convertible *byte* in the input, so when converting from UTF-8 a non-convertible character may be replaced by two or more bytes. Using sub = "c99" or sub = "Unicode" will be clearer.

See Also

localeToCharset, file.

Examples

```
## In principle, as not all systems have iconvlist
try(utils::head(iconvlist(), n = 50))

## Not run:
## convert from Latin-2 to UTF-8: two of the glibc iconv variants.
iconv(x, "ISO_8859-2", "UTF-8")
iconv(x, "LATIN2", "UTF-8")

## End(Not run)

## Both x below are in latin1 and will only display correctly in a
## locale that can represent and display latin1.
x <- "fran\xE7ais"
Encoding(x) <- "latin1"
x
charToRaw(xx <- iconv(x, "latin1", "UTF-8"))
xx

## The results in the comments are those from glibc and GNU libiconv
iconv(x, "latin1", "ASCII")           #   NA
iconv(x, "latin1", "ASCII", "?")      # "fran?ais"
iconv(x, "latin1", "ASCII", "")       # "franais"
iconv(x, "latin1", "ASCII", "byte")   # "fran<e7>ais"
iconv(xx, "UTF-8", "ASCII", "Unicode")# "fran<U+00E7>ais"
iconv(xx, "UTF-8", "ASCII", "c99")    # "fran\\u00e7ais"

## Extracts from old R help files (they are nowadays in UTF-8)
x <- c("Ekstr\xf8m", "J\xf6reskog", "bi\xdfchen Z\xfcrcher")
Encoding(x) <- "latin1"
x
try(iconv(x, "latin1", "ASCII//TRANSLIT"))  # platform-dependent
## glibc gives "Ekstroem" "Joreskog" "bisschen Zurcher"
## macOS 14 gives "Ekstrom" "J\"oreskog" "bisschen Z\"urcher"
## musl gives "Ekstr*m" "J*reskog" "bi*chen Z*rcher"
iconv(x, "latin1", "ASCII", sub = "byte")
```

```
## and for Windows' 'Unicode'
str(xx <- iconv(x, "latin1", "UTF-16LE", toRaw = TRUE))
iconv(xx, "UTF-16LE", "UTF-8")

emoji <- "\U0001f604"
iconv(emoji,, "latin1", sub = "Unicode") # "<U+1F604>"
iconv(emoji,, "latin1", sub = "c99")
```

`icuSetCollate` *Setup Collation by ICU*

Description

Controls the way collation is done by ICU (an optional part of the R build).

Usage

```
icuSetCollate(...)

icuGetCollate(type = c("actual", "valid"))
```

Arguments

`...`	named arguments, see 'Details'.
`type`	a character string: either the `"actual"` locale in use for collation, or the most specific locale which would be `"valid"`. Can be abbreviated.

Details

Optionally, R can be built to collate character strings by ICU (https://icu.unicode.org/). For such systems, `icuSetCollate` can be used to tune the way collation is done. On other builds calling this function does nothing, with a warning.

Possible arguments are

`locale`**:** A character string such as `"da_DK"` giving the language and country whose collation rules are to be used. If present, this should be the first argument.

`case_first`**:** `"upper"`, `"lower"` or `"default"`, asking for upper- or lower-case characters to be sorted first. The default is usually lower-case first, but not in all languages (not under the default settings for Danish, for example).

`alternate_handling`**:** Controls the handling of 'variable' characters (mainly punctuation and symbols). Possible values are `"non_ignorable"` (primary strength) and `"shifted"` (quaternary strength).

`strength`**:** Which components should be used? Possible values `"primary"`, `"secondary"`, `"tertiary"` (default), `"quaternary"` and `"identical"`.

`french_collation`: In a French locale the way accents affect collation is from right to left, whereas in most other locales it is from left to right. Possible values `"on"`, `"off"` and `"default"`.

`normalization`: Should strings be normalized? Possible values are `"on"` and `"off"` (default). This affects the collation of composite characters.

`case_level`: An additional level between secondary and tertiary, used to distinguish large and small Japanese Kana characters. Possible values `"on"` and `"off"` (default).

`hiragana_quaternary`: Possible values `"on"` (sort Hiragana first at quaternary level) and `"off"`.

Only the first three are likely to be of interest except to those with a detailed understanding of collation and specialized requirements.

Some special values are accepted for `locale`:

`"none"`: ICU is not used for collation: the OS's collation services are used instead.

`"ASCII"`: ICU is not used for collation: the C function `strcmp` is used instead, which should sort byte-by-byte in (unsigned) numerical order.

`"default"`: obtains the locale from the OS as is done at the start of the session (except on Windows). If environment variable `R_ICU_LOCALE` is set to a non-empty value, its value is used rather than consulting the OS, unless environment variable `LC_ALL` is set to 'C' (or unset but `LC_COLLATE` is set to 'C').

`""`, `"root"`: the 'root' collation: see https://www.unicode.org/reports/tr35/tr35-collation.html#Root_Collation.

For the specifications of 'real' ICU locales, see https://unicode-org.github.io/icu/userguide/locale/. Note that ICU does not report that a locale is not supported, but falls back to its idea of 'best fit' (which could be rather different and is reported by `icuGetCollate("actual")`, often `"root"`). Most English locales fall back to `"root"` as although e.g. `"en_GB"` is a valid locale (at least on some platforms), it contains no special rules for collation. Note that `"C"` is not a supported ICU locale and hence `R_ICU_LOCALE` should never be set to `"C"`.

Some examples are `case_level = "on"`, `strength = "primary"` to ignore accent differences and `alternate_handling = "shifted"` to ignore space and punctuation characters.

Initially ICU will not be used for collation if the OS is set to use the `C` locale for collation and `R_ICU_LOCALE` is not set. Once this function is called with a value for `locale`, ICU will be used until it is called again with `locale = "none"`. ICU will not be used once `Sys.setlocale` is called with a `"C"` value for `LC_ALL` or `LC_COLLATE`, even if `R_ICU_LOCALE` is set. ICU will be used again honoring `R_ICU_LOCALE` once `Sys.setlocale` is called to set a different collation order. Environment variables `LC_ALL` (or `LC_COLLATE`) take precedence over `R_ICU_LOCALE` if and only if they are set to 'C'. Due to the interaction with other ways of setting the collation order, `R_ICU_LOCALE` should be used with care and only when needed.

All customizations are reset to the default for the locale if `locale` is specified: the collation engine is reset if the OS collation locate category is changed by `Sys.setlocale`.

Value

For `icuGetCollate`, a character string describing the ICU locale in use (which may be reported as `"ICU not in use"`). The 'actual' locale may be simpler than the requested locale: for example `"da"` rather than `"da_DK"`: English locales are likely to report `"root"`.

Note

Except on Windows, ICU is used by default wherever it is available. As it works internally in UTF-8, it will be most efficient in UTF-8 locales.

On Windows, R is normally built including ICU, but it will only be used if environment variable R_ICU_LOCALE had been set when R is started or after icuSetCollate is called to select the locale (as ICU and Windows differ in their idea of locale names). Note that icuSetCollate(locale = "default") should work reasonably well, but finds the system default ignoring environment variables such as LC_COLLATE.

See Also

Comparison, sort.

capabilities for whether ICU is available; extSoftVersion for its version.

The ICU user guide chapter on collation (https://unicode-org.github.io/icu/userguide/collation/).

Examples

```
## These examples depend on having ICU available, and on the locale.
## As we don't know the current settings, we can only reset to the default.
if(capabilities("ICU")) withAutoprint({
    icuGetCollate()
    icuGetCollate("valid")
    x <- c("Aarhus", "aarhus", "safe", "test", "Zoo")
    sort(x)
    icuSetCollate(case_first = "upper"); sort(x)
    icuSetCollate(case_first = "lower"); sort(x)

    ## Danish collates upper-case-first and with 'aa' as a single letter
    icuSetCollate(locale = "da_DK", case_first = "default"); sort(x)
    ## Estonian collates Z between S and T
    icuSetCollate(locale = "et_EE"); sort(x)
    icuSetCollate(locale = "default"); icuGetCollate("valid")
})
```

---

identical *Test Objects for Exact Equality*

Description

The safe and reliable way to test two objects for being *exactly* equal. It returns TRUE in this case, FALSE in every other case.

Usage

```
identical(x, y, num.eq = TRUE, single.NA = TRUE, attrib.as.set = TRUE,
          ignore.bytecode = TRUE, ignore.environment = FALSE,
          ignore.srcref = TRUE, extptr.as.ref = FALSE)
```

Arguments

x, y	any R objects.
num.eq	logical indicating if (double and complex non-NA) numbers should be compared using == ('equal'), or by bitwise comparison. The latter (non-default) differentiates between -0 and +0.
single.NA	logical indicating if there is conceptually just one numeric NA and one NaN; single.NA = FALSE differentiates bit patterns.
attrib.as.set	logical indicating if attributes of x and y should be treated as *unordered* tagged pairlists ("sets"); this currently also applies to slots of S4 objects. It may well be too strict to set attrib.as.set = FALSE.
ignore.bytecode	logical indicating if byte code should be ignored when comparing closures.
ignore.environment	logical indicating if their environments should be ignored when comparing closures.
ignore.srcref	logical indicating if their "srcref" attributes should be ignored when comparing closures.
extptr.as.ref	logical indicating whether external pointer objects should be compared as reference objects and considered identical only if they are the same object in memory. By default, external pointers are considered identical if the addresses they contain are identical.

Details

A call to identical is the way to test exact equality in if and while statements, as well as in logical expressions that use && or ||. In all these applications you need to be assured of getting a single logical value.

Users often use the comparison operators, such as == or !=, in these situations. It looks natural, but it is not what these operators are designed to do in R. They return an object like the arguments. If you expected x and y to be of length 1, but it happened that one of them was not, you will *not* get a single FALSE. Similarly, if one of the arguments is NA, the result is also NA. In either case, the expression if(x == y).... won't work as expected.

The function all.equal is also sometimes used to test equality this way, but was intended for something different: it allows for small differences in numeric results.

The computations in identical are also reliable and usually fast. There should never be an error. The only known way to kill identical is by having an invalid pointer at the C level, generating a memory fault. It will usually find inequality quickly. Checking equality for two large, complicated objects can take longer if the objects are identical or nearly so, but represent completely independent copies. For most applications, however, the computational cost should be negligible.

If single.NA is true, as by default, identical sees NaN as different from NA_real_, but all NaNs are equal (and all NA of the same type are equal).

Character strings (except those in marked encoding "bytes") are regarded as identical even if they are in different marked encodings but would agree when translated to UTF-8. A character string in marked encoding "bytes" is only regarded as identical to a character string in the same encoding and with the same content.

If `attrib.as.set` is true, as by default, comparison of attributes view them as a set (and not a vector, so order is not tested).

If `ignore.bytecode` is true (the default), the compiled bytecode of a function (see `cmpfun`) will be ignored in the comparison. If it is false, functions will compare equal only if they are copies of the same compiled object (or both are uncompiled). To check whether two different compiles are equal, you should compare the results of `disassemble()`.

You almost never want to use `identical` on datetimes of class `"POSIXlt"`: not only can different times in the different time zones represent the same time and time zones have multiple names, but several of the components are optional.

Note that the strictest test for equality is

```
identical(x, y,
          num.eq = FALSE, single.NA = FALSE, attrib.as.set = FALSE,
          ignore.bytecode = FALSE, ignore.environment = FALSE,
          ignore.srcref = FALSE, extptr.as.ref = TRUE)
```

Value

A single logical value, `TRUE` or `FALSE`, never `NA` and never anything other than a single value.

Author(s)

John Chambers and R Core

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`all.equal` for descriptions of how two objects differ; `Comparison` and `Logic` for elementwise comparisons.

Examples

```
identical(1, NULL) ## FALSE -- don't try this with ==
identical(1, 1.)   ## TRUE in R (both are stored as doubles)
identical(1, as.integer(1)) ## FALSE, stored as different types

x <- 1.0; y <- 0.99999999999
## how to test for object equality allowing for numeric fuzz :
(E <- all.equal(x, y))
identical(TRUE, E)
isTRUE(E) # alternative test
## If all.equal thinks the objects are different, it returns a
## character string, and the above expression evaluates to FALSE

## even for unusual R objects :
identical(.GlobalEnv, environment())
```

```
### ------- Pickyness Flags : ----------------------------

## the infamous example:
identical(0., -0.) # TRUE, i.e. not differentiated
identical(0., -0., num.eq = FALSE)
## similar:
identical(NaN, -NaN) # TRUE
identical(NaN, -NaN, single.NA = FALSE) # differ on bit-level

### For functions ("closure"s): ------------------------------------------------
###     ~~~~~~~~~
f <- function(x) x
f
g <- compiler::cmpfun(f)
g
identical(f, g)                        # TRUE, as bytecode is ignored by default
identical(f, g, ignore.bytecode=FALSE) # FALSE: bytecode differs

## GLM families contain several functions, some of which share an environment:
p1 <- poisson() ; p2 <- poisson()
identical(p1, p2)                          # FALSE
identical(p1, p2, ignore.environment=TRUE) # TRUE

## in interactive use, the 'keep.source' option is typically true:
op <- options(keep.source = TRUE) # and so, these have differing "srcref" :
f1 <- function() {}
f2 <- function() {}
identical(f1,f2)# ignore.srcref= TRUE : TRUE
identical(f1,f2,  ignore.srcref=FALSE)# FALSE
options(op) # revert to previous state
```

`identity` *Identity Function*

Description

A trivial identity function returning its argument.

Usage

```
identity(x)
```

Arguments

x	an R object.

See Also

`diag` creates diagonal matrices, including identity ones.

`ifelse` *Conditional Element Selection*

Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Usage

```
ifelse(test, yes, no)
```

Arguments

`test`	an object which can be coerced to logical mode.
`yes`	return values for true elements of `test`.
`no`	return values for false elements of `test`.

Details

If `yes` or `no` are too short, their elements are recycled. `yes` will be evaluated if and only if any element of `test` is true, and analogously for `no`.

Missing values in `test` give missing values in the result.

Value

A vector of the same length and attributes (including dimensions and `"class"`) as `test` and data values from the values of `yes` or `no`. The mode of the answer will be coerced from logical to accommodate first any values taken from `yes` and then any values taken from `no`.

Warning

The mode of the result may depend on the value of `test` (see the examples), and the class attribute (see `oldClass`) of the result is taken from `test` and may be inappropriate for the values selected from `yes` and `no`.

Sometimes it is better to use a construction such as

```
(tmp <- yes; tmp[!test] <- no[!test]; tmp)
```

, possibly extended to handle missing values in `test`.

Further note that `if(test) yes else no` is much more efficient and often much preferable to `ifelse(test, yes, no)` whenever `test` is a simple true/false result, i.e., when `length(test) == 1`.

The `srcref` attribute of functions is handled specially: if `test` is a simple true result and `yes` evaluates to a function with `srcref` attribute, `ifelse` returns `yes` including its attribute (the same applies to a false `test` and `no` argument). This functionality is only for backwards compatibility, the form `if(test) yes else no` should be used whenever `yes` and `no` are functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

if.

Examples

```
x <- c(6:-4)
sqrt(x)  #- gives warning
sqrt(ifelse(x >= 0, x, NA))  # no warning

## Note: the following also gives the warning !
ifelse(x >= 0, sqrt(x), NA)


## ifelse() strips attributes
## This is important when working with Dates and factors
x <- seq(as.Date("2000-02-29"), as.Date("2004-10-04"), by = "1 month")
## has many "yyyy-mm-29", but a few "yyyy-03-01" in the non-leap years
y <- ifelse(as.POSIXlt(x)$mday == 29, x, NA)
head(y) # not what you expected ... ==> need restore the class attribute:
class(y) <- class(x)
y
## This is a (not atypical) case where it is better *not* to use ifelse(),
## but rather the more efficient and still clear:
y2 <- x
y2[as.POSIXlt(x)$mday != 29] <- NA
## which gives the same as ifelse()+class() hack:
stopifnot(identical(y2, y))


## example of different return modes (and 'test' alone determining length):
yes <- 1:3
no  <- pi^(1:4)
utils::str( ifelse(NA,    yes, no) ) # logical, length 1
utils::str( ifelse(TRUE,  yes, no) ) # integer, length 1
utils::str( ifelse(FALSE, yes, no) ) # double,  length 1
```

---

integer *Integer Vectors*

Description

Creates or tests for objects of type "integer".

Usage

```
integer(length = 0)
as.integer(x, ...)
is.integer(x)
```

Arguments

length	a non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that (small) integer data can be represented exactly and compactly.

Note that current implementations of R use 32-bit integers for integer vectors, so the range of representable integers is restricted to about $\pm 2 \times 10^9$: doubles can hold much larger integers exactly.

Value

integer creates a integer vector of the specified length. Each element of the vector is equal to 0.

as.integer attempts to coerce its argument to be of integer type. The answer will be NA unless the coercion succeeds. Real values larger in modulus than the largest integer are coerced to NA (unlike S which gives the most extreme integer of the same sign). Non-integral numeric values are truncated towards zero (i.e., as.integer(x) equals trunc(x) there), and imaginary parts of complex numbers are discarded (with a warning). Character strings containing optional whitespace followed by either a decimal representation or a hexadecimal representation (starting with 0x or 0X) can be converted, as well as any allowed by the platform for real numbers. Like as.vector it strips attributes including names. (To ensure that an object x is of integer type without stripping attributes, use storage.mode(x) <- "integer".)

is.integer returns TRUE or FALSE depending on whether its argument is of integer type or not, unless it is a factor when it returns FALSE.

Note

is.integer(x) does **not** test if x contains integer numbers! For that, use round, as in the function is.wholenumber(x) in the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

numeric, storage.mode.

round (and ceiling and floor on that help page) to convert to integral values.

Examples

```
## as.integer() truncates:
x <- pi * c(-1:1, 10)
as.integer(x)

is.integer(1) # is FALSE !

is.wholenumber <-
    function(x, tol = .Machine$double.eps^0.5)  abs(x - round(x)) < tol
is.wholenumber(1) # is TRUE
(x <- seq(1, 5, by = 0.5) )
is.wholenumber( x ) #-->  TRUE FALSE TRUE ...
```

---

interaction *Compute Factor Interactions*

Description

interaction computes a factor which represents the interaction of the given factors. The result of interaction is always unordered.

Usage

```
interaction(..., drop = FALSE, sep = ".", lex.order = FALSE)
```

Arguments

...	the factors for which interaction is to be computed, or a single list giving those factors.
drop	if drop is TRUE, unused factor levels are dropped from the result. The default is to retain all factor levels.
sep	string to construct the new level labels by joining the constituent ones.
lex.order	logical indicating if the order of factor concatenation should be lexically ordered.

Value

A factor which represents the interaction of the given factors. The levels are labelled as the levels of the individual factors joined by sep which is . by default.

By default, when lex.order = FALSE, the levels are ordered so the level of the first factor varies fastest, then the second and so on. This is the reverse of lexicographic ordering (which you can get by lex.order = TRUE), and differs from :. (It is done this way for compatibility with S.)

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`factor`; `:` where `f:g` is similar to `interaction(f, g, sep = ":")` when `f` and `g` are factors.

Examples

```
a <- gl(2, 4, 8)
b <- gl(2, 2, 8, labels = c("ctrl", "treat"))
s <- gl(2, 1, 8, labels = c("M", "F"))
interaction(a, b)
interaction(a, b, s, sep = ":")
stopifnot(identical(a:s,
                    interaction(a, s, sep = ":", lex.order = TRUE)),
          identical(a:s:b,
                    interaction(a, s, b, sep = ":", lex.order = TRUE)))
```

---

interactive *Is R Running Interactively?*

Description

Return `TRUE` when R is being used interactively and `FALSE` otherwise.

Usage

```
interactive()
```

Details

An interactive R session is one in which it is assumed that there is a human operator to interact with, so for example R can prompt for corrections to incorrect input or ask what to do next or if it is OK to move to the next plot.

GUI consoles will arrange to start R in an interactive session. When R is run in a terminal (via `Rterm.exe` on Windows), it assumes that it is interactive if '`stdin`' is connected to a (pseudo-)terminal and not if '`stdin`' is redirected to a file or pipe. Command-line options '`--interactive`' (Unix) and '`--ess`' (Windows, `Rterm.exe`) override the default assumption. (On a Unix-alike, whether the `readline` command-line editor is used is **not** overridden by '`--interactive`'.)

Embedded uses of R can set a session to be interactive or not.

Internally, whether a session is interactive determines

- how some errors are handled and reported, e.g. see `stop` and `options("showWarnCalls")`.
- whether one of '`--save`', '`--no-save`' or '`--vanilla`' is required, and if R ever asks whether to save the workspace.

- the choice of default graphics device launched when needed and by dev.new: see options("device")
- whether graphics devices ever ask for confirmation of a new page.

In addition, R's own R code makes use of interactive(): for example help, debugger and install.packages do.

Note

This is a primitive function.

See Also

source, .First

Examples

```
.First <- function() if(interactive()) x11()
```

Internal *Call an Internal Function*

Description

.Internal performs a call to an internal code which is built in to the R interpreter.

Only true R wizards should even consider using this function, and only R developers can add to the list of internal functions.

Usage

```
.Internal(call)
```

Arguments

call	a call expression

See Also

.Primitive, .External (the nearest equivalent available to users).

`InternalMethods` *Internal Generic Functions*

Description

Many R-internal functions are *generic* and allow methods to be written for.

Details

The following primitive and internal functions are *generic*, i.e., you can write `methods` for them:

`[`, `[[`, `$`, `[<-`, `[[<-`, `$<-`,

`length`, `length<-`, `lengths`, `dimnames`, `dimnames<-`, `dim`, `dim<-`, `names`, `names<-`, `levels<-`, `@`, `@<-`,

`c`, `unlist`, `cbind`, `rbind`,

`as.character`, `as.complex`, `as.double`, `as.integer`, `as.logical`, `as.raw`, `as.vector`, `as.call`, `as.environment` `is.array`, `is.matrix`, `is.na`, `anyNA`, `is.nan`, `is.finite` `is.infinite` `is.numeric`, `nchar` `rep`, `rep.int` `rep_len` `seq.int` (which dispatches methods for `"seq"`), `is.unsorted` and `xtfrm`

In addition, `is.name` is a synonym for `is.symbol` and dispatches methods for the latter. Similarly, `as.numeric` is a synonym for `as.double` and dispatches methods for the latter, i.e., S3 methods are for `as.double`, whereas S4 methods are to be written for `as.numeric`.

Note that all of the group generic functions are also internal/primitive and allow methods to be written for them.

`.S3PrimitiveGenerics` is a character vector listing the primitives which are internal generic and not group generic, (not only for S3 but also S4). Similarly, the `.internalGenerics` character vector contains the names of the internal (via `.Internal(..)`) non-primitive functions which are internally generic.

Note

For efficiency, internal dispatch only occurs on *objects*, that is those for which `is.object` returns true.

See Also

`methods` for the methods which are available.

`invisible` *Change the Print Mode to Invisible*

Description

Return a (temporarily) invisible copy of an object.

Usage

```
invisible(x = NULL)
```

Arguments

`x`	an arbitrary R object, by default `NULL`.

Details

This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned.

This is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`withVisible`, `return`, `function`.

Examples

```
# These functions both return their argument
f1 <- function(x) x
f2 <- function(x) invisible(x)
f1(1)  # prints
f2(1)  # does not
```

`is.finite` *Finite, Infinite and NaN Numbers*

Description

`is.finite` and `is.infinite` return a vector of the same length as `x`, indicating which elements are finite (not infinite and not missing) or infinite.

`Inf` and `-Inf` are positive and negative infinity whereas `NaN` means 'Not a Number'. (These apply to numeric values and real and imaginary parts of complex values but not to values of integer vectors.) `Inf` and `NaN` (as well as `NA`) are reserved words in the R language.

Usage

```
is.finite(x)
is.infinite(x)
is.nan(x)

Inf
NaN
```

Arguments

`x`	R object to be tested: the default methods handle atomic vectors.

Details

`is.finite` returns a vector of the same length as `x` the j-th element of which is `TRUE` if `x[j]` is finite (i.e., it is not one of the values `NA`, `NaN`, `Inf` or `-Inf`) and `FALSE` otherwise. Complex numbers are finite if both the real and imaginary parts are.

`is.infinite` returns a vector of the same length as `x` the j-th element of which is `TRUE` if `x[j]` is infinite (i.e., equal to one of `Inf` or `-Inf`) and `FALSE` otherwise. This will be false unless `x` is numeric or complex. Complex numbers are infinite if either the real or the imaginary part is.

`is.nan` tests if a numeric value is `NaN`. Do not test equality to `NaN`, or even use `identical`, since systems typically have many different NaN values. One of these is used for the numeric missing value `NA`, and `is.nan` is false for that value. A complex number is regarded as `NaN` if either the real or imaginary part is `NaN` but not `NA`. All elements of logical, integer and raw vectors are considered not to be NaN.

All three functions accept `NULL` as input and return a length zero result. The default methods accept character and raw vectors, and return `FALSE` for all entries. Prior to R version 2.14.0 they accepted all input, returning `FALSE` for most non-numeric values; cases which are not atomic vectors are now signalled as errors.

All three functions are generic: you can write methods to handle specific classes of objects, see InternalMethods.

Value

A logical vector of the same length as `x`: `dim`, `dimnames` and `names` attributes are preserved.

Note

In R, basically all mathematical functions (including basic `Arithmetic`), are supposed to work properly with `+/- Inf` and `NaN` as input or output.

The basic rule should be that calls and relations with `Inf`s really are statements with a proper mathematical *limit*.

Computations involving `NaN` will return `NaN` or perhaps `NA`: which of those two is not guaranteed and may depend on the R platform (since compilers may re-order computations).

References

The IEC 60559 standard, also known as the ANSI/IEEE 754 Floating-Point Standard.

https://en.wikipedia.org/wiki/NaN.

D. Goldberg (1991). What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Computing Surveys*, **23**(1), 5–48. doi:10.1145/103162.103163.
Also available at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

The C99 function `isfinite` is used for `is.finite`.

See Also

`NA`, '*Not Available*' which is not a number as well, however usually used for missing values and applies to many modes, not just numeric and complex.

`Arithmetic`, `double`.

Examples

```
pi / 0 ## = Inf a non-zero number divided by zero creates infinity
0 / 0  ## =  NaN

1/0 + 1/0 # Inf
1/0 - 1/0 # NaN

stopifnot(
    1/0 == Inf,
    1/Inf == 0
)
sin(Inf)
cos(Inf)
tan(Inf)
```

is.function *Is an Object of Type (Primitive) Function?*

Description

Checks whether its argument is a (primitive) function.

Usage

```
is.function(x)
is.primitive(x)
```

Arguments

x	an R object.

Details

`is.primitive(x)` tests if `x` is a primitive function, i.e, if `typeof(x)` is either `"builtin"` or `"special"`.

Value

`TRUE` if `x` is a (primitive) function, and `FALSE` otherwise.

Examples

```
is.function(1) # FALSE
is.function (is.primitive) # TRUE: it is a function, but ..
is.primitive(is.primitive) # FALSE: it's not a primitive one, whereas
is.primitive(is.function)  # TRUE: that one *is*
```

is.language *Is an Object a Language Object?*

Description

`is.language` returns `TRUE` if `x` is a variable `name`, a `call`, or an `expression`.

Usage

```
is.language(x)
```

Arguments

x	object to be tested.

Note

A name is also known as 'symbol', from its type (typeof), see is.symbol.

If typeof(x) == "language", then is.language(x) is always true, but the reverse does not hold as expressions or names y also fulfill is.language(y), see the examples.

This is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
ll <- list(a = expression(x^2 - 2*x + 1), b = as.name("Jim"),
           c = as.expression(exp(1)), d = call("sin", pi))
sapply(ll, typeof)
sapply(ll, mode)
stopifnot(sapply(ll, is.language))
```

---

is.object *Is an Object 'internally classed'?*

Description

A function mostly for internal use. It returns TRUE if the object x has the R internal OBJECT bit set, and FALSE otherwise. The OBJECT bit is set when a "class" attribute is added and removed when that attribute is removed, so this is a very efficient way to check if an object has a class attribute. (S4 objects always should.)

Note that typical basic ('atomic', see is.atomic) R vectors and arrays x are *not* objects in the above sense as attributes(x) does *not* contain "class".

Usage

```
is.object(x)
```

Arguments

x	object to be tested.

Note

This is a primitive function.

See Also

class, and methods.

isS4.

Examples

```
is.object(1) # FALSE
is.object(as.factor(1:3)) # TRUE
```

`is.recursive` *Is an Object Atomic or Recursive?*

Description

`is.atomic` returns `TRUE` if `x` is of an atomic type and `FALSE` otherwise.

`is.recursive` returns `TRUE` if `x` has a recursive (list-like) structure and `FALSE` otherwise.

Usage

```
is.atomic(x)
is.recursive(x)
```

Arguments

`x`	object to be tested.

Details

`is.atomic` is true for the atomic types (`"logical"`, `"integer"`, `"numeric"`, `"complex"`, `"character"` and `"raw"`).

Most types of objects are regarded as recursive. Exceptions are the atomic types, `NULL`, symbols (as given by `as.name`), `S4` objects with slots, external pointers, and—rarely visible from R—weak references and byte code, see `typeof`.

It is common to call the atomic types 'atomic vectors', but note that `is.vector` imposes further restrictions: an object can be atomic but not a vector (in that sense).

These are primitive functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`is.list`, `is.language`, etc, and the `demo("is.things")`.

Examples

```
require(stats)

is.a.r <- function(x) c(is.atomic(x), is.recursive(x))

is.a.r(c(a = 1, b = 3)) # TRUE FALSE
is.a.r(list())          # FALSE TRUE - a list is a list
is.a.r(list(2))         # FALSE TRUE
is.a.r(lm)              # FALSE TRUE
is.a.r(y ~ x)           # FALSE TRUE
is.a.r(expression(x+1)) # FALSE TRUE
is.a.r(quote(exp))      # FALSE FALSE
is.a.r(NULL)            # FALSE FALSE
```

---

is.single *Is an Object of Single Precision Type?*

Description

`is.single` reports an error. There are no single precision values in R.

Usage

```
is.single(x)
```

Arguments

`x`	object to be tested.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

is.unsorted *Test if an Object is Not Sorted*

Description

Test if an object is not sorted (in increasing order), without the cost of sorting it.

Usage

```
is.unsorted(x, na.rm = FALSE, strictly = FALSE)
```

Arguments

x	an R object with a class or a numeric, complex, character, logical or raw vector.
na.rm	logical. Should missing values be removed before checking?
strictly	logical indicating if the check should be for *strictly* increasing values.

Details

is.unsorted is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Value

A length-one logical value. All objects of length 0 or 1 are sorted. Otherwise, the result will be NA except for atomic vectors and objects with an S3 class (where the >= or > method is used to compare x[i] with x[i-1] for i in 2:length(x)) or with an S4 class where you have to provide a method for is.unsorted().

Note

This function is designed for objects with one-dimensional indices, as described above. Data frames, matrices and other arrays may give surprising results.

See Also

sort, order.

ISOdatetime *Date-time Conversion Functions from Numeric Representations*

Description

Convenience wrappers to create date-times from numeric representations.

Usage

```
ISOdatetime(year, month, day, hour, min, sec, tz = "")
ISOdate(year, month, day, hour = 12, min = 0, sec = 0, tz = "GMT")
```

Arguments

year, month, day	numerical values to specify a day.
hour, min, sec	numerical values for a time within a day. Fractional seconds are allowed.
tz	a time zone specification to be used for the conversion. "" is the current time zone and "GMT" is UTC. Invalid values are most commonly treated as UTC, on some platforms with a warning.

Details

ISOdatetime and ISOdate are convenience wrappers for strptime that differ only in their defaults and that ISOdate sets UTC as the time zone. For dates without times it would normally be better to use the "Date" class.

The main arguments will be recycled using the usual recycling rules.

Because these make use of strptime, only years in the range 0:9999 are accepted.

Value

An object of class "POSIXct".

See Also

DateTimeClasses for details of the date-time classes; strptime for conversions from character strings.

---

isS4 *Test for an S4 object*

Description

Tests whether the object is an instance of an S4 class.

Usage

```
isS4(object)

asS4(object, flag = TRUE, complete = TRUE)
asS3(object, flag = TRUE, complete = TRUE)
```

Arguments

object	Any R object.
flag	Optional, logical: indicate direction of conversion.
complete	Optional, logical: whether conversion to S3 is completed. Not usually needed, but see the details section.

Details

Note that isS4 does not rely on the **methods** package, so in particular it can be used to detect the need to require that package.

asS3 uses the value of complete to control whether an attempt is made to transform object into a valid object of the implied S3 class. If complete is TRUE, then an object from an S4 class extending an S3 class will be transformed into an S3 object with the corresponding S3 class (see S3Part). This includes classes extending the pseudo-classes array and matrix: such objects will have their class attribute set to NULL.

isS4 is primitive.

Value

isS4 always returns TRUE or FALSE according to whether the internal flag marking an S4 object has been turned on for this object.

asS4 and asS3 will turn this flag on or off, and asS3 will set the class from the objects .S3Class slot if one exists. Note that asS3 will *not* turn the object into an S3 object unless there is a valid conversion; that is, an object of type other than "S4" for which the S4 object is an extension, unless argument complete is FALSE.

See Also

is.object for a more general test; Introduction for general information on S4; Classes_Details for more on S4 class definitions.

Examples

```
isS4(pi) # FALSE
isS4(getClass("MethodDefinition")) # TRUE
```

---

isSymmetric *Test if a Matrix or other Object is Symmetric (Hermitian)*

Description

Generic function to test if object is symmetric or not. Currently only a matrix method is implemented, where a complex matrix Z must be "Hermitian" for isSymmetric(Z) to be true.

Usage

```
isSymmetric(object, ...)
## S3 method for class 'matrix'
isSymmetric(object, tol = 100 * .Machine$double.eps,
            tol1 = 8 * tol, ...)
```

Arguments

object	any R object; a matrix for the matrix method.
tol	numeric scalar >= 0. Smaller differences are not considered, see all.equal.numeric.
tol1	numeric scalar >= 0. isSymmetric.matrix() 'pre-tests' the first and last few rows for fast detection of 'obviously' asymmetric cases with this tolerance. Setting it to length zero will skip the pre-tests.
...	further arguments passed to methods; the matrix method passes these to all.equal. If the row and column names of object are allowed to differ for the symmetry check do use check.attributes = FALSE!

Details

The matrix method is used inside eigen by default to test symmetry of matrices *up to rounding error*, using all.equal. It might not be appropriate in all situations.

Note that a matrix m is only symmetric if its rownames and colnames are identical. Consider using unname(m).

Value

logical indicating if object is symmetric or not.

See Also

eigen which calls isSymmetric when its symmetric argument is missing.

Examples

```
isSymmetric(D3 <- diag(3)) # -> TRUE

D3[2, 1] <- 1e-100
D3
isSymmetric(D3) # TRUE
isSymmetric(D3, tol = 0) # FALSE for zero-tolerance

## Complex Matrices - Hermitian or not
Z <- sqrt(matrix(-1:2 + 0i, 2)); Z <- t(Conj(Z)) %*% Z
Z
isSymmetric(Z)      # TRUE
isSymmetric(Z + 1)  # TRUE
isSymmetric(Z + 1i) # FALSE -- a Hermitian matrix has a *real* diagonal

colnames(D3) <- c("X", "Y", "Z")
isSymmetric(D3)                          # FALSE (as row and column names differ)
isSymmetric(D3, check.attributes=FALSE) # TRUE  (as names are not checked)
```

---

jitter *'Jitter' (Add Noise) to Numbers*

Description

Add a small amount of noise to a numeric vector.

Usage

```
jitter(x, factor = 1, amount = NULL)
```

Arguments

`x`	numeric vector to which *jitter* should be added.
`factor`	numeric.
`amount`	numeric; if positive, used as *amount* (see below), otherwise, if = `0` the default is `factor * z/50`. Default (`NULL`): `factor * d/5` where `d` is about the smallest difference between `x` values.

Details

The result, say `r`, is `r <- x + runif(n, -a, a)` where `n <- length(x)` and `a` is the `amount` argument (if specified).

Let `z <- max(x) - min(x)` (assuming the usual case). The amount `a` to be added is either provided as *positive* argument `amount` or otherwise computed from `z`, as follows:

If `amount == 0`, we set `a <- factor * z/50` (same as S).

If `amount` is `NULL` (*default*), we set `a <- factor * d/5` where *d* is the smallest difference between adjacent unique (apart from fuzz) `x` values.

Value

`jitter(x, ...)` returns a numeric of the same length as `x`, but with an `amount` of noise added in order to break ties.

Author(s)

Werner Stahel and Martin Maechler, ETH Zurich

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P.A. (1983) *Graphical Methods for Data Analysis.* Wadsworth; figures 2.8, 4.22, 5.4.

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`rug` which you may want to combine with `jitter`.

Examples

```
round(jitter(c(rep(1, 3), rep(1.2, 4), rep(3, 3))), 3)
## These two 'fail' with S-plus 3.x:
jitter(rep(0, 7))
jitter(rep(10000, 5))
```

kappa *Compute or Estimate the Condition Number of a Matrix*

Description

The condition number of a regular (square) matrix is the product of the *norm* of the matrix and the norm of its inverse (or pseudo-inverse), and hence depends on the kind of matrix-norm.

`kappa()` computes by default (an estimate of) the 2-norm condition number of a matrix or of the R matrix of a QR decomposition, perhaps of a linear fit. The 2-norm condition number can be shown to be the ratio of the largest to the smallest *non-zero* singular value of the matrix.

`rcond()` computes an approximation of the **r**eciprocal **cond**ition number, see the details.

Usage

```
kappa(z, ...)
## Default S3 method:
kappa(z, exact = FALSE,
      norm = NULL, method = c("qr", "direct"),
      inv_z = solve(z),
      triangular = FALSE, uplo = "U", ...)

## S3 method for class 'lm'
kappa(z, ...)
## S3 method for class 'qr'
kappa(z, ...)

.kappa_tri(z, exact = FALSE, LINPACK = TRUE, norm = NULL, uplo = "U", ...)

rcond(x, norm = c("O","I","1"), triangular = FALSE, uplo = "U", ...)
```

Arguments

z, x	a numeric or complex matrix or a result of qr or a fit from a class inheriting from "lm".
exact	logical. Should the result be exact (up to small rounding error) as opposed to fast (but quite inaccurate)?
norm	character string, specifying the matrix norm with respect to which the condition number is to be computed, see the function norm(). For kappa(), the default is "2", for rcond() it is "O", and for .kappa_tri()), the default depends on exact: if that is true, the default is "2", otherwise "O", meaning the **O**ne- or 1-norm. For exact=FALSE, the currently only other possible value is "I" for the infinity norm. For exact=TRUE, norm may be "2", or any of the possible type values in norm(., type = *).
method	a partially matched character string specifying the method to be used; "qr" is the default for back-compatibility, mainly.

inv_z	for exact=TRUE, norm != "2", (an approximation of) solve(z); could be the pseudo inverse or a fast approximate inverse of the matrix z. By default, solve(z) is the most expensive part of the condition computation when exact is true.
triangular	logical. If true, the matrix used is just the upper or lower triangular part of z (or x), depending on
uplo	character string, either "U" or "L". Used only when triangular = TRUE, indicates if the upper or lower triangular part of the matrix is to be used.
LINPACK	logical. If true and z is not complex, the LINPACK routine dtrco() is called; otherwise the relevant LAPACK routine is.
...	further arguments passed to or from other methods; for kappa.*(), notably LINPACK when norm is not "2".

Details

For kappa(), if exact = FALSE (the default) the condition number is estimated by a cheap approximation to the 1-norm of the triangular matrix R of the qr(x) decomposition $z = QR$. However, the exact 2-norm calculation (via svd) is also likely to be quick enough.

Note that the approximate 1- and Inf-norm condition numbers via method = "direct" are much faster to calculate, and rcond() computes these ***reciprocal*** condition numbers, also for complex matrices, using standard LAPACK routines. Currently, also the kappa*() functions compute these approximations whenever exact is false, i.e., by default.

kappa and rcond are different interfaces to *partly* identical functionality.

.kappa_tri is an internal function called by kappa.qr and kappa.default; tri is for *tri*angular and its methods only consider the upper or lower triangular part of the matrix, depending on uplo = "U" or "L", where "U" was internally hard wired before R 4.4.0.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The condition number, $kappa$, or an approximation if exact = FALSE.

Author(s)

The design was inspired by (but differs considerably from) the S function of the same name described in Chambers (1992).

Source

The LAPACK routines DTRCON and ZTRCON and the LINPACK routine DTRCO.

LAPACK and LINPACK are from https://netlib.org/lapack/ and https://netlib.org/linpack/ and their guides are listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM. Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

See Also

norm; svd for the singular value decomposition and qr for the QR one.

Examples

```
kappa(x1 <- cbind(1, 1:10)) # 15.71
kappa(x1, exact = TRUE)     # 13.68
kappa(x2 <- cbind(x1, 2:11)) # high! [x2 is singular!]

hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, `+`) }
sv9 <- svd(h9 <- hilbert(9))$ d
kappa(h9)  # pretty high; by default {exact=FALSE, method="qr"} :
kappa(h9) == kappa(qr.R(qr(h9)), norm = "1")
all.equal(kappa(h9, exact = TRUE), # its definition:
          max(sv9) / min(sv9),
          tolerance = 1e-12) ## the same (typically down to 2.22e-16)
kappa(h9, exact = TRUE) / kappa(h9)  # 0.677 (i.e., rel.error = 32%)

## Exact kappa for rectangular matrix
## panmagic.6npm1(7) :
pm7 <- rbind(c( 1, 13, 18, 23, 35, 40, 45),
             c(37, 49,  5, 10, 15, 27, 32),
             c(24, 29, 41, 46,  2, 14, 19),
             c(11, 16, 28, 33, 38, 43,  6),
             c(47,  3,  8, 20, 25, 30, 42),
             c(34, 39, 44,  7, 12, 17, 22),
             c(21, 26, 31, 36, 48,  4,  9))

kappa(pm7, exact=TRUE, norm="1") # no problem for square matrix

m76 <- pm7[,1:6]
(m79 <- cbind(pm7, 50:56, 63:57))

## Moore-Penrose inverse { ~= MASS::ginv(); differing tol (value & meaning)}:
## pinv := p(seudo) inv(erse)
pinv <- function(X, s = svd(X), tol = 64*.Machine$double.eps) {
    if (is.complex(X))
        s$u <- Conj(s$u)
    dx <- dim(X)
    ## X = U D V' ==> Result =  V {1/D} U'
    pI <- function(u,d,v) tcrossprod(v, u / rep(d, each = dx[1L]))
    pos <- (d <- s$d) > max(tol * max(dx) * d[1L], 0)
```

```
    if (all(pos))
        pI(s$u, d, s$v)
    else if (!any(pos))
        array(0, dX[2L:1L])
    else { # some pos, some not:
        i <- which(pos)
        pI(s$u[, i, drop = FALSE], d[i],
           s$v[, i, drop = FALSE])
    }
}

## rectangular
kappa(m76, norm="1")
try( kappa(m76, exact=TRUE, norm="1") )# error in  solve().. must be square

## ==> use pseudo-inverse instead of solve() for rectangular {and norm != "2"}:
iZ <- pinv(m76)
kappa(m76, exact=TRUE, norm="1", inv_z = iZ)
kappa(m76, exact=TRUE, norm="M", inv_z = iZ)
kappa(m76, exact=TRUE, norm="I", inv_z = iZ)

iX <- pinv(m79)
kappa(m79, exact=TRUE, norm="1", inv_z = iX)
kappa(m79, exact=TRUE, norm="M", inv_z = iX)
kappa(m79, exact=TRUE, norm="I", inv_z = iX)
```

`kronecker` *Kronecker Products on Arrays*

Description

Computes the generalised Kronecker product of two arrays, `X` and `Y`.

Usage

```
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)
X %x% Y
```

Arguments

`X`	a vector or array.
`Y`	a vector or array.
`FUN`	a function; it may be a quoted string.
`make.dimnames`	logical: provide dimnames that are the product of the dimnames of `X` and `Y`.
`...`	optional arguments to be passed to `FUN`.

Details

If X and Y do not have the same number of dimensions, the smaller array is padded with dimensions of size one. The returned array comprises submatrices constructed by taking X one term at a time and expanding that term as FUN(x, Y, ...).

%x% is an alias for kronecker (where FUN is hardwired to "*").

Value

An array A with dimensions dim(X) * dim(Y).

Author(s)

Jonathan Rougier

References

Shayle R. Searle (1982) *Matrix Algebra Useful for Statistics.* John Wiley and Sons.

See Also

outer, on which kronecker is built and %*% for usual matrix multiplication.

Examples

```
# simple scalar multiplication
( M <- matrix(1:6, ncol = 2) )
kronecker(4, M)
# Block diagonal matrix:
kronecker(diag(1, 3), M)

# ask for dimnames

fred <- matrix(1:12, 3, 4, dimnames = list(LETTERS[1:3], LETTERS[4:7]))
bill <- c("happy" = 100, "sad" = 1000)
kronecker(fred, bill, make.dimnames = TRUE)

bill <- outer(bill, c("cat" = 3, "dog" = 4))
kronecker(fred, bill, make.dimnames = TRUE)
```

---

l10n_info *Localization Information*

Description

Report on localization information.

Usage

```
l10n_info()
```

Details

'A Latin-1 locale' includes supersets (for printable characters) such as Windows codepage 1252 but not Latin-9 (ISO 8859-15).

On **Windows** (where the resulting list contains `codepage` and `system.codepage` components additionally), common codepages are 1252 (Western European), 1250 (Central European), 1251 (Cyrillic), 1253 (Greek), 1254 (Turkish), 1255 (Hebrew), 1256 (Arabic), 1257 (Baltic), 1258 (Vietnamese), 874 (Thai), 932 (Japanese), 936 (Simplified Chinese), 949 (Korean) and 950 (Traditional Chinese). Codepage 28605 is Latin-9 and 65001 is UTF-8 (where supported). R does not allow the C locale, and uses 1252 as the default codepage.

Value

A list with three logical elements and further OS-specific elements:

`MBCS`	If a multi-byte character set in use?
`UTF-8`	Is this known to be a UTF-8 locale?
`Latin-1`	Is this known to be a Latin-1 locale?

Not on Windows:

`codeset`	character. The encoding name as reported by the OS, possibly `""`. (Added in R 4.1.0. Encoding names are OS-specific.)

Only on Windows:

`codepage`	integer: the Windows codepage corresponding to the locale R is using (and not necessarily that Windows is using).
`system.codepage`	integer: the Windows system/ANSI codepage (the codepage Windows is using). Added in R 4.1.0.

See Also

`Sys.getlocale`, `localeconv`

Examples

```
l10n_info()
```

labels *Find Labels from Object*

Description

Find a suitable set of labels from an object for use in printing or plotting, for example. A generic function.

Usage

```
labels(object, ...)
```

Arguments

object	any R object: the function is generic.
...	further arguments passed to or from other methods.

Value

A character vector or list of such vectors. For a vector the results is the names or `seq_along(x)` and for a data frame or array it is the dimnames (with `NULL` expanded to `seq_len(d[i])`).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

lapply *Apply a Function over a List or Vector*

Description

`lapply` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply` is a user-friendly version and wrapper of `lapply` by default returning a vector, matrix or, if `simplify = "array"`, an array if appropriate, by applying `simplify2array()`. `sapply(x, f, simplify = FALSE, USE.NAMES = FALSE)` is the same as `lapply(x, f)`.

`vapply` is similar to `sapply`, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

`replicate` is a wrapper for the common use of `sapply` for repeated evaluation of an expression (which will usually involve random number generation).

`simplify2array()` is the utility called from `sapply()` when `simplify` is not false and is similarly called from `mapply()`.

Usage

```
lapply(X, FUN, ...)

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)

replicate(n, expr, simplify = "array")

simplify2array(x, higher = TRUE, except = c(0L, 1L))
```

Arguments

X	a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by base::as.list.
FUN	the function to be applied to each element of X: see 'Details'. In the case of functions like +, %*%, the function name must be backquoted or quoted.
...	optional arguments to FUN.
simplify	logical or character string; should the result be simplified to a vector, matrix or higher dimensional array if possible? For sapply it must be named and not abbreviated. The default value, TRUE, returns a vector or matrix if appropriate, whereas if simplify = "array" the result may be an array of "rank" (=length(dim(.))) one higher than the result of FUN(X[[i]]).
USE.NAMES	logical; if TRUE and if X is character, use X as names for the result unless it had names already. Since this argument follows ... its name cannot be abbreviated.
FUN.VALUE	a (generalized) vector; a template for the return value from FUN. See 'Details'.
n	integer: the number of replications.
expr	the expression (a language object, usually a call) to evaluate repeatedly.
x	a list, typically returned from lapply().
higher	logical; if true, simplify2array() will produce a ("higher rank") array when appropriate, whereas higher = FALSE would return a matrix (or vector) only. These two cases correspond to sapply(*, simplify = "array") or simplify = TRUE, respectively.
except	integer vector or NULL; the default c(0L, 1L) corresponds to the exceptions used by sapply: a list with elements of common length 0 or 1 is not simplified to an array but is returned, respectively, as is or unlisted. These exceptions can be disabled by specifying only a subset of 0:1, or NULL to always simplify to an array (if possible).

Details

FUN is found by a call to match.fun and typically is specified as a function or a symbol (e.g., a backquoted name) or a character string specifying a function to be searched for from the environment of the call to lapply.

Function FUN must be able to accept as input any of the elements of X. If the latter is an atomic vector, FUN will always be passed a length-one vector of the same type as X.

Arguments in ... cannot have the same name as any of the other arguments, and care may be needed to avoid partial matching to FUN. In general-purpose code it is good practice to name the first two arguments X and FUN if ... is passed through: this both avoids partial matching to FUN and ensures that a sensible error message is given if arguments named X or FUN are passed through ....

Simplification in sapply is only attempted if X has length greater than zero and if the return values from all elements of X are all of the same (positive) length. If the common length is one the result is a vector, and if greater than one is a matrix with a column corresponding to each element of X.

Simplification is always done in vapply. This function checks that all values of FUN are compatible with the FUN.VALUE, in that they must have the same length and type. (Types may be promoted to a higher type within the ordering logical < integer < double < complex, but not demoted.)

Users of S4 classes should pass a list to lapply and vapply: the internal coercion is done by the as.list in the base namespace and not one defined by a user (e.g., by setting S4 methods on the base function).

Value

For lapply, sapply(simplify = FALSE) and replicate(simplify = FALSE), a list.

For sapply(simplify = TRUE) and replicate(simplify = TRUE): if X has length zero or n = 0, an empty list. Otherwise an atomic vector or matrix or list of the same length as X (of length n for replicate). If simplification occurs, the output type is determined from the highest type of the return values in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression, after coercion of pairlists to lists.

vapply returns a vector or array of type matching the FUN.VALUE. If length(FUN.VALUE) == 1 a vector of the same length as X is returned, otherwise an array. If FUN.VALUE is not an array, the result is a matrix with length(FUN.VALUE) rows and length(X) columns, otherwise an array a with dim(a) == c(dim(FUN.VALUE), length(X)).

The (Dim)names of the array value are taken from the FUN.VALUE if it is named, otherwise from the result of the first function call. Column names of the matrix or more generally the names of the last dimension of the array value or names of the vector value are set from X as in sapply.

Note

sapply(*, simplify = FALSE, USE.NAMES = FALSE) is equivalent to lapply(*).

For historical reasons, the calls created by lapply are unevaluated, and code has been written (e.g., bquote) that relies on this. This means that the recorded call is always of the form FUN(X[[i]], ...), with i replaced by the current (integer or double) index. This is not normally a problem, but it can be if FUN uses sys.call or match.call or if it is a primitive function that makes use of the call. This means that it is often safer to call primitive functions with a wrapper, so that e.g. lapply(ll, function(x) is.numeric(x)) is required to ensure that method dispatch for is.numeric occurs correctly.

If expr is a function call, be aware of assumptions about where it is evaluated, and in particular what ... might refer to. You can pass additional named arguments to a function call as additional named arguments to replicate: see 'Examples'.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`apply`, `tapply`, `mapply` for applying a function to **m**ultiple arguments, and `rapply` for a **r**ecursive version of `lapply()`, `eapply` for applying a function to each entry in an `environment`.

Examples

```
require(stats); require(graphics)

x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
lapply(x, mean)
# median and quartiles for each list element
lapply(x, quantile, probs = 1:3/4)
sapply(x, quantile)
i39 <- sapply(3:9, seq) # list of vectors
sapply(i39, fivenum)
vapply(i39, fivenum,
       c(Min. = 0, "1st Qu." = 0, Median = 0, "3rd Qu." = 0, Max. = 0))

## sapply(*, "array") -- artificial example
(v <- structure(10*(5:8), names = LETTERS[1:4]))
f2 <- function(x, y) outer(rep(x, length.out = 3), y)
(a2 <- sapply(v, f2, y = 2*(1:5), simplify = "array"))
a.2 <- vapply(v, f2, outer(1:3, 1:5), y = 2*(1:5))
stopifnot(dim(a2) == c(3,5,4), all.equal(a2, a.2),
          identical(dimnames(a2), list(NULL,NULL,LETTERS[1:4])))

hist(replicate(100, mean(rexp(10))))

## use of replicate() with parameters:
foo <- function(x = 1, y = 2) c(x, y)
# does not work: bar <- function(n, ...) replicate(n, foo(...))
bar <- function(n, x) replicate(n, foo(x = x))
bar(5, x = 3)
```

`Last.value` *Value of Last Evaluated Expression*

Description

The value of the internal evaluation of a top-level R expression is always assigned to `.Last.value` (in `package:base`) before further processing (e.g., printing).

Usage

```
.Last.value
```

Details

The value of a top-level assignment *is* put in `.Last.value`, unlike S.

Do not assign to `.Last.value` in the workspace, because this will always mask the object of the same name in `package:base`.

See Also

`eval`

Examples

```
## These will not work correctly from example(),
## but they will in make check or if pasted in,
## as example() does not run them at the top level
gamma(1:15)          # think of some intensive calculation...
fac14 <- .Last.value # keep them

library("splines") # returns invisibly
.Last.value    # shows what library(.) above returned
```

La_library *LAPACK Library*

Description

Report the name of the shared object file with LAPACK implementation in use.

Usage

```
La_library()
```

Value

A character vector of length one (`""` when the name is not known). The value can be used as an indication of which LAPACK implementation is in use. Typically, the R version of LAPACK will appear as `libRlapack.so` (`libRlapack.dylib`), depending on how R was built. Note that `libRlapack.so` (`libRlapack.dylib`) may also be shown for an external LAPACK implementation that had been copied, hard-linked or renamed by the system administrator. Otherwise, the shared object file will be given and its path/name may indicate the vendor/version.

The detection does not work on Windows, nor for the Accelerate framework on macOS, nor in the rare (and unsupported) case of a static external library.

It is possible to build R against an enhanced BLAS which contains some but not all LAPACK routines, in which case this function reports the library containing routine `ILAVER`.

See Also

extSoftVersion for versions of other third-party software including BLAS.

La_version for the version of LAPACK in use.

Examples

```
La_library()
```

La_version *LAPACK Version*

Description

Report the version of LAPACK in use.

Usage

```
La_version()
```

Value

A character vector of length one.

Note that this is the version as reported by the library at runtime. It may differ from the reference ('netlib') implementation, for example by having some optimized or patched routines. For the version included with R, the older (not Fortran 90) versions of

```
DLARTG DLASSQ ZLARTG ZLASSQ
```

are used.

See Also

extSoftVersion for versions of other third-party software.

La_library for binary/executable file with LAPACK in use.

Examples

```
La_version()
```

length *Length of an Object*

Description

Get or set the length of vectors (including lists) and factors, and of any other R object for which a method has been defined.

Usage

```
length(x)
length(x) <- value
```

Arguments

x	an R object. For replacement, a vector or factor.
value	a non-negative integer or double (which will be rounded down).

Details

Both functions are generic: you can write methods to handle specific classes of objects, see InternalMethods. `length<-` has a `"factor"` method.

The replacement form can be used to reset the length of a vector. If a vector is shortened, extra values are discarded and when a vector is lengthened, it is padded out to its new length with NAs (`nul` for raw vectors).

Both are primitive functions.

Value

The default method for `length` currently returns a non-negative `integer` of length 1, except for vectors of more than $2^{31} - 1$ elements, when it returns a double.

For vectors (including lists) and factors the length is the number of elements. For an environment it is the number of objects in the environment, and `NULL` has length 0. For expressions and pairlists (including language objects and dot-dot-dot lists) it is the length of the pairlist chain. All other objects (including functions) have length one: note that for functions this differs from S.

The replacement form removes all the attributes of `x` except its names, which are adjusted (and if necessary extended by `""`).

Warning

Package authors have written methods that return a result of length other than one (**Formula**) and that return a vector of type `double` (**Matrix**), even with non-integer values (earlier versions of **sets**). Where a single double value is returned that can be represented as an integer it is returned as a length-one integer vector.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`nchar` for counting the number of characters in character vectors, `lengths` for getting the length of every element in a list.

Examples

```
length(diag(4))  # = 16 (4 x 4)
length(options())  # 12 or more
length(y ~ x1 + x2 + x3)  # 3
length(expression(x, {y <- x^2; y+2}, x^y))  # 3

## from example(warpbreaks)
require(stats)

fm1 <- lm(breaks ~ wool * tension, data = warpbreaks)
length(fm1$call)      # 3, lm() and two arguments.
length(formula(fm1))  # 3, ~ lhs rhs
```

`lengths` *Lengths of List or Vector Elements*

Description

Get the length of each element of a `list` or atomic vector (`is.atomic`) as an integer or numeric vector.

Usage

```
lengths(x, use.names = TRUE)
```

Arguments

`x`	a `list`, list-like such as an `expression`, `NULL` or an atomic vector (for which the result is trivial).
`use.names`	logical indicating if the result should inherit the `names` from `x`.

Details

This function loops over `x` and returns a compatible vector containing the length of each element in `x`. Effectively, `length(x[[i]])` is called for all `i`, so any methods on `length` are considered.

`lengths` is generic: you can write methods to handle specific classes of objects, see InternalMethods.

Value

A non-negative integer of length length(x), except when any element has a length of more than $2^{31} - 1$ elements, when it returns a double vector. When use.names is true, the names are taken from the names on x, if any.

Note

One raison d'être of lengths(x) is its use as a more efficient version of sapply(x, length) and similar *apply calls to length. This is the reason why x may be an atomic vector, even though lengths(x) is trivial in that case.

See Also

length for getting the length of any R object.

Examples

```
require(stats)
## summarize by month
l <- split(airquality$Ozone, airquality$Month)
avgOz <- lapply(l, mean, na.rm=TRUE)
## merge result
airquality$avgOz <- rep(unlist(avgOz, use.names=FALSE), lengths(l))
## but this is safer and cleaner, but can be slower
airquality$avgOz <- unsplit(avgOz, airquality$Month)

## should always be true, except when a length does not fit in 32 bits
stopifnot(identical(lengths(l), vapply(l, length, integer(1L))))

## empty lists are not a problem
x <- list()
stopifnot(identical(lengths(x), integer()))

## nor are "list-like" expressions:
lengths(expression(u, v, 1+ 0:9))

## and we should dispatch to length methods
f <- c(rep(1, 3), rep(2, 6), 3)
dates <- split(as.POSIXlt(Sys.time() + 1:10), f)
stopifnot(identical(lengths(dates), vapply(dates, length, integer(1L))))
```

---

levels *Levels Attributes*

Description

levels provides access to the levels attribute of a variable. The first form returns the value of the levels of its argument and the second sets the attribute.

Usage

```
levels(x)
levels(x) <- value
```

Arguments

x	an object, for example a factor.
value	a valid value for `levels(x)`. For the default method, `NULL` or a character vector. For the `factor` method, a vector of character strings with length at least the number of levels of `x`, or a named list specifying how to rename the levels.

Details

Both the extractor and replacement forms are generic and new methods can be written for them. The most important method for the replacement function is that for `factor`s.

For the factor replacement method, a `NA` in `value` causes that level to be removed from the levels and the elements formerly with that level to be replaced by `NA`.

Note that for a factor, replacing the levels via `levels(x) <- value` is not the same as (and is preferred to) `attr(x, "levels") <- value`.

The replacement function is primitive.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`nlevels`, `relevel`, `reorder`.

Examples

```
## assign individual levels
x <- gl(2, 4, 8)
levels(x)[1] <- "low"
levels(x)[2] <- "high"
x

## or as a group
y <- gl(2, 4, 8)
levels(y) <- c("low", "high")
y

## combine some levels
z <- gl(3, 2, 12, labels = c("apple", "salad", "orange"))
z
levels(z) <- c("fruit", "veg", "fruit")
z
```

```
## same, using a named list
z <- gl(3, 2, 12, labels = c("apple", "salad", "orange"))
z
levels(z) <- list("fruit" = c("apple","orange"),
                  "veg"   = "salad")
z

## we can add levels this way:
f <- factor(c("a","b"))
levels(f) <- c("c", "a", "b")
f

f <- factor(c("a","b"))
levels(f) <- list(C = "C", A = "a", B = "b")
f
```

`libcurlVersion` *Report Version of libcurl*

Description

Report version of `libcurl` in use.

Usage

```
libcurlVersion()
```

Value

A character string, with value the `libcurl` version in use or `""` if none is. If `libcurl` is available, has attributes

`ssl_version`	A character string naming the SSL/TLS implementation and version, possibly `"none"`. It is intended for the version of OpenSSL used, but not all implementations of `libcurl` use OpenSSL — for example macOS reports `"SecureTranspart"`, its wrapper for SSL/TLS.
`libssh_version`	A character string naming the `libssh` version, which may or may not be available (it is used for e.g. `scp` and `sftp` protocols). Where present, something like `"libssh2/1.5.0"`.
`protocols`	A character vector of the names of supported protocols, also known as 'schemes' when part of a URL.

Warning

In late 2017 a `libcurl` installation was seen divided into two libraries, `libcurl` and `libcurl-feature`, and the first had been updated but not the second. As the compiled function recording the version was in the latter, the version reported by `libcurlVersion` was misleading.

See Also

extSoftVersion for versions of other third-party software.

curlGetHeaders, download.file and url for functions which (optionally) use libcurl.

https://curl.se/docs/sslcerts.html and https://curl.se/docs/ssl-compared.html for more details on SSL versions (the current standard being known as TLS). Normally libcurl used with R uses SecureTransport on macOS, OpenSSL on Windows and GnuTLS, NSS or OpenSSL on Unix-alikes. (At the time of writing Debian-based Linuxen use GnuTLS and RedHat-based ones use OpenSSL, having previously used NSS.)

Examples

```
libcurlVersion()
```

---

libPaths *Search Paths for Packages*

Description

.libPaths gets/sets the library trees within which packages are looked for.

Usage

```
.libPaths(new, include.site = TRUE)

.Library
.Library.site
```

Arguments

new	a character vector with the locations of R library trees. Tilde expansion (path.expand) is done, and if any element contains one of *?[, globbing is done where supported by the platform: see Sys.glob.
include.site	a logical value indicating whether the value of .Library.site should be included in the new set of library tree locations. Defaulting to TRUE, it is ignored when .libPaths is called without the new argument.

Details

.Library is a character string giving the location of the default library, the 'library' subdirectory of R_HOME.

.Library.site is a (possibly empty) character vector giving the locations of the site libraries.

.libPaths is used for getting or setting the library trees that R knows about and hence uses when looking for packages (the library search path). If called with argument new, by default, the library search path is set to the existing directories in unique(c(new, .Library.site, .Library)) and this is returned. If include.site is FALSE when the new argument is set, .Library.site is not

added to the new library search path. If called without the new argument, a character vector with the currently active library trees is returned.

How paths in new with a trailing slash are treated is OS-dependent. On a POSIX filesystem existing directories can usually be specified with a trailing slash. On Windows filepaths with a trailing slash (or backslash) are invalid and existing directories specified with a trailing slash may not be added to the library search path.

At startup, the library search path is initialized from the environment variables R_LIBS, R_LIBS_USER and R_LIBS_SITE, which if set should give lists of directories where R library trees are rooted, colon-separated on Unix-alike systems and semicolon-separated on Windows. For the latter two, a value of NULL indicates an empty list of directories. (Note that as from R 4.2.0, both are set by R start-up code if not already set or empty so can be interrogated from an R session to find their defaults: in earlier versions this was true only for R_LIBS_USER.)

First, .Library.site is initialized from R_LIBS_SITE. If this is unset or empty, the 'site-library' subdirectory of R_HOME is used. Only directories which exist at the time of initialization are retained. Then, .libPaths() is called with the combination of the directories given by R_LIBS and R_LIBS_USER. By default R_LIBS is unset, and if R_LIBS_USER is unset or empty, it is set to directory '`R/`*R.version$platform*`-library/`*x.y*' of the home directory on Unix-alike systems (or '`Library/R/`*m*`/`*x.y*`/library`' for CRAN macOS builds, with *m* `Sys.info()["machine"]`) and '`R/win-library/`*x.y*' subdirectory of LOCALAPPDATA on Windows, for R *x.y.z*.

Both R_LIBS_USER and R_LIBS_SITE feature possible expansion of specifiers for R-version-specific information as part of the startup process. The possible conversion specifiers all start with a '%' and are followed by a single letter (use '%%' to obtain '%'), with currently available conversion specifications as follows:

'%V' R version number including the patch level (e.g., '2.5.0').

'%v' R version number excluding the patch level (e.g., '2.5').

'%p' the platform for which R was built, the value of R.version$platform.

'%o' the underlying operating system, the value of R.version$os.

'%a' the architecture (CPU) R was built on/for, the value of R.version$arch.

(See version for details on R version information.) In addition, '%U' and '%S' expand to the R defaults for, respectively, R_LIBS_USER and R_LIBS_SITE.

Function .libPaths always uses the values of .Library and .Library.site in the base namespace. .Library.site can be set by the site in 'Rprofile.site', which should be followed by a call to .libPaths(.libPaths()) to make use of the updated value.

For consistency, the paths are always normalized by normalizePath(winslash = "/").

LOCALAPPDATA (usually C:\Users\username\AppData\Local) on Windows is a hidden directory and may not be viewed by some software. It may be opened by shell.exec(Sys.getenv("LOCALAPPDATA")).

Value

A character vector of file paths.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`library`

Examples

```
.libPaths()               # all library trees R knows about
```

`library` *Loading/Attaching and Listing of Packages*

Description

`library` and `require` load and attach add-on packages.

Usage

```
library(package, help, pos = 2, lib.loc = NULL,
        character.only = FALSE, logical.return = FALSE,
        warn.conflicts, quietly = FALSE,
        verbose = getOption("verbose"),
        mask.ok, exclude, include.only,
        attach.required = missing(include.only))

require(package, lib.loc = NULL, quietly = FALSE,
        warn.conflicts,
        character.only = FALSE,
        mask.ok, exclude, include.only,
        attach.required = missing(include.only))

conflictRules(pkg, mask.ok = NULL, exclude = NULL)
```

Arguments

`package, help`	the name of a package, given as a name or literal character string, or a character string, depending on whether `character.only` is `FALSE` (default) or `TRUE`.
`pos`	the position on the search list at which to attach the loaded namespace. Can also be the name of a position on the current search list as given by `search()`.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known to `.libPaths()`. Non-existent library trees are silently ignored.

`character.only` a logical indicating whether `package` or `help` can be assumed to be character strings.

`logical.return` logical. If it is `TRUE`, `FALSE` or `TRUE` is returned to indicate success.

`warn.conflicts` logical. If `TRUE`, warnings are printed about `conflicts` from attaching the new package. A conflict is a function masking a function, or a non-function masking a non-function. The default is `TRUE` unless specified as `FALSE` in the `conflicts.policy` option.

`verbose` a logical. If `TRUE`, additional diagnostics are printed.

`quietly` a logical. If `TRUE`, no message confirming package attaching is printed, and most often, no errors/warnings are printed if package attaching fails.

`pkg` character string naming a package.

`mask.ok` character vector of names of objects that can mask objects on the search path without signaling an error when strict conflict checking is enabled.

`exclude, include.only`
character vector of names of objects to exclude or include in the attached frame. Only one of these arguments may be used in a call to `library` or `require`.

`attach.required`
logical specifying whether required packages listed in the `Depends` clause of the `DESCRIPTION` file should be attached automatically.

Details

`library(package)` and `require(package)` both load the namespace of the package with name `package` and attach it on the search list. `require` is designed for use inside other functions; it returns `FALSE` and gives a warning (rather than an error as `library()` does by default) if the package does not exist. Both functions check and update the list of currently attached packages and do not reload a namespace which is already loaded. (If you want to reload such a package, call `detach(unload = TRUE)` or `unloadNamespace` first.) If you want to load a package without attaching it on the search list, see `requireNamespace`.

To suppress messages during the loading of packages use `suppressPackageStartupMessages`: this will suppress all messages from R itself but not necessarily all those from package authors.

If `library` is called with no `package` or `help` argument, it lists all available packages in the libraries specified by `lib.loc`, and returns the corresponding information in an object of class `"libraryIQR"`. (The structure of this class may change in future versions.) Use `.packages(all = TRUE)` to obtain just the names of all available packages, and `installed.packages()` for even more information.

`library(help = somename)` computes basic information about the package **somename**, and returns this in an object of class `"packageInfo"`. (The structure of this class may change in future versions.) When used with the default value (`NULL`) for `lib.loc`, the attached packages are searched before the libraries.

Value

Normally `library` returns (invisibly) the list of attached packages, but `TRUE` or `FALSE` if `logical.return` is `TRUE`. When called as `library()` it returns an object of class `"libraryIQR"`, and for `library(help=)`, one of class `"packageInfo"`.

`require` returns (invisibly) a logical indicating whether the required package is available.

Conflicts

Handling of conflicts depends on the setting of the `conflicts.policy` option. If this option is not set, then conflicts result in warning messages if the argument `warn.conflicts` is `TRUE`. If the option is set to the character string `"strict"`, then all unresolved conflicts signal errors. Conflicts can be resolved using the `mask.ok`, `exclude`, and `include.only` arguments to `library` and `require`. Defaults for `mask.ok` and `exclude` can be specified using `conflictRules`.

If the `conflicts.policy` option is set to the string `"depends.ok"` then conflicts resulting from attaching declared dependencies will not produce errors, but other conflicts will. This is likely to be the best setting for most users wanting some additional protection against unexpected conflicts.

The policy can be tuned further by specifying the `conflicts.policy` option as a named list with the following fields:

`error`**:** logical; if `TRUE` treat unresolved conflicts as errors.

`warn`**:** logical; unless `FALSE` issue a warning message when conflicts are found.

`generics.ok`**:** logical; if `TRUE` ignore conflicts created by defining S4 generics for functions on the search path.

`depends.ok`**:** logical; if `TRUE` do not treat conflicts with required packages as errors.

`can.mask`**:** character vector of names of packages that are allowed to be masked. These would typically be base packages attached by default.

Licenses

Some packages have restrictive licenses, and there is a mechanism to allow users to be aware of such licenses. If `getOption("checkPackageLicense") == TRUE`, then at first use of a namespace of a package with a not-known-to-be-FOSS (see below) license the user is asked to view and accept the license: a list of accepted licenses is stored in file '`~/.R/licensed`'. In a non-interactive session it is an error to use such a package whose license has not already been recorded as accepted.

Free or Open Source Software (FOSS, e.g. https://en.wikipedia.org/wiki/FOSS) packages are determined by the same filters used by `available.packages` but applied to just the current package, not its dependencies.

There can also be a site-wide file '`R_HOME/etc/licensed.site`' of packages (one per line).

Formal methods

`library` takes some further actions when package **methods** is attached (as it is by default). Packages may define formal generic functions as well as re-defining functions in other packages (notably **base**) to be generic, and this information is cached whenever such a namespace is loaded after **methods** and re-defined functions (implicit generics) are excluded from the list of conflicts. The caching and check for conflicts require looking for a pattern of objects; the search may be avoided by defining an object `.noGenerics` (with any value) in the namespace. Naturally, if the package *does* have any such methods, this will prevent them from being used.

Note

`library` and `require` can only load/attach an *installed* package, and this is detected by having a '`DESCRIPTION`' file containing a '`Built:`' field.

Under Unix-alikes, the code checks that the package was installed under a similar operating system as given by `R.version$platform` (the canonical name of the platform under which R was compiled), provided it contains compiled code. Packages which do not contain compiled code can be shared between Unix-alikes, but not to other OSes because of potential problems with line endings and OS-specific help files. If sub-architectures are used, the OS similarity is not checked since the OS used to build may differ (e.g. `i386-pc-linux-gnu` code can be built on an `x86_64-unknown-linux-gnu` OS).

The package name given to `library` and `require` must match the name given in the package's 'DESCRIPTION' file exactly, even on case-insensitive file systems such as are common on Windows and macOS.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.libPaths`, `.packages`.

`attach`, `detach`, `search`, `objects`, `autoload`, `requireNamespace`, `library.dynam`, `data`, `install.packages` and `installed.packages`; `INSTALL`, `REMOVE`.

The initial set of packages attached is set by `options(defaultPackages=)`: see also `Startup`.

Examples

```
library()                  # list all available packages
library(lib.loc = .Library) # list all packages in the default library
library(help = splines)     # documentation on package 'splines'
library(splines)            # attach package 'splines'
require(splines)            # the same
search()                    # "splines", too
detach("package:splines")

# if the package name is in a character vector, use
pkg <- "splines"
library(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep = ":"), search()))

require(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep = ":"), search()))

require(nonexistent)        # FALSE
## Not run:
## if you want to mask as little as possible, use
library(mypkg, pos = "package:base")

## End(Not run)
```

`library.dynam` *Loading DLLs from Packages*

Description

Load the specified file of compiled code if it has not been loaded already, or unloads it.

Usage

```
library.dynam(chname, package, lib.loc,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext, ...)

library.dynam.unload(chname, libpath,
                     verbose = getOption("verbose"),
                     file.ext = .Platform$dynlib.ext)

.dynLibs(new)
```

Arguments

`chname`	a character string naming a DLL (also known as a dynamic shared object or library) to load.
`package`	a character vector with the name of package.
`lib.loc`	a character vector describing the location of R library trees to search through.
`libpath`	the path to the loaded package whose DLL is to be unloaded.
`verbose`	a logical value indicating whether an announcement is printed on the console before loading the DLL. The default value is taken from the verbose entry in the system `options`.
`file.ext`	the extension (including '.' if used) to append to the file name to specify the library to be loaded. This defaults to the appropriate value for the operating system.
`...`	additional arguments needed by some libraries that are passed to the call to `dyn.load` to control how the library and its dependencies are loaded.
`new`	a list of `"DLLInfo"` objects corresponding to the DLLs loaded by packages. Can be missing.

Details

See `dyn.load` for what sort of objects these functions handle.

`library.dynam` is designed to be used inside a package rather than at the command line, and should really only be used inside `.onLoad`. The system-specific extension for DLLs (e.g., '`.so`' or '`.sl`' on Unix-alike systems, '`.dll`' on Windows) should not be added.

`library.dynam.unload` is designed for use in `.onUnload`: it unloads the DLL and updates the value of `.dynLibs()`

`.dynLibs` is used for getting (with no argument) or setting the DLLs which are currently loaded by packages (using `library.dynam`).

Value

If `chname` is not specified, `library.dynam` returns an object of class `"DLLInfoList"` corresponding to the DLLs loaded by packages.

If `chname` is specified, an object of class `"DLLInfo"` that identifies the DLL and which can be used in future calls is returned invisibly. Note that the class `"DLLInfo"` has a method for `$` which can be used to resolve native symbols within that DLL.

`library.dynam.unload` invisibly returns an object of class `"DLLInfo"` identifying the DLL successfully unloaded.

`.dynLibs` returns an object of class `"DLLInfoList"` corresponding to its current value.

Warning

Do not use `dyn.unload` on a DLL loaded by `library.dynam`: use `library.dynam.unload` to ensure that `.dynLibs` gets updated. Otherwise a subsequent call to `library.dynam` will be told the object is already loaded.

Note that whether or not it is possible to unload a DLL and then reload a revised version of the same file is OS-dependent: see the 'Value' section of the help for `dyn.unload`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`getLoadedDLLs` for information on `"DLLInfo"` and `"DLLInfoList"` objects.

`.onLoad`, `library`, `dyn.load`, `.packages`, `.libPaths`

`SHLIB` for how to create suitable DLLs.

Examples

```
## Which DLLs were dynamically loaded by packages?
library.dynam()

## More on library.dynam.unload() :
require(nlme)
nlme:::.onUnload # shows library.dynam.unload() call
detach("package:nlme")  # by default, unload=FALSE ,  so,
tail(library.dynam(), 2)# nlme still there

## How to unload the DLL ?
## Best is to unload the namespace,  unloadNamespace("nlme")
## If we need to do it separately which should be exceptional:
pd.file <- attr(packageDescription("nlme"), "file")
library.dynam.unload("nlme", libpath = sub("/Meta.*", '', pd.file))
```

```
tail(library.dynam(), 2)# 'nlme' is gone now
unloadNamespace("nlme") # now gives warning
```

`license` *The R License Terms*

Description

The license terms under which R is distributed.

Usage

```
license()
licence()
```

Details

R is distributed under the terms of the GNU GENERAL PUBLIC LICENSE, either Version 2, June 1991 or Version 3, June 2007. A copy of the version 2 license is in file '*R_HOME*`/doc/COPYING`' and can be viewed by `RShowDoc("COPYING")`. Version 3 of the license can be displayed by `RShowDoc("GPL-3")`.

A small number of files (some of the API header files) are distributed under the LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later. A copy of this license is in file '*R_SHARE_DIR*`/licenses/LGPL-2.1`' and can be viewed by `RShowDoc("LGPL-2.1")`. Version 3 of the license can be displayed by `RShowDoc("LGPL-3")`.

`list` *Lists – Generic and Dotted Pairs*

Description

Functions to construct, coerce and check for both kinds of R lists.

Usage

```
list(...)
pairlist(...)

as.list(x, ...)
## S3 method for class 'environment'
as.list(x, all.names = FALSE, sorted = FALSE, ...)
as.pairlist(x)

is.list(x)
is.pairlist(x)

alist(...)
```

Arguments

...	objects, possibly named.
x	object to be coerced or tested.
all.names	a logical indicating whether to copy all values or (default) only those whose names do not begin with a dot.
sorted	a logical indicating whether the names of the resulting list should be sorted (increasingly). Note that this is somewhat costly, but may be useful for comparison of environments.

Details

Almost all lists in R internally are *Generic Vectors*, whereas traditional *dotted pair* lists (as in LISP) remain available but rarely seen by users (except as formals of functions).

The arguments to list or pairlist are of the form value or tag = value. The functions return a list or dotted pair list composed of its arguments with each value either tagged or untagged, depending on how the argument was specified.

alist handles its arguments as if they described function arguments. So the values are not evaluated, and tagged arguments with no value are allowed whereas list simply ignores them. alist is most often used in conjunction with formals.

as.list attempts to coerce its argument to a list. For functions, this returns the concatenation of the list of formal arguments and the function body. For expressions, the list of constituent elements is returned. as.list is generic, and as the default method calls as.vector(mode = "list") for a non-list, methods for as.vector may be invoked. as.list turns a factor into a list of one-element factors, keeping names. Other attributes may be dropped unless the argument already is a list or expression. (This is inconsistent with functions such as as.character which always drop attributes, and is for efficiency since lists can be expensive to copy.)

is.list returns TRUE if and only if its argument is a list *or* a pairlist of length > 0. is.pairlist returns TRUE if and only if the argument is a pairlist or NULL (see below).

The "environment" method for as.list copies the name-value pairs (for names not beginning with a dot) from an environment to a named list. The user can request that all named objects are copied. Unless sorted = TRUE, the list is in no particular order (the order depends on the order of creation of objects and whether the environment is hashed). No enclosing environments are searched. (Objects copied are duplicated so this can be an expensive operation.) Note that there is an inverse operation, the as.environment() method for list objects.

An empty pairlist, pairlist() is the same as NULL. This is different from list(): some but not all operations will promote an empty pairlist to an empty list.

as.pairlist is implemented as as.vector(x, "pairlist"), and hence will dispatch methods for the generic function as.vector. Lists are copied element-by-element into a pairlist and the names of the list used as tags for the pairlist: the return value for other types of argument is undocumented.

list, is.list and is.pairlist are primitive functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

vector("list", length) for creation of a list with empty components; c, for concatenation; formals. unlist is an approximate inverse to as.list().

'plotmath' for the use of list in plot annotation.

Examples

```
require(graphics)

# create a plotting structure
pts <- list(x = cars[,1], y = cars[,2])
plot(pts)

is.pairlist(.Options)  # a user-level pairlist

## "pre-allocate" an empty list of length 5
vector("list", 5)

# Argument lists
f <- function() x
# Note the specification of a "..." argument:
formals(f) <- al <- alist(x = , y = 2+3, ... = )
f
al

## environment->list coercion

e1 <- new.env()
e1$a <- 10
e1$b <- 20
as.list(e1)
```

---

list.files *List the Files in a Directory/Folder*

Description

These functions produce a character vector of the names of files or directories in the named directory.

Usage

```
list.files(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE,
           ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)

       dir(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE,
```

```
            ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)

list.dirs(path = ".", full.names = TRUE, recursive = TRUE)
```

Arguments

path	a character vector of full path names; the default corresponds to the working directory, getwd(). Tilde expansion (see path.expand) is performed. Missing values will be ignored. Elements with a marked encoding will be converted to the native encoding (and if that fails, considered non-existent).
pattern	an optional regular expression. Only file names which match the regular expression will be returned.
all.files	a logical value. If FALSE, only the names of visible files are returned (following Unix-style visibility, that is files whose name does not start with a dot). If TRUE, all file names will be returned.
full.names	a logical value. If TRUE, the directory path is prepended to the file names to give a relative file path. If FALSE, the file names (rather than paths) are returned.
recursive	logical. Should the listing recurse into directories?
ignore.case	logical. Should pattern-matching be case-insensitive?
include.dirs	logical. Should subdirectory names be included in recursive listings? (They always are in non-recursive ones).
no..	logical. Should both "." and ".." be excluded also from non-recursive listings?

Value

A character vector containing the names of the files in the specified directories (empty if there were no files). If a path does not exist or is not a directory or is unreadable it is skipped.

The files are sorted in alphabetical order, on the full path if full.names = TRUE.

list.dirs implicitly has all.files = TRUE, and if recursive = TRUE, the answer includes path itself (provided it is a readable directory).

dir is an alias for list.files.

Note

File naming conventions are platform dependent. The pattern matching works with the case of file names as returned by the OS.

On a POSIX filesystem recursive listings will follow symbolic links to directories.

Author(s)

Ross Ihaka, Brian Ripley

See Also

file.info, file.access and files for many more file handling functions and file.choose for interactive selection.

glob2rx to convert wildcards (as used by system file commands and shells) to regular expressions.

Sys.glob for wildcard expansion on file paths. basename and dirname, useful for splitting paths into non-directory (aka 'filename') and directory parts.

Examples

```
list.files(R.home())
## Only files starting with a-l or r
## Note that a-l is locale-dependent, but using case-insensitive
## matching makes it unambiguous in English locales
dir("../..", pattern = "^[a-lr]", full.names = TRUE, ignore.case = TRUE)

list.dirs(R.home("doc"))
list.dirs(R.home("doc"), full.names = FALSE)
```

---

list2DF *Create Data Frame From List*

Description

Create a data frame from a list of variables.

Usage

```
list2DF(x = list(), nrow = 0)
```

Arguments

x	A list of same-length variables for the data frame.
nrow	An integer giving the desired number of rows for the data frame in case x gives no variables (i.e., has length zero).

Details

Note that all list elements are taken "as is".

Value

A data frame with the given variables.

See Also

data.frame

Examples

```
## Create a data frame holding a list of character vectors and the
## corresponding lengths:
x <- list(character(), "A", c("B", "C"))
n <- lengths(x)
list2DF(list(x = x, n = n))

## Create data frames with no variables and the desired number of rows:
list2DF()
list2DF(nrow = 3L)
```

`list2env` *From A List, Build or Add To an Environment*

Description

From a *named* `list` `x`, create an `environment` containing all list components as objects, or "multi-assign" from `x` into a pre-existing environment.

Usage

```
list2env(x, envir = NULL, parent = parent.frame(),
         hash = (length(x) > 100), size = max(29L, length(x)))
```

Arguments

`x`	a `list`, where `names(x)` must not contain empty (`""`) elements.
`envir`	an `environment` or `NULL`.
`parent`	(for the case `envir = NULL`): a parent frame aka enclosing environment, see `new.env`.
`hash`	(for the case `envir = NULL`): logical indicating if the created environment should use hashing, see `new.env`.
`size`	(in the case `envir = NULL, hash = TRUE`): hash size, see `new.env`.

Details

This will be very slow for large inputs unless hashing is used on the environment.

Environments must have uniquely named entries, but named lists need not: where the list has duplicate names it is the *last* element with the name that is used. Empty names throw an error.

Value

An `environment`, either newly created (as by `new.env`) if the `envir` argument was `NULL`, otherwise the updated environment `envir`. Since environments are never duplicated, the argument `envir` is also changed.

Author(s)

Martin Maechler

See Also

environment, new.env, as.environment; further, assign.

The (semantical) "inverse": as.list.environment.

Examples

```
L <- list(a = 1, b = 2:4, p = pi, ff = gl(3, 4, labels = LETTERS[1:3]))
e <- list2env(L)
ls(e)
stopifnot(ls(e) == sort(names(L)),
          identical(L$b, e$b)) # "$" working for environments as for lists

## consistency, when we do the inverse:
ll <- as.list(e)  # -> dispatching to the as.list.environment() method
rbind(names(L), names(ll)) # not in the same order, typically,
                           # but the same content:
stopifnot(identical(L [sort.list(names(L ))],
                    ll[sort.list(names(ll))]))

## now add to e -- can be seen as a fast "multi-assign":
list2env(list(abc = LETTERS, note = "just an example",
              df = data.frame(x = rnorm(20), y = rbinom(20, 1, prob = 0.2))),
         envir = e)
utils::ls.str(e)
```

---

load *Reload Saved Datasets*

Description

Reload datasets written with the function save.

Usage

```
load(file, envir = parent.frame(), verbose = FALSE)
```

Arguments

file	a (readable binary-mode) connection or a character string giving the name of the file to load (when tilde expansion is done).
envir	the environment where the data should be loaded.
verbose	should item names be printed during loading?

Details

load can load R objects saved in the current or any earlier format. It can read a compressed file (see save) directly from a file or from a suitable connection (including a call to url).

A not-open connection will be opened in mode "rb" and closed after use. Any connection other than a gzfile or gzcon connection will be wrapped in gzcon to allow compressed saves to be handled: note that this leaves the connection in an altered state (in particular, binary-only), and that it needs to be closed explicitly (it will not be garbage-collected).

Only R objects saved in the current format (used since R 1.4.0) can be read from a connection. If no input is available on a connection a warning will be given, but any input not in the current format will result in a error.

Loading from an earlier version will give a warning about the 'magic number': magic numbers 1971:1977 are from R < 0.99.0, and RD[ABX]1 from R 0.99.0 to R 1.3.1. These are all obsolete, and you are strongly recommended to re-save such files in a current format.

The verbose argument is mainly intended for debugging. If it is TRUE, then as objects from the file are loaded, their names will be printed to the console. If verbose is set to an integer value greater than one, additional names corresponding to attributes and other parts of individual objects will also be printed. Larger values will print names to a greater depth.

Objects can be saved with references to namespaces, usually as part of the environment of a function or formula. Such objects can be loaded even if the namespace is not available: it is replaced by a reference to the global environment with a warning. The warning identifies the first object with such a reference (but there may be more than one).

Value

A character vector of the names of objects created, invisibly.

Warning

Saved R objects are binary files, even those saved with ascii = TRUE, so ensure that they are transferred without conversion of end of line markers. load tries to detect such a conversion and gives an informative error message.

load(*file*) replaces all existing objects with the same names in the current environment (typically your workspace, .GlobalEnv) and hence potentially overwrites important data. It is considerably safer to use envir = to load into a different environment, or to attach(*file*) which load()s into a new entry in the search path.

See Also

save, download.file; further attach as wrapper for load().

For other interfaces to the underlying serialization format, see unserialize and readRDS.

Examples

```
## save all data
xx <- pi # to ensure there is some data
save(list = ls(all.names = TRUE), file= "all.rda")
```

```
rm(xx)

## restore the saved values to the current environment
local({
   load("all.rda")
   ls()
})

xx <- exp(1:3)
## restore the saved values to the user's workspace
load("all.rda") ## which is here *equivalent* to
## load("all.rda", .GlobalEnv)
## This however annihilates all objects in .GlobalEnv with the same names !
xx # no longer exp(1:3)
rm(xx)
attach("all.rda") # safer and will warn about masked objects w/ same name in .GlobalEnv
ls(pos = 2)
##  also typically need to cleanup the search path:
detach("file:all.rda")

## clean up (the example):
unlink("all.rda")


## Not run:
con <- url("http://some.where.net/R/data/example.rda")
## print the value to see what objects were created.
print(load(con))
close(con) # url() always opens the connection

## End(Not run)
```

`locales` *Query or Set Aspects of the Locale*

Description

Get details of or set aspects of the locale for the R process.

Usage

```
Sys.getlocale (category = "LC_ALL")
Sys.setlocale (category = "LC_ALL", locale = "")
.LC.categories
```

Arguments

`category`	character string. The following categories should always be supported: `"LC_ALL"`, `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"`, `"LC_NUMERIC"` and `"LC_TIME"`. Some systems (not Windows) will also support `"LC_MESSAGES"`,

"LC_PAPER" and "LC_MEASUREMENT". These category names are available in .LC.categories; even when not supported, Sys.getlocale(.) will return "", e.g., for the "LC_PAPER" example on Windows.

`locale` character string. A valid locale name on the system in use. Normally "" (the default) will pick up the default locale for the system.

Details

The locale describes aspects of the internationalization of a program. Initially most aspects of the locale of R are set to "C" (which is the default for the C language and reflects North-American usage – also known as "POSIX"). R sets "LC_CTYPE" and "LC_COLLATE", which allow the use of a different character set and alphabetic comparisons in that character set (including the use of sort), "LC_MONETARY" (for use by Sys.localeconv) and "LC_TIME" may affect the behaviour of as.POSIXlt and strptime and functions which use them (but not date).

The first seven categories described here are those specified by POSIX. "LC_MESSAGES" will be "C" on systems that do not support message translation, and is not supported on Windows, where you *must* use the LANGUAGE environment variable for message translation, see below and the Sys.setLanguage() utility. Trying to use an unsupported category is an error for Sys.setlocale.

Note that setting category "LC_ALL" sets only categories "LC_COLLATE", "LC_CTYPE", "LC_MONETARY" and "LC_TIME".

Attempts to set an invalid locale are ignored. There may or may not be a warning, depending on the OS.

Attempts to change the character set (by Sys.setlocale("LC_CTYPE",)), if that implies a different character set) during a session may not work and are likely to lead to some confusion.

Note that the LANGUAGE environment variable has precedence over "LC_MESSAGES" in selecting the language for message translation on most R platforms.

On platforms where ICU is used for collation the locale used for collation can be reset by icuSetCollate. Except on Windows, the initial setting is taken from the "LC_COLLATE" category, and it is reset when this is changed by a call to Sys.setlocale.

Value

A character string of length one describing the locale in use (after setting for Sys.setlocale), or an empty character string if the current locale settings are invalid or NULL if locale information is unavailable.

For category = "LC_ALL" the details of the string are system-specific: it might be a single locale name or a set of locale names separated by "/" (macOS) or ";" (Windows, Linux). For portability, it is best to query categories individually: it is not necessarily the case that the result of foo <- Sys.getlocale() can be used in Sys.setlocale("LC_ALL", locale = foo).

Available locales

On most Unix-alikes the POSIX shell command locale -a will list the 'available public' locales. What that means is platform-dependent. On recent Linuxen this may mean 'available to be installed' as on some RPM-based systems the locale data is in separate RPMs. On Debian/Ubuntu the set of available locales is managed by OS-specific facilities such as locale-gen and locale -a lists those currently enabled.

For Windows, Microsoft moves its documentation frequently so a Web search is the best way to find current information. From R 4.2, UCRT locale names should be used. The character set should match the system/ANSI codepage (`l10n_info()$codepage` be the same as `l10n_info()$system.codepage`). Setting it to any other value results in a warning and may cause encoding problems. As from R 4.2 on recent Windows the system codepage is 65001 and one should always use locale names ending with `".UTF-8"` (except for `"C"` and `""`), otherwise Windows may add a different character set.

Warning

Setting `"LC_NUMERIC"` to any value other than `"C"` may cause R to function anomalously, so gives a warning. Input conversions in R itself are unaffected, but the reading and writing of ASCII `save` files will be, as may packages which do their own input/output.

Setting it temporarily on a Unix-alike to produce graphical or text output may work well enough, but `options(OutDec)` is often preferable.

Almost all the output routines used by R itself under Windows ignore the setting of `"LC_NUMERIC"` since they make use of the Trio library which is not internationalized.

Note

Changing the values of locale categories whilst R is running ought to be noticed by the OS services, and usually is but exceptions have been seen (usually in collation services).

Do not use the value of `Sys.getlocale("LC_CTYPE")` to attempt to find the character set – for example UTF-8 locales can have suffix '`.UTF-8`' or '`.utf8`' (more common on Linux than '`UTF-8`') or none (as on macOS) and Latin-9 locales can have suffix '`ISO8859-15`', '`iso885915`', '`iso885915@euro`' or '`ISO8859-15@euro`'. Use `l10n_info` instead.

See Also

`strptime` for uses of `category = "LC_TIME"`. `Sys.localeconv` for details of numerical and monetary representations.

`l10n_info` gives some summary facts about the locale and its encoding (including if it is UTF-8).

The 'R Installation and Administration' manual for background on locales and how to find out locale names on your system.

Examples

```
Sys.getlocale()

## Date-time  related :
Sys.getlocale("LC_TIME") -> olcT
then <- as.POSIXlt("2001-01-01 01:01:01", tz = "UTC")
## Not run:
c(m = months(then), wd = weekdays(then)) # locale specific
Sys.setlocale("LC_TIME", "de")     # Solaris: details are OS-dependent
Sys.setlocale("LC_TIME", "de_DE")  # Many Unix-alikes
Sys.setlocale("LC_TIME", "de_DE.UTF-8")  # Linux, macOS, other Unix-alikes
Sys.setlocale("LC_TIME", "de_DE.utf8")   # some Linux versions
Sys.setlocale("LC_TIME", "German.UTF-8") # Windows
```

```
Sys.getlocale("LC_TIME") # the last one successfully set above
c(m = months(then), wd = weekdays(then)) # in C_TIME locale 'cT' ; typically German

## End(Not run)
Sys.setlocale("LC_TIME", "C")
c(m = months(then), wd = weekdays(then)) # "standard" (still platform specific ?)
Sys.setlocale("LC_TIME", olcT)           # reset to previous

## Other locales
Sys.getlocale("LC_PAPER")          # may or may not be set
.LC.categories # of length 9 on all platforms

## Not run: Sys.setlocale("LC_COLLATE", "C")   # turn off locale-specific sorting,
                                   # usually (but not on all platforms)
Sys.setenv("LANGUAGE" = "es") # set the language for error/warning messages

## End(Not run)
## some nice formatting; should work on most platforms,
          ## macOS does not name the entries.
 sep <- switch(Sys.info()[["sysname"]],
               "Darwin"=, "SunOS" = "/",
               "Linux" =, "Windows" = ";")
 ##' show a "full" Sys.getlocale() nicely:
 showL <- function(loc) {
     sl <- strsplit(strsplit(loc, sep)[[1L]], "=")
     if(all(sapply(sl, length) == 2L))
        setNames(sapply(sl, `[[`, 2L), sapply(sl, `[[`, 1L))
     else
       setNames(as.character(sl), .LC.categories[1+seq_along(sl)])
 }
 print.Dlist(lloc <- showL(Sys.getlocale()))
 ## R-supported ones (but LC_ALL):
 lloc[.LC.categories[-1]]
```

---

log *Logarithms and Exponentials*

Description

log computes logarithms, by default natural logarithms, log10 computes common (i.e., base 10) logarithms, and log2 computes binary (i.e., base 2) logarithms. The general form log(x, base) computes logarithms with base base.

log1p(x) computes $\log(1 + x)$ accurately also for $|x| \ll 1$.

exp computes the exponential function.

expm1(x) computes $\exp(x) - 1$ accurately also for $|x| \ll 1$.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

Arguments

x	a numeric or complex vector.
base	a positive or complex number: the base with respect to which logarithms are computed. Defaults to e=exp(1).

Details

All except logb are generic functions: methods can be defined for them individually or via the Math group generic.

log10 and log2 are only convenience wrappers, but logs to bases 10 and 2 (whether computed *via* log or the wrappers) will be computed more efficiently and accurately where supported by the OS. Methods can be set for them individually (and otherwise methods for log will be used).

logb is a wrapper for log for compatibility with S. If (S3 or S4) methods are set for log they will be dispatched. Do not set S4 methods on logb itself.

All except log are primitive functions.

Value

A vector of the same length as x containing the transformed values. log(0) gives -Inf, and log(x) for negative values of x is NaN. exp(-Inf) is 0.

For complex inputs to the log functions, the value is a complex number with imaginary part in the range $[-\pi, \pi]$: which end of the range is used might be platform-specific.

S4 methods

exp, expm1, log, log10, log2 and log1p are S4 generic and are members of the Math group generic.

Note that this means that the S4 generic for log has a signature with only one argument, x, but that base can be passed to methods (but will not be used for method selection). On the other hand, if you only set a method for the Math group generic then base argument of log will be ignored for your class.

Source

log1p and expm1 may be taken from the operating system, but if not available there then they are based on the Fortran subroutine dlnrel by W. Fullerton of Los Alamos Scientific Laboratory (see https://netlib.org/slatec/fnlib/dlnrel.f) and (for small x) a single Newton step for the solution of log1p(y) = x respectively.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for log, log10 and exp.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for logb.)

See Also

Trig, sqrt, Arithmetic.

Examples

```
log(exp(3))
log10(1e7) # = 7

x <- 10^-(1+2*1:9)
cbind(deparse.level=2, # to get nice column names
      x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

Logic *Logical Operators*

Description

These operators act on raw, logical and number-like vectors.

Usage

```
! x
x & y
x && y
x | y
x || y
xor(x, y)

isTRUE (x)
isFALSE(x)
```

Arguments

x, y	raw, logical or 'number-like' vectors (i.e., of types double (class numeric), integer and complex), or objects for which methods have been written.

Details

`!` indicates logical negation (NOT).

`&` and `&&` indicate logical AND and `|` and `||` indicate logical OR. The shorter forms performs elementwise comparisons in much the same way as arithmetic operators. The longer forms evaluates left to right, proceeding only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in `if` clauses.

Using vectors of more than one element in `&&` or `||` will give an error.

`xor` indicates elementwise exclusive OR.

`isTRUE(x)` is the same as `{ is.logical(x) && length(x) == 1 && !is.na(x) && x }`; `isFALSE()` is defined analogously. Consequently, `if(isTRUE(cond))` may be preferable to `if(cond)` because of `NA`s.
In earlier R versions, `isTRUE <- function(x) identical(x, TRUE)`, had the drawback to be false e.g., for `x <- c(val = TRUE)`.

Numeric and complex vectors will be coerced to logical values, with zero being false and all non-zero values being true. Raw vectors are handled without any coercion for `!`, `&`, `|` and `xor`, with these operators being applied bitwise (so `!` is the 1s-complement).

The operators `!`, `&` and `|` are generic functions: methods can be written for them individually or via the `Ops` (or S4 `Logic`, see below) group generic function. (See `Ops` for how dispatch is computed.)

`NA` is a valid logical object. Where a component of `x` or `y` is `NA`, the result will be `NA` if the outcome is ambiguous. In other words `NA & TRUE` evaluates to `NA`, but `NA & FALSE` evaluates to `FALSE`. See the examples below.

See Syntax for the precedence of these operators: unlike many other languages (including S) the AND and OR operators do not have the same precedence (the AND operators have higher precedence than the OR operators).

Value

For `!`, a logical or raw vector(for raw `x`) of the same length as `x`: names, dims and dimnames are copied from `x`, and all other attributes (including class) if no coercion is done.

For `|`, `&` and `xor` a logical or raw vector. If involving a zero-length vector the result has length zero. Otherwise, the elements of shorter vectors are recycled as necessary (with a `warning` when they are recycled only *fractionally*). The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument, the first if they are of the same length. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. For time series, these operations are allowed only if the series are compatible, when the class and `tsp` attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

For `||`, `&&` and `isTRUE`, a length-one logical vector.

S4 methods

`!`, `&` and `|` are S4 generics, the latter two part of the `Logic` group generic (and hence methods need argument names `e1`, `e2`).

Note

The elementwise operators are sometimes called as functions as e.g. `&`(x, y): see the description of how argument-matching is done in Ops.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

TRUE or logical.

any and all for OR and AND on many scalar arguments.

Syntax for operator precedence.

L %||% R which takes L if it is not NULL, and R otherwise.

bitwAnd for bitwise versions for integer vectors.

Examples

```
y <- 1 + (x <- stats::rpois(50, lambda = 1.5) / 4 - 1)
x[(x > 0) & (x < 1)]    # all x values between 0 and 1
if (any(x == 0) || any(y == 0)) "zero encountered"

## construct truth tables :

x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
outer(x, x, `&`) ## AND table
outer(x, x, `|`) ## OR  table
```

---

logical *Logical Vectors*

Description

Create or test for objects of type "logical", and the basic logical constants.

Usage

```
TRUE
FALSE
T; F

logical(length = 0)
as.logical(x, ...)
is.logical(x)
```

Arguments

`length`	a non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`TRUE` and `FALSE` are reserved words denoting logical constants in the R language, whereas `T` and `F` are global variables whose initial values set to these. All four are `logical(1)` vectors.

`as.logical` is a generic function. Methods should return an object of type `"logical"`.

Logical vectors are coerced to integer vectors in contexts where a numerical value is required, with `TRUE` being mapped to `1L`, `FALSE` to `0L` and `NA` to `NA_integer_`.

Value

`logical` creates a logical vector of the specified length. Each element of the vector is equal to `FALSE`.

`as.logical` attempts to coerce its argument to be of logical type. In numeric and complex vectors, zeros are `FALSE` and non-zero values are `TRUE`. For `factor`s, this uses the `levels` (labels). Like `as.vector` it strips attributes including names. Character strings `c("T", "TRUE", "True", "true")` are regarded as true, `c("F", "FALSE", "False", "false")` as false, and all others as `NA`.

`is.logical` returns `TRUE` or `FALSE` depending on whether its argument is of logical type or not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`NA`, the other logical constant. Logical operators are documented in `Logic`.

Examples

```
## non-zero values are TRUE
as.logical(c(pi,0))
if (length(letters)) cat("26 is TRUE\n")

## logical interpretation of particular strings
charvec <- c("FALSE", "F", "False", "false",    "fAlse", "0",
             "TRUE",  "T", "True",  "true",     "tRue",  "1")
as.logical(charvec)

## factors are converted via their levels, so string conversion is used
as.logical(factor(charvec))
as.logical(factor(c(0,1)))  # "0" and "1" give NA
```

LongVectors *Long Vectors*

Description

Vectors of 2^{31} or more elements were added in R 3.0.0.

Details

Prior to R 3.0.0, all vectors in R were restricted to at most $2^{31} - 1$ elements and could be indexed by integer vectors.

Currently all atomic (raw, logical, integer, numeric, complex, character) vectors, lists and expressions can be much longer on 64-bit platforms: such vectors are referred to as 'long vectors' and have a slightly different internal structure. In theory they can contain up to 2^{52} elements, but address space limits of current CPUs and OSes will be much smaller. Such objects will have a length that is expressed as a double, and can be indexed by double vectors.

Arrays (including matrices) can be based on long vectors provided each of their dimensions is at most $2^{31} - 1$: thus there are no 1-dimensional long arrays.

R code typically only needs minor changes to work with long vectors, maybe only checking that `as.integer` is not used unnecessarily for e.g. lengths. However, compiled code typically needs quite extensive changes. Note that the `.C` and `.Fortran` interfaces do not accept long vectors, so `.Call` (or similar) has to be used.

Because of the storage requirements (a minimum of 64 bytes per character string), character vectors are only going to be usable if they have a small number of distinct elements, and even then factors will be more efficient (4 bytes per element rather than 8). So it is expected that most of the usage of long vectors will be integer vectors (including factors) and numeric vectors.

Matrix algebra

It is now possible to use $m \times n$ matrices with more than 2 billion elements. Whether matrix algebra (including `%*%`, `crossprod`, `svd`, `qr`, `solve` and `eigen`) will actually work is somewhat implementation dependent, including the Fortran compiler used and if an external BLAS or LAPACK is used.

An efficient parallel BLAS implementation will often be important to obtain usable performance. For example on one particular platform `chol` on a 47,000 square matrix took about 5 hours with the internal BLAS, 21 minutes using an optimized BLAS on one core, and 2 minutes using an optimized BLAS on 16 cores.

lower.tri *Lower and Upper Triangular Part of a Matrix*

Description

Returns a matrix of logicals the same size of a given matrix with entries TRUE in the lower or upper triangle.

Usage

```
lower.tri(x, diag = FALSE)
upper.tri(x, diag = FALSE)
```

Arguments

x	a matrix or other R object with length(dim(x)) == 2. For back compatibility reasons, when the above is not fulfilled, as.matrix(x) is called first.
diag	logical. Should the diagonal be included?

See Also

diag, matrix; further row and col on which lower.tri() and upper.tri() are built.

Examples

```
(m2 <- matrix(1:20, 4, 5))
lower.tri(m2)
m2[lower.tri(m2)] <- NA
m2
```

ls *List Objects*

Description

ls and objects return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, ls shows what data sets and functions a user has defined. When invoked with no argument inside a function, ls returns the names of the function's local variables: this is useful in conjunction with browser.

Usage

```
ls(name, pos = -1L, envir = as.environment(pos),
   all.names = FALSE, pattern, sorted = TRUE)
objects(name, pos= -1L, envir = as.environment(pos),
        all.names = FALSE, pattern, sorted = TRUE)
```

Arguments

name	which environment to use in listing the available objects. Defaults to the *current* environment. Although called name for back compatibility, in fact this argument can specify the environment in any form; see the 'Details' section.
pos	an alternative argument to name for specifying the environment as a position in the search list. Mostly there for back compatibility.
envir	an alternative argument to name for specifying the environment. Mostly there for back compatibility.
all.names	a logical value. If TRUE, all object names are returned. If FALSE, names which begin with a '.' are omitted.
pattern	an optional regular expression. Only names matching pattern are returned. glob2rx can be used to convert wildcard patterns to regular expressions.
sorted	logical indicating if the resulting character should be sorted alphabetically. Note that this is part of ls() may take most of the time.

Details

The name argument can specify the environment from which object names are taken in one of several forms: as an integer (the position in the search list); as the character string name of an element in the search list; or as an explicit environment (including using sys.frame to access the currently active function calls). By default, the environment of the call to ls or objects is used. The pos and envir arguments are an alternative way to specify an environment, but are primarily there for back compatibility.

Note that the *order* of strings for sorted = TRUE is locale dependent, see Sys.getlocale. If sorted = FALSE the order is arbitrary, depending if the environment is hashed, the order of insertion of objects, ...

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

glob2rx for converting wildcard patterns to regular expressions.

ls.str for a long listing based on str. apropos (or find) for finding objects in the whole search path; grep for more details on 'regular expressions'; class, methods, etc., for object-oriented programming.

Examples

```
.Ob <- 1
ls(pattern = "O")
ls(pattern= "O", all.names = TRUE)    # also shows ".[foo]"

# shows an empty list because inside myfunc no variables are defined
myfunc <- function() {ls()}
```

```
myfunc()

# define a local variable inside myfunc
myfunc <- function() {y <- 1; ls()}
myfunc()                # shows "y"
```

`make.names` *Make Syntactically Valid Names*

Description

Make syntactically valid names out of character vectors.

Usage

```
make.names(names, unique = FALSE, allow_ = TRUE)
```

Arguments

`names`	character vector to be coerced to syntactically valid names. This is coerced to character if necessary.
`unique`	logical; if `TRUE`, the resulting elements are unique. This may be desired for, e.g., column names.
`allow_`	logical. For compatibility with R prior to 1.9.0.

Details

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as `".2way"` are not valid, and neither are the reserved words.

The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

The character `"X"` is prepended if necessary. All invalid characters are translated to `"."`. A missing value is translated to `"NA"`. Names which match R keywords have a dot appended to them. Duplicated values are altered by `make.unique`.

Value

A character vector of same length as `names` with each changed to a syntactically valid name, in the current locale's encoding.

Warning

Some OSes, notably FreeBSD, report extremely incorrect information about which characters are alphabetic in some locales (typically, all multi-byte locales including UTF-8 locales). However, R provides substitutes on Windows, macOS and AIX.

Note

Prior to R version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. Use `allow_ = FALSE` for back-compatibility.

`allow_ = FALSE` is also useful when creating names for export to applications which do not allow underline in names (such as some DBMSes).

See Also

`make.unique`, `names`, `character`, `data.frame`.

Examples

```
make.names(c("a and b", "a-and-b"), unique = TRUE)
# "a.and.b"  "a.and.b.1"
make.names(c("a and b", "a_and_b"), unique = TRUE)
# "a.and.b"  "a_and_b"
make.names(c("a and b", "a_and_b"), unique = TRUE, allow_ = FALSE)
# "a.and.b"  "a.and.b.1"
make.names(c("", "X"), unique = TRUE)
# "X.1" "X" currently; R up to 3.0.2 gave "X" "X.1"

state.name[make.names(state.name) != state.name] # those 10 with a space
```

---

make.unique *Make Character Strings Unique*

Description

Makes the elements of a character vector unique by appending sequence numbers to duplicates.

Usage

```
make.unique(names, sep = ".")
```

Arguments

`names`	a character vector.
`sep`	a character string used to separate a duplicate name from its sequence number.

Details

The algorithm used by `make.unique` has the property that `make.unique(c(A, B)) == make.unique(c(make.unique(A), B))`.

In other words, you can append one string at a time to a vector, making it unique each time, and get the same result as applying `make.unique` to all of the strings at once.

If character vector `A` is already unique, then `make.unique(c(A, B))` preserves `A`.

Value

A character vector of same length as `names` with duplicates changed, in the current locale's encoding.

Author(s)

Thomas P. Minka

See Also

`make.names`

Examples

```
make.unique(c("a", "a", "a"))
make.unique(c(make.unique(c("a", "a")), "a"))

make.unique(c("a", "a", "a.2", "a"))
make.unique(c(make.unique(c("a", "a")), "a.2", "a"))

## Now show a bit where this is used :
trace(make.unique)
## Applied in data.frame() constructions:
(d1 <- data.frame(x = 1, x = 2, x = 3)) # direct
 d2 <- data.frame(data.frame(x = 1, x = 2), x = 3) # pairwise
stopifnot(identical(d1, d2),
          colnames(d1) == c("x", "x.1", "x.2"))
untrace(make.unique)
```

`mapply` *Apply a Function to Multiple List or Vector Arguments*

Description

`mapply` is a multivariate version of `sapply`. `mapply` applies `FUN` to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

`.mapply()` is a bare-bones version of `mapply()`, e.g., to be used in other functions.

Usage

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
       USE.NAMES = TRUE)
.mapply(FUN, dots, MoreArgs)
```

Arguments

FUN	function to apply, found via match.fun.
...	arguments to vectorize over, will be recycled to common length (zero if one of them is). See also 'Details'.
dots	list or pairlist of arguments to vectorize over, see ... above.
MoreArgs	a list of other arguments to FUN.
SIMPLIFY	logical or character string; attempt to reduce the result to a vector, matrix or higher dimensional array; see the simplify argument of sapply.
USE.NAMES	logical; use the names of the first ... argument, or if that is an unnamed character vector, use that vector as the names.

Details

mapply calls FUN for the values of ... (re-cycled to the length of the longest, unless any have length zero where recycling to zero length will return list()), followed by the arguments given in MoreArgs. The arguments in the call will be named if ... or MoreArgs are named.

For the arguments in ... (or components in dots) class specific subsetting (such as [) and length methods will be used where applicable.

Value

A list, or for SIMPLIFY = TRUE, a vector, array or list.

See Also

sapply, after which mapply() is modelled.

outer, which applies a vectorized function to all combinations of two arguments.

Examples

```
mapply(rep, 1:4, 4:1)

mapply(rep, times = 1:4, x = 4:1)

mapply(rep, times = 1:4, MoreArgs = list(x = 42))

mapply(function(x, y) seq_len(x) + y,
       c(a =  1, b = 2, c = 3),  # names from first
       c(A = 10, B = 0, C = -10))

word <- function(C, k) paste(rep.int(C, k), collapse = "")
## names from the first, too:
utils::str(L <- mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE))

mapply(word, "A", integer()) # gave Error, now list()
```

marginSums *Compute Table Margins*

Description

For a contingency table in array form, compute the sum of table entries for a given margin or set of margins.

Usage

```
marginSums(x, margin = NULL)
margin.table(x, margin = NULL)
```

Arguments

x	an array, usually a table.
margin	a vector giving the margins to compute sums for. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. When x has named dimnames, it can be a character vector selecting dimension names.

Value

The relevant marginal table, or just the sum of all entries if margin has length zero. The class of x is copied to the output table if margin is non-NULL.

Note

margin.table is an earlier name, retained for back-compatibility.

Author(s)

Peter Dalgaard

See Also

rowSums and colSums for similar functionality.

proportions and addmargins.

Examples

```
m <- matrix(1:4, 2)
marginSums(m, 1)  # = rowSums(m)
marginSums(m, 2)  # = colSums(m)

DF <- as.data.frame(UCBAdmissions)
tbl <- xtabs(Freq ~ Gender + Admit, DF)
tbl
marginSums(tbl, "Gender")  # a 1-dim "table"
rowSums(tbl)               # a numeric vector
```

mat.or.vec *Create a Matrix or a Vector*

Description

`mat.or.vec` creates an `nr` by `nc` zero matrix if `nc` is greater than 1, and a zero vector of length `nr` if `nc` equals 1.

Usage

```
mat.or.vec(nr, nc)
```

Arguments

`nr`, `nc`	numbers of rows and columns.

Examples

```
mat.or.vec(3, 1)
mat.or.vec(3, 2)
```

match *Value Matching*

Description

`match` returns a vector of the positions of (first) matches of its first argument in its second.

`%in%` is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

```
match(x, table, nomatch = NA_integer_, incomparables = NULL)

x %in% table
```

Arguments

`x`	vector or `NULL`: the values to be matched. Long vectors are supported.
`table`	vector or `NULL`: the values to be matched against. Long vectors are not supported.
`nomatch`	the value to be returned in the case when no match is found. Note that it is coerced to `integer`.
`incomparables`	a vector of values that cannot be matched. Any value in `x` matching a value in this vector is assigned the `nomatch` value. For historical reasons, `FALSE` is equivalent to `NULL`.

Details

`%in%` is currently defined as
`"%in%" <- function(x, table) match(x, table, nomatch = 0) > 0`

Factors, raw vectors and lists are converted to character vectors, internally classed objects are transformed via `mtfrm`, and then `x` and `table` are coerced to a common type (the later of the two types in R's ordering, logical < integer < numeric < complex < character) before matching. If `incomparables` has positive length it is coerced to the common type.

Matching for lists is potentially very slow and best avoided except in simple cases.

Exactly what matches what is to some extent a matter of definition. For all types, `NA` matches `NA` and no other value. For real and complex values, `NaN` values are regarded as matching any other `NaN` value, but not matching `NA`, where for complex `x`, real and imaginary parts must match both (unless containing at least one `NA`).

Character strings will be compared as byte sequences if any input is marked as `"bytes"`, and otherwise are regarded as equal if they are in different encodings but would agree when translated to UTF-8 (see `Encoding`).

That `%in%` never returns `NA` makes it particularly useful in `if` conditions.

Value

A vector of the same length as `x`.

`match`: An integer vector giving the position in `table` of the first match if there is a match, otherwise `nomatch`.

If `x[i]` is found to equal `table[j]` then the value returned in the `i`-th position of the return value is `j`, for the smallest possible `j`. If no match is found, the value is `nomatch`.

`%in%`: A logical vector, indicating if a match was located for each element of `x`: thus the values are `TRUE` or `FALSE` and never `NA`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`pmatch` and `charmatch` for (*partial*) string matching, `match.arg`, etc for function argument matching. `findInterval` similarly returns a vector of positions, but finds numbers within intervals, rather than exact matches.

`is.element` for an S-compatible equivalent of `%in%`.

`unique` (and `duplicated`) are using the same definitions of "match" or "equality" as `match()`, and these are less strict than `==`, e.g., for `NA` and `NaN` in numeric or complex vectors, or for strings with different encodings, see also above.

Examples

```
## The intersection of two sets can be defined via match():
## Simple version:
## intersect <- function(x, y) y[match(x, y, nomatch = 0)]
intersect # the R function in base is slightly more careful
intersect(1:10, 7:20)

1:10 %in% c(1,3,5,9)
sstr <- c("c","ab","B","bba","c",NA,"@","bla","a","Ba","%")
sstr[sstr %in% c(letters, LETTERS)]

"%w/o%" <- function(x, y) x[!x %in% y] #--  x without y
(1:10) %w/o% c(3,7,12)
## Note that setdiff() is very similar and typically makes more sense:
        c(1:6,7:2) %w/o% c(3,7,12)  # -> keeps duplicates
setdiff(c(1:6,7:2),      c(3,7,12)) # -> unique values

## Illuminating example about NA matching
r <- c(1, NA, NaN)
zN <- c(complex(real = NA , imaginary =  r ), complex(real =  r , imaginary = NA ),
        complex(real =  r , imaginary = NaN), complex(real = NaN, imaginary =  r ))
zM <- cbind(Re=Re(zN), Im=Im(zN), match = match(zN, zN))
rownames(zM) <- format(zN)
zM ##--> many "NA's" (= 1) and the four non-NA's (3 different ones, at 7,9,10)

length(zN) # 12
unique(zN) # the "NA" and the 3 different non-NA NaN's
stopifnot(identical(unique(zN), zN[c(1, 7,9,10)]))

## very strict equality would have 4 duplicates (of 12):
symnum(outer(zN, zN, Vectorize(identical,c("x","y")),
                     FALSE,FALSE,FALSE,FALSE))
## removing "(very strictly) duplicates",
i <- c(5,8,11,12)  # we get 8 pairwise non-identicals :
Ixy <- outer(zN[-i], zN[-i], Vectorize(identical,c("x","y")),
                     FALSE,FALSE,FALSE,FALSE)
stopifnot(identical(Ixy, diag(8) == 1))
```

`match.arg` *Argument Verification Using Partial Matching*

Description

`match.arg` matches a character `arg` against a table of candidate values as specified by `choices`.

Usage

```
match.arg(arg, choices, several.ok = FALSE)
```

Arguments

arg	a character vector (of length one unless several.ok is TRUE) or NULL which means to take choices[1].
choices	a character vector of candidate values, often missing, see 'Details'.
several.ok	logical specifying if arg should be allowed to have more than one element.

Details

In the one-argument form match.arg(arg), the choices are obtained from a default setting for the formal argument arg of the function from which match.arg was called. (Since default argument matching will set arg to choices, this is allowed as an exception to the 'length one unless several.ok is TRUE' rule, and returns the first element.)

Matching is done using pmatch, so arg may be abbreviated and the empty string ("") never matches, not even itself, see pmatch.

Value

The unabbreviated version of the exact or unique partial match if there is one; otherwise, an error is signalled if several.ok is false, as per default. When several.ok is true and (at least) one element of arg has a match, all unabbreviated versions of matches are returned.

Warning

The error messages given are liable to change and did so in R 4.2.0. Do not test them in packages.

See Also

pmatch, match.fun, match.call.

Examples

```
require(stats)
## Extends the example for 'switch'
center <- function(x, type = c("mean", "median", "trimmed")) {
  type <- match.arg(type)
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
center(x, "t")       # Works
center(x, "med")     # Works
try(center(x, "m"))  # Error
stopifnot(identical(center(x),       center(x, "mean")),
          identical(center(x, NULL), center(x, "mean")) )

## Allowing more than one 'arg' and hence more than one match:
match.arg(c("gauss", "rect", "ep"),
          c("gaussian", "epanechnikov", "rectangular", "triangular"),
```

```
          several.ok = TRUE)
match.arg(c("a", ""),  c("", NA, "bb", "abc"), several.ok=TRUE) # |-->  "abc"
```

`match.call` *Argument Matching*

Description

`match.call` returns a call in which all of the specified arguments are specified by their full names.

Usage

```
match.call(definition = sys.function(sys.parent()),
           call = sys.call(sys.parent()),
           expand.dots = TRUE,
           envir = parent.frame(2L))
```

Arguments

`definition`	a function, by default the function from which `match.call` is called. See details.
`call`	an unevaluated call to the function specified by `definition`, as generated by `call`.
`expand.dots`	logical. Should arguments matching `...` in the call be included or left as a `...` argument?
`envir`	an environment, from which the `...` in `call` are retrieved, if any.

Details

'function' on this help page means an interpreted function (also known as a 'closure'): `match.call` does not support primitive functions (where argument matching is normally positional).

`match.call` is most commonly used in two circumstances:

- To record the call for later re-use: for example most model-fitting functions record the call as element `call` of the list they return. Here the default `expand.dots = TRUE` is appropriate.
- To pass most of the call to another function, often `model.frame`. Here the common idiom is that `expand.dots = FALSE` is used, and the `...` element of the matched call is removed. An alternative is to explicitly select the arguments to be passed on, as is done in `lm`.

Calling `match.call` outside a function without specifying `definition` is an error.

Value

An object of class `call`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

sys.call() is similar, but does *not* expand the argument names; call, pmatch, match.arg, match.fun.

Examples

```
match.call(get, call("get", "abc", i = FALSE, p = 3))
## -> get(x = "abc", pos = 3, inherits = FALSE)
fun <- function(x, lower = 0, upper = 1) {
  structure((x - lower) / (upper - lower), CALL = match.call())
}
fun(4 * atan(1), u = pi)
```

---

match.fun *Extract a Function Specified by Name*

Description

When called inside functions that take a function as argument, extract the desired function object while avoiding undesired matching to objects of other types.

Usage

```
match.fun(FUN, descend = TRUE)
```

Arguments

FUN	item to match as function: a function, symbol or character string. See 'Details'.
descend	logical; control whether to search past non-function objects.

Details

match.fun is not intended to be used at the top level since it will perform matching in the *parent* of the caller.

If FUN is a function, it is returned. If it is a symbol (for example, enclosed in backquotes) or a character vector of length one, it will be looked up using get in the environment of the parent of the caller. If it is of any other mode, it is attempted first to get the argument to the caller as a symbol (using substitute twice), and if that fails, an error is declared.

If descend = TRUE, match.fun will look past non-function objects with the given name; otherwise if FUN points to a non-function object then an error is generated.

This is used in base functions such as apply, lapply, outer, and sweep.

Value

A function matching FUN or an error is generated.

Bugs

The descend argument is a bit of misnomer and probably not actually needed by anything. It may go away in the future.

It is impossible to fully foolproof this. If one attaches a list or data frame containing a length-one character vector with the same name as a function, it may be used (although namespaces will help).

Author(s)

Peter Dalgaard and Robert Gentleman, based on an earlier version by Jonathan Rougier.

See Also

match.arg, get

Examples

```
# Same as get("*"):
match.fun("*")
# Overwrite outer with a vector
outer <- 1:5
try(match.fun(outer, descend = FALSE)) #-> Error:  not a function
match.fun(outer) # finds it anyway
is.function(match.fun("outer")) # as well
```

MathFun *Miscellaneous Mathematical Functions*

Description

abs(x) computes the absolute value of x, sqrt(x) computes the (principal) square root of x, $\sqrt{x}$.

The naming follows the standard for computer languages such as C or Fortran.

Usage

```
abs(x)
sqrt(x)
```

Arguments

x	a numeric or complex vector or array.

Details

These are internal generic primitive functions: methods can be defined for them individually or via the Math group generic. For complex arguments (and the default method), z, abs(z) == Mod(z) and sqrt(z) == z^0.5.

abs(x) returns an integer vector when x is integer or logical.

S4 methods

Both are S4 generic and members of the Math group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

Arithmetic for simple, log for logarithmic, sin for trigonometric, and Special for special mathematical functions.

'plotmath' for the use of sqrt in plot annotation.

Examples

```
require(stats) # for spline
require(graphics)
xx <- -9:9
plot(xx, sqrt(abs(xx)),  col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

---

matmult *Matrix Multiplication*

Description

Multiplies two matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors of the same length, it will return the inner product (as a matrix).

Usage

```
x %*% y
```

Arguments

x, y	numeric or complex matrices or vectors.

Details

When a vector is promoted to a matrix, its names are not promoted to row or column names, unlike as.matrix.

Promotion of a vector to a 1-row or 1-column matrix happens when one of the two choices allows x and y to get conformable dimensions.

This operator is a generic function: methods can be written for it individually or via the matOps group generic function; it dispatches to S3 and S4 methods. Methods need to be written for a function that takes two arguments named x and y.

Value

A double or complex matrix product. Use drop to remove dimensions which have only one level.

Note

The propagation of NaN/Inf values, precision, and performance of matrix products can be controlled by options("matprod").

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

For matrix *cross* products, crossprod() and tcrossprod() are typically preferable. matrix, Arithmetic, diag.

Examples

```
x <- 1:4
(z <- x %*% x)    # scalar ("inner") product (1 x 1 matrix)
drop(z)             # as scalar

y <- diag(x)
z <- matrix(1:12, ncol = 3, nrow = 4)
y %*% z
y %*% x
x %*% z
```

---

matrix *Matrices*

Description

matrix creates a matrix from the given set of values.

as.matrix attempts to turn its argument into a matrix.

is.matrix tests if its argument is a (strict) matrix.

Usage

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)

as.matrix(x, ...)
## S3 method for class 'data.frame'
as.matrix(x, rownames.force = NA, ...)

is.matrix(x)
```

Arguments

data	an optional data vector (including a list or expression vector). Non-atomic classed R objects are coerced by as.vector and all attributes discarded.
nrow	the desired number of rows.
ncol	the desired number of columns.
byrow	logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.
dimnames	a dimnames attribute for the matrix: NULL or a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
x	an R object.
...	additional arguments to be passed to or from methods.
rownames.force	logical indicating if the resulting matrix should have character (rather than NULL) rownames. The default, NA, uses NULL rownames if the data frame has 'automatic' row.names or for a zero-row data frame.

Details

If one of nrow or ncol is not given, an attempt is made to infer it from the length of data and the other parameter. If neither is given, a one-column matrix is returned.

If there are too few elements in data to fill the matrix, then the elements in data are recycled. If data has length zero, NA of an appropriate type is used for atomic vectors (0 for raw vectors) and NULL for lists.

is.matrix returns TRUE if x is a vector and has a "dim" attribute of length 2 and FALSE otherwise. Note that a data.frame is **not** a matrix by this test. The function is generic: you can write methods to handle specific classes of objects, see InternalMethods.

as.matrix is a generic function. The method for data frames will return a character matrix if there is only atomic columns and any non-(numeric/logical/complex) column, applying as.vector to factors and format to other non-character columns. Otherwise, the usual coercion hierarchy (logical < integer < double < complex) will be used, e.g., all-logical data frames will be coerced to a logical matrix, mixed logical-integer will give a integer matrix, etc.

The default method for as.matrix calls as.vector(x), and hence e.g. coerces factors to character vectors.

When coercing a vector, it produces a one-column matrix, and promotes the names (if any) of the vector to the rownames of the matrix.

is.matrix is a primitive function.

The print method for a matrix gives a rectangular layout with dimnames or indices. For a list matrix, the entries of length not one are printed in the form 'integer,7' indicating the type and length.

Note

If you just want to convert a vector to a matrix, something like

```
dim(x) <- c(nx, ny)
dimnames(x) <- list(row_names, col_names)
```

will avoid duplicating `x` *and* preserve `class(x)` which may be useful, e.g., for `Date` objects.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`data.matrix`, which attempts to convert to a numeric matrix.

A matrix is the special case of a two-dimensional `array`. `inherits(m, "array")` is true for a `matrix` `m`.

Examples

```
is.matrix(as.matrix(1:10))
!is.matrix(warpbreaks)  # data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,])  # using as.matrix.data.frame(.) method

## Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
               dimnames = list(c("row1", "row2"),
                               c("C.1", "C.2", "C.3")))
mdat
```

---

maxCol *Find Maximum Position in Matrix*

Description

Find the maximum position for each row of a matrix, breaking ties at random.

Usage

```
max.col(m, ties.method = c("random", "first", "last"))
```

Arguments

`m`	a numerical matrix.
`ties.method`	a character string specifying how ties are handled, `"random"` by default; can be abbreviated; see 'Details'.

Details

When `ties.method = "random"`, as per default, ties are broken at random. In this case, the determination of a tie assumes that the entries are probabilities: there is a relative tolerance of 10^{-5}, relative to the largest (in magnitude, omitting infinity) entry in the row.

If `ties.method = "first"`, `max.col` returns the column number of the *first* of several maxima in every row, the same as `unname(apply(m, 1, which.max))` if `m` has no missing values. Correspondingly, `ties.method = "last"` returns the *last* of possibly several indices.

Value

index of a maximal value for each row, an integer vector of length `nrow(m)`.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

See Also

`which.max` for vectors.

Examples

```
table(mc <- max.col(swiss))  # mostly "1" and "5", 5 x "2" and once "4"
swiss[unique(print(mr <- max.col(t(swiss)))) , ]  # 3 33 45 45 33 6

set.seed(1)  # reproducible example:
(mm <- rbind(x = round(2*stats::runif(12)),
             y = round(5*stats::runif(12)),
             z = round(8*stats::runif(12))))
## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    1    1    2    0    2    2    1    1     0     0     0
y    3    2    4    2    4    5    2    4    5     1     3     1
z    2    3    0    3    7    3    4    5    4     1     7     5

## End(Not run)
## column indices of all row maxima :
utils::str(lapply(1:3, function(i) which(mm[i,] == max(mm[i,]))))
max.col(mm) ; max.col(mm) # "random"
max.col(mm, "first") # -> 4 6 5
max.col(mm, "last")  # -> 7 9 11
```

mean *Arithmetic Mean*

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	an R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

memCompress *In-memory Compression and Decompression*

Description

In-memory compression or decompression for raw vectors.

Usage

```
memCompress(from, type = c("gzip", "bzip2", "xz", "none"))

memDecompress(from,
              type = c("unknown", "gzip", "bzip2", "xz", "none"),
              asChar = FALSE)
```

Arguments

from	raw vector. For memCompress, a character vector will be converted to a raw vector with character strings separated by "\n". Types except "bzip2" support long raw vectors.
type	character string, the type of compression. May be abbreviated to a single letter, defaults to the first of the alternatives.
asChar	logical: should the result be converted to a character string? NB: character strings have a limit of $2^{31} - 1$ bytes, so raw vectors should be used for large inputs.

Details

type = "none" passes the input through unchanged, but may be useful if type is a variable.

type = "unknown" attempts to detect the type of compression applied (if any): this will always succeed for bzip2 compression, and will succeed for other forms if there is a suitable header. If no type of compression is detected this is the same as type = "none" but a warning is given.

gzip compression uses whatever is the default compression level of the underlying library (usually 6). This supports the RFC 1950 format, sometimes known as 'zlib' format, for compression and decompression and for decompression only RFC 1952, the 'gzip' format (which wraps the 'zlib' format with a header and footer).

bzip2 compression always adds a header ("BZh"). The underlying library only supports in-memory (de)compression of up to $2^{31} - 1$ elements. Compression is equivalent to bzip2 -9 (the default).

Compressing with type = "xz" is equivalent to compressing a file with xz -9e (including adding the 'magic' header): decompression should cope with the contents of any file compressed by xz version 4.999 and later, as well as by some versions of lzma. There are other versions, in particular 'raw' streams, that are not currently handled.

All the types of compression can expand the input: for "gzip" and "bzip2" the maximum expansion is known and so memCompress can always allocate sufficient space. For "xz" it is possible (but extremely unlikely) that compression will fail if the output would have been too large.

Value

A raw vector or a character string (if `asChar = TRUE`).

`libdeflate`

Support for the `libdeflate` library was added for R 4.4.0. It uses different code for the RFC 1950 'zlib' format (and RFC 1952 for decompression), expected to be substantially faster than using the reference (or system) zlib library. It is used for `type = "gzip"` if available.

The headers and sources can be downloaded from https://github.com/ebiggers/libdeflate and pre-built versions are available for most Linux distributions. It is used for binary Windows distributions.

See Also

connections.

`extSoftVersion` for the versions of the `zlib` or `libdeflate`, `bzip2` and `xz` libraries in use.

https://en.wikipedia.org/wiki/Data_compression for background on data compression, https://zlib.net/, https://en.wikipedia.org/wiki/Gzip, http://www.bzip.org/, https://en.wikipedia.org/wiki/Bzip2, and https://en.wikipedia.org/wiki/XZ_Utils for references about the particular schemes used.

Examples

```
txt <- readLines(file.path(R.home("doc"), "COPYING"))
sum(nchar(txt))
txt.gz <- memCompress(txt, "g") # "gzip", the default
length(txt.gz)
txt2 <- strsplit(memDecompress(txt.gz, "g", asChar = TRUE), "\n")[[1]]
stopifnot(identical(txt, txt2))
## as from R 4.4.0 this is detected if not specified.
txt2b <- strsplit(memDecompress(txt.gz, asChar = TRUE), "\n")[[1]]
stopifnot(identical(txt2b, txt2))

txt.bz2 <- memCompress(txt, "b")
length(txt.bz2)
## can auto-detect bzip2:
txt3 <- strsplit(memDecompress(txt.bz2, asChar = TRUE), "\n")[[1]]
stopifnot(identical(txt, txt3))

## xz compression is only worthwhile for large objects
txt.xz <- memCompress(txt, "x")
length(txt.xz)
txt3 <- strsplit(memDecompress(txt.xz, asChar = TRUE), "\n")[[1]]
stopifnot(identical(txt, txt3))

## test decompressing a gzip-ed file
tf <- tempfile(fileext = ".gz")
con <- gzfile(tf, "w")
writeLines(txt, con)
close(con)
```

```
(nf <- file.size(tf))
# if (nzchar(Sys.which("file"))) system2("file", tf)
foo <- readBin(tf, "raw", n = nf)
unlink(tf)
## will detect the gzip header and choose type = "gzip"
txt3 <- strsplit(memDecompress(foo, asChar = TRUE), "\n")[[1]]
stopifnot(identical(txt, txt3))
```

`memlimits` *Query and Set Heap Size Limits*

Description

Query and set the maximal size of the vector heap and the maximal number of heap nodes for the current R process.

Usage

```
mem.maxVSize(vsize = 0)
mem.maxNSize(nsize = 0)
```

Arguments

`vsize`	numeric; new size limit in Mb.
`nsize`	numeric; new maximal node number.

Details

New limits lower than current usage are ignored. Specifying a size of `Inf` sets the limit to the maximal possible value for the platform.

The default maximal values are unlimited on most platforms, but can be adjusted using environment variables as described in `Memory`. On macOS a lower default vector heap limit is used to protect against the R process being killed when macOS over-commits memory.

Adjusting the maximal number of nodes is rarely necessary. Adjusting the vector heap size limit can be useful on macOS in particular but should be done with caution.

Value

The current or new value, in Mb for `mem.maxVSize`. `Inf` is returned if the current value is unlimited.

See Also

`Memory`.

Memory *Memory Available for Data Storage*

Description

How R manages its workspace.

Details

R has a variable-sized workspace. There are (rarely-used) command-line options to control its minimum size, but no longer any to control the maximum size.

R maintains separate areas for fixed and variable sized objects. The first of these is allocated as an array of *cons cells* (Lisp programmers will know what they are, others may think of them as the building blocks of the language itself, parse trees, etc.), and the second are thrown on a *heap* of 'Vcells' of 8 bytes each. Each cons cell occupies 28 bytes on a 32-bit build of R, (usually) 56 bytes on a 64-bit build.

The default values are (currently) an initial setting of 350k cons cells and 6Mb of vector heap. Note that the areas are not actually allocated initially: rather these values are the sizes for triggering garbage collection. These values can be set by the command line options '`--min-nsize`' and '`--min-vsize`' (or if they are not used, the environment variables `R_NSIZE` and `R_VSIZE`) when R is started. Thereafter R will grow or shrink the areas depending on usage, never decreasing below the initial values. The maximal vector heap size can be set with the environment variable `R_MAX_VSIZE`. An attempt to set a lower maximum than the current usage is ignored. Vector heap limits are given in bytes.

How much time R spends in the garbage collector will depend on these initial settings and on the trade-off the memory manager makes, when memory fills up, between collecting garbage to free up unused memory and growing these areas. The strategy used for growth can be specified by setting the environment variable `R_GC_MEM_GROW` to an integer value between 0 and 3. This variable is read at start-up. Higher values grow the heap more aggressively, thus reducing garbage collection time but using more memory.

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes) by typing `gc()` at the R prompt. Note that following `gcinfo(TRUE)`, automatic garbage collection always prints memory use statistics.

The command-line option '`--max-ppsize`' controls the maximum size of the pointer protection stack. This defaults to 50000, but can be increased to allow deep recursion or large and complicated calculations to be done. *Note* that parts of the garbage collection process goes through the full reserved pointer protection stack and hence becomes slower when the size is increased. Currently the maximum value accepted is 500000.

See Also

An Introduction to R for more command-line options.

`Memory-limits` for the design limitations.

`gc` for information on the garbage collector and total memory usage, `object.size(a)` for the (approximate) size of R object `a`. `memory.profile` for profiling the usage of cons cells.

`Memory-limits` *Memory Limits in R*

Description

R holds objects it is using in virtual memory. This help file documents the current design limitations on large objects: these differ between 32-bit and 64-bit builds of R.

Details

Currently R runs on 32- and 64-bit operating systems, and most 64-bit OSes (including Linux, Solaris, Windows and macOS) can run either 32- or 64-bit builds of R. The memory limits depends mainly on the build, but for a 32-bit build of R on Windows they also depend on the underlying OS version.

R holds all objects in virtual memory, and there are limits based on the amount of memory that can be used by all objects:

- There may be limits on the size of the heap and the number of cons cells allowed – see `Memory` – but these are usually not imposed.
- There is a limit on the (user) address space of a single process such as the R executable. This is system-specific, and can depend on the executable.
- The environment may impose limitations on the resources available to a single process: Windows' versions of R do so directly.

Error messages beginning '`cannot allocate vector of size`' indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory. Note that on a 32-bit build there may well be enough free memory available, but not a large enough contiguous block of address space into which to map it.

There are also limits on individual objects. The storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins '`cannot allocate vector of length`'. The number of bytes in a character string is limited to $2^{31} - 1 \approx 2\,10^9$, which is also the limit on each dimension of an array.

Unix

The address-space limit is system-specific: 32-bit OSes imposes a limit of no more than 4Gb: it is often 3Gb. Running 32-bit executables on a 64-bit OS will have similar limits: 64-bit executables will have an essentially infinite system-specific limit (e.g., 128Tb for Linux on x86_64 CPUs).

See the OS/shell's help on commands such as `limit` or `ulimit` for how to impose limitations on the resources available to a single process. For example a `bash` user could use

```
ulimit -t 600 -v 4000000
```

whereas a `csh` user might use

```
limit cputime 10m
limit vmemoryuse 4096m
```

to limit a process to 10 minutes of CPU time and (around) 4Gb of virtual memory. (There are other options to set the RAM in use, but they are not generally honoured.)

Windows

The address-space limit is 2Gb under 32-bit Windows unless the OS's default has been changed to allow more (up to 3Gb). See https://docs.microsoft.com/en-gb/windows/desktop/Memory/physical-address-extension and https://docs.microsoft.com/en-gb/windows/desktop/Memory/4-gigabyte-tuning. Under most 64-bit versions of Windows the limit for a 32-bit build of R is 4Gb: for the oldest ones it is 2Gb. The limit for a 64-bit build of R (imposed by the OS) is 8Tb.

It is not normally possible to allocate as much as 2Gb to a single vector in a 32-bit build of R even on 64-bit Windows because of preallocations by Windows in the middle of the address space.

See Also

`object.size(a)` for the (approximate) size of R object `a`.

`memory.profile` *Profile the Usage of Cons Cells*

Description

Lists the usage of the cons cells by `SEXPREC` type.

Usage

```
memory.profile()
```

Details

The current types and their uses are listed in the include file '`Rinternals.h`'.

Value

A vector of counts, named by the types. See `typeof` for an explanation of types.

See Also

`gc` for the overall usage of cons cells. `Rprofmem` and `tracemem` allow memory profiling of specific code or objects, but need to be enabled at compile time.

Examples

```
memory.profile()
```

merge *Merge Two Data Frames*

Description

Merge two data frames by common columns or row names, or do other versions of database *join* operations.

Usage

```
merge(x, y, ...)

## Default S3 method:
merge(x, y, ...)

## S3 method for class 'data.frame'
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x",".y"), no.dups = TRUE,
      incomparables = NULL, ...)
```

Arguments

x, y	data frames, or objects to be coerced to one.
by, by.x, by.y	specifications of the columns used for merging. See 'Details'.
all	logical; all = L is shorthand for all.x = L and all.y = L, where L is either TRUE or FALSE.
all.x	logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is FALSE, so that only rows with data from both x and y are included in the output.
all.y	logical; analogous to all.x.
sort	logical. Should the result be sorted on the by columns?
suffixes	a character vector of length 2 specifying the suffixes to be used for making unique the names of columns in the result which are not used for merging (appearing in by etc).
no.dups	logical indicating that suffixes are appended in more cases to avoid duplicated column names in the result. This was implicitly false before R version 3.5.0.
incomparables	values which cannot be matched. See match. This is intended to be used for merging on one column, so these are incomparable values of that column.
...	arguments to be passed to or from methods.

Details

merge is a generic function whose principal method is for data frames: the default method coerces its arguments to data frames and calls the "data.frame" method.

By default the data frames are merged on the columns with names they both have, but separate specifications of the columns can be given by by.x and by.y. The rows in the two data frames that match on the specified columns are extracted, and joined together. If there is more than one match, all possible matches contribute one row each. For the precise meaning of 'match', see match.

Columns to merge on can be specified by name, number or by a logical vector: the name "row.names" or the number 0 specifies the row names. If specified by name it must correspond uniquely to a named column in the input.

If by or both by.x and by.y are of length 0 (a length zero vector or NULL), the result, r, is the *Cartesian product* of x and y, i.e., dim(r) = c(nrow(x)*nrow(y), ncol(x) + ncol(y)).

If all.x is true, all the non matching cases of x are appended to the result as well, with NA filled in the corresponding columns of y; analogously for all.y.

If the columns in the data frames not used in merging have any common names, these have suffixes (".x" and ".y" by default) appended to try to make the names of the result unique. If this is not possible, an error is thrown.

If a by.x column name matches one of y, and if no.dups is true (as by default), the y version gets suffixed as well, avoiding duplicate column names in the result.

The complexity of the algorithm used is proportional to the length of the answer.

In SQL database terminology, the default value of all = FALSE gives a *natural join*, a special case of an *inner join*. Specifying all.x = TRUE gives a *left (outer) join*, all.y = TRUE a *right (outer) join*, and both (all = TRUE) a *(full) outer join*. DBMSes do not match NULL records, equivalent to incomparables = NA in R.

Value

A data frame. The rows are by default lexicographically sorted on the common columns, but for sort = FALSE are in an unspecified order. The columns are the common columns followed by the remaining columns in x and then those in y. If the matching involved row names, an extra character column called Row.names is added at the left, and in all cases the result has 'automatic' row names.

Note

This is intended to work with data frames with vector-like columns: some aspects work with data frames containing matrices, but not all.

Currently long vectors are not accepted for inputs, which are thus restricted to less than 2^31 rows. That restriction also applies to the result for 32-bit platforms.

See Also

data.frame, by, cbind.

dendrogram for a class which has a merge method.

Examples

```
authors <- data.frame(
    ## I(*) : use character columns of names to get sensible sort order
    surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
authorN <- within(authors, { name <- surname; rm(surname) })
books <- data.frame(
    name = I(c("Tukey", "Venables", "Tierney",
             "Ripley", "Ripley", "McNeil", "R Core")),
    title = c("Exploratory Data Analysis",
              "Modern Applied Statistics ...",
              "LISP-STAT",
              "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis",
              "An Introduction to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA,
                     "Venables & Smith"))

(m0 <- merge(authorN, books))
(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
 m2 <- merge(books, authors, by.x = "name", by.y = "surname")
stopifnot(exprs = {
   identical(m0, m2[, names(m0)])
   as.character(m1[, 1]) == as.character(m2[, 1])
   all.equal(m1[, -1], m2[, -1][ names(m1)[-1] ])
   identical(dim(merge(m1, m2, by = NULL)),
             c(nrow(m1)*nrow(m2), ncol(m1)+ncol(m2)))
})

## "R core" is missing from authors and appears only here :
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)


## example of using 'incomparables'
x <- data.frame(k1 = c(NA,NA,3,4,5), k2 = c(1,NA,NA,4,5), data = 1:5)
y <- data.frame(k1 = c(NA,2,NA,4,5), k2 = c(NA,NA,3,4,5), data = 1:5)
merge(x, y, by = c("k1","k2")) # NA's match
merge(x, y, by = "k1") # NA's match, so 6 rows
merge(x, y, by = "k2", incomparables = NA) # 2 rows
```

`message` *Diagnostic Messages*

Description

Generate a diagnostic message from its arguments.

Usage

```
message(..., domain = NULL, appendLF = TRUE)
suppressMessages(expr, classes = "message")

packageStartupMessage(..., domain = NULL, appendLF = TRUE)
suppressPackageStartupMessages(expr)

.makeMessage(..., domain = NULL, appendLF = FALSE)
```

Arguments

...	zero or more objects which can be coerced to character (and which are pasted together with no separator) or (for message only) a single condition object.
domain	see gettext. If NA, messages will not be translated, see also the note in stop.
appendLF	logical: should messages given as a character string have a newline appended?
expr	expression to evaluate.
classes	character, indicating which classes of messages should be suppressed.

Details

message is used for generating 'simple' diagnostic messages which are neither warnings nor errors, but nevertheless represented as conditions. Unlike warnings and errors, a final newline is regarded as part of the message, and is optional. The default handler sends the message to the stderr() connection.

If a condition object is supplied to message it should be the only argument, and further arguments will be ignored, with a warning.

While the message is being processed, a muffleMessage restart is available.

suppressMessages evaluates its expression in a context that ignores all 'simple' diagnostic messages.

packageStartupMessage is a variant whose messages can be suppressed separately by suppressPackageStartupMessages. (They are still messages, so can be suppressed by suppressMessages.)

.makeMessage is a utility used by message, warning and stop to generate a text message from the ... arguments by possible translation (see gettext) and concatenation (with no separator).

See Also

warning and stop for generating warnings and errors; conditions for condition handling and recovery.

gettext for the mechanisms for the automated translation of text.

Examples

```
message("ABC", "DEF")
suppressMessages(message("ABC"))
```

```
testit <- function() {
  message("testing package startup messages")
  packageStartupMessage("initializing ...", appendLF = FALSE)
  Sys.sleep(1)
  packageStartupMessage(" done")
}

testit()
suppressPackageStartupMessages(testit())
suppressMessages(testit())
```

`missing` *Does a Formal Argument have a Value?*

Description

`missing` can be used to test whether a value was specified as an argument to a function.

Usage

```
missing(x)
```

Arguments

`x`	a formal argument.

Details

`missing(x)` is only reliable if `x` has not been altered since entering the function: in particular it will *always* be false after `x <- match.arg(x)`.

The example shows how a plotting function can be written to work with either a pair of vectors giving x and y coordinates of points to be plotted or a single vector giving y values to be plotted against their indices.

Currently `missing` can only be used in the immediate body of the function that defines the argument, not in the body of a nested function or a `local` call. This may change in the future.

This is a 'special' primitive function: it must not evaluate its argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`substitute` for argument expression; `NA` for missing values in data.

Examples

```
myplot <- function(x, y) {
                if(missing(y)) {
                        y <- x
                        x <- 1:length(y)
                }
                plot(x, y)
        }
```

---

mode *The (Storage) Mode of an Object*

Description

Get or set the 'mode' (a kind of 'type'), or the storage mode of an R object.

Usage

```
mode(x)
mode(x) <- value
storage.mode(x)
storage.mode(x) <- value
```

Arguments

x	any R object.
value	a character string giving the desired mode or 'storage mode' (type) of the object.

Details

Both mode and storage.mode return a character string giving the (storage) mode of the object — often the same — both relying on the output of typeof(x), see the example below.

mode(x) <- "newmode" changes the mode of object x to newmode. This is only supported if there is an appropriate as.newmode function, for example "logical", "integer", "double", "complex", "raw", "character", "list", "expression", "name", "symbol" and "function". Attributes are preserved (but see below).

storage.mode(x) <- "newmode" is a more efficient primitive version of mode<-, which works for "newmode" which is one of the internal types (see typeof), but not for "single". Attributes are preserved.

As storage mode "single" is only a pseudo-mode in R, it will not be reported by mode or storage.mode: use attr(object, "Csingle") to examine this. However, mode<- can be used to set the mode to "single", which sets the real mode to "double" and the "Csingle" attribute to TRUE. Setting any other mode will remove this attribute.

Note (in the examples below) that some calls have mode "(" which is S compatible.

Mode names

Modes have the same set of names as types (see typeof) except that

- types "integer" and "double" are returned as "numeric".
- types "special", "builtin" and "closure" are returned as "function".
- type "symbol" is called mode "name".
- type "language" is returned as "(" or "call".

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

typeof for the R-internal 'mode' or 'type', type.convert, attributes.

Examples

```
require(stats)

sapply(options(), mode)

cex3 <- c("NULL", "1", "1:1", "1i", "list(1)", "data.frame(x = 1)",
  "pairlist(pi)", "c", "lm", "formals(lm)[[1]]",  "formals(lm)[[2]]",
  "y ~ x","expression((1))[[1]]", "(y ~ x)[[1]]",
  "expression(x <- pi)[[1]][[1]]")
lex3 <- sapply(cex3, function(x) eval(str2lang(x)))
mex3 <- t(sapply(lex3,
                 function(x) c(typeof(x), storage.mode(x), mode(x))))
dimnames(mex3) <- list(cex3, c("typeof(.)","storage.mode(.)","mode(.)"))
mex3

## This also makes a local copy of 'pi':
storage.mode(pi) <- "complex"
storage.mode(pi)
rm(pi)
```

---

mtfrm *Auxiliary Function for Matching*

Description

Transform objects for matching via match(), think "match form" -> "mtfrm". **base** provides the S3 generic and a default plus "POSIXct" and "POSIXlt" methods.

Usage

```
mtfrm(x)
```

Arguments

x	an R object

Details

Matching via match will use mtfrm to transform internally classed objects (see is.object) to a vector representation appropriate for matching. The default method performs as.character if this preserves the length.

Ideally, methods for mtfrm should ensure that comparisons of same-classed objects via match are consistent with those employed by methods for duplicated/unique and ==/!= (where applicable).

Value

A vector of the same length as x.

NA *'Not Available' / Missing Values*

Description

NA is a logical constant of length 1 which contains a missing value indicator. NA can be coerced to any other vector type except raw. There are also constants NA_integer_, NA_real_, NA_complex_ and NA_character_ of the other atomic vector types which support missing values: all of these are reserved words in the R language.

The generic function is.na indicates which elements are missing.

The generic function is.na<- sets elements to NA.

The generic function anyNA implements any(is.na(x)) in a possibly faster way (especially for atomic vectors).

Usage

```
NA
is.na(x)
anyNA(x, recursive = FALSE)

## S3 method for class 'data.frame'
is.na(x)

is.na(x) <- value
```

Arguments

x	an R object to be tested: the default method for is.na and anyNA handle atomic vectors, lists, pairlists, and NULL.
recursive	logical: should anyNA be applied recursively to lists and pairlists?
value	a suitable index vector for use with x.

Details

The NA of character type is distinct from the string "NA". Programmers who need to specify an explicit missing string should use NA_character_ (rather than "NA") or set elements to NA using is.na<-.

is.na and anyNA are generic: you can write methods to handle specific classes of objects, see InternalMethods.

Function is.na<- may provide a safer way to set missingness. It behaves differently for factors, for example.

Numerical computations using NA will normally result in NA: a possible exception is where NaN is also involved, in which case either might result (which may depend on the R platform). However, this is not guaranteed and future CPUs and/or compilers may behave differently. Dynamic binary translation may also impact this behavior (with valgrind, computations using NA may result in NaN even when no NaN is involved).

Logical computations treat NA as a missing TRUE/FALSE value, and so may return TRUE or FALSE if the expression does not depend on the NA operand.

The default method for anyNA handles atomic vectors without a class and NULL. It calls any(is.na(x)) on objects with classes and for recursive = FALSE, on lists and pairlists.

Value

The default method for is.na applied to an atomic vector returns a logical vector of the same length as its argument x, containing TRUE for those elements marked NA or, for numeric or complex vectors, NaN, and FALSE otherwise. (A complex value is regarded as NA if either its real or imaginary part is NA or NaN.) dim, dimnames and names attributes are copied to the result.

The default methods also work for lists and pairlists:
For is.na, elementwise the result is false unless that element is a length-one atomic vector and the single element of that vector is regarded as NA or NaN (note that any is.na method for the class of the element is ignored).
anyNA(recursive = FALSE) works the same way as is.na; anyNA(recursive = TRUE) applies anyNA (with method dispatch) to each element.

The data frame method for is.na returns a logical matrix with the same dimensions as the data frame, and with dimnames taken from the row and column names of the data frame.

anyNA(NULL) is false; is.na(NULL) is logical(0) (no longer warning since R version 3.5.0).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

NaN, is.nan, etc., and the utility function complete.cases.

na.action, na.omit, na.fail on how methods can be tuned to deal with missing values.

Examples

```
is.na(c(1, NA))        #> FALSE  TRUE
is.na(paste(c(1, NA))) #> FALSE FALSE

(xx <- c(0:4))
is.na(xx) <- c(2, 4)
xx                     #> 0 NA  2 NA  4
anyNA(xx) # TRUE

# Some logical operations do not return NA
c(TRUE, FALSE) & NA
c(TRUE, FALSE) | NA


## Measure speed difference in a favourable case:
## the difference depends on the platform, on most ca 3x.
x <- 1:10000; x[5000] <- NaN  # coerces x to be double
if(require("microbenchmark")) { # does not work reliably on all platforms
  print(microbenchmark(any(is.na(x)), anyNA(x)))
} else {
  nSim <- 2^13
  print(rbind(is.na = system.time(replicate(nSim, any(is.na(x)))),
              anyNA = system.time(replicate(nSim, anyNA(x)))))
}


## anyNA() can work recursively with list()s:
LL <- list(1:5, c(NA, 5:8), c("A","NA"), c("a", NA_character_))
L2 <- LL[c(1,3)]
sapply(LL, anyNA); c(anyNA(LL), anyNA(LL, TRUE))
sapply(L2, anyNA); c(anyNA(L2), anyNA(L2, TRUE))

## ... lists, and hence data frames, too:
dN <- dd <- USJudgeRatings; dN[3,6] <- NA
anyNA(dd) # FALSE
anyNA(dN) # TRUE
```

---

name *Names and Symbols*

Description

A 'name' (also known as a 'symbol') is a way to refer to R objects by name (rather than the value of the object, if any, bound to that name).

`as.name` and `as.symbol` are identical: they attempt to coerce the argument to a name.

`is.symbol` and the identical `is.name` return `TRUE` or `FALSE` depending on whether the argument is a name or not.

Usage

```
as.symbol(x)
is.symbol(x)

as.name(x)
is.name(x)
```

Arguments

x	object to be coerced or tested.

Details

Names are limited to 10,000 bytes (and were to 256 bytes in versions of R before 2.13.0).

`as.name` first coerces its argument internally to a character vector (so methods for `as.character` are not used). It then takes the first element and provided it is not `""`, returns a symbol of that name (and if the element is `NA_character_`, the name is `` `NA` ``).

`as.name` is implemented as `as.vector(x, "symbol")`, and hence will dispatch methods for the generic function `as.vector`.

`is.name` and `is.symbol` are primitive functions.

Value

For `as.name` and `as.symbol`, an R object of type `"symbol"` (see `typeof`).

For `is.name` and `is.symbol`, a length-one logical vector with value `TRUE` or `FALSE`.

Note

The term 'symbol' is from the LISP background of R, whereas 'name' has been the standard S term for this.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call`, `is.language`. For the internal object mode, `typeof`.

`plotmath` for another use of 'symbol'.

Examples

```
an <- as.name("arrg")
is.name(an) # TRUE
mode(an)   # name
typeof(an) # symbol
```

names *The Names of an Object*

Description

Functions to get or set the names of an object.

Usage

```
names(x)
names(x) <- value
```

Arguments

x	an R object.
value	a character vector of up to the same length as x, or NULL.

Details

names is a generic accessor function, and names<- is a generic replacement function. The default methods get and set the "names" attribute of a vector (including a list) or pairlist.

For an environment env, names(env) gives the names of the corresponding list, i.e., names(as.list(env, all.names = TRUE)) which are also given by ls(env, all.names = TRUE, sorted = FALSE). If the environment is used as a hash table, names(env) are its "keys".

If value is shorter than x, it is extended by character NAs to the length of x.

It is possible to update just part of the names attribute via the general rules: see the examples. This works because the expression there is evaluated as z <- "names<-"(z, "[<-"(names(z), 3, "c2")).

The name "" is special: it is used to indicate that there is no name associated with an element of a (atomic or generic) vector. Subscripting by "" will match nothing (not even elements which have no name).

A name can be character NA, but such a name will never be matched and is likely to lead to confusion.

Both are primitive functions.

Value

For names, NULL or a character vector of the same length as x. (NULL is given if the object has no names, including for objects of types which cannot have names.) For an environment, the length is the number of objects in the environment but the order of the names is arbitrary.

For names<-, the updated object. (Note that the value of names(x) <- value is that of the assignment, value, not the return value from the left-hand side.)

Note

For vectors, the names are one of the attributes with restrictions on the possible values. For pairlists, the names are the tags and converted to and from a character vector.

For a one-dimensional array the `names` attribute really is `dimnames[[1]]`.

Formally classed aka "S4" objects typically have `slotNames()` (and no `names()`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`slotNames`, `dimnames`.

Examples

```
# print the names attribute of the islands data set
names(islands)

# remove the names attribute
names(islands) <- NULL
islands
rm(islands) # remove the copy made

z <- list(a = 1, b = "c", c = 1:3)
names(z)
# change just the name of the third element.
names(z)[3] <- "c2"
z

z <- 1:3
names(z)
## assign just one name
names(z)[2] <- "b"
z
```

---

nargs *The Number of Arguments to a Function*

Description

When used inside a function body, `nargs` returns the number of arguments supplied to that function, *including* positional arguments left blank.

Usage

```
nargs()
```

Details

The count includes empty (missing) arguments, so that `foo(x,,z)` will be considered to have three arguments (see 'Examples'). This can occur in rather indirect ways, so for example `x[]` might dispatch a call to `` `[.some_method`(x, ) `` which is considered to have two arguments.

This is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`args`, `formals` and `sys.call`.

Examples

```
tst <- function(a, b = 3, ...) {nargs()}
tst() # 0
tst(clicketyclack) # 1 (even non-existing)
tst(c1, a2, rr3) # 3

foo <- function(x, y, z, w) {
   cat("call was ", deparse(match.call()), "\n", sep = "")
   nargs()
}
foo()      # 0
foo(, , 3) # 3
foo(z = 3) # 1, even though this is the same call

nargs()  # not really meaningful
```

---

nchar *Count the Number of Characters (or Bytes or Width)*

Description

`nchar` takes a character vector as an argument and returns a vector whose elements contain the sizes of the corresponding elements of `x`. Internally, it is a generic, for which methods can be defined (see InternalMethods).

`nzchar` is a fast way to find out if elements of a character vector are non-empty strings.

Usage

```
nchar(x, type = "chars", allowNA = FALSE, keepNA = NA)

nzchar(x, keepNA = FALSE)
```

Arguments

x	character vector, or a vector to be coerced to a character vector. Giving a factor is an error.
type	character string: partial matching to one of `c("bytes", "chars", "width")`. See 'Details'.
allowNA	logical: should NA be returned for invalid multibyte strings or "bytes"-encoded strings (rather than throwing an error)?
keepNA	logical: should NA be returned when x is NA? If false, `nchar()` returns 2, as that is the number of printing characters used when strings are written to output, and `nzchar()` is TRUE. The default for `nchar()`, NA, means to use `keepNA = TRUE` unless type is "width".

Details

The 'size' of a character string can be measured in one of three ways (corresponding to the type argument):

bytes The number of bytes needed to store the string (plus in C a final terminator which is not counted).

chars The number of characters.

width The number of columns cat will use to print the string in a monospaced font. The same as chars if this cannot be calculated.

These will often be the same, and usually will be in single-byte locales (but note how type determines the default for keepNA). There will be differences between the first two with multibyte character sequences, e.g. in UTF-8 locales.

The internal equivalent of the default method of as.character is performed on x (so there is no method dispatch). If you want to operate on non-vector objects passing them through deparse first will be required.

Value

For nchar, an integer vector giving the sizes of each element. For missing values (i.e., NA, i.e., NA_character_), `nchar()` returns NA_integer_ if keepNA is true, and 2, the number of printing characters, if false.

`type = "width"` gives (an approximation to) the number of columns used in printing each element in a terminal font, taking into account double-width, zero-width and 'composing' characters. The approximation is likely to be poor when there are unassigned or non-printing characters.

If `allowNA = TRUE` and an element is detected as invalid in a multi-byte character set such as UTF-8, its number of characters and the width will be NA. Otherwise the number of characters will be non-negative, so `!is.na(nchar(x, "chars", TRUE))` is a test of validity.

A character string marked with "bytes" encoding (see Encoding) has a number of bytes, but neither a known number of characters nor a width, so the latter two types are NA if allowNA = TRUE, otherwise an error.

Names, dims and dimnames are copied from the input.

For nzchar, a logical vector of the same length as x, true if and only if the element has non-zero size; if the element is NA, nzchar() is true when keepNA is false (the default) or NA, and NA otherwise.

Note

This does **not** by default give the number of characters that will be used to print() the string. Use encodeString to find that. Where character strings have been marked as UTF-8, the number of characters and widths will be computed in UTF-8, even though printing may use escapes such as '<U+2642>' in a non-UTF-8 locale.

The concept of 'width' is a slippery one even in a monospaced font. Some human languages have the concept of *combining* characters, in which two or more characters are rendered together: an example would be "y\u306", which is two characters of width one: combining characters are given width zero, and there are other zero-width characters such as the zero-width space "\u200b".

Some East Asian languages have 'wide' characters, ideographs which are conventionally printed across two columns when mixed with ASCII and other 'narrow' characters in those languages. The problem is that whether a computer prints wide characters over two or one columns depends on the font, with it not being uncommon to use two columns in a font intended for East Asian users and a single column in a 'Western' font. Unicode has encodings for 'fullwidth' versions of ASCII characters and 'halfwidth' versions of Katakana (Japanese) and Hangul (Korean) characters. Then there is the 'East Asian Ambiguous class' (Greek, Cyrillic, signs, some accented Latin chars, etc), for which the historical practice was to use two columns in East Asia and one elsewhere. The width quoted by nchar for characters in that class (and some others) depends on the locale, being one except in some East Asian locales on some OSes (notably Windows).

Control characters are usually given width zero: this includes CR and LF. Computing the width of a string containing control characters should be avoided (and may depend on the OS and R version).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Unicode Standard Annex #11: *East Asian Width.* https://www.unicode.org/reports/tr11/

See Also

strwidth giving width of strings for plotting; paste, substr, strsplit

Examples

```
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
nchar(x)
# 5  6  6  1 15

nchar(deparse(mean))
# 18 17  <-- unless mean differs from base::mean
```

```
## NA behaviour as function of keepNA=* :
logi <- setNames(, c(FALSE, NA, TRUE))
sapply(logi, \(k) data.frame(nchar =  nchar (NA, keepNA=k),
                             nzchar = nzchar(NA, keepNA=k)))

x[3] <- NA; x
nchar(x, keepNA= TRUE) #  5  6 NA  1 15
nchar(x, keepNA=FALSE) #  5  6  2  1 15
stopifnot(identical(nchar(x     ), nchar(x, keepNA= TRUE)),
          identical(nchar(x, "w"), nchar(x, keepNA=FALSE)),
          identical(is.na(x), is.na(nchar(x))))

##' nchar() for all three types :
nchars <- function(x, ...)
   vapply(c("chars", "bytes", "width"),
          function(tp) nchar(x, tp, ...), integer(length(x)))

nchars("\u200b") # in R versions (>= 2015-09-xx):
## chars bytes width
##     1     3     0

data.frame(x, nchars(x)) ## all three types : same unless for NA
## force the same by forcing 'keepNA':
(ncT <- nchars(x, keepNA = TRUE)) ## .... NA NA NA ....
(ncF <- nchars(x, keepNA = FALSE))## ....  2  2  2 ....
stopifnot(apply(ncT, 1, function(.) length(unique(.))) == 1,
          apply(ncF, 1, function(.) length(unique(.))) == 1)
```

`nlevels` *The Number of Levels of a Factor*

Description

Return the number of levels which its argument has.

Usage

```
nlevels(x)
```

Arguments

`x`	an object, usually a factor.

Details

This is usually applied to a factor, but other objects can have levels.

The actual factor levels (if they exist) can be obtained with the `levels` function.

Value

The length of `levels(x)`, which is zero if `x` has no levels.

See Also

`levels`, `factor`.

Examples

```
nlevels(gl(3, 7)) # = 3
```

`noquote` *Class for 'no quote' Printing of Character Strings*

Description

Print character strings without quotes.

Usage

```
noquote(obj, right = FALSE)

## S3 method for class 'noquote'
print(x, quote = FALSE, right = FALSE, ...)

## S3 method for class 'noquote'
c(..., recursive = FALSE)
```

Arguments

`obj`	any R object, typically a vector of `character` strings.
`right`	optional `logical` eventually to be passed to `print()`, used by `print.default()`, indicating whether or not strings should be right aligned.
`x`	an object of class `"noquote"`.
`quote, ...`	further options passed to next methods, such as `print`.
`recursive`	for compatibility with the generic `c` function.

Details

`noquote` returns its argument as an object of class `"noquote"`. There is a method for `c()` and subscript method (`"[.noquote"`) which ensures that the class is not lost by subsetting. The print method (`print.noquote`) prints character strings *without* quotes (`"...."` is printed as `....`).

If `right` is specified in a call `print(x, right=*)`, it takes precedence over a possible `right` setting of `x`, e.g., created by `x <- noquote(*, right=TRUE)`.

These functions exist both as utilities and as an example of using (S3) `class` and object orientation.

Author(s)

Martin Maechler <maechler@stat.math.ethz.ch>

See Also

methods, class, print.

Examples

```
letters
nql <- noquote(letters)
nql
nql[1:4] <- "oh"
nql[1:12]

cmp.logical <- function(log.v)
{
  ## Purpose: compact printing of logicals
  log.v <- as.logical(log.v)
  noquote(if(length(log.v) == 0)"()" else c(".","|")[1 + log.v])
}
cmp.logical(stats::runif(20) > 0.8)

chmat <- as.matrix(format(stackloss)) # a "typical" character matrix
## noquote(*, right=TRUE)  so it prints exactly like a data frame
chmat <- noquote(chmat, right = TRUE)
chmat
```

---

norm *Compute the Norm of a Matrix*

Description

Computes a matrix norm of x using LAPACK. The norm can be the one ("O") norm, the infinity ("I") norm, the Frobenius ("F") norm, the maximum modulus ("M") among elements of a matrix, or the "spectral" or "2"-norm, as determined by the value of type.

Usage

```
norm(x, type = c("O", "I", "F", "M", "2"))
```

Arguments

x	numeric matrix; note that packages such as **Matrix** define more norm() methods.
type	character string, specifying the *type* of matrix norm to be computed. A character indicating the type of norm desired.

"O", "o" **or** "1" specifies the **o**ne norm, (maximum absolute column sum);

"I" **or** "i" specifies the **i**nfinity norm (maximum absolute row sum);

"F", "f", "E" **or** "e" specifies the **F**robenius norm (the **E**uclidean norm of x treated as if it were a vector);

"M" **or** "m" specifies the **m**aximum modulus of all the elements in x; and

"2" specifies the "spectral" or 2-norm, which is the largest singular value (svd) of x.

The default is "O". Only the first character of type[1] is used.

Details

The **base** method of norm() calls the LAPACK function dlange.

Note that the 1-, Inf- and "M" norm is faster to calculate than the Frobenius one.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The matrix norm, a non-negative number. Zero for a 0-extent (empty) matrix.

Source

Except for norm = "2", the LAPACK routine DLANGE.

LAPACK is from https://netlib.org/lapack/.

References

Anderson, E., *et al* (1994). *LAPACK User's Guide*, 2nd edition, SIAM, Philadelphia.

See Also

rcond for the (reciprocal) condition number.

Examples

```
(x1 <- cbind(1, 1:10))
norm(x1)
norm(x1, "I")
norm(x1, "M")
stopifnot(all.equal(norm(x1, "F"),
                    sqrt(sum(x1^2))))

hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, `+`) }
h9 <- hilbert(9)
## all 5 (4 different) types of norm:
(nTyp <- eval(formals(base::norm)$type))
sapply(nTyp, norm, x = h9)
stopifnot(exprs = { # 0-extent matrices:
    sapply(nTyp, norm, x = matrix(, 1,0)) == 0
```

```
    sapply(nTyp, norm, x = matrix(, 0,0)) == 0
})
```

`normalizePath` *Express File Paths in Canonical Form*

Description

Convert file paths to canonical form for the platform, to display them in a user-understandable form and so that relative and absolute paths can be compared.

Usage

```
normalizePath(path, winslash = "\\", mustWork = NA)
```

Arguments

`path`	character vector of file paths.
`winslash`	the separator to be used on Windows – ignored elsewhere. Must be one of `c("/", "\\")`.
`mustWork`	logical: if `TRUE` then an error is given if the result cannot be determined; if `NA` then a warning.

Details

Tilde-expansion (see `path.expand`) is first done on `paths`.

Where the Unix-alike platform supports it attempts to turn paths into absolute paths in their canonical form (no '`./`', '`../`' nor symbolic links). It relies on the POSIX system function `realpath`: if the platform does not have that (we know of no current example) then the result will be an absolute path but might not be canonical. Even where `realpath` is used the canonical path need not be unique, for example *via* hard links or multiple mounts.

On Windows it converts relative paths to absolute paths, resolves symbolic links, converts short names for path elements to long names and ensures the separator is that specified by `winslash`. It will match each path element case-insensitively or case-sensitively as during the usual name lookup and return the canonical case. It relies on Windows API function `GetFinalPathNameByHandle` and in case of an error (such as insufficient permissions) it currently falls back to the R 3.6 (and older) implementation, which relies on `GetFullPathName` and `GetLongPathName` with limitations described in the Notes section. An attempt is made not to introduce UNC paths in presence of mapped drives or symbolic links: if `GetFinalPathNameByHandle` returns a UNC path, but `GetLongPathName` returns a path starting with a drive letter, R falls back to the R 3.6 (and older) implementation. UTF-8-encoded paths not valid in the current locale can be used.

`mustWork = FALSE` is useful for expressing paths for use in messages.

Value

A character vector.

If an input is not a real path the result is system-dependent (unless `mustWork = TRUE`, when this should be an error). It will be either the corresponding input element or a transformation of it into an absolute path.

Converting to an absolute file path can fail for a large number of reasons. The most common are

- One of more components of the file path does not exist.
- A component before the last is not a directory, or there is insufficient permission to read the directory.
- For a relative path, the current directory cannot be determined.
- A symbolic link points to a non-existent place or links form a loop.
- The canonicalized path would be exceed the maximum supported length of a file path.

Note

The canonical form of paths may not be what you expect. For example, on macOS absolute paths such as '`/tmp`' and '`/var`' are symbolic links. On Linux, a path produced by bash process substitution is a symbolic link (such as '`/proc/fd/63`') to a pipe and there is no canonical form of such path. In R 3.6 and older on Windows, symlinks will not be resolved and the long names for path elements will be returned with the case in which they are in `path`, which may not be canonical in case-insensitive folders.

Examples

```
cat(normalizePath(c(R.home(), tempdir())), sep = "\n")
```

NotYet *Not Yet Implemented Functions and Unused Arguments*

Description

In order to pinpoint missing functionality, the R core team uses these functions for missing R functions and not yet used arguments of existing R functions (which are typically there for compatibility purposes).

You are very welcome to contribute your code ...

Usage

```
.NotYetImplemented()
.NotYetUsed(arg, error = TRUE)
```

Arguments

`arg`	an argument of a function that is not yet used.
`error`	a logical. If `TRUE`, an error is signalled; if `FALSE`; only a warning is given.

See Also

the contrary, Deprecated and Defunct for outdated code.

Examples

```
require(graphics)
barplot(1:5, inside = TRUE) # 'inside' is not yet used
```

---

nrow *The Number of Rows/Columns of an Array*

Description

nrow and ncol return the number of rows or columns present in x. NCOL and NROW do the same treating a vector as 1-column matrix, even a 0-length vector, compatibly with as.matrix() or cbind(), see the example.

Usage

```
nrow(x)
ncol(x)
NCOL(x)
NROW(x)
```

Arguments

x	a vector, array, data frame, or NULL.

Value

an integer of length 1 or NULL, the latter only for ncol and nrow.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (ncol and nrow.)

See Also

dim which returns *all* dimensions, and length which gives a number (a 'count') also in cases where dim() is NULL, and hence nrow() and ncol() return NULL; array, matrix.

Examples

```
ma <- matrix(1:12, 3, 4)
nrow(ma)   # 3
ncol(ma)   # 4

ncol(array(1:24, dim = 2:4)) # 3, the second dimension
NCOL(1:12) # 1
NROW(1:12) # 12, the length() of the vector

## as.matrix() produces 1-column matrices from 0-length vectors,
## and so does cbind() :
dim(as.matrix(numeric())) # 0 1
dim(    cbind(numeric())) # ditto
NCOL(numeric()) # 1
## However, as.matrix(NULL) fails and cbind(NULL) gives NULL, hence for
## consistency:
NCOL(NULL)      # 0
## (This gave 1 in R < 4.4.0.)
```

---

ns-dblcolon *Double Colon and Triple Colon Operators*

Description

Accessing exported and internal variables, i.e. R objects (including lazy loaded data sets) in a namespace.

Usage

```
pkg::name
pkg:::name
```

Arguments

pkg	package name: symbol or literal character string.
name	variable name: symbol or literal character string.

Details

For a package **pkg**, pkg::name returns the value of the exported variable name in namespace pkg, whereas pkg:::name returns the value of the internal variable name. The package namespace will be loaded if it was not loaded before the call, but the package will not be attached to the search path.

Specifying a variable or package that does not exist is an error.

Note that pkg::name does **not** access the objects in the environment package:pkg (which does not exist until the package's namespace is attached): the latter may contain objects not exported from the namespace. It can access datasets made available by lazy-loading.

Note

It is typically a design mistake to use `:::` in your code since the corresponding object has probably been kept internal for a good reason. Consider contacting the package `maintainer` if you feel the need to access the object for anything but mere inspection.

See Also

`get` to access an object masked by another of the same name. `loadNamespace`, `asNamespace` for more about namespaces.

Examples

```
base::log
base::"+"

## Beware --  use ':::' at your own risk! (see "Details")
stats:::coef.default
```

---

ns-hooks *Hooks for Namespace Events*

Description

Packages can supply functions to be called when loaded, attached, detached or unloaded.

Usage

```
.onLoad(libname, pkgname)
.onAttach(libname, pkgname)
.onUnload(libpath)
.onDetach(libpath)
.Last.lib(libpath)
```

Arguments

`libname`	a character string giving the library directory where the package defining the namespace was found.
`pkgname`	a character string giving the name of the package.
`libpath`	a character string giving the complete path to the package.

Details

After loading, `loadNamespace` looks for a hook function named `.onLoad` and calls it (with two unnamed arguments) before sealing the namespace and processing exports.

When the package is attached (via `library` or `attachNamespace`), the hook function `.onAttach` is looked for and if found is called (with two unnamed arguments) before the package environment is sealed.

If a function .onDetach is in the namespace or .Last.lib is exported from the package, it will be called (with a single argument) when the package is detached. Beware that it might be called if .onAttach has failed, so it should be written defensively. (It is called within tryCatch, so errors will not stop the package being detached.)

If a namespace is unloaded (via unloadNamespace), a hook function .onUnload is run (with a single argument) before final unloading.

Note that the code in .onLoad and .onUnload should not assume any package except the base package is on the search path. Objects in the current package will be visible (unless this is circumvented), but objects from other packages should be imported or the double colon operator should be used.

.onLoad, .onUnload, .onAttach and .onDetach are looked for as internal objects in the namespace and should not be exported (whereas .Last.lib should be).

Note that packages are not detached nor namespaces unloaded at the end of an R session unless the user arranges to do so (e.g., *via* .Last).

Anything needed for the functioning of the namespace should be handled at load/unload times by the .onLoad and .onUnload hooks. For example, DLLs can be loaded (unless done by a useDynLib directive in the 'NAMESPACE' file) and initialized in .onLoad and unloaded in .onUnload. Use .onAttach only for actions that are needed only when the package becomes visible to the user (for example a start-up message) or need to be run after the package environment has been created.

Good practice

Loading a namespace should where possible be silent, with startup messages given by .onAttach. These messages (and any essential ones from .onLoad) should use packageStartupMessage so they can be silenced where they would be a distraction.

There should be no calls to library nor require in these hooks. The way for a package to load other packages is via the 'Depends' field in the 'DESCRIPTION' file: this ensures that the dependence is documented and packages are loaded in the correct order. Loading a namespace should not change the search path, so rather than attach a package, dependence of a namespace on another package should be achieved by (selectively) importing from the other package's namespace.

Uses of library with argument help to display basic information about the package should use format on the computed package information object and pass this to packageStartupMessage.

There should be no calls to installed.packages in startup code: it is potentially very slow and may fail in versions of R before 2.14.2 if package installation is going on in parallel. See its help page for alternatives.

Compiled code should be loaded (e.g., *via* library.dynam) in .onLoad or a useDynLib directive in the 'NAMESPACE' file, and not in .onAttach. Similarly, compiled code should not be unloaded (e.g., *via* library.dynam.unload) in .Last.lib nor .onDetach, only in .onUnload.

See Also

setHook shows how users can set hooks on the same events, and lists the sequence of events involving all of the hooks.

reg.finalizer for hooks to be run at the end of a session.

loadNamespace for more about namespaces.

ns-load *Loading and Unloading Name Spaces*

Description

Functions to load and unload name spaces.

Usage

```
attachNamespace(ns, pos = 2L, depends = NULL, exclude, include.only)
loadNamespace(package, lib.loc = NULL,
              keep.source = getOption("keep.source.pkgs"),
              partial = FALSE, versionCheck = NULL,
              keep.parse.data = getOption("keep.parse.data.pkgs"))
requireNamespace(package, ..., quietly = FALSE)
loadedNamespaces()
unloadNamespace(ns)
isNamespaceLoaded(name)
```

Arguments

ns	string or name space object.
pos	integer specifying position to attach.
depends	NULL or a character vector of dependencies to be recorded in object .Depends in the package.
package	string naming the package/name space to load.
lib.loc	character vector specifying library search path (the location of R library trees to search through.
keep.source	now ignored except during package installation.
keep.parse.data	ignored except during package installation.
partial	logical; if true, stop just after loading code.
versionCheck	NULL or a version specification (a list with components op and version).
quietly	logical: should progress and error messages be suppressed?
name	string or 'name', see as.symbol, of a package, e.g., "stats".
exclude, include.only	character vectors; see library.
...	further arguments to be passed to loadNamespace.

Details

The functions loadNamespace and attachNamespace are usually called implicitly when library is used to load a name space and any imports needed. However it may be useful at times to call these functions directly.

loadNamespace loads the specified name space and registers it in an internal data base. A request to load a name space when one of that name is already loaded has no effect. The arguments have the same meaning as the corresponding arguments to library, whose help page explains the details of how a particular installed package comes to be chosen. After loading, loadNamespace looks for a hook function named .onLoad as an internal variable in the name space (it should not be exported). Partial loading is used to support installation with lazy-loading.

Optionally the package licence is checked during loading: see section 'Licenses' in the help for library.

loadNamespace does not attach the name space it loads to the search path. attachNamespace can be used to attach a frame containing the exported values of a name space to the search path (but this is almost always done *via* library). The hook function .onAttach is run after the name space exports are attached.

requireNamespace is a wrapper for loadNamespace analogous to require that returns a logical value.

loadedNamespaces returns a character vector of the names of the loaded name spaces.

isNamespaceLoaded(pkg) is equivalent to but more efficient than pkg %in% loadedNamespaces().

unloadNamespace can be used to attempt to force a name space to be unloaded. If the name space is attached, it is first detached, thereby running a .onDetach or .Last.lib function in the name space if one is exported. An error is signaled and the name space is not unloaded if the name space is imported by other loaded name spaces. If defined, a hook function .onUnload is run before removing the name space from the internal registry.

See the comments in the help for detach about some issues with unloading and reloading name spaces.

Value

attachNamespace returns invisibly the package environment it adds to the search path.

loadNamespace returns the name space environment, either one already loaded or the one the function causes to be loaded.

requireNamespace returns TRUE if it succeeds or FALSE.

loadedNamespaces returns a character vector.

unloadNamespace returns NULL, invisibly.

Tracing

As from R 4.1.0 the operation of loadNamespace can be traced, which can help track down the causes of unexpected messages (including which package(s) they come from since loadNamespace is called in many ways including from itself and by :: and can be called by load). Setting the environment variable _R_TRACE_LOADNAMESPACE_ to a numerical value will generate additional messages on progress. Non-zero values, e.g. 1, report which namespace is being loaded and when

loading completes: values 2 to 4 report in increasing detail. Negative values are reserved for tracing specific features and their current meanings are documented in source-code comments.

Loading standard packages is never traced.

Author(s)

Luke Tierney and R-core

References

The 'Writing R Extensions' manual, section "Package namespaces".

See Also

`getNamespace`, `asNamespace`, `topenv`, `.onLoad` (etc); further `environment`.

Examples

```
(lns <- loadedNamespaces())
statL <- isNamespaceLoaded("stats")
stopifnot( identical(statL, "stats" %in% lns) )

## The string "foo" and the symbol 'foo' can be used interchangably here:
stopifnot( identical(isNamespaceLoaded(  "foo"   ), FALSE),
           identical(isNamespaceLoaded(quote(foo)), FALSE),
           identical(isNamespaceLoaded(quote(stats)), statL))

hasS <- isNamespaceLoaded("splines") # (to restore if needed)
Sns <- asNamespace("splines") # loads it if not already
stopifnot(   isNamespaceLoaded("splines"))
if (is.null(try(unloadNamespace(Sns)))) # try unloading the NS 'object'
stopifnot( ! isNamespaceLoaded("splines"))
if (hasS) loadNamespace("splines") # (restoring previous state)
```

`ns-topenv` *Top Level Environment*

Description

Finding the top level `environment` from an environment `envir` and its enclosing environments.

Usage

```
topenv(envir = parent.frame(),
       matchThisEnv = getOption("topLevelEnvironment"))
```

Arguments

envir	environment.
matchThisEnv	return this environment, if it matches before any other criterion is satisfied. The default, the option 'topLevelEnvironment', is set by sys.source, which treats a specific environment as the top level environment. Supplying the argument as NULL or emptyenv() means it will never match.

Details

topenv returns the first top level environment found when searching envir and its enclosing environments. If no top level environment is found, .GlobalEnv is returned. An environment is considered top level if it is the internal environment of a namespace, a package environment in the search path, or .GlobalEnv .

See Also

environment, notably parent.env() on "enclosing environments"; loadNamespace for more on namespaces.

Examples

```
topenv(.GlobalEnv)
topenv(new.env()) # also global env
topenv(environment(ls))# namespace:base
topenv(environment(lm))# namespace:stats
```

NULL *The Null Object*

Description

NULL represents the null object in R: it is a reserved word. NULL is often returned by expressions and functions whose value is undefined.

Usage

```
NULL
as.null(x, ...)
is.null(x)
```

Arguments

x	an object to be tested or coerced.
...	ignored.

Details

NULL can be indexed (see Extract) in just about any syntactically legal way: apart from NULL[[]] which is an error, the result is always NULL. Objects with value NULL can be changed by replacement operators and will be coerced to the type of the right-hand side.

NULL is also used as the empty pairlist: see the examples. Because pairlists are often promoted to lists, you may encounter NULL being promoted to an empty list.

Objects with value NULL cannot have attributes as there is only one null object: attempts to assign them are either an error (attr) or promote the object to an empty list with attribute(s) (attributes and structure).

Value

as.null ignores its argument and returns NULL.

is.null returns TRUE if its argument's value is NULL and FALSE otherwise.

Note

is.null is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

%||%: L %||% R is equivalent to if(!is.null(L)) L else R

Examples

```
is.null(list())     # FALSE (on purpose!)
is.null(pairlist()) # TRUE
is.null(integer(0)) # FALSE
is.null(logical(0)) # FALSE
as.null(list(a = 1, b = "c"))
```

---

numeric *Numeric Vectors*

Description

Creates or coerces objects of type "numeric". is.numeric is a more general test of an object being interpretable as numbers.

Usage

```
numeric(length = 0)
as.numeric(x, ...)
is.numeric(x)
```

Arguments

length	a non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

numeric is identical to double. It creates a double-precision vector of the specified length with each element equal to 0.

as.numeric is a generic function, but S3 methods must be written for as.double. It is identical to as.double.

is.numeric is an internal generic primitive function: you can write methods to handle specific classes of objects, see InternalMethods. It is **not** the same as is.double. Factors are handled by the default method, and there are methods for classes "Date", "POSIXt" and "difftime" (all of which return false). Methods for is.numeric should only return true if the base type of the class is double or integer *and* values can reasonably be regarded as numeric (e.g., arithmetic on them makes sense, and comparison should be done via the base type).

Value

for numeric and as.numeric see double.

The default method for is.numeric returns TRUE if its argument is of mode "numeric" (type "double" or type "integer") and not a factor, and FALSE otherwise. That is, is.integer(x) || is.double(x), or (mode(x) == "numeric") && !is.factor(x).

Warning

If x is a factor, as.numeric will return the underlying numeric (integer) representation, which is often meaningless as it may not correspond to the factor levels, see the 'Warning' section in factor (and the 2nd example below).

S4 methods

as.numeric and is.numeric are internally S4 generic and so methods can be set for them *via* setMethod.

To ensure that as.numeric and as.double remain identical, S4 methods can only be set for as.numeric.

Note on names

It is a historical anomaly that R has two names for its floating-point vectors, double and numeric (and formerly had real).

double is the name of the type. numeric is the name of the mode and also of the implicit class. As an S4 formal class, use "numeric".

The potential confusion is that R has used *mode* "numeric" to mean 'double or integer', which conflicts with the S4 usage. Thus is.numeric tests the mode, not the class, but as.numeric (which is identical to as.double) coerces to the class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

double, integer, storage.mode.

Examples

```
## Conversion does trim whitespace; non-numeric strings give NA + warning
as.numeric(c("-.1"," 2.7 ","B"))

## Numeric values are sometimes accidentally converted to factors.
## Converting them back to numeric is trickier than you'd expect.
f <- factor(5:10)
as.numeric(f) # not what you might expect, probably not what you want
## what you typically meant and want:
as.numeric(as.character(f))
## the same, considerably more efficient (for long vectors):
as.numeric(levels(f))[f]
```

NumericConstants *Numeric Constants*

Description

How R parses numeric constants.

Details

R parses numeric constants in its input in a very similar way to C99 floating-point constants.

Inf and NaN are numeric constants (with typeof(.) "double"). In text input (e.g., in scan and as.double), these are recognized ignoring case as is infinity as an alternative to Inf. NA_real_ and NA_integer_ are constants of types "double" and "integer" representing missing values. All other numeric constants start with a digit or period and are either a decimal or hexadecimal constant optionally followed by L.

Hexadecimal constants start with `0x` or `0X` followed by a non-empty sequence from `0-9` `a-f` `A-F` `.` which is interpreted as a hexadecimal number, optionally followed by a binary exponent. A binary exponent consists of a `P` or `p` followed by an optional plus or minus sign followed by a non-empty sequence of (decimal) digits, and indicates multiplication by a power of two. Thus `0x123p456` is 291×2^{456}.

Decimal constants consist of a non-empty sequence of digits possibly containing a period (the decimal point), optionally followed by a decimal exponent. A decimal exponent consists of an `E` or `e` followed by an optional plus or minus sign followed by a non-empty sequence of digits, and indicates multiplication by a power of ten.

Values which are too large or too small to be representable will overflow to `Inf` or underflow to `0.0`.

A numeric constant immediately followed by `i` is regarded as an imaginary complex number.

A numeric constant immediately followed by `L` is regarded as an `integer` number when possible (and with a warning if it contains a `"."`).

Only the ASCII digits 0–9 are recognized as digits, even in languages which have other representations of digits. The 'decimal separator' is always a period and never a comma.

Note that a leading plus or minus is not regarded by the parser as part of a numeric constant but as a unary operator applied to the constant.

Note

When a string is parsed to input a numeric constant, the number may or may not be representable exactly in the C double type used. If not one of the nearest representable numbers will be returned.

R's own C code is used to convert constants to binary numbers, so the effect can be expected to be the same on all platforms implementing full IEC 60559 arithmetic (the most likely area of difference being the handling of numbers less than `.Machine$double.xmin`). The same code is used by `scan`.

See Also

`Syntax`. For complex numbers, see `complex`. `Quotes` for the parsing of character constants, `Reserved` for the "reserved words" in R.

Examples

```
## You can create numbers using fixed or scientific formatting.
2.1
2.1e10
-2.1E-10

## The resulting objects have class numeric and type double.
class(2.1)
typeof(2.1)

## This holds even if what you typed looked like an integer.
class(2)
typeof(2)

## If you actually wanted integers, use an "L" suffix.
```

```
class(2L)
typeof(2L)

## These are equal but not identical
2 == 2L
identical(2, 2L)

## You can write numbers between 0 and 1 without a leading "0"
## (but typically this makes code harder to read)
.1234

sqrt(1i) # remember elementary math?
utils::str(0xA0)
identical(1L, as.integer(1))

## You can combine the "0x" prefix with the "L" suffix :
identical(0xFL, as.integer(15))
```

`numeric_version` *Numeric Versions*

Description

A simple S3 class for representing numeric versions including package versions, and associated methods.

Usage

```
numeric_version(x, strict = TRUE)
package_version(x, strict = TRUE)
R_system_version(x, strict = TRUE)
getRversion()
as.numeric_version(x)
as.package_version(x)
is.numeric_version(x)
is.package_version(x)
```

Arguments

`x`	for the creators, a character vector with suitable numeric version strings (see 'Details'); for `package_version`, alternatively an R version object as obtained by `R.version`. For `as.numeric_version` and `as.package_version`, suitable character vectors as above, or numeric version objects. For `is.numeric_version` and `is.package_version`, arbitrary R objects.
`strict`	a logical indicating whether invalid numeric versions should result in an error (default) or not.

Details

Numeric versions are sequences of one or more non-negative integers, usually (e.g., in package 'DESCRIPTION' files) represented as character strings with the elements of the sequence concatenated and separated by single '.' or '-' characters. R package versions consist of at least two such integers, an R system version of exactly three (major, minor and patch level).

Functions `numeric_version`, `package_version` and `R_system_version` create a representation from such strings (if suitable) which allows for coercion and testing, combination, comparison, summaries (min/max), inclusion in data frames, subscripting, and printing. The classes can hold a vector of such representations.

`getRversion` returns the version of the running R as an R system version object.

The `[[` operator extracts or replaces a single version. To access the integers of a version use two indices: see the examples.

See Also

`compareVersion`; `packageVersion` for the version of a specific R package. `R.version` etc for the version of R (and the information underlying `getRversion()`).

Examples

```
x <- package_version(c("1.2-4", "1.2-3", "2.1"))
x < "1.4-2.3"
c(min(x), max(x))
x[2, 2]
x$major
x$minor

if(getRversion() <= "2.5.0") { ## work around missing feature
  cat("Your version of R, ", as.character(getRversion()),
      ", is outdated.\n",
      "Now trying to work around that ...\n", sep = "")
}

x[[1]]
x[[c(1, 3)]]  # '4' as a numeric version
x[1, 3]       # same
x[[1, 3]]     # 4 as an integer

x[[2, 3]] <- 0    # zero the patchlevel
x[[c(2, 3)]] <- 0 # same
x

x[[3]] <- "2.2.3"
x

x <- c(x, package_version("0.0"))
is.na(x)[4] <- TRUE
stopifnot(identical(is.na(x), c(rep(FALSE,3), TRUE)),
   anyNA(x))
```

octmode *Integer Numbers Displayed in Octal*

Description

Integers which are displayed in octal (base-8 number system) format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Arithmetic works as for integers, and non-integer valued mathematical functions typically work by truncating the result to integer.

Usage

```
as.octmode(x)

## S3 method for class 'octmode'
as.character(x, keepStr = FALSE, ...)

## S3 method for class 'octmode'
format(x, width = NULL, ...)

## S3 method for class 'octmode'
print(x, ...)
```

Arguments

x	an object, for the methods inheriting from class "octmode".
keepStr	a logical indicating that names and dimensions should be kept; set TRUE for back compatibility, if needed.
width	NULL or a positive integer specifying the minimum field width to be used, with padding by leading zeroes.
...	further arguments passed to or from other methods.

Details

"octmode" objects are integer vectors with that class attribute, used primarily to ensure that they are printed in octal notation, specifically for Unix-like file permissions such as 755. Subsetting ([) works too, as do arithmetic or other mathematical operations, albeit truncated to integer.

as.character(x) drops all attributes (unless when keepStr=TRUE where it keeps, dim, dimnames and names for back compatibility) and converts each entry individually, hence with no leading zeroes, whereas in format(), when width = NULL (the default), the output is padded with leading zeroes to the smallest width needed for all the non-missing elements.

as.octmode can convert integers (of type "integer" or "double") and character vectors whose elements contain only digits 0-7 (or are NA) to class "octmode".

There is a ! method and methods for | and &: these recycle their arguments to the length of the longer and then apply the operators bitwise to each element.

See Also

These are auxiliary functions for `file.info`.

`hexmode`, `sprintf` for other options in converting integers to octal, `strtoi` to convert octal strings to integers.

Examples

```
(on <- as.octmode(c(16, 32, 127:129))) # "020" "040" "177" "200" "201"
unclass(on[3:4]) # subsetting

## manipulate file modes
fmode <- as.octmode("170")
(fmode | "644") & "755"

(umask <- Sys.umask()) # depends on platform
c(fmode, "666", "755") & !umask


om <- as.octmode(1:12)
om # print()s via format()
stopifnot(nchar(format(om)) == 2)
om[1:7] # *no* leading zeroes!
stopifnot(format(om[1:7]) == as.character(1:7))
om2 <- as.octmode(c(1:10, 60:70))
om2 # prints via format() -> with 3 octals
stopifnot(nchar(format(om2)) == 3)
as.character(om2) # strings of length 1, 2, 3


## Integer arithmetic (remaining "octmode"):
om^2
om * 64
-om
(fac <- factorial(om)) # !1, !2, !3, !4 .. in hexadecimals
as.integer(fac) # indeed the same as  factorial(1:12)
```

`on.exit` *Function Exit Code*

Description

`on.exit` records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error). This is useful for resetting graphical parameters or performing other cleanup actions.

If no expression is provided, i.e., the call is `on.exit()`, then the current `on.exit` code is removed.

Usage

```
on.exit(expr = NULL, add = FALSE, after = TRUE)
```

Arguments

expr	an expression to be executed.
add	if TRUE, add expr to be executed after any previously set expressions (or before if after is FALSE); otherwise (the default) expr will overwrite any previously set expressions.
after	if add is TRUE and after is FALSE, then expr will be added on top of the expressions that were already registered. The resulting last in first out order is useful for freeing or closing resources in reverse order.

Details

The expr argument passed to on.exit is recorded without evaluation. If it is not subsequently removed/replaced by another on.exit call in the same function, it is evaluated in the evaluation frame of the function when it exits (including during standard error handling). Thus any functions or variables in the expression will be looked for in the function and its environment at the time of exit: to capture the current value in expr use substitute or similar.

If multiple on.exit expressions are set using add = TRUE then all expressions will be run even if one signals an error.

This is a 'special' primitive function: it only evaluates the arguments add and after.

Value

Invisible NULL.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

sys.on.exit which returns the expression stored for use by on.exit() in the function in which sys.on.exit() is evaluated.

Examples

```
require(graphics)

opar <- par(mai = c(1,1,1,1))
on.exit(par(opar))
```

Ops.Date *Operators on the Date Class*

Description

Operators for the "Date" class.

There is an Ops method and specific methods for + and - for the Date class.

Usage

```
date + x
x + date
date - x
date1 lop date2
```

Arguments

date	an object of class "Date".
date1, date2	date objects or character vectors. (Character vectors are converted by as.Date.)
x	a numeric vector (in days) *or* an object of class "difftime", rounded to the nearest whole day.
lop	one of ==, !=, <, <=, > or >=.

Details

x does not need to be integer if specified as a numeric vector, but see the comments about fractional days in the help for Dates.

Examples

```
(z <- Sys.Date())
z + 10
z < c("2009-06-01", "2010-01-01", "2015-01-01")
```

options *Options Settings*

Description

Allow the user to set and examine a variety of global *options* which affect the way in which R computes and displays its results.

Usage

```
options(...)

getOption(x, default = NULL)

.Options
```

Arguments

...	any options can be defined, using name = value. However, only the ones below are used in base R. Options can also be passed by giving a single unnamed argument which is a named list.
x	a character string holding an option name.
default	if the specified option is not set in the options list, this value is returned. This facilitates retrieving an option and checking whether it is set and setting it separately if not.

Details

Invoking options() with no arguments returns a list with the current values of the options. Note that not all options listed below are set initially. To access the value of a single option, one should use, e.g., getOption("width") rather than options("width") which is a *list* of length one.

Value

For getOption, the current value set for option x, or default (which defaults to NULL) if the option is unset.

For options(), a list of all set options sorted by name. For options(name), a list of length one containing the set value, or NULL if it is unset. For uses setting one or more options, a list with the previous values of the options changed (returned invisibly).

Options used in base R

add.smooth: typically logical, defaulting to TRUE. Could also be set to an integer for specifying how many (simulated) smooths should be added. This is currently only used by plot.lm.

askYesNo: a function (typically set by a front-end) to ask the user binary response functions in a consistent way, or a vector of strings used by askYesNo to use as default responses for such questions.

browserNLdisabled: logical: whether newline is disabled as a synonym for "n" in the browser.

catch.script.errors: logical, false by default. If true *and* interactive() is false, e.g., when an R script is run by R CMD BATCH <script>.R, then errors do *not* stop execution of the script. Rather evaluation continues after printing the error (and jumping to top level). Also, traceback() would provide info about the error. Do use with care!

checkPackageLicense: logical, not set by default. If true, loadNamespace asks a user to accept any non-standard license at first load of the package.

`check.bounds`: logical, defaulting to `FALSE`. If true, a warning is produced whenever a vector (atomic or `list`) is extended, by something like `x <- 1:3; x[5] <- 6`.

`CBoundsCheck`: logical, controlling whether `.C` and `.Fortran` make copies to check for array overruns on the atomic vector arguments.

Initially set from value of the environment variable `R_C_BOUNDS_CHECK` (set to `yes` to enable).

`conflicts.policy`: character string or list controlling handling of conflicts found in calls to `library` or `require`. See `library` for details.

`continue`: a non-empty string setting the prompt used for lines which continue over one line.

`defaultPackages`: the packages that are attached by default when R starts up. Initially set from the value of the environment variable `R_DEFAULT_PACKAGES`, or if that is unset to `c("datasets", "utils", "grDevices", "graphics", "stats", "methods")`. (Set `R_DEFAULT_PACKAGES` to `NULL` or a comma-separated list of package names.) This option can be changed in a '`.Rprofile`' file, but it will not work to exclude the **methods** package at this stage, as the value is screened for **methods** before that file is read.

`deparse.cutoff`: integer value controlling the printing of language constructs which are `deparse`d. Default `60`.

`deparse.max.lines`: controls the number of lines used when deparsing in `browser`, upon entry to a function whose debugging flag is set, and if option `traceback.max.lines` is unset, of `traceback()`. Initially unset, and only used if set to a positive integer.

`traceback.max.lines`: controls the number of lines used when deparsing in `traceback`, if set. Initially unset, and only used if set to a positive integer.

`digits`: controls the number of *significant* (see `signif`) digits to print when printing numeric values. It is a suggestion only. Valid values are 1...22 with default 7. See the note in `print.default` about values greater than 15.

`digits.secs`: controls the maximum number of digits to print when formatting time values in seconds. Valid values are 0...6 with default 0 (equivalent to `NULL` which is used when it is undefined as on vanilla startup). See `strftime`.

`download.file.extra`: Extra command-line argument(s) for non-default methods: see `download.file`.

`download.file.method`: Method to be used for `download.file`. Currently download methods `"internal"`, `"wininet"` (Windows only), `"libcurl"`, `"wget"` and `"curl"` are available. If not set, `method = "auto"` is chosen: see `download.file`.

`echo`: logical. Only used in non-interactive mode, when it controls whether input is echoed. Command-line option '`--no-echo`' sets this to `FALSE`, but otherwise it starts the session as `TRUE`.

`encoding`: The name of an encoding, default `"native.enc"`. See `connections`.

`error`: either a function or an expression governing the handling of non-catastrophic errors such as those generated by `stop` as well as by signals and internally detected errors. If the option is a function, a call to that function, with no arguments, is generated as the expression. By default the option is not set: see `stop` for the behaviour in that case. The functions `dump.frames` and `recover` provide alternatives that allow post-mortem debugging. Note that these need to specified as e.g. `options(error = utils::recover)` in startup files such as '`.Rprofile`'.

`expressions`: sets a limit on the number of nested expressions that will be evaluated. Valid values are 25...500000 with default 5000. If you increase it, you may also want to start R with a

larger protection stack; see '--max-ppsize' in Memory. Note too that you may cause a segfault from overflow of the C stack, and on OSes where it is possible you may want to increase that. Once the limit is reached an error is thrown. The current number under evaluation can be found by calling Cstack_info.

interrupt: a function taking no arguments to be called on a user interrupt if the interrupt condition is not otherwise handled.

keep.parse.data: When internally storing source code (keep.source is TRUE), also store parse data. Parse data can then be retrieved with getParseData() and used e.g. for spell checking of string constants or syntax highlighting. The value has effect only when internally storing source code (see keep.source). The default is TRUE.

keep.parse.data.pkgs: As for keep.parse.data, used only when packages are installed. Defaults to FALSE unless the environment variable R_KEEP_PKG_PARSE_DATA is set to yes. The space overhead of parse data can be substantial even after compression and it causes performance overhead when loading packages.

keep.source: When TRUE, the source code for functions (newly defined or loaded) is stored internally allowing comments to be kept in the right places. Retrieve the source by printing or using deparse(fn, control = "useSource").

The default is interactive(), i.e., TRUE for interactive use.

keep.source.pkgs: As for keep.source, used only when packages are installed. Defaults to FALSE unless the environment variable R_KEEP_PKG_SOURCE is set to yes.

matprod: a string selecting the implementation of the matrix products %*%, crossprod, and tcrossprod for double and complex vectors:

"internal" uses an unoptimized 3-loop algorithm which correctly propagates NaN and Inf values and is consistent in precision with other summation algorithms inside R like sum or colSums (which now means that it uses a long double accumulator for summation if available and enabled, see capabilities).

"default" uses BLAS to speed up computation, but to ensure correct propagation of NaN and Inf values it uses an unoptimized 3-loop algorithm for inputs that may contain NaN or Inf values. When deemed beneficial for performance, "default" may call the 3-loop algorithm unconditionally, i.e., without checking the input for NaN/Inf values. The 3-loop algorithm uses (only) a double accumulator for summation, which is consistent with the reference BLAS implementation.

"blas" uses BLAS unconditionally without any checks and should be used with extreme caution. BLAS libraries do not propagate NaN or Inf values correctly and for inputs with NaN/Inf values the results may be undefined.

"default.simd" is experimental and will likely be removed in future versions of R. It provides the same behavior as "default", but the check whether the input contains NaN/Inf values is faster on some SIMD hardware. On older systems it will run correctly, but may be much slower than "default".

max.print: integer, defaulting to 99999. print or show methods can make use of this option, to limit the amount of information that is printed, to something in the order of (and typically slightly less than) max.print *entries*.

OutDec: character string containing a single character. The preferred character to be used as the decimal point in output conversions, that is in printing, plotting, format, formatC and as.character but not when deparsing nor by sprintf (which is sometimes used prior to printing).

`pager`: the command used for displaying text files by `file.show`, details depending on the platform:

On a unix-alike defaults to '*R_HOME*`/bin/pager`', which is a shell script running the command-line specified by the environment variable `PAGER` whose default is set at configuration, usually to `less`.

On Windows defaults to `"internal"`, which uses a pager similar to the GUI console. Another possibility is `"console"` to use the console itself.

Can be a character string or an R function, in which case it needs to accept the arguments `(files, header, title, delete.file)` corresponding to the first four arguments of `file.show`.

`papersize`: the default paper format used by `postscript`; set by environment variable `R_PAPERSIZE` when R is started: if that is unset or invalid it defaults platform dependently

on a unix-alike to a value derived from the locale category `LC_PAPER`, or if that is unavailable to a default set when R was built.

on Windows to `"a4"`, or `"letter"` in US and Canadian locales.

`PCRE_limit_recursion`: Logical: should `grep(perl = TRUE)` and similar limit the maximal recursion allowed when matching? Only relevant for PCRE1 and PCRE2 <= 10.23.

PCRE can be built not to use a recursion stack (see `pcre_config`), but it uses recursion by default with a recursion limit of 10000000 which potentially needs a very large C stack: see the discussion at `https://www.pcre.org/original/doc/html/pcrestack.html`. If true, the limit is reduced using R's estimate of the C stack size available (if known), otherwise 10000. If `NA`, the limit is imposed only if any input string has 1000 or more bytes. The limit has no effect when PCRE's Just-in-Time compiler is used.

`PCRE_study`: Logical or integer: should `grep(perl = TRUE)` and similar 'study' the patterns? Either logical or a numerical threshold for the minimum number of strings to be matched for the pattern to be studied (the default is `10`)). Missing values and negative numbers are treated as false. This option is ignored with PCRE2 (PCRE version >= 10.00) which does not have a separate study phase and patterns are automatically optimized when possible.

`PCRE_use_JIT`: Logical: should `grep(perl = TRUE)`, `strsplit(perl = TRUE)` and similar make use of PCRE's Just-In-Time compiler if available? (This applies only to studied patterns with PCRE1.) Default: true. Missing values are treated as false.

`pdfviewer`: default PDF viewer. The default is set from the environment variable `R_PDFVIEWER`, the default value of which

on a unix-alike is set when R is configured, and

on Windows is the full path to `open.exe`, a utility supplied with R.

`printcmd`: the command used by `postscript` for printing; set by environment variable `R_PRINTCMD` when R is started. This should be a command that expects either input to be piped to '`stdin`' or to be given a single filename argument. Usually set to `"lpr"` on a Unix-alike.

`prompt`: a non-empty string to be used for R's prompt; should usually end in a blank (`" "`).

`rl_word_breaks`: (Unix only:) Used for the readline-based terminal interface. Default value `" \t\n\"\\'`><=%;,|&{()}"`.

This is the set of characters use to break the input line into tokens for object- and file-name completion. Those who do not use spaces around operators may prefer
`" \t\n\"\\'`><=+-*%;,|&{()}"`

`save.defaults`, `save.image.defaults`: see `save`.

`scipen`: integer. A penalty to be applied when deciding to print numeric values in fixed or exponential notation. Positive values bias towards fixed and negative towards scientific notation: fixed notation will be preferred unless it is more than `scipen` digits wider.

`setWidthOnResize`: a logical. If set and `TRUE`, R run in a terminal using a recent `readline` library will set the `width` option when the terminal is resized.

`showWarnCalls`, `showErrorCalls`: a logical. Should warning and error messages produced by the default handlers show a summary of the call stack? By default error call stacks are shown in non-interactive sessions. When `warning` or `stop` are called on a condition object the call stacks are only shown if the value returned by `conditionCall` for the condition object is not `NULL`.

`showNCalls`: integer. Controls how long the sequence of calls must be (in bytes) before ellipses are used. Defaults to 50 and should be at least 30 and no more than 500.

`show.error.locations`: Should source locations of errors be printed? If set to `TRUE` or `"top"`, the source location that is highest on the stack (the most recent call) will be printed. `"bottom"` will print the location of the earliest call found on the stack.

Integer values can select other entries. The value `0` corresponds to `"top"` and positive values count down the stack from there. The value `-1` corresponds to `"bottom"` and negative values count up from there.

`show.error.messages`: a logical. Should error messages be printed? Intended for use with `try` or a user-installed error handler.

`texi2dvi`: used by functions `texi2dvi` and `texi2pdf` in package **tools**.

unix-alike only: Set at startup from the environment variable `R_TEXI2DVICMD`, which defaults first to the value of environment variable `TEXI2DVI`, and then to a value set when R was installed (the full path to a `texi2dvi` script if one was found). If necessary, that environment variable can be set to `"emulation"`.

`timeout`: positive integer. The timeout for some Internet operations, in seconds. Default 60 (seconds) but can be set from environment variable `R_DEFAULT_INTERNET_TIMEOUT`. (Invalid values of the option or the variable are silently ignored: non-integer numeric values will be truncated.) See `download.file` and `connections`.

`topLevelEnvironment`: see `topenv` and `sys.source`.

`url.method`: character string: the default method for `url`. Normally unset, which is equivalent to `"default"`, which is `"internal"` except on Windows.

`useFancyQuotes`: controls the use of directional quotes in `sQuote`, `dQuote` and in rendering text help (see `Rd2txt` in package **tools**). Can be `TRUE`, `FALSE`, `"TeX"` or `"UTF-8"`.

`verbose`: logical. Should R report extra information on progress? Set to `TRUE` by the command-line option '`--verbose`'.

`warn`: integer value to set the handling of warning messages by the default warning handler. If `warn` is negative all warnings are ignored. If `warn` is zero (the default) warnings are stored until the top–level function returns. If 10 or fewer warnings were signalled they will be printed otherwise a message saying how many were signalled. An object called `last.warning` is created and can be printed through the function `warnings`. If `warn` is one, warnings are printed as they occur. If `warn` is two (or larger, coercible to integer), all warnings are turned into errors. While sometimes useful for debugging, turning warnings into errors may trigger bugs and resource leaks that would not have been triggered otherwise.

`warnPartialMatchArgs`: logical. If true, warns if partial matching is used in argument matching.

`warnPartialMatchAttr`: logical. If true, warns if partial matching is used in extracting attributes via `attr`.

`warnPartialMatchDollar`: logical. If true, warns if partial matching is used for extraction by `$`.

`warning.expression`: an R code expression to be called if a warning is generated, replacing the standard message. If non-null it is called irrespective of the value of option `warn`.

`warning.length`: sets the truncation limit in bytes for error and warning messages. A non-negative integer, with allowed values 100...8170, default 1000.

`nwarnings`: the limit for the number of warnings kept when `warn = 0`, default 50. This will discard messages if called whilst they are being collected. If you increase this limit, be aware that the current implementation pre-allocates the equivalent of a named list for them, i.e., do not increase it to more than say a million.

`width`: controls the maximum number of columns on a line used in printing vectors, matrices and arrays, and when filling by `cat`.

Columns are normally the same as characters except in East Asian languages.

You may want to change this if you re-size the window that R is running in. Valid values are 10...10000 with default normally 80. (The limits on valid values are in file '`Print.h`' and can be changed by re-compiling R.) Some R consoles automatically change the value when they are resized.

See the examples on Startup for one way to set this automatically from the terminal width when R is started.

The 'factory-fresh' default settings of some of these options are

```
add.smooth            TRUE
check.bounds          FALSE
continue              "+ "
digits                7
echo                  TRUE
encoding              "native.enc"
error                 NULL
expressions           5000
keep.source           interactive()
keep.source.pkgs      FALSE
max.print             99999
OutDec                "."
prompt                "> "
scipen                0
show.error.messages   TRUE
timeout               60
verbose               FALSE
warn                  0
warning.length        1000
width                 80
```

Others are set from environment variables or are platform-dependent.

Options set in package grDevices

These will be set when package **grDevices** (or its namespace) is loaded if not already set.

`bitmapType`: (Unix only, incl. macOS) character. The default type for the bitmap devices such as `png`. Defaults to `"cairo"` on systems where that is available, or to `"quartz"` on macOS where that is available.

`device`: a character string giving the name of a function, or the function object itself, which when called creates a new graphics device of the default type for that session. The value of this option defaults to the normal screen device (e.g., `X11`, `windows` or `quartz`) for an interactive session, and `pdf` in batch use or if a screen is not available. If set to the name of a device, the device is looked for first from the global environment (that is down the usual search path) and then in the **grDevices** namespace.

The default values in interactive and non-interactive sessions are configurable via environment variables `R_INTERACTIVE_DEVICE` and `R_DEFAULT_DEVICE` respectively.

The search logic for 'the normal screen device' is that this is `windows` on Windows, and `quartz` if available on macOS (running at the console, and compiled into the build). Otherwise `X11` is used if environment variable `DISPLAY` is set.

`device.ask.default`: logical. The default for `devAskNewPage("ask")` when a device is opened.

`locatorBell`: logical. Should selection in `locator` and `identify` be confirmed by a bell? Default `TRUE`. Honoured at least on `X11` and `windows` devices.

`windowsTimeout`: (Windows-only) integer vector of length 2 representing two times in milliseconds. These control the double-buffering of `windows` devices when that is enabled: the first is the delay after plotting finishes (default 100) and the second is the update interval during continuous plotting (default 500). The values at the time the device is opened are used.

Other options used by package graphics

`max.contour.segments`: positive integer, defaulting to `25000` if not set. A limit on the number of segments in a single contour line in `contour` or `contourLines`.

Options set in package stats

These will be set when package **stats** (or its namespace) is loaded if not already set.

`contrasts`: the default `contrasts` used in model fitting such as with `aov` or `lm`. A character vector of length two, the first giving the function to be used with unordered factors and the second the function to be used with ordered factors. By default the elements are named `c("unordered", "ordered")`, but the names are unused.

`na.action`: the name of a function for treating missing values (`NA`'s) for certain situations, see `na.action` and `na.pass`.

`show.coef.Pvalues`: logical, affecting whether P values are printed in summary tables of coefficients. See `printCoefmat`.

`show.nls.convergence`: logical, should `nls` convergence messages be printed for successful fits?

`show.signif.stars`: logical, should stars be printed on summary tables of coefficients? See `printCoefmat`.

`ts.eps`: the relative tolerance for certain time series (`ts`) computations. Default `1e-05`.

`ts.S.compat`: logical. Used to select S compatibility for plotting time-series spectra. See the description of argument `log` in `plot.spec`.

Options set (or used) in package utils

These will be set (apart from Ncpus) when package **utils** (or its namespace) is loaded if not already set.

BioC_mirror: The URL of a Bioconductor mirror for use by setRepositories, e.g. the default '"https://bioconductor.org"' or the European mirror '"https://bioconductor.statistik.tu-dortmund.de"'. Can be set by chooseBioCmirror.

browser: The HTML browser to be used by browseURL. This sets the default browser on UNIX or a non-default browser on Windows. Alternatively, an R function that is called with a URL as its argument. See browseURL for further details.

ccaddress: default Cc: address used by create.post (and hence bug.report and help.request). Can be FALSE or "".

citation.bibtex.max: default 1; the maximal number of bibentries (bibentry) in a citation for which the BibTeX version is printed in addition to the text one.

de.cellwidth: integer: the cell widths (number of characters) to be used in the data editor dataentry. If this is unset (the default), 0, negative or NA, variable cell widths are used.

demo.ask: default for the ask argument of demo.

editor: a non-empty character string or an R function that sets the default text editor, e.g., for edit and file.edit. Set from the environment variable EDITOR on UNIX, or if unset VISUAL or vi. As a string it should specify the name of or path to an external command.

example.ask: default for the ask argument of example.

help.ports: optional integer vector for setting ports of the internal HTTP server, see startDynamicHelp.

help.search.types: default types of documentation to be searched by help.search and ??.

help.try.all.packages: default for an argument of help.

help_type: default for an argument of help, used also as the help type by ?.

help.htmlmath: default for the texmath argument of Rd2HTML, controlling how LaTeX-like mathematical equations are displayed in R help pages (if enabled). Useful values are "katex" (equivalent to NULL, the default) and "mathjax"; for all other values basic substitutions are used.

help.htmltoc: default for the toc argument of Rd2HTML, controlling whether a table of contents should be included.

HTTPUserAgent: string used as the 'user agent' in HTTP(S) requests by download.file, url and curlGetHeaders, or NULL when requests will be made without a user agent header. The default is "R (*version platform arch os*)" except when 'libcurl' is used when it is "libcurl/*version*" for the 'libcurl' version in use.

install.lock: logical: should per-directory package locking be used by install.packages? Most useful for binary installs on macOS and Windows, but can be used in a startup file for source installs *via* R CMD INSTALL. For binary installs, can also be the character string "pkglock".

internet.info: The minimum level of information to be printed on URL downloads etc, using the "internal" and "libcurl" methods. Default is 2, for failure causes. Set to 1 or 0 to get more detailed information (for the "internal" method 0 provides more information than 1).

`install.packages.check.source`**:** Used by `install.packages` (and indirectly `update.packages`) on platforms which support binary packages. Possible values `"yes"` and `"no"`, with unset being equivalent to `"yes"`.

`install.packages.compile.from.source`**:** Used by `install.packages(type = "both")` (and indirectly `update.packages`) on platforms which support binary packages. Possible values are `"never"`, `"interactive"` (which means ask in interactive use and `"never"` in batch use) and `"always"`. The default is taken from environment variable `R_COMPILE_AND_INSTALL_PACKAGES`, with default `"interactive"` if unset. However, `install.packages` uses `"never"` unless a `make` program is found, consulting the environment variable `MAKE`.

`mailer`**:** default emailing method used by `create.post` and hence `bug.report` and `help.request`.

`menu.graphics`**:** Logical: should graphical menus be used if available? Defaults to `TRUE`. Currently applies to `select.list`, `chooseCRANmirror`, `setRepositories` and to select from multiple (text) help files in `help`.

`Ncpus`**:** an integer $n \geq 1$, used in `install.packages` as default for the number of CPUs to use in a potentially parallel installation, as `Ncpus = getOption("Ncpus", 1L)`, i.e., when unset is equivalent to a setting of 1.

`pkgType`**:** The default type of packages to be downloaded and installed – see `install.packages`. Possible values are platform dependently

on Windows `"win.binary"`, `"source"` and `"both"` (the default).

on Unix-alikes `"source"` (the default except under a CRAN macOS build), `"mac.binary"` and `"both"` (the default for CRAN macOS builds). (`"mac.binary.el-capitan"`, `"mac.binary.mavericks"`, `"mac.binary.leopard"` and `"mac.binary.universal"` are no longer in use.)

Value `"binary"` is a synonym for the native binary type (if there is one); `"both"` is used by `install.packages` to choose between source and binary installs.

`repos`**:** character vector of repository URLs for use by `available.packages` and related functions. Initially set from entries marked as default in the '`repositories`' file, whose path is configurable via environment variable `R_REPOSITORIES` (set this to `NULL` to skip initialization at startup). The 'factory-fresh' setting from the file in `R.home("etc")` is `c(CRAN="@CRAN@")`, a value that causes some utilities to prompt for a CRAN mirror. To avoid this do set the CRAN mirror, by something like

```
local({
    r <- getOption("repos")
    r["CRAN"] <- "https://my.local.cran"
    options(repos = r)
})
```

in your '`.Rprofile`', or use a personal '`repositories`' file.

Note that you can add more repositories (Bioconductor, R-Forge, RForge.net, ...) for the current session using `setRepositories`.

`str`**:** a list of options controlling the default `str` display. Defaults to `strOptions()`.

`str.dendrogram.last`**:** see `str.dendrogram`.

`SweaveHooks`**,** `SweaveSyntax`**:** see `Sweave`.

`unzip`: a character string used by `unzip`: the path of the external program `unzip` or `"internal"`. Defaults (platform dependently)

on unix-alikes to the value of `R_UNZIPCMD`, which is set in '`etc/Renviron`' to the path of the `unzip` command found during configuration and otherwise to `""`.

on Windows to `"internal"` when the internal unzip code is used.

Options set in package parallel

These will be set when package **parallel** (or its namespace) is loaded if not already set.

`mc.cores`: an integer giving the maximum allowed number of *additional* R processes allowed to be run in parallel to the current R process. Defaults to the setting of the environment variable `MC_CORES` if set. Most applications which use this assume a limit of `2` if it is unset.

Options used on Unix only

`dvipscmd`: character string giving a command to be used in the (deprecated) off-line printing of help pages *via* PostScript. Defaults to `"dvips"`.

Options used on Windows only

`warn.FPU`: logical, by default undefined. If true, a warning is produced whenever dyn.load repairs the control word damaged by a buggy DLL.

Note

For compatibility with S there is a visible object `.Options` whose value is a pairlist containing the current `options()` (in no particular order). Assigning to it will make a local copy and not change the original. (*Using* it however is faster than calling `options()`).

An option set to `NULL` is indistinguishable from a non existing option.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
op <- options(); utils::str(op) # op is a named list

getOption("width") == options()$width # the latter needs more memory
options(digits = 15)
pi

# set the editor, and save previous value
old.o <- options(editor = "nedit")
old.o

options(check.bounds = TRUE, warn = 1)
x <- NULL; x[4] <- "yes" # gives a warning
```

```
options(digits = 5)
print(1e5)
options(scipen = 3); print(1e5)

options(op)     # reset (all) initial options
options("digits")

## Not run: ## set contrast handling to be like S
options(contrasts = c("contr.helmert", "contr.poly"))

## End(Not run)

## Not run: ## on error, terminate the R session with error status 66
options(error = quote(q("no", status = 66, runLast = FALSE)))
stop("test it")

## End(Not run)

## Not run: ## Set error actions for debugging:
## enter browser on error, see ?recover:
options(error = recover)
## allows to call debugger() afterwards, see ?debugger:
options(error = dump.frames)
## A possible setting for non-interactive sessions
options(error = quote({dump.frames(to.file = TRUE); q()}))

## End(Not run)

  # Compare the two ways to get an option and use it
  # acconting for the possibility it might not be set.
if(as.logical(getOption("performCleanp", TRUE)))
   cat("do cleanup\n")

## Not run:
  # a clumsier way of expressing the above w/o the default.
tmp <- getOption("performCleanup")
if(is.null(tmp))
  tmp <- TRUE
if(tmp)
   cat("do cleanup\n")

## End(Not run)
```

---

order *Ordering Permutation*

Description

order returns a permutation which rearranges its first argument into ascending or descending order,

breaking ties by further arguments. `sort.list` does the same, using only one argument. See the examples for how to use these functions to sort data frames, etc.

Usage

```
order(..., na.last = TRUE, decreasing = FALSE,
      method = c("auto", "shell", "radix"))

sort.list(x, partial = NULL, na.last = TRUE, decreasing = FALSE,
          method = c("auto", "shell", "quick", "radix"))
```

Arguments

`...`	a sequence of numeric, complex, character or logical vectors, all of the same length, or a classed R object.
`x`	an atomic vector for `methods` `"shell"` and `"quick"`. When `x` is a non-atomic R object, the default `"auto"` and `"radix"` methods may work if `order(x,..)` does.
`partial`	vector of indices for partial sorting. (Non-`NULL` values are not implemented.)
`decreasing`	logical. Should the sort order be increasing or decreasing? For the `"radix"` method, this can be a vector of length equal to the number of arguments in `...` and the elements are recycled as necessary. For the other methods, it must be length one.
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed (see 'Note'.)
`method`	the method to be used: partial matches are allowed. The default (`"auto"`) implies `"radix"` for numeric vectors, integer vectors, logical vectors and factors with fewer than 2^{31} elements. Otherwise, it implies `"shell"`. For details of methods `"shell"`, `"quick"`, and `"radix"`, see the help for `sort`.

Details

In the case of ties in the first vector, values in the second are used to break the ties. If the values are still tied, values in the later arguments are used to break the tie (see the first example). The sort used is *stable* (except for `method = "quick"`), so any unresolved ties will be left in their original ordering.

Complex values are sorted first by the real part, then the imaginary part.

Except for method `"radix"`, the sort order for character vectors will depend on the collating sequence of the locale in use: see `Comparison`.

The `"shell"` method is generally the safest bet and is the default method, except for short factors, numeric vectors, integer vectors and logical vectors, where `"radix"` is assumed. Method `"radix"` stably sorts logical, numeric and character vectors in linear time. It outperforms the other methods, although there are drawbacks, especially for character vectors (see `sort`). Method `"quick"` for `sort.list` is only supported for numeric `x` with `na.last = NA`, is not stable, and is slower than `"radix"`.

`partial = NULL` is supported for compatibility with other implementations of S, but no other values are accepted and ordering is always complete.

For a classed R object, the sort order is taken from `xtfrm`: as its help page notes, this can be slow unless a suitable method has been defined or `is.numeric(x)` is true. For factors, this sorts on the internal codes, which is particularly appropriate for ordered factors.

Value

An integer vector unless any of the inputs has 2^{31} or more elements, when it is a double vector.

Warning

In programmatic use it is unsafe to name the `...` arguments, as the names could match current or future control arguments such as `decreasing`. A sometimes-encountered unsafe practice is to call `do.call('order', df_obj)` where `df_obj` might be a data frame: copy `df_obj` and remove any names, for example using `unname`.

Note

`sort.list` can get called by mistake as a method for `sort` with a list argument: it gives a suitable error message for list `x`.

There is a historical difference in behaviour for `na.last = NA`: `sort.list` removes the `NA`s and then computes the order amongst the remaining elements: `order` computes the order amongst the non-`NA` elements of the original vector. Thus

```
x[order(x, na.last = NA)]
zz <- x[!is.na(x)]; zz[sort.list(x, na.last = NA)]
```

both sort the non-`NA` values of `x`.

Prior to R 3.3.0 `method = "radix"` was only supported for integers of range less than 100,000.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Addison-Wesley.

See Also

`sort`, `rank`, `xtfrm`.

Examples

```
require(stats)

(ii <- order(x <- c(1,1,3:1,1:4,3), y <- c(9,9:1), z <- c(2,1:9)))
## 6  5  2  1  7  4 10  8  3  9
rbind(x, y, z)[,ii] # shows the reordering (ties via 2nd & 3rd arg)

## Suppose we wanted descending order on y.
## A simple solution for numeric 'y' is
```

```
rbind(x, y, z)[, order(x, -y, z)]
## More generally we can make use of xtfrm
cy <- as.character(y)
rbind(x, y, z)[, order(x, -xtfrm(cy), z)]
## The radix sort supports multiple 'decreasing' values:
rbind(x, y, z)[, order(x, cy, z, decreasing = c(FALSE, TRUE, FALSE),
                       method="radix")]

## Sorting data frames:
dd <- transform(data.frame(x, y, z),
                z = factor(z, labels = LETTERS[9:1]))
## Either as above {for factor 'z' : using internal coding}:
dd[ order(x, -y, z), ]
## or along 1st column, ties along 2nd, ... *arbitrary* no.{columns}:
dd[ do.call(order, dd), ]

set.seed(1)  # reproducible example:
d4 <- data.frame(x = round(   rnorm(100)), y = round(10*runif(100)),
                 z = round( 8*rnorm(100)), u = round(50*runif(100)))
(d4s <- d4[ do.call(order, d4), ])
(i <- which(diff(d4s[, 3]) == 0))
#   in 2 places, needed 3 cols to break ties:
d4s[ rbind(i, i+1), ]

## rearrange matched vectors so that the first is in ascending order
x <- c(5:1, 6:8, 12:9)
y <- (x - 5)^2
o <- order(x)
rbind(x[o], y[o])

## tests of na.last
a <- c(4, 3, 2, NA, 1)
b <- c(4, NA, 2, 7, 1)
z <- cbind(a, b)
(o <- order(a, b)); z[o, ]
(o <- order(a, b, na.last = FALSE)); z[o, ]
(o <- order(a, b, na.last = NA)); z[o, ]


##  speed examples on an average laptop for long vectors:
##  factor/small-valued integers:
x <- factor(sample(letters, 1e7, replace = TRUE))
system.time(o <- sort.list(x, method = "quick", na.last = NA)) # 0.1 sec
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method = "radix")) # 0.05 sec, 2X faster
stopifnot(!is.unsorted(x[o]))
##  large-valued integers:
xx <- sample(1:200000, 1e7, replace = TRUE)
system.time(o <- sort.list(xx, method = "quick", na.last = NA)) # 0.3 sec
system.time(o <- sort.list(xx, method = "radix")) # 0.2 sec
##  character vectors:
xx <- sample(state.name, 1e6, replace = TRUE)
system.time(o <- sort.list(xx, method = "shell")) # 2 sec
```

```
system.time(o <- sort.list(xx, method = "radix")) # 0.007 sec, 300X faster
##  double vectors:
xx <- rnorm(1e6)
system.time(o <- sort.list(xx, method = "shell")) # 0.4 sec
system.time(o <- sort.list(xx, method = "quick", na.last = NA)) # 0.1 sec
system.time(o <- sort.list(xx, method = "radix")) # 0.05 sec, 2X faster
```

`outer` *Outer Product of Arrays*

Description

The outer product of the arrays `X` and `Y` is the array `A` with dimension `c(dim(X), dim(Y))` where element `A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...)`.

Usage

```
outer(X, Y, FUN = "*", ...)
X %o% Y
```

Arguments

`X, Y`	first and second arguments for function `FUN`. Typically a vector or array.
`FUN`	a function to use on the outer products, found *via* `match.fun` (except for the special case `"*"`).
`...`	optional arguments to be passed to `FUN`.

Details

`X` and `Y` must be suitable arguments for `FUN`. Each will be extended by `rep` to length the products of the lengths of `X` and `Y` before `FUN` is called.

`FUN` is called with these two extended vectors as arguments (plus any arguments in `...`). It must be a vectorized function (or the name of one) expecting at least two arguments and returning a value with the same length as the first (and the second).

Where they exist, the [dim]names of `X` and `Y` will be copied to the answer, and a dimension assigned which is the concatenation of the dimensions of `X` and `Y` (or lengths if dimensions do not exist).

`FUN = "*"` is handled as a special case *via* `as.vector(X) %*% t(as.vector(Y))`, and is intended only for numeric vectors and arrays.

`%o%` is binary operator providing a wrapper for `outer(x, y, "*")`.

Author(s)

Jonathan Rougier

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`%*%` for usual (*inner*) matrix vector multiplication; `kronecker` which is based on `outer`; `Vectorize` for vectorizing a non-vectorized function.

Examples

```
x <- 1:9; names(x) <- x
# Multiplication & Power Tables
x %o% x
y <- 2:8; names(y) <- paste(y,":", sep = "")
outer(y, x, `^`)

outer(month.abb, 1999:2003, FUN = paste)

## three way multiplication table:
x %o% x %o% y[1:3]
```

`Paren` *Parentheses and Braces*

Description

Open parenthesis, `(`, and open brace, `{`, are `.Primitive` functions in R.

Effectively, `(` is semantically equivalent to the identity `function(x) x`, whereas `{` is slightly more interesting, see examples.

Usage

```
( ... )

{ ... }
```

Value

For `(`, the result of evaluating the argument. This has visibility set, so will auto-print if used at top-level.

For `{`, the result of the last expression evaluated. This has the visibility of the last evaluation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

if, return, etc for other objects used in the R language itself.

Syntax for operator precedence.

Examples

```
f <- get("(")
e <- expression(3 + 2 * 4)
identical(f(e), e)

do <- get("{")
do(x <- 3, y <- 2*x-3, 6-x-y); x; y

## note the differences
(2+3)
{2+3; 4+5}
(invisible(2+3))
{invisible(2+3)}
```

---

parse *Parse R Expressions*

Description

parse() returns the parsed but unevaluated expressions in an expression, a "list" of calls.

str2expression(s) and str2lang(s) return special versions of parse(text=s, keep.source=FALSE) and can therefore be regarded as transforming character strings s to expressions, calls, etc.

Usage

```
parse(file = "", n = NULL, text = NULL, prompt = "?",
      keep.source = getOption("keep.source"), srcfile,
      encoding = "unknown")

str2lang(s)
str2expression(text)
```

Arguments

file	a connection, or a character string giving the name of a file or a URL to read the expressions from. If file is "" and text is missing or NULL then input is taken from the console.
n	integer (or coerced to integer). The maximum number of expressions to parse. If n is NULL or negative or NA the input is parsed in its entirety.
text	character vector. The text to parse. Elements are treated as if they were lines of a file. Other R objects will be coerced to character if possible.

prompt	the prompt to print when parsing from the keyboard. NULL means to use R's prompt, getOption("prompt").
keep.source	a logical value; if TRUE, keep source reference information.
srcfile	NULL, a character vector, or a srcfile object. See the 'Details' section.
encoding	encoding to be assumed for input strings. If the value is "latin1" or "UTF-8" it is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input. To do the latter, specify the encoding as part of the connection con or *via* options(encoding=): see the example under file. Arguments encoding = "latin1" and encoding = "UTF-8" are ignored with a warning when running in a MBCS locale.
s	a character vector of length 1, i.e., a "string".

Details

parse(....): If text has length greater than zero (after coercion) it is used in preference to file.

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic Mac OS). The final line can be incomplete, that is missing the final EOL marker.

When input is taken from the console, n = NULL is equivalent to n = 1, and n < 0 will read until an EOF character is read. (The EOF character is Ctrl-Z for the Windows front-ends.) The line-length limit is 4095 bytes when reading from the console (which may impose a lower limit: see 'An Introduction to R').

The default for srcfile is set as follows. If keep.source is not TRUE, srcfile defaults to a character string, either "<text>" or one derived from file. When keep.source is TRUE, if text is used, srcfile will be set to a srcfilecopy containing the text. If a character string is used for file, a srcfile object referring to that file will be used.

When srcfile is a character string, error messages will include the name, but source reference information will not be added to the result. When srcfile is a srcfile object, source reference information will be retained.

str2expression(s): for a character vector s, str2expression(s) corresponds to parse(text = s, keep.source=FALSE), which is always of type (typeof) and class expression.

str2lang(s): for a character string s, str2lang(s) corresponds to parse(text = s, keep.source=FALSE)[[1]] (plus a check that both s and the parse(*) result are of length one) which is typically a call but may also be a symbol aka name, NULL or an atomic constant such as 2, 1L, or TRUE. Put differently, the value of str2lang(.) is a call or one of its parts, in short "a call or simpler".

Currently, encoding is not handled in str2lang() and str2expression().

Value

parse() and str2expression() return an object of type "expression", for parse() with up to n elements if specified as a non-negative integer.

str2lang(s), s a string, returns "a call or simpler", see the 'Details:' section.

When srcfile is non-NULL, a "srcref" attribute will be attached to the result containing a list of srcref records corresponding to each element, a "srcfile" attribute will be attached containing

a copy of srcfile, and a "wholeSrcref" attribute will be attached containing a srcref record corresponding to all of the parsed text. Detailed parse information will be stored in the "srcfile" attribute, to be retrieved by getParseData.

A syntax error (including an incomplete expression) will throw an error.

Character strings in the result will have a declared encoding if encoding is "latin1" or "UTF-8", or if text is supplied with every element of known encoding in a Latin-1 or UTF-8 locale.

Partial parsing

When a syntax error occurs during parsing, parse signals an error. The partial parse data will be stored in the srcfile argument if it is a srcfile object and the text argument was used to supply the text. In other cases it will be lost when the error is triggered.

The partial parse data can be retrieved using getParseData applied to the srcfile object. Because parsing was incomplete, it will typically include references to "parent" entries that are not present.

Note

Using parse(text = *, ..) or its simplified and hence more efficient versions str2lang() or str2expression() is at least an order of magnitude less efficient than call(..) or as.call().

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murdoch, D. (2010). "Source References". *The R Journal*, **2**(2), 16–19. doi:10.32614/RJ2010010.

See Also

scan, source, eval, deparse.

The source reference information can be used for debugging (see e.g. setBreakpoint) and profiling (see Rprof). It can be examined by getSrcref and related functions. More detailed information is available through getParseData.

Examples

```
fil <- tempfile(fileext = ".Rdmped")
cat("x <- c(1, 4)\n  x ^ 3 -10 ; outer(1:7, 5:9)\n", file = fil)
# parse 3 statements from our temp file
parse(file = fil, n = 3)
unlink(fil)

## str2lang(<string>)  || str2expression(<character>) :
stopifnot(exprs = {
  identical( str2lang("x[3] <- 1+4"), quote(x[3] <- 1+4))
  identical( str2lang("log(y)"),      quote(log(y)) )
  identical( str2lang("abc"   ),      quote(abc) -> qa)
  is.symbol(qa) & !is.call(qa)           # a symbol/name, not a call
  identical( str2lang("1.375" ), 1.375)  # just a number, not a call
  identical( str2expression(c("# a comment", "", "42")), expression(42) )
```

```
})

# A partial parse with a syntax error
txt <- "
x <- 1
an error
"
sf <- srcfile("txt")
tryCatch(parse(text = txt, srcfile = sf), error = function(e) "Syntax error.")
getParseData(sf)
```

`paste` *Concatenate Strings*

Description

Concatenate vectors after converting to character. Concatenation happens in two basically different ways, determined by `collapse` being a string or not.

Usage

```
paste (..., sep = " ", collapse = NULL, recycle0 = FALSE)
paste0(...,            collapse = NULL, recycle0 = FALSE)
```

Arguments

`...`	one or more R objects, to be converted to character vectors.
`sep`	a character string to separate the terms. Not `NA_character_`.
`collapse`	an optional character string to separate the results. Not `NA_character_`. When `collapse` is a string, the result is always a string (`character` of length 1).
`recycle0`	`logical` indicating if zero-length character arguments should result in the zero-length `character(0)`. Note that when `collapse` is a string, `recycle0` does *not* recycle to zero-length, but to `""`.

Details

`paste` converts its arguments (*via* `as.character`) to character strings, and concatenates them (separating them by the string given by `sep`).

If the arguments are vectors, they are concatenated term-by-term to give a character vector result. Vector arguments are recycled as needed. Zero-length arguments are recycled as `""` unless `recycle0` is `TRUE` and `collapse` is `NULL`.

Note that `paste()` coerces `NA_character_`, the character missing value, to `"NA"` which may seem undesirable, e.g., when pasting two character vectors, or very desirable, e.g. in `paste("the value of p is ", p)`.

`paste0(..., collapse)` is equivalent to `paste(..., sep = "", collapse)`, slightly more efficiently.

If a value is specified for `collapse`, the values in the result are then concatenated into a single string, with the elements being separated by the value of `collapse`.

Value

A character vector of the concatenated values. This will be of length zero if all the objects are, unless `collapse` is non-NULL, in which case it is `""` (a single empty string).

If any input into an element of the result is in UTF-8 (and none are declared with encoding `"bytes"`, see `Encoding`), that element will be in UTF-8, otherwise in the current encoding in which case the encoding of the element is declared if the current locale is either Latin-1 or UTF-8, at least one of the corresponding inputs (including separators) had a declared encoding and all inputs were either ASCII or declared.

If an input into an element is declared with encoding `"bytes"`, no translation will be done of any of the elements and the resulting element will have encoding `"bytes"`. If `collapse` is non-NULL, this applies also to the second, collapsing, phase, but some translation may have been done in pasting object together in the first phase.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`toString` typically calls `paste(*, collapse=", ")`. String manipulation with `as.character`, `substr`, `nchar`, `strsplit`; further, `cat` which concatenates and writes to a file, and `sprintf` for C like string construction.

'plotmath' for the use of `paste` in plot annotation.

Examples

```
## When passing a single vector, paste0 and paste work like as.character.
paste0(1:12)
paste(1:12)        # same
as.character(1:12) # same

## If you pass several vectors to paste0, they are concatenated in a
## vectorized way.
(nth <- paste0(1:12, c("st", "nd", "rd", rep("th", 9))))

## paste works the same, but separates each input with a space.
## Notice that the recycling rules make every input as long as the longest input.
paste(month.abb, "is the", nth, "month of the year.")
paste(month.abb, letters)

## You can change the separator by passing a sep argument
## which can be multiple characters.
paste(month.abb, "is the", nth, "month of the year.", sep = "_*_")

## To collapse the output into a single string, pass a collapse argument.
paste0(nth, collapse = ", ")

## For inputs of length 1, use the sep argument rather than collapse
paste("1st", "2nd", "3rd", collapse = ", ") # probably not what you wanted
```

```
paste("1st", "2nd", "3rd", sep = ", ")

## You can combine the sep and collapse arguments together.
paste(month.abb, nth, sep = ": ", collapse = "; ")

## Using paste() in combination with strwrap() can be useful
## for dealing with long strings.
(title <- paste(strwrap(
    "Stopping distance of cars (ft) vs. speed (mph) from Ezekiel (1930)",
    width = 30), collapse = "\n"))
plot(dist ~ speed, cars, main = title)

## zero length arguments recycled as `""` -- NB: `{}` <==> character(0)  here
paste({}, 1:2)

## 'recycle0 = TRUE' allows standard vectorized behaviour, i.e., zero-length
##                   recycling resulting in zero-length result character(0):
valid <- FALSE
val <- pi
paste("The value is", val[valid], "-- not so good!") # ->  ".. value is  -- not .."
paste("The value is", val[valid], "-- good: empty!", recycle0=TRUE) # -> character(0)

## When 'collapse = <string>',  result is (length 1) string in all cases
paste("foo", {}, "bar", collapse = "|")                 # |-->  "foo  bar"
paste("foo", {},        collapse = "|", recycle0 = TRUE) # |-->  ""
## If all arguments are empty (and collapse a string),   ""  results always
paste(    collapse = "|")
paste(    collapse = "|", recycle0 = TRUE)
paste({}, collapse = "|")
paste({}, collapse = "|", recycle0 = TRUE)
```

`path.expand` *Expand File Paths*

Description

Expand a path name, for example by replacing a leading tilde by the user's home directory (if defined on that platform).

Usage

```
path.expand(path)
```

Arguments

`path`	character vector containing one or more path names.

Details

On Unix - alikes: On most builds of R a leading `~user` will expand to the home directory of `user`. There are possibly different concepts of 'home directory': that usually used is the setting of the environment variable `HOME`.

The 'path names' need not exist nor be valid path names but they do need to be representable in the session encoding.

On Windows: The definition of the 'home' directory is in the '`rw-FAQ`' Q2.14: it is taken from the `R_USER` environment variable when `path.expand` is first called in a session.

The 'path names' need not exist nor be valid path names.

Value

A character vector of possibly expanded path names: where the home directory is unknown or none is specified the path is returned unchanged.

If the expansion would exceed the maximum path length the result may be truncated or the path may be returned unchanged.

See Also

`basename`, `normalizePath`, `file.path`.

Examples

```
path.expand("~/foo")
```

---

pcre_config *Report Configuration Options for PCRE*

Description

Report some of the configuration options of the version of PCRE in use in this R session.

Usage

```
pcre_config()
```

Value

A named logical vector, currently with elements

`UTF-8` Support for UTF-8 inputs. Required.

`Unicode properties`
Support for '`\p{xx}`' and '`\P{xx}`' in regular expressions. Desirable and used by some CRAN packages. As of PCRE2, always present with support for UTF-8.

JIT	Support for just-in-time compilation. Desirable for speed (but only available as a compile-time option on certain architectures, and may be unused as unreliable on some of those, e.g. arm64).
stack	Does match recursion use a stack (TRUE, the default for PCRE1 and PCRE2 older than 10.30) or a heap? See the discussion at https://www.pcre.org/original/doc/html/pcrestack.html (Added in R 3.4.0.). No longer relevant and always FALSE in PCRE2 since version 10.30 which no longer uses function recursion to remember backtracking positions.

See Also

extSoftVersion for the PCRE version.

Examples

```
pcre_config()
```

---

pipeOp *Forward Pipe Operator*

Description

Pipe a value into a call expression or a function expression.

Usage

```
lhs |> rhs
```

Arguments

lhs	expression producing a value.
rhs	a call expression.

Details

A pipe expression passes, or 'pipes', the result of the left-hand-side expression lhs to the right-hand-side expression rhs.

The lhs is inserted as the first argument in the call. So x |> f(y) is interpreted as f(x, y).

To avoid ambiguities, functions in rhs calls may not be syntactically special, such as + or if.

It is also possible to use a named argument with the placeholder _ in the rhs call to specify where the lhs is to be inserted. The placeholder can only appear once on the rhs.

The placeholder can also be used as the first argument in an extraction call, such as _$coef. More generally, it can be used as the head of a chain of extractions, such as _$coef[[2]], using a sequence of the extraction functions $, [, [[, or @.

Pipe notation allows a nested sequence of calls to be written in a way that may make the sequence of processing steps easier to follow.

Currently, pipe operations are implemented as syntax transformations. So an expression written as `x |> f(y)` is parsed as `f(x, y)`. It is worth emphasizing that while the code in a pipeline is written sequentially, regular R semantics for evaluation apply and so piped expressions will be evaluated only when first used in the `rhs` expression.

Value

Returns the result of evaluating the transformed expression.

Background

The forward pipe operator is motivated by the pipe introduced in the **magrittr** package, but is more streamlined. It is similar to the pipe or pipeline operators introduced in other languages, including F#, Julia, and JavaScript.

Warning

This was introduced in R 4.1.0. Code using it will not be parsed as intended (probably with an error) in earlier versions of R.

Examples

```
# simple uses:
mtcars |> head()                      # same as head(mtcars)
mtcars |> head(2)                     # same as head(mtcars, 2)
mtcars |> subset(cyl == 4) |> nrow()  # same as nrow(subset(mtcars, cyl == 4))

# to pass the lhs into an argument other than the first, either
# use the _ placeholder with a named argument:
mtcars |> subset(cyl == 4) |> lm(mpg ~ disp, data = _)
# or use an anonymous function:
mtcars |> subset(cyl == 4) |> (function(d) lm(mpg ~ disp, data = d))()
mtcars |> subset(cyl == 4) |> (\(d) lm(mpg ~ disp, data = d))()
# or explicitly name the argument(s) before the "one":
mtcars |> subset(cyl == 4) |> lm(formula = mpg ~ disp)

# using the placeholder as the head of an extraction chain:
mtcars |> subset(cyl == 4) |> lm(formula = mpg ~ disp) |> _$coef[[2]]

# the pipe operator is implemented as a syntax transformation:
quote(mtcars |> subset(cyl == 4) |> nrow())

# regular R evaluation semantics apply
stop() |> (function(...) {})() # stop() is not used on RHS so is not evaluated
```

plot *Generic X-Y Plotting*

Description

Generic function for plotting of R objects.

For simple scatter plots, plot.default will be used. However, there are plot methods for many R objects, including functions, data.frames, density objects, etc. Use methods(plot) and the documentation for these. Most of these methods are implemented using traditional graphics (the **graphics** package), but this is not mandatory.

For more details about graphical parameter arguments used by traditional graphics, see par.

Usage

```
plot(x, y, ...)
```

Arguments

x	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any* R *object with a* plot *method* can be provided.
y	the y coordinates of points in the plot, *optional* if x is an appropriate structure.
...	arguments to be passed to methods, such as graphical parameters (see par). Many methods will accept the following arguments:

type what type of plot should be drawn. Possible types are

- "p" for **p**oints,
- "l" for **l**ines,
- "b" for **b**oth,
- "c" for the lines part alone of "b",
- "o" for both '**o**verplotted',
- "h" for '**h**istogram' like (or 'high-density') vertical lines,
- "s" for stair **s**teps,
- "S" for other **s**teps, see 'Details' below,
- "n" for no plotting.

All other types give a warning or an error; using, e.g., type = "punkte" being equivalent to type = "p" for S compatibility. Note that some methods, e.g. plot.factor, do not accept this.

main an overall title for the plot: see title.

sub a subtitle for the plot: see title.

xlab a title for the x axis: see title.

ylab a title for the y axis: see title.

asp the y/x aspect ratio, see plot.window.

Details

The two step types differ in their x-y preference: Going from $(x1, y1)$ to $(x2, y2)$ with $x1 < x2$, `type = "s"` moves first horizontal, then vertical, whereas `type = "S"` moves the other way around.

Note

The `plot` generic was moved from the **graphics** package to the **base** package in R 4.0.0. It is currently re-exported from the **graphics** namespace to allow packages importing it from there to continue working, but this may change in future versions of R.

See Also

`plot.default`, `plot.formula` and other methods; `points`, `lines`, `par`. For thousands of points, consider using `smoothScatter()` instead of `plot()`.

For X-Y-Z plotting see `contour`, `persp` and `image`.

Examples

```
require(stats) # for lowess, rpois, rnorm
require(graphics) # for plot methods
plot(cars)
lines(lowess(cars))

plot(sin, -pi, 2*pi) # see ?plot.function

## Discrete Distribution Plot:
plot(table(rpois(100, 5)), type = "h", col = "red", lwd = 10,
     main = "rpois(100, lambda = 5)")

## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
points(x, cex = .5, col = "dark red")
```

---

pmatch *Partial String Matching*

Description

`pmatch` seeks matches for the elements of its first argument among those of its second.

Usage

```
pmatch(x, table, nomatch = NA_integer_, duplicates.ok = FALSE)
```

Arguments

x	the values to be matched: converted to a character vector by as.character. Long vectors are supported.
table	the values to be matched against: converted to a character vector. Long vectors are not supported.
nomatch	the value to be returned at non-matching or multiply partially matching positions. Note that it is coerced to integer.
duplicates.ok	should elements in table be used more than once?

Details

The behaviour differs by the value of duplicates.ok. Consider first the case if this is true. First exact matches are considered, and the positions of the first exact matches are recorded. Then unique partial matches are considered, and if found recorded. (A partial match occurs if the whole of the element of x matches the beginning of the element of table.) Finally, all remaining elements of x are regarded as unmatched. In addition, an empty string can match nothing, not even an exact match to an empty string. This is the appropriate behaviour for partial matching of character indices, for example.

If duplicates.ok is FALSE, values of table once matched are excluded from the search for subsequent matches. This behaviour is equivalent to the R algorithm for argument matching, except for the consideration of empty strings (which in argument matching are matched after exact and partial matching to any remaining arguments).

charmatch is similar to pmatch with duplicates.ok true, the differences being that it differentiates between no match and an ambiguous partial match, it does match empty strings, and it does not allow multiple exact matches.

NA values are treated as if they were the string constant "NA".

Value

An integer vector (possibly including NA if nomatch = NA) of the same length as x, giving the indices of the elements in table which matched, or nomatch.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

match, charmatch and match.arg, match.fun, match.call, for function argument matching etc., startsWith for particular checking of initial matches; grep etc for more general (regexp) matching of strings.

Examples

```
pmatch("", "")                           # returns NA
pmatch("m",   c("mean", "median", "mode")) # returns NA
pmatch("med", c("mean", "median", "mode")) # returns 2

pmatch(c("", "ab", "ab"), c("abc", "ab"), duplicates.ok = FALSE)
pmatch(c("", "ab", "ab"), c("abc", "ab"), duplicates.ok = TRUE)
## compare
charmatch(c("", "ab", "ab"), c("abc", "ab"))
```

`polyroot` *Find Zeros of a Real or Complex Polynomial*

Description

Find zeros of a real or complex polynomial.

Usage

```
polyroot(z)
```

Arguments

`z`	the vector of polynomial coefficients in increasing order.

Details

A polynomial of degree $n - 1$,

$$p(x) = z_1 + z_2 x + \cdots + z_n x^{n-1}$$

is given by its coefficient vector `z[1:n]`. `polyroot` returns the $n - 1$ complex zeros of $p(x)$ using the Jenkins-Traub algorithm.

If the coefficient vector `z` has zeroes for the highest powers, these are discarded.

There is no maximum degree, but numerical stability may be an issue for all but low-degree polynomials.

Value

A complex vector of length $n - 1$, where n is the position of the largest non-zero element of `z`.

Source

C translation by Ross Ihaka of Fortran code in the reference, with modifications by the R Core Team.

References

Jenkins, M. A. and Traub, J. F. (1972). Algorithm 419: zeros of a complex polynomial. *Communications of the ACM*, **15**(2), 97–99. doi:10.1145/361254.361262.

See Also

`uniroot` for numerical root finding of arbitrary functions; `complex` and the `zero` example in the demos directory.

Examples

```
polyroot(c(1, 2, 1))
round(polyroot(choose(8, 0:8)), 11) # guess what!
for (n1 in 1:4) print(polyroot(1:n1), digits = 4)
polyroot(c(1, 2, 1, 0, 0)) # same as the first
```

---

pos.to.env *Convert Positions in the Search Path to Environments*

Description

Returns the environment at a specified position in the search path.

Usage

```
pos.to.env(x)
```

Arguments

x	an integer between 1 and `length(search())`, the length of the search path, or -1.

Details

Several R functions for manipulating objects in environments (such as `get` and `ls`) allow specifying environments via corresponding positions in the search path. `pos.to.env` is a convenience function for programmers which converts these positions to corresponding environments; users will typically have no need for it. It is primitive.

`-1` is interpreted as the environment the function is called from.

This is a primitive function.

Examples

```
pos.to.env(1) # R_GlobalEnv
# the next returns the base environment
pos.to.env(length(search()))
```

pretty *Pretty Breakpoints*

Description

Compute a sequence of about n+1 equally spaced 'round' values which cover the range of the values in x. The values are chosen so that they are 1, 2 or 5 times a power of 10.

Usage

```
pretty(x, ...)

## Default S3 method:
pretty(x, n = 5, min.n = n %/% 3,  shrink.sml = 0.75,
       high.u.bias = 1.5, u5.bias = .5 + 1.5*high.u.bias,
       eps.correct = 0, f.min = 2^-20, ...)

.pretty(x, n = 5L, min.n = n %/% 3,  shrink.sml = 0.75,
       high.u.bias = 1.5, u5.bias = .5 + 1.5*high.u.bias,
       eps.correct = 0L, f.min = 2^-20, bounds = TRUE)
```

Arguments

x	an object coercible to numeric by as.numeric.
n	integer giving the *desired* number of intervals. Non-integer values are rounded down.
min.n	nonnegative integer giving the *minimal* number of intervals. If min.n == 0, pretty(.) may return a single value.
shrink.sml	positive number, a factor (smaller than one) by which a default scale is shrunk in the case when range(x) is very small (usually 0).
high.u.bias	non-negative numeric, typically $>$ 1. The interval unit is determined as {1,2,5,10} times b, a power of 10. Larger high.u.bias values favor larger units.
u5.bias	non-negative numeric multiplier favoring factor 5 over 2. Default and 'optimal': u5.bias = .5 + 1.5*high.u.bias.
eps.correct	integer code, one of {0,1,2}. If non-0, an *epsilon correction* is made at the boundaries such that the result boundaries will be outside range(x); in the *small* case, the correction is only done if eps.correct >= 2.
f.min	positive factor multiplied by .Machine$double.xmin to get the smallest "acceptable" cell c_m which determines the unit of the algorithm. Smaller cell values are set to c_n signalling a warning about being "corrected". New from R 4.2.0,: previously f.min = 20 was hardcoded in the algorithm.
bounds	a logical indicating if the resulting vector should *cover* the full range(x), i.e., strictly include the bounds of x. New from R 4.2.0, allowing bound=FALSE to reproduce how R's graphics engine computes axis tick locations (in GEPretty()).
...	further arguments for methods.

Details

pretty ignores non-finite values in x.

Let d <- max(x) - min(x) ≥ 0. If d is not (very close) to 0, we let c <- d/n, otherwise more or less c <- max(abs(range(x)))*shrink.sml / min.n. Then, the *10 base* b is $10^{\lfloor \log_{10}(c) \rfloor}$ such that $b \leq c < 10b$.

Now determine the basic *unit* u as one of $\{1, 2, 5, 10\}b$, depending on $c/b \in [1, 10)$ and the two '*bias*' coefficients, $h =$high.u.bias and $f =$u5.bias.

.....

Value

pretty() returns an numeric vector of *approximately* n increasing numbers which are "pretty" in decimal notation. (in extreme range cases, the numbers can no longer be "pretty" given the other constraints; e.g., for pretty(..)

For ease of investigating the underlying C R_pretty() function, .pretty() returns a named list. By default, when bounds=TRUE, the entries are l, u, and n, whereas for bounds=FALSE, they are ns, nu, n, and (a "pretty") unit where the n*'s are integer valued (but only n is of class integer). Programmers may use this to create pretty sequence (iterator) objects.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

axTicks for the computation of pretty axis tick locations in plots, particularly on the log scale.

Examples

```
pretty(1:15)                    # 0  2  4  6  8 10 12 14 16
pretty(1:15, high.u.bias = 2)   # 0  5 10 15
pretty(1:15, n = 4)             # 0  5 10 15
pretty(1:15 * 2)                # 0  5 10 15 20 25 30
pretty(1:20)                    # 0  5 10 15 20
pretty(1:20, n = 2)             # 0 10 20
pretty(1:20, n = 10)            # 0  2  4 ... 20

for(k in 5:11) {
  cat("k=", k, ": "); print(diff(range(pretty(100 + c(0, pi*10^-k)))))}

##-- more bizarre, when  min(x) == max(x):
pretty(pi)

add.names <- function(v) { names(v) <- paste(v); v}
utils::str(lapply(add.names(-10:20), pretty))
## min.n = 0  returns a length-1 vector "if pretty":
utils::str(lapply(add.names(0:20),  pretty, min.n = 0))
sapply(    add.names(0:20),   pretty, min.n = 4)
```

```
pretty(1.234e100)
pretty(1001.1001)
pretty(1001.1001, shrink.sml = 0.2)
for(k in -7:3)
  cat("shrink=", formatC(2^k, width = 9),":",
      formatC(pretty(1001.1001, shrink.sml = 2^k), width = 6),"\n")
```

`Primitive` *Look Up a Primitive Function*

Description

`.Primitive` looks up by name a 'primitive' (internally implemented) function.

Usage

```
.Primitive(name)
```

Arguments

`name`	name of the R function.

Details

The advantage of `.Primitive` over `.Internal` functions is the potential efficiency of argument passing, and that positional matching can be used where desirable, e.g. in `switch`. For more details, see the 'R Internals' manual.

All primitive functions are in the base namespace.

This function is almost never used: `` `name` `` or, more carefully, `get(name, envir = baseenv())` work equally well and do not depend on knowing which functions are primitive (which does change as R evolves).

See Also

`is.primitive` showing that primitive functions come in two types (`typeof`), `.Internal`.

Examples

```
mysqrt <- .Primitive("sqrt")
c
.Internal # this one *must* be primitive!
`if` # need backticks
```

print *Print Values*

Description

print prints its argument and returns it *invisibly* (via invisible(x)). It is a generic function which means that new printing methods can be easily added for new classes.

Usage

```
print(x, ...)

## S3 method for class 'factor'
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table'
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0",
      right = is.numeric(x) || is.complex(x),
      justify = "none", ...)

## S3 method for class 'function'
print(x, useSource = TRUE, ...)
```

Arguments

x	an object used to select a method.
...	further arguments passed to or from other methods.
quote	logical, indicating whether or not strings should be printed with surrounding quotes.
max.levels	integer, indicating how many levels should be printed for a factor; if 0, no extra "Levels" line will be printed. The default, NULL, entails choosing max.levels such that the levels print on one line of width width.
width	only used when max.levels is NULL, see above.
digits	minimal number of *significant* digits, see print.default.
na.print	character string (or NULL) indicating NA values in printed output, see print.default.
zero.print	character specifying how zeros (0) should be printed; for sparse tables, using "." can produce more readable results, similar to printing sparse matrices in **Matrix**.
right	logical, indicating whether or not strings should be right aligned.
justify	character indicating if strings should left- or right-justified or left alone, passed to format.
useSource	logical indicating if internally stored source should be used for printing when present, e.g., if options(keep.source = TRUE) has been in use.

Details

The default method, `print.default` has its own help page. Use `methods("print")` to get all the methods for the `print` generic.

`print.factor` allows some customization and is used for printing `ordered` factors as well.

`print.table` for printing `table`s allows other customization. As of R 3.0.0, it only prints a description in case of a table with 0-extents (this can happen if a classifier has no valid data).

See `noquote` as an example of a class whose main purpose is a specific `print` method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

The default method `print.default`, and help for the methods above; further `options`, `noquote`.

For more customizable (but cumbersome) printing, see `cat`, `format` or also `write`. For a simple prototypical print method, see `.print.via.format` in package **tools**.

Examples

```
require(stats)

ts(1:20)  #-- print is the "Default function" --> print.ts(.) is called
for(i in 1:3) print(1:i)

## Printing of factors
attenu$station ## 117 levels -> 'max.levels' depending on width

## ordered factors: levels  "l1 < l2 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df = 1.8)))
t2 <- round(abs(rt(200, df = 1.4)))
table(t1, t2) # simple
print(table(t1, t2), zero.print = ".") # nicer to read

## same for non-integer "table":
T <- table(t2,t1)
T <- T * (1+round(rlnorm(length(T)))/4)
print(T, zero.print = ".") # quite nicer,
print.table(T[,2:8] * 1e9, digits=3, zero.print = ".")
## still slightly inferior to  Matrix::Matrix(T)  for larger T

## Corner cases with empty extents:
table(1, NA) # < table of extent 1 x 0 >
```

`print.data.frame` *Printing Data Frames*

Description

Print a data frame.

Usage

```
## S3 method for class 'data.frame'
print(x, ..., digits = NULL,
      quote = FALSE, right = TRUE, row.names = TRUE, max = NULL)
```

Arguments

`x`	object of class `data.frame`.
`...`	optional arguments to `print` methods.
`digits`	the minimum number of significant digits to be used: see `print.default`.
`quote`	logical, indicating whether or not entries should be printed with surrounding quotes.
`right`	logical, indicating whether or not strings should be right-aligned. The default is right-alignment.
`row.names`	logical (or character vector), indicating whether (or what) row names should be printed.
`max`	numeric or `NULL`, specifying the maximal number of entries to be printed. By default, when `NULL`, `getOption("max.print")` used.

Details

This calls `format` which formats the data frame column-by-column, then converts to a character matrix and dispatches to the `print` method for matrices.

When `quote = TRUE` only the entries are quoted not the row names nor the column names.

See Also

`data.frame`.

Examples

```
(dd <- data.frame(x = 1:8, f = gl(2,4), ch = I(letters[1:8])))
     # print() with defaults
print(dd, quote = TRUE, row.names = FALSE)
     # suppresses row.names and quotes all entries
```

print.default *Default Printing*

Description

print.default is the *default* method of the generic print function which prints its argument.

Usage

```
## Default S3 method:
print(x, digits = NULL, quote = TRUE,
      na.print = NULL, print.gap = NULL, right = FALSE,
      max = NULL, width = NULL, useSource = TRUE, ...)
```

Arguments

x	the object to be printed.
digits	a non-null value for digits specifies the minimum number of significant digits to be printed in values. The default, NULL, uses getOption("digits"). (For the interpretation for complex numbers see signif.) Non-integer values will be rounded down, and only values greater than or equal to 1 and no greater than 22 are accepted.
quote	logical, indicating whether or not strings (characters) should be printed with surrounding quotes.
na.print	a character string which is used to indicate NA values in printed output, or NULL (see 'Details').
print.gap	a non-negative integer ≤ 1024, or NULL (meaning 1), giving the spacing between adjacent columns in printed vectors, matrices and arrays.
right	logical, indicating whether or not strings should be right aligned. The default is left alignment.
max	a non-null value for max specifies the approximate maximum number of entries to be printed. The default, NULL, uses getOption("max.print"): see that help page for more details.
width	controls the maximum number of columns on a line used in printing vectors, matrices, etc. The default, NULL, uses getOption("width"): see that help page for more details including allowed values.
useSource	logical, indicating whether to use source references or copies rather than deparsing language objects. The default is to use the original source if it is available.
...	further arguments to be passed to or from other methods. They are ignored in this function.

Details

The default for printing NAs is to print NA (without quotes) unless this is a character NA *and* quote = FALSE, when '<NA>' is printed.

The same number of decimal places is used throughout a vector. This means that digits specifies the minimum number of significant digits to be used, and that at least one entry will be encoded with that minimum number. However, if all the encoded elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit. Decimal points are only included if at least one decimal place is selected.

You can suppress "exponential" / scientific notation in printing of numbers (atomic vectors x), via format(., scientific=FALSE), see the prI() example below, or also by increasing global option scipen, e.g., options(scipen = 12).

Attributes are printed respecting their class(es), using the values of digits to print.default, but using the default values (for the methods called) of the other arguments.

Option width controls the printing of vectors, matrices and arrays, and option deparse.cutoff controls the printing of language objects such as calls and formulae.

When the **methods** package is attached, print will call show for R objects with formal classes ('S4') if called with no optional arguments.

Large number of digits

Note that for large values of digits, currently for digits >= 16, the calculation of the number of significant digits will depend on the platform's internal (C library) implementation of 'sprintf()' functionality.

Single-byte locales

If a non-printable character is encountered during output, it is represented as one of the ANSI escape sequences ('\a', '\b', '\f', '\n', '\r', '\t', '\v', '\\' and '\0': see Quotes), or failing that as a 3-digit octal code: for example the UK currency pound sign in the C locale (if implemented correctly) is printed as '\243'. Which characters are non-printable depends on the locale. (Because some versions of Windows get this wrong, all bytes with the upper bit set are regarded as printable on Windows in a single-byte locale.)

Unicode and other multi-byte locales

In all locales, the characters in the ASCII range ('0x00' to '0x7f') are printed in the same way, as-is if printable, otherwise via ANSI escape sequences or 3-digit octal escapes as described for single-byte locales. Whether a character is printable depends on the current locale and the operating system (C library).

Multi-byte non-printing characters are printed as an escape sequence of the form '\uxxxx' or '\Uxxxxxxxx' (in hexadecimal). This is the internal code for the wide-character representation of the character. If this is not known to be Unicode code points, a warning is issued. The only known exceptions are certain Japanese ISO 2022 locales on commercial Unixes, which use a concatenation of the bytes: it is unlikely that R compiles on such a system.

It is possible to have a character string in a character vector that is not valid in the current locale. If a byte is encountered that is not part of a valid character it is printed in hex in the form '\xab' and

this is repeated until the start of a valid character. (This will rapidly recover from minor errors in UTF-8.)

See Also

The generic `print`, `options`. The `"noquote"` class and print method.

`encodeString`, which encodes a character vector the way it would be printed.

Examples

```
pi
print(pi, digits = 16)
LETTERS[1:16]
print(LETTERS, quote = FALSE)

M <- cbind(I = 1, matrix(1:10000, ncol = 10,
                         dimnames = list(NULL, LETTERS[1:10])))
utils::head(M)        # makes more sense than
print(M, max = 1000)  # prints 90 rows and a message about omitting 910

(x <- 2^seq(-8, 30, by=1/4)) # auto-prints; by default all in "exponential" format
prI <- function(x) noquote(format(x, scientific = FALSE))
prI(x) # prints more "nicely" (using a bit more space)
```

---

prmatrix *Print Matrices, Old-style*

Description

An earlier method for printing matrices, provided for S compatibility.

Usage

```
prmatrix(x, rowlab =, collab =,
         quote = TRUE, right = FALSE, na.print = NULL, ...)
```

Arguments

`x`	numeric or character matrix.
`rowlab, collab`	(optional) character vectors giving row or column names respectively. By default, these are taken from `dimnames(x)`.
`quote`	logical; if `TRUE` and `x` is of mode `"character"`, *quotes* (‘"’) are used.
`right`	if `TRUE` and `x` is of mode `"character"`, the output columns are *right*-justified.
`na.print`	how `NA`s are printed. If this is non-null, its value is used to represent `NA`.
`...`	arguments for `print` methods.

Details

prmatrix is an earlier form of print.matrix, and is very similar to the S function of the same name.

Value

Invisibly returns its argument, x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

print.default, and other print methods.

Examples

```
prmatrix(m6 <- diag(6), rowlab = rep("", 6), collab = rep("", 6))

chm <- matrix(scan(system.file("help", "AnIndex", package = "splines"),
                   what = ""), , 2, byrow = TRUE)
chm  # uses print.matrix()
prmatrix(chm, collab = paste("Column", 1:3), right = TRUE, quote = FALSE)
```

---

proc.time *Running Time of R*

Description

proc.time determines how much real and CPU time (in seconds) the currently running R process has already taken.

Usage

```
proc.time()
```

Details

proc.time returns five elements for backwards compatibility, but its print method prints a named vector of length 3. The first two entries are the total user and system CPU times of the current R process and any child processes on which it has waited, and the third entry is the 'real' elapsed time since the process was started.

Value

An object of class "proc_time" which is a numeric vector of length 5, containing the user, system, and total elapsed times for the currently running R process, and the cumulative sum of user and system times of any child processes spawned by it on which it has waited. (The print method uses the summary method to combine the child times with those of the main process.)

The definition of 'user' and 'system' times is from your OS. Typically it is something like

The 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process.

Times of child processes are not available on Windows and will always be given as NA.

The resolution of the times will be system-specific and on Unix-alikes times are rounded down to milliseconds. On modern systems they will be that accurate, but on older systems they might be accurate to 1/100 or 1/60 sec. They are typically available to 10ms on Windows.

This is a primitive function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

system.time for timing an R expression, gc.time for how much of the time was spent in garbage collection.

setTimeLimit to *limit* the CPU or elapsed time for the session or an expression.

Examples

```
## a way to time an R expression: system.time is preferred
ptm <- proc.time()
for (i in 1:50) mad(stats::runif(500))
proc.time() - ptm
```

---

prod *Product of Vector Elements*

Description

prod returns the product of all the values present in its arguments.

Usage

```
prod(..., na.rm = FALSE)
```

Arguments

...	numeric or complex or logical vectors.
na.rm	logical. Should missing values be removed?

Details

If na.rm is FALSE an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were numeric(0).

Value

The product, a numeric (of type "double") or complex vector of length one. **NB:** the product of an empty set is one, by definition.

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

sum, cumprod, cumsum.

'plotmath' for the use of prod in plot annotation.

Examples

```
print(prod(1:7)) == print(gamma(8))
```

---

proportions *Express Table Entries as Fraction of Marginal Table*

Description

Returns conditional proportions given margins, i.e., entries of x, divided by the appropriate marginal sums.

Usage

```
proportions(x, margin = NULL)
prop.table(x, margin = NULL)
```

Arguments

x	an array, usually a table.
margin	a vector giving the margins to split by. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. When x has named dimnames, it can be a character vector selecting dimension names.

Value

A table or array like x, expressed relative to margin.

Note

prop.table is an earlier name, retained for back-compatibility.

Author(s)

Peter Dalgaard

See Also

marginSums.

apply and sweep are more general mechanisms for sweeping out marginal statistics.

Examples

```
m <- matrix(1:4, 2)
m
proportions(m, 1)

DF <- as.data.frame(UCBAdmissions)
tbl <- xtabs(Freq ~ Gender + Admit, DF)
tbl
proportions(tbl, "Gender")
```

---

pushBack *Push Text Back on to a Connection*

Description

Functions to push back text lines onto a connection, and to enquire how many lines are currently pushed back.

Usage

```
pushBack(data, connection, newLine = TRUE,
         encoding = c("", "bytes", "UTF-8"))
pushBackLength(connection)
clearPushBack(connection)
```

Arguments

data	a character vector.
connection	a connection.
newLine	logical. If true, a newline is appended to each string pushed back.
encoding	character string, partially matched. See details.

Details

Several character strings can be pushed back on one or more occasions. The occasions form a stack, so the first line to be retrieved will be the first string from the last call to pushBack. Lines which are pushed back are read prior to the normal input from the connection, by the normal text-reading functions such as readLines and scan.

Pushback is only allowed for readable connections in text mode.

Not all uses of connections respect pushbacks, in particular the input connection is still wired directly, so for example parsing commands from the console and scan("") ignore pushbacks on stdin.

When character strings with a marked encoding (see Encoding) are pushed back they are converted to the current encoding if encoding = "". This may involve representing characters as '<U+xxxx>' if they cannot be converted. They will be converted to UTF-8 if encoding = "UTF-8" or left as-is if encoding = "bytes".

Value

pushBack and clearPushBack() return nothing, invisibly.

pushBackLength returns the number of lines currently pushed back.

See Also

connections, readLines.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
pushBack(c("aa", "bb"), zz)
pushBackLength(zz)
readLines(zz, 1)
pushBackLength(zz)
readLines(zz, 1)
readLines(zz, 1)
close(zz)
```

qr *The QR Decomposition of a Matrix*

Description

qr computes the QR decomposition of a matrix.

Usage

```
qr(x, ...)
## Default S3 method:
qr(x, tol = 1e-07 , LAPACK = FALSE, ...)

qr.coef(qr, y)
qr.qy(qr, y)
qr.qty(qr, y)
qr.resid(qr, y)
qr.fitted(qr, y, k = qr$rank)
qr.solve(a, b, tol = 1e-7)
## S3 method for class 'qr'
solve(a, b, ...)

is.qr(x)
as.qr(x)
```

Arguments

x	a numeric or complex matrix whose QR decomposition is to be computed. Logical matrices are coerced to numeric.
tol	the tolerance for detecting linear dependencies in the columns of x. Only used if LAPACK is false and x is real.
qr	a QR decomposition of the type computed by qr.
y, b	a vector or matrix of right-hand sides of equations.
a	a QR decomposition or (qr.solve only) a rectangular matrix.
k	effective rank.
LAPACK	logical. For real x, if true use LAPACK otherwise use LINPACK (the default).
...	further arguments passed to or from other methods.

Details

The QR decomposition plays an important role in many statistical techniques. In particular it can be used to solve the equation $\boldsymbol{Ax} = \boldsymbol{b}$ for given matrix $\boldsymbol{A}$, and vector $\boldsymbol{b}$. It is useful for computing regression coefficients and in applying the Newton-Raphson algorithm.

The functions qr.coef, qr.resid, and qr.fitted return the coefficients, residuals and fitted values obtained when fitting y to the matrix with QR decomposition qr. (If pivoting is used, some

of the coefficients will be NA.) qr.qy and qr.qty return Q %*% y and t(Q) %*% y, where Q is the (complete) $\boldsymbol{Q}$ matrix.

All the above functions keep dimnames (and names) of x and y if there are any.

solve.qr is the method for solve for qr objects. qr.solve solves systems of equations via the QR decomposition: if a is a QR decomposition it is the same as solve.qr, but if a is a rectangular matrix the QR decomposition is computed first. Either will handle over- and under-determined systems, providing a least-squares fit if appropriate.

is.qr returns TRUE if x is a list and inherits from "qr".

It is not possible to coerce objects to mode "qr". Objects either are QR decompositions or they are not.

The LINPACK interface is restricted to matrices x with less than 2^{31} elements.

qr.fitted and qr.resid only support the LINPACK interface.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The QR decomposition of the matrix as computed by LINPACK(*) or LAPACK. The components in the returned value correspond directly to the values returned by DQRDC(2)/DGEQP3/ZGEQP3.

qr	a matrix with the same dimensions as x. The upper triangle contains the $\boldsymbol{R}$ of the decomposition and the lower triangle contains information on the $\boldsymbol{Q}$ of the decomposition (stored in compact form). Note that the storage used by DQRDC and DGEQP3 differs.
qraux	a vector of length ncol(x) which contains additional information on $\boldsymbol{Q}$.
rank	the rank of x as computed by the decomposition(*): always full rank in the LAPACK case.
pivot	information on the pivoting strategy used during the decomposition.

Non-complex QR objects computed by LAPACK have the attribute "useLAPACK" with value TRUE.

*) dqrdc2 instead of LINPACK's DQRDC

In the (default) LINPACK case (LAPACK = FALSE), qr() uses a *modified* version of LINPACK's DQRDC, called 'dqrdc2'. It differs by using the tolerance tol for a pivoting strategy which moves columns with near-zero 2-norm to the right-hand edge of the x matrix. This strategy means that sequential one degree-of-freedom effects can be computed in a natural way.

Note

To compute the determinant of a matrix (do you *really* need it?), the QR decomposition is much more efficient than using eigenvalues (eigen). See det.

Using LAPACK (including in the complex case) uses column pivoting and does not attempt to detect rank-deficient matrices.

Source

For qr, the LINPACK routine DQRDC (but modified to dqrdc2(*)) and the LAPACK routines DGEQP3 and ZGEQP3. Further LINPACK and LAPACK routines are used for qr.coef, qr.qy and qr.aty.

LAPACK and LINPACK are from https://netlib.org/lapack/ and https://netlib.org/linpack/ and their guides are listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

See Also

qr.Q, qr.R, qr.X for reconstruction of the matrices. lm.fit, lsfit, eigen, svd.

det (using qr) to compute the determinant of a matrix.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, `+`) }
h9 <- hilbert(9); h9
qr(h9)$rank           #--> only 7
qrh9 <- qr(h9, tol = 1e-10)
qrh9$rank             #--> 9
##-- Solve linear equation system  H %*% x = y :
y <- 1:9/10
x <- qr.solve(h9, y, tol = 1e-10) # or equivalently :
x <- qr.coef(qrh9, y) #-- is == but much better than
                      #-- solve(h9) %*% y
h9 %*% x              # = y


## overdetermined system
A <- matrix(runif(12), 4)
b <- 1:4
qr.solve(A, b) # or solve(qr(A), b)
solve(qr(A, LAPACK = TRUE), b)
# this is a least-squares solution, cf. lm(b ~ 0 + A)

## underdetermined system
A <- matrix(runif(12), 3)
b <- 1:3
qr.solve(A, b)
solve(qr(A, LAPACK = TRUE), b)
# solutions will have one zero, not necessarily the same one
```

QR.Auxiliaries *Reconstruct the Q, R, or X Matrices from a QR Object*

Description

Returns the original matrix from which the object was constructed or the components of the decomposition.

Usage

```
qr.X(qr, complete = FALSE, ncol =)
qr.Q(qr, complete = FALSE, Dvec =)
qr.R(qr, complete = FALSE)
```

Arguments

qr	object representing a QR decomposition. This will typically have come from a previous call to qr or lsfit.
complete	logical expression of length 1. Indicates whether an arbitrary orthogonal completion of the $\boldsymbol{Q}$ or $\boldsymbol{X}$ matrices is to be made, or whether the $\boldsymbol{R}$ matrix is to be completed by binding zero-value rows beneath the square upper triangle.
ncol	integer in the range 1:nrow(qr$qr). The number of columns to be in the reconstructed $\boldsymbol{X}$. The default when complete is FALSE is the first min(ncol(X), nrow(X)) columns of the original $\boldsymbol{X}$ from which the qr object was constructed. The default when complete is TRUE is a square matrix with the original $\boldsymbol{X}$ in the first ncol(X) columns and an arbitrary orthogonal completion (unitary completion in the complex case) in the remaining columns.
Dvec	vector (not matrix) of diagonal values. Each column of the returned $\boldsymbol{Q}$ will be multiplied by the corresponding diagonal value. Defaults to all 1s.

Value

qr.X returns $\boldsymbol{X}$, the original matrix from which the qr object was constructed, provided ncol(X) <= nrow(X). If complete is TRUE or the argument ncol is greater than ncol(X), additional columns from an arbitrary orthogonal (unitary) completion of X are returned.

qr.Q returns part or all of **Q**, the orthogonal (unitary) transformation of order nrow(X) represented by qr. If complete is TRUE, **Q** has nrow(X) columns. If complete is FALSE, **Q** has ncol(X) columns. When Dvec is specified, each column of **Q** is multiplied by the corresponding value in Dvec.

Note that qr.Q(qr, *) is a special case of qr.qy(qr, y) (with a "diagonal" y), and qr.X(qr, *) is basically qr.qy(qr, R) (apart from pivoting and dimnames setting).

qr.R returns **R**. This may be pivoted, e.g., if a <- qr(x) then x[, a$pivot] = **QR**. The number of rows of **R** is either nrow(X) or ncol(X) (and may depend on whether complete is TRUE or FALSE).

See Also

qr, qr.qy.

Examples

```
p <- ncol(x <- LifeCycleSavings[, -1]) # not the 'sr'
qrstr <- qr(x)   # dim(x) == c(n,p)
qrstr $ rank # = 4 = p
Q <- qr.Q(qrstr) # dim(Q) == dim(x)
R <- qr.R(qrstr) # dim(R) == ncol(x)
X <- qr.X(qrstr) # X == x
range(X - as.matrix(x))  # ~ < 6e-12
## X == Q %*% R if there has been no pivoting, as here:
all.equal(unname(X),
          unname(Q %*% R))

# example of pivoting
x <- cbind(int = 1,
           b1 = rep(1:0, each = 3), b2 = rep(0:1, each = 3),
           c1 = rep(c(1,0,0), 2), c2 = rep(c(0,1,0), 2), c3 = rep(c(0,0,1),2))
x # is singular, columns "b2" and "c3" are "extra"
a <- qr(x)
zapsmall(qr.R(a)) # columns are int b1 c1 c2 b2 c3
a$pivot
pivI <- sort.list(a$pivot) # the inverse permutation
all.equal (x,            qr.Q(a) %*% qr.R(a)) # no, no
stopifnot(
 all.equal(x[, a$pivot], qr.Q(a) %*% qr.R(a)),          # TRUE
 all.equal(x           , qr.Q(a) %*% qr.R(a)[, pivI]))  # TRUE too!
```

---

quit *Terminate an R Session*

Description

The function quit or its alias q terminate the current R session.

Usage

```
quit(save = "default", status = 0, runLast = TRUE)
   q(save = "default", status = 0, runLast = TRUE)
```

Arguments

save	a character string indicating whether the environment (workspace) should be saved, one of "no", "yes", "ask" or "default".
status	the (numerical) error status to be returned to the operating system, where relevant. Conventionally 0 indicates successful completion.
runLast	should .Last() be executed?

Details

save must be one of "no", "yes", "ask" or "default". In the first case the workspace is not saved, in the second it is saved and in the third the user is prompted and can also decide *not* to quit. The default is to ask in interactive use but may be overridden by command-line arguments (which must be supplied in non-interactive use).

Immediately *before* normal termination, .Last() is executed if the function .Last exists and runLast is true. If in interactive use there are errors in the .Last function, control will be returned to the command prompt, so do test the function thoroughly. There is a system analogue, .Last.sys(), which is run after .Last() if runLast is true.

Exactly what happens at termination of an R session depends on the platform and GUI interface in use. A typical sequence is to run .Last() and .Last.sys() (unless runLast is false), to save the workspace if requested (and in most cases also to save the session history: see savehistory), then run any finalizers (see reg.finalizer) that have been set to be run on exit, close all open graphics devices, remove the session temporary directory and print any remaining warnings (e.g., from .Last() and device closure).

Some error status values are used by R itself. The default error handler for non-interactive use effectively calls q("no", 1, FALSE) and returns error status 1. Error status 2 is used for R 'suicide', that is a catastrophic failure, and other small numbers are used by specific ports for initialization failures. It is recommended that users choose statuses of 10 or more.

Valid values of status are system-dependent, but 0:255 are normally valid. (Many OSes will report the last byte of the value, that is report the value modulo 256. But not all.)

Warning

The value of .Last is for the end user to control: as it can be replaced later in the session, it cannot safely be used programmatically, e.g. by a package. The other way to set code to be run at the end of the session is to use a *finalizer*: see reg.finalizer.

Note

The R.app GUI on macOS has its own version of these functions with slightly different behaviour for the save argument (the GUI's 'Startup' preferences for this action are taken into account).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

.First for setting things on startup.

Examples

```
## Not run: ## Unix-flavour example
.Last <- function() {
  graphics.off() # close devices before printing
  cat("Now sending PDF graphics to the printer:\n")
```

```
  system("lpr Rplots.pdf")
  cat("bye bye...\n")
}
quit("yes")
## End(Not run)
```

`Quotes` *Quotes*

Description

Descriptions of the various uses of quoting in R.

Details

Three types of quotes are part of the syntax of R: single and double quotation marks and the backtick (or back quote, ‘`’). In addition, backslash is used to escape the following character inside character constants.

Character constants

Single and double quotes delimit character constants. They can be used interchangeably but double quotes are preferred (and character constants are printed using double quotes), so single quotes are normally only used to delimit character constants containing double quotes.

Backslash is used to start an escape sequence inside character constants. Escaping a character not in the following table is an error.

Single quotes need to be escaped by backslash in single-quoted strings, and double quotes in double-quoted strings.

‘`\n`’	newline (aka ‘line feed’)
‘`\r`’	carriage return
‘`\t`’	tab
‘`\b`’	backspace
‘`\a`’	alert (bell)
‘`\f`’	form feed
‘`\v`’	vertical tab
‘`\\`’	backslash ‘`\`’
‘`\'`’	ASCII apostrophe ‘’’
‘`\"`’	ASCII quotation mark ‘"’
‘`\‘`’	ASCII grave accent (backtick) ‘‘’
‘`\nnn`’	character with given octal code (1, 2 or 3 digits)
‘`\xnn`’	character with given hex code (1 or 2 hex digits)
‘`\unnnn`’	Unicode character with given code (1–4 hex digits)
‘`\Unnnnnnnn`’	Unicode character with given code (1–8 hex digits)

Alternative forms for the last two are '\u{nnnn}' and '\U{nnnnnnnn}'. All except the Unicode escape sequences are also supported when reading character strings by scan and read.table if allowEscapes = TRUE. Unicode escapes can be used to enter Unicode characters not in the current locale's charset (when the string will be stored internally in UTF-8). The maximum allowed value for '\nnn' is '\377' (the same character as '\xff').

As from R 4.1.0 the largest allowed '\U' value is '\U10FFFF', the maximum Unicode point.

The parser does not allow the use of both octal/hex and Unicode escapes in a single string.

These forms will also be used by print.default when outputting non-printable characters (including backslash).

Embedded NULs are not allowed in character strings, so using escapes (such as '\0') for a NUL will result in the string being truncated at that point (usually with a warning).

Raw character constants are also available using a syntax similar to the one used in C++: r"(...)" with ... any character sequence, except that it must not contain the closing sequence ')"'. The delimiter pairs [] and {} can also be used, and R can be used in place of r. For additional flexibility, a number of dashes can be placed between the opening quote and the opening delimiter, as long as the same number of dashes appear between the closing delimiter and the closing quote.

Names and Identifiers

Identifiers consist of a sequence of letters, digits, the period (.) and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit. Reserved words are not valid identifiers.

The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

Such identifiers are also known as *syntactic names* and may be used directly in R code. Almost always, other names can be used provided they are quoted. The preferred quote is the backtick (``), and deparse will normally use it, but under many circumstances single or double quotes can be used (as a character constant will often be converted to a name). One place where backticks may be essential is to delimit variable names in formulae: see formula.

Note

UTF-16 surrogate pairs in '\unnnn\uoooo' form will be converted to a single Unicode point, so for example '\uD834\uDD1E' gives the single character '\U1D11E'. However, unpaired values in the surrogate range such as in the string "abc\uD834de" will be converted to a non-standard-conformant UTF-8 string (as is done by most other software): this may change in future.

See Also

Syntax for other aspects of the syntax.

sQuote for quoting English text.

shQuote for quoting OS commands.

The 'R Language Definition' manual.

Examples

```
'single quotes can be used more-or-less interchangeably'
"with double quotes to create character vectors"

## Single quotes inside single-quoted strings need backslash-escaping.
## Ditto double quotes inside double-quoted strings.
##
identical('"It\'s alive!", he screamed.',
          "\"It's alive!\", he screamed.") # same

## Backslashes need doubling, or they have a special meaning.
x <- "In ALGOL, you could do logical AND with /\\."
print(x)      # shows it as above ("input-like")
writeLines(x) # shows it as you like it ;-)

## Single backslashes followed by a letter are used to denote
## special characters like tab(ulator)s and newlines:
x <- "long\tlines can be\nbroken with newlines"
writeLines(x) # see also ?strwrap

## Backticks are used for non-standard variable names.
## (See make.names and ?Reserved for what counts as
## non-standard.)
`x y` <- 1:5
`x y`
d <- data.frame(`1st column` = rchisq(5, 2), check.names = FALSE)
d$`1st column`

## Backslashes followed by up to three numbers are interpreted as
## octal notation for ASCII characters.
"\110\145\154\154\157\40\127\157\162\154\144\41"

## \x followed by up to two numbers is interpreted as
## hexadecimal notation for ASCII characters.
(hw1 <- "\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x21")

## Mixing octal and hexadecimal in the same string is OK
(hw2 <- "\110\x65\154\x6c\157\x20\127\x6f\162\x6c\144\x21")

## \u is also hexadecimal, but supports up to 4 digits,
## using Unicode specification.  In the previous example,
## you can simply replace \x with \u.
(hw3 <- "\u48\u65\u6c\u6c\u6f\u20\u57\u6f\u72\u6c\u64\u21")

## The last three are all identical to
hw <- "Hello World!"
stopifnot(identical(hw, hw1), identical(hw1, hw2), identical(hw2, hw3))

## Using Unicode makes more sense for non-latin characters.
(nn <- "\u0126\u0119\u1114\u022d\u2001\u03e2\u0954\u0f3f\u13d3\u147b\u203c")

## Mixing \x and \u throws a _parse_ error (which is not catchable!)
```

```
## Not run:
  "\x48\u65\x6c\u6c\x6f\u20\x57\u6f\x72\u6c\x64\u21"

## End(Not run)
##   -->   Error: mixing Unicode and octal/hex escapes .....

## \U works like \u, but supports up to six hex digits.
## So we can replace \u with \U in the previous example.
n2 <- "\U0126\U0119\U1114\U022d\U2001\U03e2\U0954\U0f3f\U13d3\U147b\U203c"
stopifnot(identical(nn, n2))

## Under systems supporting multi-byte locales (and not Windows),
## \U also supports the rarer characters outside the usual 16^4 range.
## See the R language manual,
## https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Literal-constants
## and bug 16098 https://bugs.r-project.org/show_bug.cgi?id=16098
## This character may or not be printable (the platform decides)
## and if it is, may not have a glyph in the font used.
"\U1d4d7" # On Windows this used to give the incorrect value of "\Ud4d7"

## nul characters (for terminating strings in C) are not allowed (parse errors)
## Not run:
  "foo\0bar"     # Error: nul character not allowed (line 1)
  "foo\u0000bar" # same error

## End(Not run)

## A Windows path written as a raw string constant:
r"(c:\Program files\R)"

## More raw strings:
r"{(\1\2)}"
r"(use both "double" and 'single' quotes)"
r"---(\1--)-)---"
```

`R.Version` *Version Information*

Description

`R.Version()` provides detailed information about the version of R running.

`R.version` is a variable (a `list`) holding this information (and `version` is a copy of it for S compatibility).

Usage

```
R.Version()
R.version
R.version.string
```

```
version

R_compiled_by()
```

Details

This gives details of the OS under which R was built, not the one under which it is currently running (for which see Sys.info).

Note that OS names might not be what you expect: for example macOS Mavericks 10.9.4 identifies itself as 'darwin13.3.0', Linux usually as 'linux-gnu', Solaris 10 as 'solaris2.10' and Windows as 'mingw32'.

R.version$crt is supported on Windows since R 4.2.0 and returns "ucrt" to denote the Universal C Runtime. It would return "msvcrt" for the older Microsoft Visual C++ Runtime (but R does not use that runtime since 4.2.0).

Value

R.Version returns a list with character-string components

platform	the platform for which R was built. A triplet of the form CPU-VENDOR-OS, as determined by the configure script. E.g, "i686-unknown-linux-gnu" or "i386-pc-mingw32".
arch	the architecture (CPU) R was built on/for.
os	the underlying operating system.
crt	the C runtime on Windows.
system	CPU and OS, separated by a comma.
status	the status of the version (e.g., "alpha").
major	the major version number.
minor	the minor version number, including the patch level.
year	the year the version was released.
month	the month the version was released.
day	the day the version was released.
svn rev	the Subversion revision number, which should be either "unknown" or a single number. (A range of numbers or a number with 'M' or 'S' appended indicates inconsistencies in the sources used to build this version of R.)
language	always "R".
version.string	a character string concatenating some of the info above, useful for plotting, etc.

R.version and version are lists of class "simple.list" which has a print method.

R_compiled_by returns a two-element character vector giving details of the C and Fortran compilers used to build R. (Empty strings if no information is available.)

Note

Do *not* use `R.version$os` to test the platform the code is running on: use `.Platform$OS.type` instead. Slightly different versions of the OS may report different values of `R.version$os`, as may different versions of R. Alternatively, `osVersion` typically contains more details about the platform R is running on.

`R.version.string` is a copy of `R.version$version.string` for simplicity and backwards compatibility.

See Also

`sessionInfo` which provides additional information; `getRversion` typically used inside R code, `osVersion`, `.Platform`, `Sys.info`.

Examples

```
require(graphics)

R.version$os # to check how lucky you are ...
plot(0) # any plot
mtext(R.version.string, side = 1, line = 4, adj = 1) # a useful bottom-right note

## a good way to detect macOS:
if(grepl("^darwin", R.version$os)) message("running on macOS")

## Short R version string, ("space free", useful in file/directory names;
##                          also fine for unreleased versions of R):
shortRversion <- function() {
   rvs <- R.version.string
   if(grepl("devel", (st <- R.version$status)))
       rvs <- sub(paste0(" ",st," "), "-devel_", rvs, fixed=TRUE)
   gsub("[()]", "", gsub(" ", "_", sub(" version ", "-", rvs)))
}
shortRversion()
```

Random *Random Number Generation*

Description

`.Random.seed` is an integer vector, containing the random number generator (RNG) **state** for random number generation in R. It can be saved and restored, but should not be altered by the user.

`RNGkind` is a more friendly interface to query or set the kind of RNG in use.

`RNGversion` can be used to set the random generators as they were in an earlier R version (for reproducibility).

`set.seed` is the recommended way to specify seeds.

Usage

```
.Random.seed <- c(rng.kind, n1, n2, ...)

RNGkind(kind = NULL, normal.kind = NULL, sample.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL, normal.kind = NULL, sample.kind = NULL)
```

Arguments

`kind`	character or `NULL`. If `kind` is a character string, set R's RNG to the kind desired. Use `"default"` to return to the R default. See 'Details' for the interpretation of `NULL`.
`normal.kind`	character string or `NULL`. If it is a character string, set the method of Normal generation. Use `"default"` to return to the R default. `NULL` makes no change.
`sample.kind`	character string or `NULL`. If it is a character string, set the method of discrete uniform generation (used in `sample`, for instance). Use `"default"` to return to the R default. `NULL` makes no change.
`seed`	a single value, interpreted as an integer, or `NULL` (see 'Details').
`vstr`	a character string containing a version number, e.g., `"1.6.2"`. The default RNG configuration of the current R version is used if `vstr` is greater than the current version.
`rng.kind`	integer code in `0:k` for the above `kind`.
`n1, n2, ...`	integers. See the details for how many are required (which depends on `rng.kind`).

Details

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is `"Mersenne-Twister"`.

`"Wichmann-Hill"` The seed, `.Random.seed[-1] == r[1:3]` is an integer vector of length 3, where each `r[i]` is in `1:(p[i] - 1)`, where `p` is the length 3 vector of primes, `p = (30269, 30307, 30323)`. The Wichmann–Hill generator has a cycle length of 6.9536×10^{12} (= `prod(p-1)/4`, see *Applied Statistics* (1984) **33**, 123 which corrects the original article). It exhibits 12 clear failures in the TestU01 Crush suite and 22 in the BigCrush suite (L'Ecuyer, 2007).

`"Marsaglia-Multicarry"`**:** A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing list '`sci.stat.math`'. It has a period of more than 2^{60}.

It exhibits 40 clear failures in L'Ecuyer's TestU01 Crush suite. Combined with Ahrens-Dieter or Kinderman-Ramage it exhibits deviations from normality even for univariate distribution generation. See PR#18168 for a discussion.

The seed is two integers (all values allowed).

`"Super-Duper"`**:** Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of $\approx 4.6 \times 10^{18}$ for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd).

We use the implementation by Reeds et al. (1982–84).

The two seeds are the Tausworthe and congruence long integers, respectively.

It exhibits 25 clear failures in the TestU01 Crush suite (L'Ecuyer, 2007).

`"Mersenne-Twister"`: From Matsumoto and Nishimura (1998); code updated in 2002. A twisted GFSR with period $2^{19937} - 1$ and equidistribution in 623 consecutive dimensions (over the whole period). The 'seed' is a 624-dimensional set of 32-bit integers plus a current position in that set.

R uses its own initialization method due to B. D. Ripley and is not affected by the initialization issue in the 1998 code of Matsumoto and Nishimura addressed in a 2002 update.

It exhibits 2 clear failures in each of the TestU01 Crush and the BigCrush suite (L'Ecuyer, 2007).

`"Knuth-TAOCP-2002"`: A 32-bit integer GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is

$$X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$$

and the 'seed' is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around 2^{129}.

`"Knuth-TAOCP"`: An earlier version from Knuth (1997).

The 2002 version was not backwards compatible with the earlier version: the initialization of the GFSR from the seed was altered. R did not allow you to choose consecutive seeds, the reported 'weakness', and already scrambled the seeds. Otherwise, the algorithm is identical to Knuth-TAOCP-2002, with the same lagged Fibonacci recurrence formula.

Initialization of this generator is done in interpreted R code and so takes a short but noticeable time.

It exhibits 3 clear failure in the TestU01 Crush suite and 4 clear failures in the BigCrush suite (*L'Ecuyer*, 2007).

`"L'Ecuyer-CMRG"`: A 'combined multiple-recursive generator' from L'Ecuyer (1999), each element of which is a feedback multiplicative generator with three integer elements: thus the seed is a (signed) integer vector of length 6. The period is around 2^{191}.

The 6 elements of the seed are internally regarded as 32-bit unsigned integers. Neither the first three nor the last three should be all zero, and they are limited to less than `4294967087` and `4294944443` respectively.

This is not particularly interesting of itself, but provides the basis for the multiple streams used in package **parallel**.

It exhibits 6 clear failures in each of the TestU01 Crush and the BigCrush suite (*L'Ecuyer*, 2007).

`"user-supplied"`: Use a user-supplied generator. See `Random.user` for details.

`normal.kind` can be `"Kinderman-Ramage"`, `"Buggy Kinderman-Ramage"` (not for `set.seed`), `"Ahrens-Dieter"`, `"Box-Muller"`, `"Inversion"` (the default), or `"user-supplied"`. (For inversion, see the reference in `qnorm`.) The Kinderman-Ramage generator used in versions prior to 1.7.0 (now called `"Buggy"`) had several approximation errors and should only be used for reproduction of old results. The `"Box-Muller"` generator is stateful as pairs of normals are generated and returned sequentially. The state is reset whenever it is selected (even if it is the current normal generator) and when `kind` is changed.

`sample.kind` can be `"Rounding"` or `"Rejection"`, or partial matches to these. The former was the default in versions prior to 3.6.0: it made `sample` noticeably non-uniform on large populations, and should only be used for reproduction of old results. See PR#17494 for a discussion.

`set.seed` uses a single integer argument to set as many seeds as are required. It is intended as a simple way to get quite different seeds by specifying small integer arguments, and also as a way to get valid seed sets for the more complicated methods (especially `"Mersenne-Twister"` and `"Knuth-TAOCP"`). There is no guarantee that different values of `seed` will seed the RNG differently, although any exceptions would be extremely rare. If called with `seed = NULL` it re-initializes (see 'Note') as if no seed had yet been set.

The use of `kind = NULL`, `normal.kind = NULL` or `sample.kind = NULL` in `RNGkind` or `set.seed` selects the currently-used generator (including that used in the previous session if the workspace has been restored): if no generator has been used it selects `"default"`.

Value

`.Random.seed` is an `integer` vector whose first element *codes* the kind of RNG and normal generator. The lowest two decimal digits are in `0:(k-1)` where `k` is the number of available RNGs. The hundreds represent the type of normal generator (starting at `0`), and the ten thousands represent the type of discrete uniform sampler.

In the underlying C, `.Random.seed[-1]` is `unsigned`; therefore in R `.Random.seed[-1]` can be negative, due to the representation of an unsigned integer by a signed integer.

`RNGkind` returns a three-element character vector of the RNG, normal and sample kinds selected *before* the call, invisibly if either argument is not `NULL`. A type starts a session as the default, and is selected either by a call to `RNGkind` or by setting `.Random.seed` in the workspace. (NB: prior to R 3.6.0 the first two kinds were returned in a two-element character vector.)

`RNGversion` returns the same information as `RNGkind` about the defaults in a specific R version.

`set.seed` returns `NULL`, invisibly.

Note

Initially, there is no seed; a new one is created from the current time and the process ID when one is required. Hence different sessions will give different simulation results, by default. However, the seed might be restored from a previous session if a previously saved workspace is restored.

`.Random.seed` saves the seed set for the uniform random-number generator, at least for the system generators. It does not necessarily save the state of other generators, and in particular does not save the state of the Box–Muller normal generator. If you want to reproduce work later, call `set.seed` (preferably with explicit values for `kind` and `normal.kind`) rather than set `.Random.seed`.

The object `.Random.seed` is only looked for in the user's workspace.

Do not rely on randomness of low-order bits from RNGs. Most of the supplied uniform generators return 32-bit integer values that are converted to doubles, so they take at most 2^{32} distinct values and long runs will return duplicated values (Wichmann-Hill is the exception, and all give at least 30 varying bits.)

Author(s)

of RNGkind: Martin Maechler. Current implementation, B. D. Ripley with modifications by Duncan Murdoch.

References

Ahrens, J. H. and Dieter, U. (1973). Extensions of Forsythe's method for random sampling from the normal distribution. *Mathematics of Computation*, **27**, 927–937.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole. (`set.seed`, storing in `.Random.seed`.)

Box, G. E. P. and Muller, M. E. (1958). A note on the generation of normal random deviates. *Annals of Mathematical Statistics*, **29**, 610–611. doi:10.1214/aoms/1177706645.

De Matteis, A. and Pagnutti, S. (1993). Long-range Correlation Analysis of the Wichmann-Hill Random Number Generator. *Statistics and Computing*, **3**, 67–70. doi:10.1007/BF00153065.

Kinderman, A. J. and Ramage, J. G. (1976). Computer generation of normal random variables. *Journal of the American Statistical Association*, **71**, 893–896. doi:10.2307/2286857.

Knuth, D. E. (1997). *The Art of Computer Programming*. Volume 2, third edition.
Source code at `https://www-cs-faculty.stanford.edu/~knuth/taocp.html`.

Knuth, D. E. (2002). *The Art of Computer Programming*. Volume 2, third edition, ninth printing.

L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, **47**, 159–164. doi:10.1287/opre.47.1.159.

L'Ecuyer, P. and Simard, R. (2007). TestU01: A C Library for Empirical Testing of Random Number Generators *ACM Transactions on Mathematical Software*, **33**, Article 22. doi:10.1145/1268776.1268777.
The TestU01 C library is available from `http://simul.iro.umontreal.ca/testu01/tu01.html` or also `https://github.com/umontreal-simul/TestU01-2009`.

Marsaglia, G. (1997). *A random number generator for C*. Discussion paper, posting on Usenet newsgroup `sci.stat.math` on September 29, 1997.

Marsaglia, G. and Zaman, A. (1994). Some portable very-long-period random number generators. *Computers in Physics*, **8**, 117–121. doi:10.1063/1.168514.

Matsumoto, M. and Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.
Source code formerly at `http://www.math.keio.ac.jp/~matumoto/emt.html`.
Now see `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/c-lang.html`.

Reeds, J., Hubert, S. and Abrahams, M. (1982–4). C implementation of SuperDuper, University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka.)

Wichmann, B. A. and Hill, I. D. (1982). Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator. *Applied Statistics*, **31**, 188–190; Remarks: **34**, 198 and **35**, 89. doi:10.2307/2347988.

See Also

`sample` for random sampling with and without replacement.

Distributions for functions for random-variate generation from standard distributions.

Examples

```
require(stats)

## Seed the current RNG, i.e., set the RNG status
set.seed(42); u1 <- runif(30)
set.seed(42); u2 <- runif(30) # the same because of identical RNG status:
stopifnot(identical(u1, u2))

## the default random seed is 626 integers, so only print a few
 runif(1); .Random.seed[1:6]; runif(1); .Random.seed[1:6]
 ## If there is no seed, a "random" new one is created:
 rm(.Random.seed); runif(1); .Random.seed[1:6]

ok <- RNGkind()
RNGkind("Wich")  # (partial string matching on 'kind')

## This shows how 'runif(.)' works for Wichmann-Hill,
## using only R functions:

p.WH <- c(30269, 30307, 30323)
a.WH <- c(  171,   172,   170)
next.WHseed <- function(i.seed = .Random.seed[-1])
  { (a.WH * i.seed) %% p.WH }
my.runif1 <- function(i.seed = .Random.seed)
  { ns <- next.WHseed(i.seed[-1]); sum(ns / p.WH) %% 1 }
set.seed(1998-12-04)# (when the next lines were added to the souRce)
rs <- .Random.seed
(WHs <- next.WHseed(rs[-1]))
u <- runif(1)
stopifnot(
 next.WHseed(rs[-1]) == .Random.seed[-1],
 all.equal(u, my.runif1(rs))
)

## ----
.Random.seed
RNGkind("Super") # matches  "Super-Duper"
RNGkind()
.Random.seed # new, corresponding to  Super-Duper

## Reset:
RNGkind(ok[1])

RNGversion(getRversion()) # the default version for this R version

## ----
sum(duplicated(runif(1e6))) # around 110 for default generator
## and we would expect about almost sure duplicates beyond about
qbirthday(1 - 1e-6, classes = 2e9) # 235,000
```

Random.user *User-supplied Random Number Generation*

Description

Function RNGkind allows user-coded uniform and normal random number generators to be supplied. The details are given here.

Details

A user-specified uniform RNG is called from entry points in dynamically-loaded compiled code. The user must supply the entry point user_unif_rand, which takes no arguments and returns a *pointer to* a double. The example below will show the general pattern. The generator should have at least 25 bits of precision.

Optionally, the user can supply the entry point user_unif_init, which is called with an unsigned int argument when RNGkind (or set.seed) is called, and is intended to be used to initialize the user's RNG code. The argument is intended to be used to set the 'seeds'; it is the seed argument to set.seed or an essentially random seed if RNGkind is called.

If only these functions are supplied, no information about the generator's state is recorded in .Random.seed. Optionally, functions user_unif_nseed and user_unif_seedloc can be supplied which are called with no arguments and should return pointers to the number of seeds and to an integer (specifically, 'Int32') array of seeds. Calls to GetRNGstate and PutRNGstate will then copy this array to and from .Random.seed.

A user-specified normal RNG is specified by a single entry point user_norm_rand, which takes no arguments and returns a *pointer to* a double.

Warning

As with all compiled code, mis-specifying these functions can crash R. Do include the 'R_ext/Random.h' header file for type checking.

Examples

```
## Not run:
##  Marsaglia's congruential PRNG
#include <R_ext/Random.h>

static Int32 seed;
static double res;
static int nseed = 1;

double * user_unif_rand(void)
{
    seed = 69069 * seed + 1;
    res = seed * 2.32830643653869e-10;
    return &res;
}
```

```
void  user_unif_init(Int32 seed_in) { seed = seed_in; }
int * user_unif_nseed(void) { return &nseed; }
int * user_unif_seedloc(void) { return (int *) &seed; }

/*  ratio-of-uniforms for normal  */
#include <math.h>
static double x;

double * user_norm_rand(void)
{
    double u, v, z;
    do {
        u = unif_rand();
        v = 0.857764 * (2. * unif_rand() - 1);
        x = v/u; z = 0.25 * x * x;
        if (z < 1. - u) break;
        if (z > 0.259/u + 0.35) continue;
    } while (z > -log(u));
    return &x;
}

## Use under Unix:
R CMD SHLIB urand.c
R
> dyn.load("urand.so")
> RNGkind("user")
> runif(10)
> .Random.seed
> RNGkind(, "user")
> rnorm(10)
> RNGkind()
[1] "user-supplied" "user-supplied"

## End(Not run)
```

---

range *Range of Values*

Description

range returns a vector containing the minimum and maximum of all the given arguments.

Usage

```
range(..., na.rm = FALSE)
## Default S3 method:
range(..., na.rm = FALSE, finite = FALSE)
## same for classes 'Date' and 'POSIXct'

.rangeNum(..., na.rm, finite, isNumeric)
```

Arguments

...	any numeric or character objects.
na.rm	logical, indicating if NA's should be omitted.
finite	logical, indicating if all non-finite elements should be omitted.
isNumeric	a function returning TRUE or FALSE when called on c(..., recursive = TRUE), is.numeric() for the default range() method.

Details

range is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

If na.rm is FALSE, NA and NaN values in any of the arguments will cause NA values to be returned, otherwise NA values are ignored.

If finite is TRUE, the minimum and maximum of all finite values is computed, i.e., finite = TRUE *includes* na.rm = TRUE.

A special situation occurs when there is no (after omission of NAs) nonempty argument left, see min.

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

min, max.

The extendrange() utility in package **grDevices**.

Examples

```
(r.x <- range(stats::rnorm(100)))
diff(r.x) # the SAMPLE range

x <- c(NA, 1:3, -1:1/0); x
range(x)
range(x, na.rm = TRUE)
range(x, finite = TRUE)
```

rank *Sample Ranks*

Description

Returns the sample ranks of the values in a vector. Ties (i.e., equal values) and missing values can be handled in several ways.

Usage

```
rank(x, na.last = TRUE,
     ties.method = c("average", "first", "last", "random", "max", "min"))
```

Arguments

x	a numeric, complex, character or logical vector.
na.last	a logical or character string controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed; if "keep" they are kept with rank NA.
ties.method	a character string specifying how ties are treated, see 'Details'; can be abbreviated.

Details

If all components are different (and no NAs), the ranks are well defined, with values in seq_along(x). With some values equal (called 'ties'), the argument ties.method determines the result at the corresponding indices. The "first" method results in a permutation with increasing values at each index set of ties, and analogously "last" with decreasing values. The "random" method puts these in random order whereas the default, "average", replaces them by their mean, and "max" and "min" replaces them by their maximum and minimum respectively, the latter being the typical sports ranking.

NA values are never considered to be equal: for na.last = TRUE and na.last = FALSE they are given distinct ranks in the order in which they occur in x.

NB: rank is not itself generic but xtfrm is, and rank(xtfrm(x), ...) will have the desired result if there is a xtfrm method. Otherwise, rank will make use of ==, >, is.na and extraction methods for classed objects, possibly rather slowly.

Value

A numeric vector of the same length as x with names copied from x (unless na.last = NA, when missing values are removed). The vector is of integer type unless x is a long vector or ties.method = "average" when it is of double type (whether or not there are any ties).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

order and sort; xtfrm, see above.

Examples

```
(r1 <- rank(x1 <- c(3, 1, 4, 15, 92)))
x2 <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
names(x2) <- letters[1:11]
(r2 <- rank(x2)) # ties are averaged

## rank() is "idempotent": rank(rank(x)) == rank(x) :
stopifnot(rank(r1) == r1, rank(r2) == r2)

## ranks without averaging
rank(x2, ties.method= "first")  # first occurrence wins
rank(x2, ties.method= "last")   #  last occurrence wins
rank(x2, ties.method= "random") # ties broken at random
rank(x2, ties.method= "random") # and again

## keep ties ties, no average
(rma <- rank(x2, ties.method= "max"))  # as used classically
(rmi <- rank(x2, ties.method= "min"))  # as in Sports
stopifnot(rma + rmi == round(r2 + r2))

## Comparing all tie.methods:
tMeth <- eval(formals(rank)$ties.method)
rx2 <- sapply(tMeth, function(M) rank(x2, ties.method=M))
cbind(x2, rx2)
## ties.method's does not matter w/o ties:
x <- sample(47)
rx <- sapply(tMeth, function(MM) rank(x, ties.method=MM))
stopifnot(all(rx[,1] == rx))
```

---

rapply *Recursively Apply a Function to a List*

Description

rapply is a recursive version of lapply with flexibility in *how* the result is structured (how = "..").

Usage

```
rapply(object, f, classes = "ANY", deflt = NULL,
       how = c("unlist", "replace", "list"), ...)
```

Arguments

object	a list or expression, i.e., "list-like".
f	a function of one "principal" argument, passing further arguments via ....
classes	character vector of class names, or "ANY" to match any class.
deflt	the default result (not used if how = "replace").
how	character string partially matching the three possibilities given: see 'Details'.
...	additional arguments passed to the call to f.

Details

This function has two basic modes. If how = "replace", each element of object which is not itself list-like and has a class included in classes is replaced by the result of applying f to the element.

Otherwise, with mode how = "list" or how = "unlist", conceptually object is copied, all non-list elements which have a class included in classes are replaced by the result of applying f to the element and all others are replaced by deflt. Finally, if how = "unlist", unlist(recursive = TRUE) is called on the result.

The semantics differ in detail from lapply: in particular the arguments are evaluated before calling the C code.

In R 3.5.x and earlier, object was required to be a list, which was *not* the case for its list-like components.

Value

If how = "unlist", a vector, otherwise "list-like" of similar structure as object.

References

Chambers, J. A. (1998) *Programming with Data*. Springer.
(rapply is only described briefly there.)

See Also

lapply, dendrapply.

Examples

```
X <- list(list(a = pi, b = list(c = 1L)), d = "a test")
# the "identity operation":
rapply(X, function(x) x, how = "replace") -> X.; stopifnot(identical(X, X.))
rapply(X, sqrt, classes = "numeric", how = "replace")
rapply(X, deparse, control = "all") # passing extras. argument of deparse()
rapply(X, nchar, classes = "character", deflt = NA_integer_, how = "list")
rapply(X, nchar, classes = "character", deflt = NA_integer_, how = "unlist")
rapply(X, nchar, classes = "character",                      how = "unlist")
rapply(X, log, classes = "numeric", how = "replace", base = 2)

## with expression() / list():
```

```
E  <- expression(list(a = pi, b = expression(c = C1 * C2)), d = "a test")
LE <- list(expression(a = pi, b = expression(c = C1 * C2)), d = "a test")
rapply(E, nchar, how="replace") # "expression(c = C1 * C2)" are 23 chars
rapply(E, nchar, classes = "character", deflt = NA_integer_, how = "unlist")
rapply(LE, as.character) # a "pi" | b1 "expression" | b2 "C1 * C2" ..
rapply(LE, nchar)        # (see above)
stopifnot(exprs = {
  identical(E , rapply(E , identity, how = "replace"))
  identical(LE, rapply(LE, identity, how = "replace"))
})
```

---

raw *Raw Vectors*

Description

Creates or tests for objects of type "raw".

Usage

```
raw(length = 0)
as.raw(x)
is.raw(x)
```

Arguments

length	desired length.
x	object to be coerced.

Details

The raw type is intended to hold raw bytes. It is possible to extract subsequences of bytes, and to replace elements (but only by elements of a raw vector). The relational operators (see Comparison, using the numerical order of the byte representation) work, as do the logical operators (see Logic) with a bitwise interpretation.

A raw vector is printed with each byte separately represented as a pair of hex digits. If you want to see a character representation (with escape sequences for non-printing characters) use rawToChar.

Coercion to raw treats the input values as representing small (decimal) integers, so the input is first coerced to integer, and then values which are outside the range [0 . . . 255] or are NA are set to 0 (the nul byte).

as.raw and is.raw are primitive functions.

Value

raw creates a raw vector of the specified length. Each element of the vector is equal to 0. Raw vectors are used to store fixed-length sequences of bytes.

as.raw attempts to coerce its argument to be of raw type. The (elementwise) answer will be 0 unless the coercion succeeds (or if the original value successfully coerces to 0).

is.raw returns true if and only if typeof(x) == "raw".

See Also

charToRaw, rawShift, etc.

& for bitwise operations on raw vectors.

Examples

```
xx <- raw(2)
xx[1] <- as.raw(40)     # NB, not just 40.
xx[2] <- charToRaw("A")
xx       ## 28 41   -- raw prints hexadecimals
dput(xx) ## as.raw(c(0x28, 0x41))
as.integer(xx) ## 40 65

x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE
rawToChar(y)
is.raw(x)
is.raw(y)
stopifnot( charToRaw("\xa3") == as.raw(0xa3) )

isASCII <-  function(txt) all(charToRaw(txt) <= as.raw(127))
isASCII(x)  # true
isASCII("\xa325.63") # false (in Latin-1, this is an amount in UK pounds)
```

---

rawConnection *Raw Connections*

Description

Input and output raw connections.

Usage

```
rawConnection(object, open = "r")

rawConnectionValue(con)
```

Arguments

object	character or raw vector. A description of the connection. For an input this is an R raw vector object, and for an output connection the name for the connection.
open	character. Any of the standard connection open modes.
con	an output raw connection.

Details

An input raw connection is opened and the raw vector is copied at the time the connection object is created, and `close` destroys the copy.

An output raw connection is opened and creates an R raw vector internally. The raw vector can be retrieved *via* `rawConnectionValue`.

If a connection is open for both input and output the initial raw vector supplied is copied when the connections is open

Value

For `rawConnection`, a connection object of class `"rawConnection"` which inherits from class `"connection"`.

For `rawConnectionValue`, a raw vector.

Note

As output raw connections keep the internal raw vector up to date call-by-call, they are relatively expensive to use (although over-allocation is used), and it may be better to use an anonymous `file()` connection to collect output.

On (rare) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

See Also

`connections`, `showConnections`.

Examples

```
zz <- rawConnection(raw(0), "r+") # start with empty raw vector
writeBin(LETTERS, zz)
seek(zz, 0)
readLines(zz) # raw vector has embedded nuls
seek(zz, 0)
writeBin(letters[1:3], zz)
rawConnectionValue(zz)
close(zz)
```

---

rawConversion *Convert to or from (Bit/Packed) Raw Vectors*

Description

Conversion to and from and manipulation of objects of type `"raw"`, both used as bits or "packed" 8 bits.

Usage

```
charToRaw(x)
rawToChar(x, multiple = FALSE)

rawShift(x, n)

rawToBits(x)
intToBits(x)
packBits(x, type = c("raw", "integer", "double"))

numToInts(x)
numToBits(x)
```

Arguments

x	object to be converted or shifted.
multiple	logical: should the conversion be to a single character string or multiple individual characters?
n	the number of bits to shift. Positive numbers shift right and negative numbers shift left: allowed values are -8 ... 8.
type	the result type, partially matched.

Details

packBits accepts raw, integer or logical inputs, the last two without any NAs.

numToBits(.) and packBits(., type="double") are *inverse* functions of each other, see also the examples.

Note that 'bytes' are not necessarily the same as characters, e.g. in UTF-8 locales.

Value

charToRaw converts a length-one character string to raw bytes. It does so without taking into account any declared encoding (see Encoding).

rawToChar converts raw bytes either to a single character string or a character vector of single bytes (with "" for 0). (Note that a single character string could contain embedded NULs; only trailing nulls are allowed and will be removed.) In either case it is possible to create a result which is invalid in a multibyte locale, e.g. one using UTF-8. Long vectors are allowed if multiple is true.

rawShift(x, n) shift the bits in x by n positions to the right, see the argument n, above.

rawToBits returns a raw vector of 8 times the length of a raw vector with entries 0 or 1. intToBits returns a raw vector of 32 times the length of an integer vector with entries 0 or 1. (Non-integral numeric values are truncated to integers.) In both cases the unpacking is least-significant bit first.

packBits packs its input (using only the lowest bit for raw or integer vectors) least-significant bit first to a raw, integer or double ("numeric") vector.

numToInts() and numToBits() split double precision numeric vectors either into to two integers each or into 64 bits each, stored as raw. In both cases the unpacking is least-significant element first.

Examples

```
x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE

rawToChar(y)
rawToChar(y, multiple = TRUE)
(xx <- c(y,  charToRaw("&"), charToRaw(" more")))
rawToChar(xx)

rawShift(y, 1)
rawShift(y,-2)

rawToBits(y)

showBits <- function(r) stats::symnum(as.logical(rawToBits(r)))

z <- as.raw(5)
z ; showBits(z)
showBits(rawShift(z, 1)) # shift to right
showBits(rawShift(z, 2))
showBits(z)
showBits(rawShift(z, -1)) # shift to left
showBits(rawShift(z, -2)) # ..
showBits(rawShift(z, -3)) # shifted off entirely

packBits(as.raw(0:31))
i <- -2:3
stopifnot(exprs = {
  identical(i, packBits(intToBits(i), "integer"))
  identical(packBits(       0:31) ,
            packBits(as.raw(0:31)))
})
str(pBi <- packBits(intToBits(i)))
data.frame(B = matrix(pBi, nrow=6, byrow=TRUE),
           hex = format(as.hexmode(i)), i)


## Look at internal bit representation of ...

## ... of integers :
bitI <- function(x) vapply(as.integer(x), function(x) {
            b <- substr(as.character(rev(intToBits(x))), 2L, 2L)
            paste0(c(b[1L], " ", b[2:32]), collapse = "")
          }, "")
print(bitI(-8:8), width = 35, quote = FALSE)

## ... of double precision numbers in format  'sign exp | mantissa'
## where  1 bit sign  1 <==> "-";
##       11 bit exp   is the base-2 exponent biased by 2^10 - 1 (1023)
##       52 bit mantissa is without the implicit leading '1'
#
```

```
## Bit representation  [ sign | exponent | mantissa ] of double prec numbers :

bitC <- function(x) noquote(vapply(as.double(x), function(x) { # split one double
    b <- substr(as.character(rev(numToBits(x))), 2L, 2L)
    paste0(c(b[1L], " ", b[2:12], " | ", b[13:64]), collapse = "")
  }, ""))
bitC(17)
bitC(c(-1,0,1))
bitC(2^(-2:5))
bitC(1+2^-(1:53))# from 0.5 converge to 1

###  numToBits(.)  <==>   intToBits(numToInts(.)) :
d2bI <- function(x) vapply(as.double(x), function(x) intToBits(numToInts(x)), raw(64L))
d2b  <- function(x) vapply(as.double(x), function(x)           numToBits(x) , raw(64L))
set.seed(1)
x <- c(sort(rt(2048, df=1.5)),  2^(-10:10), 1+2^-(1:53))
str(bx <- d2b(x)) # a  64 x 2122  raw matrix
stopifnot( identical(bx, d2bI(x)) )

## Show that  packBits(*, "double")  is the inverse of numToBits() :
packBits(numToBits(pi), type="double")
bitC(2050)
b <- numToBits(2050)
identical(b, numToBits(packBits(b, type="double")))
pbx <- apply(bx, 2, packBits, type="double")
stopifnot( identical(pbx, x))
```

RdUtils *Utilities for Processing Rd Files*

Description

Utilities for converting files in R documentation (Rd) format to other formats or create indices from them, and for converting documentation in other formats to Rd format.

Usage

```
R CMD Rdconv [options] file
R CMD Rd2pdf [options] files
```

Arguments

file	the path to a file to be processed.
files	a list of file names specifying the R documentation sources to use, by either giving the paths to the files, or the path to a directory with the sources of a package.
options	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

R CMD Rdconv converts Rd format to plain text, HTML or LaTeX formats: it can also extract the examples.

R CMD Rd2pdf is the user-level program for producing PDF output from Rd sources. It will make use of the environment variables R_PAPERSIZE (set by R CMD, with a default set when R was installed: values for R_PAPERSIZE are a4, letter, legal and executive) and R_PDFVIEWER (the PDF previewer). Also, RD2PDF_INPUTENC can be set to inputenx to make use of the LaTeX package of that name rather than inputenc: this might be needed for better support of the UTF-8 encoding.

R CMD Rd2pdf calls tools::texi2pdf to produce its PDF file: see its help for the possibilities for the texi2dvi command which that function uses (and which can be overridden by setting environment variable R_TEXI2DVICMD).

Use R CMD *foo* --help to obtain usage information on utility *foo*.

See Also

The section 'Processing documentation files' in the 'Writing R Extensions' manual: RShowDoc("R-exts").

---

readBin *Transfer Binary Data To and From Connections*

Description

Read binary data from or write binary data to a connection or raw vector.

Usage

```
readBin(con, what, n = 1L, size = NA_integer_, signed = TRUE,
        endian = .Platform$endian)

writeBin(object, con, size = NA_integer_,
         endian = .Platform$endian, useBytes = FALSE)
```

Arguments

con	A connection object or a character string naming a file or a raw vector.
what	Either an object whose mode will give the mode of the vector to be read, or a character vector of length one describing the mode: one of "numeric", "double", "integer", "int", "logical", "complex", "character", "raw".
n	numeric. The (maximal) number of records to be read. You can use an overestimate here, but not too large as storage is reserved for n items.
size	integer. The number of bytes per element in the byte stream. The default, NA_integer_, uses the natural size. Size changing is not supported for raw and complex vectors.

signed	logical. Only used for integers of sizes 1 and 2, when it determines if the quantity on file should be regarded as a signed or unsigned integer.
endian	The endianness ("big" or "little") of the target system for the file. Using "swap" will force swapping endianness.
object	An R object to be written to the connection.
useBytes	See writeLines.

Details

These functions can only be used with binary-mode connections. If con is a character string, the functions call file to obtain a binary-mode file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call in an appropriate mode (binary read or write) and then closed again. An open connection must be in binary mode.

If readBin is called with con a raw vector, the data in the vector is used as input. If writeBin is called with con a raw vector, it is just an indication that a raw vector should be returned.

If size is specified and not the natural size of the object, each element of the vector is coerced to an appropriate type before being written or as it is read. Possible sizes are 1, 2, 4 and possibly 8 for integer or logical vectors, and 4, 8 and possibly 12/16 for numeric vectors. (Note that coercion occurs as signed types except if signed = FALSE when reading integers of sizes 1 and 2.) Changing sizes is unlikely to preserve NAs, and the extended precision sizes are unlikely to be portable across platforms.

readBin and writeBin read and write C-style zero-terminated character strings. Input strings are limited to 10000 characters. readChar and writeChar can be used to read and write fixed-length strings. No check is made that the string is valid in the current locale's encoding.

Handling R's missing and special (Inf, -Inf and NaN) values is discussed in the 'R Data Import/Export' manual.

Only $2^{31} - 1$ bytes can be written in a single call (and that is the maximum capacity of a raw vector on 32-bit platforms).

'Endian-ness' is relevant for size > 1, and should always be set for portable code (the default is only appropriate when writing and then reading files on the same platform).

Value

For readBin, a vector of appropriate mode and length the number of items read (which might be less than n).

For writeBin, a raw vector (if con is a raw vector) or invisibly NULL.

Note

Integer read/writes of size 8 will be available if either C type long is of size 8 bytes or C type long long exists and is of size 8 bytes.

Real read/writes of size sizeof(long double) (usually 12 or 16 bytes) will be available only if that type is available and different from double.

If `readBin(what = character())` is used incorrectly on a file which does not contain C-style character strings, warnings (usually many) are given. From a file or connection, the input will be broken into pieces of length 10000 with any final part being discarded.

See Also

The 'R Data Import/Export' manual.

`readChar` to read/write fixed-length strings.

`connections`, `readLines`, `writeLines`.

`.Machine` for the sizes of `long`, `long long` and `long double`.

Examples

```
zzfil <- tempfile("testbin")
zz <- file(zzfil, "wb")
writeBin(1:10, zz)
writeBin(pi, zz, endian = "swap")
writeBin(pi, zz, size = 4)
writeBin(pi^2, zz, size = 4, endian = "swap")
writeBin(pi+3i, zz)
writeBin("A test of a connection", zz)
z <- paste("A very long string", 1:100, collapse = " + ")
writeBin(z, zz)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    writeBin(as.integer(5^(1:10)), zz, size = 8)
if((s <- .Machine$sizeof.longdouble) > 8)
    writeBin((pi/3)^(1:10), zz, size = s)
close(zz)

zz <- file(zzfil, "rb")
readBin(zz, integer(), 4)
readBin(zz, integer(), 6)
readBin(zz, numeric(), 1, endian = "swap")
readBin(zz, numeric(), size = 4)
readBin(zz, numeric(), size = 4, endian = "swap")
readBin(zz, complex(), 1)
readBin(zz, character(), 1)
z2 <- readBin(zz, character(), 1)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    readBin(zz, integer(), 10,  size = 8)
if((s <- .Machine$sizeof.longdouble) > 8)
    readBin(zz, numeric(), 10, size = s)
close(zz)
unlink(zzfil)
stopifnot(z2 == z)

## signed vs unsigned ints
zzfil <- tempfile("testbin")
zz <- file(zzfil, "wb")
x <- as.integer(seq(0, 255, 32))
writeBin(x, zz, size = 1)
```

```
writeBin(x, zz, size = 1)
x <- as.integer(seq(0, 60000, 10000))
writeBin(x, zz, size = 2)
writeBin(x, zz, size = 2)
close(zz)
zz <- file(zzfil, "rb")
readBin(zz, integer(), 8, size = 1)
readBin(zz, integer(), 8, size = 1, signed = FALSE)
readBin(zz, integer(), 7, size = 2)
readBin(zz, integer(), 7, size = 2, signed = FALSE)
close(zz)
unlink(zzfil)

## use of raw
z <- writeBin(pi^{1:5}, raw(), size = 4)
readBin(z, numeric(), 5, size = 4)
z <- writeBin(c("a", "test", "of", "character"), raw())
readBin(z, character(), 4)
```

---

readChar *Transfer Character Strings To and From Connections*

Description

Transfer character strings to and from connections, without assuming they are null-terminated on the connection.

Usage

```
readChar(con, nchars, useBytes = FALSE)

writeChar(object, con, nchars = nchar(object, type = "chars"),
          eos = "", useBytes = FALSE)
```

Arguments

con	a connection object, or a character string naming a file, or a raw vector.
nchars	integer vector, giving the lengths in characters of (unterminated) character strings to be read or written. Elements must be >= 0 and not NA.
useBytes	logical: For readChar, should nchars be regarded as a number of bytes not characters in a multi-byte locale? For writeChar, see writeLines.
object	a character vector to be written to the connection, at least as long as nchars.
eos	'end of string': character string. The terminator to be written after each string, followed by an ASCII nul; use NULL for no terminator at all.

Details

These functions complement readBin and writeBin which read and write C-style zero-terminated character strings. They are for strings of known length, and can optionally write an end-of-string mark. They are intended only for character strings valid in the current locale.

These functions are intended to be used with binary-mode connections. If con is a character string, the functions call file to obtain a binary-mode file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call in an appropriate mode (binary read or write) and then closed again. An open connection must be in binary mode.

If readChar is called with con a raw vector, the data in the vector is used as input. If writeChar is called with con a raw vector, it is just an indication that a raw vector should be returned.

Character strings containing ASCII nul(s) will be read correctly by readChar but truncated at the first nul with a warning.

If the character length requested for readChar is longer than the data available on the connection, what is available is returned. For writeChar if too many characters are requested the output is zero-padded, with a warning.

Missing strings are written as NA.

Value

For readChar, a character vector of length the number of items read (which might be less than length(nchars)).

For writeChar, a raw vector (if con is a raw vector) or invisibly NULL.

Note

Earlier versions of R allowed embedded NUL bytes within character strings, but not R >= 2.8.0. readChar was commonly used to read fixed-size zero-padded byte fields for which readBin was unsuitable. readChar can still be used for such fields if there are no embedded NULs: otherwise readBin(what = "raw") provides an alternative.

nchars will be interpreted in bytes not characters in a non-UTF-8 multi-byte locale, with a warning.

There is little validity checking of UTF-8 reads.

Using these functions on a text-mode connection may work but should not be mixed with text-mode access to the connection, especially if the connection was opened with an encoding argument.

See Also

The 'R Data Import/Export' manual.

connections, readLines, writeLines, readBin

Examples

```
## test fixed-length strings
zzfil <- tempfile("testchar")
zz <- file(zzfil, "wb")
x <- c("a", "this will be truncated", "abc")
nc <- c(3, 10, 3)
writeChar(x, zz, nc, eos = NULL)
writeChar(x, zz, eos = "\r\n")
close(zz)

zz <- file(zzfil, "rb")
readChar(zz, nc)
readChar(zz, nchar(x)+3) # need to read the terminator explicitly
close(zz)
unlink(zzfil)
```

---

readline *Read a Line from the Terminal*

Description

`readline` reads a line from the terminal (in interactive use).

Usage

```
readline(prompt = "")
```

Arguments

`prompt`	the string printed when prompting the user for input. Should usually end with a space `" "`.

Details

The prompt string will be truncated to a maximum allowed length, normally 256 chars (but can be changed in the source code).

This can only be used in an interactive session.

Value

A character vector of length one. Both leading and trailing spaces and tabs are stripped from the result.

In non-interactive use the result is as if the response was RETURN and the value is `""`.

See Also

`readLines` for reading text lines from connections, including files.

Examples

```
fun <- function() {
  ANSWER <- readline("Are you a satisfied R user? ")
  ## a better version would check the answer less cursorily, and
  ## perhaps re-prompt
  if (substr(ANSWER, 1, 1) == "n")
    cat("This is impossible.  YOU LIED!\n")
  else
    cat("I knew it.\n")
}
if(interactive()) fun()
```

---

readLines *Read Text Lines from a Connection*

Description

Read some or all text lines from a connection.

Usage

```
readLines(con = stdin(), n = -1L, ok = TRUE, warn = TRUE,
          encoding = "unknown", skipNul = FALSE)
```

Arguments

con	a connection object or a character string.
n	integer. The (maximal) number of lines to read. Negative values indicate that one should read up to the end of input on the connection.
ok	logical. Is it OK to reach the end of the connection before n > 0 lines are read? If not, an error will be generated.
warn	logical. Warn if a text file is missing a final EOL or if there are embedded NULs in the file.
encoding	encoding to be assumed for input strings. It is used to mark character strings as known to be in Latin-1, UTF-8 or to be bytes: it is not used to re-encode the input. To do the latter, specify the encoding as part of the connection con or via options(encoding=): see the examples and 'Details'.
skipNul	logical: should NULs be skipped?

Details

If the con is a character string, the function calls file to obtain a file connection which is opened for the duration of the function call. This can be a compressed file. (tilde expansion of the file path is done by file.)

If the connection is open it is read from its current position. If it is not open, it is opened in "rt" mode for the duration of the call and then closed (but not destroyed; one must call close to do that).

If the final line is incomplete (no final EOL marker) the behaviour depends on whether the connection is blocking or not. For a non-blocking text-mode connection the incomplete line is pushed back, silently. For all other connections the line will be accepted, with a warning.

Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line.

Embedded NULs in the input stream will terminate the line currently being read, with a warning (unless `skipNul = TRUE` or `warn = FALSE`).

If `con` is a not-already-open connection with a non-default `encoding` argument, the text is converted to UTF-8 and declared as such (and the `encoding` argument to `readLines` is ignored). See the examples.

Value

A character vector of length the number of lines read.

The elements of the result have a declared encoding if `encoding` is `"latin1"` or `"UTF-8"`,

Note

The default connection, `stdin`, may be different from `con = "stdin"`: see `file`.

See Also

`connections`, `writeLines`, `readBin`, `scan`

Examples

```
fil <- tempfile(fileext = ".data")
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = fil,
    sep = "\n")
readLines(fil, n = -1)
unlink(fil) # tidy up

## difference in blocking
fil <- tempfile("test")
cat("123\nabc", file = fil)
readLines(fil) # line with a warning

con <- file(fil, "r", blocking = FALSE)
readLines(con) # "123"
cat(" def\n", file = fil, append = TRUE)
readLines(con) # gets both
close(con)

unlink(fil) # tidy up

## Not run:
# read a 'Windows Unicode' file
A <- readLines(con <- file("Unicode.txt", encoding = "UCS-2LE"))
close(con)
unique(Encoding(A)) # will most likely be UTF-8
```

```
## End(Not run)
```

---

readRDS *Serialization Interface for Single Objects*

Description

Functions to write a single R object to a file, and to restore it.

Usage

```
saveRDS(object, file = "", ascii = FALSE, version = NULL,
        compress = TRUE, refhook = NULL)

readRDS(file, refhook = NULL)
infoRDS(file)
```

Arguments

object	R object to serialize.
file	a connection or the name of the file where the R object is saved to or read from.
ascii	a logical. If TRUE or NA, an ASCII representation is written; otherwise (default), a binary one is used. See the comments in the help for save.
version	the workspace format version to use. NULL specifies the current default version (3). The only other supported value is 2, the default from R 1.4.0 to R 3.5.0.
compress	a logical specifying whether saving to a named file is to use "gzip" compression, or one of "gzip", "bzip2" or "xz" to indicate the type of compression to be used. Ignored if file is a connection.
refhook	a hook function for handling reference objects.

Details

saveRDS and readRDS provide the means to save a single R object to a connection (typically a file) and to restore the object, quite possibly under a different name. This differs from save and load, which save and restore one or more named objects into an environment. They are widely used by R itself, for example to store metadata for a package and to store the help.search databases: the ".rds" file extension is most often used.

Functions serialize and unserialize provide a slightly lower-level interface to serialization: objects serialized to a connection by serialize can be read back by readRDS and conversely.

Function infoRDS retrieves meta-data about serialization produced by saveRDS or serialize. infoRDS cannot be used to detect whether a file is a serialization nor whether it is valid.

All of these interfaces use the same serialization format, but save writes a single line header (typically "RDXs\n") before the serialization of a single object (a pairlist of all the objects to be saved).

If file is a file name, it is opened by gzfile except for save(compress = FALSE) which uses file. Only for the exception are marked encodings of file which cannot be translated to the native encoding handled on Windows.

Compression is handled by the connection opened when file is a file name, so is only possible when file is a connection if handled by the connection. So e.g. url connections will need to be wrapped in a call to gzcon.

If a connection is supplied it will be opened (in binary mode) for the duration of the function if not already open: if it is already open it must be in binary mode for saveRDS(ascii = FALSE) or to read non-ASCII saves.

Value

For readRDS, an R object.

For saveRDS, NULL invisibly.

For infoRDS, an R list with elements version (version number, currently 2 or 3), writer_version (version of R that produced the serialization), min_reader_version (minimum version of R that can read the serialization), format (data representation) and native_encoding (native encoding of the session that produced the serialization, available since version 3). The data representation is given as "xdr" for big-endian binary representation, "ascii" for ASCII representation (produced via ascii = TRUE or ascii = NA) or "binary" (binary representation with native 'endianness' which can be produced by serialize).

Warning

Files produced by saveRDS (or serialize to a file connection) are not suitable as an interchange format between machines, for example to download from a website. The files produced by save have a header identifying the file type and so are better protected against erroneous use.

See Also

serialize, save and load.

The 'R Internals' manual for details of the format used.

Examples

```
fil <- tempfile("women", fileext = ".rds")
## save a single object to file
saveRDS(women, fil)
## restore it under a different name
women2 <- readRDS(fil)
identical(women, women2)
## or examine the object via a connection, which will be opened as needed.
con <- gzfile(fil)
readRDS(con)
close(con)

## Less convenient ways to restore the object
## which demonstrate compatibility with unserialize()
con <- gzfile(fil, "rb")
```

```
identical(unserialize(con), women)
close(con)
con <- gzfile(fil, "rb")
wm <- readBin(con, "raw", n = 1e4) # size is a guess
close(con)
identical(unserialize(wm), women)

## Format compatibility with serialize():
fil2 <- tempfile("women")
con <- file(fil2, "w")
serialize(women, con) # ASCII, uncompressed
close(con)
identical(women, readRDS(fil2))
fil3 <- tempfile("women")
con <- bzfile(fil3, "w")
serialize(women, con) # binary, bzip2-compressed
close(con)
identical(women, readRDS(fil3))

unlink(c(fil, fil2, fil3))
```

`readRenviron` *Set Environment Variables from a File*

Description

Read as file such as '`.Renviron`' or '`Renviron.site`' in the format described in the help for Startup, and set environment variables as defined in the file.

Usage

```
readRenviron(path)
```

Arguments

`path`	A length-one character vector giving the path to the file. Tilde-expansion is performed where supported.

Value

Scalar logical indicating if the file was read successfully. Returned invisibly. If the file cannot be opened for reading, a warning is given.

See Also

`Startup` for the file format.

Examples

```
## Not run:
## re-read a startup file (or read it in a vanilla session)
readRenviron("~/.Renviron")

## End(Not run)
```

`Recall` *Recursive Calling*

Description

`Recall` is used as a placeholder for the name of the function in which it is called. It allows the definition of recursive functions which still work after being renamed, see example below.

Usage

```
Recall(...)
```

Arguments

`...`	all the arguments to be passed.

Note

`Recall` will not work correctly when passed as a function argument, e.g. to the `apply` family of functions.

See Also

`do.call` and `call`.

`local` for another way to write anonymous recursive functions.

Examples

```
## A trivial (but inefficient!) example:
fib <- function(n)
   if(n<=2) { if(n>=0) 1 else 0 } else Recall(n-1) + Recall(n-2)
fibonacci <- fib; rm(fib)
## renaming wouldn't work without Recall
fibonacci(10) # 55
```

`reg.finalizer` *Finalization of Objects*

Description

Registers an R function to be called upon garbage collection of object or (optionally) at the end of an R session.

Usage

```
reg.finalizer(e, f, onexit = FALSE)
```

Arguments

`e`	object to finalize. Must be an environment or an external pointer.
`f`	function to call on finalization. Must accept a single argument, which will be the object to finalize.
`onexit`	logical: should the finalizer be run if the object is still uncollected at the end of the R session?

Details

The main purpose of this function is to allow objects that refer to external items (a temporary file, say) to perform cleanup actions when they are no longer referenced from within R. This only makes sense for objects that are never copied on assignment, hence the restriction to environments and external pointers.

Inter alia, it provides a way to program code to be run at the end of an R session without manipulating `.Last`. For use in a package, it is often a good idea to set a finalizer on an object in the namespace: then it will be called at the end of the session, or soon after the namespace is unloaded if that is done during the session.

Value

`NULL`.

Note

R's interpreter is not re-entrant and the finalizer could be run in the middle of a computation. So there are many functions which it is potentially unsafe to call from `f`: one example which caused trouble is `options`. Finalizers are scheduled at garbage collection but only run at a relatively safe time thereafter.

See Also

`gc` and `Memory` for garbage collection and memory management.

Examples

```
f <- function(e) print("cleaning....")
g <- function(x){ e <- environment(); reg.finalizer(e, f) }
g()
invisible(gc()) # trigger cleanup
```

---

regex *Regular Expressions as used in R*

Description

This help page documents the regular expression patterns supported by grep and related functions grepl, regexpr, gregexpr, sub and gsub, as well as by strsplit and optionally by agrep and agrepl.

Details

A 'regular expression' is a pattern that describes a set of strings. Two types of regular expressions are used in R, *extended* regular expressions (the default) and *Perl-like* regular expressions used by perl = TRUE. There is also fixed = TRUE which can be considered to use a *literal* regular expression.

Other functions which use regular expressions (often via the use of grep) include apropos, browseEnv, help.search, list.files and ls. These will all use *extended* regular expressions.

Patterns are described here as they would be printed by cat: (*do remember that backslashes need to be doubled when entering* R *character strings*, e.g. from the keyboard).

Long regular expression patterns may or may not be accepted: the POSIX standard only requires up to 256 *bytes*.

Extended Regular Expressions

This section covers the regular expressions allowed in the default mode of grep, grepl, regexpr, gregexpr, sub, gsub, regexec and strsplit. They use an implementation of the POSIX 1003.2 standard: that allows some scope for interpretation and the interpretations here are those currently used by R. The implementation supports some extensions to the standard.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. The whole expression matches zero or more characters (read 'character' as 'byte' if useBytes = TRUE).

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. The metacharacters in extended regular expressions are '. \ | ( ) [{ ^ $ * + ?', but note that whether these have a special meaning depends on the context.

Escaping non-metacharacters with a backslash is implementation-dependent. The current implementation interprets '\a' as 'BEL', '\e' as 'ESC', '\f' as 'FF', '\n' as 'LF', '\r' as 'CR' and '\t' as 'TAB'. (Note that these will be interpreted by R's parser in literal character strings.)

A *character class* is a list of characters enclosed between '[' and ']' which matches any single character in that list; unless the first character of the list is the caret '^', when it matches any character *not* in the list. For example, the regular expression '[0123456789]' matches any single digit, and '[^abc]' matches anything except the characters 'a', 'b' or 'c'. A range of characters may be specified by giving the first and last characters, separated by a hyphen. (Because their interpretation is locale- and implementation-dependent, character ranges are best avoided. Some but not all implementations include both cases in ranges when doing caseless matching.) The only portable way to specify all ASCII letters is to list them all as the character class '[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]'.
(The current implementation uses numerical order of the encoding, normally a single-byte encoding or Unicode points.)

Certain named classes of characters are predefined. Their interpretation depends on the *locale* (see locales); the interpretation below is that of the POSIX locale.

'[:alnum:]' Alphanumeric characters: '[:alpha:]' and '[:digit:]'.

'[:alpha:]' Alphabetic characters: '[:lower:]' and '[:upper:]'.

'[:blank:]' Blank characters: space and tab, and possibly other locale-dependent characters, but on most platforms not including non-breaking space.

'[:cntrl:]' Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In another character set, these are the equivalent characters, if any.

'[:digit:]' Digits: '0 1 2 3 4 5 6 7 8 9'.

'[:graph:]' Graphical characters: '[:alnum:]' and '[:punct:]'.

'[:lower:]' Lower-case letters in the current locale.

'[:print:]' Printable characters: '[:alnum:]', '[:punct:]' and space.

'[:punct:]' Punctuation characters:
'! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~'.

'[:space:]' Space characters: tab, newline, vertical tab, form feed, carriage return, space and possibly other locale-dependent characters – on most platforms this does not include non-breaking spaces.

'[:upper:]' Upper-case letters in the current locale.

'[:xdigit:]' Hexadecimal digits:
'0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f'.

For example, '[[:alnum:]]' means '[0-9A-Za-z]', except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside a character class. To include a literal ']', place it first in the list. Similarly, to include a literal '^', place it anywhere but first. Finally, to include a literal '-', place it first or last (or, for perl = TRUE only, precede it by a backslash). (Only '^ - \ ]' are special inside character classes.)

The period '.' matches any single character. The symbol '\w' matches a 'word' character (a synonym for '[[:alnum:]_]', an extension) and '\W' is its negation ('[^[:alnum:]_]'). Symbols '\d', '\s', '\D' and '\S' denote the digit and space classes and their negations (these are all extensions).

The caret '^' and the dollar sign '$' are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols '\<' and '\>' match the empty string at the beginning

and end of a word. The symbol '\b' matches the empty string at either edge of a word, and '\B' matches the empty string provided it is not at an edge of a word. (The interpretation of 'word' depends on the locale and implementation: these are all extensions.)

A regular expression may be followed by one of several repetition quantifiers:

'?' The preceding item is optional and will be matched at most once.

'*' The preceding item will be matched zero or more times.

'+' The preceding item will be matched one or more times.

'{n}' The preceding item is matched exactly n times.

'{n,}' The preceding item is matched n or more times.

'{n,m}' The preceding item is matched at least n times, but not more than m times.

By default repetition is greedy, so the maximal possible number of repeats is used. This can be changed to 'minimal' by appending ? to the quantifier. (There are further quantifiers that allow approximate matching: see the TRE documentation.)

Regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating the substrings that match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator '|'; the resulting regular expression matches any string matching either subexpression. For example, 'abba|cde' matches either the string abba or the string cde. Note that alternation does not work inside character classes, where '|' has its literal meaning.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference '\N', where 'N = 1 ... 9', matches the substring previously matched by the Nth parenthesized subexpression of the regular expression. (This is an extension for extended regular expressions: POSIX defines them only for basic ones.)

Perl-like Regular Expressions

The perl = TRUE argument to grep, regexpr, gregexpr, sub, gsub and strsplit switches to the PCRE library that implements regular expression pattern matching using the same syntax and semantics as Perl 5.x, with just a few differences.

For complete details please consult the man pages for PCRE, especially man pcrepattern and man pcreapi, on your system or from the sources at https://www.pcre.org. (The version in use can be found by calling extSoftVersion. It need not be the version described in the system's man page. PCRE1 (reported as version < 10.00 by extSoftVersion) has been feature-frozen for some time (essentially 2012), the man pages at https://www.pcre.org/original/doc/html/ should be a good match. PCRE2 (PCRE version >= 10.00) has man pages at https://www.pcre.org/current/doc/html/).

Perl regular expressions can be computed byte-by-byte or (UTF-8) character-by-character: the latter is used in all multibyte locales and if any of the inputs are marked as UTF-8 (see Encoding, or as Latin-1 except in a Latin-1 locale.

All the regular expressions described for extended regular expressions are accepted except '\<' and '\>': in Perl all backslashed metacharacters are alphanumeric and backslashed symbols always are interpreted as a literal character. '{' is not special if it would be the start of an invalid interval

specification. There can be more than 9 backreferences (but the replacement in sub can only refer to the first 9).

Character ranges are interpreted in the numerical order of the characters, either as bytes in a single-byte locale or as Unicode code points in UTF-8 mode. So in either case '[A-Za-z]' specifies the set of ASCII letters.

In UTF-8 mode the named character classes only match ASCII characters: see '\p' below for an alternative.

The construct '(?...)' is used for Perl extensions in a variety of ways depending on what immediately follows the '?'.

Perl-like matching can work in several modes, set by the options '(?i)' (caseless, equivalent to Perl's '/i'), '(?m)' (multiline, equivalent to Perl's '/m'), '(?s)' (single line, so a dot matches all characters, even new lines: equivalent to Perl's '/s') and '(?x)' (extended, whitespace data characters are ignored unless escaped and comments are allowed: equivalent to Perl's '/x'). These can be concatenated, so for example, '(?im)' sets caseless multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and to combine setting and unsetting such as '(?im-sx)'. These settings can be applied within patterns, and then apply to the remainder of the pattern. Additional options not in Perl include '(?U)' to set 'ungreedy' mode (so matching is minimal unless '?' is used as part of the repetition quantifier, when it is greedy). Initially none of these options are set.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between '\Q' and '\E'. This is different from Perl in that '$' and '@' are handled as literals in '\Q...\E' sequences in PCRE, whereas in Perl, '$' and '@' cause variable interpolation.

The escape sequences '\d', '\s' and '\w' represent any decimal digit, space character and 'word' character (letter, digit or underscore in the current locale: in UTF-8 mode only ASCII letters and digits are considered) respectively, and their upper-case versions represent their negation. Vertical tab was not regarded as a space character in a C locale before PCRE 8.34. Sequences '\h', '\v', '\H' and '\V' match horizontal and vertical space or the negation. (In UTF-8 mode, these do match non-ASCII Unicode code points.)

There are additional escape sequences: '\cx' is 'cntrl-x' for any 'x', '\ddd' is the octal character (for up to three digits unless interpretable as a backreference, as '\1' to '\7' always are), and '\xhh' specifies a character by two hex digits. In a UTF-8 locale, '\x{h...}' specifies a Unicode code point by one or more hex digits. (Note that some of these will be interpreted by R's parser in literal character strings.)

Outside a character class, '\A' matches at the start of a subject (even in multiline mode, unlike '^'), '\Z' matches at the end of a subject or before a newline at the end, '\z' matches only at end of a subject. and '\G' matches at first matching position in a subject (which is subtly different from Perl's end of the previous match). '\C' matches a single byte, including a newline, but its use is warned against. In UTF-8 mode, '\R' matches any Unicode newline character (not just CR), and '\X' matches any number of Unicode characters that form an extended Unicode sequence. '\X', '\R' and '\B' cannot be used inside a character class (with PCRE1, they are treated as characters 'X', 'R' and 'B'; with PCRE2 they cause an error).

A hyphen (minus) inside a character class is treated as a range, unless it is first or last character in the class definition. It can be quoted to represent the hyphen literal ('\-'). PCRE1 allows an unquoted hyphen at some other locations inside a character class where it cannot represent a valid range, but PCRE2 reports an error in such cases.

In UTF-8 mode, some Unicode properties may be supported via '\p{xx}' and '\P{xx}' which match characters with and without property 'xx' respectively. For a list of supported properties see the PCRE documentation, but for example 'Lu' is 'upper case letter' and 'Sc' is 'currency symbol'. Note that properties such as '\w', '\W', '\d', '\D', '\s', '\S', '\b' and '\B' by default do not refer to full Unicode, but one can override this by starting a pattern with '(*UCP)' (which comes with a performance penalty). (This support depends on the PCRE library being compiled with 'Unicode property support' which can be checked *via* pcre_config. PCRE2 when compiled with Unicode support always supports also Unicode properties.)

The sequence '(?#' marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part at all in the pattern matching.

If the extended option is set, an unescaped '#' character outside a character class introduces a comment that continues up to the next newline character in the pattern.

The pattern '(?:...)' groups characters just as parentheses do but does not make a backreference.

Patterns '(?=...)' and '(?!...)' are zero-width positive and negative lookahead *assertions*: they match if an attempt to match the ... forward from the current position would succeed (or not), but use up no characters in the string being processed. Patterns '(?<=...)' and '(?<!...)' are the lookbehind equivalents: they do not allow repetition quantifiers nor '\C' in ....

regexpr and gregexpr support 'named capture'. If groups are named, e.g., "(?<first>[A-Z][a-z]+)" then the positions of the matches are also returned by name. (Named backreferences are not supported by sub.)

Atomic grouping, possessive qualifiers and conditional and recursive patterns are not covered here.

Author(s)

This help page is based on the TRE documentation and the POSIX standard, and the pcre2pattern man page from PCRE2 10.35.

See Also

grep, apropos, browseEnv, glob2rx, help.search, list.files, ls, strsplit and agrep.

The TRE regexp syntax.

The POSIX 1003.2 standard at https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

The pcre2pattern or pcrepattern man page (found as part of https://www.pcre.org/original/pcre.txt), and details of Perl's own implementation at https://perldoc.perl.org/perlre.

---

regmatches *Extract or Replace Matched Substrings*

Description

Extract or replace matched substrings from match data obtained by regexpr, gregexpr, regexec or gregexec.

Usage

```
regmatches(x, m, invert = FALSE)
regmatches(x, m, invert = FALSE) <- value
```

Arguments

x	a character vector.
m	an object with match data.
invert	a logical: if TRUE, extract or replace the non-matched substrings.
value	an object with suitable replacement values for the matched or non-matched substrings (see Details).

Details

If invert is FALSE (default), regmatches extracts the matched substrings as specified by the match data. For vector match data (as obtained from regexpr), empty matches are dropped; for list match data, empty matches give empty components (zero-length character vectors).

If invert is TRUE, regmatches extracts the non-matched substrings, i.e., the strings are split according to the matches similar to strsplit (for vector match data, at most a single split is performed).

If invert is NA, regmatches extracts both non-matched and matched substrings, always starting and ending with a non-match (empty if the match occurred at the beginning or the end, respectively).

Note that the match data can be obtained from regular expression matching on a modified version of x with the same numbers of characters.

The replacement function can be used for replacing the matched or non-matched substrings. For vector match data, if invert is FALSE, value should be a character vector with length the number of matched elements in m. Otherwise, it should be a list of character vectors with the same length as m, each as long as the number of replacements needed. Replacement coerces values to character or list and generously recycles values as needed. Missing replacement values are not allowed.

Value

For regmatches, a character vector with the matched substrings if m is a vector and invert is FALSE. Otherwise, a list with the matched or/and non-matched substrings.

For regmatches<-, the updated character vector.

Examples

```
x <- c("A and B", "A, B and C", "A, B, C and D", "foobar")
pattern <- "[[:space:]]*(,|and)[[:space:]]"
## Match data from regexpr()
m <- regexpr(pattern, x)
regmatches(x, m)
regmatches(x, m, invert = TRUE)
## Match data from gregexpr()
m <- gregexpr(pattern, x)
regmatches(x, m)
regmatches(x, m, invert = TRUE)
```

```
## Consider
x <- "John (fishing, hunting), Paul (hiking, biking)"
## Suppose we want to split at the comma (plus spaces) between the
## persons, but not at the commas in the parenthesized hobby lists.
## One idea is to "blank out" the parenthesized parts to match the
## parts to be used for splitting, and extract the persons as the
## non-matched parts.
## First, match the parenthesized hobby lists.
m <- gregexpr("\\([^)]*\\)", x)
## Create blank strings with given numbers of characters.
blanks <- function(n) strrep(" ", n)
## Create a copy of x with the parenthesized parts blanked out.
s <- x
regmatches(s, m) <- Map(blanks, lapply(regmatches(s, m), nchar))
s
## Compute the positions of the split matches (note that we cannot call
## strsplit() on x with match data from s).
m <- gregexpr(", *", s)
## And finally extract the non-matched parts.
regmatches(x, m, invert = TRUE)

## regexec() and gregexec() return overlapping ranges because the
## first match is the full match.  This conflicts with regmatches()<-
## and regmatches(..., invert=TRUE).  We can work-around by dropping
## the first match.
drop_first <- function(x) {
    if(!anyNA(x) && all(x > 0)) {
        ml <- attr(x, 'match.length')
        if(is.matrix(x)) x <- x[-1,] else x <- x[-1]
        attr(x, 'match.length') <- if(is.matrix(ml)) ml[-1,] else ml[-1]
    }
    x
}
m <- gregexec("(\\w+) \\(((?:\\w+(?:, )?)+)\\)", x)
regmatches(x, m)
try(regmatches(x, m, invert=TRUE))
regmatches(x, lapply(m, drop_first))
## invert=TRUE loses matrix structure because we are retrieving what
## is in between every sub-match
regmatches(x, lapply(m, drop_first), invert=TRUE)
y <- z <- x
## Notice **list**(...) on the RHS
regmatches(y, lapply(m, drop_first)) <- list(c("<NAME>", "<HOBBY-LIST>"))
y
regmatches(z, lapply(m, drop_first), invert=TRUE) <-
    list(sprintf("<%d>", 1:5))
z

## With `perl = TRUE` and `invert = FALSE` capture group names
## are preserved.  Collect functions and arguments in calls:
NEWS <- head(readLines(file.path(R.home(), 'doc', 'NEWS.2')), 100)
m <- gregexec("(?<fun>\\w+)\\((?<args>[^)]*)\\)", NEWS, perl = TRUE)
```

```
y <- regmatches(NEWS, m)
y[[16]]
## Make tabular, adding original line numbers
mdat <- as.data.frame(t(do.call(cbind, y)))
mdat <- cbind(mdat, line=rep(seq_along(y), lengths(y) / ncol(mdat)))
head(mdat)
NEWS[head(mdat[['line']])]
```

---

remove *Remove Objects from a Specified Environment*

Description

remove and rm are identical R functions that can be used to remove objects. These can be specified successively as character strings, or in the character vector list, or through a combination of both. All objects thus specified will be removed.

If envir is NULL then the currently active environment is searched first.

If inherits is TRUE then parents of the supplied directory are searched until a variable with the given name is encountered. A warning is printed for each variable that is not found.

Usage

```
remove(..., list = character(), pos = -1,
       envir = as.environment(pos), inherits = FALSE)

rm    (..., list = character(), pos = -1,
       envir = as.environment(pos), inherits = FALSE)
```

Arguments

...	the objects to be removed, as names (unquoted) or character strings (quoted).
list	a character vector (or NULL) naming objects to be removed.
pos	where to do the removal. By default, uses the current environment. See 'details' for other possibilities.
envir	the environment to use. See 'details'.
inherits	should the enclosing frames of the environment be inspected?

Details

The pos argument can specify the environment from which to remove the objects in any of several ways: as an integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The envir argument is an alternative way to specify an environment, but is primarily there for back compatibility.

It is not allowed to remove variables from the base environment and base namespace, nor from any environment which is locked (see lockEnvironment).

Earlier versions of R incorrectly claimed that supplying a character vector in ... removed the objects named in the character vector, but it removed the character vector. Use the list argument to specify objects *via* a character vector.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ls, objects

Examples

```
tmp <- 1:4
## work with tmp  and cleanup
rm(tmp)

## Not run:
## remove (almost) everything in the working environment.
## You will get no warning, so don't do this unless you are really sure.
rm(list = ls())

## End(Not run)
```

---

rep *Replicate Elements of Vectors and Lists*

Description

rep replicates the values in x. It is a generic function, and the (internal) default method is described here.

rep.int and rep_len are faster simplified versions for two common cases. Internally, they are generic, so methods can be defined for them (see InternalMethods).

Usage

```
rep(x, ...)

rep.int(x, times)

rep_len(x, length.out)
```

Arguments

x	a vector (of any mode including a list) or a factor or (for rep only) a POSIXct or POSIXlt or Date object; or an S4 object containing such an object.
...	further arguments to be passed to or from other methods. For the internal default method these can include:
	times an integer-valued vector giving the (non-negative) number of times to repeat each element if of length length(x), or to repeat the whole vector if of length 1. Negative or NA values are an error. A double vector is accepted, other inputs being coerced to an integer or double vector.
	length.out non-negative integer. The desired length of the output vector. Other inputs will be coerced to a double vector and the first element taken. Ignored if NA or invalid.
	each non-negative integer. Each element of x is repeated each times. Other inputs will be coerced to an integer or double vector and the first element taken. Treated as 1 if NA or invalid.
times, length.out	see ... above.

Details

The default behaviour is as if the call was

```
rep(x, times = 1, length.out = NA, each = 1)
```

. Normally just one of the additional arguments is specified, but if each is specified with either of the other two, its replication is performed first, and then that implied by times or length.out.

If times consists of a single integer, the result consists of the whole input repeated this many times. If times is a vector of the same length as x (after replication by each), the result consists of x[1] repeated times[1] times, x[2] repeated times[2] times and so on.

length.out may be given in place of times, in which case x is repeated as many times as is necessary to create a vector of this length. If both are given, length.out takes priority and times is ignored.

Non-integer values of times will be truncated towards zero. If times is a computed quantity it is prudent to add a small fuzz or use round. And analogously for each.

If x has length zero and length.out is supplied and is positive, the values are filled in using the extraction rules, that is by an NA of the appropriate class for an atomic vector (0 for raw vectors) and NULL for a list.

Value

An object of the same type as x.

rep.int and rep_len return no attributes (except the class if returning a factor).

The default method of rep gives the result names (which will almost always contain duplicates) if x had names, but retains no other attributes.

Note

Function `rep.int` is a simple case which was provided as a separate function partly for S compatibility and partly for speed (especially when names can be dropped). The performance of `rep` has been improved since, but `rep.int` is still at least twice as fast when `x` has names.

The name `rep.int` long precedes making `rep` generic.

Function `rep` is a primitive, but (partial) matching of argument names is performed as for normal functions.

For historical reasons `rep` (only) works on `NULL`: the result is always `NULL` even when `length.out` is positive.

Although it has never been documented, these functions have always worked on expression vectors.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`seq`, `sequence`, `replicate`.

Examples

```
rep(1:4, 2)
rep(1:4, each = 2)       # not the same.
rep(1:4, c(2,2,2,2))     # same as second.
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, length.out = 4)    # first 4 only.
rep(1:4, each = 2, length.out = 10)   # 8 integers plus two recycled 1's.
rep(1:4, each = 2, times = 3)         # length 24, 3 complete replications

rep(1, 40*(1-.8)) # length 7 on most platforms
rep(1, 40*(1-.8)+1e-7) # better

## replicate a list
fred <- list(happy = 1:10, name = "squash")
rep(fred, 5)

# date-time objects
x <- .leap.seconds[1:3]
rep(x, 2)
rep(as.POSIXlt(x), rep(2, 3))

## named factor
x <- factor(LETTERS[1:4]); names(x) <- letters[1:4]
x
rep(x, 2)
rep(x, each = 2)
rep.int(x, 2)  # no names
rep_len(x, 10)
```

replace *Replace Values in a Vector*

Description

`replace` replaces the values in `x` with indices given in `list` by those given in `values`. If necessary, the values in `values` are recycled.

Usage

```
replace(x, list, values)
```

Arguments

`x`	a vector.
`list`	an index vector.
`values`	replacement values.

Value

A vector with the values replaced.

Note

`x` is unchanged: remember to assign the result.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Reserved *Reserved Words in R*

Description

The reserved words in R's parser are

`if` `else` `repeat` `while` `function` `for` `in` `next` `break`

`TRUE` `FALSE` `NULL` `Inf` `NaN` `NA` `NA_integer_` `NA_real_` `NA_complex_` `NA_character_`

`...` and `..1`, `..2` etc, which are used to refer to arguments passed down from a calling function, see `...`.

Details

Reserved words outside quotes are always parsed to be references to the objects linked to in the 'Description', and hence they are not allowed as syntactic names (see `make.names`). They **are** allowed as non-syntactic names, e.g. inside backtick quotes.

rev *Reverse Elements*

Description

rev provides a reversed version of its argument. It is generic function with a default method for vectors and one for dendrograms.

Note that this is no longer needed (nor efficient) for obtaining vectors sorted into descending order, since that is now rather more directly achievable by sort(x, decreasing = TRUE).

Usage

```
rev(x)
```

Arguments

x	a vector or another object for which reversal is defined.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

seq, sort.

Examples

```
x <- c(1:5, 5:3)
## sort into descending order; first more efficiently:
stopifnot(sort(x, decreasing = TRUE) == rev(sort(x)))
stopifnot(rev(1:7) == 7:1)  #- don't need 'rev' here
```

Rhome *Return the R Home Directory*

Description

Return the R home directory, or the full path to a component of the R installation.

Usage

```
R.home(component = "home")
```

Arguments

component	"home" gives the R home directory, other known values are "bin", "doc", "etc", "include", "modules" and "share" giving the paths to the corresponding parts of an R installation.

Details

The R home directory is the top-level directory of the R installation being run.

The R home directory is often referred to as *R_HOME*, and is the value of an environment variable of that name in an R session. It can be found outside an R session by R RHOME.

The paths to components often are subdirectories of *R_HOME* but need not be: "doc", "include" and "share" are not for some Linux binary installations of R.

Value

A character string giving the R home directory or path to a particular component. Normally the components are all subdirectories of the R home directory, but this need not be the case in a Unix-like installation.

The value for "modules" and on Windows "bin" is a sub-architecture-specific location. (This is not so for "etc", which may have sub-architecture-specific files as well as common ones.)

On a Unix-alike, the constructed paths are based on the current values of the environment variables R_HOME and where set R_SHARE_DIR, R_DOC_DIR and R_INCLUDE_DIR (these are set on startup and should not be altered).

On Windows the values of R.home() and R_HOME are switched to the 8.3 short form of path elements if required and if the Windows service to do that is enabled. The value of R_HOME is set to use forward slashes (since many package maintainers pass it unquoted to shells, for example in 'Makefile's).

See Also

commandArgs()[1] may provide related information.

Examples

```
## These result quite platform-dependently :
rbind(home = R.home(),
      bin  = R.home("bin")) # often the 'bin' sub directory of 'home'
                            # but not always ...
list.files(R.home("bin"))
```

rle *Run Length Encoding*

Description

Compute the lengths and values of runs of equal values in a vector – or the reverse operation.

Usage

```
rle(x)
inverse.rle(x, ...)

## S3 method for class 'rle'
print(x, digits = getOption("digits"), prefix = "", ...)
```

Arguments

x	a vector (atomic, not a list) for rle(); an object of class "rle" for inverse.rle().
...	further arguments; ignored here.
digits	number of significant digits for printing, see print.default.
prefix	character string, prepended to each printed line.

Details

'vector' is used in the sense of is.vector.

Missing values are regarded as unequal to the previous value, even if that is also missing.

inverse.rle() is the inverse function of rle(), reconstructing x from the runs.

Value

rle() returns an object of class "rle" which is a list with components:

lengths	an integer vector containing the length of each run.
values	a vector of the same length as lengths with the corresponding values.

inverse.rle() returns an atomic vector.

Examples

```
x <- rev(rep(6:10, 1:5))
rle(x)
## lengths [1:5]  5 4 3 2 1
## values  [1:5] 10 9 8 7 6

z <- c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE)
rle(z)
```

```
rle(as.character(z))
print(rle(z), prefix = "..| ")

N <- integer(0)
stopifnot(x == inverse.rle(rle(x)),
          identical(N, inverse.rle(rle(N))),
          z == inverse.rle(rle(z)))
```

Round *Rounding of Numbers*

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). See 'Details' about "round to even" when rounding off a 5.

`signif` rounds the values in its first argument to the specified number of *significant* digits. Hence, for `numeric` `x`, `signif(x, dig)` is the same as `round(x, dig - ceiling(log10(abs(x))))`.

Usage

```
ceiling(x)
floor(x)
trunc(x, ...)

round(x, digits = 0, ...)
signif(x, digits = 6)
```

Arguments

`x`	a numeric vector. Or, for `round` and `signif`, a complex vector.
`digits`	integer indicating the number of decimal places (`round`) or significant digits (`signif`) to be used. For `round`, negative values are allowed (see 'Details').
`...`	arguments to be passed to methods.

Details

These are generic functions: methods can be defined for them individually or via the `Math` group generic.

Note that for rounding off a 5, the IEC 60559 standard (see also 'IEEE 754') is expected to be used, '*go to the even digit*'. Therefore `round(0.5)` is `0` and `round(-1.5)` is `-2`. However, this is

dependent on OS services and on representation error (since e.g. `0.15` is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so `round(0.15, 1)` could be either `0.1` or `0.2`).

Rounding to a negative number of digits means rounding to a power of ten, so for example `round(x, digits = -2)` rounds to the nearest hundred.

For `signif` the recognized values of `digits` are `1...22`, and non-missing values are rounded to the nearest integer in that range. Each element of the vector is rounded individually, unlike printing.

These are all primitive functions.

S4 methods

These are all (internally) S4 generic.

`ceiling`, `floor` and `trunc` are members of the `Math` group generic. As an S4 generic, `trunc` has only one argument.

`round` and `signif` are members of the `Math2` group generic.

Warning

The realities of computer arithmetic can cause unexpected results, especially with `floor` and `ceiling`. For example, we 'know' that `floor(log(x, base = 8))` for `x = 8` is `1`, but `0` has been seen on an R platform. It is normally necessary to use a tolerance.

Rounding to decimal digits in binary arithmetic is non-trivial (when `digits != 0`) and may be surprising. Be aware that most decimal fractions are *not* exactly representable in binary double precision. In R 4.0.0, the algorithm for `round(x, d)`, for $d > 0$, has been improved to *measure* and round "to nearest even", contrary to earlier versions of R (or also to `sprintf()` or `format()` based rounding).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

The ISO/IEC/IEEE 60559:2011 standard is available for money from https://www.iso.org.

The IEEE 754:2008 standard is more openly documented, e.g, at https://en.wikipedia.org/wiki/IEEE_754.

See Also

`as.integer`. Package **round**'s `roundX()` for several versions or implementations of rounding, including some previous and the current R version (as `version = "3d.C"`).

Examples

```
round(.5 + -2:4) # IEEE / IEC rounding: -2  0  0  2  2  4  4
## (this is *good* behaviour -- do *NOT* report it as bug !)

( x1 <- seq(-2, 4, by = .5) )
round(x1) #-- IEEE / IEC rounding !
```

```
x1[trunc(x1) != floor(x1)]
x1[round(x1) != floor(x1 + .5)]
(non.int <- ceiling(x1) != floor(x1))

x2 <- pi * 100^(-1:3)
round(x2, 3)
signif(x2, 3)
```

---

round.POSIXt *Round / Truncate Date-Time Objects*

Description

Round or truncate date-time objects.

Usage

```
## S3 method for class 'POSIXt'
round(x,
      units = c("secs", "mins", "hours", "days", "months", "years"))
## S3 method for class 'POSIXt'
trunc(x,
      units = c("secs", "mins", "hours", "days", "months", "years"),
      ...)

## S3 method for class 'Date'
round(x, ...)
## S3 method for class 'Date'
trunc(x,
      units = c("secs", "mins", "hours", "days", "months", "years"),
      ...)
```

Arguments

x	an object inheriting from "POSIXt" or "Date".
units	one of the units listed, a string. Can be abbreviated.
...	arguments to be passed to or from other methods, notably digits for round.

Details

The time is rounded or truncated to the second, minute, hour, day, month or year. Time zones are only relevant to days or more, when midnight in the current time zone is used.

For units arguments besides "months" and "years", the methods for class "Date" are of little use except to remove fractional days.

Value

An object of class "POSIXlt" or "Date".

See Also

round for the generic function and default methods.

DateTimeClasses, Date

Examples

```
round(.leap.seconds + 1000, "hour")

         trunc(Sys.time(), "day")
(timM <- trunc(Sys.time() -> St, "months")) # shows timezone
(datM <- trunc(Sys.Date() -> Sd, "months"))
(timY <- trunc(St, "years")) # + timezone
(datY <- trunc(Sd, "years"))

stopifnot(inherits(datM, "Date"), inherits(timM, "POSIXt"),
          substring(format(datM), 9,10) == "01", # first of month
          substring(format(datY), 6,10) == "01-01", # Jan 1
          identical(format(datM), format(timM)),
          identical(format(datY), format(timY)))
```

---

row *Row Indexes*

Description

Returns a matrix of integers indicating their row number in a matrix-like object, or a factor indicating the row labels.

Usage

```
row(x, as.factor = FALSE)
.row(dim)
```

Arguments

x	a matrix-like object, that is one with a two-dimensional dim.
dim	a matrix dimension, i.e., an integer valued numeric vector of length two (with non-negative entries).
as.factor	a logical value indicating whether the value should be returned as a factor of row labels (created if necessary) rather than as numbers.

Value

An integer (or factor) matrix with the same dimensions as x and whose ij-th element is equal to i (or the i-th row label).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`col` to get columns; `slice.index` for a general way to get slice indices in an array.

Examples

```
x <- matrix(1:12, 3, 4)
# extract the diagonal of a matrix - more slowly than diag(x)
dx <- x[row(x) == col(x)]
dx

# create an identity 5-by-5 matrix more slowly than diag(n = 5):
x <- matrix(0, nrow = 5, ncol = 5)
x[row(x) == col(x)] <- 1
x

(i34 <- .row(3:4))
stopifnot(identical(i34, .row(c(3,4)))) # 'dim' maybe "double"
```

`row+colnames` *Row and Column Names*

Description

Retrieve or set the row or column names of a matrix-like object.

Usage

```
rownames(x, do.NULL = TRUE, prefix = "row")
rownames(x) <- value

colnames(x, do.NULL = TRUE, prefix = "col")
colnames(x) <- value
```

Arguments

`x`	a matrix-like R object, with at least two dimensions for `colnames`.
`do.NULL`	logical. If `FALSE` and names are `NULL`, names are created.
`prefix`	for created names.
`value`	a valid value for that component of `dimnames(x)`. For a matrix or array this is either `NULL` or a character vector of non-zero length equal to the appropriate dimension.

Details

The extractor functions try to do something sensible for any matrix-like object `x`. If the object has `dimnames` the first component is used as the row names, and the second component (if any) is used for the column names. For a data frame, `rownames` and `colnames` eventually call `row.names` and `names` respectively, but the latter are preferred.

If `do.NULL` is `FALSE`, a character vector (of length `NROW(x)` or `NCOL(x)`) is returned in any case, prepending `prefix` to simple numbers, if there are no dimnames or the corresponding component of the dimnames is `NULL`.

The replacement methods for arrays/matrices coerce vector and factor values of `value` to character, but do not dispatch methods for `as.character`.

For a data frame, `value` for `rownames` should be a character vector of non-duplicated and non-missing names (this is enforced), and for `colnames` a character vector of (preferably) unique syntactically-valid names. In both cases, `value` will be coerced by `as.character`, and setting `colnames` will convert the row names to character.

Note

If the replacement versions are called on a matrix without any existing dimnames, they will add suitable dimnames. But constructions such as

```
rownames(x)[3] <- "c"
```

may not work unless `x` already has dimnames, since this will create a length-3 `value` from the `NULL` value of `rownames(x)`.

See Also

`dimnames`, `case.names`, `variable.names`.

Examples

```
m0 <- matrix(NA, 4, 0)
rownames(m0)

m2 <- cbind(1, 1:4)
colnames(m2, do.NULL = FALSE)
colnames(m2) <- c("x","Y")
rownames(m2) <- rownames(m2, do.NULL = FALSE, prefix = "Obs.")
m2
```

`row.names` *Get and Set Row Names for Data Frames*

Description

All data frames have row names, a character vector of length the number of rows with no duplicates nor missing values.

There are generic functions for getting and setting row names, with default methods for arrays. The description here is for the `data.frame` method.

`` `.rowNamesDF<-` `` is a (non-generic replacement) function to set row names for data frames, with extra argument `make.names`. This function only exists as workaround as we cannot easily change the `row.names<-` generic without breaking legacy code in existing packages.

Usage

```
row.names(x)
row.names(x) <- value
.rowNamesDF(x, make.names=FALSE) <- value
```

Arguments

`x`	object of class `"data.frame"`, or any other class for which a method has been defined.
`make.names`	`logical`, i.e., one of `FALSE`, `NA`, `TRUE`, indicating what should happen if the specified row names, i.e., `value`, are invalid, e.g., duplicated or `NA`. The default (is back compatible), `FALSE`, will signal an error, where `NA` will "automatic" row names and `TRUE` will call `make.names(value, unique=TRUE)` for constructing valid names.
`value`	an object to be coerced to character unless an integer vector. It should have (after coercion) the same length as the number of rows of `x` with no duplicated nor missing values. `NULL` is also allowed: see 'Details'.

Details

A data frame has (by definition) a vector of *row names* which has length the number of rows in the data frame, and contains neither missing nor duplicated values. Where a row names sequence has been added by the software to meet this requirement, they are regarded as 'automatic'.

Row names are currently allowed to be integer or character, but for backwards compatibility (with R <= 2.4.0) `row.names` will always return a character vector. (Use `attr(x, "row.names")` if you need to retrieve an integer-valued set of row names.)

Using `NULL` for the value resets the row names to `seq_len(nrow(x))`, regarded as 'automatic'.

Value

`row.names` returns a character vector.

`row.names<-` returns a data frame with the row names changed.

Note

row.names is similar to rownames for arrays, and it has a method that calls rownames for an array argument.

Row names of the form 1:n for n > 2 are stored internally in a compact form, which might be seen from C code or by deparsing but never via row.names or attr(x, "row.names"). Additionally, some names of this sort are marked as 'automatic' and handled differently by as.matrix and data.matrix (and potentially other functions). (All zero-row data frames are regarded as having automatic row names.)

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, rownames, names.

.row_names_info for the internal representations.

Examples

```
## To illustrate the note:
df <- data.frame(x = c(TRUE, FALSE, NA, NA), y = c(12, 34, 56, 78))
row.names(df) <- 1 : 4
attr(df, "row.names") #> 1:4
deparse(df) # or dput(df)
##--> c(NA, 4L) : Compact storage, *not* regarded as automatic.

row.names(df) <- NULL
attr(df, "row.names") #> 1:4
deparse(df) # or dput(df) -- shows
##--> c(NA, -4L) : Compact storage, regarded as automatic.
```

---

rowsum — *Give Column Sums of a Matrix or Data Frame, Based on a Grouping Variable*

Description

Compute column sums across rows of a numeric matrix-like object for each level of a grouping variable. rowsum is generic, with a method for data frames and a default method for vectors and matrices.

Usage

```
rowsum(x, group, reorder = TRUE, ...)

## S3 method for class 'data.frame'
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)

## Default S3 method:
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)
```

Arguments

x	a matrix, data frame or vector of numeric data. Missing values are allowed. A numeric vector will be treated as a column vector.
group	a vector or factor giving the grouping, with one element per row of x. Missing values will be treated as another group and a warning will be given.
reorder	if TRUE, then the result will be in order of sort(unique(group)), if FALSE, it will be in the order that groups were encountered.
na.rm	logical (TRUE or FALSE). Should NA (including NaN) values be discarded?
...	other arguments to be passed to or from methods.

Details

The default is to reorder the rows to agree with tapply as in the example below. Reordering should not add noticeably to the time except when there are very many distinct values of group and x has few columns.

The original function was written by Terry Therneau, but this is a new implementation using hashing that is much faster for large matrices.

To sum over all the rows of a matrix (i.e., a single group) use colSums, which should be even faster.

For integer arguments, over/underflow in forming the sum results in NA.

Value

A matrix or data frame containing the sums. There will be one row per unique value of group.

See Also

tapply, aggregate, rowSums

Examples

```
require(stats)

x <- matrix(runif(100), ncol = 5)
group <- sample(1:8, 20, TRUE)
(xsum <- rowsum(x, group))
## Slower versions
tapply(x, list(group[row(x)], col(x)), sum)
```

```
t(sapply(split(as.data.frame(x), group), colSums))
aggregate(x, list(group), sum)[-1]
```

S3method *Register S3 Methods*

Description

Register S3 methods in R scripts.

Usage

```
.S3method(generic, class, method)
```

Arguments

generic	a character string naming an S3 generic function.
class	a character string naming an S3 class.
method	a character string or function giving the S3 method to be registered. If not given, the function named *generic*.*class* is used.

Details

This function should only be used in R scripts: for package code, one should use the corresponding 'S3method' 'NAMESPACE' directive.

Examples

```
## Create a generic function and register a method for objects
## inheriting from class 'cls':
gen <- function(x) UseMethod("gen")
met <- function(x) writeLines("Hello world.")
.S3method("gen", "cls", met)
## Create an object inheriting from class 'cls', and call the
## generic on it:
x <- structure(123, class = "cls")
gen(x)
```

sample *Random Samples and Permutations*

Description

sample takes a sample of the specified size from the elements of x using either with or without replacement.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)

sample.int(n, size = n, replace = FALSE, prob = NULL,
           useHash = (n > 1e+07 && !replace && is.null(prob) && size <= n/2))
```

Arguments

x	either a vector of one or more elements from which to choose, or a positive integer. See 'Details.'
n	a positive number, the number of items to choose from. See 'Details.'
size	a non-negative integer giving the number of items to choose.
replace	should sampling be with replacement?
prob	a vector of probability weights for obtaining the elements of the vector being sampled.
useHash	logical indicating if the hash-version of the algorithm should be used. Can only be used for replace = FALSE, prob = NULL, and size <= n/2, and really should be used for large n, as useHash=FALSE will use memory proportional to n.

Details

If x has length 1, is numeric (in the sense of is.numeric) and x >= 1, sampling *via* sample takes place from 1:x. *Note* that this convenience feature may lead to undesired behaviour when x is of varying length in calls such as sample(x). See the examples.

Otherwise x can be any R object for which length and subsetting by integers make sense: S3 or S4 methods for these operations will be dispatched as appropriate.

For sample the default for size is the number of items inferred from the first argument, so that sample(x) generates a random permutation of the elements of x (or 1:x).

It is allowed to ask for size = 0 samples with n = 0 or a length-zero x, but otherwise n > 0 or positive length(x) is required.

Non-integer positive numerical values of n or x will be truncated to the next smallest integer, which has to be no larger than .Machine$integer.max.

The optional prob argument can be used to give a vector of weights for obtaining the elements of the vector being sampled. They need not sum to one, but they should be non-negative and not all

zero. If replace is true, Walker's alias method (Ripley, 1987) is used when there are more than 200 reasonably probable values: this gives results incompatible with those from R < 2.2.0.

If replace is false, these probabilities are applied sequentially, that is the probability of choosing the next item is proportional to the weights amongst the remaining items. The number of nonzero weights must be at least size in this case.

sample.int is a bare interface in which both n and size must be supplied as integers.

Argument n can be larger than the largest integer of type integer, up to the largest representable integer in type double. Only uniform sampling is supported. Two random numbers are used to ensure uniform sampling of large integers.

Value

For sample a vector of length size with elements drawn from either x or from the integers 1:x.

For sample.int, an integer vector of length size with elements from 1:n, or a double vector if $n \geq 2^{31}$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Ripley, B. D. (1987) *Stochastic Simulation*. Wiley.

See Also

RNGkind(sample.kind = ..) about random number generation, notably the change of sample() results with R version 3.6.0.

CRAN package **sampling** for other methods of weighted sampling without replacement.

Examples

```
x <- 1:12
# a random permutation
sample(x)
# bootstrap resampling -- only if length(x) > 1 !
sample(x, replace = TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)

## More careful bootstrapping --  Consider this when using sample()
## programmatically (i.e., in your function or simulation)!

# sample()'s surprise -- example
x <- 1:10
    sample(x[x >  8]) # length 2
    sample(x[x >  9]) # oops -- length 10!
    sample(x[x > 10]) # length 0

## safer version:
```

```
resample <- function(x, ...) x[sample.int(length(x), ...)]
resample(x[x >  8]) # length 2
resample(x[x >  9]) # length 1
resample(x[x > 10]) # length 0

## R 3.0.0 and later
sample.int(1e10, 12, replace = TRUE)
sample.int(1e10, 12) # not that there is much chance of duplicates
```

---

save *Save R Objects*

Description

save writes an external representation of R objects to the specified file. The objects can be read back from the file at a later date by using the function load or attach (or data in some cases).

save.image() is just a short-cut for 'save my current workspace', i.e., save(list = ls(all.names = TRUE), file = ".RData", envir = .GlobalEnv). It is also what happens with q("yes").

Usage

```
save(..., list = character(),
     file = stop("'file' must be specified"),
     ascii = FALSE, version = NULL, envir = parent.frame(),
     compress = isTRUE(!ascii), compression_level,
     eval.promises = TRUE, precheck = TRUE)

save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = !ascii, safe = TRUE)
```

Arguments

...	the names of the objects to be saved (as symbols or character strings).
list	a character vector (or NULL) containing the names of objects to be saved.
file	a (writable binary-mode) connection or the name of the file where the data will be saved (when tilde expansion is done). Must be a file name for save.image or version = 1.
ascii	if TRUE, an ASCII representation of the data is written. The default value of ascii is FALSE which leads to a binary file being written. If NA and version >= 2, a different ASCII representation is used which writes double/complex numbers as binary fractions.
version	the workspace format version to use. NULL specifies the current default format (3). Version 1 was the default from R 0.99.0 to R 1.3.1 and version 2 from R 1.4.0 to 3.5.0. Version 3 is supported from R 3.5.0.
envir	environment to search for objects to be saved.

`compress` logical or character string specifying whether saving to a named file is to use compression. `TRUE` corresponds to `gzip` compression, and character strings `"gzip"`, `"bzip2"` or `"xz"` specify the type of compression. Ignored when `file` is a connection and for workspace format version 1.

`compression_level` integer: the level of compression to be used. Defaults to `6` for `gzip` compression and to `9` for `bzip2` or `xz` compression. See the help for `file` for possible values and their merits.

`eval.promises` logical: should objects which are promises be forced before saving?

`precheck` logical: should the existence of the objects be checked before starting to save (and in particular before opening the file/connection)? Does not apply to version 1 saves.

`safe` logical. If `TRUE`, a temporary file is used for creating the saved workspace. The temporary file is renamed to `file` if the save succeeds. This preserves an existing workspace `file` if the save fails, but at the cost of using extra disk space during the save.

Details

The names of the objects specified either as symbols (or character strings) in `...` or as a character vector in `list` are used to look up the objects from environment `envir`. By default promises are evaluated, but if `eval.promises = FALSE` promises are saved (together with their evaluation environments). (Promises embedded in objects are always saved unevaluated.)

All R platforms use the XDR (big-endian) representation of C ints and doubles in binary save-d files, and these are portable across all R platforms.

ASCII saves used to be useful for moving data between platforms but are now mainly of historical interest. They can be more compact than binary saves where compression is not used, but are almost always slower to both read and write: binary saves compress much better than ASCII ones. Further, decimal ASCII saves may not restore double/complex values exactly, and what value is restored may depend on the R platform.

Default values for the `ascii`, `compress`, `safe` and `version` arguments can be modified with the `"save.defaults"` option (used both by `save` and `save.image`), see also the 'Examples' section. If a `"save.image.defaults"` option is set it is used in preference to `"save.defaults"` for function `save.image` (which allows this to have different defaults). In addition, `compression_level` can be part of the `"save.defaults"` option.

A connection that is not already open will be opened in mode `"wb"`. Supplying a connection which is open and not in binary mode gives an error.

Compression

Large files can be reduced considerably in size by compression. A particular 46MB R object was saved as 35MB without compression in 2 seconds, 22MB with `gzip` compression in 8 secs, 19MB with `bzip2` compression in 13 secs and 9.4MB with `xz` compression in 40 secs. The load times were 1.3, 2.8, 5.5 and 5.7 seconds respectively. These results are indicative, but the relative performances do depend on the actual file: `xz` compressed unusually well here.

It is possible to compress later (with gzip, bzip2 or xz) a file saved with compress = FALSE: the effect is the same as saving with compression. Also, a saved file can be uncompressed and re-compressed under a different compression scheme (and see resaveRdaFiles for a way to do so from within R).

Parallel compression

That file can be a connection can be exploited to make use of an external parallel compression utility such as pigz (https://zlib.net/pigz/) or pbzip2 (https://launchpad.net/pbzip2) *via* a pipe connection. For example, using 8 threads,

```
con <- pipe("pigz -p8 > fname.gz", "wb")
save(myObj, file = con); close(con)

con <- pipe("pbzip2 -p8 -9 > fname.bz2", "wb")
save(myObj, file = con); close(con)

con <- pipe("xz -T8 -6 -e > fname.xz", "wb")
save(myObj, file = con); close(con)
```

where the last requires xz 5.1.1 or later built with support for multiple threads (and parallel compression is only effective for large objects: at level 6 it will compress in serialized chunks of 12MB).

Warnings

The ... arguments only give the *names* of the objects to be saved: they are searched for in the environment given by the envir argument, and the actual objects given as arguments need not be those found.

Saved R objects are binary files, even those saved with ascii = TRUE, so ensure that they are transferred without conversion of end-of-line markers and of 8-bit characters. The lines are delimited by LF on all platforms.

Although the default version was not changed between R 1.4.0 and R 3.4.4 nor since R 3.5.0, this does not mean that saved files are necessarily backwards compatible. You will be able to load a saved image into an earlier version of R which supports its version unless use is made of later additions (for example for version 2, raw vectors, external pointers and some S4 objects).

One such 'later addition' was long vectors, introduced in R 3.0.0 and loadable only on 64-bit platforms.

Loading files saved with ASCII = NA requires a C99-compliant C function sscanf: this is a problem on Windows, first worked around in R 3.1.2: version-2 files in that format should be readable in earlier versions of R on all other platforms.

Note

For saving single R objects, saveRDS() is mostly preferable to save(), notably because of the *functional* nature of readRDS(), as opposed to load().

The most common reason for failure is lack of write permission in the current directory. For save.image and for saving at the end of a session this will shown by messages like

```
Error in gzfile(file, "wb") : unable to open connection
In addition: Warning message:
In gzfile(file, "wb") :
  cannot open compressed file '.RDataTmp',
  probable reason 'Permission denied'
```

See Also

dput, dump, load, data.

For other interfaces to the underlying serialization format, see serialize and saveRDS.

Examples

```
x <- stats::runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.RData")
save.image() # creating ".RData" in current working directory
unlink("xy.RData")

# set save defaults using option:
options(save.defaults = list(ascii = TRUE, safe = FALSE))
save.image() # creating ".RData"
if(interactive()) withAutoprint({
   file.info(".RData")
   readLines(".RData", n = 7) # first 7 lines; first starts w/ "RDA"..
})
unlink(".RData")
```

---

scale *Scaling and Centering of Matrix-like Objects*

Description

scale is generic function whose default method centers and/or scales the columns of a numeric matrix.

Usage

```
scale(x, center = TRUE, scale = TRUE)
```

Arguments

x	a numeric matrix(like object).
center	either a logical value or numeric-alike vector of length equal to the number of columns of x, where 'numeric-alike' means that as.numeric(.) will be applied successfully if is.numeric(.) is not true.
scale	either a logical value or a numeric-alike vector of length equal to the number of columns of x.

Details

The value of center determines how column centering is performed. If center is a numeric-alike vector with length equal to the number of columns of x, then each column of x has the corresponding value from center subtracted from it. If center is TRUE then centering is done by subtracting the column means (omitting NAs) of x from their corresponding columns, and if center is FALSE, no centering is done.

The value of scale determines how column scaling is performed (after centering). If scale is a numeric-alike vector with length equal to the number of columns of x, then each column of x is divided by the corresponding value from scale. If scale is TRUE then scaling is done by dividing the (centered) columns of x by their standard deviations if center is TRUE, and the root mean square otherwise. If scale is FALSE, no scaling is done.

The root-mean-square for a (possibly centered) column is defined as $\sqrt{\sum(x^2)/(n-1)}$, where x is a vector of the non-missing values and n is the number of non-missing values. In the case center = TRUE, this is the same as the standard deviation, but in general it is not. (To scale by the standard deviations without centering, use scale(x, center = FALSE, scale = apply(x, 2, sd, na.rm = TRUE)).)

Value

For scale.default, the centered, scaled matrix. The numeric centering and scalings used (if any) are returned as attributes "scaled:center" and "scaled:scale"

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

sweep which allows centering (and scaling) with arbitrary statistics.

For working with the scale of a plot, see par.

Examples

```
require(stats)
x <- matrix(1:10, ncol = 2)
(centered.x <- scale(x, scale = FALSE))
cov(centered.scaled.x <- scale(x)) # all 1
```

---

scan *Read Data Values*

Description

Read data into a vector or list from the console or file.

Usage

```
scan(file = "", what = double(), nmax = -1, n = -1, sep = "",
     quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE,
     quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
     comment.char = "", allowEscapes = FALSE,
     fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
```

Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (or whatever stdin() reads if input is redirected or R is embedded). (In this case input can be terminated by a blank line or an EOF signal, 'Ctrl-D' on Unix and 'Ctrl-Z' on Windows.) Otherwise, the file name is interpreted *relative* to the current working directory (given by getwd()), unless it specifies an *absolute* path. Tilde-expansion is performed where supported. When running R from a script, file = "stdin" can be used to refer to the process's stdin file stream. This can be a compressed file (see file). Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call. Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line and so will match sep = "\n". file can also be a complete URL. (For the supported URL schemes, see the 'URLs' section of the help for url.) To read a data file not in the current encoding (for example a Latin-1 file in a UTF-8 locale or conversely) use a file connection setting its encoding argument (or scan's fileEncoding argument).
what	the type of what gives the type of data to be read. (Here 'type' is used in the sense of typeof.) The supported types are logical, integer, numeric, complex, character, raw and list. If what is a list, it is assumed that the lines of the data file are records each containing length(what) items ('fields') and the list components should have elements which are one of the first six (atomic) types listed or NULL, see section 'Details' below.
nmax	the maximum number of data values to be read, or if what is a list, the maximum number of records to be read. If omitted or not positive or an invalid value for an integer (and nlines is not set to a positive value), scan will read to the end of file.
n	integer: the maximum number of data values to be read, defaulting to no limit. Invalid values will be ignored.
sep	by default, scan expects to read 'white-space' delimited input fields. Alternatively, sep can be used to specify a character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted. If specified this should be the empty character string (the default) or NULL or a character string containing just one single-byte character.

`quote` the set of quoting characters as a single character string or `NULL`. In a multibyte locale the quoting characters must be ASCII (single-byte).

`dec` decimal point character. This should be a character string containing just one single-byte character. (`NULL` and a zero-length character vector are also accepted, and taken as the default.)

`skip` the number of lines of the input file to skip before beginning to read data values.

`nlines` if positive, the maximum number of lines of data to be read.

`na.strings` character vector. Elements of this vector are to be interpreted as missing (`NA`) values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields. Note that the test happens *after* white space is stripped from the input (if enabled), so `na.strings` values may need their own white space stripped in advance.

`flush` logical: if `TRUE`, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field, but precludes putting more than one record on a line.

`fill` logical: if `TRUE`, `scan` will implicitly add empty fields to any lines with fewer fields than implied by `what`.

`strip.white` vector of logical value(s) corresponding to items in the `what` argument. It is used only when `sep` has been specified, and allows the stripping of leading and trailing 'white space' from `character` fields (other fields are always stripped). Note: white space inside quoted strings is not stripped.

If `strip.white` is of length 1, it applies to all fields; otherwise, if `strip.white[i]` is `TRUE` *and* the `i`-th field is of mode character (because `what[i]` is) then the leading and trailing unquoted white space from field `i` is stripped.

`quiet` logical: if `FALSE` (default), scan() will print a line, saying how many items have been read.

`blank.lines.skip` logical: if `TRUE` blank lines in the input are ignored, except when counting `skip` and `nlines`.

`multi.line` logical. Only used if `what` is a list. If `FALSE`, all of a record must appear on one line (but more than one record can appear on a single line). Note that using `fill = TRUE` implies that a record will be terminated at the end of a line.

`comment.char` character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether (the default).

`allowEscapes` logical. Should C-style escapes such as '`\n`' be processed (the default) or read verbatim? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character).

The escapes which are interpreted are the control characters '`\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`' and octal and hexadecimal representations like '`\040`' and '`\0x2A`'. Any other escaped character is treated as itself, including backslash. Note that Unicode escapes (starting '`\u`' or '`\U`': see Quotes) are never processed.

`fileEncoding` character string: if non-empty declares the encoding used on a file (not a connection nor the keyboard) so the character data can be re-encoded. See the 'Encoding' section of the help for `file`, and the 'R Data Import/Export Manual'.

encoding	encoding to be assumed for input strings. If the value is "latin1" or "UTF-8" it is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input (see fileEncoding). See also 'Details'.
text	character string: if file is not supplied and this is, then data are read from the value of text via a text connection.
skipNul	logical: should NULs be skipped when reading character fields?

Details

The value of what can be a list of types, in which case scan returns a list of vectors with the types given by the types of the elements in what. This provides a way of reading columnar data. If any of the types is NULL, the corresponding field is skipped (but a NULL component appears in the result).

The type of what or its components can be one of the six atomic vector types or NULL (see is.atomic).

'White space' is defined for the purposes of this function as one or more contiguous characters from the set space, horizontal tab, carriage return and line feed (aka "newline", "\n"). It does not include form feed nor vertical tab, but in Latin-1 and Windows 8-bit locales (but not UTF-8) 'space' includes the non-breaking space '"\xa0"'.

Empty numeric fields are always regarded as missing values. Empty character fields are scanned as empty character vectors, unless na.strings contains "" when they are regarded as missing values.

The allowed input for a numeric field is optional whitespace, followed by either NA or an optional sign followed by a decimal or hexadecimal constant (see NumericConstants), or NaN, Inf or infinity (ignoring case). Out-of-range values are recorded as Inf, -Inf or 0.

For an integer field the allowed input is optional whitespace, followed by either NA or an optional sign and one or more digits ('0-9'): all out-of-range values are converted to NA_integer_.

If sep is the default (""), the character '\' in a quoted string escapes the following character, so quotes may be included in the string by escaping them.

If sep is non-default, the fields may be quoted in the style of '.csv' files where separators inside quotes ('' or "") are ignored and quotes may be put inside strings by doubling them. However, if sep = "\n" it is assumed by default that one wants to read entire lines verbatim.

Quoting is only interpreted in character fields and in NULL fields (which might be skipping character fields).

Note that since sep is a separator and not a terminator, reading a file by scan("foo", sep = "\n", blank.lines.skip = FALSE) will give an empty final line if the file ends in a line feed ("\n") and not if it does not. This might not be what you expected; see also readLines.

If comment.char occurs (except inside a quoted character field), it signals that the rest of the line should be regarded as a comment and be discarded. Lines beginning with a comment character (possibly after white space with the default separator) are treated as blank lines.

There is a line-length limit of 4095 bytes when reading from the console (which may impose a lower limit: see 'An Introduction to R').

There is a check for a user interrupt every 1000 lines if what is a list, otherwise every 10000 items.

If file is a character string and fileEncoding is non-default, or if it is a not-already-open connection with a non-default encoding argument, the text is converted to UTF-8 and declared as such (and the encoding argument to scan is ignored). See the examples of readLines.

Embedded NULs in the input stream will terminate the field currently being read, with a warning once per call to scan. Setting skipNul = TRUE causes them to be ignored.

Value

if what is a list, a list of the same length and same names (as any) as what.

Otherwise, a vector of the type of what.

Character strings in the result will have a declared encoding if encoding is "latin1" or "UTF-8".

Note

The default for multi.line differs from S. To read one record per line, use flush = TRUE and multi.line = FALSE. (Note that quoted character strings can still include embedded newlines.)

If number of items is not specified, the internal mechanism re-allocates memory in powers of two and so could use up to three times as much memory as needed. (It needs both old and new copies.) If you can, specify either n or nmax whenever inputting a large vector, and nmax or nlines when inputting a large list.

Using scan on an open connection to read partial lines can lose chars: use an explicit separator to avoid this.

Having nul bytes in fields (including '\0' if allowEscapes = TRUE) may lead to interpretation of the field being terminated at the nul. They not normally present in text files – see readBin.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

read.table for more user-friendly reading of data matrices; readLines to read a file a line at a time. write.

Quotes for the details of C-style escape sequences.

readChar and readBin to read fixed or variable length character strings or binary representations of numbers a few at a time from a connection.

Examples

```
cat("TITLE extra line", "2 3 5 7", "11 13 17", file = "ex.data", sep = "\n")
pp <- scan("ex.data", skip = 1, quiet = TRUE)
scan("ex.data", skip = 1)
scan("ex.data", skip = 1, nlines = 1) # only 1 line after the skipped one
scan("ex.data", what = list("","","")) # flush is F -> read "7"
scan("ex.data", what = list("","",""), flush = TRUE)
unlink("ex.data") # tidy up

## "inline" usage
scan(text = "1 2 3")
```

search *Give Search Path for R Objects*

Description

Gives a list of attached *packages* (see library), and R objects, usually data.frames.

Usage

```
search()
searchpaths()
```

Value

A character vector, starting with ".GlobalEnv", and ending with "package:base" which is R's **base** package required always.

searchpaths gives a similar character vector, with the entries for packages being the path to the package used to load the code.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (search.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (searchpaths.)

See Also

.packages to list just the packages on search path.

loadedNamespaces to list loaded namespaces.

attach and detach to change the search path, objects to find R objects in there.

Examples

```
search()
searchpaths()
```

seek *Functions to Reposition Connections*

Description

Functions to re-position connections.

Usage

```
seek(con, ...)
## S3 method for class 'connection'
seek(con, where = NA, origin = "start", rw = "", ...)

isSeekable(con)

truncate(con, ...)
```

Arguments

con	a connection.
where	numeric. A file position (relative to the origin specified by origin), or NA.
rw	character string. Empty or "read" or "write", partial matches allowed.
origin	character string. One of "start", "current", "end": see 'Details'.
...	further arguments passed to or from other methods.

Details

seek with where = NA returns the current byte offset of a connection (from the beginning), and with a non-missing where argument the connection is re-positioned (if possible) to the specified position. isSeekable returns whether the connection in principle supports seek: currently only (possibly gz-compressed) file connections do.

where is stored as a real but should represent an integer: non-integer values are likely to be truncated. Note that the possible values can exceed the largest representable number in an R integer on 64-bit builds, and on some 32-bit builds.

File connections can be open for both writing/appending, in which case R keeps separate positions for reading and writing. Which seek refers to can be set by its rw argument: the default is the last mode (reading or writing) which was used. Most files are only opened for reading or writing and so default to that state. If a file is open for both reading and writing but has not been used, the default is to give the reading position (0).

The initial file position for reading is always at the beginning. The initial position for writing is at the beginning of the file for modes "r+" and "r+b", otherwise at the end of the file. Some platforms only allow writing at the end of the file in the append modes. (The reported write position for a file opened in an append mode will typically be unreliable until the file has been written to.)

gzfile connections support seek with a number of limitations, using the file position of the uncompressed file. They do not support origin = "end". When writing, seeking is only possible

forwards: when reading seeking backwards is supported by rewinding the file and re-reading from its start.

If `seek` is called with a non-`NA` value of `where`, any pushback on a text-mode connection is discarded.

`truncate` truncates a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms: on others (including Windows) it will not work for large (> 2Gb) files.

None of these should be expected to work on text-mode connections with re-encoding selected.

Value

`seek` returns the current position (before any move), as a (numeric) byte offset from the origin, if relevant, or `0` if not. Note that the position can exceed the largest representable number in an R `integer` on 64-bit builds, and on some 32-bit builds.

`truncate` returns `NULL`: it stops with an error if it fails (or is not implemented).

`isSeekable` returns a logical value, whether the connection supports `seek`.

Warning

Use of `seek` on Windows is discouraged. We have found so many errors in the Windows implementation of file positioning that users are advised to use it only at their own risk, and asked not to waste the R developers' time with bug reports on Windows' deficiencies.

See Also

`connections`

---

seq *Sequence Generation*

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

...	arguments passed to or from methods.
from, to	the starting and (maximal) end values of the sequence. Of length 1 unless just from is supplied as an unnamed argument.
by	number: increment of the sequence.
length.out	desired length of the sequence. A non-negative number, which for seq and seq.int will be rounded up if fractional.
along.with	take the length from the length of this argument.

Details

Numerical inputs should all be finite (that is, not infinite, NaN or NA).

The interpretation of the unnamed arguments of seq and seq.int is *not* standard, and it is recommended always to name the arguments when programming.

seq is generic, and only the default method is described here. Note that it dispatches on the class of the **first** argument irrespective of argument names. This can have unintended consequences if it is called with just one argument intending this to be taken as along.with: it is much better to use seq_along in that case.

seq.int is an internal generic which dispatches on methods for "seq" based on the class of the first supplied argument (before argument matching).

Typical usages are

```
seq(from, to)
seq(from, to, by= )
seq(from, to, length.out= )
seq(along.with= )
seq(from)
seq(length.out= )
```

The first form generates the sequence from, from+/-1, ..., to (identical to from:to).

The second form generates from, from+by, ..., up to the sequence value less than or equal to to. Specifying to - from and by of opposite signs is an error. Note that the computed final value can go just beyond to to allow for rounding error, but is truncated to to. ('Just beyond' is by up to 10^{-10} times abs(from - to).)

The third generates a sequence of length.out equally spaced values from from to to. (length.out is usually abbreviated to length or len, and seq_len is much faster.)

The fourth form generates the integer sequence 1, 2, ..., length(along.with). (along.with is usually abbreviated to along, and seq_along is much faster.)

The fifth form generates the sequence 1, 2, ..., length(from) (as if argument along.with had been specified), *unless* the argument is numeric of length 1 when it is interpreted as 1:from (even for seq(0) for compatibility with S). Using either seq_along or seq_len is much preferred (unless strict S compatibility is essential).

The final form generates the integer sequence 1, 2, ..., length.out unless length.out = 0, when it generates integer(0).

Very small sequences (with `from - to` of the order of 10^{-14} times the larger of the ends) will return `from`.

For `seq` (only), up to two of `from`, `to` and `by` can be supplied as complex values provided `length.out` or `along.with` is specified. More generally, the default method of `seq` will handle classed objects with methods for the `Math`, `Ops` and `Summary` group generics.

`seq.int`, `seq_along` and `seq_len` are primitive.

Value

`seq.int` and the default method of `seq` for numeric arguments return a vector of type `"integer"` or `"double"`: programmers should not rely on which.

`seq_along` and `seq_len` return an integer vector, unless it is a *long vector* when it will be double.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The methods `seq.Date` and `seq.POSIXt`.

`:`, `rep`, `sequence`, `row`, `col`.

Examples

```
seq(0, 1, length.out = 11)
seq(stats::rnorm(20)) # effectively 'along'
seq(1, 9, by = 2)     # matches 'end'
seq(1, 9, by = pi)    # stays below 'end'
seq(1, 6, by = 3)
seq(1.575, 5.125, by = 0.05)
seq(17) # same as 1:17, or even better seq_len(17)
```

---

seq.Date *Generate Regular Sequences of Dates*

Description

The method for `seq` for objects of class `"Date"` representing calendar dates.

Usage

```
## S3 method for class 'Date'
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	starting date. Required.
to	end date. Optional.
by	increment of the sequence. Optional. See 'Details'.
length.out	integer, optional. Desired length of the sequence.
along.with	take the length from the length of this argument.
...	arguments passed to or from other methods.

Details

by can be specified in several ways.

- A number, taken to be in days.
- A object of class difftime
- A character string, containing one of "day", "week", "month", "quarter" or "year". This can optionally be preceded by a (positive or negative) integer and a space, or followed by "s".
  See seq.POSIXt for the details of "month".

Value

A vector of class "Date".

See Also

Date

Examples

```
## first days of years
seq(as.Date("1910/1/1"), as.Date("1999/1/1"), "years")
## by month
seq(as.Date("2000/1/1"), by = "month", length.out = 12)
## quarters
seq(as.Date("2000/1/1"), as.Date("2003/1/1"), by = "quarter")

## find all 7th of the month between two dates, the last being a 7th.
st <- as.Date("1998-12-17")
en <- as.Date("2000-1-7")
ll <- seq(en, st, by = "-1 month")
rev(ll[ll > st & ll < en])
```

seq.POSIXt *Generate Regular Sequences of Times*

Description

The method for seq for date-time classes.

Usage

```
## S3 method for class 'POSIXt'
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	starting date. Required.
to	end date. Optional.
by	increment of the sequence. Optional. See 'Details'.
length.out	integer, optional. Desired length of the sequence.
along.with	take the length from the length of this argument.
...	arguments passed to or from other methods.

Details

by can be specified in several ways.

- A number, taken to be in seconds.
- A object of class difftime
- A character string, containing one of "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year". This can optionally be preceded by a (positive or negative) integer and a space, or followed by "s".

The difference between "day" and "DSTday" is that the former ignores changes to/from daylight savings time and the latter takes the same clock time each day. "week" ignores DST (it is a period of 144 hours), but "7 DSTdays" can be used as an alternative. "month" and "year" allow for DST.

The time zone of the result is taken from from: remember that GMT means UTC (and not the time zone of Greenwich, England) and so does not have daylight savings time.

Using "month" first advances the month without changing the day: if this results in an invalid day of the month, it is counted forward into the next month: see the examples.

Value

A vector of class "POSIXct".

See Also

DateTimeClasses

Examples

```
## first days of years
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
## by month
seq(ISOdate(2000,1,1), by = "month", length.out = 12)
seq(ISOdate(2000,1,31), by = "month", length.out = 4)
## quarters
seq(ISOdate(1990,1,1), ISOdate(2000,1,1), by = "quarter") # or "3 months"
## days vs DSTdays: use c() to lose the time zone.
seq(c(ISOdate(2000,3,20)), by = "day", length.out = 10)
seq(c(ISOdate(2000,3,20)), by = "DSTday", length.out = 10)
seq(c(ISOdate(2000,3,20)), by = "7 DSTdays", length.out = 4)
```

---

sequence *Create A Vector of Sequences*

Description

The default method for `sequence` generates the sequence `seq(from[i], by = by[i], length.out = nvec[i])` for each element `i` in the parallel (and recycled) vectors `from`, `by` and `nvec`. It then returns the result of concatenating those sequences.

Usage

```
sequence(nvec, ...)
## Default S3 method:
sequence(nvec, from = 1L, by = 1L, ...)
```

Arguments

`nvec`	coerced to a non-negative integer vector each element of which specifies the length of a sequence.
`from`	coerced to an integer vector each element of which specifies the first element of a sequence.
`by`	coerced to an integer vector each element of which specifies the step size between elements of a sequence.
`...`	additional arguments passed to methods.

Details

Negative values are supported for `from` and `by`. `sequence(nvec, from, by=0L)` is equivalent to `rep(from, each=nvec)`.

This function was originally implemented in R with fewer features, but it has since become more flexible, and the default method is implemented in C for speed.

Author(s)

Of the current version, Michael Lawrence based on code from the S4Vectors Bioconductor package

See Also

`gl`, `seq`, `rep`.

Examples

```
sequence(c(3, 2)) # the concatenated sequences 1:3 and 1:2.
#> [1] 1 2 3 1 2
sequence(c(3, 2), from=2L)
#> [1] 2 3 4 2 3
sequence(c(3, 2), from=2L, by=2L)
#> [1] 2 4 6 2 4
sequence(c(3, 2), by=c(-1L, 1L))
#> [1] 1 0 -1 1 2
```

`serialize` *Simple Serialization Interface*

Description

A simple low-level interface for serializing to connections.

Usage

```
serialize(object, connection, ascii, xdr = TRUE,
          version = NULL, refhook = NULL)

unserialize(connection, refhook = NULL)
```

Arguments

`object`	R object to serialize.
`connection`	an open connection or (for `serialize`) `NULL` or (for `unserialize`) a raw vector (see 'Details').
`ascii`	a logical. If `TRUE` or `NA`, an ASCII representation is written; otherwise (default) a binary one. See also the comments in the help for `save`.
`xdr`	a logical: if a binary representation is used, should a big-endian one (XDR) be used?
`version`	the workspace format version to use. `NULL` specifies the current default version (3). The only other supported value is 2, the default from R 1.4.0 to R 3.5.0.
`refhook`	a hook function for handling reference objects.

Details

The function serialize serializes object to the specified connection. If connection is NULL then object is serialized to a raw vector, which is returned as the result of serialize.

Sharing of reference objects is preserved within the object but not across separate calls to serialize.

unserialize reads an object (as written by serialize) from connection or a raw vector.

The refhook functions can be used to customize handling of non-system reference objects (all external pointers and weak references, and all environments other than namespace and package environments and .GlobalEnv). The hook function for serialize should return a character vector for references it wants to handle; otherwise it should return NULL. The hook for unserialize will be called with character vectors supplied to serialize and should return an appropriate object.

For a text-mode connection, the default value of ascii is set to TRUE: only ASCII representations can be written to text-mode connections and attempting to use ascii = FALSE will throw an error.

The format consists of a single line followed by the data: the first line contains a single character: X for binary serialization and A for ASCII serialization, followed by a new line. (The format used is identical to that used by readRDS.)

As almost all systems in current use are little-endian, xdr = FALSE can be used to avoid byte-shuffling at both ends when transferring data from one little-endian machine to another (or between processes on the same machine). Depending on the system, this can speed up serialization and unserialization by a factor of up to 3x.

Value

For serialize, NULL unless connection = NULL, when the result is returned in a raw vector.

For unserialize an R object.

Warning

These functions have provided a stable interface since R 2.4.0 (when the storage of serialized objects was changed from character to raw vectors). However, the serialization format may change in future versions of R, so this interface should not be used for long-term storage of R objects.

On 32-bit platforms a raw vector is limited to $2^{31} - 1$ bytes, but R objects can exceed this and their serializations will normally be larger than the objects.

See Also

saveRDS for a more convenient interface to serialize an object to a file or connection.

save and load to serialize and restore one or more named objects.

The 'R Internals' manual for details of the format used.

Examples

```
x <- serialize(list(1,2,3), NULL)
unserialize(x)

## see also the examples for saveRDS
```

sets *Set Operations*

Description

Performs **set** union, intersection, (asymmetric!) difference, equality and membership on two vectors.

Usage

```
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)

is.element(el, set)
```

Arguments

x, y, el, set	vectors (of the same mode) containing a sequence of items (conceptually) with no duplicated values.

Details

Each of union, intersect, setdiff and setequal will discard any duplicated values in the arguments, and they apply as.vector to their arguments (and so in particular coerce factors to character vectors).

is.element(x, y) is identical to x %in% y.

Value

For union, a vector of a common mode.

For intersect, a vector of a common mode, or NULL if x or y is NULL.

For setdiff, a vector of the same mode as x.

A logical scalar for setequal and a logical of the same length as x for is.element.

See Also

%in%

'plotmath' for the use of union and intersect in plot annotation.

Examples

```
(x <- c(sort(sample(1:20, 9)), NA))
(y <- c(sort(sample(3:23, 7)), NA))
union(x, y)
intersect(x, y)
setdiff(x, y)
setdiff(y, x)
setequal(x, y)

## True for all possible x & y :
setequal( union(x, y),
          c(setdiff(x, y), intersect(x, y), setdiff(y, x)))

is.element(x, y) # length 10
is.element(y, x) # length  8
```

`setTimeLimit` *Set CPU and/or Elapsed Time Limits*

Description

Functions to set CPU and/or elapsed time limits for top-level computations or the current session.

Usage

```
setTimeLimit(cpu = Inf, elapsed = Inf, transient = FALSE)

setSessionTimeLimit(cpu = Inf, elapsed = Inf)
```

Arguments

`cpu`, `elapsed`	double (of length one). Set a limit on the total or elapsed CPU time in seconds, respectively.
`transient`	logical. If `TRUE`, the limits apply only to the rest of the current computation.

Details

`setTimeLimit` sets limits which apply to each top-level computation, that is a command line (including any continuation lines) entered at the console or from a file. If it is called from within a computation the limits apply to the rest of the computation and (unless `transient = TRUE`) to subsequent top-level computations.

`setSessionTimeLimit` sets limits for the rest of the session. Once a session limit is reached it is reset to `Inf`.

Setting any limit has a small overhead – well under 1% on the systems measured.

Time limits are checked whenever a user interrupt could occur. This will happen frequently in R code and during `Sys.sleep`, but only at points in compiled C and Fortran code identified by the code author.

'Total CPU time' includes that used by child processes where the latter is reported.

showConnections *Display Connections*

Description

Display aspects of connections.

Usage

```
showConnections(all = FALSE)
getConnection(what)
closeAllConnections()

stdin()
stdout()
stderr()
nullfile()

isatty(con)

getAllConnections()
```

Arguments

all	logical: if true all connections, including closed ones and the standard ones are displayed. If false only open user-created connections are included.
what	integer: a row number of the table given by showConnections.
con	a connection.

Details

stdin(), stdout() and stderr() are standard connections corresponding to input, output and error on the console respectively (and not necessarily to file streams). They are text-mode connections of class "terminal" which cannot be opened or closed, and are read-only, write-only and write-only respectively. The stdout() and stderr() connections can be re-directed by sink (and in some circumstances the output from stdout() can be split: see the help page).

The encoding for stdin() when redirected can be set by the command-line flag '--encoding'.

nullfile() returns filename of the null device ("/dev/null" on Unix, "nul:" on Windows).

showConnections returns a matrix of information. If a connection object has been lost or forgotten, getConnection will take a row number from the table and return a connection object for that connection, which can be used to close the connection, for example. However, if there is no R level object referring to the connection it will be closed automatically at the next garbage collection (except for gzcon connections).

closeAllConnections closes (and destroys) all user connections, restoring all sink diversions as it does so.

`isatty` returns true if the connection is one of the class `"terminal"` connections and it is apparently connected to a terminal, otherwise false. This may not be reliable in embedded applications, including GUI consoles.

`getAllConnections` returns a sequence of integer connection descriptors for use with `getConnection`, corresponding to the row names of the table returned by `showConnections(all = TRUE)`.

Value

`stdin()`, `stdout()` and `stderr()` return connection objects.

`showConnections` returns a character matrix of information with a row for each connection, by default only for open non-standard connections.

`getConnection` returns a connection object, or `NULL`.

Note

`stdin()` refers to the 'console' and not to the C-level '`stdin`' of the process. The distinction matters in GUI consoles (which may not have an active '`stdin`', and if they do it may not be connected to console input), and also in embedded applications. If you want access to the C-level file stream '`stdin`', use `file("stdin")`.

When R is reading a script from a file, the *file* is the 'console': this is traditional usage to allow in-line data (see 'An Introduction to R' for an example).

See Also

`connections`

Examples

```
showConnections(all = TRUE)
## Not run:
textConnection(letters)
# oops, I forgot to record that one
showConnections()
#  class     description      mode text   isopen   can read can write
#3 "letters" "textConnection" "r"  "text" "opened" "yes"    "no"
mycon <- getConnection(3)

## End(Not run)

c(isatty(stdin()), isatty(stdout()), isatty(stderr()))
```

shQuote *Quote Strings for Use in OS Shells*

Description

Quote a string to be passed to an operating system shell.

Usage

```
shQuote(string, type = c("sh", "csh", "cmd", "cmd2"))
```

Arguments

string	a character vector, usually of length one.
type	character: the type of shell quoting. Partial matching is supported. "cmd" and "cmd2" refer to the Windows shell. "cmd" is the default under Windows.

Details

The default type of quoting supported under Unix-alikes is that for the Bourne shell sh. If the string does not contain single quotes, we can just surround it with single quotes. Otherwise, the string is surrounded in double quotes, which suppresses all special meanings of metacharacters except dollar, backquote and backslash, so these (and of course double quote) are preceded by backslash. This type of quoting is also appropriate for bash, ksh and zsh.

The other type of quoting is for the C-shell (csh and tcsh). Once again, if the string does not contain single quotes, we can just surround it with single quotes. If it does contain single quotes, we can use double quotes provided it does not contain dollar or backquote (and we need to escape backslash, exclamation mark and double quote). As a last resort, we need to split the string into pieces not containing single quotes (some may be empty) and surround each with single quotes, and the single quotes with double quotes.

In Windows, command line interpretation is done by the application as well as the shell. It may depend on the compiler used: Microsoft's rules for the C run-time are given at https://learn.microsoft.com/en-us/cpp/c-language/parsing-c-command-line-arguments?view=msvc-160. It may depend on the whim of the programmer of the application: check its documentation. The type = "cmd" prepares the string for parsing as an argument by the Microsoft's rules and makes shQuote safe for use with many applications when used with system or system2. It surrounds the string by double quotes and escapes internal double quotes by a backslash. Any trailing backslashes and backslashes that were originally before double quotes are doubled.

The Windows cmd.exe shell (used by default with shell) uses type = "cmd2" quoting: special characters are prefixed with "^". In some cases, two types of quoting should be used: first for the application, and then type = "cmd2" for cmd.exe. See the examples below.

Value

A character vector of the same length as string.

References

Loukides, M. *et al* (2002) *Unix Power Tools* Third Edition. O'Reilly. Section 27.12.

Discussion in PR#16636.

See Also

Quotes for quoting R code.

sQuote for quoting English text.

Examples

```
test <- "abc$def`gh`i\\j"
cat(shQuote(test), "\n")
## Not run: system(paste("echo", shQuote(test)))
test <- "don't do it!"
cat(shQuote(test), "\n")

tryit <- paste("use the", sQuote("-c"), "switch\nlike this")
cat(shQuote(tryit), "\n")
## Not run: system(paste("echo", shQuote(tryit)))
cat(shQuote(tryit, type = "csh"), "\n")

## Windows-only example, assuming cmd.exe:
perlcmd <- 'print "Hello World\\n";'
## Not run:
shell(shQuote(paste("perl -e",
                    shQuote(perlcmd, type = "cmd")),
              type = "cmd2"))

## End(Not run)
```

---

sign *Sign Function*

Description

sign returns a vector with the signs of the corresponding elements of x (the sign of a real number is 1, 0, or -1 if the number is positive, zero, or negative, respectively).

Note that sign does not operate on complex vectors.

Usage

```
sign(x)
```

Arguments

x	a numeric vector

Details

This is an internal generic primitive function: methods can be defined for it directly or via the `Math` group generic.

See Also

`abs`

Examples

```
sign(pi)    # == 1
sign(-2:3)  # -1 -1 0 1 1 1
```

`Signals` *Interrupting Execution of R*

Description

On receiving `SIGUSR1` R will save the workspace and quit. `SIGUSR2` has the same result except that the `.Last` function and `on.exit` expressions will not be called.

Usage

```
kill -USR1 pid
kill -USR2 pid
```

Arguments

`pid`	The process ID of the R process.

Details

The commands history will also be saved if would be at normal termination.

This is not available on Windows, and possibly on other OSes which do not support these signals.

Warning

It is possible that one or more R objects will be undergoing modification at the time the signal is sent. These objects could be saved in a corrupted form.

See Also

`Sys.getpid` to report the process ID for future use.

sink *Send R Output to a File*

Description

sink diverts R output to a connection (and stops such diversions).

sink.number() reports how many diversions are in use.

sink.number(type = "message") reports the number of the connection currently being used for error messages.

Usage

```
sink(file = NULL, append = FALSE, type = c("output", "message"),
     split = FALSE)

sink.number(type = c("output", "message"))
```

Arguments

file	a writable connection or a character string naming the file to write to, or NULL to stop sink-ing.
append	logical. If TRUE, output will be appended to file; otherwise, it will overwrite the contents of file.
type	character string. Either the output stream or the messages stream. The name will be partially matched so can be abbreviated.
split	logical: if TRUE, output will be sent to the new sink and to the current output stream, like the Unix program tee.

Details

sink diverts R output to a connection (and must be used again to finish such a diversion, see below!). If file is a character string, a file connection with that name will be established for the duration of the diversion.

Normal R output (to connection stdout) is diverted by the default type = "output". Only prompts and (most) messages continue to appear on the console. Messages sent to stderr() (including those from message, warning and stop) can be diverted by sink(type = "message") (see below).

sink() or sink(file = NULL) ends the last diversion (of the specified type). There is a stack of diversions for normal output, so output reverts to the previous diversion (if there was one). The stack is of up to 21 connections (20 diversions).

If file is a connection it will be opened if necessary (in "wt" mode) and closed once it is removed from the stack of diversions.

split = TRUE only splits R output (via Rvprintf) and the default output from writeLines: it does not split all output that might be sent to stdout().

Sink-ing the messages stream should be done only with great care. For that stream `file` must be an already open connection, and there is no stack of connections.

If `file` is a character string, the file will be opened using the current encoding. If you want a different encoding (e.g., to represent strings which have been stored in UTF-8), use a `file` connection — but some ways to produce R output will already have converted such strings to the current encoding.

Value

`sink` returns `NULL`.

For `sink.number()` the number (0, 1, 2, ...) of diversions of output in place.

For `sink.number("message")` the connection number used for messages, 2 if no diversion has been used.

Warning

Do not use a connection that is open for `sink` for any other purpose. The software will stop you closing one such inadvertently.

Do not sink the messages stream unless you understand the source code implementing it and hence the pitfalls.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`capture.output`

Examples

```
sink("sink-examp.txt")
i <- 1:10
outer(i, i)
sink()


## capture all the output to a file.
zz <- file("all.Rout", open = "wt")
sink(zz)
sink(zz, type = "message")
try(log("a"))
## revert output back to the console -- only then access the file!
sink(type = "message")
sink()
file.show("all.Rout", delete.file = TRUE)
```

slice.index *Slice Indexes in an Array*

Description

Returns a matrix of integers indicating the number of their slice in a given array.

Usage

```
slice.index(x, MARGIN)
```

Arguments

x	an array. If x has no dimension attribute, it is considered a one-dimensional array.
MARGIN	an integer vector giving the dimension numbers to slice by.

Details

If MARGIN gives a single dimension, then all elements of slice number i with respect to this have value i. In general, slice numbers are obtained by numbering all combinations of indices in the dimensions given by MARGIN in column-major order. I.e., with $m_1, \ldots, m_k$ the dimension numbers (elements of MARGIN) sliced by and $d_{m_1}, \ldots, d_{m_k}$ the corresponding extents, and $n_1 = 1, n_2 = d_{m_1}$, $\ldots, n_k = d_{m_1} \cdots d_{m_{k-1}}$, the number of the slice where dimension m_1 has value $i_1, \ldots,$ dimension m_k has value i_k is $1 + n_1(i_1 - 1) + \cdots + n_k(i_k - 1)$.

Value

An integer array y with dimensions corresponding to those of x.

See Also

row and col for determining row and column indexes; in fact, these are special cases of slice.index corresponding to MARGIN equal to 1 and 2, respectively when x is a matrix.

Examples

```
x <- array(1 : 24, c(2, 3, 4))
slice.index(x, 2)
slice.index(x, c(1, 3))
## When slicing by dimensions 1 and 3, slice index 5 is obtained for
## dimension 1 has value 1 and dimension 3 has value 3 (see above):
which(slice.index(x, c(1, 3)) == 5, arr.ind = TRUE)
```

slotOp *Extract or Replace a Slot or Property*

Description

Extract or replace the contents of a slot or property of an object.

Usage

```
object@name
object@name <- value
```

Arguments

object	An object from a formally defined (S4) class, or an object with a class for which '@' or '@<-' S3 methods are defined.
name	The name of the slot or property, supplied as a character string or unquoted symbol. If object has an S4 class, then name must be the name of a slot in the definition of the class of object.
value	A suitable replacement value for the slot or property. For an S4 object this must be from a class compatible with the class defined for this slot in the definition of the class of object.

Details

If object is not an S4 object, then a suitable S3 method for '@' or '@<-' is searched for. If no method is found, then an error is signaled.

if object is an S4 object, then these operators are for slot access, and are enabled only when package **methods** is loaded (as per default). The slot must be formally defined. (There is an exception for the name .Data, intended for internal use only.) The replacement operator checks that the slot already exists on the object (which it should if the object is really from the class it claims to be). See slot for further details, in particular for the differences between slot() and the @ operator.

These are internal generic operators: see InternalMethods.

Value

The current contents of the slot.

See Also

Extract, slot

socketSelect *Wait on Socket Connections*

Description

Waits for the first of several socket connections and server sockets to become available.

Usage

```
socketSelect(socklist, write = FALSE, timeout = NULL)
```

Arguments

socklist	list of open socket connections and server sockets.
write	logical. If TRUE wait for corresponding socket to become available for writing; otherwise wait for it to become available for reading or for accepting an incoming connection (server sockets).
timeout	numeric or NULL. Time in seconds to wait for a socket to become available; NULL means wait indefinitely.

Details

The values in write are recycled if necessary to make up a logical vector the same length as socklist. Socket connections can appear more than once in socklist; this can be useful if you want to determine whether a socket is available for reading or writing.

Value

Logical the same length as socklist indicating whether the corresponding socket connection is available for output or input, depending on the corresponding value of write. Server sockets can only become available for input.

Examples

```
## Not run:
## test whether socket connection s is available for writing or reading
socketSelect(list(s, s), c(TRUE, FALSE), timeout = 0)

## End(Not run)
```

`solve` *Solve a System of Equations*

Description

This generic function solves the equation `a %*% x = b` for `x`, where `b` can be either a vector or a matrix.

Usage

```
solve(a, b, ...)

## Default S3 method:
solve(a, b, tol, LINPACK = FALSE, ...)
```

Arguments

`a`	a square numeric or complex matrix containing the coefficients of the linear system. Logical matrices are coerced to numeric.
`b`	a numeric or complex vector or matrix giving the right-hand side(s) of the linear system. If missing, `b` is taken to be an identity matrix and `solve` will return the inverse of `a`.
`tol`	the tolerance for detecting linear dependencies in the columns of `a`. The default is `.Machine$double.eps`.
`LINPACK`	logical. Defunct and an error.
`...`	further arguments passed to or from other methods.

Details

`a` or `b` can be complex, but this uses double complex arithmetic which might not be available on all platforms.

The row and column names of the result are taken from the column names of `a` and of `b` respectively. If `b` is missing the column names of the result are the row names of `a`. No check is made that the column names of `a` match the row names of `b`.

For back-compatibility `a` can be a (real) QR decomposition, although `qr.solve` should be called in that case. `qr.solve` can handle non-square systems.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

What happens if `a` and/or `b` contain missing, `NaN` or infinite values is platform-dependent, including on the version of LAPACK is in use.

`tol` is a tolerance for the (estimated 1-norm) 'reciprocal condition number': the check is skipped if `tol <= 0`.

For historical reasons, the default method accepts `a` as an object of class `"qr"` (with a warning) and passes it on to `solve.qr`.

Source

The default method is an interface to the LAPACK routines DGESV and ZGESV.

LAPACK is from https://netlib.org/lapack/.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

solve.qr for the qr method, chol2inv for inverting from the Cholesky factor backsolve, qr.solve.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, `+`) }
h8 <- hilbert(8); h8
sh8 <- solve(h8)
round(sh8 %*% h8, 3)

A <- hilbert(4)
A[] <- as.complex(A)
## might not be supported on all platforms
try(solve(A))
```

---

sort *Sorting or Ordering Vectors*

Description

Sort (or *order*) a vector or factor (partially) into ascending or descending order. For ordering along more than one variable, e.g., for sorting data frames, see order.

Usage

```
sort(x, decreasing = FALSE, ...)

## Default S3 method:
sort(x, decreasing = FALSE, na.last = NA, ...)

sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
         method = c("auto", "shell", "quick", "radix"), index.return = FALSE)
```

Arguments

`x`	for `sort` an R object with a class or a numeric, complex, character or logical vector. For `sort.int`, a numeric, complex, character or logical vector, or a factor.
`decreasing`	logical. Should the sort be increasing or decreasing? Not available for partial sorting.
`...`	arguments to be passed to or from methods or (for the default methods and objects without a class) to `sort.int`.
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed.
`partial`	`NULL` or a vector of indices for partial sorting.
`method`	character string specifying the algorithm used. Not available for partial sorting. Can be abbreviated.
`index.return`	logical indicating if the ordering index vector should be returned as well. Supported by `method == "radix"` for any `na.last` mode and data type, and the other methods when `na.last = NA` (the default) and fully sorting non-factors.

Details

`sort` is a generic function for which methods can be written, and `sort.int` is the internal method which is compatible with S if only the first three arguments are used.

The default `sort` method makes use of `order` for classed objects, which in turn makes use of the generic function `xtfrm` (and can be slow unless a `xtfrm` method has been defined or `is.numeric(x)` is true).

Complex values are sorted first by the real part, then the imaginary part.

The `"auto"` method selects `"radix"` for short (less than 2^{31} elements) numeric vectors, integer vectors, logical vectors and factors; otherwise, `"shell"`.

Except for method `"radix"`, the sort order for character vectors will depend on the collating sequence of the locale in use: see `Comparison`. The sort order for factors is the order of their levels (which is particularly appropriate for ordered factors).

If `partial` is not `NULL`, it is taken to contain indices of elements of the result which are to be placed in their correct positions in the sorted array by partial sorting. For each of the result values in a specified position, any values smaller than that one are guaranteed to have a smaller index in the sorted array and any values which are greater are guaranteed to have a bigger index in the sorted array. (This is included for efficiency, and many of the options are not available for partial sorting. It is only substantially more efficient if `partial` has a handful of elements, and a full sort is done (a Quicksort if possible) if there are more than 10.) Names are discarded for partial sorting.

Method `"shell"` uses Shellsort (an $O(n^{4/3})$ variant from Sedgewick (1986)). If `x` has names a stable modification is used, so ties are not reordered. (This only matters if names are present.)

Method `"quick"` uses Singleton (1969)'s implementation of Hoare's Quicksort method and is only available when `x` is numeric (double or integer) and `partial` is `NULL`. (For other types of `x` Shellsort is used, silently.) It is normally somewhat faster than Shellsort (perhaps 50% faster on vectors of length a million and twice as fast at a billion) but has poor performance in the rare worst case.

(Peto's modification using a pseudo-random midpoint is used to make the worst case rarer.) This is not a stable sort, and ties may be reordered.

Method `"radix"` relies on simple hashing to scale time linearly with the input size, i.e., its asymptotic time complexity is O(n). The specific variant and its implementation originated from the data.table package and are due to Matt Dowle and Arun Srinivasan. For small inputs (< 200), the implementation uses an insertion sort (O(n^2)) that operates in-place to avoid the allocation overhead of the radix sort. For integer vectors of range less than 100,000, it switches to a simpler and faster linear time counting sort. In all cases, the sort is stable; the order of ties is preserved. It is the default method for integer vectors and factors.

The `"radix"` method generally outperforms the other methods, especially for small integers. Compared to quick sort, it is slightly faster for vectors with large integer or real values (but unlike quick sort, radix is stable and supports all `na.last` options). The implementation is orders of magnitude faster than shell sort for character vectors, but collation *does not respect the locale* and so gives incorrect answers even in English locales.

However, there are some caveats for the radix sort:

- If `x` is a `character` vector, all elements must share the same encoding. Only UTF-8 (including ASCII) and Latin-1 encodings are supported. Collation follows that with `LC_COLLATE=C`, that is lexicographically byte-by-byte using numerical ordering of bytes.
- Long vectors (with 2^{31} or more elements) and `complex` vectors are not supported.

Value

For `sort`, the result depends on the S3 method which is dispatched. If `x` does not have a class `sort.int` is used and it description applies. For classed objects which do not have a specific method the default method will be used and is equivalent to `x[order(x, ...)]`: this depends on the class having a suitable method for `[` (and also that `order` will work, which requires a `xtfrm` method).

For `sort.int` the value is the sorted vector unless `index.return` is true, when the result is a list with components named `x` and `ix` containing the sorted numbers and the ordering index vector. In the latter case, if `method == "quick"` ties may be reversed in the ordering (unlike `sort.list`) as quicksort is not stable. For `method == "radix"`, `index.return` is supported for all `na.last` modes. The other methods only support `index.return` when `na.last` is `NA`. The index vector refers to element numbers *after removal of* `NA`*s*: see `order` if you want the original element numbers.

All attributes are removed from the return value (see Becker et al., 1988, p.146) except names, which are sorted. (If `partial` is specified even the names are removed.) Note that this means that the returned value has no class, except for factors and ordered factors (which are treated specially and whose result is transformed back to the original class).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley.

Sedgewick, R. (1986). A new upper bound for Shellsort. *Journal of Algorithms*, **7**, 159–173. doi:10.1016/01966774(86)900015.

Singleton, R. C. (1969). Algorithm 347: an efficient algorithm for sorting with minimal storage. *Communications of the ACM*, **12**, 185–186. doi:10.1145/362875.362901.

See Also

'Comparison' for how character strings are collated.

`order` for sorting on or reordering multiple variables.

`is.unsorted`. `rank`.

Examples

```
require(stats)

x <- swiss$Education[1:25]
x; sort(x); sort(x, partial = c(10, 15))

## illustrate 'stable' sorting (of ties):
sort(c(10:3, 2:12), method = "shell", index.return = TRUE) # is stable
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9  8 10  7 11  6 12  5 13  4 14  3 15  2 16  1 17 18 19
sort(c(10:3, 2:12), method = "quick", index.return = TRUE) # is not
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9 10  8  7 11  6 12  5 13  4 14  3 15 16  2 17  1 18 19

x <- c(1:3, 3:5, 10)
is.unsorted(x)                  # FALSE: is sorted
is.unsorted(x, strictly = TRUE) # TRUE : is not (and cannot be)
                                # sorted strictly
## Not run:
## Small speed comparison simulation:
N <- 2000
Sim <- 20
rep <- 1000 # << adjust to your CPU
c1 <- c2 <- numeric(Sim)
for(is in seq_len(Sim)){
  x <- rnorm(N)
  c1[is] <- system.time(for(i in 1:rep) sort(x, method = "shell"))[1]
  c2[is] <- system.time(for(i in 1:rep) sort(x, method = "quick"))[1]
  stopifnot(sort(x, method = "shell") == sort(x, method = "quick"))
}
rbind(ShellSort = c1, QuickSort = c2)
cat("Speedup factor of quick sort():\n")
summary({qq <- c1 / c2; qq[is.finite(qq)]})

## A larger test
x <- rnorm(1e7)
system.time(x1 <- sort(x, method = "shell"))
system.time(x2 <- sort(x, method = "quick"))
system.time(x3 <- sort(x, method = "radix"))
stopifnot(identical(x1, x2))
stopifnot(identical(x1, x3))
```

```
## End(Not run)
```

---

sort_by *Sorting Vectors or Data Frames by Other Vectors*

Description

Generic function to sort an object in the order determined by one or more other objects, typically vectors. A method is defined for data frames to sort its rows (typically by one or more columns), and the default method handles vector-like objects.

Usage

```
sort_by(x, y, ...)

## Default S3 method:
sort_by(x, y, ...)

## S3 method for class 'data.frame'
sort_by(x, y, ...)
```

Arguments

x	An object to be sorted, typically a vector or data frame.
y	Variables to sort by. For the default method, this can be a vector, or more generally any object that has a xtfrm method. For the data.frame method, typically a formula specifying the variables to sort by. The formula can take the forms ~ g or ~ list(g) to sort by the variable g, or more generally the forms ~ g1 + ... + gk or ~ list(g1, ..., gk) to sort by the variables g1, ..., gk, using the later ones to resolve ties in the preceding ones. These variables are evaluated in the data frame x using the usual non-standard evaluation rules. If not a formula, y = g is equivalent to y = ~ g and y = list(g1, ..., gk) is equivalent to y = ~ list(g1, ..., gk). However, non-standard evaluation in x is not done in this case.
...	Additional arguments, typically passed on to order. These may include additional variables to sort by, as well as named arguments recognized by order.

Value

A sorted version of x. If x is a data frame, this means that the rows of x have been reordered to sort the variables specified in y.

See Also

sort, order.

Examples

```
mtcars$am
mtcars$mpg
with(mtcars, sort_by(mpg, am)) # group mpg by am

## data.frame method
sort_by(mtcars, runif(nrow(mtcars))) # random row permutation
sort_by(mtcars, list(mtcars$am, mtcars$mpg))

# formula interface
sort_by(mtcars, ~ am + mpg) |> subset(select = c(am, mpg))
sort_by.data.frame(mtcars, ~ list(am, -mpg)) |> subset(select = c(am, mpg))
```

---

source *Read R Code from a File, a Connection or Expressions*

Description

source causes R to accept its input from the named file or URL or connection or expressions directly. Input is read and parsed from that file until the end of the file is reached, then the parsed expressions are evaluated sequentially in the chosen environment.

withAutoprint(exprs) is a wrapper for source(exprs = exprs, ..) with different defaults. Its main purpose is to evaluate and auto-print expressions as if in a toplevel context, e.g, as in the R console.

Usage

```
source(file, local = FALSE, echo = verbose, print.eval = echo,
       exprs, spaced = use_file,
       verbose = getOption("verbose"),
       prompt.echo = getOption("prompt"),
       max.deparse.length = 150, width.cutoff = 60L,
       deparseCtrl = "showAttributes",
       chdir = FALSE,
       catch.aborts = FALSE,
       encoding = getOption("encoding"),
       continue.echo = getOption("continue"),
       skip.echo = 0, keep.source = getOption("keep.source"))

withAutoprint(exprs, evaluated = FALSE, local = parent.frame(),
              print. = TRUE, echo = TRUE, max.deparse.length = Inf,
              width.cutoff = max(20, getOption("width")),
              deparseCtrl = c("keepInteger", "showAttributes", "keepNA"),
              skip.echo = 0,
              ...)
```

Arguments

file	a connection or a character string giving the pathname of the file or URL to read from. The stdin() connection reads from the console when interactive.
local	TRUE, FALSE or an environment, determining where the parsed expressions are evaluated. FALSE (the default) corresponds to the user's workspace (the global environment) and TRUE to the environment from which source is called.
echo	logical; if TRUE, each expression is printed after parsing, before evaluation.
print.eval, print.	logical; if TRUE, the result of eval(i) is printed for each expression i; defaults to the value of echo.
exprs	for source() and withAutoprint(*, evaluated=TRUE): *instead* of specifying file, an expression, call, or list of call's, but *not* an unevaluated "expression". for withAutoprint() (with default evaluated=FALSE): one or more unevaluated "expressions".
evaluated	logical indicating that exprs is passed to source(exprs= *) and hence must be evaluated, i.e., a formal expression, call or list of calls.
spaced	logical indicating if newline (hence empty line) should be printed before each expression (when echo = TRUE).
verbose	if TRUE, more diagnostics (than just echo = TRUE) are printed during parsing and evaluation of input, including extra info for **each** expression.
prompt.echo	character; gives the prompt to be used if echo = TRUE.
max.deparse.length	integer; is used only if echo is TRUE and gives the maximal number of characters output for the deparse of a single expression.
width.cutoff	integer, passed to deparse() which is used (only) when there are no source references.
deparseCtrl	character vector, passed as control to deparse(), see also .deparseOpts. In R version <= 3.3.x, this was hardcoded to "showAttributes", which is the default currently; deparseCtrl = "all" may be preferable, when strict back compatibility is not of importance.
chdir	logical; if TRUE and file is a pathname, the R working directory is temporarily changed to the directory containing file for evaluating.
catch.aborts	logical indicating that "abort"ing errors should be caught.
encoding	character vector. The encoding(s) to be assumed when file is a character string: see file. A possible value is "unknown" when the encoding is guessed: see the 'Encodings' section.
continue.echo	character; gives the prompt to use on continuation lines if echo = TRUE.
skip.echo	integer; how many comment lines at the start of the file to skip if echo = TRUE.
keep.source	logical: should the source formatting be retained when echoing expressions, if possible?
...	(for withAutoprint():) further (non-file related) arguments to be passed to source(.).

Details

Note that running code via `source` differs in a few respects from entering it at the R command line. Since expressions are not executed at the top level, auto-printing is not done. So you will need to include explicit `print` calls for things you want to be printed (and remember that this includes plotting by **lattice**, FAQ Q7.22). Since the complete file is parsed before any of it is run, syntax errors result in none of the code being run. If an error occurs in running a syntactically correct script, anything assigned into the workspace by code that has been run will be kept (just as from the command line), but diagnostic information such as `traceback()` will contain additional calls to `withVisible`.

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic Mac OS) and map this to newline. The final line can be incomplete, that is missing the final end-of-line marker.

If `keep.source` is true (the default in interactive use), the source of functions is kept so they can be listed exactly as input.

Unlike input from a console, lines in the file or on a connection can contain an unlimited number of characters.

When `skip.echo > 0`, that many comment lines at the start of the file will not be echoed. This does not affect the execution of the code at all. If there are executable lines within the first `skip.echo` lines, echoing will start with the first of them.

If `echo` is true and a deparsed expression exceeds `max.deparse.length`, that many characters are output followed by `.... [TRUNCATED]`.

Encodings

By default the input is read and parsed in the current encoding of the R session. This is usually what is required, but occasionally re-encoding is needed, e.g. if a file from a UTF-8-using system is to be read on Windows (or *vice versa*).

The rest of this paragraph applies if `file` is an actual filename or URL (and not a connection). If `encoding = "unknown"`, an attempt is made to guess the encoding: the result of `localeToCharset()` is used as a guide. If `encoding` has two or more elements, they are tried in turn until the file/URL can be read without error in the trial encoding. If an actual `encoding` is specified (rather than the default or `"unknown"`) in a Latin-1 or UTF-8 locale then character strings in the result will be translated to the current encoding and marked as such (see `Encoding`).

If `file` is a connection, it is not possible to re-encode the input inside `source`, and so the `encoding` argument is just used to mark character strings in the parsed input in Latin-1 and UTF-8 locales: see `parse`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`demo` which uses `source`; `eval`, `parse` and `scan`; `options("keep.source")`.

`sys.source` which is a streamlined version to source a file into an environment.

'The R Language Definition' for a discussion of source directives.

Examples

```
someCond <- 7 > 6
## want an if-clause to behave "as top level" wrt auto-printing :
## (all should look "as if on top level", e.g. non-assignments should print:)
if(someCond) withAutoprint({
   x <- 1:12
   x-1
   (y <- (x-5)^2)
   z <- y
   z - 10
})

## If you want to source() a bunch of files, something like
## the following may be useful:
sourceDir <- function(path, trace = TRUE, ...) {
    op <- options(); on.exit(options(op)) # to reset after each
    for (nm in list.files(path, pattern = "[.][RrSsQq]$")) {
       if(trace) cat(nm,":")
       source(file.path(path, nm), ...)
       if(trace) cat("\n")
       options(op)
    }
}

suppressWarnings( rm(x,y) ) # remove 'x' or 'y' from global env
withAutoprint({ x <- 1:2; cat("x=",x, "\n"); y <- x^2 })
## x and y now exist:
stopifnot(identical(x, 1:2), identical(y, x^2))

withAutoprint({ formals(sourceDir); body(sourceDir) },
              max.deparse.length = 20, verbose = TRUE)

## Continuing after (catchable) errors:
tc <- textConnection('1:3
 2 + "3"
 cat(" .. in spite of error: happily continuing! ..\n")
 6*7')
r <- source(tc, catch.aborts = TRUE)
## Error in 2 + "3" ....
## .. in spite of error: happily continuing! ..
stopifnot(identical(r, list(value = 42, visible=TRUE)))
```

`Special` *Special Functions of Mathematics*

Description

Special mathematical functions related to the beta and gamma functions.

Usage

```
beta(a, b)
lbeta(a, b)

gamma(x)
lgamma(x)
psigamma(x, deriv = 0)
digamma(x)
trigamma(x)

choose(n, k)
lchoose(n, k)
factorial(x)
lfactorial(x)
```

Arguments

a, b	non-negative numeric vectors.
x, n	numeric vectors.
k, deriv	integer vectors.

Details

The functions `beta` and `lbeta` return the beta function and the natural logarithm of the beta function,

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The formal definition is

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt$$

(Abramowitz and Stegun section 6.2.1, page 258). Note that it is only defined in R for non-negative `a` and `b`, and is infinite if either is zero.

The functions `gamma` and `lgamma` return the gamma function $\Gamma(x)$ and the natural logarithm of *the absolute value of* the gamma function. The gamma function is defined by (Abramowitz and Stegun section 6.1.1, page 255)

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$$

for all real `x` except zero and negative integers (when `NaN` is returned). There will be a warning on possible loss of precision for values which are too close (within about 10^{-8}) to a negative integer less than '`-10`'.

`factorial(x)` ($x!$ for non-negative integer `x`) is defined to be `gamma(x+1)` and `lfactorial` to be `lgamma(x+1)`.

The functions `digamma` and `trigamma` return the first and second derivatives of the logarithm of the gamma function. `psigamma(x, deriv)` (`deriv >= 0`) computes the `deriv`-th derivative of $\psi(x)$.

$$\texttt{digamma(x)} = \psi(x) = \frac{d}{dx}\ln\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

ψ and its derivatives, the `psigamma()` functions, are often called the 'polygamma' functions, e.g. in Abramowitz and Stegun (section 6.4.1, page 260); and higher derivatives (`deriv = 2:4`) have occasionally been called 'tetragamma', 'pentagamma', and 'hexagamma'.

The functions `choose` and `lchoose` return binomial coefficients and the logarithms of their absolute values. Note that `choose(n, k)` is defined for all real numbers n and integer k. For $k \geq 1$ it is defined as $n(n-1)\cdots(n-k+1)/k!$, as 1 for $k = 0$ and as 0 for negative k. Non-integer values of `k` are rounded to an integer, with a warning.
`choose(*, k)` uses direct arithmetic (instead of `[l]gamma` calls) for small `k`, for speed and accuracy reasons. Note the function `combn` (package **utils**) for enumeration of all possible combinations.

The `gamma`, `lgamma`, `digamma` and `trigamma` functions are internal generic primitive functions: methods can be defined for them individually or via the `Math` group generic.

Source

`gamma`, `lgamma`, `beta` and `lbeta` are based on C translations of Fortran subroutines by W. Fullerton of Los Alamos Scientific Laboratory (now available as part of SLATEC).

`digamma`, `trigamma` and `psigamma` for `x >= 0` are based on

Amos, D. E. (1983). A portable Fortran subroutine for derivatives of the psi function, Algorithm 610, *ACM Transactions on Mathematical Software* **9(4)**, 494–502.

For, `x < 0` and `deriv <= 5`, the reflection formula (6.4.7) of Abramowitz and Stegun is used.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (For `gamma` and `lgamma`.)

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.
Chapter 6: Gamma and Related Functions.

See Also

`Arithmetic` for simple, `sqrt` for miscellaneous mathematical functions and `Bessel` for the real Bessel functions.

For the incomplete gamma function see `pgamma`.

Examples

```
require(graphics)

choose(5, 2)
for (n in 0:10) print(choose(n, k = 0:n))

factorial(100)
lfactorial(10000)

## gamma has 1st order poles at 0, -1, -2, ...
## this will generate loss of precision warnings, so turn off
```

```
op <- options("warn")
options(warn = -1)
x <- sort(c(seq(-3, 4, length.out = 201), outer(0:-3, (-1:1)*1e-6, `+`)))
plot(x, gamma(x), ylim = c(-20,20), col = "red", type = "l", lwd = 2,
     main = expression(Gamma(x)))
abline(h = 0, v = -3:0, lty = 3, col = "midnightblue")
options(op)

x <- seq(0.1, 4, length.out = 201); dx <- diff(x)[1]
par(mfrow = c(2, 3))
for (ch in c("", "l","di","tri","tetra","penta")) {
  is.deriv <- nchar(ch) >= 2
  nm <- paste0(ch, "gamma")
  if (is.deriv) {
    dy <- diff(y) / dx # finite difference
    der <- which(ch == c("di","tri","tetra","penta")) - 1
    nm2 <- paste0("psigamma(*, deriv = ", der,")")
    nm  <- if(der >= 2) nm2 else paste(nm, nm2, sep = " ==\n")
    y <- psigamma(x, deriv = der)
  } else {
    y <- get(nm)(x)
  }
  plot(x, y, type = "l", main = nm, col = "red")
  abline(h = 0, col = "lightgray")
  if (is.deriv) lines(x[-1], dy, col = "blue", lty = 2)
}
par(mfrow = c(1, 1))

## "Extended" Pascal triangle:
fN <- function(n) formatC(n, width=2)
for (n in -4:10) {
    cat(fN(n),":", fN(choose(n, k = -2:max(3, n+2))))
    cat("\n")
}

## R code version of choose()  [simplistic; warning for k < 0]:
mychoose <- function(r, k)
    ifelse(k <= 0, (k == 0),
           sapply(k, function(k) prod(r:(r-k+1))) / factorial(k))
k <- -1:6
cbind(k = k, choose(1/2, k), mychoose(1/2, k))

## Binomial theorem for n = 1/2 ;
## sqrt(1+x) = (1+x)^(1/2) = sum_{k=0}^Inf  choose(1/2, k) * x^k :
k <- 0:10 # 10 is sufficient for ~ 9 digit precision:
sqrt(1.25)
sum(choose(1/2, k)* .25^k)
```

---

split *Divide into Groups and Reassemble*

Description

split divides the data in the vector x into the groups defined by f. The replacement forms replace values corresponding to such a division. unsplit reverses the effect of split.

Usage

```
split(x, f, drop = FALSE, ...)
## Default S3 method:
split(x, f, drop = FALSE, sep = ".", lex.order = FALSE, ...)

split(x, f, drop = FALSE, ...) <- value
unsplit(value, f, drop = FALSE)
```

Arguments

x	vector or data frame containing values to be divided into groups.
f	a 'factor' in the sense that as.factor(f) defines the grouping, or a list of such factors in which case their interaction is used for the grouping. If x is a data frame, f can also be a formula of the form ~ g to split by the variable g, or more generally of the form ~ g1 + ... + gk to split by the interaction of the variables g1, ..., gk, where these variables are evaluated in the data frame x using the usual non-standard evaluation rules.
drop	logical indicating if levels that do not occur should be dropped (if f is a factor or a list).
value	a list of vectors or data frames compatible with a splitting of x. Recycling applies if the lengths do not match.
sep	character string, passed to interaction in the case where f is a list.
lex.order	logical, passed to interaction when f is a list.
...	further potential arguments passed to methods.

Details

split and split<- are generic functions with default and data.frame methods. The data frame method can also be used to split a matrix into a list of matrices, and the replacement form likewise, provided they are invoked explicitly.

unsplit works with lists of vectors or data frames (assumed to have compatible structure, as if created by split). It puts elements or rows back in the positions given by f. In the data frame case, row names are obtained by unsplitting the row name vectors from the elements of value.

f is recycled as necessary and if the length of x is not a multiple of the length of f a warning is printed.

Any missing values in f are dropped together with the corresponding values of x.

The default method calls interaction when f is a list. If the levels of the factors contain '.' the factors may not be split as expected, unless sep is set to string not present in the factor levels.

Value

The value returned from `split` is a list of vectors containing the values for the groups. The components of the list are named by the levels of `f` (after converting to a factor, or if already a factor and `drop = TRUE`, dropping unused levels).

The replacement forms return their right hand side. `unsplit` returns a vector or data frame for which `split(x, f)` equals `value`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`cut` to categorize numeric values.

`strsplit` to split strings.

Examples

```
require(stats); require(graphics)
n <- 10; nn <- 100
g <- factor(round(n * runif(n * nn)))
x <- rnorm(n * nn) + sqrt(as.numeric(g))
xg <- split(x, g)
boxplot(xg, col = "lavender", notch = TRUE, varwidth = TRUE)
sapply(xg, length)
sapply(xg, mean)

### Calculate 'z-scores' by group (standardize to mean zero, variance one)
z <- unsplit(lapply(split(x, g), scale), g)

# or

zz <- x
split(zz, g) <- lapply(split(x, g), scale)

# and check that the within-group std dev is indeed one
tapply(z, g, sd)
tapply(zz, g, sd)


### data frame variation

## Notice that assignment form is not used since a variable is being added

g <- airquality$Month
l <- split(airquality, g)

## Alternative using a formula
identical(l, split(airquality, ~ Month))
```

```
l <- lapply(l, transform, Oz.Z = scale(Ozone))
aq2 <- unsplit(l, g)
head(aq2)
with(aq2, tapply(Oz.Z,  Month, sd, na.rm = TRUE))


### Split a matrix into a list by columns
ma <- cbind(x = 1:10, y = (-4:5)^2)
split(ma, col(ma))

split(1:10, 1:2)
```

---

sprintf *Use C-style String Formatting Commands*

Description

A wrapper for the C function `sprintf`, that returns a character vector containing a formatted combination of text and variable values.

Usage

```
sprintf(fmt, ...)
gettextf(fmt, ..., domain = NULL, trim = TRUE)
```

Arguments

`fmt`	a character vector of format strings, each of up to 8192 bytes.
`...`	values to be passed into `fmt`. Only logical, integer, real and character vectors are supported, but some coercion will be done: see the 'Details' section. Up to 100.
`trim`, `domain`	see `gettext`.

Details

`sprintf` is a wrapper for the system `sprintf` C-library function. Attempts are made to check that the mode of the values passed match the format supplied, and R's special values (`NA`, `Inf`, `-Inf` and `NaN`) are handled correctly.

`gettextf` is a convenience function which provides C-style string formatting with possible translation of the format string.

The arguments (including `fmt`) are recycled if possible a whole number of times to the length of the longest, and then the formatting is done in parallel. Zero-length arguments are allowed and will give a zero-length result. All arguments are evaluated even if unused, and hence some types (e.g., `"symbol"` or `"language"`, see `typeof`) are not allowed. Arguments unused by `fmt` result in a warning. (The format `%.0s` can be used to "skip" an argument.)

The following is abstracted from Kernighan and Ritchie (1988): however the actual implementation will follow the C99 standard and fine details (especially the behaviour under user error) may depend on the platform. References to numbered arguments come from POSIX.

The string `fmt` contains normal characters, which are passed through to the output string, and also conversion specifications which operate on the arguments provided through `...`. The allowed conversion specifications start with a `%` and end with one of the letters in the set `aAdifeEgGosxX%`. These letters denote the following types:

`d`, `i`, `o`, `x`, `X` Integer value, `o` being octal, `x` and `X` being hexadecimal (using the same case for `a-f` as the code). Numeric variables with exactly integer values will be coerced to integer. Formats `d` and `i` can also be used for logical variables, which will be converted to `0`, `1` or `NA`.

`f` Double precision value, in "**f**ixed point" decimal notation of the form '`"[-]mmm.ddd"`'. The number of decimal places ("d") is specified by the precision: the default is 6; a precision of 0 suppresses the decimal point. Non-finite values are converted to `NA`, `NaN` or (perhaps a sign followed by) `Inf`.

`e`, `E` Double precision value, in "**e**xponential" decimal notation of the form `[-]m.ddde[+-]xx` or `[-]m.dddE[+-]xx`.

`g`, `G` Double precision value, in `%e` or `%E` format if the exponent is less than -4 or greater than or equal to the precision, and `%f` format otherwise. (The precision (default 6) specifies the number of *significant* digits here, whereas in `%f`, `%e`, it is the number of digits after the decimal point.)

`a`, `A` Double precision value, in binary notation of the form `[-]0xh.hhhp[+-]d`. This is a binary fraction expressed in hex multiplied by a (decimal) power of 2. The number of hex digits after the decimal point is specified by the precision: the default is enough digits to represent exactly the internal binary representation. Non-finite values are converted to `NA`, `NaN` or (perhaps a sign followed by) `Inf`. Format `%a` uses lower-case for `x`, `p` and the hex values: format `%A` uses upper-case.

This should be supported on all platforms as it is a feature of C99. The format is not uniquely defined: although it would be possible to make the leading `h` always zero or one, this is not always done. Most systems will suppress trailing zeros, but a few do not. On a well-written platform, for normal numbers there will be a leading one before the decimal point plus (by default) 13 hexadecimal digits, hence 53 bits. The treatment of denormalized (aka 'subnormal') numbers is very platform-dependent.

`s` Character string. Character `NA`s are converted to `"NA"`.

`%` Literal `%` (none of the extra formatting characters given below are permitted in this case).

Conversion by `as.character` is used for non-character arguments with `s` and by `as.double` for non-double arguments with `f`, `e`, `E`, `g`, `G`. NB: the length is determined before conversion, so do not rely on the internal coercion if this would change the length. The coercion is done only once, so if `length(fmt) > 1` then all elements must expect the same types of arguments.

In addition, between the initial `%` and the terminating conversion character there may be, in any order:

`m.n` Two numbers separated by a period, denoting the field width (`m`) and the precision (`n`).

`-` Left adjustment of converted argument in its field.

`+` Always print number with sign: by default only negative numbers are printed with a sign.

a space Prefix a space if the first character is not a sign.

`0` For numbers, pad to the field width with leading zeros. For characters, this zero-pads on some platforms and is ignored on others.

`#` specifies "alternate output" for numbers, its action depending on the type: For `x` or `X`, `0x` or `0X` will be prefixed to a non-zero result. For `e`, `E`, `f`, `g` and `G`, the output will always have a decimal point; for `g` and `G`, trailing zeros will not be removed.

Further, immediately after `%` may come `1$` to `99$` to refer to a numbered argument: this allows arguments to be referenced out of order and is mainly intended for translators of error messages. If this is done it is best if all formats are numbered: if not the unnumbered ones process the arguments in order. See the examples. This notation allows arguments to be used more than once, in which case they must be used as the same type (integer, double or character).

A field width or precision (but not both) may be indicated by an asterisk `*`: in this case an argument specifies the desired number. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is treated as if the precision were omitted. The argument should be integer, but a double argument will be coerced to integer.

There is a limit of 8192 bytes on elements of `fmt`, and on strings included from a single *%letter* conversion specification.

Field widths and precisions of `%s` conversions are interpreted as bytes, not characters, as described in the C standard.

The C doubles used for R numerical vectors have signed zeros, which `sprintf` may output as `-0`, `-0.000` ....

Value

A character vector of length that of the longest input. If any element of `fmt` or any character argument is declared as UTF-8, the element of the result will be in UTF-8 and have the encoding declared as UTF-8. Otherwise it will be in the current locale's encoding.

Warning

The format string is passed down the OS's `sprintf` function, and incorrect formats can cause the latter to crash the R process . R does perform sanity checks on the format, but not all possible user errors on all platforms have been tested, and some might be terminal.

The behaviour on inputs not documented here is 'undefined', which means it is allowed to differ by platform.

Author(s)

Original code by Jonathan Rougier.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition, Prentice Hall. Describes the format options in table B-1 in the Appendix.

The C Standards, especially ISO/IEC 9899:1999 for 'C99'. Links can be found at https://developer.r-project.org/Portability.html.

https://pubs.opengroup.org/onlinepubs/9699919799/functions/snprintf.html for POSIX extensions such as numbered arguments.

man sprintf on a Unix-alike system.

See Also

formatC for a way of formatting vectors of numbers in a similar fashion.

paste for another way of creating a vector combining text and values.

gettext for the mechanisms for the automated translation of text.

Examples

```
## be careful with the format: most things in R are floats
## only integer-valued reals get coerced to integer.

sprintf("%s is %f feet tall\n", "Sven", 7.1)      # OK
try(sprintf("%s is %i feet tall\n", "Sven", 7.1)) # not OK
    sprintf("%s is %i feet tall\n", "Sven", 7  )  # OK

## use a literal % :

sprintf("%.0f%% said yes (out of a sample of size %.0f)", 66.666, 3)

## various formats of pi :

sprintf("%f", pi)
sprintf("%.3f", pi)
sprintf("%1.0f", pi)
sprintf("%5.1f", pi)
sprintf("%05.1f", pi)
sprintf("%+f", pi)
sprintf("% f", pi)
sprintf("%-10f", pi) # left justified
sprintf("%e", pi)
sprintf("%E", pi)
sprintf("%g", pi)
sprintf("%g",   1e6 * pi) # -> exponential
sprintf("%.9g", 1e6 * pi) # -> "fixed"
sprintf("%G", 1e-6 * pi)

## no truncation:
sprintf("%1.f", 101)

## re-use one argument three times, show difference between %x and %X
xx <- sprintf("%1$d %1$x %1$X", 0:15)
xx <- matrix(xx, dimnames = list(rep("", 16), "%d%x%X"))
noquote(format(xx, justify = "right"))

## More sophisticated:

sprintf("min 10-char string '%10s'",
```

```
        c("a", "ABC", "and an even longer one"))

n <- 1:18
sprintf(paste0("e with %2d digits = %.", n, "g"), n, exp(1))

## Platform-dependent bad example: may pad with spaces or zeroes
sprintf("%09s", month.name)

## Using arguments out of order
sprintf("second %2$1.0f, first %1$5.2f, third %3$1.0f", pi, 2, 3)

## Using asterisk for width or precision
sprintf("precision %.*f, width '%*.3f'", 3, pi, 8, pi)

## Asterisk and argument re-use, 'e' example reiterated:
sprintf("e with %1$2d digits = %2$.*1$g", n, exp(1))

## re-cycle arguments
sprintf("%s %d", "test", 1:3)

## binary output showing rounding/representation errors
x <- seq(0, 1.0, 0.1); y <- c(0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1)
cbind(x, sprintf("%a", x), sprintf("%a", y))
```

---

sQuote *Quote Text*

Description

Single or double quote text by combining with appropriate single or double left and right quotation marks.

Usage

```
sQuote(x, q = getOption("useFancyQuotes"))
dQuote(x, q = getOption("useFancyQuotes"))
```

Arguments

x	an R object, to be coerced to a character vector.
q	the kind of quotes to be used, see 'Details'.

Details

The purpose of the functions is to provide a simple means of markup for quoting text to be used in the R output, e.g., in warnings or error messages.

The choice of the appropriate quotation marks depends on both the locale and the available character sets. Older Unix/X11 fonts displayed the grave accent (ASCII code 0x60) and the apostrophe (0x27) in a way that they could also be used as matching open and close single quotation marks. Using

modern fonts, or non-Unix systems, these characters no longer produce matching glyphs. Unicode provides left and right single quotation mark characters (U+2018 and U+2019); if Unicode markup cannot be assumed to be available, it seems good practice to use the apostrophe as a non-directional single quotation mark.

Similarly, Unicode has left and right double quotation mark characters (U+201C and U+201D); if only ASCII's typewriter characteristics can be employed, than the ASCII quotation mark (0x22) should be used as both the left and right double quotation mark.

Some other locales also have the directional quotation marks, notably on Windows. TeX uses grave and apostrophe for the directional single quotation marks, and doubled grave and doubled apostrophe for the directional double quotation marks.

What rendering is used depends on `q` which by default depends on the `options` setting for `useFancyQuotes`. If this is `FALSE` then the undirectional ASCII quotation style is used. If this is `TRUE` (the default), Unicode directional quotes are used are used where available (currently, UTF-8 locales on Unix-alikes and all Windows locales except `C`): if set to `"UTF-8"` UTF-8 markup is used (whatever the current locale). If set to `"TeX"`, TeX-style markup is used. Finally, if this is set to a character vector of length four, the first two entries are used for beginning and ending single quotes and the second two for beginning and ending double quotes: this can be used to implement non-English quoting conventions such as the use of guillemets.

Where fancy quotes are used, you should be aware that they may not be rendered correctly as not all fonts include the requisite glyphs: for example some have directional single quotes but not directional double quotes.

Value

A character vector of the same length as `x` (after any coercion) in the current locale's encoding.

References

Markus Kuhn, "ASCII and Unicode quotation marks". https://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html

See Also

Quotes for quoting R code.

`shQuote` for quoting OS commands.

Examples

```
op <- options("useFancyQuotes")
paste("argument", sQuote("x"), "must be non-zero")
options(useFancyQuotes = FALSE)
cat("\ndistinguish plain", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
options(useFancyQuotes = TRUE)
cat("\ndistinguish fancy", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
options(useFancyQuotes = "TeX")
cat("\ndistinguish TeX", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
```

```
if(l10n_info()$`Latin-1`) {
    options(useFancyQuotes = c("\xab", "\xbb", "\xbf", "?"))
    cat("\n", sQuote("guillemet"), "and",
        dQuote("Spanish question"), "styles\n")
} else if(l10n_info()$`UTF-8`) {
    options(useFancyQuotes = c("\xc2\xab", "\xc2\xbb", "\xc2\xbf", "?"))
    cat("\n", sQuote("guillemet"), "and",
        dQuote("Spanish question"), "styles\n")
}
options(op)
```

`srcfile` *References to Source Files and Code*

Description

These functions are for working with source files and more generally with "source references" (`"srcref"`), i.e., references to source code. The resulting data is used for printing and source level debugging, and is typically available in interactive R sessions, namely when `options(keep.source = TRUE)`.

Usage

```
srcfile(filename, encoding = getOption("encoding"), Enc = "unknown")
srcfilecopy(filename, lines, timestamp = Sys.time(), isFile = FALSE)
srcfilealias(filename, srcfile)
getSrcLines(srcfile, first, last)
srcref(srcfile, lloc)
## S3 method for class 'srcfile'
print(x, ...)
## S3 method for class 'srcfile'
summary(object, ...)
## S3 method for class 'srcfile'
open(con, line, ...)
## S3 method for class 'srcfile'
close(con, ...)
## S3 method for class 'srcref'
print(x, useSource = TRUE, ...)
## S3 method for class 'srcref'
summary(object, useSource = FALSE, ...)
## S3 method for class 'srcref'
as.character(x, useSource = TRUE, to = x, ...)
.isOpen(srcfile)
```

Arguments

`filename`	The name of a file.
`encoding`	The character encoding to assume for the file.

Enc	The encoding with which to make strings: see the encoding argument of parse.
lines	A character vector of source lines. Other R objects will be coerced to character.
timestamp	The timestamp to use on a copy of a file.
isFile	Is this srcfilecopy known to come from a file system file?
srcfile	A srcfile object.
first, last, line	Line numbers.
lloc	A vector of four, six or eight values giving a source location; see 'Details'.
x, object, con	An object of the appropriate class.
useSource	Whether to read the srcfile to obtain the text of a srcref.
to	An optional second srcref object to mark the end of the character range.
...	Additional arguments to the methods; these will be ignored.

Details

These functions and classes handle source code references.

The srcfile function produces an object of class srcfile, which contains the name and directory of a source code file, along with its timestamp, for use in source level debugging (not yet implemented) and source echoing. The encoding of the file is saved; see file for a discussion of encodings, and iconvlist for a list of allowable encodings on your platform.

The srcfilecopy function produces an object of the descendant class srcfilecopy, which saves the source lines in a character vector. It copies the value of the isFile argument, to help debuggers identify whether this text comes from a real file in the file system.

The srcfilealias function produces an object of the descendant class srcfilealias, which gives an alternate name to another srcfile. This is produced by the parser when a #line directive is used.

The getSrcLines function reads the specified lines from srcfile.

The srcref function produces an object of class srcref, which describes a range of characters in a srcfile. The lloc value gives the following values:

```
c(first_line, first_byte, last_line, last_byte, first_column,
  last_column, first_parsed, last_parsed)
```

Bytes (elements 2, 4) and columns (elements 5, 6) may be different due to multibyte characters. If only four values are given, the columns and bytes are assumed to match. Lines (elements 1, 3) and parsed lines (elements 7, 8) may differ if a #line directive is used in code: the former will respect the directive, the latter will just count lines. If only 4 or 6 elements are given, the parsed lines will be assumed to match the lines.

Methods are defined for print, summary, open, and close for classes srcfile and srcfilecopy. The open method opens its internal file connection at a particular line; if it was already open, it will be repositioned to that line.

Methods are defined for print, summary and as.character for class srcref. The as.character method will read the associated source file to obtain the text corresponding to the reference. If the to argument is given, it should be a second srcref that follows the first, in the same file; they will

be treated as one reference to the whole range. The exact behaviour depends on the class of the source file. If the source file inherits from class `srcfilecopy`, the lines are taken from the saved copy using the "parsed" line counts. If not, an attempt is made to read the file, and the original line numbers of the `srcref` record (i.e., elements 1 and 3) are used. If an error occurs (e.g., the file no longer exists), text like '`<srcref: "file" chars 1:1 to 2:10>`' will be returned instead, indicating the `line:column` ranges of the first and last character. The `summary` method defaults to this type of display.

Lists of `srcref` objects may be attached to expressions as the `"srcref"` attribute. (The list of `srcref` objects should be the same length as the expression.) By default, expressions are printed by `print.default` using the associated `srcref`. To see deparsed code instead, call `print` with argument `useSource = FALSE`. If a `srcref` object is printed with `useSource = FALSE`, the '`<srcref: ....>`' record will be printed.

`.isOpen` is intended for internal use: it checks whether the connection associated with a `srcfile` object is open.

Value

`srcfile` returns a `srcfile` object.

`srcfilecopy` returns a `srcfilecopy` object.

`getSrcLines` returns a character vector of source code lines.

`srcref` returns a `srcref` object.

Author(s)

Duncan Murdoch

See Also

`getSrcFilename` for extracting information from a source reference, or `removeSource` to remove it from a (non-primitive) function (aka 'closure').

Examples

```
src <- srcfile(system.file("DESCRIPTION", package = "base"))
summary(src)
getSrcLines(src, 1, 4)
ref <- srcref(src, c(1, 1, 2, 1000))
ref
print(ref, useSource = FALSE)
```

`StackOverflows` *Stack Overflow Errors*

Description

Errors signaled by R when stacks used in evaluation overflow.

Details

R uses several stacks in evaluating expressions: the C stack, the pointer protection stack, and the node stack used by the byte code engine. In addition, the number of nested R expressions currently under evaluation is limited by the value set as `options("expressions")`. Overflowing these stacks or limits signals an error that inherits from classes `stackOverflowError`, `error`, and `condition`.

The specific classes signaled are:

- `CStackOverflowError`: Signaled when the C stack overflows. The `usage` field of the error object contains the current stack usage.
- `protectStackOverflowError`: Signaled when the pointer protection stack overflows.
- `nodeStackOverflowError`: Signaled when the node stack used by the byte code engine overflows.
- `expressionStackOverflowError`: Signaled when the the evaluation depth, the number of nested R expressions currently under evaluation, exceeds the limit set by `options("expressions")`

Stack overflow errors can be caught and handled by exiting handlers established with `tryCatch()` Calling handlers established by `withCallingHandlers()` may fail since there may not be enough stack space to run the handler. In this case the next available exiting handler will be run, or error handling will fall back to the default handler. Default handlers set by `tryCatch("error")` may also fail to run in a stack overflow situation.

See Also

`Cstack_info` for information on the environment and the evaluation depth limit.

`Memory` and `options` for information on the protection stack.

`standardGeneric` *Formal Method System – Dispatching S4 Methods*

Description

The function `standardGeneric` initiates dispatch of S4 methods: see the references and the documentation of the **methods** package. Usually, calls to this function are generated automatically and not explicitly by the programmer.

Usage

```
standardGeneric(f, fdef)
```

Arguments

f	The name of the generic.
fdef	The generic function definition. Never passed when defining a new generic.

Details

`standardGeneric` dispatches the method defined for a generic function named `f`, using the actual arguments in the frame from which it is called.

The argument `fdef` is inserted (automatically) when dispatching methods for a primitive function. If present, it must always be the function definition for the corresponding generic. Don't insert this argument by hand, as there is no validity checking and miss-specifying the function definition will cause certain failure.

For more, use the **methods** package, and see the documentation in `GenericFunctions`.

Author(s)

John Chambers

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

---

startsWith *Does String Start or End With Another String?*

Description

Determines if entries of `x` start or end with string (entries of) `prefix` or `suffix` respectively, where strings are recycled to common lengths.

Usage

```
startsWith(x, prefix)
  endsWith(x, suffix)
```

Arguments

x	`character` vector whose "starts" or "ends" are considered.
prefix, suffix	`character` vector, typically of length one, i.e., a string.

Details

`startsWith()` is equivalent to but much faster than

```
substring(x, 1, nchar(prefix)) == prefix
```

or also

```
grepl("^<prefix>", x)
```

where `prefix` is not to contain special regular expression characters (and for `grepl`, `x` does not contain missing values, see below).

The code has an optimized branch for the most common usage in which `prefix` or `suffix` is of length one, and is further optimized in a UTF-8 or 8-byte locale if that is an ASCII string.

Value

A `logical` vector, of "common length" of `x` and `prefix` (or `suffix`), i.e., of the longer of the two lengths unless one of them is zero when the result is also of zero length. A shorter input is recycled to the output length.

See Also

`grepl`, `substring`; the partial string matching functions `charmatch` and `pmatch` solve a different task.

Examples

```
startsWith(search(), "package:") # typically at least two FALSE, nowadays often three

x1 <- c("Foobar", "bla bla", "something", "another", "blu", "brown",
        "blau blüht der Enzian")# non-ASCII
x2 <- cbind(
      startsWith(x1, "b"),
      startsWith(x1, "bl"),
      startsWith(x1, "bla"),
        endsWith(x1, "n"),
        endsWith(x1, "an"))
rownames(x2) <- x1; colnames(x2) <- c("b", "b1", "bla", "n", "an")
x2

## Non-equivalence in case of missing values in 'x', see Details:
x <- c("all", "but", NA_character_)
cbind(startsWith(x, "a"),
      substring(x, 1L, 1L) == "a",
      grepl("^a", x))
```

Startup *Initialization at Start of an R Session*

Description

In R, the startup mechanism is as follows.

Unless '--no-environ' was given on the command line, R searches for site and user files to process for setting environment variables. The name of the site file is the one pointed to by the environment variable R_ENVIRON; if this is unset, '*R_HOME*/etc/Renviron.site' is used (if it exists, which it does not in a 'factory-fresh' installation). The name of the user file can be specified by the R_ENVIRON_USER environment variable; if this is unset, the files searched for are '.Renviron' in the current or in the user's home directory (in that order). See 'Details' for how the files are read.

Then R searches for the site-wide startup profile file of R code unless the command line option '--no-site-file' was given. The path of this file is taken from the value of the R_PROFILE environment variable (after tilde expansion). If this variable is unset, the default is '*R_HOME*/etc/Rprofile.site', which is used if it exists (which it does not in a 'factory-fresh' installation). This code is sourced into the workspace (global environment). Users need to be careful not to unintentionally create objects in the workspace, and it is normally advisable to use `local` if code needs to be executed: see the examples. `.Library.site` may be assigned to and the assignment will effectively modify the value of the variable in the base namespace where `.libPaths()` finds it. One may also assign to `.First` and `.Last`, but assigning to other variables in the execution environment is not recommended and does not work in some older versions of R.

Then, unless '--no-init-file' was given, R searches for a user profile, a file of R code. The path of this file can be specified by the R_PROFILE_USER environment variable (and tilde expansion will be performed). If this is unset, a file called '.Rprofile' is searched for in the current directory or in the user's home directory (in that order). The user profile file is sourced into the workspace.

Note that when the site and user profile files are sourced only the **base** package is loaded, so objects in other packages need to be referred to by e.g. `utils::dump.frames` or after explicitly loading the package concerned.

R then loads a saved image of the user workspace from '.RData' in the current directory if there is one (unless '--no-restore-data' or '--no-restore' was specified on the command line).

Next, if a function `.First` is found on the search path, it is executed as `.First()`. Finally, function `.First.sys()` in the **base** package is run. This calls `require` to attach the default packages specified by `options("defaultPackages")`. If the **methods** package is included, this will have been attached earlier (by function `.OptRequireMethods()`) so that namespace initializations such as those from the user workspace will proceed correctly.

A function `.First` (and `.Last`) can be defined in appropriate '.Rprofile' or 'Rprofile.site' files or have been saved in '.RData'. If you want a different set of packages than the default ones when you start, insert a call to `options` in the '.Rprofile' or 'Rprofile.site' file. For example, `options(defaultPackages = character())` will attach no extra packages on startup (only the **base** package) (or set R_DEFAULT_PACKAGES=NULL as an environment variable before running R). Using `options(defaultPackages = "")` or R_DEFAULT_PACKAGES="" enforces the R *system* default.

On front-ends which support it, the commands history is read from the file specified by the environment variable `R_HISTFILE` (default '`.Rhistory`' in the current directory) unless '`--no-restore-history`' or '`--no-restore`' was specified.

The command-line option '`--vanilla`' implies '`--no-site-file`', '`--no-init-file`', '`--no-environ`' and (except for `R CMD`) '`--no-restore`'

Details

Note that there are two sorts of files used in startup: *environment files* which contain lists of environment variables to be set, and *profile files* which contain R code.

Lines in a site or user environment file should be either comment lines starting with #, or lines of the form *name*=*value*. The latter sets the environmental variable *name* to *value*, overriding an existing value. If *value* contains an expression of the form `${foo-bar}`, the value is that of the environmental variable `foo` if that is set, otherwise `bar`. For `${foo:-bar}`, the value is that of `foo` if that is set to a non-empty value, otherwise `bar`. (If it is of the form `${foo}`, the default is `""`.) This construction can be nested, so `bar` can be of the same form (as in `${foo-${bar-blah}}`). Note that the braces are essential: for example `$HOME` will not be interpreted.

Leading and trailing white space in *value* are stripped. *value* is then processed in a similar way to a Unix shell: in particular (single or double) quotes not preceded by backslash are removed and backslashes are removed except inside such quotes.

For readability and future compatibility it is recommended to only use constructs that have the same behavior as in a Unix shell. Hence, expansions of variables should be in double quotes (e.g. `"${HOME}"`, in case they may contain a backslash) and literals including a backslash should be in single quotes. If a variable value may end in a backslash, such as `PATH` on Windows, it may be necessary to protect the following quote from it, e.g. `"${PATH}/"`. It is recommended to use forward slashes instead of backslashes. It is ok to mix text in single and double quotes, see examples below.

On systems with sub-architectures (mainly Windows), the files '`Renviron.site`' and '`Rprofile.site`' are looked for first in architecture-specific directories, e.g. '*R_HOME*`/etc/i386/Renviron.site`'. And e.g. '`.Renviron.i386`' will be used in preference to '`.Renviron`'.

There is a 100,000 byte limit on the length of a line (after expansions) in environment files.

Note

It is not intended that there be interaction with the user during startup code. Attempting to do so can crash the R process.

On Unix versions of R there is also a file '*R_HOME*`/etc/Renviron`' which is read very early in the start-up processing. It contains environment variables set by R in the configure process. Values in that file can be overridden in site or user environment files: do not change '*R_HOME*`/etc/Renviron`' itself. Note that this is distinct from '*R_HOME*`/etc/Renviron.site`'.

Command-line options may well not apply to alternative front-ends: they do not apply to `R.app` on macOS.

`R CMD check` and `R CMD build` do not always read the standard startup files, but they do always read specific '`Renviron`' files. The location of these can be controlled by the environment variables `R_CHECK_ENVIRON` and `R_BUILD_ENVIRON`. If these are set their value is used as the path for

the 'Renviron' file; otherwise, files '~/.R/check.Renviron' or '~/.R/build.Renviron' or sub-architecture-specific versions are employed.

If you want '~/.Renviron' or '~/.Rprofile' to be ignored by child R processes (such as those run by R CMD check and R CMD build), set the appropriate environment variable R_ENVIRON_USER or R_PROFILE_USER to (if possible, which it is not on Windows) "" or to the name of a non-existent file.

See Also

For the definition of the 'home' directory on Windows see the 'rw-FAQ' Q2.14. It can be found from a running R by Sys.getenv("R_USER").

.Last for final actions at the close of an R session. commandArgs for accessing the command line arguments.

There are examples of using startup files to set defaults for graphics devices in the help for X11 and quartz.

An Introduction to R for more command-line options: those affecting memory management are covered in the help file for Memory.

readRenviron to read '.Renviron' files.

For profiling code, see Rprof.

Examples

```
## Not run:
## Example ~/.Renviron on Unix
R_LIBS=~/R/library
PAGER=/usr/local/bin/less

## Example .Renviron on Windows
R_LIBS=C:/R/library
MY_TCLTK="c:/Program Files/Tcl/bin"
# Variable expansion in double quotes, string literals with backslashes in
# single quotes.
R_LIBS_USER="${APPDATA}"'\R-library'

## Example of setting R_DEFAULT_PACKAGES (from R CMD check)
R_DEFAULT_PACKAGES='utils,grDevices,graphics,stats'
# this loads the packages in the order given, so they appear on
# the search path in reverse order.

## Example of .Rprofile
options(width=65, digits=5)
options(show.signif.stars=FALSE)
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal=FALSE))
set.seed(1234)
.First <- function() cat("\n   Welcome to R!\n\n")
.Last <- function()  cat("\n   Goodbye!\n\n")

## Example of Rprofile.site
```

```
local({
  # add MASS to the default packages, set a CRAN mirror
  old <- getOption("defaultPackages"); r <- getOption("repos")
  r["CRAN"] <- "http://my.local.cran"
  options(defaultPackages = c(old, "MASS"), repos = r)
  ## (for Unix terminal users) set the width from COLUMNS if set
  cols <- Sys.getenv("COLUMNS")
  if(nzchar(cols)) options(width = as.integer(cols))
  # interactive sessions get a fortune cookie (needs fortunes package)
  if (interactive())
    fortunes::fortune()
})

## if .Renviron contains
FOOBAR="coo\bar"doh\ex"abc\"def'"

## then we get
# > cat(Sys.getenv("FOOBAR"), "\n")
# coo\bardoh\exabc"def'

## End(Not run)
```

---

stop *Stop Function Execution*

Description

stop stops execution of the current expression and executes an error action.

geterrmessage gives the last error message.

Usage

```
stop(..., call. = TRUE, domain = NULL)
geterrmessage()
```

Arguments

...	zero or more objects which can be coerced to character (and which are pasted together with no separator) or a single condition object.
call.	logical, indicating if the call should become part of the error message.
domain	see gettext. If NA, messages will not be translated.

Details

The error action is controlled by error handlers established within the executing code and by the current default error handler set by options(error=). The error is first signaled as if using signalCondition(). If there are no handlers or if all handlers return, then the error message is printed (if options("show.error.messages") is true) and the default error handler is used. The

default behaviour (the NULL error-handler) in interactive use is to return to the top level prompt or the top level browser, and in non-interactive use to (effectively) call q("no", status = 1, runLast = FALSE) unless getOption("catch.script.errors") is true.

The default handler stores the error message in a buffer; it can be retrieved by geterrmessage(). It also stores a trace of the call stack that can be retrieved by traceback().

Errors will be truncated to getOption("warning.length") characters, default 1000.

If a condition object is supplied it should be the only argument, and further arguments will be ignored, with a warning.

Value

geterrmessage gives the last error message, as a character string ending in "\n".

Note

Use domain = NA whenever ... contain a result from gettextf() as that is translated already.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

warning, try to catch errors and retry, and options for setting error handlers. stopifnot for validity testing. tryCatch and withCallingHandlers can be used to establish custom handlers while executing an expression.

gettext for the mechanisms for the automated translation of messages.

Examples

```
iter <- 12
try(if(iter > 10) stop("too many iterations"))

tst1 <- function(...) stop("dummy error")
try(tst1(1:10, long, calling, expression))

tst2 <- function(...) stop("dummy error", call. = FALSE)
try(tst2(1:10, longcalling, expression, but.not.seen.in.Error))
```

stopifnot *Ensure the Truth of R Expressions*

Description

If any of the expressions (in `...` or `exprs`) are not `all TRUE`, `stop` is called, producing an error message indicating the *first* expression which was not (`all`) true.

Usage

```
stopifnot(..., exprs, exprObject, local = TRUE)
```

Arguments

`..., exprs`	any number of R expressions, which should each evaluate to (a logical vector of all) `TRUE`. Use *either* `...` *or* `exprs`, the latter typically an unevaluated expression of the form
	`{`
	`  expr1`
	`  expr2`
	`  ....`
	`}`
	Note that e.g., positive numbers are *not* `TRUE`, even when they are coerced to `TRUE`, e.g., inside `if(.)` or in arithmetic computations in R.
	If names are provided to `...`, they will be used in lieu of the default error message.

`exprObject`	alternative to `exprs` or `...`: an 'expression-like' object, typically an `expression`, but also a `call`, a `name`, or atomic constant such as `TRUE`.
`local`	(only when `exprs` is used:) indicates the `environment` in which the expressions should be evaluated; by default the one from where `stopifnot()` has been called.

Details

This function is intended for use in regression tests or also argument checking of functions, in particular to make them easier to read.

`stopifnot(A, B)` or equivalently `stopifnot(exprs= {A ; B})` are conceptually equivalent to

```
{ if(any(is.na(A)) || !all(A)) stop(...);
  if(any(is.na(B)) || !all(B)) stop(...) }
```

Since R version 3.6.0, `stopifnot()` no longer handles potential errors or warnings (by `tryCatch()` etc) for each single expression and may use `sys.call(n)` to get a meaningful and short error message in case an expression did not evaluate to all TRUE. This provides considerably less overhead.

Since R version 3.5.0, expressions *are* evaluated sequentially, and hence evaluation stops as soon as there is a "non-TRUE", as indicated by the above conceptual equivalence statement.

Also, since R version 3.5.0, `stopifnot(exprs = { ... })` can be used alternatively and may be preferable in the case of several expressions, as they are more conveniently evaluated interactively ("no extraneous , ").

Since R version 3.4.0, when an expression (from `...`) is not true *and* is a call to `all.equal`, the error message will report the (first part of the) differences reported by `all.equal(*)`; since R 4.3.0, this happens for all calls where `"all.equal"` `pmatch()`es the function called, e.g., when that is called `all.equalShow`, see the example in `all.equal`.

Value

(`NULL` if all statements in `...` are `TRUE`.)

Note

Trying to use the `stopifnot(exprs = ..)` version via a shortcut, say,

```
assertWRONG <- function(exprs) stopifnot(exprs = exprs)
```

is delicate and the above is *not a good idea*. Contrary to `stopifnot()` which takes care to evaluate the parts of `exprs` one by one and stop at the first non-TRUE, the above short cut would typically evaluate all parts of `exprs` and pass the result, i.e., typically of the *last* entry of `exprs` to `stopifnot()`.

However, a more careful version,

```
assert <- function(exprs) eval.parent(substitute(stopifnot(exprs = exprs)))
```

may be a nice short cut for `stopifnot(exprs = *)` calls using the more commonly known verb as function name.

See Also

`stop`, `warning`; `assertCondition` in package **tools** complements `stopifnot()` for testing warnings and errors.

Examples

```
## NB: Some of these examples are expected to produce an error. To
##     prevent them from terminating a run with example() they are
##     piped into a call to try().

stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

m <- matrix(c(1,3,3,1), 2, 2)
stopifnot(m == t(m), diag(m) == rep(1, 2)) # all(.) |=>  TRUE

stopifnot(length(10)) |> try() # gives an error: '1' is *not* TRUE
## even when   if(1) "ok"   works
```

```
stopifnot(all.equal(pi, 3.141593),  2 < 2, (1:10 < 12), "a" < "b") |> try()
## More convenient for interactive "line by line" evaluation:
stopifnot(exprs = {
  all.equal(pi, 3.1415927)
  2 < 2
  1:10 < 12
  "a" < "b"
}) |> try()

eObj <- expression(2 < 3, 3 <= 3:6, 1:10 < 2)
stopifnot(exprObject = eObj) |> try()
stopifnot(exprObject = quote(3 == 3))
stopifnot(exprObject = TRUE)


# long all.equal() error messages are abbreviated:
stopifnot(all.equal(rep(list(pi),4), list(3.1, 3.14, 3.141, 3.1415))) |> try()

# The default error message can be overridden to be more informative:
m[1,2] <- 12
stopifnot("m must be symmetric"= m == t(m)) |> try()
#=> Error: m must be symmetric

##' warnifnot(): a "only-warning" version of stopifnot()
##'   {Yes, learn how to use do.call(substitute, ...) in a powerful manner !!}
warnifnot <- stopifnot ; N <- length(bdy <- body(warnifnot))
bdy        <- do.call(substitute, list(bdy,   list(stopifnot = quote(warnifnot))))
bdy[[N-1]] <- do.call(substitute, list(bdy[[N-1]], list(stop = quote(warning))))
body(warnifnot) <- bdy
warnifnot(1 == 1, 1 < 2, 2 < 2) # => warns " 2 < 2 is not TRUE  "
warnifnot(exprs = {
    1 == 1
    3 < 3  # => warns "3 < 3 is not TRUE"
})
```

`strptime` *Date-time Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times.

Usage

```
## S3 method for class 'POSIXct'
format(x, format = "", tz = "", usetz = FALSE, ...)
## S3 method for class 'POSIXlt'
format(x, format = "", usetz = FALSE,
       digits = getOption("digits.secs"), ...)
```

```
## S3 method for class 'POSIXt'
as.character(x, digits = if(inherits(x, "POSIXlt")) 14L else 6L,
             OutDec = ".", ...)

strftime(x, format = "", tz = "", usetz = FALSE, ...)
strptime(x, format, tz = "")
```

Arguments

x	an object to be converted: a character vector for strptime, an object which can be converted to "POSIXlt" for strftime.
tz	a character string specifying the time zone to be used for the conversion. System-specific (see as.POSIXlt), but "" is the current time zone, and "GMT" is UTC. Invalid values are most commonly treated as UTC, on some platforms with a warning.
format	a character string. The default for the format methods is "%Y-%m-%d %H:%M:%S" if any element has a time component which is not midnight, and "%Y-%m-%d" otherwise. If options("digits.secs") is set, up to the specified number of digits will be printed for seconds.
...	further arguments to be passed from or to other methods.
usetz	logical. Should the time zone abbreviation be appended to the output? This is used in printing times, and more reliable than using "%Z".
digits	integer determining the format()ing of seconds when needed. Note that the defaults for format() and as.character() differ on purpose, as.character() giving close to full accuracy as it does for numbers.
OutDec	a 1-character string specifying the decimal point to be used; the default is *not* getOption("OutDec") on purpose.

Details

The format and as.character methods and strftime convert objects from the classes "POSIXlt" and "POSIXct" to character vectors.

strptime converts character vectors to class "POSIXlt": its input x is first converted by as.character. Each input string is processed as far as necessary for the format specified: any trailing characters are ignored.

strftime is a wrapper for format.POSIXlt, and it and format.POSIXct first convert to class "POSIXlt" by calling as.POSIXlt (so they also work for class "Date"). Note that only that conversion depends on the time zone. Since R version 4.2.0, as.POSIXlt() conversion now treats the non-finite numeric -Inf, Inf, NA and NaN differently (where previously all were treated as NA). Also the format() method for POSIXlt now treats these different non-finite times and dates analogously to type double.

The usual vector re-cycling rules are applied to x and format so the answer will be of length of the longer of these vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months, the AM/PM indicator (if used) and the separators

in output formats such as `%x` and `%X`, *via* the setting of the `LC_TIME` locale category. The 'current locale' of the descriptions might mean the locale in use at the start of the R session or when these functions are first used. (For input, the locale-specific conversions can be changed by calling `Sys.setlocale` with category `LC_TIME` (or `LC_ALL`). For output, what happens depends on the OS but usually works.)

The details of the formats are platform-specific, but the following are likely to be widely available: most are defined by the POSIX standard. A *conversion specification* is introduced by `%`, usually followed by a single letter or `O` or `E` and then a single letter. Any character in the format string not part of a conversion specification is interpreted literally (and `%%` gives `%`). Widely implemented conversion specifications include

`%a` Abbreviated weekday name in the current locale on this platform. (Also matches full name on input: in some locales there are no abbreviations of names.)

`%A` Full weekday name in the current locale. (Also matches abbreviated name on input.)

`%b` Abbreviated month name in the current locale on this platform. (Also matches full name on input: in some locales there are no abbreviations of names.)

`%B` Full month name in the current locale. (Also matches abbreviated name on input.)

`%c` Date and time. Locale-specific on output, `"%a %b %e %H:%M:%S %Y"` on input.

`%C` Century (00–99): the integer part of the year divided by 100.

`%d` Day of the month as decimal number (01–31).

`%D` Date format such as `%m/%d/%y`: the C99 standard says it should be that exact format (but not all OSes comply).

`%e` Day of the month as decimal number (1–31), with a leading space for a single-digit number.

`%F` Equivalent to %Y-%m-%d (the ISO 8601 date format).

`%g` The last two digits of the week-based year (see `%V`). (Accepted but ignored on input.)

`%G` The week-based year (see `%V`) as a decimal number. (Accepted but ignored on input.)

`%h` Equivalent to `%b`.

`%H` Hours as decimal number (00–23). As a special exception strings such as '`24:00:00`' are accepted for input, since ISO 8601 allows these.

`%I` Hours as decimal number (01–12).

`%j` Day of year as decimal number (001–366): For input, 366 is only valid in a leap year.

`%m` Month as decimal number (01–12).

`%M` Minute as decimal number (00–59).

`%n` Newline on output, arbitrary whitespace on input.

`%p` AM/PM indicator in the locale. Used in conjunction with `%I` and **not** with `%H`. An empty string in some locales (for example on some OSes, non-English European locales including Russia). The behaviour is undefined if used for input in such a locale.

Some platforms accept `%P` for output, which uses a lower-case version (`%p` may also use lower case): others will output `P`.

`%r` For output, the 12-hour clock time (using the locale's AM or PM): only defined in some locales, and on some OSes misleading in locales which do not define an AM/PM indicator. For input, equivalent to `%I:%M:%S %p`.

%R Equivalent to %H:%M.

%S Second as integer (00–61), allowing for up to two leap-seconds (but POSIX-compliant implementations will ignore leap seconds).

%t Tab on output, arbitrary whitespace on input.

%T Equivalent to %H:%M:%S.

%u Weekday as a decimal number (1–7, Monday is 1).

%U Week of the year as decimal number (00–53) using Sunday as the first day 1 of the week (and typically with the first Sunday of the year as day 1 of week 1). The US convention.

%V Week of the year as decimal number (01–53) as defined in ISO 8601. If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. See %G (%g) for the year corresponding to the week given by %V. (Accepted but ignored on input.)

%w Weekday as decimal number (0–6, Sunday is 0).

%W Week of the year as decimal number (00–53) using Monday as the first day of week (and typically with the first Monday of the year as day 1 of week 1). The UK convention.

%x Date. Locale-specific on output, "%y/%m/%d" on input.

%X Time. Locale-specific on output, "%H:%M:%S" on input.

%y Year without century (00–99). On input, values 00 to 68 are prefixed by 20 and 69 to 99 by 19 – that is the behaviour specified by the 2018 POSIX standard, but it does also say 'it is expected that in a future version the default century inferred from a 2-digit year will change'.

%Y Year with century. Note that whereas there was no zero in the original Gregorian calendar, ISO 8601:2004 defines it to be valid (interpreted as 1BC): see https://en.wikipedia.org/wiki/0_(year). However, the standards also say that years before 1582 in its calendar should only be used with agreement of the parties involved.

For input, only years 0:9999 are accepted.

%z Signed offset in hours and minutes from UTC, so -0800 is 8 hours behind UTC. (Standard only for output. For input R currently supports it on all platforms – values from -1400 to +1400 are accepted.)

%Z (Output only.) Time zone abbreviation as a character string (empty if not available). This may not be reliable when a time zone has changed abbreviations over the years.

Where leading zeros are shown they will be used on output but are optional on input. Names are matched case-insensitively on input: whether they are capitalized on output depends on the platform and the locale. Note that abbreviated names are platform-specific (although the standards specify that in the 'C' locale they must be the first three letters of the capitalized English name: this convention is widely used in English-language locales but for example the French month abbreviations are not the same on any two of Linux, macOS, Solaris and Windows). Knowing what the abbreviations are is essential if you wish to use %a, %b or %h as part of an input format: see the examples for how to check.

When %z or %Z is used for output with an object with an assigned time zone an attempt is made to use the values for that time zone — but it is not guaranteed to succeed.

The definition of 'whitespace' for %n and %t is platform-dependent: for most it does not include non-breaking spaces.

Not in the standards and less widely implemented are

`%k` The 24-hour clock time with single digits preceded by a blank.

`%l` The 12-hour clock time with single digits preceded by a blank.

`%s` (Output only.) The number of seconds since the epoch.

`%+` (Output only.) Similar to `%c`, often `"%a %b %e %H:%M:%S %Z %Y"`. May depend on the locale.

For output there are also `%O[dHImMUVwWy]` which may emit numbers in an alternative locale-dependent format (e.g., roman numerals), and `%E[cCyYxX]` which can use an alternative 'era' (e.g., a different religious calendar). Which of these are supported is OS-dependent. These are accepted for input, but with the standard interpretation.

Specific to R is `%OSn`, which for output gives the seconds truncated to `0 <= n <= 6` decimal places (and if `%OS` is not followed by a digit, it uses the setting of `getOption("digits.secs")`, or if that is unset, `n = 0`). Further, for `strptime` `%OS` will input seconds including fractional seconds. Note that `%S` does not read fractional parts on output.

The behaviour of other conversion specifications (and even if other character sequences commencing with `%` *are* conversion specifications) is system-specific. Some systems document that the use of multi-byte characters in `format` is unsupported: UTF-8 locales are unlikely to cause a problem.

Value

The `format` methods and `strftime` return character vectors representing the time. `NA` times are returned as `NA_character_`.

`strptime` turns character representations into an object of class `"POSIXlt"`. The time zone is used to set the `isdst` component and to set the `"tzone"` attribute if `tz != ""`. If the specified time is invalid (for example '`"2010-02-30 08:00"`') all the components of the result are `NA`. (NB: this does means exactly what it says – if it is an invalid time, not just a time that does not exist in some time zone.)

Printing years

Everyone agrees that years from 1000 to 9999 should be printed with 4 digits, but the standards do not define what is to be done outside that range. For years 0 to 999 most OSes pad with zeros or spaces to 4 characters, but Linux/glibc outputs just the number.

OS facilities will probably not print years before 1 CE (aka 1 AD) 'correctly' (they tend to assume the existence of a year 0: see https://en.wikipedia.org/wiki/0_(year), and some OSes get them completely wrong). Common formats are `-45` and `-045`.

Years after 9999 and before -999 are normally printed with five or more characters.

Some platforms support modifiers from POSIX 2008 (and others). On Linux/glibc the format `"%04Y"` assures a minimum of four characters and zero-padding (the default is no padding). The internal code (as used on Windows and by default on macOS) uses zero-padding by default (this can be controlled by environment variable `R_PAD_YEARS_BY_ZERO`). On those platforms, formats `%04Y`, `%_4Y` and `%_Y` can be used for zero, space and no padding respectively. (On macOS, the native code (not the default) supports none of these and uses zero-padding to 4 digits.)

Time zone offsets

Offsets from GMT (also known as UTC) are part of the conversion between timezones and to/from class `"POSIXct"`, but cause difficulties as they are often computed incorrectly.

They conventionally have the opposite sign from time-zone specifications (see Sys.timezone): positive values are East of the meridian. Although there have been time zones with offsets like +00:09:21 (Paris in 1900), and -00:44:30 (Liberia until 1972), offsets are usually treated as whole numbers of minutes, and are most often seen in RFC 5322 email headers in forms like -0800 (e.g., used on the Pacific coast of the USA in winter).

Format %z can be used for input or output: it is a character string, conventionally plus or minus followed by two digits for hours and two for minutes: the standards say that an empty string should be output if the offset is undetermined, but some systems use +0000 or the offsets for the time zone in use for the current year. (On some platforms this works better after conversion to "POSIXct". Some platforms only recognize hour or half-hour offsets for output.)

Using %z for input makes most sense with tz = "UTC".

Sources

Input uses the POSIX function strptime and output the C99 function strftime.

However, not all OSes (notably Windows) provided strptime and many issues were found for those which did, so since 2000 R has used a fork of code from 'glibc'. The forked code uses the system's strftime to find the locale-specific day and month names and any AM/PM indicator.

On some platforms (including Windows and by default on macOS) the system's strftime is replaced (along with most of the rest of the C-level datetime code) by code modified from IANA's 'tzcode' distribution (https://www.iana.org/time-zones).

Note that as strftime is used for output (and not wcsftime), argument format is translated if necessary to the session encoding.

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-28" and a time as "14:01:02" using leading zeroes as here. (The ISO form uses no space, possibly 'T', to separate dates and times: R uses a space by default.)

For strptime the input string need not specify the date completely: it is assumed that unspecified seconds, minutes or hours are zero, and an unspecified year, month or day is the current one. (However, if a month is specified, the day of that month has to be specified by %d or %e since the current day of the month need not be valid for the specified month.) Some components may be returned as NA (but an unknown tzone component is represented by an empty string).

If the time zone specified is invalid on your system, what happens is system-specific but it will probably be ignored.

Remember that in most time zones some times do not occur and some occur twice because of transitions to/from 'daylight saving' (also known as 'summer') time. strptime does not validate such times (it does not assume a specific time zone), but conversion by as.POSIXct will do so. Conversion by strftime and formatting/printing uses OS facilities and may return nonsensical results for non-existent times at DST transitions.

In a C locale %c is required to be "%a %b %e %H:%M:%S %Y". As Windows does not comply (and uses a date format not understood outside N. America), that format is used by R on Windows in all locales.

There is a limit of 2048 bytes on each string produced by strftime and the format methods. As from R 4.3.0 attempting to exceed this is an error (previous versions silently truncated at 255 bytes).

References

International Organization for Standardization (2004, 2000, ...) 'ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.', slightly updated to International Organization for Standardization (2019) 'ISO 8601-1:2019. Date and time – Representations for information interchange – Part 1: Basic rules', and further amended in 2022. For links to versions available on-line see (at the time of writing) https://dotat.at/tmp/ISO_8601-2004_E.pdf and https://www.qsl.net/g1smd/isopdf.htm; for information on the current official version, see https://www.iso.org/iso/iso8601 and https://en.wikipedia.org/wiki/ISO_8601.

The POSIX 1003.1 standard, which is in some respects stricter than ISO 8601.

See Also

DateTimeClasses for details of the date-time classes; locales to query or set a locale.

Your system's help page on `strftime` to see how to specify their formats. (On some systems, including Windows, `strftime` is replaced by more comprehensive internal code.)

Examples

```
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y %Z")

## time to sub-second accuracy (if supported by the OS)
format(Sys.time(), "%H:%M:%OS3")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some non-English locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- strptime(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z
(chz <- as.character(z)) # same w/o TZ
## *here* (but not in general), the same as format():
stopifnot(exprs = {
     identical(chz, format(z))
     grepl("^1960-0[137]-[03][012]$", chz[!is.na(z)])
})

## read in date/time info in format 'm/d/y h:m:s'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
x <- paste(dates, times)
z2 <- strptime(x, "%m/%d/%y %H:%M:%S")
z2
## *here* (but not in general), the same as format():
stopifnot(identical(format(z2), as.character(z2)))

## time with fractional seconds
```

```
z3 <- strptime("20/2/06 11:16:16.683", "%d/%m/%y %H:%M:%OS")
z3 # prints without fractional seconds by default, digits.sec = NULL ("= 0")
op <- options(digits.secs = 3)
z3 # shows the 3 extra digits
as.character(z3) # ditto
options(op)

## time zone names are not portable, but 'EST5EDT' comes pretty close.
## (but its interpretation may not be universal: see ?timezones)
z4 <- strptime(c("2006-01-08 10:07:52", "2006-08-07 19:33:02"),
               "%Y-%m-%d %H:%M:%S", tz = "EST5EDT")
z4
attr(z4, "tzone")
as.character(z4)
z4$sec[2] <- pi # "very" fractional seconds
as.character(z4) # shows full precision
format(z4) # no fractional sec
format(z4, digits=8) # shows only 6  (hard-wired maximum)
format(z4, digits=4)


## An RFC 5322 header (Eastern Canada, during DST)
## In a non-English locale the commented lines may be needed.

## prev <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
strptime("Tue, 23 Mar 2010 14:36:38 -0400", "%a, %d %b %Y %H:%M:%S %z")
## Sys.setlocale("LC_TIME", prev)

## Make sure you know what the abbreviated names are for you if you wish
## to use them for input (they are matched case-insensitively):
format(s1 <- seq.Date(as.Date('1978-01-01'), by = 'day',   len =  7), "%a")
format(s2 <- seq.Date(as.Date('2000-01-01'), by = 'month', len = 12), "%b")

## Non-finite date-times :
format(as.POSIXct(Inf)) # "Inf"  (was  NA  in R <= 4.1.x)
format(as.POSIXlt(c(-Inf,Inf,NaN,NA))) # were all NA
```

`strrep` *Repeat the Elements of a Character Vector*

Description

Repeat the character strings in a character vector a given number of times (i.e., concatenate the respective numbers of copies of the strings).

Usage

```
strrep(x, times)
```

Arguments

x	a character vector, or an object which can be coerced to a character vector using as.character.
times	an integer vector giving the (non-negative) numbers of times to repeat the respective elements of x.

Details

The elements of x and times will be recycled as necessary (if one has no elements, and empty character vector is returned). Missing elements in x or times result in missing elements of the return value.

Value

A character vector with the elements of the given character vector repeated the given numbers of times.

Examples

```
strrep("ABC", 2)
strrep(c("A", "B", "C"), 1 : 3)
## Create vectors with the given numbers of spaces:
strrep(" ", 1 : 5)
```

---

strsplit *Split the Elements of a Character Vector*

Description

Split the elements of a character vector x into substrings according to the matches to substring split within them.

Usage

```
strsplit(x, split, fixed = FALSE, perl = FALSE, useBytes = FALSE)
```

Arguments

x	character vector, each element of which is to be split. Other inputs, including a factor, will give an error.
split	character vector (or object which can be coerced to such) containing regular expression(s) (unless fixed = TRUE) to use for splitting. If empty matches occur, in particular if split has length 0, x is split into single characters. If split has length greater than 1, it is re-cycled along x.
fixed	logical. If TRUE match split exactly, otherwise use regular expressions. Has priority over perl.

perl	logical. Should Perl-compatible regexps be used?
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character, and inputs with marked encodings are not converted. This is forced (with a warning) if any input is found which is marked as "bytes" (see Encoding).

Details

Argument split will be coerced to character, so you will see uses with split = NULL to mean split = character(0), including in the examples below.

Note that splitting into single characters can be done *via* split = character(0) or split = ""; the two are equivalent. The definition of 'character' here depends on the locale: in a single-byte locale it is a byte, and in a multi-byte locale it is the unit represented by a 'wide character' (almost always a Unicode code point).

A missing value of split does not split the corresponding element(s) of x at all.

The algorithm applied to each input string is

```
repeat {
    if the string is empty
        break.
    if there is a match
        add the string to the left of the match to the output.
        remove the match and all to the left of it.
    else
        add the string to the output.
        break.
}
```

Note that this means that if there is a match at the beginning of a (non-empty) string, the first element of the output is "", but if there is a match at the end of the string, the output is the same as with the match removed.

Note also that if there is an empty match at the beginning of a non-empty string, the first character is returned and the algorithm continues with the rest of the string. This needs to be kept in mind when designing the regular expressions. For example, when looking for a word boundary followed by a letter ("[[:<:]]" with perl = TRUE), one can disallow a match at the beginning of a string (via "(?!^)[[:<:]]").

Invalid inputs in the current locale are warned about up to 5 times.

Value

A list of the same length as x, the i-th element of which contains the vector of splits of x[i].

If any element of x or split is declared to be in UTF-8 (see Encoding), all non-ASCII character strings in the result will be in UTF-8 and have their encoding declared as UTF-8. (This also holds if any element is declared to be Latin-1 except in a Latin-1 locale.) For perl = TRUE, useBytes = FALSE all non-ASCII strings in a multibyte locale are translated to UTF-8.

If any element of x or split is marked as "bytes" (see Encoding), all non-ASCII character strings created by the splitting in the result will be marked as "bytes", but encoding of the resulting

character strings not split is unspecified (may be "bytes" or the original). If no element of x or split is marked as "bytes", but useBytes = TRUE, even the encoding of the resulting character strings created by splitting is unspecified (may be "bytes" or "unknown", possibly invalid in the current encoding). Mixed use of "bytes" and other marked encodings is discouraged, but if still desired one may use iconv to re-encode the result e.g. to UTF-8 with suitably substituted invalid bytes.

See Also

paste for the reverse, grep and sub for string search and manipulation; also nchar, substr.

'regular expression' for the details of the pattern specification.

Option PCRE_use_JIT controls the details when perl = TRUE.

Examples

```
noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x, "e")

unlist(strsplit("a.b.c", "."))
## [1] "" "" "" "" ""
## Note that 'split' is a regexp!
## If you really want to split on '.', use
unlist(strsplit("a.b.c", "[.]"))
## [1] "a" "b" "c"
## or
unlist(strsplit("a.b.c", ".", fixed = TRUE))

## a useful function: rev() for strings
strReverse <- function(x)
        sapply(lapply(strsplit(x, NULL), rev), paste, collapse = "")
strReverse(c("abc", "Statistics"))

## get the first names of the members of R-core
a <- readLines(file.path(R.home("doc"),"AUTHORS"))[-(1:8)]
a <- a[(0:2)-length(a)]
(a <- sub(" .*","", a))
# and reverse them
strReverse(a)

## Note that final empty strings are not produced:
strsplit(paste(c("", "a", ""), collapse="#"), split="#")[[1]]
# [1] ""  "a"
## and also an empty string is only produced before a definite match:
strsplit("", " ")[[1]]    # character(0)
strsplit(" ", " ")[[1]]   # [1] ""
```

`strtoi` *Convert Strings to Integers*

Description

Convert strings to integers according to the given base using the C function `strtol`, or choose a suitable base following the C rules.

Usage

```
strtoi(x, base = 0L)
```

Arguments

`x`	a character vector, or something coercible to this by `as.character`.
`base`	an integer which is between 2 and 36 inclusive, or zero (default).

Details

Conversion is based on the C library function `strtol`.

For the default `base = 0L`, the base chosen from the string representation of that element of `x`, so different elements can have different bases (see the first example). The standard C rules for choosing the base are that octal constants (prefix `0` not followed by `x` or `X`) and hexadecimal constants (prefix `0x` or `0X`) are interpreted as base `8` and `16`; all other strings are interpreted as base `10`.

For a base greater than `10`, letters `a` to `z` (or `A` to `Z`) are used to represent `10` to `35`.

Value

An integer vector of the same length as `x`. Values which cannot be interpreted as integers or would overflow are returned as `NA_integer_`.

See Also

For decimal strings `as.integer` is equally useful.

Examples

```
strtoi(c("0xff", "077", "123"))
strtoi(c("ffff", "FFFF"), 16L)
strtoi(c("177", "377"), 8L)
```

strtrim *Trim Character Strings to Specified Display Widths*

Description

Trim character strings to specified display widths.

Usage

```
strtrim(x, width)
```

Arguments

x	a character vector, or an object which can be coerced to a character vector by as.character.
width	positive integer values: recycled to the length of x.

Details

'Width' is interpreted as the display width in a monospaced font. What happens with non-printable characters (such as backspace, tab) is implementation-dependent and may depend on the locale (e.g., they may be included in the count or they may be omitted).

Using this function rather than substr is important when there might be double-width (e.g., Chinese/Japanese/Korean) characters in the character vector.

Value

A character vector of the same length and with the same attributes as x (after possible coercion).

Elements of the result will have the encoding declared as that of the current locale (see Encoding) if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8.

Examples

```
strtrim(c("abcdef", "abcdef", "abcdef"), c(1,5,10))
```

structure *Attribute Specification*

Description

structure returns the given object with further attributes set.

Usage

```
structure(.Data, ...)
```

Arguments

.Data	an object which will have various attributes attached to it.
...	attributes, specified in tag = value form, which will be attached to data.

Details

Adding a class "factor" will ensure that numeric codes are given integer storage mode.

For historical reasons (these names are used when deparsing), attributes ".Dim", ".Dimnames", ".Names", ".Tsp" and ".Label" are renamed to "dim", "dimnames", "names", "tsp" and "levels".

It is possible to give the same tag more than once, in which case the last value assigned wins. As with other ways of assigning attributes, using tag = NULL removes attribute tag from .Data if it is present.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attributes, attr.

Examples

```
structure(1:6, dim = 2:3)
```

---

strwrap *Wrap Character Strings to Format Paragraphs*

Description

Each character string in the input is first split into paragraphs (or lines containing whitespace only). The paragraphs are then formatted by breaking lines at word boundaries. The target columns for wrapping lines and the indentation of the first and all subsequent lines of a paragraph can be controlled independently.

Usage

```
strwrap(x, width = 0.9 * getOption("width"), indent = 0,
        exdent = 0, prefix = "", simplify = TRUE, initial = prefix)
```

Arguments

x	a character vector, or an object which can be converted to a character vector by as.character.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a non-negative integer giving the indentation of the first line in a paragraph.
exdent	a non-negative integer specifying the indentation of subsequent lines in paragraphs.
prefix, initial	a character string to be used as prefix for each line except the first, for which initial is used.
simplify	a logical. If TRUE, the result is a single character vector of line text; otherwise, it is a list of the same length as x the elements of which are character vectors of line text obtained from the corresponding element of x. (Hence, the result in the former case is obtained by unlisting that of the latter.)

Details

Whitespace (space, tab or newline characters) in the input is destroyed. Double spaces after periods, question and explanation marks (thought as representing sentence ends) are preserved. Currently, possible sentence ends at line breaks are not considered specially.

Indentation is relative to the number of characters in the prefix string.

Value

A character vector (if simplify is TRUE), or a list of such character vectors, with declared input encodings preserved.

Examples

```
## Read in file 'THANKS'.
x <- paste(readLines(file.path(R.home("doc"), "THANKS")), collapse = "\n")
## Split into paragraphs and remove the first three ones
x <- unlist(strsplit(x, "\n[ \t\n]*\n"))[-(1:3)]
## Join the rest
x <- paste(x, collapse = "\n\n")
## Now for some fun:
writeLines(strwrap(x, width = 60))
writeLines(strwrap(x, width = 60, indent = 5))
writeLines(strwrap(x, width = 60, exdent = 5))
writeLines(strwrap(x, prefix = "THANKS> "))

## Note that messages are wrapped AT the target column indicated by
## 'width' (and not beyond it).
## From an R-devel posting by J. Hosking <jh910@juno.com>.
x <- paste(sapply(sample(10, 100, replace = TRUE),
           function(x) substring("aaaaaaaaaa", 1, x)), collapse = " ")
sapply(10:40,
       function(m)
       c(target = m, actual = max(nchar(strwrap(x, m)))))
```

subset *Subsetting Vectors, Matrices and Data Frames*

Description

Return subsets of vectors, matrices or data frames which meet conditions.

Usage

```
subset(x, ...)

## Default S3 method:
subset(x, subset, ...)

## S3 method for class 'matrix'
subset(x, subset, select, drop = FALSE, ...)

## S3 method for class 'data.frame'
subset(x, subset, select, drop = FALSE, ...)
```

Arguments

x	object to be subsetted.
subset	logical expression indicating elements or rows to keep: missing values are taken as false.
select	expression, indicating columns to select from a data frame.
drop	passed on to [indexing operator.
...	further arguments to be passed to or from other methods.

Details

This is a generic function, with methods supplied for matrices, data frames and vectors (including lists). Packages and users can add further methods.

For ordinary vectors, the result is simply `x[subset & !is.na(subset)]`.

For data frames, the `subset` argument works on the rows. Note that `subset` will be evaluated in the data frame, so columns can be referred to (by name) as variables in the expression (see the examples).

The `select` argument exists only for the methods for data frames and matrices. It works by first replacing column names in the selection expression with the corresponding column numbers in the data frame and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that for example ranges of columns can be specified easily, or single columns can be dropped (see the examples).

The `drop` argument is passed on to the indexing method for matrices and data frames: note that the default for matrices is different from that for indexing.

Factors may have empty levels after subsetting; unused levels are not automatically removed. See `droplevels` for a way to drop all unused levels from a data frame.

Value

An object similar to `x` contain just the selected elements (for a vector), rows and columns (for a matrix or data frame), and so on.

Warning

This is a convenience function intended for use interactively. For programming it is better to use the standard subsetting functions like `[`, and in particular the non-standard evaluation of argument `subset` can have unanticipated consequences.

Author(s)

Peter Dalgaard and Brian Ripley

See Also

`[`, `transform` `droplevels`

Examples

```
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)

with(airquality, subset(Ozone, Temp > 80))

## sometimes requiring a logical 'subset' argument is a nuisance
nm <- rownames(state.x77)
start_with_M <- nm %in% grep("^M", nm, value = TRUE)
subset(state.x77, start_with_M, Illiteracy:Murder)
# but in recent versions of R this can simply be
subset(state.x77, grepl("^M", nm), Illiteracy:Murder)
```

`substitute` *Substituting and Quoting Expressions*

Description

`substitute` returns the parse tree for the (unevaluated) expression `expr`, substituting any variables bound in `env`.

`quote` simply returns its argument. The argument is not evaluated and can be any R expression.

`enquote` is a simple one-line utility which transforms a call of the form `Foo(....)` into the call `quote(Foo(....))`. This is typically used to protect a `call` from early evaluation.

Usage

```
substitute(expr, env)
quote(expr)
enquote(cl)
```

Arguments

expr	any syntactically valid R expression.
cl	a call, i.e., an R object of class (and mode) "call".
env	an environment or a list object. Defaults to the current evaluation environment.

Details

The typical use of substitute is to create informative labels for data sets and plots. The myplot example below shows a simple use of this facility. It uses the functions deparse and substitute to create labels for a plot which are character string versions of the actual arguments to the function myplot.

Substitution takes place by examining each component of the parse tree as follows: If it is not a bound symbol in env, it is unchanged. If it is a promise object, i.e., a formal argument to a function or explicitly created using delayedAssign(), the expression slot of the promise replaces the symbol. If it is an ordinary variable, its value is substituted, unless env is .GlobalEnv in which case the symbol is left unchanged.

Both quote and substitute are 'special' primitive functions which do not evaluate their arguments.

Value

The mode of the result is generally "call" but may in principle be any type. In particular, single-variable expressions have mode "name" and constants have the appropriate base mode.

Note

substitute works on a purely lexical basis. There is no guarantee that the resulting expression makes any sense.

Substituting and quoting often cause confusion when the argument is expression(...). The result is a call to the expression constructor function and needs to be evaluated with eval to give the actual expression object.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

missing for argument 'missingness', bquote for partial substitution, sQuote and dQuote for adding quotation marks to strings. Quotes about forward, back, and double quotes ''', '`', and '"'.

all.names to retrieve the symbol names from an expression or call.

Examples

```
require(graphics)
(s.e <- substitute(expression(a + b), list(a = 1)))  #> expression(1 + b)
(s.s <- substitute( a + b,            list(a = 1)))  #> 1 + b
c(mode(s.e), typeof(s.e)) #  "call", "language"
c(mode(s.s), typeof(s.s)) #   (the same)
# but:
(e.s.e <- eval(s.e))          #>  expression(1 + b)
c(mode(e.s.e), typeof(e.s.e)) #  "expression", "expression"

substitute(x <- x + 1, list(x = 1)) # nonsense

myplot <- function(x, y)
    plot(x, y, xlab = deparse1(substitute(x)),
               ylab = deparse1(substitute(y)))

## Simple examples about lazy evaluation, etc:

f1 <- function(x, y = x)             { x <- x + 1; y }
s1 <- function(x, y = substitute(x)) { x <- x + 1; y }
s2 <- function(x, y) { if(missing(y)) y <- substitute(x); x <- x + 1; y }
a <- 10
f1(a)  # 11
s1(a)  # 11
s2(a)  # a
typeof(s2(a))  # "symbol"
```

---

substr *Substrings of a Character Vector*

Description

Extract or replace substrings in a character vector.

Usage

```
substr(x, start, stop)
substring(text, first, last = 1000000L)

substr(x, start, stop) <- value
substring(text, first, last = 1000000L) <- value
```

Arguments

x, text	a character vector.
start, first	integer. The first element to be extracted or replaced.
stop, last	integer. The last element to be extracted or replaced.
value	a character vector, recycled if necessary.

Details

substring is compatible with S, with first and last instead of start and stop. For vector arguments, it expands the arguments cyclically to the length of the longest *provided* none are of zero length.

When extracting, if start is larger than the string length then "" is returned.

For the extraction functions, x or text will be converted to a character vector by as.character if it is not already one.

For the replacement functions, if start is larger than the string length then no replacement is done. If the portion to be replaced is longer than the replacement string, then only the portion the length of the string is replaced.

If any argument is an NA element, the corresponding element of the answer is NA.

Elements of the result will be have the encoding declared as that of the current locale (see Encoding) if the corresponding input had a declared Latin-1 or UTF-8 encoding and the current locale is either Latin-1 or UTF-8.

If an input element has declared "bytes" encoding (see Encoding), the subsetting is done in units of bytes not characters.

Value

For substr, a character vector of the same length and with the same attributes as x (after possible coercion).

For substring, a character vector of length the longest of the arguments. This will have names taken from x (if it has any after coercion, repeated as needed), and other attributes copied from x if it is the longest of the arguments).

For the replacement functions, a character vector of the same length as x or text, with attributes such as names preserved.

Elements of x or text with a declared encoding (see Encoding) will be returned with the same encoding.

Note

The S version of substring<- ignores last; this version does not.

These functions are often used with nchar to truncate a display. That does not really work (you want to limit the width, not the number of characters, so it would be better to use strtrim), but at least make sure you use the default nchar(type = "chars").

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (substring.)

See Also

strsplit, paste, nchar.

Examples

```
substr("abcdef", 2, 4)
substring("abcdef", 1:6, 1:6)
## strsplit() is more efficient ...

substr(rep("abcdef", 4), 1:4, 4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
substr(x, 2, 5)
substring(x, 2, 4:6)

X <- x
names(X) <- LETTERS[seq_along(x)]
comment(X) <- noquote("is a named vector")
str(aX <- attributes(X))
substring(x, 2) <- c("..", "+++")
substring(X, 2) <- c("..", "+++")
X
stopifnot(x == X, identical(aX, attributes(X)), nzchar(comment(X)))
```

---

sum *Sum of Vector Elements*

Description

sum returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

...	numeric or complex or logical vectors.
na.rm	logical. Should missing values (including NaN) be removed?

Details

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

If na.rm is FALSE an NA or NaN value in any of the arguments will cause a value of NA or NaN to be returned, otherwise NA and NaN values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were integer(0).

Loss of accuracy can occur when summing values of different signs: this can even occur for sufficiently long integer inputs if the partial sums would cause integer overflow. Where possible extended-precision accumulators are used, typically well supported with C99 and newer, but possibly platform-dependent.

Value

The sum. If all of the ... arguments are of type integer or logical, then the sum is integer when possible and is double otherwise. Integer overflow should no longer happen since R version 3.5.0. For other argument types it is a length-one numeric (double) or complex vector.

NB: the sum of an empty set is zero, by definition.

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

'plotmath' for the use of sum in plot annotation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

colSums for row and column sums.

Examples

```
## Pass a vector to sum, and it will add the elements together.
sum(1:5)

## Pass several numbers to sum, and it also adds the elements.
sum(1, 2, 3, 4, 5)

## In fact, you can pass vectors into several arguments, and everything gets added.
sum(1:2, 3:5)

## If there are missing values, the sum is unknown, i.e., also missing, ....
sum(1:5, NA)
## ... unless  we exclude missing values explicitly:
sum(1:5, NA, na.rm = TRUE)
```

---

summary *Object Summaries*

Description

summary is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular methods which depend on the class of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits, quantile.type = 7)
## S3 method for class 'data.frame'
summary(object, maxsum = 7,
       digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor'
summary(object, maxsum = 100, ...)

## S3 method for class 'matrix'
summary(object, ...)

## S3 method for class 'summaryDefault'
format(x, digits = max(3L, getOption("digits") - 3L), ...)
 ## S3 method for class 'summaryDefault'
print(x, digits = max(3L, getOption("digits") - 3L), ...)
```

Arguments

`object`	an object for which a summary is desired.
`x`	a result of the *default* method of `summary()`.
`maxsum`	integer, indicating how many levels should be shown for `factor`s.
`digits`	integer, used for number formatting with `signif()` (for `summary.default`) or `format()` (for `summary.data.frame`). In `summary.default`, if not specified (i.e., `missing(.)`), `signif()` will *not* be called anymore (since R >= 3.4.0, where the default has been changed to only round in the `print` and `format` methods).
`quantile.type`	integer code used in `quantile(*, type=quantile.type)` for the default method.
`...`	additional arguments affecting the summary produced.

Details

For `factor`s, the frequency of the first `maxsum - 1` most frequent levels is shown, and the less frequent levels are summarized in `"(Others)"` (resulting in at most `maxsum` frequencies).

The functions `summary.lm` and `summary.glm` are examples of particular methods which summarize the results produced by `lm` and `glm`.

Value

The form of the value returned by `summary` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

The default method returns an object of class `c("summaryDefault", "table")` which has specialized `format` and `print` methods. The `factor` method returns an integer vector.

The matrix and data frame methods return a matrix of class "table", obtained by applying summary to each column and collating the results.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

anova, summary.glm, summary.lm.

Examples

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)

lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

---

svd *Singular Value Decomposition of a Matrix*

Description

Compute the singular-value decomposition of a rectangular matrix.

Usage

```
svd(x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)

La.svd(x, nu = min(n, p), nv = min(n, p))
```

Arguments

x	a numeric or complex matrix whose SVD decomposition is to be computed. Logical matrices are coerced to numeric.
nu	the number of left singular vectors to be computed. This must between 0 and n = nrow(x).
nv	the number of right singular vectors to be computed. This must be between 0 and p = ncol(x).
LINPACK	logical. Defunct and an error.

Details

The singular value decomposition plays an important role in many statistical techniques. `svd` and `La.svd` provide two interfaces which differ in their return values.

Computing the singular vectors is the slow part for large matrices. The computation will be more efficient if both `nu <= min(n, p)` and `nv <= min(n, p)`, and even more so if both are zero.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code (most often `1`): these can only be interpreted by detailed study of the FORTRAN code but mean that the algorithm failed to converge.

Missing, `NaN` or infinite values in `x` will given an error.

Value

The SVD decomposition of the matrix as computed by LAPACK,

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}',$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, $\boldsymbol{V}'$ means *V transposed* (and conjugated for complex input), and $\boldsymbol{D}$ is a diagonal matrix with the (non-negative) singular values D_{ii} in decreasing order. Equivalently, $\boldsymbol{D} = \boldsymbol{U}'\boldsymbol{X}\boldsymbol{V}$, which is verified in the examples.

The returned value is a list with components

`d`	a vector containing the singular values of `x`, of length `min(n, p)`, sorted decreasingly.
`u`	a matrix whose columns contain the left singular vectors of `x`, present if `nu > 0`. Dimension `c(n, nu)`.
`v`	a matrix whose columns contain the right singular vectors of `x`, present if `nv > 0`. Dimension `c(p, nv)`.

Recall that the singular vectors are only defined up to sign (a constant of modulus one in the complex case). If a left singular vector has its sign changed, changing the sign of the corresponding right vector gives an equivalent decomposition.

For `La.svd` the return value replaces `v` by `vt`, the (conjugated if complex) transpose of `v`.

Source

The main functions used are the LAPACK routines `DGESDD` and `ZGESDD`.

LAPACK is from https://netlib.org/lapack/ and its guide is listed in the references.

References

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at https://netlib.org/lapack/lug/lapack_lug.html.

The 'Singular-value decomposition' Wikipedia article.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

eigen, qr.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, `+`) }
X <- hilbert(9)[, 1:6]
(s <- svd(X))
D <- diag(s$d)
s$u %*% D %*% t(s$v) #  X = U D V'
t(s$u) %*% X %*% s$v #  D = U' X V
```

---

sweep *Sweep out Array Summaries*

Description

Return an array obtained from an input array by sweeping out a summary statistic.

Usage

```
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
```

Arguments

x	an array, including a matrix.
MARGIN	a vector of indices giving the extent(s) of x which correspond to STATS. Where x has named dimnames, it can be a character vector selecting dimension names.
STATS	the summary statistic which is to be swept out.
FUN	the function to be used to carry out the sweep.
check.margin	logical. If TRUE (the default), warn if the length or dimensions of STATS do not match the specified dimensions of x. Set to FALSE for a small speed gain when you *know* that dimensions match.
...	optional arguments to FUN.

Details

FUN is found by a call to match.fun. As in the default, binary operators can be supplied if quoted or backquoted.

FUN should be a function of two arguments: it will be called with arguments x and an array of the same dimensions generated from STATS by aperm.

The consistency check among STATS, MARGIN and x is stricter if STATS is an array than if it is a vector. In the vector case, some kinds of recycling are allowed without a warning. Use sweep(x, MARGIN, as.array(STATS)) if STATS is a vector and you want to be warned if any recycling occurs.

Value

An array with the same shape as x, but with the summary statistics swept out.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

apply on which sweep used to be based; scale for centering and scaling.

Examples

```
require(stats) # for median
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)  # subtract the column medians

## More sweeping:
A <- array(1:24, dim = 4:2)

## no warnings in normal use
sweep(A, 1, 5)
(A.min <- apply(A, 1, min))  # == 1:4
sweep(A, 1, A.min)
sweep(A, 1:2, apply(A, 1:2, median))

## warnings when mismatch
sweep(A, 1, 1:3)  # STATS does not recycle
sweep(A, 1, 6:1)  # STATS is longer

## exact recycling:
sweep(A, 1, 1:2)  # no warning
sweep(A, 1, as.array(1:2))  # warning

## Using named dimnames

dimnames(A) <- list(fee=1:4, fie=1:3, fum=1:2)

mn_fum_fie <- apply(A, c("fum", "fie"), mean)
mn_fum_fie
sweep(A, c("fum", "fie"), mn_fum_fie)
```

---

switch *Select One of a List of Alternatives*

Description

switch evaluates EXPR and accordingly chooses one of the further arguments (in . . .).

Usage

```
switch(EXPR, ...)
```

Arguments

EXPR	an expression evaluating to a number or a character string.
...	the list of alternatives. If it is intended that EXPR has a character-string value these will be named, perhaps except for one alternative to be used as a 'default' value.

Details

switch works in two distinct ways depending whether the first argument evaluates to a character string or a number.

If the value of EXPR is not a character string it is coerced to integer. Note that this also happens for factors, with a warning, as typically the character level is meant. If the integer is between 1 and nargs()-1 then the corresponding element of ... is evaluated and the result returned: thus if the first argument is 3 then the fourth argument is evaluated and returned.

If EXPR evaluates to a character string then that string is matched (exactly) to the names of the elements in .... If there is a match then that element is evaluated unless it is missing, in which case the next non-missing element is evaluated, so for example switch("cc", a = 1, cc =, cd =, d = 2) evaluates to 2. If there is more than one match, the first matching element is used. In the case of no match, if there is an unnamed element of ... its value is returned. (If there is more than one such argument an error is signaled.)

The first argument is always taken to be EXPR: if it is named its name must (partially) match.

A warning is signaled if no alternatives are provided, as this is usually a coding error.

This is implemented as a primitive function that only evaluates its first argument and one other if one is selected.

Value

The value of one of the elements of ..., or NULL, invisibly (whenever no element is selected).

The result has the visibility (see invisible) of the element evaluated.

Warning

It is possible to write calls to switch that can be confusing and may not work in the same way in earlier versions of R. For compatibility (and clarity), always have EXPR as the first argument, naming it if partial matching is a possibility. For the character-string form, have a single unnamed argument as the default after the named values.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats)
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
centre(x, "mean")
centre(x, "median")
centre(x, "trimmed")

ccc <- c("b","QQ","a","A","bb")
# note: cat() produces no output for NULL
for(ch in ccc)
    cat(ch,":", switch(EXPR = ch, a = 1, b = 2:3), "\n")
for(ch in ccc)
    cat(ch,":", switch(EXPR = ch, a =, A = 1, b = 2:3, "Otherwise: last"),"\n")

## switch(f, *) with a factor f
ff <- gl(3,1, labels=LETTERS[3:1])
ff[1] # C
## so one might expect " is C" here, but
switch(ff[1], A = "I am A", B="Bb..", C=" is C")# -> "I am A"
## so we give a warning

## Numeric EXPR does not allow a default value to be specified
## -- it is always NULL
for(i in c(-1:3, 9))  print(switch(i, 1, 2 , 3, 4))

## visibility
switch(1, invisible(pi), pi)
switch(2, invisible(pi), pi)
```

`Syntax` *Operator Syntax and Precedence*

Description

Outlines R syntax and gives the precedence of operators.

Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

`::` `:::`	access variables in a namespace
`$` `@`	component / slot extraction

`[ [[`	indexing
`^`	exponentiation (right to left)
`- +`	unary minus and plus
`:`	sequence operator
`%any% |>`	special operators (including %% and %/%)
`* /`	multiply, divide
`+ -`	(binary) add, subtract
`< > <= >= == !=`	ordering and comparison
`!`	negation
`& &&`	and
`| ||`	or
`~`	as in formulae
`-> ->>`	rightwards assignment
`<- <<-`	assignment (right to left)
`=`	assignment (right to left)
`?`	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated. (Note that = is not necessarily an operator.)

The binary operators `::`, `:::`, `$` and `@` require names or string constants on the right hand side, and the first two also require them on the left.

The links in the **See Also** section cover most other aspects of the basic syntax.

Note

There are substantial precedence differences between R and S. In particular, in S ? has the same precedence as (binary) + - and & && | || have equal precedence.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Arithmetic`, `Comparison`, `Control`, `Extract`, `Logic`, `NumericConstants`, `Paren`, `Quotes`, `Reserved`.

The 'R Language Definition' manual.

Examples

```
## Logical AND ("&&") has higher precedence than OR ("||"):
TRUE || TRUE && FALSE   # is the same as
TRUE || (TRUE && FALSE) # and different from
(TRUE || TRUE) && FALSE

## Special operators have higher precedence than "!" (logical NOT).
## You can use this for %in% :
```

```
! 1:10 %in% c(2, 3, 5, 7) # same as !(1:10 %in% c(2, 3, 5, 7))
## but we strongly advise to use the "!( ... )" form in this case!


## '=' has lower precedence than '<-' ... so you should not mix them
##     (and '<-' is considered better style anyway):
## Not run: ## Consequently, this gives a ("non-catchable") error
 x <- y = 5  #->  Error in (x <- y) = 5 : ....

## End(Not run)
```

`Sys.getenv` *Get Environment Variables*

Description

`Sys.getenv` obtains the values of the environment variables.

Usage

```
Sys.getenv(x = NULL, unset = "", names = NA)
```

Arguments

`x`	a character vector, or `NULL`.
`unset`	a character string.
`names`	logical: should the result be named? If `NA` (the default) single-element results are not named whereas multi-element results are.

Details

Both arguments will be coerced to character if necessary.

Setting `unset = NA` will enable unset variables and those set to the value `""` to be distinguished, *if the OS does*. POSIX requires the OS to distinguish, and all known current R platforms do.

Value

A vector of the same length as `x`, with (if `names == TRUE`) the variable names as its `names` attribute. Each element holds the value of the environment variable named by the corresponding component of `x` (or the value of `unset` if no environment variable with that name was found).

On most platforms `Sys.getenv()` will return a named vector giving the values of all the environment variables, sorted in the current locale. It may be confused by names containing `=` which some platforms allow but POSIX does not. (Windows is such a platform: there names including `=` are truncated just before the first `=`.)

When `x` is missing and `names` is not false, the result is of class `"Dlist"` in order to get a nice `print` method.

See Also

Sys.setenv, Sys.getlocale for the locale in use, getwd for the working directory.

The help for 'environment variables' lists many of the environment variables used by R.

Examples

```
## whether HOST is set will be shell-dependent e.g. Solaris' csh did not.
Sys.getenv(c("R_HOME", "R_PAPERSIZE", "R_PRINTCMD", "HOST"))

s <- Sys.getenv() # *all* environment variables
op <- options(width=111) # (nice printing)
names(s)    # all settings (the values could be very long)
head(s, 12) # using the Dlist print() method

## Language and Locale settings -- but rather use Sys.getlocale()
s[grep("^L(C|ANG)", names(s))]
## typically R-related:
s[grep("^_?R_", names(s))]
options(op)# reset
```

Sys.getpid *Get the Process ID of the R Session*

Description

Get the process ID of the R Session. It is guaranteed by the operating system that two R sessions running simultaneously will have different IDs, but it is possible that R sessions running at different times will have the same ID.

Usage

```
Sys.getpid()
```

Value

An integer, often between 1 and 32767 under Unix-alikes (but for example FreeBSD and macOS use IDs up to 99999) and a positive integer (up to 32767) under Windows.

Examples

```
Sys.getpid()

## Show files opened from this R process
if(.Platform$OS.type == "unix") ## on Unix-alikes such Linux, macOS, FreeBSD:
   system(paste("lsof -p", Sys.getpid()))
```

`Sys.glob` *Wildcard Expansion on File Paths*

Description

Function to do wildcard expansion (also known as 'globbing') on file paths.

Usage

```
Sys.glob(paths, dirmark = FALSE)
```

Arguments

`paths`	character vector of patterns for relative or absolute filepaths. Missing values will be ignored.
`dirmark`	logical: should matches to directories from patterns that do not already end in `/` have a slash appended? May not be supported on all platforms.

Details

This expands tilde (see tilde expansion) and wildcards in file paths. For precise details of wildcards expansion, see your system's documentation on the `glob` system call. There is a POSIX 1003.2 standard (see `https://pubs.opengroup.org/onlinepubs/9699919799/functions/glob.html`) but some OSes will go beyond this.

All systems should interpret `*` (match zero or more characters), `?` (match a single character) and (probably) `[` (begin a character class or range). The handling of paths ending with a separator is system-dependent. On a POSIX-2008 compliant OS they will match directories (only), but as they are not valid filepaths on Windows, they match nothing there. (Earlier POSIX standards allowed them to match files.)

The rest of these details are indicative (and based on the POSIX standard).

If a filename starts with `.` this may need to be matched explicitly: for example `Sys.glob("*.RData")` may or may not match '`.RData`' but will not usually match '`.aa.RData`'. Note that this is platform-dependent: e.g. on Solaris `Sys.glob("*.*")` matches '`.`' and '`..`'.

`[` begins a character class. If the first character in `[...]` is not `!`, this is a character class which matches a single character against any of the characters specified. The class cannot be empty, so `]` can be included provided it is first. If the first character is `!`, the character class matches a single character which is *none* of the specified characters. Whether `.` in a character class matches a leading `.` in the filename is OS-dependent.

Character classes can include ranges such as `[A-Z]`: include `-` as a character by having it first or last in a class. (The interpretation of ranges should be locale-specific, so the example is not a good idea in an Estonian locale.)

One can remove the special meaning of `?`, `*` and `[` by preceding them by a backslash (except within a character class).

Value

A character vector of matched file paths. The order is system-specific (but in the order of the elements of `paths`): it is normally collated in either the current locale or in byte (ASCII) order; however, on Windows collation is in the order of Unicode points.

Directory errors are normally ignored, so the matches are to accessible file paths (but not necessarily accessible files).

See Also

`path.expand`.

Quotes for handling backslashes in character strings.

Examples

```
Sys.glob(file.path(R.home(), "library", "*", "R", "*.rdx"))
```

Sys.info *Extract System and User Information*

Description

Reports system and user information.

Usage

```
Sys.info()
```

Details

This uses POSIX or Windows system calls. Note that OS names (`sysname`) might not be what you expect: for example macOS identifies itself as '`Darwin`' and Solaris as '`SunOS`'.

`Sys.info()` returns details of the platform R is running on, whereas `R.version` gives details of the platform R was built on: the `release` and `version` may well be different.

Value

A character vector with fields

`sysname`	The operating system name.
`release`	The OS release.
`version`	The OS version.
`nodename`	A name by which the machine is known on the network (if any).
`machine`	A concise description of the hardware, often the CPU type.
`login`	The user's login name, or `"unknown"` if it cannot be ascertained.
`user`	The name of the real user ID, or `"unknown"` if it cannot be ascertained.

`effective_user` The name of the effective user ID, or `"unknown"` if it cannot be ascertained. This may differ from the real user in 'set-user-ID' processes.

On Unix-alike platforms: The first five fields come from the `uname(2)` system call. The login name comes from `getlogin(2)`, and the user names from `getpwuid(getuid())` and `getpwuid(geteuid())`.

On Windows: The last three fields give the same value.

Note

The meaning of `release` and `version` is system-dependent: on a Unix-alike they normally refer to the kernel. There, usually `release` contains a numeric version and `version` gives additional information. Examples for `release`:

```
"4.17.11-200.fc28.x86_64" # Linux (Fedora)
"3.16.0-5-amd64"          # Linux (Debian)
"17.7.0"                  # macOS 10.13.6
"5.11"                    # Solaris
```

There is no guarantee that the node or login or user names will be what you might reasonably expect. (In particular on some Linux distributions the login name is unknown from sessions with re-directed inputs.)

The use of alternatives such as `system("whoami")` is not portable: the POSIX command `system("id")` is much more portable on Unix-alikes, provided only the POSIX options '`-[Ggu][nr]`' are used (and not the many BSD and GNU extensions). `whoami` is equivalent to `id -un` (on Solaris, `/usr/xpg4/bin/id -un`).

Windows may report unexpected versions: there, see the help for

See Also

`.Platform`, and `R.version`. `sessionInfo()` gives a synopsis of both your system and the R session (and gives the OS version in a human-readable form).

Examples

```
Sys.info()
## An alternative (and probably better) way to get the login name on Unix
Sys.getenv("LOGNAME")
```

`Sys.localeconv` *Find Details of the Numerical and Monetary Representations in the Current Locale*

Description

Get details of the numerical and monetary representations in the current locale.

Usage

```
Sys.localeconv()
```

Details

Normally R is run without looking at the value of `LC_NUMERIC`, so the decimal point remains '.'. So the first three of these components will only be useful if you have set the locale category `LC_NUMERIC` using `Sys.setlocale` in the current R session (when R may not work correctly).

The monetary components will only be set to non-default values (see the 'Examples' section) if the `LC_MONETARY` category is set. It often is not set: set the examples for how to trigger setting it.

Value

A character vector with 18 named components. See your ISO C documentation for details of the meaning.

It is possible to compile R without support for locales, in which case the value will be `NULL`.

See Also

`Sys.setlocale` for ways to set locales.

Examples

```
Sys.localeconv()
## The results in the C locale are
##    decimal_point     thousands_sep          grouping   int_curr_symbol
##              "."                ""                ""                ""
##  currency_symbol mon_decimal_point mon_thousands_sep      mon_grouping
##                ""                ""                ""                ""
##    positive_sign     negative_sign   int_frac_digits       frac_digits
##                ""                ""             "127"             "127"
##    p_cs_precedes    p_sep_by_space     n_cs_precedes    n_sep_by_space
##             "127"             "127"             "127"             "127"
##      p_sign_posn       n_sign_posn
##             "127"             "127"

## Now try your default locale (which might be "C").
old <- Sys.getlocale()
## The category may not be set:
```

```
## the following may do so, but it might not be supported.
Sys.setlocale("LC_MONETARY", locale = "")
Sys.localeconv()
## or set an appropriate value yourself, e.g.
Sys.setlocale("LC_MONETARY", "de_AT")
Sys.localeconv()
Sys.setlocale(locale = old)

## Not run: read.table("foo", dec=Sys.localeconv()["decimal_point"])
```

`sys.parent` *Functions to Access the Function Call Stack*

Description

These functions provide access to environments ('frames' in S terminology) associated with functions further up the calling stack.

Usage

```
sys.call(which = 0)
sys.frame(which = 0)
sys.nframe()
sys.function(which = 0)
sys.parent(n = 1)

sys.calls()
sys.frames()
sys.parents()
sys.on.exit()
sys.status()
parent.frame(n = 1)
```

Arguments

`which`	the frame number if non-negative, the number of frames to go back if negative.
`n`	the number of generations to go back. (See the 'Details' section.)

Details

`.GlobalEnv` is given number 0 in the list of frames. Each subsequent function evaluation increases the frame stack by 1. The call, function definition and the environment for evaluation of that function are returned by `sys.call`, `sys.function` and `sys.frame` with the appropriate index.

`sys.call`, `sys.function` and `sys.frame` accept integer values for the argument `which`. Non-negative values of `which` are frame numbers starting from `.GlobalEnv` whereas negative values are counted back from the frame number of the current evaluation.

The parent frame of a function evaluation is the environment in which the function was called. It is not necessarily numbered one less than the frame number of the current evaluation, nor is it the environment within which the function was defined. `sys.parent` returns the number of the parent frame if `n` is 1 (the default), the grandparent if `n` is 2, and so on. See also the 'Note'.

`sys.nframe` returns an integer, the number of the current frame as described in the first paragraph.

`sys.calls` and `sys.frames` give a pairlist of all the active calls and frames, respectively, and `sys.parents` returns an integer vector of indices of the parent frames of each of those frames.

Notice that even though the `sys.`*xxx* functions (except `sys.status`) are interpreted, their contexts are not counted nor are they reported. There is no access to them.

`sys.status()` returns a list with components `sys.calls`, `sys.parents` and `sys.frames`, the results of calls to those three functions (which will include the call to `sys.status`: see the first example).

`sys.on.exit()` returns the expression stored for use by `on.exit` in the function currently being evaluated. (Note that this differs from S, which returns a list of expressions for the current frame and its parents.)

`parent.frame(n)` is a convenient shorthand for `sys.frame(sys.parent(n))` (implemented slightly more efficiently).

Value

`sys.call` returns a call, `sys.function` a function definition, and `sys.frame` and `parent.frame` return an environment.

For the other functions, see the 'Details' section.

Note

Strictly, `sys.parent` and `parent.frame` refer to the *context* of the parent interpreted function. So internal functions (which may or may not set contexts and so may or may not appear on the call stack) may not be counted, and S3 methods can also do surprising things.

As an effect of lazy evaluation, these functions look at the call stack at the time they are evaluated, not at the time they are called. Passing calls to them as function arguments is unlikely to be a good idea, but these functions still look at the call stack and count frames from the frame of the function evaluation from which they were called.

Hence, when these functions are called to provide default values for function arguments, they are evaluated in the evaluation of the called function and they count frames accordingly (see e.g. the `envir` argument of `eval`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Not `parent.frame`.)

See Also

`eval` for a usage of `sys.frame` and `parent.frame`.

Examples

```
require(utils)

## Note: the first two examples will give different results
## if run by example().
ff <- function(x) gg(x)
gg <- function(y) sys.status()
str(ff(1))

gg <- function(y) {
    ggg <- function() {
        cat("current frame is", sys.nframe(), "\n")
        cat("parents are", sys.parents(), "\n")
        print(sys.function(0)) # ggg
        print(sys.function(2)) # gg
    }
    if(y > 0) gg(y-1) else ggg()
}
gg(3)

t1 <- function() {
  aa <- "here"
  t2 <- function() {
    ## in frame 2 here
    cat("current frame is", sys.nframe(), "\n")
    str(sys.calls()) ## list with two components t1() and t2()
    cat("parents are frame numbers", sys.parents(), "\n") ## 0 1
    print(ls(envir = sys.frame(-1))) ## [1] "aa" "t2"
    invisible()
  }
  t2()
}
t1()

test.sys.on.exit <- function() {
  on.exit(print(1))
  ex <- sys.on.exit()
  str(ex)
  cat("exiting...\n")
}
test.sys.on.exit()
## gives 'language print(1)', prints 1 on exit

## An example where the parent is not the next frame up the stack
## since method dispatch uses a frame.
as.double.foo <- function(x)
{
    str(sys.calls())
    print(sys.frames())
    print(sys.parents())
    print(sys.frame(-1)); print(parent.frame())
    x
```

```
}
t2 <- function(x) as.double(x)
a <- structure(pi, class = "foo")
t2(a)
```

`Sys.readlink` *Read File Symbolic Links*

Description

Find out if a file path is a symbolic link, and if so what it is linked to, *via* the system call `readlink`.

Symbolic links are a POSIX concept, not implemented on Windows but for most filesystems on Unix-alikes.

Usage

```
Sys.readlink(paths)
```

Arguments

`paths`	character vector of file paths. Tilde expansion is done: see `path.expand`.

Value

A character vector of the same length as `paths`. The entries are the path of the file linked to, `""` if the path is not a symbolic link, and `NA` if there is an error (e.g., the path does not exist or cannot be converted to the native encoding).

On platforms without the `readlink` system call, all elements are `""`.

See Also

`file.symlink` for the creation of symbolic links (and their Windows analogues), `file.info`

Examples

```
##' To check if files (incl. directories) are symbolic links:
is.symlink <- function(paths) isTRUE(nzchar(Sys.readlink(paths), keepNA=TRUE))
## will return all FALSE when the platform has no `readlink` system call.
is.symlink("/foo/bar")
```

`Sys.setenv` *Set or Unset Environment Variables*

Description

`Sys.setenv` sets environment variables (for other processes called from within R or future calls to `Sys.getenv` from this R process).

`Sys.unsetenv` removes environment variables.

Usage

```
Sys.setenv(...)

Sys.unsetenv(x)
```

Arguments

`...`	named arguments with values coercible to a character string.
`x`	a character vector, or an object coercible to character.

Details

Non-standard R names must be quoted in `Sys.setenv`: see the examples. Most platforms (and POSIX) do not allow names containing `"="`. Windows does, but the facilities provided by R may not handle these correctly so they should be avoided. Most platforms allow setting an environment variable to `""`, but Windows does not and there `Sys.setenv(FOO = "")` unsets `FOO`.

There may be system-specific limits on the maximum length of the values of individual environment variables or of names+values of all environment variables.

Recent versions of Windows have a maximum length of 32,767 characters for a environment variable; however `cmd.exe` has a limit of 8192 characters for a command line, hence `set` can only set 8188.

Value

A logical vector, with elements being true if (un)setting the corresponding variable succeeded. (For `Sys.unsetenv` this includes attempting to remove a non-existent variable.)

Note

On Unix-alikes, if `Sys.unsetenv` is not supported, it will at least try to set the value of the environment variable to `""`, with a warning.

See Also

Sys.getenv, Startup for ways to set environment variables for the R session.

setwd for the working directory.

Sys.setlocale to set (and get) language locale variables, and notably Sys.setLanguage to set the LANGUAGE environment variable which is used for conditionMessage translations.

The help for 'environment variables' lists many of the environment variables used by R.

Examples

```
print(Sys.setenv(R_TEST = "testit", "A+C" = 123))  # `A+C` could also be used
Sys.getenv("R_TEST")
Sys.unsetenv("R_TEST") # on Unix-alike may warn and not succeed
Sys.getenv("R_TEST", unset = NA)
```

Sys.setFileTime *Set File Time*

Description

Uses system calls to set the times on a file or directory.

Usage

```
Sys.setFileTime(path, time)
```

Arguments

path	A character vector containing file or directory paths.
time	A date-time of class "POSIXct" or an object which can be coerced to one. Fractions of a second may be ignored. Recycled along paths.

Details

This attempts sets the file time to the value specified.

On a Unix-alike it uses the system call utimensat if that is available, otherwise utimes or utime. On a POSIX file system it sets both the last-access and modification times. Fractional seconds will set as from R 3.4.0 on OSes with the requisite system calls and suitable filesystems.

On Windows it uses the system call SetFileTime to set the 'last write time'. Some Windows file systems only record the time at a resolution of two seconds.

Sys.setFileTime has been vectorized in R 3.6.0. Earlier versions of R required path and time to be vectors of length one.

Value

A logical vector indicating if the operation succeeded for each of the files and directories attempted, returned invisibly.

`Sys.sleep` *Suspend Execution for a Time Interval*

Description

Suspend execution of R expressions for a specified time interval.

Usage

```
Sys.sleep(time)
```

Arguments

`time`	The time interval to suspend execution for, in seconds.

Details

Using this function allows R to temporarily be given very low priority and hence not to interfere with more important foreground tasks. A typical use is to allow a process launched from R to set itself up and read its input files before R execution is resumed.

The intention is that this function suspends execution of R expressions but wakes the process up often enough to respond to GUI events, typically every half second. It can be interrupted (e.g. by '`Ctrl-C`' or '`Esc`' at the R console).

There is no guarantee that the process will sleep for the whole of the specified interval (sleep might be interrupted), and it may well take slightly longer in real time to resume execution.

`time` must be non-negative (and not `NA` nor `NaN`): `Inf` is allowed (and might be appropriate if the intention is to wait indefinitely for an interrupt). The resolution of the time interval is system-dependent, but will normally be 20ms or better. (On modern Unix-alikes it will be better than 1ms.)

Value

Invisible `NULL`.

Note

Despite its name, this is not currently implemented using the `sleep` system call (although on Windows it does make use of `Sleep`).

Examples

```
testit <- function(x)
{
    p1 <- proc.time()
    Sys.sleep(x)
    proc.time() - p1 # The cpu usage should be negligible
}
testit(3.7)
```

`sys.source` *Parse and Evaluate Expressions from a File*

Description

Parses expressions in the given file, and then successively evaluates them in the specified environment.

Usage

```
sys.source(file, envir = baseenv(), chdir = FALSE,
           keep.source = getOption("keep.source.pkgs"),
           keep.parse.data = getOption("keep.parse.data.pkgs"),
           toplevel.env = as.environment(envir))
```

Arguments

`file`	a character string naming the file to be read from.
`envir`	an R object specifying the environment in which the expressions are to be evaluated. May also be a list or an integer. The default `baseenv()` corresponds to evaluation in the base environment. This is probably not what you want; you should typically supply an explicit `envir` argument, see the 'Note'.
`chdir`	logical; if `TRUE`, the R working directory is changed to the directory containing `file` for evaluating.
`keep.source`	logical. If `TRUE`, functions keep their source including comments, see `options(keep.source = *)` for more details.
`keep.parse.data`	logical. If `TRUE` and `keep.source` is also `TRUE`, functions keep parse data with their source, see `options(keep.parse.data = *)` for more details.
`toplevel.env`	an R environment to be used as top level while evaluating the expressions. This argument is useful for frameworks running package tests; the default should be used in other cases.

Details

For large files, `keep.source = FALSE` may save quite a bit of memory. Disabling only parse data via `keep.parse.data = FALSE` can already save a lot.

Note on `envir`

In order for the code being evaluated to use the correct environment (for example, in global assignments), source code in packages should call `topenv()`, which will return the namespace, if any, the environment set up by `sys.source`, or the global environment if a saved image is being used.

See Also

`source`, and `loadNamespace` which is called from `library(.)` and uses `sys.source(.)`.

Examples

```
## a simple way to put some objects in an environment
## high on the search path
tmp <- tempfile()
writeLines("aaa <- pi", tmp)
env <- attach(NULL, name = "myenv")
sys.source(tmp, env)
unlink(tmp)
search()
aaa
detach("myenv")
```

`Sys.time` *Get Current Date and Time*

Description

`Sys.time` and `Sys.Date` returns the system's idea of the current date with and without time.

Usage

```
Sys.time()
Sys.Date()
```

Details

`Sys.time` returns an absolute date-time value which can be converted to various time zones and may return different days.

`Sys.Date` returns the current day in the current time zone.

Value

`Sys.time` returns an object of class `"POSIXct"` (see DateTimeClasses). On almost all systems it will have sub-second accuracy, possibly microseconds or better. On Windows it increments in clock ticks (usually 1/60 of a second) reported to millisecond accuracy.

`Sys.Date` returns an object of class `"Date"` (see Date).

Note

`Sys.time` may return fractional seconds, but they are ignored by the default conversions (e.g., printing) for class `"POSIXct"`. See the examples and `format.POSIXct` for ways to reveal them.

See Also

`date` for the system time in a fixed-format character string.

`Sys.timezone`.

`system.time` for measuring elapsed/CPU time of expressions.

Examples

```
Sys.time()
## print with possibly greater accuracy:
op <- options(digits.secs = 6)
Sys.time()
options(op)

## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y")

Sys.Date()
```

`Sys.which` *Find Full Paths to Executables*

Description

This is an interface to the system command `which`, or to an emulation on Windows.

Usage

```
Sys.which(names)
```

Arguments

`names`	Character vector of names or paths of possible executables.

Details

The system command `which` reports on the full path names of an executable (including an executable script) as would be executed by a shell, accepting either absolute paths or looking on the path.

On Windows an 'executable' is a file with extension '`.exe`', '`.com`', '`.cmd`' or '`.bat`'. Such files need not actually be executable, but they are what `system` tries.

On a Unix-alike the full path to `which` (usually '`/usr/bin/which`') is found when R is installed.

Value

A character vector of the same length as `names`, named by `names`. The elements are either the full path to the executable or some indication that no executable of that name was found. Typically the indication is `""`, but this does depend on the OS (and the known exceptions are changed to `""`). Missing values in `names` have missing return values.

On Windows the paths will be short paths (8+3 components, no spaces) with \ as the path delimiter.

Note

Except on Windows this calls the system command `which`: since that is not part of e.g. the POSIX standards, exactly what it does is OS-dependent. It will usually do tilde-expansion and it may make use of `csh` aliases.

Examples

```
## the first two are likely to exist everywhere
## texi2dvi exists on most Unix-alikes and under MiKTeX
Sys.which(c("ftp", "ping", "texi2dvi", "this-does-not-exist"))
```

---

system *Invoke a System Command*

Description

`system` invokes the OS command specified by `command`.

Usage

```
system(command, intern = FALSE,
       ignore.stdout = FALSE, ignore.stderr = FALSE,
       wait = TRUE, input = NULL, show.output.on.console = TRUE,
       minimized = FALSE, invisible = TRUE, timeout = 0,
       receive.console.signals = wait)
```

Arguments

`command`	the system command to be invoked, as a character string.
`intern`	a logical (not `NA`) which indicates whether to capture the output of the command as an R character vector.
`ignore.stdout, ignore.stderr`	a logical (not `NA`) indicating whether messages written to '`stdout`' or '`stderr`' should be ignored.
`wait`	a logical (not `NA`) indicating whether the R interpreter should wait for the command to finish, or run it asynchronously. This will be ignored (and the interpreter will always wait) if `intern = TRUE`. When running the command asynchronously, no output will be displayed on the `Rgui` console in Windows (it will be dropped, instead).
`input`	if a character vector is supplied, this is copied one string per line to a temporary file, and the standard input of `command` is redirected to the file.
`timeout`	timeout in seconds, ignored if 0. This is a limit for the elapsed time running `command` in a separate process. Fractions of seconds are ignored.

`receive.console.signals`
: a logical (not `NA`) indicating whether the command should receive events from the terminal/console that R runs from, particularly whether it should be interrupted by `Ctrl-C`. This will be ignored and events will always be received when `intern = TRUE` or `wait = TRUE`.

`show.output.on.console`, `minimized`, `invisible`
: arguments that are accepted on Windows but ignored on this platform, with a warning.

Details

This interface has become rather complicated over the years: see `system2` for a more portable and flexible interface which is recommended for new code.

`command` is parsed as a command plus arguments separated by spaces. So if the path to the command (or a single argument such as a file path) contains spaces, it must be quoted e.g. by `shQuote`. Unix-alikes pass the command line to a shell (normally '`/bin/sh`', and POSIX requires that shell), so `command` can be anything the shell regards as executable, including shell scripts, and it can contain multiple commands separated by `;`.

On Windows, `system` does not use a shell and there is a separate function `shell` which passes command lines to a shell.

If `intern` is `TRUE` then `popen` is used to invoke the command and the output collected, line by line, into an R `character` vector. If `intern` is `FALSE` then the C function `system` is used to invoke the command.

`wait` is implemented by appending `&` to the command: this is in principle shell-dependent, but required by POSIX and so widely supported.

When `timeout` is non-zero, the command is terminated after the given number of seconds. The termination works for typical commands, but is not guaranteed: it is possible to write a program that would keep running after the time is out. Timeouts can only be set with `wait = TRUE`.

Timeouts cannot be used with interactive commands: the command is run with standard input redirected from '`/dev/null`' and it must not modify terminal settings. As long as tty `tostop` option is disabled, which it usually is by default, the executed command may write to standard output and standard error. One cannot rely on that the execution time of the child processes will be included into `user.child` and `sys.child` element of `proc_time` returned by `proc.time`. For the time to be included, all child processes have to be waited for by their parents, which has to be implemented in the parent applications.

The ordering of arguments after the first two has changed from time to time: it is recommended to name all arguments after the first.

There are many pitfalls in using `system` to ascertain if a command can be run — `Sys.which` is more suitable.

`receive.console.signals = TRUE` is useful when running asynchronous processes (using `wait = FALSE`) to implement a synchronous operation. In all other cases it is recommended to use the default.

Value

If `intern = TRUE`, a character vector giving the output of the command, one line per character string. (Output lines of more than 8095 bytes will be split on some systems.) If the command could not be

run an R error is generated. If command runs but gives a non-zero exit status this will be reported with a warning and in the attribute "status" of the result: an attribute "errmsg" may also be available.

If intern = FALSE, the return value is an error code (0 for success), given the invisible attribute (so needs to be printed explicitly). If the command could not be run for any reason, the value is 127 and a warning is issued (as from R 3.5.0). Otherwise if wait = TRUE the value is the exit status returned by the command, and if wait = FALSE it is 0 (the conventional success value).

If the command times out, a warning is reported and the exit status is 124.

Stdout and stderr

For command-line R, error messages written to 'stderr' will be sent to the terminal unless ignore.stderr = TRUE. They can be captured (in the most likely shells) by

```
system("some command 2>&1", intern = TRUE)
```

For GUIs, what happens to output sent to 'stdout' or 'stderr' if intern = FALSE is interface-specific, and it is unsafe to assume that such messages will appear on a GUI console (they do on the macOS GUI's console, but not on some others).

Differences between Unix and Windows

How processes are launched differs fundamentally between Windows and Unix-alike operating systems, as do the higher-level OS functions on which this R function is built. So it should not be surprising that there are many differences between OSes in how system behaves. For the benefit of programmers, the more important ones are summarized in this section.

- The most important difference is that on a Unix-alike system launches a shell which then runs command. On Windows the command is run directly – use shell for an interface which runs command *via* a shell (by default the Windows shell cmd.exe, which has many differences from a POSIX shell).
  This means that it cannot be assumed that redirection or piping will work in system (redirection sometimes does, but we have seen cases where it stopped working after a Windows security patch), and system2 (or shell) must be used on Windows.
- What happens to stdout and stderr when not captured depends on how R is running: Windows batch commands behave like a Unix-alike, but from the Windows GUI they are generally lost. system(intern = TRUE) captures 'stderr' when run from the Windows GUI console unless ignore.stderr = TRUE.
- The behaviour on error is different in subtle ways (and has differed between R versions).
- The quoting conventions for command differ, but shQuote is a portable interface.
- Arguments show.output.on.console, minimized, invisible only do something on Windows (and are most relevant to Rgui there).

See Also

man system and man sh for how this is implemented on the OS in use.

.Platform for platform-specific variables.

pipe to set up a pipe connection.

Examples

```
# list all files in the current directory using the -F flag
## Not run: system("ls -F")

# t1 is a character vector, each element giving a line of output from who
# (if the platform has who)
t1 <- try(system("who", intern = TRUE))

try(system("ls fizzlipuzzli", intern = TRUE, ignore.stderr = TRUE))
# zero-length result since file does not exist, and will give warning.
```

`system.file` *Find Names of R System Files*

Description

Finds the full file names of files in packages etc.

Usage

```
system.file(..., package = "base", lib.loc = NULL,
            mustWork = FALSE)
```

Arguments

`...`	character vectors, specifying subdirectory and file(s) within some package. The default, none, returns the root of the package. Wildcards are not supported.
`package`	a character string with the name of a single package. An error occurs if more than one package name is given.
`lib.loc`	a character vector with path names of R libraries. See 'Details' for the meaning of the default value of `NULL`.
`mustWork`	logical. If `TRUE`, an error is given if there are no matching files.

Details

This checks the existence of the specified files with `file.exists`. So file paths are only returned if there are sufficient permissions to establish their existence.

The unnamed arguments in `...` are usually character strings, but if character vectors they are recycled to the same length.

This uses `find.package` to find the package, and hence with the default `lib.loc = NULL` looks first for attached packages then in each library listed in `.libPaths()`. Note that if a namespace is loaded but the package is not attached, this will look only on `.libPaths()`.

Value

A character vector of positive length, containing the file paths that matched . . . , or the empty string, `""`, if none matched (unless `mustWork = TRUE`).

If matching the root of a package, there is no trailing separator.

`system.file()` with no arguments gives the root of the **base** package.

See Also

`R.home` for the root directory of the R installation, `list.files`.

`Sys.glob` to find paths via wildcards.

Examples

```
system.file()                  # The root of the 'base' package
system.file(package = "stats") # The root of package 'stats'
system.file("INDEX")
system.file("help", "AnIndex", package = "splines")
```

---

system.time *CPU Time Used*

Description

Return CPU (and other) times that `expr` used.

Usage

```
system.time(expr, gcFirst = TRUE)
```

Arguments

`expr`	Valid R expression to be timed.
`gcFirst`	Logical - should a garbage collection be performed immediately before the timing? Default is `TRUE`.

Details

`system.time` calls the function `proc.time`, evaluates `expr`, and then calls `proc.time` once more, returning the difference between the two `proc.time` calls.

`unix.time` has been an alias of `system.time`, for compatibility with S, has been deprecated in 2016 and finally became defunct in 2022.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When `gcFirst` is `TRUE` a garbage collection (`gc`) will be performed immediately before the evaluation of `expr`. This will usually produce more consistent timings.

Value

A object of class "proc_time": see proc.time for details.

See Also

proc.time, time which is for time series.

setTimeLimit to limit the (CPU/elapsed) time R is allowed to use.

Sys.time to get the current date & time.

Examples

```
require(stats)
system.time(for(i in 1:100) mad(runif(1000)))
## Not run:
exT <- function(n = 10000) {
  # Purpose: Test if system.time works ok;   n: loop size
  system.time(for(i in 1:n) x <- mean(rt(1000, df = 4)))
}
#-- Try to interrupt one of the following (using Ctrl-C / Escape):
exT()                #- about 4 secs on a 2.5GHz Xeon
system.time(exT())   #~ +/- same

## End(Not run)
```

---

system2 *Invoke a System Command*

Description

system2 invokes the OS command specified by command.

Usage

```
system2(command, args = character(),
        stdout = "", stderr = "", stdin = "", input = NULL,
        env = character(), wait = TRUE,
        minimized = FALSE, invisible = TRUE, timeout = 0,
        receive.console.signals = wait)
```

Arguments

command	the system command to be invoked, as a character string.
args	a character vector of arguments to command. The arguments have to be quoted e.g. by shQuote in case they contain space or other special characters (a double quote or backslash on Windows, shell-specific special characters on Unix).

`stdout`, `stderr` where output to 'stdout' or 'stderr' should be sent. Possible values are `""`, to the R console (the default), `NULL` or `FALSE` (discard output), `TRUE` (capture the output in a character vector) or a character string naming a file.

`stdin` should input be diverted? `""` means the default, alternatively a character string naming a file. Ignored if `input` is supplied.

`input` if a character vector is supplied, this is copied one string per line to a temporary file, and the standard input of `command` is redirected to the file.

`env` character vector of name=value strings to set environment variables.

`wait` a logical (not `NA`) indicating whether the R interpreter should wait for the command to finish, or run it asynchronously. This will be ignored (and the interpreter will always wait) if `stdout = TRUE` or `stderr = TRUE`. When running the command asynchronously, no output will be displayed on the `Rgui` console in Windows (it will be dropped, instead).

`timeout` timeout in seconds, ignored if 0. This is a limit for the elapsed time running `command` in a separate process. Fractions of seconds are ignored.

`receive.console.signals`
a logical (not `NA`) indicating whether the command should receive events from the terminal/console that R runs from, particularly whether it should be interrupted by `Ctrl-C`. This will be ignored and events will always be received when `intern = TRUE` or `wait = TRUE`.

`minimized`, `invisible`
arguments that are accepted on Windows but ignored on this platform, with a warning.

Details

Unlike `system`, `command` is always quoted by `shQuote`, so it must be a single command without arguments.

For details of how `command` is found see `system`.

On Windows, `env` is only supported for commands such as `R` and `make` which accept environment variables on their command line.

Some Unix commands (such as some implementations of `ls`) change their output if they consider it to be piped or redirected: `stdout = TRUE` uses a pipe whereas `stdout = "some_file_name"` uses redirection.

Because of the way it is implemented, on a Unix-alike `stderr = TRUE` implies `stdout = TRUE`: a warning is given if this is not what was specified.

When `timeout` is non-zero, the command is terminated after the given number of seconds. The termination works for typical commands, but is not guaranteed: it is possible to write a program that would keep running after the time is out. Timeouts can only be set with `wait = TRUE`.

Timeouts cannot be used with interactive commands: the command is run with standard input redirected from `/dev/null` and it must not modify terminal settings. As long as tty `tostop` option is disabled, which it usually is by default, the executed command may write to standard output and standard error.

`receive.console.signals = TRUE` is useful when running asynchronous processes (using `wait = FALSE`) to implement a synchronous operation. In all other cases it is recommended to use the default.

Value

If stdout = TRUE or stderr = TRUE, a character vector giving the output of the command, one line per character string. (Output lines of more than 8095 bytes will be split.) If the command could not be run an R error is generated. If command runs but gives a non-zero exit status this will be reported with a warning and in the attribute "status" of the result: an attribute "errmsg" may also be available.

In other cases, the return value is an error code (0 for success), given the invisible attribute (so needs to be printed explicitly). If the command could not be run for any reason, the value is 127 and a warning is issued (as from R 3.5.0). Otherwise if wait = TRUE the value is the exit status returned by the command, and if wait = FALSE it is 0 (the conventional success value).

If the command times out, a warning is issued and the exit status is 124.

Note

system2 is a more portable and flexible interface than system. It allows redirection of output without needing to invoke a shell on Windows, a portable way to set environment variables for the execution of command, and finer control over the redirection of stdout and stderr. Conversely, system (and shell on Windows) allows the invocation of arbitrary command lines.

There is no guarantee that if stdout and stderr are both TRUE or the same file that the two streams will be interleaved in order. This depends on both the buffering used by the command and the OS.

See Also

system.

---

t *Matrix Transpose*

Description

Given a matrix or data.frame x, t returns the transpose of x.

Usage

```
t(x)
```

Arguments

x	a matrix or data frame, typically.

Details

This is a generic function for which methods can be written. The description here applies to the default and "data.frame" methods.

A data frame is first coerced to a matrix: see as.matrix. When x is a vector, it is treated as a column, i.e., the result is a 1-row matrix.

Value

A matrix, with `dim` and `dimnames` constructed appropriately from those of `x`, and other attributes except names copied across.

Note

The *conjugate* transpose of a complex matrix A, denoted A^H or A^*, is computed as `Conj(t(A))`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`aperm` for permuting the dimensions of arrays.

Examples

```
a <- matrix(1:30, 5, 6)
ta <- t(a) ##-- i.e.,  a[i, j] == ta[j, i] for all i,j :
for(j in seq(ncol(a)))
  if(! all(a[, j] == ta[j, ])) stop("wrong transpose")
```

---

table *Cross Tabulation and Table Creation*

Description

`table` uses cross-classifying factors to build a contingency table of the counts at each combination of factor levels.

Usage

```
table(...,
      exclude = if (useNA == "no") c(NA, NaN),
      useNA = c("no", "ifany", "always"),
      dnn = list.names(...), deparse.level = 1)

as.table(x, ...)
is.table(x)

## S3 method for class 'table'
as.data.frame(x, row.names = NULL, ...,
              responseName = "Freq", stringsAsFactors = TRUE,
              sep = "", base = list(LETTERS))
```

Arguments

...	one or more objects which can be interpreted as factors (including numbers or character strings), or a list (such as a data frame) whose components can be so interpreted. (For as.table, arguments passed to specific methods; for as.data.frame, unused.)
exclude	levels to remove for all factors in .... If it does not contain NA and useNA is not specified, it implies useNA = "ifany". See 'Details' for its interpretation for non-factor arguments.
useNA	whether to include NA values in the table. See 'Details'. Can be abbreviated.
dnn	the names to be given to the dimensions in the result (the *dimnames names*).
deparse.level	controls how the default dnn is constructed. See 'Details'.
x	an arbitrary R object, or an object inheriting from class "table" for the as.data.frame method. Note that as.data.frame.table(x, *) may be called explicitly for non-table x for "reshaping" arrays.
row.names	a character vector giving the row names for the data frame.
responseName	the name to be used for the column of table entries, usually counts.
stringsAsFactors	logical: should the classifying factors be returned as factors (the default) or character vectors?
sep, base	passed to provideDimnames.

Details

If the argument dnn is not supplied, the internal function list.names is called to compute the 'dimname names' as follows: If ... is one list with its own names(), these names are used. Otherwise, if the arguments in ... are named, those names are used. For the remaining arguments, deparse.level = 0 gives an empty name, deparse.level = 1 uses the supplied argument if it is a symbol, and deparse.level = 2 will deparse the argument.

Only when exclude is specified (i.e., not by default) and non-empty, will table potentially drop levels of factor arguments.

useNA controls if the table includes counts of NA values: the allowed values correspond to never ("no"), only if the count is positive ("ifany") and even for zero counts ("always"). Note the somewhat "pathological" case of two different kinds of NAs which are treated differently, depending on both useNA and exclude, see d.patho in the 'Examples:' below.

Both exclude and useNA operate on an "all or none" basis. If you want to control the dimensions of a multiway table separately, modify each argument using factor or addNA.

Non-factor arguments a are coerced via factor(a, exclude=exclude). Since R 3.4.0, care is taken *not* to count the excluded values (where they were included in the NA count, previously).

The summary method for class "table" (used for objects created by table or xtabs) which gives basic information and performs a chi-squared test for independence of factors (note that the function chisq.test currently only handles 2-d tables).

Value

`table()` returns a *contingency table*, an object of class `"table"`, an array of integer values. Note that unlike S the result is always an `array`, a 1D array if one factor is given.

`as.table` and `is.table` coerce to and test for contingency table, respectively.

The `as.data.frame` method for objects inheriting from class `"table"` can be used to convert the array-based representation of a contingency table to a data frame containing the classifying factors and the corresponding entries (the latter as component named by `responseName`). This is the inverse of `xtabs`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`tabulate` is the underlying function and allows finer control.

Use `ftable` for printing (and more) of multidimensional tables. `margin.table`, `prop.table`, `addmargins`.

`addNA` for constructing factors with `NA` as a level.

`xtabs` for cross tabulation of data frames with a formula interface.

Examples

```
require(stats) # for rpois and xtabs
## Simple frequency distribution
table(rpois(100, 5))
## Check the design:
with(warpbreaks, table(wool, tension))
table(state.division, state.region)

# simple two-way contingency table
with(airquality, table(cut(Temp, quantile(Temp)), Month))

a <- letters[1:3]
table(a, sample(a))                   # dnn is c("a", "")
table(a, sample(a), dnn = NULL)       # dimnames() have no names
table(a, sample(a), deparse.level = 0) # dnn is c("", "")
table(a, sample(a), deparse.level = 2) # dnn is c("a", "sample(a)")

## xtabs() <-> as.data.frame.table() :
UCBAdmissions ## already a contingency table
DF <- as.data.frame(UCBAdmissions)
class(tab <- xtabs(Freq ~ ., DF)) # xtabs & table
## tab *is* "the same" as the original table:
all(tab == UCBAdmissions)
all.equal(dimnames(tab), dimnames(UCBAdmissions))

a <- rep(c(NA, 1/0:3), 10)
```

```
table(a)                 # does not report NA's
table(a, exclude = NULL) # reports NA's
b <- factor(rep(c("A","B","C"), 10))
table(b)
table(b, exclude = "B")
d <- factor(rep(c("A","B","C"), 10), levels = c("A","B","C","D","E"))
table(d, exclude = "B")
print(table(b, d), zero.print = ".")

## NA counting:
is.na(d) <- 3:4
d. <- addNA(d)
d.[1:7]
table(d.) # ", exclude = NULL" is not needed
## i.e., if you want to count the NA's of 'd', use
table(d, useNA = "ifany")

## "pathological" case:
d.patho <- addNA(c(1,NA,1:2,1:3))[-7]; is.na(d.patho) <- 3:4
d.patho
## just 3 consecutive NA's ? --- well, have *two* kinds of NAs here :
as.integer(d.patho) # 1 4 NA NA 1 2
##
## In R >= 3.4.0, table() allows to differentiate:
table(d.patho)                  # counts the "unusual" NA
table(d.patho, useNA = "ifany")  # counts all three
table(d.patho, exclude = NULL)   #  (ditto)
table(d.patho, exclude = NA)     # counts none

## Two-way tables with NA counts. The 3rd variant is absurd, but shows
## something that cannot be done using exclude or useNA.
with(airquality,
   table(OzHi = Ozone > 80, Month, useNA = "ifany"))
with(airquality,
   table(OzHi = Ozone > 80, Month, useNA = "always"))
with(airquality,
   table(OzHi = Ozone > 80, addNA(Month)))
```

`tabulate` *Tabulation for Vectors*

Description

`tabulate` takes the integer-valued vector `bin` and counts the number of times each integer occurs in it.

Usage

```
tabulate(bin, nbins = max(1, bin, na.rm = TRUE))
```

Arguments

`bin`	a numeric vector (of positive integers), or a factor. Long vectors are supported.
`nbins`	the number of bins to be used.

Details

`tabulate` is the workhorse for the `table` function.

If `bin` is a factor, its internal integer representation is tabulated.

If the elements of `bin` are numeric but not integers, they are truncated by `as.integer`.

Value

An integer valued `integer` or `double` vector (without names). There is a bin for each of the values `1, ..., nbins`; values outside that range and `NA`s are (silently) ignored.

On 64-bit platforms `bin` can have 2^{31} or more elements (i.e., `length(bin) > .Machine$integer.max`), and hence a count could exceed the maximum integer. For this reason, the return value is of type double for such long `bin` vectors.

See Also

`table`, `factor`.

Examples

```
tabulate(c(2,3,5))
tabulate(c(2,3,3,5), nbins = 10)
tabulate(c(-2,0,2,3,3,5))  # -2 and 0 are ignored
tabulate(c(-2,0,2,3,3,5), nbins = 3)
tabulate(factor(letters[1:10]))
```

`Tailcall` `Tailcall` *and* `Exec`

Description

`Tailcall` and `Exec` allow writing more stack-space-efficient recursive functions in R.

Usage

```
Tailcall(FUN, ...)
Exec(expr, envir)
```

Arguments

FUN	a function or a non-empty character string naming the function to be called.
...	all the arguments to be passed.
expr	a call expression.
envir	environment for evaluating expr; default is the environment from which Exec is called.

Details

Tailcall evaluates a call to FUN with arguments ... in the current environment, and Exec evaluates the call expr in environment envir. If a Tailcall or Exec expression appears in tail position in an R function, and if there are no on.exit expressions set, then the evaluation context of the new calls replaces the currently executing call context with a new one. If the requirements for context re-use are not met, then evaluation proceeds in the standard way adding another context to the stack.

Using Tailcall it is possible to define tail-recursive functions that do not grow the evaluation stack. Exec can be used to simplify the call stack for functions that create and then evaluate an expression.

Because of lazy evaluation of arguments in R it may be necessary to force evaluation of some arguments to avoid accumulating deferred evaluations.

This *tail call optimization* has the advantage of not growing the call stack and permitting arbitrarily deep tail recursions. It does also mean that stack traces produced by traceback or sys.calls will only show the call specified by Tailcall or Exec, not the previous call whose stack entry has been replaced.

Note

Tailcall and Exec are experimental and may be changed or dropped in future released versions of R.

See Also

Recall and force.

Examples

```
## tail-recursive log10-factorial
lfact <- function(n) {
    lfact_iter <- function(val, n) {
        if (n <= 0)
            val
        else {
            val <- val + log10(n) # forces val
            Tailcall(lfact_iter, val, n - 1)
        }
    }
    lfact_iter(0, n)
}
10 ^ lfact(3)
lfact(100000)
```

```
## simplified variant of do.call using Exec:
docall <- function (what, args, quote = FALSE) {
    if (!is.list(args))
        stop("second argument must be a list")
    if (quote)
        args <- lapply(args, enquote)
    Exec(as.call(c(list(substitute(what)), args)), parent.frame())
}
## the call stack does not contain the call to docall:
docall(function() sys.calls(), list()) |>
    Find(function(x) identical(x[[1]], quote(docall)), x = _)
## contrast to do.call:
do.call(function(x) sys.calls(), list()) |>
    Find(function(x) identical(x[[1]], quote(do.call)), x = _)
```

---

tapply *Apply a Function Over a Ragged Array*

Description

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values or data rows given by a unique combination of the levels of certain factors.

Usage

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

Arguments

X	an R object for which a split method exists. Typically vector-like, allowing subsetting with [, or a data frame.
INDEX	a list of one or more factors, each of same length as X. The elements are coerced to factors by as.factor. Can also be a formula, which is useful if X is a data frame; see the f argument in split for interpretation.
FUN	a function (or name of a function) to be applied, or NULL. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces.
...	optional arguments to FUN: the Note section.
default	(only in the case of simplification to an array) the value with which the array is initialized as array(default, dim = ..). Before R 3.4.0, this was hard coded to array()'s default NA. If it is NA (the default), the missing value of the answer type, e.g. NA_real_, is chosen (as.raw(0) for "raw"). In a numerical case, it may be set, e.g., to FUN(integer(0)), e.g., in the case of FUN = sum to 0 or 0L.
simplify	logical; if FALSE, tapply always returns an array of mode "list"; in other words, a list with a dim attribute. If TRUE (the default), then if FUN always returns a scalar, tapply returns an array with the mode of the scalar.

Details

If FUN is not NULL, it is passed to match.fun, and hence it can be a function or a symbol or character string naming a function.

Value

When FUN is present, tapply calls FUN for each cell that has any data in it. If FUN returns a single atomic value for each such cell (e.g., functions mean or var) and when simplify is TRUE, tapply returns a multi-way array containing the values, and NA for the empty cells. The array has the same number of dimensions as INDEX has components; the number of levels in a dimension is the number of levels (nlevels()) in the corresponding component of INDEX. Note that if the return value has a class (e.g., an object of class "Date") the class is discarded.

simplify = TRUE always returns an array, possibly 1-dimensional.

If FUN does not return a single atomic value, tapply returns an array of mode list whose components are the values of the individual calls to FUN, i.e., the result is a list with a dim attribute.

When there is an array answer, its dimnames are named by the names of INDEX and are based on the levels of the grouping factors (possibly after coercion).

For a list result, the elements corresponding to empty cells are NULL.

The array2DF function can be used to convert the array returned by tapply into a data frame, which may be more convenient for further analysis.

Note

Optional arguments to FUN supplied by the ... argument are not divided into cells. It is therefore inappropriate for FUN to expect additional arguments with the same length as X.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

the convenience functions by and aggregate (using tapply); apply, lapply with its versions sapply and mapply.

array2DF to convert the result into a data frame.

Examples

```
require(stats)
groups <- as.factor(rbinom(32, n = 5, prob = 0.4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

## contingency table from data.frame : array with named dimnames
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
tapply(warpbreaks$breaks, warpbreaks[, 3, drop = FALSE], sum)
```

```
n <- 17; fac <- factor(rep_len(1:3, n), levels = 1:5)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, default = 0) # maybe more desirable
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)
tapply(1:n, fac, length) ## NA's
tapply(1:n, fac, length, default = 0) # == table(fac)

## example of ... argument: find quarterly means
tapply(presidents, cycle(presidents), mean, na.rm = TRUE)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
tapply(1:3, ind) #-> the split vector
tapply(1:3, ind, sum)

## Some assertions (not held by all patch propsals):
nq <- names(quantile(1:5))
stopifnot(
  identical(tapply(1:3, ind), c(1L, 2L, 4L)),
  identical(tapply(1:3, ind, sum),
            matrix(c(1L, 2L, NA, 3L), 2, dimnames = list(c("1", "2"), c("A", "B")))),
  identical(tapply(1:n, fac, quantile)[-1],
            array(list(`2` = structure(c(2, 5.75, 9.5, 13.25, 17), names = nq),
                 `3` = structure(c(3, 6, 9, 12, 15), names = nq),
                 `4` = NULL, `5` = NULL), dim=4, dimnames=list(as.character(2:5)))))
```

`taskCallback` *Add or Remove a Top-Level Task Callback*

Description

`addTaskCallback` registers an R function that is to be called each time a top-level task is completed.

`removeTaskCallback` un-registers a function that was registered earlier via `addTaskCallback`.

These provide low-level access to the internal/native mechanism for managing task-completion actions. One can use `taskCallbackManager` at the R-language level to manage R functions that are called at the completion of each task. This is easier and more direct.

Usage

```
addTaskCallback(f, data = NULL, name = character())
removeTaskCallback(id)
```

Arguments

f	the function that is to be invoked each time a top-level task is successfully completed. This is called with 5 or 4 arguments depending on whether data is specified or not, respectively. The return value should be a logical value indicating whether to keep the callback in the list of active callbacks or discard it.
data	if specified, this is the 5-th argument in the call to the callback function f.
id	a string or an integer identifying the element in the internal callback list to be removed. Integer indices are 1-based, i.e the first element is 1. The names of currently registered handlers is available using getTaskCallbackNames and is also returned in a call to addTaskCallback.
name	character: names to be used.

Details

Top-level tasks are individual expressions rather than entire lines of input. Thus an input line of the form expression1 ; expression2 will give rise to 2 top-level tasks.

A top-level task callback is called with the expression for the top-level task, the result of the top-level task, a logical value indicating whether it was successfully completed or not (always TRUE at present), and a logical value indicating whether the result was printed or not. If the data argument was specified in the call to addTaskCallback, that value is given as the fifth argument.

The callback function should return a logical value. If the value is FALSE, the callback is removed from the task list and will not be called again by this mechanism. If the function returns TRUE, it is kept in the list and will be called on the completion of the next top-level task.

Value

addTaskCallback returns an integer value giving the position in the list of task callbacks that this new callback occupies. This is only the current position of the callback. It can be used to remove the entry as long as no other values are removed from earlier positions in the list first.

removeTaskCallback returns a logical value indicating whether the specified element was removed. This can fail (i.e., return FALSE) if an incorrect name or index is given that does not correspond to the name or position of an element in the list.

Note

There is also C-level access to top-level task callbacks to allow C routines rather than R functions be used.

See Also

getTaskCallbackNames taskCallbackManager https://developer.r-project.org/TaskHandlers.pdf

Examples

```
times <- function(total = 3, str = "Task a") {
  ctr <- 0
  function(expr, value, ok, visible) {
    ctr <<- ctr + 1
    cat(str, ctr, "\n")
    keep.me <- (ctr < total)
    if (!keep.me)
      cat("handler removing itself\n")

    # return
    keep.me
  }
}

# add the callback that will work for
# 4 top-level tasks and then remove itself.
n <- addTaskCallback(times(4))

# now remove it, assuming it is still first in the list.
removeTaskCallback(n)

## See how the handler is called every time till "self destruction":

addTaskCallback(times(4)) # counts as once already

sum(1:10) ; mean(1:3) # two more
sinpi(1)              # 4th - and "done"
cospi(1)
tanpi(1)
```

`taskCallbackManager` *Create an R-level Task Callback Manager*

Description

This provides an entirely R-language mechanism for managing callbacks or actions that are invoked at the conclusion of each top-level task. Essentially, we register a single R function from this manager with the underlying, native task-callback mechanism and this function handles invoking the other R callbacks under the control of the manager. The manager consists of a collection of functions that access shared variables to manage the list of user-level callbacks.

Usage

```
taskCallbackManager(handlers = list(), registered = FALSE,
                    verbose = FALSE)
```

Arguments

handlers	this can be a list of callbacks in which each element is a list with an element named "f" which is a callback function, and an optional element named "data" which is the 5-th argument to be supplied to the callback when it is invoked. Typically this argument is not specified, and one uses add to register callbacks after the manager is created.
registered	a logical value indicating whether the evaluate function has already been registered with the internal task callback mechanism. This is usually FALSE and the first time a callback is added via the add function, the evaluate function is automatically registered. One can control when the function is registered by specifying TRUE for this argument and calling addTaskCallback manually.
verbose	a logical value, which if TRUE, causes information to be printed to the console about certain activities this dispatch manager performs. This is useful for debugging callbacks and the handler itself.

Value

A list containing 6 functions:

add()	register a callback with this manager, giving the function, an optional 5-th argument, an optional name by which the callback is stored in the list, and a register argument which controls whether the evaluate function is registered with the internal C-level dispatch mechanism if necessary.
remove()	remove an element from the manager's collection of callbacks, either by name or position/index.
evaluate()	the 'real' callback function that is registered with the C-level dispatch mechanism and which invokes each of the R-level callbacks within this manager's control.
suspend()	a function to set the suspend state of the manager. If it is suspended, none of the callbacks will be invoked when a task is completed. One sets the state by specifying a logical value for the status argument.
register()	a function to register the evaluate function with the internal C-level dispatch mechanism. This is done automatically by the add function, but can be called manually.
callbacks()	returns the list of callbacks being maintained by this manager.

References

Duncan Temple Lang (2001) *Top-level Task Callbacks in R*, https://developer.r-project.org/TaskHandlers.pdf

See Also

addTaskCallback, removeTaskCallback, getTaskCallbackNames and the reference.

Examples

```
# create the manager
h <- taskCallbackManager()

# add a callback
h$add(function(expr, value, ok, visible) {
                      cat("In handler\n")
                      return(TRUE)
                    }, name = "simpleHandler")

# look at the internal callbacks.
getTaskCallbackNames()

# look at the R-level callbacks
names(h$callbacks())

removeTaskCallback("R-taskCallbackManager")
```

---

taskCallbackNames *Query the Names of the Current Internal Top-Level Task Callbacks*

Description

This provides a way to get the names (or identifiers) for the currently registered task callbacks that are invoked at the conclusion of each top-level task. These identifiers can be used to remove a callback.

Usage

```
getTaskCallbackNames()
```

Value

A character vector giving the name for each of the registered callbacks which are invoked when a top-level task is completed successfully. Each name is the one used when registering the callbacks and returned as the in the call to addTaskCallback.

Note

One can use taskCallbackManager to manage user-level task callbacks, i.e., S-language functions, entirely within the S language and access the names more directly.

See Also

addTaskCallback, removeTaskCallback, taskCallbackManager\ https://developer.r-project.org/TaskHandlers.pdf

Examples

```
n <- addTaskCallback(function(expr, value, ok, visible) {
                       cat("In handler\n")
                       return(TRUE)
                     }, name = "simpleHandler")

getTaskCallbackNames()

  # now remove it by name
removeTaskCallback("simpleHandler")


h <- taskCallbackManager()
h$add(function(expr, value, ok, visible) {
                       cat("In handler\n")
                       return(TRUE)
                     }, name = "simpleHandler")
getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

---

tempfile *Create Names for Temporary Files*

Description

`tempfile` returns a vector of character strings which can be used as names for temporary files.

Usage

```
tempfile(pattern = "file", tmpdir = tempdir(), fileext = "")
tempdir(check = FALSE)
```

Arguments

`pattern`	a non-empty character vector giving the initial part of the name.
`tmpdir`	a non-empty character vector giving the directory name.
`fileext`	a non-empty character vector giving the file extension.
`check`	`logical` indicating if `tmpdir()` should be checked and recreated if no longer valid.

Details

The length of the result is the maximum of the lengths of the three arguments; values of shorter arguments are recycled.

The names are very likely to be unique among calls to `tempfile` in an R session and across simultaneous R sessions (unless `tmpdir` is specified). The filenames are guaranteed not to be currently in use.

The file name is made by concatenating the path given by `tmpdir`, the `pattern` string, a random string in hex and a suffix of `fileext`.

By default, `tmpdir` will be the directory given by `tempdir()`. This will be a subdirectory of the per-session temporary directory found by the following rule when the R session is started. The environment variables `TMPDIR`, `TMP` and `TEMP` are checked in turn and the first found which points to a writable directory is used: if none succeeds '`/tmp`' is used. The path must not contain spaces. Note that setting any of these environment variables in the R session has no effect on `tempdir()`: the per-session temporary directory is created before the interpreter is started.

Value

For `tempfile` a character vector giving the names of possible (temporary) files. Note that no files are generated by `tempfile`.

For `tempdir`, the path of the per-session temporary directory.

On Windows, both will use a backslash as the path separator.

On a Unix-alike, the value will be an absolute path (unless `tmpdir` is set to a relative path), but it need not be canonical (see `normalizePath`) and on macOS it often is not.

Note on parallel use

R processes forked by functions such as `mclapply` and `makeForkCluster` in package **parallel** share a per-session temporary directory. Further, the 'guaranteed not to be currently in use' applies only at the time of asking, and two children could ask simultaneously. This is circumvented by ensuring that `tempfile` calls in different children try different names.

Source

The final component of `tempdir()` is created by the POSIX system call `mkdtemp`, or if this is not available (e.g. on Windows) a version derived from the source code of GNU `glibc`.

It will be of the form '`RtmpXXXXXX`' where the last 6 characters are replaced in a platform-specific way. POSIX only requires that the replacements be ASCII, which allows `.` (so the value may appear to have a file extension) and regexp metacharacters such as `+`. Most commonly the replacements are from the regexp pattern `[A-Za-z0-9]`, but `.` *has* been seen.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unlink` for deleting files.

Examples

```
tempfile(c("ab", "a b c"))   # give file name with spaces in!

tempfile("plot", fileext = c(".ps", ".pdf"))
```

```
tempdir() # works on all platforms with a platform-dependent result


## Show how 'check' is working on some platforms:
if(exists("I'm brave") && `I'm brave` &&
   identical(.Platform$OS.type, "unix") && grepl("^/tmp/", tempdir())) {
  cat("Current tempdir(): ", tempdir(), "\n")
  cat("Removing it :", file.remove(tempdir()),
      "; dir.exists(tempdir()):", dir.exists(tempdir()), "\n")
  cat("and now  tempdir(check = TRUE) :", tempdir(check = TRUE),"\n")
}
```

`textConnection` *Text Connections*

Description

Input and output text connections.

Usage

```
textConnection(object, open = "r", local = FALSE,
               name = deparse1(substitute(object)),
               encoding = c("", "bytes", "UTF-8"))

textConnectionValue(con)
```

Arguments

`object`	character. A description of the connection. For an input this is an R character vector object, and for an output connection the name for the R character vector to receive the output, or `NULL` (for none).
`open`	character string. Either `"r"` (or equivalently `""`) for an input connection or `"w"` or `"a"` for an output connection.
`local`	logical. Used only for output connections. If `TRUE`, output is assigned to a variable in the calling environment. Otherwise the global environment is used.
`name`	a `character` string specifying the connection name.
`encoding`	character string, partially matched. Used only for input connections. How marked strings in `object` should be handled: converted to the current locale, used byte-by-byte or translated to UTF-8.
`con`	an output text connection.

Details

An input text connection is opened and the character vector is copied at time the connection object is created, and `close` destroys the copy. `object` should be the name of a character vector: however, short expressions will be accepted provided they deparse to less than 60 bytes.

An output text connection is opened and creates an R character vector of the given name in the user's workspace or in the calling environment, depending on the value of the `local` argument. This object will at all times hold the completed lines of output to the connection, and `isIncomplete` will indicate if there is an incomplete final line. Closing the connection will output the final line, complete or not. (A line is complete once it has been terminated by end-of-line, represented by `"\n"` in R.) The output character vector has locked bindings (see `lockBinding`) until `close` is called on the connection. The character vector can also be retrieved *via* `textConnectionValue`, which is the only way to do so if `object = NULL`. If the current locale is detected as Latin-1 or UTF-8, non-ASCII elements of the character vector will be marked accordingly (see `Encoding`).

Opening a text connection with `mode = "a"` will attempt to append to an existing character vector with the given name in the user's workspace or the calling environment. If none is found (even if an object exists of the right name but the wrong type) a new character vector will be created, with a warning.

You cannot `seek` on a text connection, and `seek` will always return zero as the position.

Text connections have slightly unusual semantics: they are always open, and throwing away an input text connection without closing it (so it get garbage-collected) does not give a warning.

Value

For `textConnection`, a connection object of class `"textConnection"` which inherits from class `"connection"`.

For `textConnectionValue`, a character vector.

Note

As output text connections keep the character vector up to date line-by-line, they are relatively expensive to use, and it is often better to use an anonymous `file()` connection to collect output.

On (rare) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.
[S has input text connections only.]

See Also

`connections`, `showConnections`, `pushBack`, `capture.output`.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
scan(zz, "", 4)
pushBack(c("aa", "bb"), zz)
scan(zz, "", 4)
close(zz)

zz <- textConnection("foo", "w")
writeLines(c("testit1", "testit2"), zz)
cat("testit3 ", file = zz)
isIncomplete(zz)
cat("testit4\n", file = zz)
isIncomplete(zz)
close(zz)
foo

# capture R output: use part of example from help(lm)
zz <- textConnection("foo", "w")
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
sink(zz)
anova(lm.D9 <- lm(weight ~ group))
cat("\nSummary of Residuals:\n\n")
summary(resid(lm.D9))
sink()
close(zz)
cat(foo, sep = "\n")
```

---

tilde *Tilde Operator*

Description

Tilde is used to separate the left- and right-hand sides in a model formula.

Usage

```
y ~ model
```

Arguments

`y`, `model`	symbolic expressions.

Details

The left-hand side is optional, and one-sided formulae are used in some contexts.

A formula has mode `call`. It can be subsetted by `[[`: the components are `~`, the left-hand side (if present) and the right-hand side *in that order*. (Thus one-sided formulae have two components.)

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`formula`

`timezones` *Time Zones*

Description

Information about time zones in R. `Sys.timezone` returns the name of the current time zone.

Usage

```
Sys.timezone(location = TRUE)

OlsonNames(tzdir = NULL)
```

Arguments

`location`	logical. Defunct, with a warning if `FALSE`.
`tzdir`	the time-zone database to be used: the default is to try known locations until one is found.

Details

Time zones are a system-specific topic, but these days almost all R platforms use similar underlying code, used by Linux, macOS, Solaris, AIX and FreeBSD, and installed with R on Windows. (Unfortunately there are many system-specific errors in the implementations.) It is possible to use the R sources' version of the code on Unix-alikes as well as on Windows: this is the default on macOS.

It should be possible to set the current time zone via the environment variable `TZ`: see the section on 'Time zone names' for suitable values. `Sys.timezone()` will return the value of `TZ` if set initially (and on some OSes it is always set), otherwise it will try to retrieve from the OS a value which if set for `TZ` would give the initial time zone. ('Initially' means before any time-zone functions are used: if `TZ` is being set to override the OS setting or if the 'try' does not get this right, it should be set before the R process is started or (probably early enough) in file `.Rprofile`).

If TZ is set but invalid, most platforms default to 'UTC', the time zone colloquially known as 'GMT' (see https://en.wikipedia.org/wiki/Coordinated_Universal_Time). (Some but not all platforms will give a warning for invalid values.) If it is unset or empty the *system* time zone is used (the one returned by Sys.timezone).

Time zones did not come into use until the middle of the nineteenth century and were not widely adopted until the twentieth, and *daylight saving time* (DST, also known as *summer time*) was first introduced in the early twentieth century, most widely in 1916. Over the last 100 years places have changed their affiliation between major time zones, have opted out of (or in to) DST in various years or adopted DST rule changes late or not at all. (For example, the UK experimented with DST throughout 1971, only.) In a few countries (one is the Irish Republic) it is the summer time which is the 'standard' time and a different name is used in winter. And there can be multiple changes during a year, for example for Ramadan.

A quite common system implementation of POSIXct was as signed 32-bit integers and so only went back to the end of 1901: on such systems R assumes that dates prior to that are in the same time zone as they were in 1902. Most of the world had not adopted time zones by 1902 (so used local 'mean time' based on longitude) but for a few places there had been time-zone changes before then. 64-bit representations are becoming by far the most common; unfortunately on some 64-bit OSes the database information is 32-bit and so only available for the range 1901–2038, and incompletely for the end years.

When a time zone location is first found in a session its value is cached in object .sys.timezone in the base environment.

Value

Sys.timezone returns an OS-specific character string, possibly NA or an empty string (which on some OSes means 'UTC'). This will be a location such as "Europe/London" if one can be ascertained.

A time zone region may be known by several names: for example '"Europe/London"' may also be known as 'GB', 'GB-Eire', 'Europe/Belfast', 'Europe/Guernsey', 'Europe/Isle_of_Man' and 'Europe/Jersey'. A few regions are also known by a summary of their time zone, e.g. 'PST8PDT' is (on most but not all systems) an alias for 'America/Los_Angeles'.

OlsonNames returns a character vector, see the examples for typical cases. It may have an attribute "Version", something like '"2023a"'. (It does on systems using '--with-internal-tzcode' and those like Fedora distributing file 'tzdata.zi'.)

Time zone names

Names "UTC" and its synonym "GMT" are accepted on all platforms.

Where OSes describe their valid time zones can be obscure. The help for the C function tzset can be helpful, but it can also be inaccurate. There is a cumbersome POSIX specification (listed under environment variable TZ at https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_08), which is often at least partially supported, but there are other more user-friendly ways to specify time zones.

Almost all R platforms make use of a time-zone database originally compiled by Arthur David Olson and now managed by IANA, in which the preferred way to refer to a time zone is by a location (typically of a city), e.g., Europe/London, America/Los_Angeles, Pacific/Easter within a 'time zone region'. Some traditional designations are also allowed such as EST5EDT or GB. (Beware that

some of these designations may not be what you expect: in particular EST is a time zone used in Canada *without* daylight saving time, and not EST5EDT nor (Australian) Eastern Standard Time.) The designation can also be an optional colon prepended to the path to a file giving complied zone information (and the examples above are all files in a system-specific location). See https://data.iana.org/time-zones/tz-link.html for more details and references. By convention, regions with a unique time-zone history since 1970 have specific names in the database, but those with different earlier histories may not. Each time zone has one or two (the second for 'summer') *abbreviations* used when formatting times.

Increasingly OSes are (optionally or always) not including 'legacy' names such as US/Eastern: only names of the forms Continent/City and Etc/... are fully portable.

The abbreviations used have changed over the years: for example France used 'PMT' ('Paris Mean Time') from 1891 to 1911 then 'WET/WEST' up to 1940 and 'CET/CEST' from 1946. (In almost all time zones the abbreviations have been stable since 1970.) The POSIX standard allows only one or two abbreviations per time zone, so you may see the current abbreviation(s) used for older times.

For some time zones abbreviations are like '-03' and '+0845': this is done when there is no official abbreviation. (Negative values are behind (West of) UTC, as for the "%z" format for strftime.)

The function OlsonNames returns the time-zone names known to the currently selected Olson/IANA database. The system-specific location in the file system varies, e.g. '/usr/share/zoneinfo' (Linux, macOS, FreeBSD), '/usr/share/lib/zoneinfo' (Solaris, AIX), ... It is likely that there is a file named something like 'zone1970.tab' or (older) 'zone.tab' under that directory listing the locations known as time-zone names (but not for example EST5EDT). See also https://en.wikipedia.org/wiki/Zone.tab.

Where R was configured with option '--with-internal-tzcode' (the default on Windows), the database at file.path(R.home("share"), "zoneinfo") is used by default: file 'VERSION' in that directory states the version. That option is also the default on macOS but there whichever is more recent of the system database at '/var/db/timezone/zoneinfo' and that distributed with R is used by default. Environment variable TZDIR can be used to give the full path to a different 'zoneinfo' database: value "internal" indicates the database from the R sources and "macOS" indicates the system database. (Setting either of those values would not be recognized by other software using TZDIR.)

Setting TZDIR is also supported by the native services on some OSes, e.g. Linux using glibc except in secure modes.

Time zones given by name (*via* environment variable TZ, in tz arguments to functions such as as.POSIXlt and perhaps the system time zone) are loaded from the currently selected 'zoneinfo' database.

On Windows only: An attempt is made (once only per session) to map Windows' idea of the current time zone to a location, following a version of http://unicode.org/repos/cldr/trunk/common/supplemental/windowsZones.xml with additional values deduced from the Windows Registry and documentation. It can be overridden by setting the TZ environment variable before any date-times are used in the session.

Most platforms support time zones of the form 'Etc/GMT+n' and 'Etc/GMT-n' (possibly also without prefix 'Etc/'), which assume a fixed offset from UTC (hence no DST). Contrary to some expectations (but consistent with names such as 'PST8PDT'), negative offsets are times ahead of (East of) UTC, positive offsets are times behind (West of) UTC.

Immediately prior to the advent of legislated time zones, most people used time based on their longitude (or that of a nearby town), known as 'Local Mean Time' and abbreviated as 'LMT' in the

databases: in many countries that was codified with a specific name before the switch to a standard time. For example, Paris codified its LMT as 'Paris Mean Time' in 1891 (to be used throughout mainland France) and switched to 'GMT+0' in 1911.

Some systems (notably Linux) have a `tzselect` command which allows the interactive selection of a supported time zone name. On systems using `systemd` (notably Linux), the OS command `timedatectl list-timezones` will list all available time zone names.

Warnings

There is a system-specific upper limit on the number of bytes in (abbreviated) time-zone names which can be as low as 6 (as required by POSIX). Some OSes allow the setting of time zones with names which exceed their limit, and that can crash the R session.

Information about future times is speculative ('proleptic'): the database provides the best-known information based on current rules set by civil authorities. For the period 1900–1970 those rules (and which of any authority's rules were enacted) are often obscure, and the databases do get corrected frequently.

`OlsonNames` tries to find an Olson database in known locations. It might not succeed (when it returns an empty vector with a warning) and even if it does it might not locate the database used by the date-time code linked into R. Fortunately names are added rarely and most databases are pretty complete. On the other hand, many names which duplicate other named timezones have been moved to the 'backward' list – these are regarded as optional and omitted on minimal installations. Similarly, there are timezones named in file 'backzone' which differ only from those in the main lists prior to 1970 – these are usually included but may not be in minimalist systems.

For many years, the legacy names `EST5EDT` and `PST8PDT` were portable, but `musl` (the C runtime used by Alpine Linux) does not use DST with those names.

How the system time zone is found – on Unix-alikes

This section is of background interest for users of a Unix-alike, but may help if an `NA` value is returned unexpectedly.

Commercial Unixen such as Solaris and AIX set `TZ`, so the value when R is started is used.

All other common platforms (Linux, macOS, *BSD) use similar schemes, either derived from `tzcode` (currently distributed from https://www.iana.org/time-zones) or independently coded (`glibc`, `musl-libc`). Such systems read the time-zone information from a file 'localtime', usually under '/etc' (but possibly under '/usr/local/etc' or '/usr/local/etc/zoneinfo'). As the usual Linux manual page for `localtime` says

'Because the time zone identifier is extracted from the symlink target name of '/etc/localtime', this file may not be a normal file or hardlink.'

Nevertheless, some Linux distributions (including the one from which that quote was taken) or sysadmins have chosen to copy a time-zone file to 'localtime'. For a non-symlink, the ultimate fallback is to compare that file to all files in the time-zone database.

Some Linux platforms provide two other mechanisms which are tried in turn before looking at '/etc/localtime'.

- 'Modern' Linux systems use `systemd` which provides mechanisms to set and retrieve the time zone (amongst other things). There is a command `timedatectl` to give details. (Unfortunately RHEL/Centos 6.x were not 'modern'.)

- Debian-derived systems since *ca* 2007 have supplied a file '`/etc/timezone`'. Its format is undocumented but empirically it contains a single line of text naming the time zone.

In each case a sanity check is performed that the time-zone name is the name of a file in the time-zone database. (The systems probably use the time-zone file (symlinked to) '`/etc/localtime`', but the `Sys.timezone` code does not check that is the same as the named file in the database. This is deliberate as they may be from different dates.)

Note

Since 2007 there has been considerable disruption over changes to the timings of the DST transitions; these often have short notice and time-zone databases may not be up to date. (Morocco in 2013 announced a change to the end of DST at *a day's* notice. In 2023 there was chaos in Lebanon as the authorities changed their minds repeatedly and some changes were not widely implemented.)

There have also been changes to the 'standard' time with little notice (Kazakhstan switched to a single time zone in Mar 2024 with six weeks' notice), and to whether 'summer' or 'winter' time is regarded as 'standard' (and hence to abbreviations).

On platforms with case-insensitive file systems, time zone names will be case-insensitive. They may or may not be on other platforms and so, for example, `"gmt"` is valid on some platforms and not on others.

Note that except where replaced, the operation of time zones is an OS service, and even where replaced a third-party database is used and can be updated (see the section on 'Time zone names'). Incorrect results will never be an R issue, so please ensure that you have the courtesy not to blame R for them.

See Also

`Sys.time`, `as.POSIXlt`.

`https://en.wikipedia.org/wiki/Time_zone` and `https://data.iana.org/time-zones/tz-link.html` for extensive sets of links.

`https://data.iana.org/time-zones/theory.html` for the 'rules' of the Olson/IANA database.

Examples

```
Sys.timezone()

str(OlsonNames()) ## typically around six hundred names,
## typically some acronyms/aliases such as "UTC", "NZ", "MET", "Eire", ..., but
## mostly pairs (and triplets) such as "Pacific/Auckland"
table(sl <- grepl("/", OlsonNames()))
OlsonNames()[ !sl ] # the simple ones
head(Osl <- strsplit(OlsonNames()[sl], "/"))
(tOS1 <- table(vapply(Osl, `[[`, "", 1))) # Continents, countries, ...
table(lengths(Osl))# most are pairs, some triplets
str(Osl[lengths(Osl) >= 3])# "America" South and North ...
```

toString *Convert an R Object to a Character String*

Description

This is a helper function for format to produce a single character string describing an R object.

Usage

```
toString(x, ...)

## Default S3 method:
toString(x, width = NULL, ...)
```

Arguments

x	The object to be converted.
width	Suggestion for the maximum field width. Values of NULL or 0 indicate no maximum. The minimum value accepted is 6 and smaller values are taken as 6.
...	Optional arguments passed to or from methods.

Details

This is a generic function for which methods can be written: only the default method is described here. Most methods should honor the width argument to specify the maximum display width (as measured by nchar(type = "width")) of the result.

The default method first converts x to character and then concatenates the elements separated by ", ". If width is supplied and is not NULL, the default method returns the first width - 4 characters of the result with .... appended, if the full result would use more than width characters.

Value

A character vector of length 1 is returned.

Author(s)

Robert Gentleman

See Also

format

Examples

```
x <- c("a", "b", "aaaaaaaaaa")
toString(x)
toString(x, width = 8)
```

trace *Interactive Tracing and Debugging of Calls to a Function or Method*

Description

A call to trace allows you to insert debugging code (e.g., a call to browser or recover) at chosen places in any function. A call to untrace cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the generic function. Trace code (tracer) can be any R expression. Tracing can be temporarily turned on or off globally by calling tracingState.

Usage

```
trace(what, tracer, exit, at, print, signature,
      where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
.doTrace(expr, msg)
returnValue(default = NULL)
```

Arguments

what	the name, possibly quote()d, of a function to be traced or untraced. For untrace or for trace with more than one argument, more than one name can be given in the quoted form, and the same action will be applied to each one. For "hidden" functions such as S3 methods in a namespace, where = * typically needs to be specified as well.
tracer	either a function or an unevaluated expression. The function will be called or the expression will be evaluated either at the beginning of the call, or before those steps in the call specified by the argument at. See the details section.
exit	either a function or an unevaluated expression. The function will be called or the expression will be evaluated on exiting the function. See the details section.
at	optional numeric vector or list. If supplied, tracer will be called just before the corresponding step in the body of the function. See the details section.
print	if TRUE (as per default), a descriptive line is printed before any trace expression is evaluated.
signature	an optional signature for a method for function what. If supplied, the method, and *not* the function itself, is traced.
edit	For complicated tracing, such as tracing within a loop inside the function, you will need to insert the desired calls by editing the body of the function. If so, supply the edit argument either as TRUE, or as the name of the editor you want to use. Then trace() will call edit and use the version of the function after you edit it. See the details section for additional information.

`where` where to look for the function to be traced; by default, the top-level environment of the call to `trace`.

An important use of this argument is to trace functions from a package which are "hidden" or called from another package. The namespace mechanism imports the functions to be called (with the exception of functions in the base package). The functions being called are *not* the same objects seen from the top-level (in general, the imported packages may not even be attached). Therefore, you must ensure that the correct versions are being traced. The way to do this is to set argument `where` to a function in the namespace (or that namespace). The tracing computations will then start looking in the environment of that function (which will be the namespace of the corresponding package). (Yes, it's subtle, but the semantics here are central to how namespaces work in R.)

`on` logical; a call to the support function `tracingState` returns `TRUE` if tracing is globally turned on, `FALSE` otherwise. An argument of one or the other of those values sets the state. If the tracing state is `FALSE`, none of the trace actions will actually occur (used, for example, by debugging functions to shut off tracing during debugging).

`expr`, `msg` arguments to the support function `.doTrace`, calls to which are inserted into the modified function or method: `expr` is the tracing action (such as a call to `browser()`), and `msg` is a string identifying the place where the trace action occurs.

`default` if `returnValue` finds no return value (e.g., when a function exited because of an error, restart or as a result of evaluating a return from a caller function), it will return `default` instead.

Details

The `trace` function operates by constructing a revised version of the function (or of the method, if `signature` is supplied), and assigning the new object back where the original was found. If only the `what` argument is given, a line of trace printing is produced for each call to the function (back compatible with the earlier version of `trace`).

The object constructed by `trace` is from a class that extends `"function"` and which contains the original, untraced version. A call to `untrace` re-assigns this version.

If the argument `tracer` or `exit` is the name of a function, the tracing expression will be a call to that function, with no arguments. This is the easiest and most common case, with the functions `browser` and `recover` the likeliest candidates; the former browses in the frame of the function being traced, and the latter allows browsing in any of the currently active calls. The arguments `tracer` and `exit` are evaluated to see whether they are functions, but only their names are used in the tracing expressions. The lookup is done again when the traced function executes, so it may not be `tracer` or `exit` that will be called while tracing.

The `tracer` or `exit` argument can also be an unevaluated expression (such as returned by a call to `quote` or `substitute`). This expression itself is inserted in the traced function, so it will typically involve arguments or local objects in the traced function. An expression of this form is useful if you only want to interact when certain conditions apply (and in this case you probably want to supply `print = FALSE` in the call to `trace` also).

When the `at` argument is supplied, it can be a vector of integers referring to the substeps of the body of the function (this only works if the body of the function is enclosed in `{ ... }`). In this

case tracer is *not* called on entry, but instead just before evaluating each of the steps listed in at. (Hint: you don't want to try to count the steps in the printed version of a function; instead, look at as.list(body(f)) to get the numbers associated with the steps in function f.)

The at argument can also be a list of integer vectors. In this case, each vector refers to a step nested within another step of the function. For example, at = list(c(3,4)) will call the tracer just before the fourth step of the third step of the function. See the example below.

Using setBreakpoint (from package **utils**) may be an alternative, calling trace(...., at, ...).

The exit argument is called during on.exit processing. In an on.exit expression, the experimental returnValue() function may be called to obtain the value about to be returned by the function. Calling this function in other circumstances will give undefined results.

An intrinsic limitation in the exit argument is that it won't work if the function itself uses on.exit with add= FALSE (the default), since the existing calls will override the one supplied by trace.

Tracing does not nest. Any call to trace replaces previously traced versions of that function or method (except for edited versions as discussed below), and untrace always restores an untraced version. (Allowing nested tracing has too many potentials for confusion and for accidentally leaving traced versions behind.)

When the edit argument is used repeatedly with no call to untrace on the same function or method in between, the previously edited version is retained. If you want to throw away all the previous tracing and then edit, call untrace before the next call to trace. Editing may be combined with automatic tracing; just supply the other arguments such as tracer, and the edit argument as well. The edit = TRUE argument uses the default editor (see edit).

Tracing primitive functions (builtins and specials) from the base package works, but only by a special mechanism and not very informatively. Tracing a primitive causes the primitive to be replaced by a function with argument ... (only). You can get a bit of information out, but not much. A warning message is issued when trace is used on a primitive.

The practice of saving the traced version of the function back where the function came from means that tracing carries over from one session to another, *if* the traced function is saved in the session image. (In the next session, untrace will remove the tracing.) On the other hand, functions that were in a package, not in the global environment, are not saved in the image, so tracing expires with the session for such functions.

Tracing an S4 method is basically just like tracing a function, with the exception that the traced version is stored by a call to setMethod rather than by direct assignment, and so is the untraced version after a call to untrace.

The version of trace described here is largely compatible with the version in S-Plus, although the two work by entirely different mechanisms. The S-Plus trace uses the session frame, with the result that tracing never carries over from one session to another (R does not have a session frame). Another relevant distinction has nothing directly to do with trace: The browser in S-Plus allows changes to be made to the frame being browsed, and the changes will persist after exiting the browser. The R browser allows changes, but they disappear when the browser exits. This may be relevant in that the S-Plus version allows you to experiment with code changes interactively, but the R version does not. (A future revision may include a 'destructive' browser for R.)

Value

In the simple version (just the first argument), trace returns an invisible NULL. Otherwise, the traced function(s) name(s). The relevant consequence is the assignment that takes place.

`untrace` returns the function name invisibly.

`tracingState` returns the current global tracing state, and possibly changes it.

When called during `on.exit` processing, `returnValue` returns the value about to be returned by the exiting function. Behaviour in other circumstances is undefined.

Note

Using `trace()` is conceptually a generalization of `debug`, implemented differently. Namely by calling `browser` via its `tracer` or `exit` argument.

The version of function tracing that includes any of the arguments except for the function name requires the **methods** package (because it uses special classes of objects to store and restore versions of the traced functions).

If methods dispatch is not currently on, `trace` will load the methods namespace, but will not put the methods package on the `search` list.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`browser` and `recover`, the likeliest tracing functions; also, `quote` and `substitute` for constructing general expressions.

Examples

```
require(stats)

##  Very simple use
trace(sum)
hist(rnorm(100)) # shows about 3-4 calls to sum()
untrace(sum)

## Show how pt() is called from inside power.t.test():
if(FALSE)
  trace(pt) ## would show ~20 calls, but we want to see more:
trace(pt, tracer = quote(cat(sprintf("tracing pt(*, ncp = %.15g)\n", ncp))),
      print = FALSE) # <- not showing typical extra
power.t.test(20, 1, power=0.8, sd=NULL)  ##--> showing the ncp root finding:
untrace(pt)

f <- function(x, y) {
    y <- pmax(y, 0.001)
    if (x > 0) x ^ y else stop("x must be positive")
}

## arrange to call the browser on entering and exiting
## function f
trace("f", quote(browser(skipCalls = 4)),
```

```
      exit = quote(browser(skipCalls = 4)))

## instead, conditionally assign some data, and then browse
## on exit, but only then.  Don't bother me otherwise

trace("f", quote(if(any(y < 0)) yOrig <- y),
      exit = quote(if(exists("yOrig")) browser(skipCalls = 4)),
      print = FALSE)

## Enter the browser just before stop() is called.  First, find
## the step numbers

untrace(f) # (as it has changed f's body !)
as.list(body(f))
as.list(body(f)[[3]]) # -> stop(..) is [[4]]

## Now call the browser there

trace("f", quote(browser(skipCalls = 4)), at = list(c(3,4)))
## Not run:
f(-1,2) # --> enters browser just before stop(..)

## End(Not run)

## trace a utility function, with recover so we
## can browse in the calling functions as well.

trace("as.matrix", recover)

## turn off the tracing (that happened above)

untrace(c("f", "as.matrix"))

## Not run:
## Useful to find how system2() is called in a higher-up function:
trace(base::system2, quote(print(ls.str())))

## End(Not run)

##-------- Tracing hidden functions : need 'where = *'
##
## 'where' can be a function whose environment is meant:
trace(quote(ar.yw.default), where = ar)
a <- ar(rnorm(100)) # "Tracing ..."
untrace(quote(ar.yw.default), where = ar)

## trace() more than one function simultaneously:
##         expression(E1, E2, ...)  here is equivalent to
##          c(quote(E1), quote(E2), quote(.*), ..)
trace(expression(ar.yw, ar.yw.default), where = ar)
a <- ar(rnorm(100)) # --> 2 x "Tracing ..."
# and turn it off:
untrace(expression(ar.yw, ar.yw.default), where = ar)
```

```
## Not run:
## trace calls to the function lm() that come from
## the nlme package.
trace("lm", where = asNamespace("nlme"))
      lm    (len ~ log(dose) * supp, ToothGrowth) -> fit1  # NOT traced
nlme::lmList(len ~ log(dose) | supp, ToothGrowth) -> fit2  # traced
untrace("lm", where = asNamespace("nlme"))

## End(Not run)
```

---

traceback *Get and Print Call Stacks*

Description

By default `traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print the current stack or arbitrary lists of calls.

`.traceback()` now *returns* the above call stack (and `traceback(x, *)` can be regarded as convenience function for printing the result of `.traceback(x)`).

Usage

```
 traceback(x = NULL, max.lines = getOption("traceback.max.lines",
                                           getOption("deparse.max.lines", -1L)))
.traceback(x = NULL, max.lines = getOption("traceback.max.lines",
                                           getOption("deparse.max.lines", -1L)))
```

Arguments

`x`	`NULL` (default, meaning `.Traceback`), or an integer count of calls to skip in the current stack, or a list or pairlist of calls. See the details.
`max.lines`	a number, the maximum number of lines to be printed *per call*. The default is unlimited. Applies only when `x` is `NULL`, a `list` or a `pairlist` of calls, see the details.

Details

The default display is of the stack of the last uncaught error as stored as a list of `call`s in `.Traceback`, which `traceback` prints in a user-friendly format. The stack of calls always contains all function calls and all foreign function calls (such as `.Call`): if profiling is in progress it will include calls to some primitive functions. (Calls to builtins are included, but not to specials.)

Errors which are caught *via* `try` or `tryCatch` do not generate a traceback, so what is printed is the call sequence for the last uncaught error, and not necessarily for the last error.

If x is numeric, then the current stack is printed, skipping x entries at the top of the stack. For example, options(error = function() traceback(3)) will print the stack at the time of the error, skipping the call to traceback() and .traceback() and the error function that called it.

Otherwise, x is assumed to be a list or pairlist of calls or deparsed calls and will be displayed in the same way.

.traceback() and by extension traceback() may trigger deparsing of calls. This is an expensive operation for large calls so it may be advisable to set max.lines to a reasonable value when such calls are on the call stack.

Value

.traceback() returns the deparsed call stack deepest call first as a list or pairlist. The number of lines deparsed from the call can be limited via max.lines. Calls for which max.lines results in truncated output will gain a "truncated" attribute.

traceback() formats, prints, and returns the call stack produced by .traceback() invisibly.

Warning

It is undocumented where .Traceback is stored nor that it is visible, and this is subject to change. Currently .Traceback contains the calls as language objects.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }
## Not run:
foo(2) # gives a strange error
traceback()
## End(Not run)
## 2: bar(2)
## 1: foo(2)
bar
## Ah, this is the culprit ...

## This will print the stack trace at the time of the error.
options(error = function() traceback(3))
```

`tracemem` *Trace Copying of Objects*

Description

This function marks an object so that a message is printed whenever the internal code copies the object. It is a major cause of hard-to-predict memory use in R.

Usage

```
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

Arguments

`x`	An R object, not a function or environment or `NULL`.
`previous`	A value as returned by `tracemem` or `retracemem`.

Details

This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. `tracemem` and `untracemem` give errors when R is not compiled with memory profiling; `retracemem` does not (so it can be left in code during development).

It is enabled in the CRAN macOS and Windows builds of R.

When an object is traced any copying of the object by the C function `duplicate` produces a message to standard output, as does type coercion and copying when passing arguments to `.C` or `.Fortran`.

The message consists of the string `tracemem`, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. `retracemem()` is used to indicate that a variable should be considered a copy of a previous variable (e.g., after subscripting).

The messages can be turned off with `tracingState`.

It is not possible to trace functions, as this would conflict with `trace` and it is not useful to trace `NULL`, environments, promises, weak references, or external pointer objects, as these are not duplicated.

These functions are primitive.

Value

A character string for identifying the object in the trace output (an address in hex enclosed in angle brackets), or `NULL` (invisibly).

See Also

capabilities("profmem") to see if this was enabled for this build of R.

trace, Rprofmem

https://developer.r-project.org/memory-profiling.html

Examples

```
## Not run:
a <- 1:10
tracemem(a)
## b and a share memory
b <- a
b[1] <- 1
untracemem(a)

## copying in lm: less than R <= 2.15.0
d <- stats::rnorm(10)
tracemem(d)
lm(d ~ a+log(b))

## f is not a copy and is not traced
f <- d[-1]
f+1
## indicate that f should be traced as a copy of d
retracemem(f, retracemem(d))
f+1

## End(Not run)
```

---

transform *Transform an Object, for Example a Data Frame*

Description

transform is a generic function, which—at least currently—only does anything useful with data frames. transform.default converts its first argument to a data frame if possible and calls transform.data.frame.

Usage

```
transform(`_data`, ...)
```

Arguments

_data	The object to be transformed
...	Further arguments of the form tag=value

Details

The ... arguments to transform.data.frame are tagged vector expressions, which are evaluated in the data frame _data. The tags are matched against names(_data), and for those that match, the value replace the corresponding variable in _data, and the others are appended to _data.

Value

The modified value of _data.

Warning

This is a convenience function intended for use interactively. For programming it is better to use the standard subsetting arithmetic functions, and in particular the non-standard evaluation of argument transform can have unanticipated consequences.

Note

If some of the values are not vectors of the appropriate length, you deserve whatever you get!

Author(s)

Peter Dalgaard

See Also

within for a more flexible approach, subset, list, data.frame

Examples

```
transform(airquality, Ozone = -Ozone)
transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)

attach(airquality)
transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
detach(airquality)
```

Trig *Trigonometric Functions*

Description

These functions give the obvious trigonometric functions. They respectively compute the cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent, and the two-argument arc-tangent.

cospi(x), sinpi(x), and tanpi(x), compute cos(pi*x), sin(pi*x), and tan(pi*x).

Usage

```
cos(x)
sin(x)
tan(x)

acos(x)
asin(x)
atan(x)
atan2(y, x)

cospi(x)
sinpi(x)
tanpi(x)
```

Arguments

x, y	numeric or complex vectors.

Details

The arc-tangent of two arguments `atan2(y, x)` returns the angle between the x-axis and the vector from the origin to (x, y), i.e., for positive arguments `atan2(y, x)` == `atan(y/x)`.

Angles are in radians, not degrees, for the standard versions (i.e., a right angle is $\pi/2$), and in 'half-rotations' for `cospi` etc.

`cospi(x)`, `sinpi(x)`, and `tanpi(x)` are accurate for `x` values which are multiples of a half.

All except `atan2` are internal generic primitive functions: methods can be defined for them individually or via the `Math` group generic.

These are all wrappers to system calls of the same name (with prefix `c` for complex arguments) where available. (`cospi`, `sinpi`, and `tanpi` are part of a C11 extension and provided by e.g. macOS and Solaris: where not yet available call to `cos` *etc* are used, with special cases for multiples of a half.)

Value

`tanpi(0.5)` is `NaN`. Similarly for other inputs with fractional part `0.5`.

Complex values

For the inverse trigonometric functions, branch cuts are defined as in Abramowitz and Stegun, figure 4.4, page 79.

For `asin` and `acos`, there are two cuts, both along the real axis: $(-\infty, -1]$ and $[1, \infty)$.

For `atan` there are two cuts, both along the pure imaginary axis: $(-\infty i, -1i]$ and $[1i, \infty i)$.

The behaviour actually on the cuts follows the C99 standard which requires continuity coming round the endpoint in a counter-clockwise direction.

Complex arguments for `cospi`, `sinpi`, and `tanpi` are not yet implemented, and they are a 'future direction' of ISO/IEC TS 18661-4.

S4 methods

All except atan2 are S4 generic functions: methods can be defined for them individually or via the Math group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions*. New York: Dover.
Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

For cospi, sinpi, and tanpi the C11 extension ISO/IEC TS 18661-4:2015 (draft at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1950.pdf).

Examples

```
x <- seq(-3, 7, by = 1/8)
tx <- cbind(x, cos(pi*x), cospi(x), sin(pi*x), sinpi(x),
               tan(pi*x), tanpi(x), deparse.level=2)
op <- options(digits = 4, width = 90) # for nice formatting
head(tx)
tx[ (x %% 1) %in% c(0, 0.5) ,]
options(op)
```

---

trimws *Remove Leading/Trailing Whitespace*

Description

Remove leading and/or trailing whitespace from character strings.

Usage

```
trimws(x, which = c("both", "left", "right"), whitespace = "[ \t\r\n]")
```

Arguments

x	a character vector.
which	a character string specifying whether to remove both leading and trailing whitespace (default), or only leading ("left") or trailing ("right"). Can be abbreviated.
whitespace	a string specifying a regular expression to match (one character of) "white space", see Details for alternatives to the default.

Details

Internally, `sub(re, "", *, perl = TRUE)`, i.e., PCRE library regular expressions are used. For portability, the default 'whitespace' is the character class `[ \t\r\n]` (space, horizontal tab, carriage return, newline). Alternatively, `[\h\v]` is a good (PCRE) generalization to match all Unicode horizontal and vertical white space characters, see also https://www.pcre.org.

Examples

```
x <- "  Some text. "
x
trimws(x)
trimws(x, "l")
trimws(x, "r")

## Unicode --> need "stronger" 'whitespace' to match all :
tt <- "text with unicode 'non breakable space'."
xu <- paste(" \t\v", tt, "\u00a0 \n\r")
(tu <- trimws(xu, whitespace = "[\\h\\v]"))
stopifnot(identical(tu, tt))
```

---

try *Try an Expression Allowing Error Recovery*

Description

`try` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.

Usage

```
try(expr, silent = FALSE,
    outFile = getOption("try.outFile", default = stderr()))
```

Arguments

`expr`	an R expression to try.
`silent`	logical: should the report of error messages be suppressed?
`outFile`	a connection, or a character string naming the file to print to (via `cat(*, file = outFile)`); used only if `silent` is false, as by default.

Details

`try` evaluates an expression and traps any errors that occur during the evaluation. If an error occurs then the error message is printed to the `stderr` connection unless `options("show.error.messages")` is false or the call includes `silent = TRUE`. The error message is also stored in a buffer where it can be retrieved by `geterrmessage`. (This should not be needed as the value returned in case of an error contains the error message.)

`try` is implemented using `tryCatch`; for programming, instead of `try(expr, silent = TRUE)`, something like `tryCatch(expr, error = function(e) e)` (or other simple error handler functions) may be more efficient and flexible.

It may be useful to set the default for `outFile` to `stdout()`, i.e.,

```
options(try.outFile = stdout())
```

instead of the default `stderr()`, notably when `try()` is used inside a `Sweave` code chunk and the error message should appear in the resulting document.

Value

The value of the expression if `expr` is evaluated without error: otherwise an invisible object inheriting from class `"try-error"` containing the error message with the error condition as the `"condition"` attribute.

Warning

Do not test

```
if (class(res) == "try-error"))
```

as if there is no error, the result might (now or in future) have a class of length > 1. Use `if(inherits(res, "try-error"))` instead.

See Also

`options` for setting error handlers and suppressing the printing of error messages; `geterrmessage` for retrieving the last error message. The underlying `tryCatch` provides more flexible means of catching and handling errors.

`assertCondition` in package **tools** is related and useful for testing.

Examples

```
## this example will not work correctly in example(try), but
## it does work correctly if pasted in
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))

## run a simulation, keep only the results that worked.
set.seed(123)
x <- stats::rnorm(50)
doit <- function(x)
{
    x <- sample(x, replace = TRUE)
```

```
    if(length(unique(x)) > 30) mean(x)
    else stop("too few unique points")
}
## alternative 1
res <- lapply(1:100, function(i) try(doit(x), TRUE))
## alternative 2
## Not run: res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x), TRUE)
## End(Not run)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])
```

---

typeof *The Type of an Object*

Description

typeof determines the (R internal) type or storage mode of any object

Usage

```
typeof(x)
```

Arguments

x	any R object.

Value

A character string. The possible values are listed in the structure TypeTable in 'src/main/util.c'. Current values are the vector types "logical", "integer", "double", "complex", "character", "raw" and "list", "NULL", "closure" (function), "special" and "builtin" (basic functions and operators), "environment", "S4" (some S4 objects) and others that are unlikely to be seen at user level ("symbol", "pairlist", "promise", "object", "language", "char", "...", "any", "expression", "externalptr", "bytecode" and "weakref").

See Also

mode, storage.mode.

isS4 to determine if an object has an S4 class.

Examples

```
typeof(2)
mode(2)
## for a table of examples, see  ?mode  /  examples(mode)
```

unique *Extract Unique Elements*

Description

unique returns a vector, data frame or array like x but with duplicate elements/rows removed.

Usage

```
unique(x, incomparables = FALSE, ...)

## Default S3 method:
unique(x, incomparables = FALSE, fromLast = FALSE,
        nmax = NA, ...)

## S3 method for class 'matrix'
unique(x, incomparables = FALSE, MARGIN = 1,
       fromLast = FALSE, ...)

## S3 method for class 'array'
unique(x, incomparables = FALSE, MARGIN = 1,
       fromLast = FALSE, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	a vector of values that cannot be compared. FALSE is a special value, meaning that all values can be compared, and may be the only value accepted for methods other than the default. It will be coerced internally to the same type as x.
fromLast	logical indicating if duplication should be considered from the last, i.e., the last (or rightmost) of identical elements will be kept. This only matters for names or dimnames.
nmax	the maximum number of unique items expected (greater than one). See duplicated.
...	arguments for particular methods.
MARGIN	the array margin to be held fixed: a single integer.

Details

This is a generic function with methods for vectors, data frames and arrays (including matrices).

The array method calculates for each element of the dimension specified by MARGIN if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used for matrices to find unique rows (the default) or columns (with MARGIN = 2).

Note that unlike the Unix command uniq this omits *duplicated* and not just *repeated* elements/rows. That is, an element is omitted if it is equal to any previous element and not just if it is equal the immediately previous one. (For the latter, see rle).

Missing values ("NA") are regarded as equal, numeric and complex ones differing from NaN; character strings will be compared in a "common encoding"; for details, see match (and duplicated) which use the same concept.

Values in incomparables will never be marked as duplicated. This is intended to be used for a fairly small set of values and will not be efficient for a very large set.

When used on a data frame with more than one column, or an array or matrix when comparing dimensions of length greater than one, this tests for identity of character representations. This will catch people who unwisely rely on exact equality of floating-point numbers!

Value

For a vector, an object of the same type of x, but with only one copy of each duplicated element. No attributes are copied (so the result has no names).

For a data frame, a data frame is returned with the same columns but possibly fewer rows (and with row names from the first occurrences of the unique rows).

A matrix or array is subsetted by [, drop = FALSE], so dimensions and dimnames are copied appropriately, and the result always has the same number of dimensions as x.

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see vector) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

duplicated which gives the indices of duplicated elements.

rle which is the equivalent of the Unix uniq -c command.

Examples

```
x <- c(3:5, 11:8, 8 + 0:5)
(ux <- unique(x))
(u2 <- unique(x, fromLast = TRUE)) # different order
stopifnot(identical(sort(ux), sort(u2)))

length(unique(sample(100, 100, replace = TRUE)))
## approximately 100(1 - 1/e) = 63.21

unique(iris)
```

units *Units*

Description

Get or set units.

Usage

```
units(x)
units(x) <- value
```

Arguments

x	an R object
value	an R object

Details

These are generic functions, with methods for "difftime" objects.

unlink *Delete Files and Directories*

Description

unlink deletes the file(s) or directories specified by x.

Usage

```
unlink(x, recursive = FALSE, force = FALSE, expand = TRUE)
```

Arguments

x	a character vector with the names of the file(s) or directories to be deleted.
recursive	logical. Should directories be deleted recursively?
force	logical. Should permissions be changed (if possible) to allow the file or directory to be removed?
expand	logical. Should wildcards (see 'Details' below) and tilde (see path.expand) be expanded?

Details

If `recursive = FALSE` directories are not deleted, not even empty ones.

On most platforms 'file' includes symbolic links, fifos and sockets. `unlink(x, recursive = TRUE)` deletes just the symbolic link if the target of such a link is a directory.

Wildcard expansion (normally '`*`' and '`?`' are allowed) is done by the internal code of `Sys.glob`. Wildcards never match a leading '`.`' in the filename, and files '`.`', '`..`' and '`~`' will never be considered for deletion. Wildcards will only be expanded if the system supports it. Most systems will support not only '`*`' and '`?`' but also character classes such as '`[a-z]`' (see the `man` pages for the system call `glob` on your OS). The metacharacters `* ? [` can occur in Unix filenames, and this makes it difficult to use `unlink` to delete such files (see `file.remove`), although escaping the metacharacters by backslashes usually works. If a metacharacter matches nothing it is considered as a literal character.

`recursive = TRUE` might not be supported on all platforms, when it will be ignored, with a warning: however there are no known current examples.

Value

`0` for success, `1` for failure, invisibly. Not deleting a non-existent file is not a failure, nor is being unable to delete a directory if `recursive = FALSE`. However, missing values in `x` are regarded as failures.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`file.remove`.

`unlist` *Flatten Lists*

Description

Given a list structure `x`, `unlist` simplifies it to produce a vector which contains all the atomic components which occur in `x`.

Usage

```
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`x`	an R object, typically a list or vector.
`recursive`	logical. Should unlisting be applied to list components of `x`?
`use.names`	logical. Should names be preserved?

Details

unlist is generic: you can write methods to handle specific classes of objects, see InternalMethods, and note, e.g., relist with the unlist method for relistable objects.

If recursive = FALSE, the function will not recurse beyond the first level items in x.

Factors are treated specially. If all non-list elements of x are factor (or ordered factor) objects then the result will be a factor with levels the union of the level sets of the elements, in the order the levels occur in the level sets of the elements (which means that if all the elements have the same level set, that is the level set of the result).

x can be an atomic vector, but then unlist does nothing useful, not even drop names.

By default, unlist tries to retain the naming information present in x. If use.names = FALSE all naming information is dropped.

Where possible the list elements are coerced to a common mode during the unlisting, and so the result often ends up as a character vector. Vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression: pairlists are treated as lists.

A list is a (generic) vector, and the simplified vector might still be a list (and might be unchanged). Non-vector elements of the list (for example language elements such as names, formulas and calls) are not coerced, and so a list containing one or more of these remains a list. (The effect of unlisting an lm fit is a list which has individual residuals as components.) Note that unlist(x) now returns x unchanged also for non-vector x, instead of signalling an error in that case.

Value

NULL or an expression or a vector of an appropriate mode to hold the list components.

The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression, after coercion of pairlists to lists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c, as.list, relist.

Examples

```
unlist(options())
unlist(options(), use.names = FALSE)

l.ex <- list(a = list(1:5, LETTERS[1:5]), b = "Z", c = NA)
unlist(l.ex, recursive = FALSE)
unlist(l.ex, recursive = TRUE)

l1 <- list(a = "a", b = 2, c = pi+2i)
```

```
unlist(l1) # a character vector
l2 <- list(a = "a", b = as.name("b"), c = pi+2i)
unlist(l2) # remains a list

ll <- list(as.name("sinc"), quote( a + b ), 1:10, letters, expression(1+x))
utils::str(ll)
for(x in ll)
  stopifnot(identical(x, unlist(x)))
```

---

unname *Remove* names *or* dimnames

Description

Remove the names or dimnames attribute of an R object.

Usage

```
unname(obj, force = FALSE)
```

Arguments

obj	an R object.
force	logical; if true, the dimnames (names and row names) are removed even from data.frames.

Value

Object as obj but without names or dimnames.

Examples

```
require(graphics); require(stats)

## Answering a question on R-help (14 Oct 1999):
col3 <- 750+ 100*rt(1500, df = 3)
breaks <- factor(cut(col3, breaks = 360+5*(0:155)))
z <- table(breaks)
z[1:5] # The names are larger than the data ...
barplot(unname(z), axes = FALSE)
```

use *Use Packages*

Description

Use packages in R scripts by loading their namespace and attaching a package environment including (a subset of) their exports to the search path.

Usage

```
use(package, include.only)
```

Arguments

package	a character string given the name of a package.
include.only	character vector of names of objects to include in the attached environment frame. If missing, all exports are included.

Details

This is a simple wrapper around library which always uses attach.required = FALSE, so that packages listed in the Depends clause of the DESCRIPTION file of the package to be used never get attached automatically to the search path.

This therefore allows to write R scripts with full control over what gets found on the search path. In addition, such scripts can easily be integrated as package code, replacing the calls to use by the corresponding ImportFrom directives in 'NAMESPACE' files.

Value

(invisibly) a logical indicating whether the package to be used is available.

Note

This functionality is still experimental: interfaces may change in future versions.

UseMethod *Class Methods*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class(es) of the first argument to the generic function or of the object supplied as an argument to UseMethod or NextMethod.

Usage

```
UseMethod(generic, object)

NextMethod(generic = NULL, object = NULL, ...)
```

Arguments

generic	a character string naming a function (and not a built-in operator). Required for UseMethod.
object	for UseMethod: an object whose class will determine the method to be dispatched. Defaults to the first argument of the enclosing function.
...	further arguments to be passed to the next method.

Details

An R object is a data object which has a class attribute (and this can be tested by is.object). A class attribute is a character vector giving the names of the classes from which the object *inherits*.

If the object does not have a class attribute, it has an *implicit class*. Matrices and arrays have class "matrix" or "array" followed by the class of the underlying vector. Most vectors have class the result of mode(x), except that integer vectors have class c("integer", "numeric") and real vectors have class c("double", "numeric"). Function .class2(x) (since R 4.0.x) returns the full implicit (or explicit) class vector of x.

When a function calling UseMethod("fun") is applied to an object with class vector c("first", "second"), the system searches for a function called fun.first and, if it finds it, applies it to the object. If no such function is found a function called fun.second is tried. If no class name produces a suitable function, the function fun.default is used, if it exists, or an error results.

Function methods can be used to find out about the methods for a particular generic function or class.

UseMethod is a primitive function but uses standard argument matching. It is not the only means of dispatch of methods, for there are internal generic and group generic functions. UseMethod currently dispatches on the implicit class even for arguments that are not objects, but the other means of dispatch do not.

NextMethod invokes the next method (determined by the class vector, either of the object supplied to the generic, or of the first argument to the function containing NextMethod if a method was invoked directly). Normally NextMethod is used with only one argument, generic, but if further arguments are supplied these modify the call to the next method.

NextMethod should not be called except in methods called by UseMethod or from internal generics (see InternalGenerics). In particular it will not work inside anonymous calling functions (e.g., get("print.ts")(AirPassengers)).

Namespaces can register methods for generic functions. To support this, UseMethod and NextMethod search for methods in two places: in the environment in which the generic function is called, and in the registration data base for the environment in which the generic is defined (typically a namespace). So methods for a generic function need to be available in the environment of the call to the generic, or they must be registered. (It does not matter whether they are visible in the environment in which the generic is defined.) As from R 3.5.0, the registration data base is searched

after the top level environment (see topenv) of the calling environment (but before the parents of the top level environment).

Technical Details

Now for some obscure details that need to appear somewhere. These comments will be slightly different than those in Chambers(1992). (See also the draft 'R Language Definition'.) UseMethod creates a new function call with arguments matched as they came in to the generic. [Previously local variables defined before the call to UseMethod were retained; as of R 4.4.0 this is no longer the case.] Any statements after the call to UseMethod will not be evaluated as UseMethod does not return. UseMethod can be called with more than two arguments: a warning will be given and additional arguments ignored. (They are not completely ignored in S.) If it is called with just one argument, the class of the first argument of the enclosing function is used as object: unlike S this is the first actual argument passed and not the current value of the object of that name.

NextMethod works by creating a special call frame for the next method. If no new arguments are supplied, the arguments will be the same in number, order and name as those to the current method but their values will be promises to evaluate their name in the current method and environment. Any named arguments matched to ... are handled specially: they either replace existing arguments of the same name or are appended to the argument list. They are passed on as the promise that was supplied as an argument to the current environment. (S does this differently!) If they have been evaluated in the current (or a previous environment) they remain evaluated. (This is a complex area, and subject to change: see the draft 'R Language Definition'.)

The search for methods for NextMethod is slightly different from that for UseMethod. Finding no fun.default is not necessarily an error, as the search continues to the generic itself. This is to pick up an internal generic like [which has no separate default method, and succeeds only if the generic is a primitive function or a wrapper for a .Internal function of the same name. (When a primitive is called as the default method, argument matching may not work as described above due to the different semantics of primitives.)

You will see objects such as .Generic, .Method, and .Class used in methods. These are set in the environment within which the method is evaluated by the dispatch mechanism, which is as follows:

1. Find the context for the calling function (the generic): this gives us the unevaluated arguments for the original call.
2. Evaluate the object (usually an argument) to be used for dispatch, and find a method (possibly the default method) or throw an error.
3. Create an environment for evaluating the method and insert special variables (see below) into that environment. Also copy any variables in the environment of the generic that are not formal (or actual) arguments.
4. Fix up the argument list to be the arguments of the call matched to the formals of the method.

.Generic is a length-one character vector naming the generic function.

.Method is a character vector (normally of length one) naming the method function. (For functions in the group generic Ops it is of length two.)

.Class is a character vector of classes used to find the next method. NextMethod adds an attribute "previous" to .Class giving the .Class last used for dispatch, and shifts .Class along to that used for dispatch.

`.GenericCallEnv` and `.GenericDefEnv` are the environments of the call to be generic and defining the generic respectively. (The latter is used to find methods registered for the generic.)

Note that `.Class` is set when the generic is called, and is unchanged if the class of the dispatching argument is changed in a method. It is possible to change the method that `NextMethod` would dispatch by manipulating `.Class`, but 'this is not recommended unless you understand the inheritance mechanism thoroughly' (Chambers & Hastie, 1992, p. 469).

Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The draft 'R Language Definition'.

`methods`, `class` incl `.class2()`; `getS3method`, `is.object`.

---

userhooks *Functions to Get and Set Hooks for Load, Attach, Detach and Unload*

Description

These functions allow users to set actions to be taken before packages are attached/detached and namespaces are (un)loaded.

Usage

```
getHook(hookName)
setHook(hookName, value,
        action = c("append", "prepend", "replace"))

packageEvent(pkgname,
             event = c("onLoad", "attach", "detach", "onUnload"))
```

Arguments

`hookName`	character string: the hook name.
`pkgname`	character string: the package/namespace name.
`event`	character string: an event for the package. Can be abbreviated.
`value`	a function or a list of functions, or for `action = "replace"`, `NULL`.
`action`	the action to be taken. Can be abbreviated.

Details

`setHook` provides a general mechanism for users to register hooks, a list of functions to be called from system (or user) functions. The initial set of hooks was associated with events on packages/namespaces: these hooks are named via calls to `packageEvent`.

To remove a hook completely, call `setHook(hookName, NULL, "replace")`.

When an R package is attached by `library` or loaded by other means, it can call initialization code. See `.onLoad` for a description of the package hook functions called during initialization. Users can add their own initialization code via the hooks provided by `setHook()`, functions which will be called as `funname(pkgname, pkgpath)` inside a `try` call.

The sequence of events depends on which hooks are defined, and whether a package is attached or just loaded. In the case where all hooks are defined and a package is attached, the order of initialization events is as follows:

1. The package namespace is loaded.
2. The package's `.onLoad` function is run.
3. If S4 methods dispatch is on, any actions set by `setLoadAction` are run.
4. The namespace is sealed.
5. The user's `"onLoad"` hook is run.
6. The package is added to the search path.
7. The package's `.onAttach` function is run.
8. The package environment is sealed.
9. The user's `"attach"` hook is run.

A similar sequence (but in reverse) is run when a package is detached and its namespace unloaded:

1. The user's `"detach"` hook is run.
2. The package's `.Last.lib` function is run.
3. The package is removed from the search path.
4. The user's `"onUnload"` hook is run.
5. The package's `.onUnload` function is run.
6. The package namespace is unloaded.

Note that when an R session is finished, packages are not detached and namespaces are not unloaded, so the corresponding hooks will not be run.

Also note that some of the user hooks are run without the package being on the search path, so in those hooks objects in the package need to be referred to using the double (or triple) colon operator, as in the example.

If multiple hooks are added, they are normally run in the order shown by `getHook`, but the `"detach"` and `"onUnload"` hooks are run in reverse order so the default for package events is to add hooks 'inside' existing ones.

The hooks are stored in the environment `.userHooksEnv` in the base package, with 'mangled' names.

Value

For `getHook` function, a list of functions (possibly empty). For `setHook` function, no return value. For `packageEvent`, the derived hook name (a character string).

Note

Hooks need to be set before the event they modify: for standard packages this can be problematic as **methods** is loaded and attached early in the startup sequence. The usual place to set hooks such as the example below is in the '`.Rprofile`' file, but that will not work for **methods**.

See Also

`library`, `detach`, `loadNamespace`.

See `::` for a discussion of the double and triple colon operators.

Other hooks may be added later: functions `plot.new` and `persp` already have them.

Examples

```
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal = FALSE))
```

---

utf8Conversion *Convert Integer Vectors to or from UTF-8-encoded Character Vectors*

Description

Conversion of UTF-8 encoded character vectors to and from integer vectors representing a UTF-32 encoding.

Usage

```
utf8ToInt(x)
intToUtf8(x, multiple = FALSE, allow_surrogate_pairs = FALSE)
```

Arguments

`x`	object to be converted.
`multiple`	logical: should the conversion be to a single character string or multiple individual characters?
`allow_surrogate_pairs`	logical: should interpretation of surrogate pairs be attempted? (See 'Details'.) Only supported for `multiple = FALSE`.

Details

These will work in any locale, including on platforms that do not otherwise support multi-byte character sets.

Unicode defines a name and a number of all of the glyphs it encompasses: the numbers are called *code points*: since RFC3629 they run from `0` to `0x10FFFF` (with about 5% being assigned by version 13.0 of the Unicode standard and 7% reserved for 'private use').

`intToUtf8` does not by default handle surrogate pairs: inputs in the surrogate ranges are mapped to `NA`. They might occur if a UTF-16 byte stream has been read as 2-byte integers (in the correct byte order), in which case `allow_surrogate_pairs = TRUE` will try to interpret them (with unmatched surrogate values still treated as `NA`).

Value

`utf8ToInt` converts a length-one character string encoded in UTF-8 to an integer vector of Unicode code points.

`intToUtf8` converts a numeric vector of Unicode code points either (default) to a single character string or a character vector of single characters. Non-integral numeric values are truncated to integers. For output to a single character string `0` is silently omitted: otherwise `0` is mapped to `""`. The `Encoding` of a non-`NA` return value is declared as `"UTF-8"`.

Invalid and `NA` inputs are mapped to `NA` output.

Validity

Which code points are regarded as valid has changed over the lifetime of UTF-8. Originally all 32-bit unsigned integers were potentially valid and could be converted to up to 6 bytes in UTF-8. Since 2003 it has been stated that there will never be valid code points larger than `0x10FFFF`, and so valid UTF-8 encodings are never more than 4 bytes.

The code points in the surrogate-pair range `0xD800` to `0xDFFF` are prohibited in UTF-8 and so are regarded as invalid by `utf8ToInt` and by default by `intToUtf8`.

The position of 'noncharacters' (notably `0xFFFE` and `0xFFFF`) was clarified by 'Corrigendum 9' in 2013. These are valid but will never be given an official interpretation. (In some earlier versions of R `utf8ToInt` treated them as invalid.)

References

https://www.rfc-editor.org/rfc/rfc3629, the current standard for UTF-8.

https://www.unicode.org/versions/corrigendum9.html for non-characters.

Examples

```
## will only display in some locales and fonts
intToUtf8(0x03B2L) # Greek beta

utf8ToInt("bi\u00dfchen")
utf8ToInt("\xfa\xb4\xbf\xbf\x9f")

## A valid UTF-16 surrogate pair (for U+10437)
```

```
x <- c(0xD801, 0xDC37)
intToUtf8(x)
intToUtf8(x, TRUE)
(xx <- intToUtf8(x, , TRUE)) # will only display in some locales and fonts
charToRaw(xx)

## An example of how surrogate pairs might occur
x <- "\U10437"
charToRaw(x)
foo <- tempfile()
writeLines(x, file(foo, encoding = "UTF-16LE"))
## next two are OS-specific, but are mandated by POSIX
system(paste("od -x", foo)) # 2-byte units, correct on little-endian platforms
system(paste("od -t x1", foo)) # single bytes as hex
y <- readBin(foo, "integer", 2, 2, FALSE, endian = "little")
sprintf("%X", y)
intToUtf8(y, , TRUE)
```

`UTF8filepaths` *File Paths not in the Native Encoding*

Description

Most modern file systems store file-path components (names of directories and files) in a character encoding of wide scope: usually UTF-8 on a Unix-alike and UCS-2/UTF-16 on Windows. However, this was not true when R was first developed and there are still exceptions amongst file systems, e.g. FAT32.

This was not something anticipated by the C and POSIX standards which only provide means to access files *via* file paths encoded in the current locale, for example those specified in Latin-1 in a Latin-1 locale.

Everything here apart from the specific section on Windows is about Unix-alikes.

Details

It is possible to mark character strings (elements of character vectors) as being in UTF-8 or Latin-1 (see `Encoding`). This allows file paths not in the native encoding to be expressed in R character vectors but there is almost no way to use them unless they can be translated to the native encoding. That is of course not a problem if that is UTF-8, so these details are really only relevant to the use of a non-UTF-8 locale (including a C locale) on a Unix-alike.

Functions to open a file such as `file`, `fifo`, `pipe`, `gzfile`, `bzfile`, `xzfile` and `unz` give an error for non-native filepaths. Where functions look at existence such as `file.exists`, `dir.exists`, `unlink`, `file.info` and `list.files`, non-native filepaths are treated as non-existent.

Many other functions use `file` or `gzfile` to open their files.

`file.path` allows non-native file paths to be combined, marking them as UTF-8 if needed.

`path.expand` only handles paths in the native encoding.

Windows

Windows provides proprietary entry points to access its file systems, and these gained 'wide' versions in Windows NT that allowed file paths in UCS-2/UTF-16 to be accessed from any locale.

Some R functions use these entry points when file paths are marked as Latin-1 or UTF-8 to allow access to paths not in the current encoding. These include `file`, `file.access`, `file.append`, `file.copy`, `file.create`, `file.exists`, `file.info`, `file.link`, `file.remove`, `file.rename`, `file.symlink` and `dir.create`, `dir.exists`, `normalizePath`, `path.expand`, `pipe`, `Sys.glob`, `Sys.junction`, `unlink` but not `gzfile` `bzfile`, `xzfile` nor `unz`.

For functions using `gzfile` (including `load`, `readRDS`, `read.dcf` and `tar`), it is often possible to use a `gzcon` connection wrapping a `file` connection.

Other notable exceptions are `list.files`, `list.dirs`, `system` and file-path inputs for graphics devices.

Historical comment

Before R 4.0.0, file paths marked as being in Latin-1 or UTF-8 were silently translated to the native encoding using escapes such as '<e7>' or '<U+00e7>'. This created valid file names but maybe not those intended.

Note

This document is still a work-in-progress.

---

validUTF8 *Check if a Character Vector is Validly Encoded*

Description

Check if each element of a character vector is valid in its implied encoding.

Usage

```
validUTF8(x)
validEnc(x)
```

Arguments

x	a character vector.

Details

These use similar checks to those used by functions such as `grep`.

`validUTF8` ignores any marked encoding (see `Encoding`) and so looks directly if the bytes in each string are valid UTF-8. (For the validity of 'noncharacters' see the help for `intToUtf8`.)

`validEnc` regards character strings as validly encoded unless their encodings are marked as UTF-8 or they are unmarked and the R session is in a UTF-8 or other multi-byte locale. (The checks in other multi-byte locales depend on the OS and as with `iconv` not all invalid inputs may be detected.)

Value

A logical vector of the same length as `x`. `NA` elements are regarded as validly encoded.

Note

It would be possible to check for the validity of character strings in a Latin-1 encoding, but extensions such as CP1252 are widely accepted as 'Latin-1' and 8-bit encodings rarely need to be checked for validity.

Examples

```
x <-
  ## from example(text)
c("Jetz", "no", "chli", "z\xc3\xbcrit\xc3\xbc\xc3\xbctsch:",
  "(noch", "ein", "bi\xc3\x9fchen", "Z\xc3\xbc", "deutsch)",
   ## from a CRAN check log
   "\xfa\xb4\xbf\xbf\x9f")
validUTF8(x)
validEnc(x) # depends on the locale
Encoding(x) <-"UTF-8"
validEnc(x) # typically the last, x[10], is invalid

## Maybe advantageous to declare it "unknown":
G <- x ; Encoding(G[!validEnc(G)]) <- "unknown"
try( substr(x, 1,1) ) # gives 'invalid multibyte string' error in a UTF-8 locale
try( substr(G, 1,1) ) # works in a UTF-8 locale
nchar(G) # fine, too
## but it is not "more valid" typically:
all.equal(validEnc(x),
          validEnc(G)) # typically TRUE
```

`vector` *Vectors - Creation, Coercion, etc*

Description

A *vector* in R is either an atomic vector i.e., one of the atomic types, see 'Details', or of type (`typeof`) or mode `list` or `expression`.

`vector` produces a 'simple' vector of the given length and mode, where a 'simple' vector has no attribute, i.e., fulfills `is.null(attributes(.))`.

`as.vector`, a generic, attempts to coerce its argument into a vector of mode `mode` (the default is to coerce to whichever vector mode is most convenient): if the result is atomic (`is.atomic`), all attributes are removed. For `mode="any"`, see 'Details'.

`is.vector(x)` returns `TRUE` if `x` is a vector of the specified mode having no attributes *other than names*. For `mode="any"`, see 'Details'.

Usage

```
vector(mode = "logical", length = 0)
as.vector(x, mode = "any")
is.vector(x, mode = "any")
```

Arguments

mode	character string naming an atomic mode or "list" or "expression" or (except for vector) "any". Currently, is.vector() allows any type (see typeof) for mode, and when mode is not "any", is.vector(x, mode) is almost the same as typeof(x) == mode.
length	a non-negative integer specifying the desired length. For a long vector, i.e., length > .Machine$integer.max, it has to be of type "double". Supplying an argument of length other than one is an error.
x	an R object.

Details

The atomic modes are "logical", "integer", "numeric" (synonym "double"), "complex", "character" and "raw".

If mode = "any", is.vector may return TRUE for the atomic modes, list and expression. For any mode, it will return FALSE if x has any attributes except names. (This is incompatible with S.) On the other hand, as.vector removes *all* attributes including names for results of atomic mode.

For mode = "any", and atomic vectors x, as.vector(x) strips all attributes (including names), returning a simple atomic vector.
However, when x is of type "list" or "expression", as.vector(x) currently returns the argument x unchanged, unless there is an as.vector method for class(x).

Note that factors are *not* vectors; is.vector returns FALSE and as.vector converts a factor to a character vector for mode = "any".

Value

For vector, a vector of the given length and mode. Logical vector elements are initialized to FALSE, numeric vector elements to 0, character vector elements to "", raw vector elements to nul bytes and list/expression elements to NULL.

For as.vector, a vector (atomic or of type list or expression). All attributes are removed from the result if it is of an atomic mode, but not in general for a list or expression result. The default method handles 24 input types and 12 values of type: the details of most coercions are undocumented and subject to change.

For is.vector, TRUE or FALSE. is.vector(x, mode = "numeric") can be true for vectors of types "integer" or "double" whereas is.vector(x, mode = "double") can only be true for those of type "double".

Methods for as.vector()

Writers of methods for as.vector need to take care to follow the conventions of the default method. In particular

- Argument mode can be "any", any of the atomic modes, "list", "expression", "symbol", "pairlist" or one of the aliases "double" and "name".
- The return value should be of the appropriate mode. For mode = "any" this means an atomic vector or list or expression.
- Attributes should be treated appropriately: in particular when the result is an atomic vector there should be no attributes, not even names.
- is.vector(as.vector(x, m), m) should be true for any mode m, including the default "any".

  Currently this is not fulfilled in R when m == "any" and x is of type list or expression with attributes in addition to names — typically the case for (S3 or S4) objects (see is.object) which are lists internally.

Note

as.vector and is.vector are quite distinct from the meaning of the formal class "vector" in the **methods** package, and hence as(x, "vector") and is(x, "vector").

Note that as.vector(x) is not necessarily a null operation if is.vector(x) is true: any names will be removed from an atomic vector.

Non-vector modes "symbol" (synonym "name") and "pairlist" are accepted but have long been undocumented: they are used to implement as.name and as.pairlist, and those functions should preferably be used directly. None of the description here applies to those modes: see the help for the preferred forms.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c, is.numeric, is.list, etc.

Examples

```
df <- data.frame(x = 1:3, y = 5:7)
## Error:
try(as.vector(data.frame(x = 1:3, y = 5:7), mode = "numeric"))

x <- c(a = 1, b = 2)
is.vector(x)
as.vector(x)
all.equal(x, as.vector(x)) ## FALSE


###-- All the following are TRUE:
is.list(df)
! is.vector(df)
! is.vector(df, mode = "list")

is.vector(list(), mode = "list")
```

Vectorize *Vectorize a Scalar Function*

Description

`Vectorize` creates a function wrapper that vectorizes the action of its argument `FUN`.

Usage

```
Vectorize(FUN, vectorize.args = arg.names, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

Arguments

`FUN`	function to apply, found via `match.fun`.
`vectorize.args`	a character vector of arguments which should be vectorized. Defaults to all arguments of `FUN`.
`SIMPLIFY`	logical or character string; attempt to reduce the result to a vector, matrix or higher dimensional array; see the `simplify` argument of `sapply`.
`USE.NAMES`	logical; use names if the first ... argument has names, or if it is a character vector, use that character vector as the names.

Details

The arguments named in the `vectorize.args` argument to `Vectorize` are the arguments passed in the `...` list to `mapply`. Only those that are actually passed will be vectorized; default values will not. See the examples.

`Vectorize` cannot be used with primitive functions as they do not have a value for `formals`.

It also cannot be used with functions that have arguments named `FUN`, `vectorize.args`, `SIMPLIFY` or `USE.NAMES`, as they will interfere with the `Vectorize` arguments. See the `combn` example below for a workaround.

Value

A function with the same arguments as `FUN`, wrapping a call to `mapply`.

Examples

```
# We use rep.int as rep is primitive
vrep <- Vectorize(rep.int)
vrep(1:4, 4:1)
vrep(times = 1:4, x = 4:1)

vrep <- Vectorize(rep.int, "times")
vrep(times = 1:4, x = 42)

f <- function(x = 1:3, y) c(x, y)
```

```
vf <- Vectorize(f, SIMPLIFY = FALSE)
f(1:3, 1:3)
vf(1:3, 1:3)
vf(y = 1:3) # Only vectorizes y, not x

# Nonlinear regression contour plot, based on nls() example
require(graphics)
SS <- function(Vm, K, resp, conc) {
    pred <- (Vm * conc)/(K + conc)
    sum((resp - pred)^2 / pred)
}
vSS <- Vectorize(SS, c("Vm", "K"))
Treated <- subset(Puromycin, state == "treated")

Vm <- seq(140, 310, length.out = 50)
K <- seq(0, 0.15, length.out = 40)
SSvals <- outer(Vm, K, vSS, Treated$rate, Treated$conc)
contour(Vm, K, SSvals, levels = (1:10)^2, xlab = "Vm", ylab = "K")

# combn() has an argument named FUN
combnV <- Vectorize(function(x, m, FUNV = NULL) combn(x, m, FUN = FUNV),
                    vectorize.args = c("x", "m"))
combnV(4, 1:4)
combnV(4, 1:4, sum)
```

`warning` *Warning Messages*

Description

Generates a warning message that corresponds to its argument(s) and (optionally) the expression or function from which it was called.

Usage

```
warning(..., call. = TRUE, immediate. = FALSE, noBreaks. = FALSE,
        domain = NULL)
suppressWarnings(expr, classes = "warning")
```

Arguments

`...`	*either* zero or more objects which can be coerced to character (and which are pasted together with no separator) *or* a single condition object.
`call.`	logical, indicating if the call should become part of the warning message.
`immediate.`	logical, indicating if the warning should be output immediately, even if `getOption("warn") <= 0`. **NB:** this is not respected for condition objects.
`noBreaks.`	logical, indicating as far as possible the message should be output as a single line when `options(warn = 1)`.

expr	expression to evaluate.
domain	see gettext. If NA, messages will not be translated, see also the note in stop.
classes	character, indicating which classes of warnings should be suppressed.

Details

The result *depends* on the value of options("warn") and on handlers established in the executing code.

If a condition object is supplied it should be the only argument, and further arguments will be ignored, with a message. options(warn = 1) can be used to request an immediate report.

warning signals a warning condition by (effectively) calling signalCondition. If there are no handlers or if all handlers return, then the value of warn = getOption("warn") is used to determine the appropriate action. If warn is negative warnings are ignored; if it is zero they are stored and printed after the top–level function has completed; if it is one they are printed as they occur and if it is 2 (or larger) warnings are turned into errors. Calling warning(immediate. = TRUE) turns warn <= 0 into warn = 1 for this call only.

If warn is zero (the default), a read-only variable last.warning is created. It contains the warnings which can be printed via a call to warnings.

Warnings will be truncated to getOption("warning.length") characters, default 1000, indicated by [... truncated].

While the warning is being processed, a muffleWarning restart is available. If this restart is invoked with invokeRestart, then warning returns immediately.

An attempt is made to coerce other types of inputs to warning to character vectors.

suppressWarnings evaluates its expression in a context that ignores all warnings.

Value

The warning message as character string, invisibly.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

stop for fatal errors, message for diagnostic messages, warnings, and options with argument warn=.

gettext for the mechanisms for the automated translation of messages.

Examples

```
testit <- function() warning("testit")
testit() ## shows call
testit <- function() warning("problem in testit", call. = FALSE)
testit() ## no call
suppressWarnings(warning("testit"))
```

warnings *Print Warning Messages*

Description

warnings and its print method print the variable last.warning in a pleasing form.

Usage

```
warnings(...)

## S3 method for class 'warnings'
summary(object, ...)

## S3 method for class 'warnings'
print(x, tags,
      header = ngettext(n, "Warning message:\n", "Warning messages:\n"),
      ...)
## S3 method for class 'summary.warnings'
print(x, ...)
```

Arguments

...	arguments to be passed to cat (for warnings()).
object	a "warnings" object as returned by warnings().
x	a "warnings" or "summary.warnings" object.
tags	if not missing, a character vector of the same length as x, to "label" the messages. Defaults to paste0(seq_len(n), ": ") for $n \geq 2$ where n <- length(x).
header	a character string cat()ed before the messages are printed.

Details

See the description of options("warn") for the circumstances under which there is a last.warning object and warnings() is used. In essence this is if options(warn = 0) and warning has been called at least once.

Note that the length(last.warning) is maximally getOption("nwarnings") (at the time the warnings are generated) which is 50 by default. To increase, use something like

```
options(nwarnings = 10000)
```

It is possible that last.warning refers to the last recorded warning and not to the last warning, for example if options(warn) has been changed or if a catastrophic error occurred.

Value

`warnings()` returns an object of S3 class `"warnings"`, basically a named `list`. In R versions before 4.4.0, it returned `NULL` when there were no warnings, contrary to the above documentation.

`summary(<warnings>)` returns a `"summary.warnings"` object which is basically the `list` of unique warnings (`unique(object)`) with a `"counts"` attribute, somewhat experimentally.

Warning

It is undocumented where `last.warning` is stored nor that it is visible, and this is subject to change.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`warning`.

Examples

```
## NB this example is intended to be pasted in,
##    rather than run by example()
ow <- options("warn")
for(w in -1:1) {
   options(warn = w); cat("\n warn =", w, "\n")
   for(i in 1:3) { cat(i,"..\n"); m <- matrix(1:7, 3,4) }
   cat("--==--=--\n")
}
## at the end prints all three warnings, from the 'option(warn = 0)' above
options(ow) # reset to previous, typically 'warn = 0'
tail(warnings(), 2) # see the last two warnings only (via '[' method)

## Often the most useful way to look at many warnings:
summary(warnings())

op <- options(nwarnings = 10000) ## <- get "full statistics"
x <- 1:36; for(n in 1:13) for(m in 1:12) A <- matrix(x, n,m) # There were 105 warnings ...
summary(warnings())
options(op) # revert to previous (keeping 50 messages by default)
```

---

weekdays *Extract Parts of a POSIXt or Date Object*

Description

Extract the weekday, month or quarter, or the Julian time (days since some origin). These are generic functions: the methods for the internal date-time classes are documented here.

Usage

```
weekdays(x, abbreviate)
## S3 method for class 'POSIXt'
weekdays(x, abbreviate = FALSE)
## S3 method for class 'Date'
weekdays(x, abbreviate = FALSE)

months(x, abbreviate)
## S3 method for class 'POSIXt'
months(x, abbreviate = FALSE)
## S3 method for class 'Date'
months(x, abbreviate = FALSE)

quarters(x, abbreviate)
## S3 method for class 'POSIXt'
quarters(x, ...)
## S3 method for class 'Date'
quarters(x, ...)

julian(x, ...)
## S3 method for class 'POSIXt'
julian(x, origin = as.POSIXct("1970-01-01", tz = "GMT"), ...)
## S3 method for class 'Date'
julian(x, origin = as.Date("1970-01-01"), ...)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
abbreviate	logical vector (possibly recycled). Should the names be abbreviated?
origin	an length-one object inheriting from class "POSIXt" or "Date".
...	arguments for other methods.

Value

weekdays and months return a character vector of names in the locale in use, i.e., Sys.getlocale("LC_TIME").

quarters returns a character vector of "Q1" to "Q4".

julian returns the number of days (possibly fractional) since the origin, with the origin as a "origin" attribute. All time calculations in R are done ignoring leap-seconds.

Note

Other components such as the day of the month or the year are very easy to compute: just use as.POSIXlt and extract the relevant component. Alternatively (especially if the components are desired as character strings), use strftime.

See Also

DateTimeClasses, Date; Sys.getlocale("LC_TIME") crucially for months() and weekdays().

Examples

```
## first two are locale dependent:
weekdays(.leap.seconds)
months  (.leap.seconds)
quarters(.leap.seconds)

## Show how easily you get month, day, year, day (of {month, week, yr}), ... :
## (remember to count from 0 (!): mon = 0..11, wday = 0..6,  etc !!)

##' Transform (Time-)Date vector  to  convenient data frame :
dt2df <- function(dt, dName = deparse(substitute(dt))) {
    DF <- as.data.frame(unclass(as.POSIXlt( dt )))
    `names<-`(cbind(dt, DF, deparse.level=0L), c(dName, names(DF)))
}
## e.g.,
dt2df(.leap.seconds)    # date+time
dt2df(Sys.Date() + 0:9) # date

##' Even simpler:  Date -> Matrix - dropping time info {sec,min,hour, isdst}
d2mat <- function(x) simplify2array(unclass(as.POSIXlt(x))[4:7])
## e.g.,
d2mat(seq(as.Date("2000-02-02"), by=1, length.out=30)) # has R 1.0.0's release date


## Julian Day Number (JDN, https://en.wikipedia.org/wiki/Julian_day)
## is the number of days since noon UTC on the first day of 4317 BCE.
## in the proleptic Julian calendar.  To more recently, in
## 'Terrestrial Time' which differs from UTC by a few seconds
## See https://en.wikipedia.org/wiki/Terrestrial_Time
julian(Sys.Date(), -2440588) # from a day
floor(as.numeric(julian(Sys.time())) + 2440587.5) # from a date-time
```

---

which *Which indices are TRUE?*

Description

Give the TRUE indices of a logical object, allowing for array indices.

Usage

```
which(x, arr.ind = FALSE, useNames = TRUE)
arrayInd(ind, .dim, .dimnames = NULL, useNames = FALSE)
```

Arguments

`x`	a `logical` vector or array. `NA`s are allowed and omitted (treated as if `FALSE`).
`arr.ind`	logical; should **arr**ay **ind**ices be returned when `x` is an array? Anything other than a single true value is treated as false.
`ind`	integer-valued index vector, as resulting from `which(x)`.
`.dim`	`dim(.)` integer vector.
`.dimnames`	optional list of character `dimnames(.)`. If `useNames` is true, to be used for constructing dimnames for `arrayInd()` (and hence, `which(*, arr.ind=TRUE)`). If `names(.dimnames)` is not empty, these are used as column names. `.dimnames[[1]]` is used as row names.
`useNames`	logical indicating if the value of `arrayInd()` should have (non-null) dimnames at all.

Value

If `arr.ind == FALSE` (the default), an integer vector, or a double vector if `x` is a *long vector*, with `length` equal to `sum(x)`, i.e., to the number of `TRUE`s in `x`.

Basically, the result is `(1:length(x))[x]` in typical cases; more generally, including when `x` has `NA`'s, `which(x)` is `seq_along(x)[!is.na(x) & x]` plus `names` when `x` has.

If `arr.ind == TRUE` and `x` is an `array` (has a `dim` attribute), the result is `arrayInd(which(x), dim(x), dimnames(x))`, namely a matrix whose rows each are the indices of one element of `x`; see Examples below.

Note

Unlike most other base R functions this does not coerce `x` to logical: only arguments with `typeof` logical are accepted and others give an error.

Author(s)

Werner Stahel and Peter Holzer (ETH Zurich) proposed the `arr.ind` option.

See Also

`Logic`, `which.min` for the index of the minimum or maximum, and `match` for the first index of an element in a vector, i.e., for a scalar `a`, `match(a, x)` is equivalent to `min(which(x == a))` but much more efficient.

Examples

```
which(LETTERS == "R")
which(ll <- c(TRUE, FALSE, TRUE, NA, FALSE, FALSE, TRUE)) #> 1 3 7
names(ll) <- letters[seq(ll)]
which(ll)
which((1:12)%%2 == 0) # which are even?
which(1:10 > 3, arr.ind = TRUE)
```

```
( m <- matrix(1:12, 3, 4) )
div.3 <- m %% 3 == 0
which(div.3)
which(div.3, arr.ind = TRUE)
rownames(m) <- paste("Case", 1:3, sep = "_")
which(m %% 5 == 0, arr.ind = TRUE)

dim(m) <- c(2, 2, 3); m
which(div.3, arr.ind = FALSE)
which(div.3, arr.ind = TRUE)

vm <- c(m)
dim(vm) <- length(vm) #-- funny thing with  length(dim(...)) == 1
which(div.3, arr.ind = TRUE)
```

`which.min` *Where is the Min() or Max() or first TRUE or FALSE ?*

Description

Determines the location, i.e., index of the (first) minimum or maximum of a numeric (or logical) vector.

Usage

```
which.min(x)
which.max(x)
```

Arguments

`x`	numeric (logical, integer or double) vector or an R object for which the internal coercion to `double` works whose `min` or `max` is searched for.

Value

Missing and `NaN` values are discarded.

an `integer` or on 64-bit platforms, if `length(x) =: n` $\geq 2^{31}$ an integer valued `double` of length 1 or 0 (iff `x` has no non-`NA`s), giving the index of the *first* minimum or maximum respectively of `x`.

If this extremum is unique (or empty), the results are the same as (but more efficient than) `which(x == min(x, na.rm = TRUE))` or `which(x == max(x, na.rm = TRUE))` respectively.

Logical `x` – First `TRUE` or `FALSE`

For a `logical` vector `x` with both `FALSE` and `TRUE` values, `which.min(x)` and `which.max(x)` return the index of the first `FALSE` or `TRUE`, respectively, as `FALSE < TRUE`. However, `match(FALSE, x)` or `match(TRUE, x)` are typically *preferred*, as they do indicate mismatches.

Author(s)

Martin Maechler

See Also

`which`, `max.col`, `max`, etc.

Use `arrayInd()`, if you need array/matrix indices instead of 1D vector ones.

`which.is.max` in package **nnet** differs in breaking ties at random (and having a 'fuzz' in the definition of ties).

Examples

```
x <- c(1:4, 0:5, 11)
which.min(x)
which.max(x)

## it *does* work with NA's present, by discarding them:
presidents[1:30]
range(presidents, na.rm = TRUE)
which.min(presidents) # 28
which.max(presidents) #  2

## Find the first occurrence, i.e. the first TRUE, if there is at least one:
x <- rpois(10000, lambda = 10); x[sample.int(50, 20)] <- NA
## where is the first value >= 20 ?
which.max(x >= 20)

## Also works for lists (which can be coerced to numeric vectors):
which.min(list(A = 7, pi = pi)) ##  ->  c(pi = 2L)
```

---

with *Evaluate an Expression in a Data Environment*

Description

Evaluate an R expression in an environment constructed from data, possibly modifying (a copy of) the original data.

Usage

```
with(data, expr, ...)
within(data, expr, ...)
## S3 method for class 'list'
within(data, expr, keepAttrs = TRUE, ...)
```

Arguments

data	data to use for constructing an environment. For the default with method this may be an environment, a list, a data frame, or an integer as in sys.call. For within, it can be a list or a data frame.

expr	expression to evaluate; particularly for within() often a "compound" expression, i.e., of the form
	{
	a <- somefun()
	b <- otherfun()
	.....
	rm(unused1, temp)
	}

keepAttrs	for the list method of within(), a logical specifying if the resulting list should keep the attributes from data and have its names in the same order. Often this is unneeded as the result is a *named* list anyway, and then keepAttrs = FALSE is more efficient.
...	arguments to be passed to (future) methods.

Details

with is a generic function that evaluates expr in a local environment constructed from data. The environment has the caller's environment as its parent. This is useful for simplifying calls to modeling functions. (Note: if data is already an environment then this is used with its existing parent.)

Note that assignments within expr take place in the constructed environment and not in the user's workspace.

within is similar, except that it examines the environment after the evaluation of expr and makes the corresponding modifications to a copy of data (this may fail in the data frame case if objects are created which cannot be stored in a data frame), and returns it. within can be used as an alternative to transform.

Value

For with, the value of the evaluated expr. For within, the modified object.

Note

For *interactive* use this is very effective and nice to read. For *programming* however, i.e., in one's functions, more care is needed, and typically one should refrain from using with(), as, e.g., variables in data may accidentally override local variables, see the reference.

Further, when using modeling or graphics functions with an explicit data argument (and typically using formulas), it is typically preferred to use the data argument of that function rather than to use with(data, ...).

References

Thomas Lumley (2003) *Standard nonstandard evaluation rules*. https://developer.r-project.org/nonstandard-eval.pdf

See Also

evalq, attach, assign, transform.

Examples

```
with(mtcars, mpg[cyl == 8  &  disp > 350])
    # is the same as, but nicer than
mtcars$mpg[mtcars$cyl == 8  &  mtcars$disp > 350]

require(stats); require(graphics)

# examples from glm:
with(data.frame(u = c(5,10,15,20,30,40,60,80,100),
                lot1 = c(118,58,42,35,27,25,21,19,18),
                lot2 = c(69,35,26,21,18,16,13,12,12)),
    list(summary(glm(lot1 ~ log(u), family = Gamma)),
         summary(glm(lot2 ~ log(u), family = Gamma))))

aq <- within(airquality, {     # Notice that multiple vars can be changed
    lOzone <- log(Ozone)
    Month <- factor(month.abb[Month])
    cTemp <- round((Temp - 32) * 5/9, 1) # From Fahrenheit to Celsius
    S.cT <- Solar.R / cTemp  # using the newly created variable
    rm(Day, Temp)
})
head(aq)




# example from boxplot:
with(ToothGrowth, {
    boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
            subset = (supp == "VC"), col = "yellow",
            main = "Guinea Pigs' Tooth Growth",
            xlab = "Vitamin C dose mg",
            ylab = "tooth length", ylim = c(0, 35))
    boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
            subset = supp == "OJ", col = "orange")
    legend(2, 9, c("Ascorbic acid", "Orange juice"),
           fill = c("yellow", "orange"))
})

# alternate form that avoids subset argument:
with(subset(ToothGrowth, supp == "VC"),
     boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
             col = "yellow", main = "Guinea Pigs' Tooth Growth",
             xlab = "Vitamin C dose mg",
             ylab = "tooth length", ylim = c(0, 35)))
with(subset(ToothGrowth,  supp == "OJ"),
     boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
             col = "orange"))
```

```
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))
```

`withVisible` *Return both a Value and its Visibility*

Description

This function evaluates an expression, returning it in a two element list containing its value and a flag showing whether it would automatically print.

Usage

```
withVisible(x)
```

Arguments

`x`	an expression to be evaluated.

Details

The argument, *not* an `expression` object, rather an (unevaluated function) `call`, is evaluated in the caller's context.

This is a primitive function.

Value

`value`	The value of `x` after evaluation.
`visible`	logical; whether the value would auto-print.

See Also

`invisible`, `eval`; `withAutoprint()` calls `source()` which itself uses `withVisible()` in order to correctly "auto print".

Examples

```
x <- 1
withVisible(x <- 1) # *$visible is FALSE
x
withVisible(x)      # *$visible is TRUE

# Wrap the call in evalq() for special handling

df <- data.frame(a = 1:5, b = 1:5)
evalq(withVisible(a + b), envir = df)
```

write *Write Data to a File*

Description

Write data x to a file or other connection.
As it simply calls cat(), less formatting happens than with print()ing. If x is a matrix you need to transpose it (and typically set ncolumns) to get the columns in file the same as those in the internal representation.

Whereas atomic vectors (numeric, character, etc, including matrices) are written plainly, i.e., without any names, less simple vector-like objects such as "factor", "Date", or "POSIXt" may be formatted to character before writing.

Usage

```
write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE, sep = " ")
```

Arguments

x	the data to be written out.
file	a connection, or a character string naming the file to write to. If "", print to the standard output connection. When .Platform$OS.type != "windows", and it is "|cmd", the output is piped to the command given by 'cmd'.
ncolumns	the number of columns to write the data in.
append	if TRUE the data x are appended to the connection.
sep	a string used to separate columns. Using sep = "\t" gives tab delimited output; default is " ".

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

write is a wrapper for cat, which gives further details on the format used.

write.table for matrix and data frame objects, writeLines for lines of text, and scan for reading data.

saveRDS and save are often preferable (for writing any R objects).

Examples

```
# Demonstrate default ncolumns, writing to the console
write(month.abb,  "")  # 1 element  per line for "character"
write(stack.loss, "")  # 5 elements per line for "numeric"

# Build a file with sequential calls
fil <- tempfile("data")
write("# Model settings", fil)
write(month.abb, fil, ncolumns = 6, append = TRUE)
write("\n# Initial parameter values", fil, append = TRUE)
write(sqrt(stack.loss), fil, append = TRUE)
if(interactive()) file.show(fil)
unlink(fil) # tidy up
```

---

writeLines *Write Lines to a Connection*

Description

Write text lines to a connection.

Usage

```
writeLines(text, con = stdout(), sep = "\n", useBytes = FALSE)
```

Arguments

text	a character vector.
con	a connection object or a character string.
sep	character string. A string to be written to the connection after each line of text.
useBytes	logical. See 'Details'.

Details

If the con is a character string, the function calls file to obtain a file connection which is opened for the duration of the function call. (tilde expansion of the file path is done by file.)

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call in "wt" mode and then closed again.

Normally writeLines is used with a text-mode connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows). For more control, open a binary connection and specify the precise value you want written to the file in sep. For even more control, use writeChar on a binary connection.

useBytes is for expert use. Normally (when false) character strings with marked encodings are converted to the current encoding before being passed to the connection (which might do further re-encoding). useBytes = TRUE suppresses the re-encoding of marked strings so they are passed byte-by-byte to the connection: this can be useful when strings have already been re-encoded by e.g. iconv. (It is invoked automatically for strings with marked encoding "bytes".)

See Also

connections, writeChar, writeBin, readLines, cat

---

xtfrm *Auxiliary Function for Sorting and Ranking*

Description

A generic auxiliary function that produces a numeric vector which will sort in the same order as x.

Usage

```
xtfrm(x)
```

Arguments

x	an R object.

Details

This is a special case of ranking, but as a less general function than rank is more suitable to be made generic. The default method is similar to rank(x, ties.method = "min", na.last = "keep"), so NA values are given rank NA and all tied values are given equal integer rank.

The factor method extracts the codes.

The default method will unclass the object if is.numeric(x) is true but otherwise make use of == and > methods for the class of x[i] (for integers i), and the is.na method for the class of x, but might be rather slow when doing so.

This is an internal generic primitive, so S3 or S4 methods can be written for it. Differently to other internal generics, the default method is called explicitly when no other dispatch has happened.

Value

A numeric (usually integer) vector of the same length as x.

See Also

rank, sort, order.

zapsmall *Rounding of Numbers: Zapping Small Ones to Zero*

Description

`zapsmall` determines a `digits` argument `dr` for calling `round(x, digits = dr)` such that values close to zero (compared with the maximal absolute value in the vector) are 'zapped', i.e., replaced by `0`.

Usage

```
zapsmall(x, digits = getOption("digits"),
         mFUN = function(x, ina) max(abs(x[!ina])),
         min.d = 0L)
```

Arguments

`x`	a numeric or complex vector or any R number-like object which has a `round` method and basic arithmetic methods including `log10()`.
`digits`	integer indicating the precision to be used.
`mFUN`	a `function(x, ina)` of the numeric (or complex) `x` and the `logical` `ina := is.na(x)` returning a positive number in the order of magnitude of the maximal `abs(x)` value. The default is back compatible but not robust, and e.g., not very useful when `x` has infinite entries.
`min.d`	an integer specifying the minimal number of digits to use in the resulting `round(x, digits=*)` call when `mFUN(*) > 0`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

Examples

```
x2 <- pi * 100^(-2:2)/10
   print(  x2, digits = 4)
zapsmall(  x2) # automatical digits
zapsmall(  x2, digits = 4)
zapsmall(c(x2, Inf)) # round()s to integer ..
zapsmall(c(x2, Inf), min.d=-Inf) # everything  is small wrt  Inf

(z <- exp(1i*0:4*pi/2))
zapsmall(z)

zapShow <- function(x, ...) rbind(orig = x, zapped = zapsmall(x, ...))
zapShow(x2)

## using a *robust* mFUN
mF_rob <- function(x, ina) boxplot.stats(x, do.conf=FALSE)$stats[5]
```

```
## with robust mFUN(), 'Inf' is no longer distorting the picture:
zapShow(c(x2, Inf), mFUN = mF_rob)
zapShow(c(x2, Inf), mFUN = mF_rob, min.d = -5) # the same
zapShow(c(x2, 999), mFUN = mF_rob) # same *rounding* as w/ Inf
zapShow(c(x2, 999), mFUN = mF_rob, min.d =  3) # the same
zapShow(c(x2, 999), mFUN = mF_rob, min.d =  8) # small diff
```

---

zpackages *Listing of Packages*

Description

.packages returns information about package availability.

Usage

```
.packages(all.available = FALSE, lib.loc = NULL)
```

Arguments

all.available	logical; if TRUE return a character vector of all available packages in lib.loc.
lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to .libPaths().

Details

.packages() returns the names of the currently attached packages *invisibly* whereas .packages(all.available = TRUE) gives (visibly) *all* packages available in the library location path lib.loc.

For a package to be regarded as being 'available' it must have valid metadata (and hence be an installed package). However, this will report a package as available if the metadata does not match the directory name: use find.package to confirm that the metadata match or installed.packages for a much slower but more comprehensive check of 'available' packages.

Value

A character vector of package base names, invisible unless all.available = TRUE.

Note

.packages(all.available = TRUE) is not a way to find out if a small number of packages are available for use: not only is it expensive when thousands of packages are installed, it is an incomplete test. See the help for find.package for why require should be used.

Author(s)

R core; Guido Masarotto for the all.available = TRUE part of .packages.

See Also

library, .libPaths, installed.packages.

Examples

```
(.packages())               # maybe just "base"
.packages(all.available = TRUE) # return all available as character vector
require(splines)
(.packages())               # "splines", too
detach("package:splines")
```

---

zutils *Miscellaneous Internal/Programming Utilities*

Description

Miscellaneous internal/programming utilities.

Usage

```
.standard_regexps()
```

Details

.standard_regexps returns a list of 'standard' regexps, including elements named valid_package_name and valid_package_version with the obvious meanings. The regexps are not anchored.

Chapter 2

The `compiler` package

`compile` *Byte Code Compiler*

Description

These functions provide an interface to a byte code compiler for R.

Usage

```
cmpfun(f, options = NULL)
compile(e, env = .GlobalEnv, options = NULL, srcref = NULL)
cmpfile(infile, outfile, ascii = FALSE, env = .GlobalEnv,
        verbose = FALSE, options = NULL, version = NULL)
loadcmp(file, envir = .GlobalEnv, chdir = FALSE)
disassemble(code)
enableJIT(level)
compilePKGS(enable)
getCompilerOption(name, options)
setCompilerOptions(...)
```

Arguments

`f`	a closure.
`options`	list of named compiler options: see 'Details'.
`env`	the top level environment for the compiling.
`srcref`	initial source reference for the expression.
`file`, `infile`, `outfile`	pathnames; `outfile` defaults to `infile` with a '`.Rc`' extension in place of any existing extension.
`ascii`	logical; should the compiled file be saved in ascii format?
`verbose`	logical; should the compiler show what is being compiled?

version	the workspace format version to use. NULL specifies the current default format (3). Version 1 was the default from R 0.99.0 to R 1.3.1 and version 2 from R 1.4.0 to 3.5.0. Version 3 is supported from R 3.5.0.
envir	environment to evaluate loaded expressions in.
chdir	logical; change directory before evaluation?
code	byte code expression or compiled closure
e	expression to compile.
level	integer; the JIT level to use (0 to 3, or negative to *return* it).
enable	logical; enable compiling packages if TRUE.
name	character string; name of option to return.
...	named compiler options to set.

Details

The function cmpfun compiles the body of a closure and returns a new closure with the same formals and the body replaced by the compiled body expression.

compile compiles an expression into a byte code object; the object can then be evaluated with eval.

cmpfile parses the expressions in infile, compiles them, and writes the compiled expressions to outfile. If outfile is not provided, it is formed from infile by replacing or appending a .Rc suffix.

loadcmp is used to load compiled files. It is similar to sys.source, except that its default loading environment is the global environment rather than the base environment.

disassemble produces a printed representation of the code that may be useful to give a hint of what is going on.

enableJIT enables or disables just-in-time (JIT) compilation. JIT is disabled if the argument is 0. If level is 1 then larger closures are compiled before their first use. If level is 2, then some small closures are also compiled before their second use. If level is 3 then in addition all top level loops are compiled before they are executed. JIT level 3 requires the compiler option optimize to be 2 or 3. The JIT level can also be selected by starting R with the environment variable R_ENABLE_JIT set to one of these values. Calling enableJIT with a negative argument returns the current JIT level. The default JIT level is 3.

compilePKGS enables or disables compiling packages when they are installed. This requires that the package uses lazy loading as compilation occurs as functions are written to the lazy loading data base. This can also be enabled by starting R with the environment variable _R_COMPILE_PKGS_ set to a positive integer value. This should not be enabled outside package installation, because it causes any serialized function to be compiled, which comes with time and space overhead. R_COMPILE_PKGS can be used, instead, to instruct INSTALL to enable/disable compilation of packages during installation.

Currently the compiler warns about a variety of things. It does this by using cat to print messages. Eventually this should use the condition handling mechanism.

The options argument can be used to control compiler operation. There are currently four options: optimize, suppressAll, suppressUndefined, and suppressNoSuperAssignVar. optimize specifies the optimization level, an integer from 0 to 3 (the current out-of-the-box default is 2).

suppressAll should be a scalar logical; if TRUE no messages will be shown (this is the default). suppressUndefined can be TRUE to suppress all messages about undefined variables, or it can be a character vector of the names of variables for which messages should not be shown. suppressNoSuperAssignVar can be TRUE to suppress messages about super assignments to a variable for which no binding is visible at compile time. During compilation of packages, suppressAll is currently FALSE, suppressUndefined is TRUE and suppressNoSuperAssignVar is TRUE.

getCompilerOption returns the value of the specified option. The default value is returned unless a value is supplied in the options argument; the options argument is primarily for internal use. setCompilerOption sets the default option values. Options to set are identified by argument names, e.g. setCompilerOptions(suppressAll = TRUE, optimize = 3). It returns a named list of the previous values.

Calling the compiler a byte code compiler is actually a bit of a misnomer: the external representation of code objects currently uses int operands, and when compiled with gcc the internal representation is actually threaded code rather than byte code.

Author(s)

Luke Tierney

Examples

```
oldJIT <- enableJIT(0)
# a simple example
f <- function(x) x+1
fc <- cmpfun(f)
fc(2)
disassemble(fc)

# old R version of lapply
la1 <- function(X, FUN, ...) {
    FUN <- match.fun(FUN)
    if (!is.list(X))
 X <- as.list(X)
    rval <- vector("list", length(X))
    for(i in seq_along(X))
 rval[i] <- list(FUN(X[[i]], ...))
    names(rval) <- names(X)    # keep `names' !
    return(rval)
}
# a small variation
la2 <- function(X, FUN, ...) {
    FUN <- match.fun(FUN)
    if (!is.list(X))
 X <- as.list(X)
    rval <- vector("list", length(X))
    for(i in seq_along(X)) {
        v <- FUN(X[[i]], ...)
        if (is.null(v)) rval[i] <- list(v)
        else rval[[i]] <- v
    }
    names(rval) <- names(X)    # keep `names' !
```

```
    return(rval)
}
# Compiled versions
la1c <- cmpfun(la1)
la2c <- cmpfun(la2)
# some timings
x <- 1:10
y <- 1:100

system.time(for (i in 1:10000) lapply(x, is.null))
system.time(for (i in 1:10000) la1(x, is.null))
system.time(for (i in 1:10000) la1c(x, is.null))
system.time(for (i in 1:10000) la2(x, is.null))
system.time(for (i in 1:10000) la2c(x, is.null))
system.time(for (i in 1:1000) lapply(y, is.null))
system.time(for (i in 1:1000) la1(y, is.null))
system.time(for (i in 1:1000) la1c(y, is.null))
system.time(for (i in 1:1000) la2(y, is.null))
system.time(for (i in 1:1000) la2c(y, is.null))


enableJIT(oldJIT)
```

Chapter 3

The datasets package

`datasets-package` *The R Datasets Package*

Description

Base R datasets

Details

This package contains a variety of datasets. For a complete list, use `library(help = "datasets")`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

`ability.cov` *Ability and Intelligence Tests*

Description

Six tests were given to 112 individuals. The covariance matrix is given in this object.

Usage

```
ability.cov
```

Details

The tests are described as

general: a non-verbal measure of general intelligence using Cattell's culture-fair test.

picture: a picture-completion test

blocks: block design

maze: mazes

reading: reading comprehension

vocab: vocabulary

Bartholomew gives both covariance and correlation matrices, but these are inconsistent. Neither are in the original paper.

Source

Bartholomew, D. J. (1987). *Latent Variable Analysis and Factor Analysis*. Griffin.

Bartholomew, D. J. and Knott, M. (1990). *Latent Variable Analysis and Factor Analysis*. Second Edition, Arnold.

References

Smith, G. A. and Stanley G. (1983). Clocking g: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368. doi:10.1016/01602896(83)900107.

Examples

```
require(stats)
(ability.FA <- factanal(factors = 1, covmat = ability.cov))
update(ability.FA, factors = 2)
## The signs of factors and hence the signs of correlations are
## arbitrary with promax rotation.
update(ability.FA, factors = 2, rotation = "promax")
```

---

airmiles *Passenger Miles on Commercial US Airlines, 1937–1960*

Description

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

Usage

```
airmiles
```

Format

A time series of 24 observations; yearly, 1937–1960.

Source

F.A.A. Statistical Handbook of Aviation.

References

Brown, R. G. (1963) *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall.

Examples

```
require(graphics)
plot(airmiles, main = "airmiles data",
     xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

AirPassengers *Monthly Airline Passenger Numbers 1949-1960*

Description

The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960.

Usage

```
AirPassengers
```

Format

A monthly time series, in thousands.

Source

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) *Time Series Analysis, Forecasting and Control.* Third Edition. Holden-Day. Series G.

Examples

```
## The classic 'airline model', by full ML
(fit <- arima(log10(AirPassengers), c(0, 1, 1),
              seasonal = list(order = c(0, 1, 1), period = 12)))
update(fit, method = "CSS")
update(fit, x = window(log10(AirPassengers), start = 1954))
pred <- predict(fit, n.ahead = 24)
tl <- pred$pred - 1.96 * pred$se
tu <- pred$pred + 1.96 * pred$se
ts.plot(AirPassengers, 10^tl, 10^tu, log = "y", lty = c(1, 2, 2))
```

```
## full ML fit is the same if the series is reversed, CSS fit is not
ap0 <- rev(log10(AirPassengers))
attributes(ap0) <- attributes(AirPassengers)
arima(ap0, c(0, 1, 1), seasonal = list(order = c(0, 1, 1), period = 12))
arima(ap0, c(0, 1, 1), seasonal = list(order = c(0, 1, 1), period = 12),
      method = "CSS")

## Structural Time Series
ap <- log10(AirPassengers) - 2
(fit <- StructTS(ap, type = "BSM"))
par(mfrow = c(1, 2))
plot(cbind(ap, fitted(fit)), plot.type = "single")
plot(cbind(ap, tsSmooth(fit)), plot.type = "single")
```

`airquality` *New York Air Quality Measurements*

Description

Daily air quality measurements in New York, May to September 1973.

Usage

```
airquality
```

Format

A data frame with 153 observations on 6 variables.

`[,1]`	`Ozone`	numeric	Ozone (ppb)
`[,2]`	`Solar.R`	numeric	Solar R (lang)
`[,3]`	`Wind`	numeric	Wind (mph)
`[,4]`	`Temp`	numeric	Temperature (degrees F)
`[,5]`	`Month`	numeric	Month (1–12)
`[,6]`	`Day`	numeric	Day of month (1–31)

Details

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- `Ozone`: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- `Solar.R`: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- `Wind`: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- `Temp`: Maximum daily temperature in degrees Fahrenheit at LaGuardia Airport.

Source

The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Examples

```
require(graphics)
pairs(airquality, panel = panel.smooth, main = "airquality data")
```

---

anscombe *Anscombe's Quartet of 'Identical' Simple Linear Regressions*

Description

Four x-y datasets which have the same traditional statistical properties (mean, variance, correlation, regression line, etc.), yet are quite different.

Usage

```
anscombe
```

Format

A data frame with 11 observations on 8 variables.

`x1` == `x2` == `x3`	the integers 4:14, specially arranged
`x4`	values 8 and 19
`y1`, `y2`, `y3`, `y4`	numbers in (3, 12.5) with mean 7.5 and standard deviation 2.03

Source

Tufte, Edward R. (1989). *The Visual Display of Quantitative Information*, 13–14. Graphics Press.

References

Anscombe, Francis J. (1973). Graphs in statistical analysis. *The American Statistician*, **27**, 17–21. doi:10.2307/2682899.

Examples

```
require(stats); require(graphics)
summary(anscombe)

##-- now some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
mods <- setNames(as.list(1:4), paste0("lm", 1:4))
for(i in 1:4) {
  ff[2:3] <- lapply(paste0(c("y","x"), i), as.name)
  ## or   ff[[2]] <- as.name(paste0("y", i))
  ##      ff[[3]] <- as.name(paste0("x", i))
  mods[[i]] <- lmi <- lm(ff, data = anscombe)
  print(anova(lmi))
}

## See how close they are (numerically!)
sapply(mods, coef)
lapply(mods, function(fm) coef(summary(fm)))

## Now, do what you should have done in the first place: PLOTS
op <- par(mfrow = c(2, 2), mar = 0.1+c(4,4,1,1), oma =  c(0, 0, 2, 0))
for(i in 1:4) {
  ff[2:3] <- lapply(paste0(c("y","x"), i), as.name)
  plot(ff, data = anscombe, col = "red", pch = 21, bg = "orange", cex = 1.2,
       xlim = c(3, 19), ylim = c(3, 13))
  abline(mods[[i]], col = "blue")
}
mtext("Anscombe's 4 Regression data sets", outer = TRUE, cex = 1.5)
par(op)
```

---

attenu *The Joyner–Boore Attenuation Data*

Description

This data gives peak accelerations measured at various observation stations for 23 earthquakes in California. The data have been used by various workers to estimate the attenuating affect of distance on ground acceleration.

Usage

```
attenu
```

Format

A data frame with 182 observations on 5 variables.

[,1]	event	numeric	Event Number
[,2]	mag	numeric	Moment Magnitude

[,3]	station	factor	Station Number
[,4]	dist	numeric	Station-hypocenter distance (km)
[,5]	accel	numeric	Peak acceleration (g)

Source

Joyner, W.B., D.M. Boore and R.D. Porcella (1981). Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California earthquake. USGS Open File report 81-365. Menlo Park, Ca.

References

Boore, D. M. and Joyner, W. B.(1982). The empirical prediction of ground motion, *Bulletin of the Seismological Society of America*, **72**, S269–S268.

Bolt, B. A. and Abrahamson, N. A. (1982). New attenuation relations for peak and expected accelerations of strong ground motion. *Bulletin of the Seismological Society of America*, **72**, 2307–2321.

Bolt B. A. and Abrahamson, N. A. (1983). Reply to W. B. Joyner & D. M. Boore's "Comments on: New attenuation relations for peak and expected accelerations for peak and expected accelerations of strong ground motion", *Bulletin of the Seismological Society of America*, **73**, 1481–1483.

Brillinger, D. R. and Preisler, H. K. (1984). An exploratory analysis of the Joyner-Boore attenuation data, *Bulletin of the Seismological Society of America*, **74**, 1441–1449.

Brillinger, D. R. and Preisler, H. K. (1984). *Further analysis of the Joyner-Boore attenuation data*. Manuscript.

Examples

```
require(graphics)
## check the data class of the variables
sapply(attenu, data.class)
summary(attenu)
pairs(attenu, main = "attenu data")
coplot(accel ~ dist | as.factor(event), data = attenu, show.given = FALSE)
coplot(log(accel) ~ log(dist) | as.factor(event),
       data = attenu, panel = panel.smooth, show.given = FALSE)
```

---

attitude *The Chatterjee–Price Attitude Data*

Description

From a survey of the clerical employees of a large financial organization, the data are aggregated from the questionnaires of the approximately 35 employees for each of 30 (randomly selected) departments. The numbers give the percent proportion of favourable responses to seven questions in each department.

Usage

```
attitude
```

Format

A data frame with 30 observations on 7 variables. The first column are the short names from the reference, the second one the variable names in the data frame:

Y	rating	numeric	Overall rating
X[1]	complaints	numeric	Handling of employee complaints
X[2]	privileges	numeric	Does not allow special privileges
X[3]	learning	numeric	Opportunity to learn
X[4]	raises	numeric	Raises based on performance
X[5]	critical	numeric	Too critical
X[6]	advance	numeric	Advancement

Source

Chatterjee, S. and Price, B. (1977) *Regression Analysis by Example*. New York: Wiley. (Section 3.7, p.68ff of 2nd ed.(1991).)

Examples

```
require(stats); require(graphics)
pairs(attitude, main = "attitude data")
summary(attitude)
summary(fm1 <- lm(rating ~ ., data = attitude))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
summary(fm2 <- lm(rating ~ complaints, data = attitude))
plot(fm2)
par(opar)
```

---

austres *Quarterly Time Series of the Number of Australian Residents*

Description

Numbers (in thousands) of Australian residents measured quarterly from March 1971 to March 1994. The object is of class `"ts"`.

Usage

```
austres
```

Source

P. J. Brockwell and R. A. Davis (1996) *Introduction to Time Series and Forecasting.* Springer

`beavers` *Body Temperature Series of Two Beavers*

Description

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

Usage

```
beaver1
beaver2
```

Format

The `beaver1` data frame has 114 rows and 4 columns on body temperature measurements at 10 minute intervals.

The `beaver2` data frame has 100 rows and 4 columns on body temperature measurements at 10 minute intervals.

The variables are as follows:

day Day of observation (in days since the beginning of 1990), December 12–13 (`beaver1`) and November 3–4 (`beaver2`).

time Time of observation, in the form `0330` for 3:30am

temp Measured body temperature in degrees Celsius.

activ Indicator of activity outside the retreat.

Note

The observation at 22:20 is missing in `beaver1`.

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry.* New York: John Wiley and Sons.

Examples

```
require(graphics)
(yl <- range(beaver1$temp, beaver2$temp))

beaver.plot <- function(bdat, ...) {
  nam <- deparse(substitute(bdat))
  with(bdat, {
```

```
    # Hours since start of day:
    hours <- time %/% 100 + 24*(day - day[1]) + (time %% 100)/60
    plot (hours, temp, type = "l", ...,
          main = paste(nam, "body temperature"))
    abline(h = 37.5, col = "gray", lty = 2)
    is.act <- activ == 1
    points(hours[is.act], temp[is.act], col = 2, cex = .8)
  })
}
op <- par(mfrow = c(2, 1), mar = c(3, 3, 4, 2), mgp = 0.9 * 2:0)
 beaver.plot(beaver1, ylim = yl)
 beaver.plot(beaver2, ylim = yl)
par(op)
```

`BJsales` *Sales Data with Leading Indicator*

Description

The sales time series `BJsales` and leading indicator `BJsales.lead` each contain 150 observations. The objects are of class `"ts"`.

Usage

```
BJsales
BJsales.lead
```

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at https://robjhyndman.com/TSDL/

References

G. E. P. Box and G. M. Jenkins (1976): *Time Series Analysis, Forecasting and Control*, Holden-Day, San Francisco, p. 537.

P. J. Brockwell and R. A. Davis (1991): *Time Series: Theory and Methods*, Second edition, Springer Verlag, NY, pp. 414.

BOD *Biochemical Oxygen Demand*

Description

The BOD data frame has 6 rows and 2 columns giving the biochemical oxygen demand versus time in an evaluation of water quality.

Usage

```
BOD
```

Format

This data frame contains the following columns:

Time A numeric vector giving the time of the measurement (days).

demand A numeric vector giving the biochemical oxygen demand (mg/l).

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.4.

Originally from Marske (1967), *Biochemical Oxygen Demand Data Interpretation Using Sum of Squares Surface* M.Sc. Thesis, University of Wisconsin – Madison.

Examples

```
require(stats)
# simplest form of fitting a first-order model to these data
fm1 <- nls(demand ~ A*(1-exp(-exp(lrc)*Time)), data = BOD,
           start = c(A = 20, lrc = log(.35)))
coef(fm1)
fm1

# using the plinear algorithm  (trace o/p differs by platform)
fm2 <- nls(demand ~ (1-exp(-exp(lrc)*Time)), data = BOD,
           start = c(lrc = log(.35)), algorithm = "plinear", trace = TRUE)

# using a self-starting model
fm3 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
summary(fm3)
```

cars *Speed and Stopping Distances of Cars*

Description

The data give the speed of cars and the distances taken to stop. Note that the data were recorded in the 1920s.

Usage

```
cars
```

Format

A data frame with 50 observations on 2 variables.

[,1]	speed	numeric	Speed (mph)
[,2]	dist	numeric	Stopping distance (ft)

Source

Ezekiel, M. (1930) *Methods of Correlation Analysis*. Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats); require(graphics)
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1)
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
title(main = "cars data")
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy")
title(main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
summary(fm1 <- lm(log(dist) ~ log(speed), data = cars))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)

## An example of polynomial regression
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
    las = 1, xlim = c(0, 25))
d <- seq(0, 25, length.out = 200)
for(degree in 1:4) {
```

```
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep = "."), fm)
  lines(d, predict(fm, data.frame(speed = d)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)
```

`ChickWeight` *Weight versus age of chicks on different diets*

Description

The `ChickWeight` data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks.

Usage

```
ChickWeight
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

weight a numeric vector giving the body weight of the chick (gm).

Time a numeric vector giving the number of days since birth when the measurement was made.

Chick an ordered factor with levels `18` < ... < `48` giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

Diet a factor with levels 1, ..., 4 indicating which experimental diet the chick received.

Details

The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups on chicks on different protein diets.

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Crowder, M. and Hand, D. (1990), *Analysis of Repeated Measures*, Chapman and Hall (example 5.3)

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall (table A.2)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

See Also

SSlogis for models fitted to this dataset.

Examples

```
require(graphics)
coplot(weight ~ Time | Chick, data = ChickWeight,
       type = "b", show.given = FALSE)
```

---

chickwts *Chicken Weights by Feed Type*

Description

An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens.

Usage

```
chickwts
```

Format

A data frame with 71 observations on the following 2 variables.

`weight` a numeric variable giving the chick weight.

`feed` a factor giving the feed type.

Details

Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types.

Source

Anonymous (1948) *Biometrika*, **35**, 214.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
boxplot(weight ~ feed, data = chickwts, col = "lightgray",
    varwidth = TRUE, notch = TRUE, main = "chickwt data",
    ylab = "Weight at six weeks (gm)")
anova(fm1 <- lm(weight ~ feed, data = chickwts))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

CO2 *Carbon Dioxide Uptake in Grass Plants*

Description

The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*.

Usage

```
CO2
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

Plant an ordered factor with levels `Qn1` < `Qn2` < `Qn3` < ... < `Mc1` giving a unique identifier for each plant.

Type a factor with levels `Quebec` `Mississippi` giving the origin of the plant

Treatment a factor with levels `nonchilled` `chilled`

conc a numeric vector of ambient carbon dioxide concentrations (mL/L).

uptake a numeric vector of carbon dioxide uptake rates (μmol/m^2 sec).

Details

The CO_2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO_2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Potvin, C., Lechowicz, M. J. and Tardif, S. (1990) "The statistical analysis of ecophysiological response curves obtained from experiments involving repeated measures", *Ecology*, **71**, 1389–1400.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)

coplot(uptake ~ conc | Plant, data = CO2, show.given = FALSE, type = "b")
## fit the data for the first plant
fm1 <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
   data = CO2, subset = Plant == "Qn1")
summary(fm1)
## fit each plant separately
fmlist <- list()
for (pp in levels(CO2$Plant)) {
  fmlist[[pp]] <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
      data = CO2, subset = Plant == pp)
}
## check the coefficients by plant
print(sapply(fmlist, coef), digits = 3)
```

---

co2 *Mauna Loa Atmospheric CO2 Concentration*

Description

Atmospheric concentrations of CO_2 are expressed in parts per million (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale.

Usage

```
co2
```

Format

A time series of 468 observations; monthly from 1959 to 1997.

Details

The values for February, March and April of 1964 were missing and have been obtained by interpolating linearly between the values for January and May of 1964.

Source

Keeling, C. D. and Whorf, T. P., Scripps Institution of Oceanography (SIO), University of California, La Jolla, California USA 92093-0220.

https://scrippsco2.ucsd.edu/data/atmospheric_co2/.

Note that the data are subject to revision (based on recalibration of standard gases) by the Scripps institute, and hence may not agree exactly with the data provided by R.

References

Cleveland, W. S. (1993) *Visualizing Data*. New Jersey: Summit Press.

Examples

```
require(graphics)
plot(co2, ylab = expression("Atmospheric concentration of CO"[2]),
     las = 1)
title(main = "co2 data set")
```

`crimtab` *Student's 3000 Criminals Data*

Description

Data of 3000 male criminals over 20 years old undergoing their sentences in the chief prisons of England and Wales.

Usage

```
crimtab
```

Format

A `table` object of `integer` counts, of dimension 42×22 with a total count, `sum(crimtab)` of 3000.

The 42 `rownames` (`"9.4"`, `"9.5"`, ...) correspond to midpoints of intervals of finger lengths whereas the 22 column names (`colnames`) (`"142.24"`, `"144.78"`, ...) correspond to (body) heights of 3000 criminals, see also below.

Details

Student is the pseudonym of William Sealy Gosset. In his 1908 paper he wrote (on page 13) at the beginning of section VI entitled *Practical Test of the forgoing Equations*:

"Before I had succeeded in solving my problem analytically, I had endeavoured to do so empirically. The material used was a correlation table containing the height and left middle finger measurements of 3000 criminals, from a paper by W. R. MacDonell (1902, p. 219). The measurements were written out on 3000 pieces of cardboard, which were then very thoroughly shuffled and drawn at random. As each card was drawn its numbers were written down in a book, which thus contains the measurements of 3000 criminals in a random order. Finally, each consecutive set of 4 was taken as a sample—750 in all—and the mean, standard deviation, and correlation of each sample determined. The difference between the mean of each sample and the mean of the population was then divided by the standard deviation of the sample, giving us the z of Section III."

The table is in fact page 216 and not page 219 in MacDonell (1902). In the MacDonell table, the middle finger lengths were given in mm and the heights in feet/inches intervals, they are both converted into cm here. The midpoints of intervals were used, e.g., where MacDonell has $4'7''9/16 - -8''9/16$, we have 142.24 which is 2.54*56 = 2.54*($4'8''$).

MacDonell credited the source of data (page 178) as follows: *The data on which the memoir is based were obtained, through the kindness of Dr Garson, from the Central Metric Office, New Scotland Yard...* He pointed out on page 179 that : *The forms were drawn at random from the mass on the office shelves; we are therefore dealing with a random sampling.*

Source

https://pbil.univ-lyon1.fr/R/donnees/criminals1902.txt thanks to Jean R. Lobry and Anne-Béatrice Dufour.

References

Garson, J.G. (1900). The metric system of identification of criminals, as used in Great Britain and Ireland. *The Journal of the Anthropological Institute of Great Britain and Ireland*, **30**, 161–198. doi:10.2307/2842627.

MacDonell, W.R. (1902). On criminal anthropometry and the identification of criminals. *Biometrika*, **1**(2), 177–227. doi:10.2307/2331487.

Student (1908). The probable error of a mean. *Biometrika*, **6**, 1–25. doi:10.2307/2331554.

Examples

```
require(stats)
dim(crimtab)
utils::str(crimtab)
## for nicer printing:
local({cT <- crimtab
       colnames(cT) <- substring(colnames(cT), 2, 3)
       print(cT, zero.print = " ")
})

## Repeat Student's experiment:

# 1) Reconstitute 3000 raw data for heights in inches and rounded to
#    nearest integer as in Student's paper:

(heIn <- round(as.numeric(colnames(crimtab)) / 2.54))
d.hei <- data.frame(height = rep(heIn, colSums(crimtab)))

# 2) shuffle the data:

set.seed(1)
d.hei <- d.hei[sample(1:3000), , drop = FALSE]

# 3) Make 750 samples each of size 4:

d.hei$sample <- as.factor(rep(1:750, each = 4))

# 4) Compute the means and standard deviations (n) for the 750 samples:

h.mean <- with(d.hei, tapply(height, sample, FUN = mean))
h.sd   <- with(d.hei, tapply(height, sample, FUN = sd)) * sqrt(3/4)

# 5) Compute the difference between the mean of each sample and
#    the mean of the population and then divide by the
#    standard deviation of the sample:

zobs <- (h.mean - mean(d.hei[,"height"]))/h.sd
```

```
# 6) Replace infinite values by +/- 6 as in Student's paper:

zobs[infZ <- is.infinite(zobs)] # none of them
zobs[infZ] <- 6 * sign(zobs[infZ])

# 7) Plot the distribution:

require(grDevices); require(graphics)
hist(x = zobs, probability = TRUE, xlab = "Student's z",
     col = grey(0.8), border = grey(0.5),
     main = "Distribution of Student's z score  for 'crimtab' data")
```

`discoveries` *Yearly Numbers of Important Discoveries*

Description

The numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959.

Usage

```
discoveries
```

Format

A time series of 100 values.

Source

The World Almanac and Book of Facts, 1975 Edition, pages 315–318.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
plot(discoveries, ylab = "Number of important discoveries",
     las = 1)
title(main = "discoveries data set")
```

DNase *Elisa assay of DNase*

Description

The DNase data frame has 176 rows and 3 columns of data obtained during development of an ELISA assay for the recombinant protein DNase in rat serum.

Usage

```
DNase
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

Run an ordered factor with levels `10` < ... < `3` indicating the assay run.

conc a numeric vector giving the known concentration of the protein.

density a numeric vector giving the measured optical density (dimensionless) in the assay. Duplicate optical density measurements were obtained.

Details

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)

coplot(density ~ conc | Run, data = DNase,
       show.given = FALSE, type = "b")
coplot(density ~ log(conc) | Run, data = DNase,
       show.given = FALSE, type = "b")
## fit a representative run
fm1 <- nls(density ~ SSlogis( log(conc), Asym, xmid, scal ),
    data = DNase, subset = Run == 1)
## compare with a four-parameter logistic
fm2 <- nls(density ~ SSfpl( log(conc), A, B, xmid, scal ),
    data = DNase, subset = Run == 1)
summary(fm2)
anova(fm1, fm2)
```

esoph *Smoking, Alcohol and (O)esophageal Cancer*

Description

Data from a case-control study of (o)esophageal cancer in Ille-et-Vilaine, France.

Usage

```
esoph
```

Format

A data frame with records for 88 age/alcohol/tobacco combinations.

[,1]	agegp	Age group	1 25–34 years
			2 35–44
			3 45–54
			4 55–64
			5 65–74
			6 75+
[,2]	alcgp	Alcohol consumption	1 0–39 gm/day
			2 40–79
			3 80–119
			4 120+
[,3]	tobgp	Tobacco consumption	1 0– 9 gm/day
			2 10–19
			3 20–29
			4 30+
[,4]	ncases	Number of cases	
[,5]	ncontrols	Number of controls	

Author(s)

Thomas Lumley

Source

Breslow, N. E. and Day, N. E. (1980) *Statistical Methods in Cancer Research. Volume 1: The Analysis of Case-Control Studies.* IARC Lyon / Oxford University Press.

Examples

```
require(stats)
require(graphics) # for mosaicplot
summary(esoph)
## effects of alcohol, tobacco and interaction, age-adjusted
model1 <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp,
```

```
              data = esoph, family = binomial())
anova(model1)
## Try a linear effect of alcohol and tobacco
model2 <- glm(cbind(ncases, ncontrols) ~ agegp + unclass(tobgp)
                                         + unclass(alcgp),
              data = esoph, family = binomial())
summary(model2)
## Re-arrange data for a mosaic plot
ttt <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
o <- with(esoph, order(tobgp, alcgp, agegp))
ttt[ttt == 1] <- esoph$ncases[o]
tt1 <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
tt1[tt1 == 1] <- esoph$ncontrols[o]
tt <- array(c(ttt, tt1), c(dim(ttt),2),
            c(dimnames(ttt), list(c("Cancer", "control"))))
mosaicplot(tt, main = "esoph data set", color = TRUE)
```

`euro` *Conversion Rates of Euro Currencies*

Description

Conversion rates between the various Euro currencies.

Usage

```
euro
euro.cross
```

Format

`euro` is a named vector of length 11, `euro.cross` a matrix of size 11 by 11, with dimnames.

Details

The data set `euro` contains the value of 1 Euro in all currencies participating in the European monetary union (Austrian Schilling ATS, Belgian Franc BEF, German Mark DEM, Spanish Peseta ESP, Finnish Markka FIM, French Franc FRF, Irish Punt IEP, Italian Lira ITL, Luxembourg Franc LUF, Dutch Guilder NLG and Portuguese Escudo PTE). These conversion rates were fixed by the European Union on December 31, 1998. To convert old prices to Euro prices, divide by the respective rate and round to 2 digits.

The data set `euro.cross` contains conversion rates between the various Euro currencies, i.e., the result of `outer(1 / euro, euro)`.

Examples

```
cbind(euro)

## These relations hold:
euro == signif(euro, 6) # [6 digit precision in Euro's definition]
all(euro.cross == outer(1/euro, euro))

## Convert 20 Euro to Belgian Franc
20 * euro["BEF"]
## Convert 20 Austrian Schilling to Euro
20 / euro["ATS"]
## Convert 20 Spanish Pesetas to Italian Lira
20 * euro.cross["ESP", "ITL"]

require(graphics)
dotchart(euro,
         main = "euro data: 1 Euro in currency unit")
dotchart(1/euro,
         main = "euro data: 1 currency unit in Euros")
dotchart(log(euro, 10),
         main = "euro data: log10(1 Euro in currency unit)")
```

---

eurodist *Distances Between European Cities and Between US Cities*

Description

The `eurodist` gives the road distances (in km) between 21 cities in Europe. The data are taken from a table in *The Cambridge Encyclopaedia*.

`UScitiesD` gives "straight line" distances between 10 cities in the US.

Usage

```
eurodist
UScitiesD
```

Format

`dist` objects based on 21 and 10 objects, respectively. (You must have the **stats** package loaded to have the methods for this kind of object available).

Source

Crystal, D. Ed. (1990) *The Cambridge Encyclopaedia*. Cambridge: Cambridge University Press,

The US cities distances were provided by Pierre Legendre.

EuStockMarkets *Daily Closing Prices of Major European Stock Indices, 1991–1998*

Description

Contains the daily closing prices of major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. The data are sampled in business time, i.e., weekends and holidays are omitted.

Usage

```
EuStockMarkets
```

Format

A multivariate time series with 1860 observations on 4 variables. The object is of class `"mts"`.

Source

The data were kindly provided by Erste Bank AG, Vienna, Austria.

faithful *Old Faithful Geyser Data*

Description

Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

Usage

```
faithful
```

Format

A data frame with 272 observations on 2 variables.

[,1]	`eruptions`	numeric	Eruption time in mins
[,2]	`waiting`	numeric	Waiting time to next eruption (in mins)

Details

A closer look at `faithful$eruptions` reveals that these are heavily rounded times originally in seconds, where multiples of 5 are more frequent than expected under non-human measurement. For a better version of the eruption times, see the example below.

There are many versions of this dataset around: Azzalini and Bowman (1990) use a more complete version.

Source

W. Härdle.

References

Härdle, W. (1991). *Smoothing Techniques with Implementation in S*. New York: Springer.

Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics*, **39**, 357–365. doi:10.2307/2347385.

See Also

`geyser` in package **MASS** for the Azzalini–Bowman version.

Examples

```
require(stats); require(graphics)
f.tit <-  "faithful data: Eruptions of Old Faithful"

ne60 <- round(e60 <- 60 * faithful$eruptions)
all.equal(e60, ne60)             # relative diff. ~ 1/10000
table(zapsmall(abs(e60 - ne60))) # 0, 0.02 or 0.04
faithful$better.eruptions <- ne60 / 60
te <- table(ne60)
te[te >= 4]                      # (too) many multiples of 5 !
plot(names(te), te, type = "h", main = f.tit, xlab = "Eruption time (sec)")

plot(faithful[, -3], main = f.tit,
     xlab = "Eruption time (min)",
     ylab = "Waiting time to next eruption (min)")
lines(lowess(faithful$eruptions, faithful$waiting, f = 2/3, iter = 3),
      col = "red")
```

`Formaldehyde` *Determination of Formaldehyde*

Description

These data are from a chemical experiment to prepare a standard curve for the determination of formaldehyde by the addition of chromotropic acid and concentrated sulphuric acid and the reading of the resulting purple color on a spectrophotometer.

Usage

```
Formaldehyde
```

Format

A data frame with 6 observations on 2 variables.

[,1]	carb	numeric	Carbohydrate (ml)
[,2]	optden	numeric	Optical Density

Source

Bennett, N. A. and N. L. Franklin (1954) *Statistical Analysis in Chemistry and the Chemical Industry*. New York: Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis.* New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(optden ~ carb, data = Formaldehyde,
     xlab = "Carbohydrate (ml)", ylab = "Optical Density",
     main = "Formaldehyde data", col = 4, las = 1)
abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
summary(fm1)
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0))
plot(fm1)
par(opar)
```

---

freeny *Freeny's Revenue Data*

Description

Freeny's data on quarterly revenue and explanatory variables.

Usage

```
freeny
freeny.x
freeny.y
```

Format

There are three 'freeny' data sets.

`freeny.y` is a time series with 39 observations on quarterly revenue from (1962,2Q) to (1971,4Q).

`freeny.x` is a matrix of explanatory variables. The columns are `freeny.y` lagged 1 quarter, price index, income level, and market potential.

Finally, `freeny` is a data frame with variables `y`, `lag.quarterly.revenue`, `price.index`, `income.level`, and `market.potential` obtained from the above two data objects.

Source

A. E. Freeny (1977) *A Portable Linear Regression Package with Test Programs*. Bell Laboratories memorandum.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
summary(freeny)
pairs(freeny, main = "freeny data")
# gives warning: freeny$y has class "ts"

summary(fm1 <- lm(y ~ ., data = freeny))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

HairEyeColor *Hair and Eye Color of Statistics Students*

Description

Distribution of hair and eye color and sex in 592 statistics students.

Usage

```
HairEyeColor
```

Format

A 3-dimensional array resulting from cross-tabulating 592 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Hair	Black, Brown, Red, Blond
2	Eye	Brown, Blue, Hazel, Green
3	Sex	Male, Female

Details

The Hair $\times$ Eye table comes from a survey of students at the University of Delaware reported by Snee (1974). The split by Sex was added by Friendly (1992a) for didactic purposes.

This data set is useful for illustrating various techniques for the analysis of contingency tables, such as the standard chi-squared test or, more generally, log-linear modelling, and graphical methods such as mosaic plots, sieve diagrams or association plots.

Source

http://www.datavis.ca/sas/vcd/catdata/haireye.sas

Snee (1974) gives the two-way table aggregated over Sex. The Sex split of the 'Brown hair, Brown eye' cell was changed to agree with that used by Friendly (2000).

References

Snee, R. D. (1974). Graphical display of two-way contingency tables. *The American Statistician*, **28**, 9–12. doi:10.2307/2683520.

Friendly, M. (1992a). Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://datavis.ca/papers/sugi/sugi17.pdf

Friendly, M. (1992b). Mosaic displays for loglinear models. *Proceedings of the Statistical Graphics Section*, American Statistical Association, pp. 61–68. http://www.datavis.ca/papers/asa92.html

Friendly, M. (2000). *Visualizing Categorical Data*. SAS Institute, ISBN 1-58025-660-0.

See Also

chisq.test, loglin, mosaicplot

Examples

```
require(graphics)
## Full mosaic
mosaicplot(HairEyeColor)
## Aggregate over sex (as in Snee's original data)
x <- apply(HairEyeColor, c(1, 2), sum)
x
mosaicplot(x, main = "Relation between hair and eye color")
```

Harman23.cor *Harman Example 2.3*

Description

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

Usage

```
Harman23.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 2.3.

Examples

```
require(stats)
(Harman23.FA <- factanal(factors = 1, covmat = Harman23.cor))
for(factors in 2:4) print(update(Harman23.FA, factors = factors))
```

`Harman74.cor` *Harman Example 7.4*

Description

A correlation matrix of 24 psychological tests given to 145 seventh and eight-grade children in a Chicago suburb by Holzinger and Swineford.

Usage

```
Harman74.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 7.4.

Examples

```
require(stats)
(Harman74.FA <- factanal(factors = 1, covmat = Harman74.cor))
for(factors in 2:5) print(update(Harman74.FA, factors = factors))
Harman74.FA <- factanal(factors = 5, covmat = Harman74.cor,
                        rotation = "promax")
print(Harman74.FA$loadings, sort = TRUE)
```

`Indometh` *Pharmacokinetics of Indomethacin*

Description

The `Indometh` data frame has 66 rows and 3 columns of data on the pharmacokinetics of indometacin (or, older spelling, 'indomethacin').

Usage

```
Indometh
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

Subject an ordered factor with containing the subject codes. The ordering is according to increasing maximum response.

time a numeric vector of times at which blood samples were drawn (hr).

conc a numeric vector of plasma concentrations of indometacin (mcg/ml).

Details

Each of the six subjects were given an intravenous injection of indometacin.

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Kwan, Breault, Umbenhauer, McMahon and Duggan (1976) Kinetics of Indomethacin absorption, elimination, and enterohepatic circulation in man. *Journal of Pharmacokinetics and Biopharmaceutics* **4**, 255–280.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 129)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

See Also

`SSbiexp` for models fitted to this dataset.

---

infert *Infertility after Spontaneous and Induced Abortion*

Description

This is a matched case-control study dating from before the availability of conditional logistic regression.

Usage

```
infert
```

Format

1.	Education	0 = 0-5 years
		1 = 6-11 years
		2 = 12+ years
2.	age	age in years of case
3.	parity	count
4.	number of prior	0 = 0
	induced abortions	1 = 1
		2 = 2 or more
5.	case status	1 = case
		0 = control
6.	number of prior	0 = 0
	spontaneous abortions	1 = 1
		2 = 2 or more
7.	matched set number	1-83
8.	stratum number	1-63

Note

One case with two prior spontaneous abortions and two prior induced abortions is omitted.

Source

Trichopoulos, D., Handanos, N., Danezis, J., Kalandidi, A., & Kalapothaki, V. (1976). Induced abortion and secondary infertility. *British Journal of Obstetrics and Gynaecology*, **83**, 645–650. doi:10.1111/j.14710528.1976.tb00904.x.

Examples

```
require(stats)
model1 <- glm(case ~ spontaneous+induced, data = infert, family = binomial())
summary(model1)
## adjusted for other potential confounders:
summary(model2 <- glm(case ~ age+parity+education+spontaneous+induced,
                      data = infert, family = binomial()))
## Really should be analysed by conditional logistic regression
## which is in the survival package
if(require(survival)){
  model3 <- clogit(case ~ spontaneous+induced+strata(stratum), data = infert)
  print(summary(model3))
  detach()  # survival (conflicts)
}
```

`InsectSprays` *Effectiveness of Insect Sprays*

Description

The counts of insects in agricultural experimental units treated with different insecticides.

Usage

```
InsectSprays
```

Format

A data frame with 72 observations on 2 variables.

[,1]	`count`	numeric	Insect count
[,2]	`spray`	factor	The type of spray

Source

Beall, G., (1942) The Transformation of data from entomological field experiments, *Biometrika*, **29**, 243–262.

References

McNeil, D. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
boxplot(count ~ spray, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
fm1 <- aov(count ~ spray, data = InsectSprays)
summary(fm1)
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0))
plot(fm1)
fm2 <- aov(sqrt(count) ~ spray, data = InsectSprays)
summary(fm2)
plot(fm2)
par(opar)
```

`iris` *Edgar Anderson's Iris Data*

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

Usage

```
iris
iris3
```

Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as once provided by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

Source

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188. doi:10.1111/j.14691809.1936.tb02137.x.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of the American Iris Society*, **59**, 2–5.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (has `iris3` as `iris`.)

See Also

`matplot` some examples of which use `iris`.

Examples

```
summary(iris)

## Fisher's (1936) research question: whether (compound measurements of)
## Iris versicolor "differs twice as much from I. setosa as from I. virginica"
pairs(iris[1:4], col = iris$Species)
legend(0.5, 1, levels(iris$Species), fill = 1:3, bty = "n",
       horiz = TRUE, xjust = 0.5, yjust = 0, xpd = TRUE)
```

```
## equivalence of legacy array (iris3) and data.frame (iris) representation
dni3 <- dimnames(iris3)
ii <- data.frame(matrix(aperm(iris3, c(1,3,2)), ncol = 4,
                        dimnames = list(NULL, sub(" L.",".Length",
                                                  sub(" W.",".Width", dni3[[2]])))),
    Species = gl(3, 50, labels = sub("S", "s", sub("V", "v", dni3[[3]]))))
stopifnot(all.equal(ii, iris))
```

`islands` *Areas of the World's Major Landmasses*

Description

The areas in thousands of square miles of the landmasses which exceed 10,000 square miles.

Usage

```
islands
```

Format

A named vector of length 48.

Source

The World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
dotchart(log(islands, 10),
   main = "islands data: log10(area) (log10(sq. miles))")
dotchart(log(islands[order(islands)], 10),
   main = "islands data: log10(area) (log10(sq. miles))")
```

JohnsonJohnson *Quarterly Earnings per Johnson & Johnson Share*

Description

Quarterly earnings (dollars) per Johnson & Johnson share 1960–80.

Usage

```
JohnsonJohnson
```

Format

A quarterly time series

Source

Shumway, R. H. and Stoffer, D. S. (2000) *Time Series Analysis and its Applications*. Second Edition. Springer. Example 1.1.

Examples

```
require(stats); require(graphics)
JJ <- log10(JohnsonJohnson)
plot(JJ)
## This example gives a possible-non-convergence warning on some
## platforms, but does seem to converge on x86 Linux and Windows.
(fit <- StructTS(JJ, type = "BSM"))
tsdiag(fit)
sm <- tsSmooth(fit)
plot(cbind(JJ, sm[, 1], sm[, 3]-0.5), plot.type = "single",
     col = c("black", "green", "blue"))
abline(h = -0.5, col = "grey60")

monthplot(fit)
```

LakeHuron *Level of Lake Huron 1875–1972*

Description

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

Usage

```
LakeHuron
```

Format

A time series of length 98.

Source

Brockwell, P. J. and Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. Springer, New York. Series A, page 555.

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.

---

lh *Luteinizing Hormone in Blood Samples*

Description

A regular time series giving the luteinizing hormone in blood samples at 10 mins intervals from a human female, 48 samples.

Usage

```
lh
```

Source

P.J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.1, series 3

LifeCycleSavings *Intercountry Life-Cycle Savings Data*

Description

Data on the savings ratio 1960–1970.

Usage

```
LifeCycleSavings
```

Format

A data frame with 50 observations on 5 variables.

[,1]	`sr`	numeric	aggregate personal savings
[,2]	`pop15`	numeric	% of population under 15
[,3]	`pop75`	numeric	% of population over 75
[,4]	`dpi`	numeric	real per-capita disposable income
[,5]	`ddpi`	numeric	% growth rate of dpi

Details

Under the life-cycle savings hypothesis as developed by Franco Modigliani, the savings ratio (aggregate personal saving divided by disposable income) is explained by per-capita disposable income, the percentage rate of change in per-capita disposable income, and two demographic variables: the percentage of population less than 15 years old and the percentage of the population over 75 years old. The data are averaged over the decade 1960–1970 to remove the business cycle or other short-term fluctuations.

Source

The data were obtained from Belsley, Kuh and Welsch (1980). They in turn obtained the data from Sterling (1977).

References

Sterling, Arnie (1977) Unpublished BS Thesis. Massachusetts Institute of Technology.

Belsley, D. A., Kuh. E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

Examples

```
require(stats); require(graphics)
pairs(LifeCycleSavings, panel = panel.smooth,
      main = "LifeCycleSavings data")
fm1 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(fm1)
```

`Loblolly` *Growth of Loblolly Pine Trees*

Description

The `Loblolly` data frame has 84 rows and 3 columns of records of the growth of Loblolly pine trees.

Usage

```
Loblolly
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

height a numeric vector of tree heights (ft).

age a numeric vector of tree ages (yr).

Seed an ordered factor indicating the seed source for the tree. The ordering is according to increasing maximum height.

Details

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Kung, F. H. (1986), Fitting logistic growth curve with predetermined carrying capacity, in *Proceedings of the Statistical Computing Section, American Statistical Association*, 340–343.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
plot(height ~ age, data = Loblolly, subset = Seed == 329,
     xlab = "Tree age (yr)", las = 1,
     ylab = "Tree height (ft)",
     main = "Loblolly data and fitted curve (Seed 329 only)")
fm1 <- nls(height ~ SSasymp(age, Asym, R0, lrc),
           data = Loblolly, subset = Seed == 329)
age <- seq(0, 30, length.out = 101)
lines(age, predict(fm1, list(age = age)))
```

---

longley *Longley's Economic Regression Data*

Description

A macroeconomic data set which provides a well-known example for a highly collinear regression.

Usage

```
longley
```

Format

A data frame with 7 economical variables, observed yearly from 1947 to 1962 ($n = 16$).

`GNP.deflator` GNP implicit price deflator ($1954 = 100$)

`GNP` Gross National Product.

`Unemployed` number of unemployed.

`Armed.Forces` number of people in the armed forces.

`Population` 'noninstitutionalized' population ≥ 14 years of age.

`Year` the year (time).

`Employed` number of people employed.

The regression `lm(Employed ~ .)` is known to be highly collinear.

Source

J. W. Longley (1967) An appraisal of least-squares programs from the point of view of the user. *Journal of the American Statistical Association* **62**, 819–841.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
## give the data set in the form it was used in S-PLUS:
longley.x <- data.matrix(longley[, 1:6])
longley.y <- longley[, "Employed"]
pairs(longley, main = "longley data")
summary(fm1 <- lm(Employed ~ ., data = longley))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

`lynx` *Annual Canadian Lynx trappings 1821–1934*

Description

Annual numbers of lynx trappings for 1821–1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).

Usage

```
lynx
```

Source

Brockwell, P. J. and Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. Springer. Series G (page 557).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Campbell, M. J. and Walker, A. M. (1977). A Survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis. *Journal of the Royal Statistical Society Series A*, **140**, 411–431. doi:10.2307/2345277.

`morley` *Michelson Speed of Light Data*

Description

A classical data of Michelson (but not this one with Morley) on measurements done in 1879 on the speed of light. The data consists of five experiments, each consisting of 20 consecutive 'runs'. The response is the speed of light measurement, suitably coded (km/sec, with `299000` subtracted).

Usage

```
morley
```

Format

A data frame with 100 observations on the following 3 variables.

`Expt` The experiment number, from 1 to 5.

`Run` The run number within each experiment.

`Speed` Speed-of-light measurement.

Details

The data is here viewed as a randomized block experiment with 'experiment' and 'run' as the factors. 'run' may also be considered a quantitative variate to account for linear (or polynomial) changes in the measurement over the course of a single experiment.

Note

This is the same dataset as `michelson` in package **MASS**.

Source

A. J. Weekes (1986) *A Genstat Primer*. London: Edward Arnold.

S. M. Stigler (1977) Do robust estimators work with real data? *Annals of Statistics* **5**, 1055–1098. (See Table 6.)

A. A. Michelson (1882) Experimental determination of the velocity of light made at the United States Naval Academy, Annapolis. *Astronomic Papers* **1** 135–8. U.S. Nautical Almanac Office. (See Table 24.)

Examples

```
require(stats); require(graphics)
michelson <- transform(morley,
                       Expt = factor(Expt), Run = factor(Run))
xtabs(~ Expt + Run, data = michelson)  # 5 x 20 balanced (two-way)
plot(Speed ~ Expt, data = michelson,
     main = "Speed of Light Data", xlab = "Experiment No.")
fm <- aov(Speed ~ Run + Expt, data = michelson)
summary(fm)
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
```

---

mtcars *Motor Trend Car Road Tests*

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 (numeric) variables.

[, 1]	`mpg`	Miles/(US) gallon
[, 2]	`cyl`	Number of cylinders
[, 3]	`disp`	Displacement (cu.in.)
[, 4]	`hp`	Gross horsepower
[, 5]	`drat`	Rear axle ratio
[, 6]	`wt`	Weight (1000 lbs)
[, 7]	`qsec`	1/4 mile time
[, 8]	`vs`	Engine (0 = V-shaped, 1 = straight)
[, 9]	`am`	Transmission (0 = automatic, 1 = manual)
[,10]	`gear`	Number of forward gears

[,11] carb Number of carburetors

Note

Henderson and Velleman (1981) comment in a footnote to Table 1: 'Hocking [original transcriber]'s noncrucial coding of the Mazda's rotary engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the inclusion of the diesel Mercedes 240D, have been retained to enable direct comparisons to be made with previous analyses.'

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Examples

```
require(graphics)
pairs(mtcars, main = "mtcars data", gap = 1/4)
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
## possibly more meaningful, e.g., for summary() or bivariate plots:
mtcars2 <- within(mtcars, {
   vs <- factor(vs, labels = c("V", "S"))
   am <- factor(am, labels = c("automatic", "manual"))
   cyl  <- ordered(cyl)
   gear <- ordered(gear)
   carb <- ordered(carb)
})
summary(mtcars2)
```

---

nhtemp *Average Yearly Temperatures in New Haven*

Description

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

Usage

```
nhtemp
```

Format

A time series of 60 observations.

Source

Vaux, J. E. and Brinker, N. B. (1972) *Cycles*, **1972**, 117–121.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(nhtemp, main = "nhtemp data",
  ylab = "Mean annual temperature in New Haven, CT (deg. F)")
```

`Nile` *Flow of the River Nile*

Description

Measurements of the annual flow of the river Nile at Aswan (formerly `Assuan`), 1871–1970, in $10^8 m^3$, "with apparent changepoint near 1898" (Cobb(1978), Table 1, p.249).

Usage

```
Nile
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

References

Balke, N. S. (1993). Detecting level shifts in time series. *Journal of Business and Economic Statistics*, **11**, 81–92. doi:10.2307/1391308.

Cobb, G. W. (1978). The problem of the Nile: conditional solution to a change-point problem. *Biometrika* **65**, 243–51. doi:10.2307/2335202.

Examples

```
require(stats); require(graphics)
par(mfrow = c(2, 2))
plot(Nile)
acf(Nile)
pacf(Nile)
ar(Nile) # selects order 2
cpgram(ar(Nile)$resid)
par(mfrow = c(1, 1))
arima(Nile, c(2, 0, 0))
```

```
## Now consider missing values, following Durbin & Koopman
NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
arima(NileNA, c(2, 0, 0))
plot(NileNA)
pred <-
   predict(arima(window(NileNA, 1871, 1890), c(2, 0, 0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty = 2, col = "blue")
lines(pred$pred - 2*pred$se, lty = 2, col = "blue")
pred <-
   predict(arima(window(NileNA, 1871, 1930), c(2, 0, 0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty = 2, col = "blue")
lines(pred$pred - 2*pred$se, lty = 2, col = "blue")

## Structural time series models
par(mfrow = c(3, 1))
plot(Nile)
## local level model
(fit <- StructTS(Nile, type = "level"))
lines(fitted(fit), lty = 2)              # contemporaneous smoothing
lines(tsSmooth(fit), lty = 2, col = 4)   # fixed-interval smoothing
plot(residuals(fit)); abline(h = 0, lty = 3)
## local trend model
(fit2 <- StructTS(Nile, type = "trend")) ## constant trend fitted
pred <- predict(fit, n.ahead = 30)
## with 50% confidence interval
ts.plot(Nile, pred$pred,
        pred$pred + 0.67*pred$se, pred$pred -0.67*pred$se)

## Now consider missing values
plot(NileNA)
(fit3 <- StructTS(NileNA, type = "level"))
lines(fitted(fit3), lty = 2)
lines(tsSmooth(fit3), lty = 3)
plot(residuals(fit3)); abline(h = 0, lty = 3)
```

`nottem` *Average Monthly Temperatures at Nottingham, 1920–1939*

Description

A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

Usage

```
nottem
```

Source

Anderson, O. D. (1976) *Time Series Analysis and Forecasting: The Box-Jenkins approach.* Butterworths. Series R.

Examples

```
require(stats); require(graphics)
nott <- window(nottem, end = c(1936,12))
fit <- arima(nott, order = c(1,0,0), list(order = c(2,1,0), period = 12))
nott.fore <- predict(fit, n.ahead = 36)
ts.plot(nott, nott.fore$pred, nott.fore$pred+2*nott.fore$se,
        nott.fore$pred-2*nott.fore$se, gpars = list(col = c(1,1,4,4)))
```

---

npk *Classical N, P, K Factorial Experiment*

Description

A classical N, P, K (nitrogen, phosphate, potassium) factorial experiment on the growth of peas conducted on 6 blocks. Each half of a fractional factorial design confounding the NPK interaction was used on 3 of the plots.

Usage

```
npk
```

Format

The npk data frame has 24 rows and 5 columns:

`block` which block (label 1 to 6).

`N` indicator (0/1) for the application of nitrogen.

`P` indicator (0/1) for the application of phosphate.

`K` indicator (0/1) for the application of potassium.

`yield` Yield of peas, in pounds/plot (the plots were (1/70) acre).

Source

Imperial College, London, M.Sc. exercise sheet.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer.

Examples

```
options(contrasts = c("contr.sum", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
npk.aov
summary(npk.aov)
coef(npk.aov)
options(contrasts = c("contr.treatment", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, data = npk)
summary.lm(npk.aov1)
se.contrast(npk.aov1, list(N=="0", N=="1"), data = npk)
model.tables(npk.aov1, type = "means", se = TRUE)
```

---

occupationalStatus *Occupational Status of Fathers and their Sons*

Description

Cross-classification of a sample of British males according to each subject's occupational status and his father's occupational status.

Usage

```
occupationalStatus
```

Format

A table of counts, with classifying factors `origin` (father's occupational status; levels `1:8`) and `destination` (son's occupational status; levels `1:8`).

Source

Goodman, L. A. (1979) Simple Models for the Analysis of Association in Cross-Classifications having Ordered Categories. *J. Am. Stat. Assoc.*, **74** (367), 537–552.

The data set has been in package **gnm** and been provided by the package authors.

Examples

```
require(stats); require(graphics)

plot(occupationalStatus)

##  Fit a uniform association model separating diagonal effects
Diag <- as.factor(diag(1:8))
Rscore <- scale(as.numeric(row(occupationalStatus)), scale = FALSE)
Cscore <- scale(as.numeric(col(occupationalStatus)), scale = FALSE)
modUnif <- glm(Freq ~ origin + destination + Diag + Rscore:Cscore,
               family = poisson, data = occupationalStatus)

summary(modUnif)
plot(modUnif) # 4 plots, with warning about  h_ii ~= 1
```

`Orange` *Growth of Orange Trees*

Description

The `Orange` data frame has 35 rows and 3 columns of records of the growth of orange trees.

Usage

```
Orange
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

Tree an ordered factor indicating the tree on which the measurement is made. The ordering is according to increasing maximum diameter.

age a numeric vector giving the age of the tree (days since 1968/12/31)

circumference a numeric vector of trunk circumferences (mm). This is probably "circumference at breast height", a standard measurement in forestry.

Details

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Draper, N. R. and Smith, H. (1998), *Applied Regression Analysis (3rd ed)*, Wiley (exercise 24.N).

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
coplot(circumference ~ age | Tree, data = Orange, show.given = FALSE)
fm1 <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
           data = Orange, subset = Tree == 3)
plot(circumference ~ age, data = Orange, subset = Tree == 3,
     xlab = "Tree age (days since 1968/12/31)",
     ylab = "Tree circumference (mm)", las = 1,
     main = "Orange tree data and fitted model (Tree 3 only)")
age <- seq(0, 1600, length.out = 101)
lines(age, predict(fm1, list(age = age)))
```

`OrchardSprays` *Potency of Orchard Sprays*

Description

An experiment was conducted to assess the potency of various constituents of orchard sprays in repelling honeybees, using a Latin square design.

Usage

```
OrchardSprays
```

Format

A data frame with 64 observations on 4 variables.

[,1]	`rowpos`	numeric	Row of the design
[,2]	`colpos`	numeric	Column of the design
[,3]	`treatment`	factor	Treatment level
[,4]	`decrease`	numeric	Response

Details

Individual cells of dry comb were filled with measured amounts of lime sulphur emulsion in sucrose solution. Seven different concentrations of lime sulphur ranging from a concentration of 1/100 to 1/1,562,500 in successive factors of 1/5 were used as well as a solution containing no lime sulphur.

The responses for the different solutions were obtained by releasing 100 bees into the chamber for two hours, and then measuring the decrease in volume of the solutions in the various cells.

An 8×8 Latin square design was used and the treatments were coded as follows:

A	highest level of lime sulphur
B	next highest level of lime sulphur
.	
.	
.	
G	lowest level of lime sulphur
H	no lime sulphur

Source

Finney, D. J. (1947) *Probit Analysis*. Cambridge.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
pairs(OrchardSprays, main = "OrchardSprays data")
```

PlantGrowth *Results from an Experiment on Plant Growth*

Description

Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

Usage

```
PlantGrowth
```

Format

A data frame of 30 cases on 2 variables.

[, 1]	weight	numeric
[, 2]	group	factor

The levels of group are 'ctrl', 'trt1', and 'trt2'.

Source

Dobson, A. J. (1983) *An Introduction to Statistical Modelling*. London: Chapman and Hall.

Examples

```
## One factor ANOVA example from Dobson's book, cf. Table 7.4:
require(stats); require(graphics)
boxplot(weight ~ group, data = PlantGrowth, main = "PlantGrowth data",
        ylab = "Dried weight of plants", col = "lightgray",
        notch = TRUE, varwidth = TRUE)
anova(lm(weight ~ group, data = PlantGrowth))
```

precip *Annual Precipitation in US Cities*

Description

The average amount of precipitation (rainfall) in inches for each of 70 United States (and Puerto Rico) cities.

Usage

```
precip
```

Format

A named vector of length 70.

Note

The dataset version up to Nov.16, 2016 had a typo in `"Cincinnati"`'s name. The examples show how to recreate that version.

Source

Statistical Abstracts of the United States, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
dotchart(precip[order(precip)], main = "precip data")
title(sub = "Average annual precipitation (in.)")

## Old ("wrong") version of dataset (just name change):
precip.0 <- local({
   p <- precip; names(p)[names(p) == "Cincinnati"] <- "Cincinati" ; p })
stopifnot(all(precip == precip.0),
   match("Cincinnati", names(precip)) == 46,
   identical(names(precip)[-46], names(precip.0)[-46]))
```

presidents *Quarterly Approval Ratings of US Presidents*

Description

The (approximately) quarterly approval rating for the President of the United States from the first quarter of 1945 to the last quarter of 1974.

Usage

```
presidents
```

Format

A time series of 120 values.

Details

The data are actually a fudged version of the approval ratings. See McNeil's book for details.

Source

The Gallup Organisation.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(presidents, las = 1, ylab = "Approval rating (%)",
     main = "presidents data")
```

pressure *Vapor Pressure of Mercury as a Function of Temperature*

Description

Data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury).

Usage

```
pressure
```

Format

A data frame with 19 observations on 2 variables.

[, 1]	`temperature`	numeric	temperature (deg C)
[, 2]	`pressure`	numeric	pressure (mm)

Source

Weast, R. C., ed. (1973) *Handbook of Chemistry and Physics*. CRC Press.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
plot(pressure, xlab = "Temperature (deg C)",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
plot(pressure, xlab = "Temperature (deg C)",  log = "y",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
```

`Puromycin` *Reaction Velocity of an Enzymatic Reaction*

Description

The `Puromycin` data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

Usage

```
Puromycin
```

Format

This data frame contains the following columns:

`conc` a numeric vector of substrate concentrations (ppm)

`rate` a numeric vector of instantaneous reaction rates (counts/min/min)

`state` a factor with levels `treated` `untreated`

Details

Data on the velocity of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate (or velocity) of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.3.

Treloar, M. A. (1974), *Effects of Puromycin on Galactosyltransferase in Golgi Membranes*, M.Sc. Thesis, U. of Toronto.

See Also

SSmicmen for other models fitted to this dataset.

Examples

```
require(stats); require(graphics)

plot(rate ~ conc, data = Puromycin, las = 1,
     xlab = "Substrate concentration (ppm)",
     ylab = "Reaction velocity (counts/min/min)",
     pch = as.integer(Puromycin$state),
     col = as.integer(Puromycin$state),
     main = "Puromycin data and fitted Michaelis-Menten curves")
## simplest form of fitting the Michaelis-Menten model to these data
fm1 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "treated",
           start = c(Vm = 200, K = 0.05))
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05))
summary(fm1)
summary(fm2)
## add fitted lines to the plot
conc <- seq(0, 1.2, length.out = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)
legend(0.8, 120, levels(Puromycin$state),
       col = 1:2, lty = 1:2, pch = 1:2)

## using partial linearity
fm3 <- nls(rate ~ conc/(K + conc), data = Puromycin,
           subset = state == "treated", start = c(K = 0.05),
           algorithm = "plinear")
```

quakes *Locations of Earthquakes off Fiji*

Description

The data set give the locations of 1000 seismic events of MB > 4.0. The events occurred in a cube near Fiji since 1964.

Usage

```
quakes
```

Format

A data frame with 1000 observations on 5 variables.

[,1]	`lat`	numeric	Latitude of event
[,2]	`long`	numeric	Longitude
[,3]	`depth`	numeric	Depth (km)
[,4]	`mag`	numeric	Richter Magnitude
[,5]	`stations`	numeric	Number of stations reporting

Details

There are two clear planes of seismic activity. One is a major plate junction; the other is the Tonga trench off New Zealand. These data constitute a subsample from a larger dataset of containing 5000 observations.

Source

This is one of the Harvard PRIM-H project data sets. They in turn obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

Examples

```
require(graphics)
pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main = 1.2, pch = ".")
```

randu *Random Numbers from Congruential Generator RANDU*

Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

Usage

```
randu
```

Format

A data frame with 400 observations on 3 variables named `x`, `y` and `z` which give the first, second and third random number in the triple.

Details

In three dimensional displays it is evident that the triples fall on 15 parallel planes in 3-space. This can be shown theoretically to be true for all triples from the RANDU generator.

These particular 400 triples start 5 apart in the sequence, that is they are ((U[5i+1], U[5i+2], U[5i+3]), i= 0, . . . , 399), and they are rounded to 6 decimal places.

Under VMS versions 2.0 and higher, this problem has been fixed.

Source

David Donoho

Examples

```
## We could re-generate the dataset by the following R code
seed <- as.double(1)
RANDU <- function() {
    seed <<- ((2^16 + 3) * seed) %% (2^31)
    seed/(2^31)
}
myrandu <- matrix(NA_real_, 400, 3, dimnames = list(NULL, c("x","y","z")))
for(i in 1:400) {
    U <- c(RANDU(), RANDU(), RANDU(), RANDU(), RANDU())
    myrandu[i,] <- round(U[1:3], 6)
}
stopifnot(all.equal(randu, as.data.frame(myrandu), tolerance = 1e-5))
```

rivers *Lengths of Major North American Rivers*

Description

This data set gives the lengths (in miles) of 141 "major" rivers in North America, as compiled by the US Geological Survey.

Usage

```
rivers
```

Format

A vector containing 141 observations.

Source

World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

rock *Measurements on Petroleum Rock Samples*

Description

Measurements on 48 rock samples from a petroleum reservoir.

Usage

```
rock
```

Format

A data frame with 48 rows and 4 numeric columns.

[,1]	`area`	area of pores space, in pixels out of 256 by 256
[,2]	`peri`	perimeter in pixels
[,3]	`shape`	perimeter/sqrt(area)
[,4]	`perm`	permeability in millidarcies

Details

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

Source

Data from BP Research, image analysis by Ronit Katz, U. Oxford.

`sleep` *Student's Sleep Data*

Description

Data which show the effect of two soporific drugs (increase in hours of sleep compared to control) on 10 patients.

Usage

```
sleep
```

Format

A data frame with 20 observations on 3 variables.

[, 1]	extra	numeric	increase in hours of sleep
[, 2]	group	factor	drug given
[, 3]	ID	factor	patient ID

Details

The `group` variable name may be misleading about the data: They represent measurements on 10 persons, not in groups.

Source

Cushny, A. R. and Peebles, A. R. (1905) The action of optical isomers: II hyoscines. *The Journal of Physiology* **32**, 501–510.

Student (1908) The probable error of the mean. *Biometrika*, **6**, 20.

References

Scheffé, Henry (1959) *The Analysis of Variance*. New York, NY: Wiley.

Examples

```
require(stats)
## Student's paired t-test
with(sleep,
     t.test(extra[group == 1],
            extra[group == 2], paired = TRUE))

## The sleep *prolongations*
sleep1 <- with(sleep, extra[group == 2] - extra[group == 1])
summary(sleep1)
stripchart(sleep1, method = "stack", xlab = "hours",
           main = "Sleep prolongation (n = 10)")
boxplot(sleep1, horizontal = TRUE, add = TRUE,
        at = .6, pars = list(boxwex = 0.5, staplewex = 0.25))
```

---

stackloss *Brownlee's Stack Loss Plant Data*

Description

Operational data of a plant for the oxidation of ammonia to nitric acid.

Usage

```
stackloss

stack.x
stack.loss
```

Format

`stackloss` is a data frame with 21 observations on 4 variables.

[,1]	`Air Flow`	Flow of cooling air
[,2]	`Water Temp`	Cooling Water Inlet Temperature
[,3]	`Acid Conc.`	Concentration of acid [per 1000, minus 500]
[,4]	`stack.loss`	Stack loss

For historical compatibility with S-PLUS, the data sets `stack.x`, a matrix with the first three (independent) variables of the data frame, and `stack.loss`, the numeric vector giving the fourth (dependent) variable, are also provided.

Details

"Obtained from 21 days of operation of a plant for the oxidation of ammonia (NH_3) to nitric acid (HNO_3). The nitric oxides produced are absorbed in a countercurrent absorption tower". (Brownlee, cited by Dodge, slightly reformatted by MM.)

`Air Flow` represents the rate of operation of the plant. `Water Temp` is the temperature of cooling water circulated through coils in the absorption tower. `Acid Conc.` is the concentration of the acid circulating, minus 50, times 10: that is, 89 corresponds to 58.9 per cent acid. `stack.loss` (the dependent variable) is 10 times the percentage of the ingoing ammonia to the plant that escapes from the absorption column unabsorbed; that is, an (inverse) measure of the over-all efficiency of the plant.

Source

Brownlee, K. A. (1960, 2nd ed. 1965) *Statistical Theory and Methodology in Science and Engineering*. New York: Wiley. pp. 491–500.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dodge, Y. (1996) The guinea pig of multiple regression. In: *Robust Statistics, Data Analysis, and Computer Intensive Methods; In Honor of Peter Huber's 60th Birthday*, 1996, *Lecture Notes in Statistics* **109**, Springer-Verlag, New York.

Examples

```
require(stats)
summary(lm.stack <- lm(stack.loss ~ stack.x))
```

---

state *US State Facts and Figures*

Description

Data sets related to the 50 states of the United States of America.

Usage

```
state.abb
state.area
state.center
state.division
state.name
state.region
state.x77
```

Details

R currently contains the following "state" data sets. Note that all data are arranged according to alphabetical order of the state names.

`state.abb`: character vector of 2-letter abbreviations for the state names.

`state.area`: numeric vector of state areas (in square miles).

`state.center`: list with components named `x` and `y` giving the approximate geographic center of each state in negative longitude and latitude. Alaska and Hawaii are placed just off the West Coast. See 'Examples' on how to "correct".

`state.division`: `factor` giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain, and Pacific).

`state.name`: character vector giving the full state names.

`state.region`: `factor` giving the region (Northeast, South, North Central, West) that each state belongs to.

`state.x77`: matrix with 50 rows and 8 columns giving the following statistics in the respective columns.

- `Population`: population estimate as of July 1, 1975
- `Income`: per capita income (1974)
- `Illiteracy`: illiteracy (1970, percent of population)
- `Life Exp`: life expectancy in years (1969–71)
- `Murder`: murder and non-negligent manslaughter rate per 100,000 population (1976)
- `HS Grad`: percent high-school graduates (1970)
- `Frost`: mean number of days with minimum temperature below freezing (1931–1960) in capital or large city
- `Area`: land area in square miles

Note that a square mile is by definition exactly `(cm(1760 * 3 * 12) / 100 / 1000)^2` km^2, i.e., $2.589988110336 km^2$.

Source

U.S. Department of Commerce, Bureau of the Census (1977) *Statistical Abstract of the United States*.

U.S. Department of Commerce, Bureau of the Census (1977) *County and City Data Book*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
(dst <- dxy <- data.frame(state.center, row.names=state.abb))
## Alaska and Hawaii are placed just off the West Coast (for compact map drawing):
dst[c("AK", "HI"),]
## state.center2 := version of state.center with "correct" coordinates for AK & HI:
## From https://pubs.usgs.gov/gip/Elevations-Distances/elvadist.html#Geographic%20Centers
##   Alaska   63°50' N., 152°00' W., 60 miles northwest of Mount McKinley
##   Hawaii   20°15' N., 156°20' W., off Maui Island
dxy["AK",] <- c(-152.  , 63.83) # or  c(-152.11, 65.17)
dxy["HI",] <- c(-156.33, 20.25) # or  c(-156.69, 20.89)
state.center2 <- as.list(dxy)

plot(dxy, asp=1.2, pch=3, col=2)
text(state.center2, state.abb, cex=1/2, pos=4, offset=1/4)
i <- c("AK","HI")
do.call(arrows, c(setNames(c(dst[i,], dxy[i,]), c("x0","y0", "x1","y1")),
                  col=adjustcolor(4, .7), length=1/8))
points(dst[i,], col=2)
if(FALSE) { # if(require("maps")) {
   map("state", interior = FALSE,          add = TRUE)
   map("state", boundary = FALSE, lty = 2, add = TRUE)
}
```

`sunspot.month` *Monthly Sunspot Data, from 1749 to "Present"*

Description

Monthly numbers of sunspots, as from the World Data Center, aka SIDC. This is the version of the data that will occasionally be updated when new counts become available.

Usage

```
sunspot.month
```

Format

The univariate time series `sunspot.year` and `sunspot.month` contain 289 and 2988 observations, respectively. The objects are of class `"ts"`.

Author(s)

R

Source

WDC-SILSO, Solar Influences Data Analysis Center (SIDC), Royal Observatory of Belgium, Av. Circulaire, 3, B-1180 BRUSSELS Currently at http://www.sidc.be/silso/datafiles

See Also

sunspot.month is a longer version of sunspots; the latter runs until 1983 and is kept fixed (for reproducibility as example dataset).

Examples

```
require(stats); require(graphics)
## Compare the monthly series
plot (sunspot.month,
      main="sunspot.month & sunspots [package'datasets']", col=2)
lines(sunspots) # -> faint differences where they overlap

## Now look at the difference :
all(tsp(sunspots)     [c(1,3)] ==
    tsp(sunspot.month)[c(1,3)]) ## Start & Periodicity are the same
n1 <- length(sunspots)
table(eq <- sunspots == sunspot.month[1:n1]) #>  132  are different !
i <- which(!eq)
rug(time(eq)[i])
s1 <- sunspots[i] ; s2 <- sunspot.month[i]
cbind(i = i, time = time(sunspots)[i], sunspots = s1, ss.month = s2,
      perc.diff = round(100*2*abs(s1-s2)/(s1+s2), 1))

## How to recreate the "old" sunspot.month (R <= 3.0.3):
.sunspot.diff <- cbind(
    i = c(1202L, 1256L, 1258L, 1301L, 1407L, 1429L, 1452L, 1455L,
          1663L, 2151L, 2329L, 2498L, 2594L, 2694L, 2819L),
    res10 = c(1L, 1L, 1L, -1L, -1L, -1L, 1L, -1L,
          1L, 1L, 1L, 1L, 1L, 20L, 1L))
ssm0 <- sunspot.month[1:2988]
with(as.data.frame(.sunspot.diff), ssm0[i] <<- ssm0[i] - res10/10)
sunspot.month.0 <- ts(ssm0, start = 1749, frequency = 12)
```

---

sunspot.year *Yearly Sunspot Data, 1700–1988*

Description

Yearly numbers of sunspots from 1700 to 1988 (rounded to one digit).

Note that monthly numbers are available as sunspot.month, though starting slightly later.

Usage

```
sunspot.year
```

Format

The univariate time series sunspot.year contains 289 observations, and is of class "ts".

Source

H. Tong (1996) *Non-Linear Time Series*. Clarendon Press, Oxford, p. 471.

See Also

For *monthly* sunspot numbers, see `sunspot.month` and `sunspots`.

Regularly updated yearly sunspot numbers are available from WDC-SILSO, Royal Observatory of Belgium, at `http://www.sidc.be/silso/datafiles`

Examples

```
utils::str(sm <- sunspots)# the monthly version we keep unchanged
utils::str(sy <- sunspot.year)
## The common time interval
(t1 <- c(max(start(sm), start(sy)),     1)) # Jan 1749
(t2 <- c(min(  end(sm)[1],end(sy)[1]), 12)) # Dec 1983
s.m <- window(sm, start=t1, end=t2)
s.y <- window(sy, start=t1, end=t2[1]) # {irrelevant warning}
stopifnot(length(s.y) * 12 == length(s.m),
          ## The yearly series *is* close to the averages of the monthly one:
          all.equal(s.y, aggregate(s.m, FUN = mean), tolerance = 0.0020))
## NOTE: Strangely, correctly weighting the number of days per month
##       (using 28.25 for February) is *not* closer than the simple mean:
ndays <- c(31, 28.25, rep(c(31,30, 31,30, 31), 2))
all.equal(s.y, aggregate(s.m, FUN = mean))                    # 0.0013
all.equal(s.y, aggregate(s.m, FUN = weighted.mean, w = ndays)) # 0.0017
```

`sunspots` *Monthly Sunspot Numbers, 1749–1983*

Description

Monthly mean relative sunspot numbers from 1749 to 1983. Collected at Swiss Federal Observatory, Zurich until 1960, then Tokyo Astronomical Observatory.

Usage

```
sunspots
```

Format

A time series of monthly data from 1749 to 1983.

Source

Andrews, D. F. and Herzberg, A. M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. New York: Springer-Verlag.

See Also

sunspot.month has a longer (and a bit different) series, sunspot.year is a much shorter one. See there for getting more current sunspot numbers.

Examples

```
require(graphics)
plot(sunspots, main = "sunspots data", xlab = "Year",
     ylab = "Monthly sunspot numbers")
```

---

swiss *Swiss Fertility and Socioeconomic Indicators (1888) Data*

Description

Standardized fertility measure and socioeconomic indicators for each of 47 French-speaking provinces of Switzerland at about 1888.

Usage

```
swiss
```

Format

A data frame with 47 observations on 6 variables, *each* of which is in percent, i.e., in $[0, 100]$.

[,1]	Fertility	I_g, 'common standardized fertility measure'
[,2]	Agriculture	% of males involved in agriculture as occupation
[,3]	Examination	% draftees receiving highest mark on army examination
[,4]	Education	% education beyond primary school for draftees.
[,5]	Catholic	% 'catholic' (as opposed to 'protestant').
[,6]	Infant.Mortality	live births who live less than 1 year.

All variables but Fertility give proportions of the population.

Details

(paraphrasing Mosteller and Tukey):

Switzerland, in 1888, was entering a period known as the *demographic transition*; i.e., its fertility was beginning to fall from the high level typical of underdeveloped countries.

The data collected are for 47 French-speaking "provinces" at about 1888.

Here, all variables are scaled to $[0, 100]$, where in the original, all but Catholic were scaled to $[0, 1]$.

Note

Files for all 182 districts in 1888 and other years have been available at https://oprdata.princeton.edu/archive/pefp/switz.aspx.

They state that variables Examination and Education are averages for 1887, 1888 and 1889.

Source

Project "16P5", pages 549–551 in

Mosteller, F. and Tukey, J. W. (1977) *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley, Reading Mass.

indicating their source as "Data used by permission of Franice van de Walle. Office of Population Research, Princeton University, 1976. Unpublished data assembled under NICHD contract number No 1-HD-O-2077."

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
pairs(swiss, panel = panel.smooth, main = "swiss data",
      col = 3 + (swiss$Catholic > 50))
summary(lm(Fertility ~ . , data = swiss))
```

Theoph *Pharmacokinetics of Theophylline*

Description

The Theoph data frame has 132 rows and 5 columns of data from an experiment on the pharmacokinetics of theophylline.

Usage

```
Theoph
```

Format

An object of class c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame") containing the following columns:

Subject an ordered factor with levels 1, ..., 12 identifying the subject on whom the observation was made. The ordering is by increasing maximum concentration of theophylline observed.

Wt weight of the subject (kg).

Dose dose of theophylline administered orally to the subject (mg/kg).

Time time since drug administration when the sample was drawn (hr).

conc theophylline concentration in the sample (mg/L).

Details

Boeckmann, Sheiner and Beal (1994) report data from a study by Dr. Robert Upton of the kinetics of the anti-asthmatic drug theophylline. Twelve subjects were given oral doses of theophylline then serum concentrations were measured at 11 time points over the next 25 hours.

These data are analyzed in Davidian and Giltinan (1995) and Pinheiro and Bates (2000) using a two-compartment open pharmacokinetic model, for which a self-starting model function, `SSfol`, is available.

This dataset was originally part of package `nlme`, and that has methods (including for `[`, `as.data.frame`, `plot` and `print`) for its grouped-data classes.

Source

Boeckmann, A. J., Sheiner, L. B. and Beal, S. L. (1994), *NONMEM Users Guide: Part V*, NONMEM Project Group, University of California, San Francisco.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.5, p. 145 and section 6.6, p. 176)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer (Appendix A.29)

See Also

`SSfol`

Examples

```
require(stats); require(graphics)

coplot(conc ~ Time | Subject, data = Theoph, show.given = FALSE)
Theoph.4 <- subset(Theoph, Subject == 4)
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl),
           data = Theoph.4)
summary(fm1)
plot(conc ~ Time, data = Theoph.4,
     xlab = "Time since drug administration (hr)",
     ylab = "Theophylline concentration (mg/L)",
     main = "Observed concentrations and fitted model",
     sub  = "Theophylline data - Subject 4 only",
     las = 1, col = 4)
xvals <- seq(0, par("usr")[2], length.out = 55)
lines(xvals, predict(fm1, newdata = list(Time = xvals)),
      col = 4)
```

`Titanic` *Survival of passengers on the Titanic*

Description

This data set provides information on the fate of passengers on the fatal maiden voyage of the ocean liner 'Titanic', summarized according to economic status (class), sex, age and survival.

Usage

```
Titanic
```

Format

A 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables. The variables and their levels are as follows:

No	Name	Levels
1	`Class`	1st, 2nd, 3rd, Crew
2	`Sex`	Male, Female
3	`Age`	Child, Adult
4	`Survived`	No, Yes

Details

The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts—from the proportions of first-class passengers to the 'women and children first' policy, and the fact that that policy was not entirely successful in saving the women and children in the third class—are reflected in the survival rates for various classes of passenger.

These data were originally collected by the British Board of Trade in their investigation of the sinking. Note that there is not complete agreement among primary sources as to the exact numbers on board, rescued, or lost.

Due in particular to the very successful film 'Titanic', the last years saw a rise in public interest in the Titanic. Very detailed data about the passengers is now available on the Internet, at sites such as *Encyclopedia Titanica* (`https://www.encyclopedia-titanica.org/`).

Source

Dawson, Robert J. MacG. (1995), The 'Unusual Episode' Data Revisited. *Journal of Statistics Education*, **3**. doi:10.1080/10691898.1995.11910499.

The source provides a data set recording class, sex, age, and survival status for each person on board of the Titanic, and is based on data originally collected by the British Board of Trade and reprinted in:

British Board of Trade (1990), *Report on the Loss of the 'Titanic' (S.S.)*. British Board of Trade Inquiry Report (reprint). Gloucester, UK: Allan Sutton Publishing.

Examples

```
require(graphics)
mosaicplot(Titanic, main = "Survival on the Titanic")
## Higher survival rates in children?
apply(Titanic, c(3, 4), sum)
## Higher survival rates in females?
apply(Titanic, c(2, 4), sum)
## Use loglm() in package 'MASS' for further analysis ...
```

ToothGrowth *The Effect of Vitamin C on Tooth Growth in Guinea Pigs*

Description

The response is the length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid (a form of vitamin C and coded as `VC`).

Usage

```
ToothGrowth
```

Format

A data frame with 60 observations on 3 variables.

[,1]	`len`	numeric	Tooth length
[,2]	`supp`	factor	Supplement type (VC or OJ).
[,3]	`dose`	numeric	Dose in milligrams/day

Source

C. I. Bliss (1952). *The Statistics of Bioassay*. Academic Press.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Crampton, E. W. (1947). The growth of the odontoblast of the incisor teeth as a criterion of vitamin C intake of the guinea pig. *The Journal of Nutrition*, **33**(5), 491–504. doi:10.1093/jn/33.5.491.

Examples

```
require(graphics)
coplot(len ~ dose | supp, data = ToothGrowth, panel = panel.smooth,
       xlab = "ToothGrowth data: length vs dose, given type of supplement")
```

treering *Yearly Tree-Ring Data, -6000–1979*

Description

Contains normalized tree-ring widths in dimensionless units.

Usage

```
treering
```

Format

A univariate time series with 7981 observations. The object is of class `"ts"`.

Each tree ring corresponds to one year.

Details

The data were recorded by Donald A. Graybill, 1980, from Gt Basin Bristlecone Pine 2805M, 3726-11810 in Methuselah Walk, California.

Source

Time Series Data Library: https://robjhyndman.com/TSDL/, series 'CA535.DAT'

References

For some photos of Methuselah Walk see https://web.archive.org/web/20110523225828/http://www.ltrr.arizona.edu/~hallman/sitephotos/meth.html

trees *Diameter, Height and Volume for Black Cherry Trees*

Description

This data set provides measurements of the diameter, height and volume of timber in 31 felled black cherry trees. Note that the diameter (in inches) is erroneously labelled Girth in the data. It is measured at 4 ft 6 in above the ground.

Usage

```
trees
```

Format

A data frame with 31 observations on 3 variables.

[,1]	Girth	numeric	Tree diameter (rather than girth, actually) in inches
[,2]	Height	numeric	Height in ft
[,3]	Volume	numeric	Volume of timber in cubic ft

Source

Meyer, H. A. (1953) *Forest Mensuration*. Penns Valley Publishers, Inc.

Ryan, T. A., Joiner, B. L. and Ryan, B. F. (1976) *The Minitab Student Handbook*. Duxbury Press.

References

Atkinson, A. C. (1985) *Plots, Transformations and Regression*. Oxford University Press.

Examples

```
require(stats); require(graphics)
pairs(trees, panel = panel.smooth, main = "trees data")
plot(Volume ~ Girth, data = trees, log = "xy")
coplot(log(Volume) ~ log(Girth) | Height, data = trees,
       panel = panel.smooth)
summary(fm1 <- lm(log(Volume) ~ log(Girth), data = trees))
summary(fm2 <- update(fm1, ~ . + log(Height), data = trees))
step(fm2)
## i.e., Volume ~= c * Height * Girth^2  seems reasonable
```

UCBAdmissions *Student Admissions at UC Berkeley*

Description

Aggregate data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex.

Usage

```
UCBAdmissions
```

Format

A 3-dimensional array resulting from cross-tabulating 4526 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	`Admit`	Admitted, Rejected
2	`Gender`	Male, Female
3	`Dept`	A, B, C, D, E, F

Details

This data set is frequently used for illustrating Simpson's paradox, see Bickel et al. (1975). At issue is whether the data show evidence of sex bias in admission practices. There were 2691 male applicants, of whom 1198 (44.5%) were admitted, compared with 1835 female applicants of whom 557 (30.4%) were admitted. This gives a sample odds ratio of 1.83, indicating that males were almost twice as likely to be admitted. In fact, graphical methods (as in the example below) or log-linear modelling show that the apparent association between admission and sex stems from differences in the tendency of males and females to apply to the individual departments (females used to apply *more* to departments with higher rejection rates).

This data set can also be used for illustrating methods for graphical display of categorical data, such as the general-purpose mosaicplot or the fourfoldplot for 2-by-2-by-k tables.

References

Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975). Sex bias in graduate admissions: Data from Berkeley. *Science*, **187**, 398–403. doi:10.1126/science.187.4175.398.

Examples

```
require(graphics)
## Data aggregated over departments
apply(UCBAdmissions, c(1, 2), sum)
mosaicplot(apply(UCBAdmissions, c(1, 2), sum),
           main = "Student admissions at UC Berkeley")
## Data for individual departments
opar <- par(mfrow = c(2, 3), oma = c(0, 0, 2, 0))
for(i in 1:6)
  mosaicplot(UCBAdmissions[,,i],
    xlab = "Admit", ylab = "Sex",
    main = paste("Department", LETTERS[i]))
mtext(expression(bold("Student admissions at UC Berkeley")),
      outer = TRUE, cex = 1.5)
par(opar)
```

`UKDriverDeaths` *Road Casualties in Great Britain 1969–84*

Description

`UKDriverDeaths` is a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

`Seatbelts` is more information on the same problem.

Usage

```
UKDriverDeaths
Seatbelts
```

Format

`Seatbelts` is a multiple time series, with columns

`DriversKilled` car drivers killed.

`drivers` same as `UKDriverDeaths`.

`front` front-seat passengers killed or seriously injured.

`rear` rear-seat passengers killed or seriously injured.

`kms` distance driven.

`PetrolPrice` petrol price.

`VanKilled` number of van ('light goods vehicle') drivers.

`law` 0/1: was the law in effect that month?

Source

Harvey, A.C. (1989). *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, pp. 519–523.

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

References

Harvey, A. C. and Durbin, J. (1986). The effects of seat belt legislation on British road casualties: A case study in structural time series modelling. *Journal of the Royal Statistical Society* series A, **149**, 187–227. doi:10.2307/2981553.

Examples

```
require(stats); require(graphics)
## work with pre-seatbelt period to identify a model, use logs
work <- window(log10(UKDriverDeaths), end = 1982+11/12)
par(mfrow = c(3, 1))
plot(work); acf(work); pacf(work)
par(mfrow = c(1, 1))
(fit <- arima(work, c(1, 0, 0), seasonal = list(order = c(1, 0, 0))))
z <- predict(fit, n.ahead = 24)
ts.plot(log10(UKDriverDeaths), z$pred, z$pred+2*z$se, z$pred-2*z$se,
        lty = c(1, 3, 2, 2), col = c("black", "red", "blue", "blue"))

## now see the effect of the explanatory variables
X <- Seatbelts[, c("kms", "PetrolPrice", "law")]
X[, 1] <- log10(X[, 1]) - 4
arima(log10(Seatbelts[, "drivers"]), c(1, 0, 0),
      seasonal = list(order = c(1, 0, 0)), xreg = X)
```

UKgas *UK Quarterly Gas Consumption*

Description

Quarterly UK gas consumption from 1960Q1 to 1986Q4, in millions of therms.

Usage

```
UKgas
```

Format

A quarterly time series of length 108.

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

Examples

```
## maybe str(UKgas) ; plot(UKgas) ...
```

UKLungDeaths *Monthly Deaths from Lung Diseases in the UK*

Description

Three time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979, both sexes (`ldeaths`), males (`mdeaths`) and females (`fdeaths`).

Usage

```
ldeaths
fdeaths
mdeaths
```

Source

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.3

Examples

```
require(stats); require(graphics) # for time
plot(ldeaths)
plot(mdeaths, fdeaths)
## Better labels:
yr <- floor(tt <- time(mdeaths))
plot(mdeaths, fdeaths,
     xy.labels = paste(month.abb[12*(tt - yr)], yr-1900, sep = "'"))
```

USAccDeaths *Accidental Deaths in the US 1973–1978*

Description

A time series giving the monthly totals of accidental deaths in the USA. The values for the first six months of 1979 are 7798 7406 8363 8460 9217 9316.

Usage

```
USAccDeaths
```

Source

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods.* Springer, New York.

`USArrests` *Violent Crime Rates by US State*

Description

This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

Usage

```
USArrests
```

Format

A data frame with 50 observations on 4 variables.

[,1]	`Murder`	numeric	Murder arrests (per 100,000)
[,2]	`Assault`	numeric	Assault arrests (per 100,000)
[,3]	`UrbanPop`	numeric	Percent urban population
[,4]	`Rape`	numeric	Rape arrests (per 100,000)

Note

`USArrests` contains the data as in McNeil's monograph. For the `UrbanPop` percentages, a review of the table (No. 21) in the Statistical Abstracts 1975 reveals a transcription error for Maryland (and that McNeil used the same "round to even" rule that R's `round()` uses), as found by Daniel S Coven (Arizona).

See the example below on how to correct the error and improve accuracy for the '<n>.5' percentages.

Source

World Almanac and Book of facts 1975. (Crime rates).

Statistical Abstracts of the United States 1975, p.20, (Urban rates), possibly available as `https://books.google.ch/books?id=zl9qAAAAMAAJ&pg=PA20`.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

See Also

The `state` data sets.

Examples

```
summary(USArrests)

require(graphics)
pairs(USArrests, panel = panel.smooth, main = "USArrests data")

## Difference between 'USArrests' and its correction
USArrests["Maryland", "UrbanPop"] # 67 -- the transcription error
UA.C <- USArrests
UA.C["Maryland", "UrbanPop"] <- 76.6

## also +/- 0.5 to restore the original  <n>.5  percentages
s5u <- c("Colorado", "Florida", "Mississippi", "Wyoming")
s5d <- c("Nebraska", "Pennsylvania")
UA.C[s5u, "UrbanPop"] <- UA.C[s5u, "UrbanPop"] + 0.5
UA.C[s5d, "UrbanPop"] <- UA.C[s5d, "UrbanPop"] - 0.5

## ==> UA.C  is now a *C*orrected version of  USArrests
```

USJudgeRatings *Lawyers' Ratings of State Judges in the US Superior Court*

Description

Lawyers' ratings of state judges in the US Superior Court.

Usage

```
USJudgeRatings
```

Format

A data frame containing 43 observations on 12 numeric variables.

[,1]	CONT	Number of contacts of lawyer with judge.
[,2]	INTG	Judicial integrity.
[,3]	DMNR	Demeanor.
[,4]	DILG	Diligence.
[,5]	CFMG	Case flow managing.
[,6]	DECI	Prompt decisions.
[,7]	PREP	Preparation for trial.
[,8]	FAMI	Familiarity with law.
[,9]	ORAL	Sound oral rulings.
[,10]	WRIT	Sound written rulings.
[,11]	PHYS	Physical ability.
[,12]	RTEN	Worthy of retention.

Source

New Haven Register, 14 January, 1977 (from John Hartigan).

Examples

```
require(graphics)
pairs(USJudgeRatings, main = "USJudgeRatings data")
```

USPersonalExpenditure *Personal Expenditure Data*

Description

This data set consists of United States personal expenditures (in billions of dollars) in the categories; food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

Usage

```
USPersonalExpenditure
```

Format

A matrix with 5 rows and 5 columns.

Source

The World Almanac and Book of Facts, 1962, page 756.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats) # for medpolish
USPersonalExpenditure
medpolish(log10(USPersonalExpenditure))
```

uspop *Populations Recorded by the US Census*

Description

This data set gives the population of the United States (in millions) as recorded by the decennial census for the period 1790–1970.

Usage

```
uspop
```

Format

A time series of 19 values.

Source

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
plot(uspop, log = "y", main = "uspop data", xlab = "Year",
     ylab = "U.S. Population (millions)")
```

VADeaths *Death Rates in Virginia (1940)*

Description

Death rates per 1000 in Virginia in 1940.

Usage

```
VADeaths
```

Format

A matrix with 5 rows and 4 columns.

Details

The death rates are measured per 1000 population per year. They are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.

This provides a rather nice 3-way analysis of variance example.

Source

Molyneaux, L., Gilliam, S. K., and Florant, L. C.(1947) Differences in Virginia death rates by color, sex, age, and rural or urban residence. *American Sociological Review*, **12**, 525–535.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats); require(graphics)
n <- length(dr <- c(VADeaths))
nam <- names(VADeaths)
d.VAD <- data.frame(
 Drate = dr,
 age = rep(ordered(rownames(VADeaths)), length.out = n),
 gender = gl(2, 5, n, labels = c("M", "F")),
 site =  gl(2, 10, labels = c("rural", "urban")))
coplot(Drate ~ as.numeric(age) | gender * site, data = d.VAD,
       panel = panel.smooth, xlab = "VADeaths data - Given: gender")
summary(aov.VAD <- aov(Drate ~ .^2, data = d.VAD))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0))
plot(aov.VAD)
par(opar)
```

---

volcano *Topographic Information on Auckland's Maunga Whau Volcano*

Description

Maunga Whau (Mt Eden) is one of about 50 volcanos in the Auckland volcanic field. This data set gives topographic information for Maunga Whau on a 10m by 10m grid.

Usage

```
volcano
```

Format

A matrix with 87 rows and 61 columns, rows corresponding to grid lines running east to west and columns to grid lines running south to north.

Source

Digitized from a topographic map by Ross Ihaka. These data should not be regarded as accurate.

See Also

`filled.contour` for a nice plot.

Examples

```
require(grDevices); require(graphics)
filled.contour(volcano, color.palette = terrain.colors, asp = 1)
title(main = "volcano data: filled contour map")
```

`warpbreaks` *The Number of Breaks in Yarn during Weaving*

Description

This data set gives the number of warp breaks per loom, where a loom corresponds to a fixed length of yarn.

Usage

```
warpbreaks
```

Format

A data frame with 54 observations on 3 variables.

`[,1]`	`breaks`	numeric	The number of breaks
`[,2]`	`wool`	factor	The type of wool (A or B)
`[,3]`	`tension`	factor	The level of tension (L, M, H)

There are measurements on 9 looms for each of the six types of warp (`AL`, `AM`, `AH`, `BL`, `BM`, `BH`).

Source

Tippett, L. H. C. (1950) *Technological Applications of Statistics*. Wiley. Page 106.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

See Also

`xtabs` for ways to display these data as a table.

Examples

```
require(stats); require(graphics)
summary(warpbreaks)
opar <- par(mfrow = c(1, 2), oma = c(0, 0, 1.1, 0))
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "A", main = "Wool A")
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "B", main = "Wool B")
mtext("warpbreaks data", side = 3, outer = TRUE)
par(opar)
summary(fm1 <- lm(breaks ~ wool*tension, data = warpbreaks))
anova(fm1)
```

---

women *Average Heights and Weights for American Women*

Description

This data set gives the average heights and weights for American women aged 30–39.

Usage

```
women
```

Format

A data frame with 15 observations on 2 variables.

`[,1]`	`height`	numeric	Height (in)
`[,2]`	`weight`	numeric	Weight (lbs)

Details

The data set appears to have been taken from the American Society of Actuaries *Build and Blood Pressure Study* for some (unknown to us) earlier year.

The World Almanac notes: "The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes".

Source

The World Almanac and Book of Facts, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

`WorldPhones` *The World's Telephones*

Description

The number of telephones in various regions of the world (in thousands).

Usage

```
WorldPhones
```

Format

A matrix with 7 rows and 8 columns. The columns of the matrix give the figures for a given region, and the rows the figures for a year.

The regions are: North America, Europe, Asia, South America, Oceania, Africa, Central America.

The years are: 1951, 1956, 1957, 1958, 1959, 1960, 1961.

Source

AT&T (1961) *The World's Telephones*.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
matplot(rownames(WorldPhones), WorldPhones, type = "b", log = "y",
        xlab = "Year", ylab = "Number of telephones (1000's)")
legend(1951.5, 80000, colnames(WorldPhones), col = 1:6, lty = 1:5,
       pch = rep(21, 7))
title(main = "World phones data: log scale for response")
```

WWWusage *Internet Usage per Minute*

Description

A time series of the numbers of users connected to the Internet through a server every minute.

Usage

```
WWWusage
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

References

Makridakis, S., Wheelwright, S. C. and Hyndman, R. J. (1998). *Forecasting: Methods and Applications*. Wiley.

Examples

```
require(graphics)
work <- diff(WWWusage)
par(mfrow = c(2, 1)); plot(WWWusage); plot(work)
## Not run:
require(stats)
aics <- matrix(, 6, 6, dimnames = list(p = 0:5, q = 0:5))
for(q in 1:5) aics[1, 1+q] <- arima(WWWusage, c(0, 1, q),
    optim.control = list(maxit = 500))$aic
for(p in 1:5)
   for(q in 0:5) aics[1+p, 1+q] <- arima(WWWusage, c(p, 1, q),
       optim.control = list(maxit = 500))$aic
round(aics - min(aics, na.rm = TRUE), 2)

## End(Not run)
```

Chapter 4

The grDevices package

---

grDevices-package *The R Graphics Devices and Support for Colours and Fonts*

Description

Graphics devices and support for base and grid graphics

Details

This package contains functions which support both base and grid graphics.

For a complete list of functions, use `library(help = "grDevices")`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

---

adjustcolor *Adjust Colors in One or More Directions Conveniently*

Description

Adjust or modify a vector of colors by "turning knobs" on one or more coordinates in (r, g, b, α) space, typically by up or down scaling them.

Usage

```
adjustcolor(col, alpha.f = 1, red.f = 1, green.f = 1, blue.f = 1,
            offset = c(0, 0, 0, 0),
            transform = diag(c(red.f, green.f, blue.f, alpha.f)))
```

Arguments

col	vector of colors, in any format that col2rgb() accepts
alpha.f	factor modifying the opacity alpha; typically in [0,1]
red.f, green.f, blue.f	factors modifying the "red-", "green-" or "blue-"ness of the colors, respectively.
offset	numeric vector of length 4 to offset x := c(r,g,b,alpha), where x is the $[0, 1]$-scaled result of col2rgb(col, alpha=TRUE).
transform	a 4x4 numeric matrix applied to x + offset.

Value

a color vector of the same length as col, effectively the result of rgb().

See Also

rgb, col2rgb. For more sophisticated color constructions: convertColor

Examples

```
## Illustrative examples :
opal <- palette("default")
stopifnot(identical(adjustcolor(1:8,       0.75),
                    adjustcolor(palette(), 0.75)))
cbind(palette(), adjustcolor(1:8, 0.75))

##  alpha = 1/2 * previous alpha --> opaque colors
x <- palette(adjustcolor(palette(), 0.5))

sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, type = "b", pch = 21:23, col = 2:5, bg = 2:5,
        main = "Using an 'opaque ('translucent') color palette")

x. <- adjustcolor(x, offset = c(0.5, 0.5, 0.5, 0), # <- "more white"
                  transform = diag(c(.7, .7, .7, 0.6)))
cbind(x, x.)
op <- par(bg = adjustcolor("goldenrod", offset = -rep(.4, 4)), xpd = NA)
plot(0:9, 0:9, type = "n", axes = FALSE, xlab = "", ylab = "",
     main = "adjustcolor() -> translucent")
text(1:8, labels = paste0(x,"++"), col = x., cex = 8)
par(op)

## and

(M <- cbind( rbind( matrix(1/3, 3, 3), 0), c(0, 0, 0, 1)))
adjustcolor(x, transform = M)

## revert to previous palette: active
palette(opal)
```

as.graphicsAnnot *Coerce an Object for Graphics Annotation*

Description

Coerce an R object into a form suitable for graphics annotation.

Usage

```
as.graphicsAnnot(x)
```

Arguments

x	an R object

Details

Expressions, calls and names (as used by plotmath) are passed through unchanged. All other objects with an explicit class (as determined by `is.object`) are coerced by `as.character` to character vectors.

All the **graphics** and **grid** functions which use this coerce calls and names to expressions internally.

Value

A language object or a character vector.

as.raster *Create a Raster Object*

Description

Functions to create a raster object (representing a bitmap image) and coerce other objects to a raster object.

Usage

```
is.raster(x)
as.raster(x, ...)

## S3 method for class 'matrix'
as.raster(x, max = 1, ...)
## S3 method for class 'array'
as.raster(x, max = 1, ...)

## S3 method for class 'logical'
```

```
as.raster(x, max = 1, ...)
## S3 method for class 'numeric'
as.raster(x, max = 1, ...)
## S3 method for class 'character'
as.raster(x, max = 1, ...)
## S3 method for class 'raw'
as.raster(x, max = 255L, ...)
```

Arguments

x	any R object.
max	number giving the maximum of the color values range.
...	further arguments passed to or from other methods.

Details

An object of class "raster" is a matrix of colour values as given by rgb representing a bitmap image.

It is not expected that the user will need to call these functions directly; functions to render bitmap images in graphics packages will make use of the as.raster() function to generate a raster object from their input.

The as.raster() function is (S3) generic so methods can be written to convert other R objects to a raster object.

The default implementation for numeric matrices interprets scalar values on black-to-white scale.

Raster objects can be subsetted like a matrix and it is possible to assign to a subset of a raster object.

There is a method for converting a raster object to a matrix (of colour strings).

Raster objects can be compared for equality or inequality (with each other or with a colour string).

There is a is.na method which returns a logical matrix of the same dimensions as the raster object. Note that NA values are interpreted as the fully transparent colour by some (but not all) graphics devices.

Value

For as.raster(), a raster object.

For is.raster(), a logical indicating whether x is a raster object.

Note

Raster images are internally represented row-first, which can cause confusion when trying to manipulate a raster object. The recommended approach is to coerce a raster to a matrix, perform the manipulation, then convert back to a raster.

Examples

```
# A red gradient
as.raster(matrix(hcl(0, 80, seq(50, 80, 10)),
                 nrow = 4, ncol = 5))

# Vectors are 1-column matrices ...
#   character vectors are color names ...
as.raster(hcl(0, 80, seq(50, 80, 10)))
#   numeric vectors are greyscale ...
as.raster(1:5, max = 5)
#   logical vectors are black and white ...
as.raster(1:10 %% 2 == 0)

# ... unless nrow/ncol are supplied ...
as.raster(1:10 %% 2 == 0, nrow = 1)

# Matrix can also be logical or numeric (or raw) ...
as.raster(matrix(c(TRUE, FALSE), nrow = 3, ncol = 2))
as.raster(matrix(1:3/4, nrow = 3, ncol = 4))

# An array can be 3-plane numeric (R, G, B planes) ...
as.raster(array(c(0:1, rep(0.5, 4)), c(2, 1, 3)))

# ... or 4-plane numeric (R, G, B, A planes)
as.raster(array(c(0:1, rep(0.5, 6)), c(2, 1, 4)))

# subsetting
r <- as.raster(matrix(colors()[1:100], ncol = 10))
r[, 2]
r[2:4, 2:5]

# assigning to subset
r[2:4, 2:5] <- "white"

# comparison
r == "white"
```

`axisTicks` *Compute Pretty Axis Tick Scales*

Description

Compute pretty axis scales and tick mark locations, the same way as traditional R graphics do it. This is interesting particularly for log scale axes.

Usage

```
axisTicks(usr, log, axp = NULL, nint = 5)
.axisPars(usr, log = FALSE,  nintLog = 5)
```

Arguments

usr	numeric vector of length 2, with c(min, max) axis extents.
log	logical indicating if a log scale is (thought to be) in use.
axp	numeric vector of length 3, c(mi, ma, n.), with identical meaning to par("?axp") (where ? is x or y), namely "pretty" axis extents, and an integer *code* n..
nint, nintLog	positive integer value indicating (*approximately*) the desired number of intervals. nintLog is used **only** for the case log = TRUE.

Details

axisTicks(usr, *) calls .axisPars(usr, ..) to set axp when that is missing or NULL.

Apart from that, axisTicks() just calls the C function CreateAtVector() in '*R*/src/main/plot.c' which is also called by the base **graphics** package function axis(side, *) when its argument at is not specified.

Since R 4.1.0, the underlying C CreateAtVector() has been tuned to provide a considerably more balanced (symmetric) set of tick locations.

Value

axisTicks() returns a numeric vector of potential axis tick locations, of length approximately nint+1.

.axisPars() returns a list with components

axp	numeric vector of length 2, c(min., max.), of pretty axis extents.
n	integer (code), with the same meaning as par("?axp")[3].

See Also

axTicks, axis, and par all from the **graphics** package.

Examples

```
##--- Demonstrating correspondence between graphics'
##--- axis() and the graphics-engine agnostic  axisTicks() :

require("graphics")
plot(10*(0:10)); (pu <- par("usr"))
aX <- function(side, at, ...)
    axis(side, at = at, labels = FALSE, lwd.ticks = 2, col.ticks = 2,
         tck = 0.05, ...)
aX(1, print(xa <- axisTicks(pu[1:2], log = FALSE)))  # x axis
aX(2, print(ya <- axisTicks(pu[3:4], log = FALSE)))  # y axis

axisTicks(pu[3:4], log = FALSE, nint = 10)

plot(10*(0:10), log = "y"); (pu <- par("usr"))
aX(2, print(ya <- axisTicks(pu[3:4], log = TRUE)))  # y axis
```

```
plot(2^(0:9), log = "y"); (pu <- par("usr"))
aX(2, print(ya <- axisTicks(pu[3:4], log = TRUE)))  # y axis
```

`boxplot.stats` *Box Plot Statistics*

Description

This function is typically called by another function to gather the statistics necessary for producing box plots, but may be invoked separately.

Usage

```
boxplot.stats(x, coef = 1.5, do.conf = TRUE, do.out = TRUE)
```

Arguments

`x`	a numeric vector for which the boxplot will be constructed (`NA`s and `NaN`s are allowed and omitted).
`coef`	this determines how far the plot 'whiskers' extend out from the box. If `coef` is positive, the whiskers extend to the most extreme data point which is no more than `coef` times the length of the box away from the box. A value of zero causes the whiskers to extend to the data extremes (and no outliers be returned).
`do.conf`, `do.out`	logicals; if `FALSE`, the `conf` or `out` component respectively will be empty in the result.

Details

The two 'hinges' are versions of the first and third quartile, i.e., close to `quantile(x, c(1,3)/4)`. The hinges equal the quartiles for odd n (where `n <- length(x)`) and differ for even n. Whereas the quartiles only equal observations for `n %% 4 == 1` ($n \equiv 1 \mod 4$), the hinges do so *additionally* for `n %% 4 == 2` ($n \equiv 2 \mod 4$), and are in the middle of two observations otherwise.

The notches (if requested) extend to `+/-1.58 IQR/sqrt(n)`. This seems to be based on the same calculations as the formula with 1.57 in Chambers et al. (1983, p. 62), given in McGill et al. (1978, p. 16). They are based on asymptotic normality of the median and roughly equal sample sizes for the two medians being compared, and are said to be rather insensitive to the underlying distributions of the samples. The idea appears to be to give roughly a 95% confidence interval for the difference in two medians.

Value

A list with named components as follows:

`stats`	a vector of length 5, containing the extreme of the lower whisker, the lower 'hinge', the median, the upper 'hinge' and the extreme of the upper whisker. For `coef = 0`, this vector is identical to `fivenum(x, na.rm = TRUE)`.

`n`	the number of non-`NA` observations in the sample.
`conf`	the lower and upper extremes of the 'notch' (`if(do.conf)`). See the details.
`out`	the values of any data points which lie beyond the extremes of the whiskers (`if(do.out)`).

Note that `stats` and `conf` are sorted in *in*creasing order, unlike S, and that `n` and `out` include any `+- Inf` values.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*. Section 2C.

McGill, R., Tukey, J. W. and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, **32**, 12–16. doi:10.2307/2683468.

Velleman, P. F. and Hoaglin, D. C. (1981). *Applications, Basics and Computing of Exploratory Data Analysis*. Duxbury Press.

Emerson, J. D and Strenio, J. (1983). Boxplots and batch comparison. Chapter 3 of *Understanding Robust and Exploratory Data Analysis*, eds. D. C. Hoaglin, F. Mosteller and J. W. Tukey. Wiley.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

See Also

`fivenum`, `boxplot`, `bxp`.

Examples

```
require(stats)
x <- c(1:100, 1000)
(b1 <- boxplot.stats(x))
(b2 <- boxplot.stats(x, do.conf = FALSE, do.out = FALSE))
stopifnot(b1 $ stats == b2 $ stats) # do.out = FALSE is still robust
boxplot.stats(x, coef = 3, do.conf = FALSE)

## no outlier treatment:
(b3 <- boxplot.stats(x, coef = 0))
stopifnot(b3$stats == fivenum(x))

## missing values are ignored
stopifnot(identical(boxplot.stats(c(x, NA)), b1))
## ... infinite values are not:
(r <- boxplot.stats(c(x, -1:1/0)))
stopifnot(r$out == c(1000, -Inf, Inf))
```

bringToTop *Assign Focus to a Window*

Description

bringToTop brings the specified screen device's window to the front of the window stack (and gives it focus). With first argument -1 it brings the console to the top.

If stay = TRUE, the window is designated as a topmost window, i.e. it will stay on top of any regular window. stay may only be used when RGui is run in SDI mode. This corresponds to the "Stay on top" popup menu item in RGui.

Usage

```
bringToTop(which = dev.cur(), stay = FALSE)
```

Arguments

which	a device number, or -1.
stay	whether to make the window stay on top.

See Also

msgWindow, windows

cairo *Cairographics-based SVG, PDF and PostScript Graphics Devices*

Description

Graphics devices for SVG, PDF and PostScript graphics files using the cairo graphics API.

Usage

```
svg(filename = if(onefile) "Rplots.svg" else "Rplot%03d.svg",
    width = 7, height = 7, pointsize = 12,
    onefile = FALSE, family = "sans", bg = "white",
    antialias = c("default", "none", "gray", "subpixel"),
    symbolfamily)

cairo_pdf(filename = if(onefile) "Rplots.pdf" else "Rplot%03d.pdf",
          width = 7, height = 7, pointsize = 12,
          onefile = TRUE, family = "sans", bg = "white",
          antialias = c("default", "none", "gray", "subpixel"),
          fallback_resolution = 300, symbolfamily)
```

```
cairo_ps(filename = if(onefile) "Rplots.ps" else "Rplot%03d.ps",
         width = 7, height = 7, pointsize = 12,
         onefile = TRUE, family = "sans", bg = "white",
         antialias = c("default", "none", "gray", "subpixel"),
         fallback_resolution = 300, symbolfamily)
```

Arguments

filename	the file path of the output file(s). The page number is substituted if a C integer format is included in the character string, as in the default. (Depending on the platform, the result must be less than PATH_MAX characters long, and may be truncated if not. See pdf for further details.) Tilde expansion is performed where supported by the platform.
width	the width of the device in inches.
height	the height of the device in inches.
pointsize	the default pointsize of plotted text (in big points).
onefile	should all plots appear in one file or in separate files?
family	one of the device-independent font families, "sans", "serif" and "mono", or a character string specify a font family to be searched for in a system-dependent way. On unix-alikes (incl.\ macOS), see the 'Cairo fonts' section in the help for X11.
bg	the initial background colour: can be overridden by setting par("bg").
antialias	string, the type of anti-aliasing (if any) to be used; defaults to "default".
fallback_resolution	numeric: the resolution in dpi used when falling back to bitmap output.
symbolfamily	a length-one character string that specifies the font family to be used as the "symbol" font (e.g., for plotmath output).

Details

SVG (Scalar Vector Graphics) is a W3C standard for vector graphics. See https://www.w3.org/Graphics/SVG/. The output from svg is SVG version 1.1 for onefile = FALSE (the default), otherwise SVG 1.2. (SVG 1.2 never passed the draft stage. Few SVG viewers are capable of displaying multi-page SVG files, and they have been dropped from SVG 2.0 (still in draft).)

Note that unlike pdf and postscript, cairo_pdf and cairo_ps sometimes record *bitmaps* and not vector graphics. On the other hand, they can (on suitable platforms) include a much wider range of UTF-8 glyphs, and embed the fonts used.

The output produced by cairo_ps(onefile = FALSE) will be encapsulated postscript on a platform with cairo >= 1.6.

R can be compiled without support for any of these devices: this will be reported if you attempt to use them on a system where they are not supported.

If you plot more than one page on one of these devices and do not include something like %d for the sequence number in filename (or set onefile = TRUE) the file will contain the last page plotted.

There is full support of semi-transparency, but using this is one of the things liable to trigger bitmap output (and will always do so for cairo_ps).

Value

A plot device is opened: nothing is returned to the R interpreter.

Anti-aliasing

Anti-aliasing is applied to both graphics and fonts. It is generally preferable for lines and text, but can lead to undesirable effects for fills, e.g. for `image` plots, and so is never used for fills.

`antialias = "default"` is in principle platform-dependent, but seems most often equivalent to `antialias = "gray"`.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is in pixels (`svg`) or inches.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are multiples of 1/96 inch.
- Circle radii have a minimum of 1/72 inch.
- Colours are interpreted by the viewing application.

Warning

Support for all these devices are optional, so in packages they should be used conditionally after checking `capabilities("cairo")`.

Note

In principle these devices are independent of X11 (as is seen by their presence on Windows). But on a Unix-alike the cairo libraries may be distributed as part of the X11 system and hence that (for example, on macOS, XQuartz) may need to be installed.

See Also

`Devices`, `dev.print`, `pdf`, `postscript`

`capabilities` to see if cairo is supported.

cairoSymbolFont *Specify a Symbol Font*

Description

Specify a symbol font for a Cairo-based graphics device. This function provides the opportunity to specify whether the font supports Private Use Area code points.

Usage

```
cairoSymbolFont(family, usePUA = TRUE)
```

Arguments

family	A character vector giving the symbol font family name.
usePUA	Does the font support Private Use Area code points?

Details

On Cairo-based graphics devices, when drawing with a symbol font (e.g., plotmath), Adobe Symbol Encoding characters are converted to UTF-8 code points. This conversion can use Private Use Area code points or not. It is useful to be able to specify this option because some fonts (e.g., the OpenSymbol font that is included in LibreOffice) have glyphs mapped to the Private Use Area and some fonts (e.g., Nimbus Sans L, the URW Fonts equivalent of Helvetica) do not.

Value

An object of class "CairoSymbolFont".

See Also

cairo_pdf.

Examples

```
## Not run:
## If a font uses PUA, we can just specify the font name ...
cairo_pdf(symbolfamily="OpenSymbol")
dev.off()
## ... or equivalently ...
cairo_pdf(symbolfamily=cairoSymbolFont("OpenSymbol"))
dev.off()

## If a font does not use PUA, we must indicate that ...
cairo_pdf(symbolfamily=cairoSymbolFont("Nimbus Sans", usePUA=FALSE))
dev.off()

## End(Not run)
```

check.options *Set Options with Consistency Checks*

Description

Utility function for setting options with some consistency checks. The attributes of the new settings in new are checked for consistency with the *model* (often default) list in name.opt.

Usage

```
check.options(new, name.opt, reset = FALSE, assign.opt = FALSE,
              envir = .GlobalEnv,
              check.attributes = c("mode", "length"),
              override.check = FALSE)
```

Arguments

new	a *named* list
name.opt	character with the name of R object containing the default list.
reset	logical; if TRUE, reset the options from name.opt. If there is more than one R object with name name.opt, remove the first one in the search() path.
assign.opt	logical; if TRUE, assign the ...
envir	the environment used for get and assign.
check.attributes	character containing the attributes which check.options should check.
override.check	logical vector of length length(new) (or 1 which entails recycling). For those new[i] where override.check[i] == TRUE, the checks are overridden and the changes made anyway.

Value

A list of components with the same names as the one called name.opt. The values of the components are changed from the new list, as long as these pass the checks (when these are not overridden according to override.check).

Note

Option "names" is exempt from all the checks or warnings, as in the application it can be NULL or a variable-length character vector.

Author(s)

Martin Maechler

See Also

ps.options and pdf.options, which use check.options.

Examples

```
(L1 <- list(a = 1:3, b = pi, ch = "CH"))
check.options(list(a = 0:2), name.opt = "L1")
check.options(NULL, reset = TRUE, name.opt = "L1")
```

`chull` *Compute Convex Hull of a Set of Points*

Description

Computes the subset of points which lie on the convex hull of the set of points specified.

Usage

```
chull(x, y = NULL)
```

Arguments

x, y	coordinate vectors of points. This can be specified as two vectors `x` and `y`, a 2-column matrix `x`, a list `x` with two components, etc, see `xy.coords`.

Details

`xy.coords` is used to interpret the specification of the points. Infinite, missing and `NaN` values are not allowed.

The algorithm is that given by Eddy (1977).

Value

An integer vector giving the indices of the unique points lying on the convex hull, in clockwise order. (The first will be returned for duplicate points.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Eddy, W. F. (1977). A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, **3**, 398–403. doi:10.1145/355759.355766.

Eddy, W. F. (1977). Algorithm 523: CONVEX, A new convex hull algorithm for planar sets [Z]. *ACM Transactions on Mathematical Software*, **3**, 411–412. doi:10.1145/355759.355768.

See Also

`xy.coords`, `polygon`

Examples

```
X <- matrix(stats::rnorm(2000), ncol = 2)
chull(X)

plot(X, cex = 0.5)
polygon(X[chull(X), ])
```

---

cm *Unit Transformation*

Description

Translates from inches to cm (centimeters).

Usage

```
cm(x)
```

Arguments

x	numeric vector

Examples

```
cm(1)  # = 2.54

## Translate *from* cm *to* inches:

10 / cm(1) # -> 10cm  are 3.937 inches
```

---

col2rgb *Color to RGB Conversion*

Description

R color to RGB (red/green/blue) conversion.

Usage

```
col2rgb(col, alpha = FALSE)
```

Arguments

col	vector of any of the three kinds of R color specifications, i.e., either a color name (as listed by colors()), a hexadecimal string (see Details), or a positive integer i meaning palette()[i].
alpha	logical value indicating whether the alpha channel (opacity) values should be returned.

Details

NA (as integer or character) and "NA" mean transparent, which can also be specified as "transparent".

Values of col not of one of these types are coerced: real vectors are coerced to integer and other types to character. (factors are coerced to character: in all other cases the class is ignored when doing the coercion.)

Hexadecimal string colors can be in the long hexadecimal form (e.g., "#rrggbb" or "#rrggbbaa") or the short form (e.g, "#rgb" or "#rgba"). The short form is expanded to the long form by replicating digits (not by adding zeroes), e.g., "#rgb" becomes "#rrggbb".

Zero and negative values of col are an error.

Value

An integer matrix with three or four (for alpha = TRUE) rows and number of columns the length of col. If col has names these are used as the column names of the return value.

Author(s)

Martin Maechler and the R core team.

See Also

rgb, colors, palette, etc.

The newer, more flexible interface, convertColor().

Examples

```
col2rgb("peachpuff")
col2rgb(c(blu = "royalblue", reddish = "tomato"))  # note: colnames

col2rgb(1:8)  # the ones from the palette() (if the default)

col2rgb(paste0("gold", 1:4))

col2rgb("#08a0ff")
## all three kinds of color specifications:
col2rgb(c(red = "red", hex = "#abcdef"))
col2rgb(c(palette = 1:3))

# long and short form of hexadecimal notation
col2rgb(c(long = "#559955", short = "#595"))
# with alpha
col2rgb(c(long = "#559955BB", short = "#595B"), alpha = TRUE)

##-- NON-INTRODUCTORY examples --

grC <- col2rgb(paste0("gray", 0:100))
table(print(diff(grC["red",])))  # '2' or '3': almost equidistant
## The 'named' grays are in between {"slate gray" is not gray, strictly}
```

```
col2rgb(c(g66 = "gray66", darkg =  "dark gray", g67 = "gray67",
          g74 = "gray74", gray  =       "gray", g75 = "gray75",
          g82 = "gray82", light = "light gray", g83 = "gray83"))

crgb <- col2rgb(cc <- colors())
colnames(crgb) <- cc
t(crgb)  # The whole table

## How many names are 'aliases' of each other?
ccodes <- c(256^(2:0) %*% crgb)
cl <- split(cc, ccodes)
length(cl) # 502 distinct colors
table(tcc <- lengths(cl))
## All the multiply named colors:
clmult <- cl[tcc >= 2]
names(clmult) <- sapply(clmult, function(x) paste(crgb[,x[1]], collapse = ","))
utils::str(clmult)

## Look at the color cube:
tc <- t(crgb[, !duplicated(ccodes)])
cNms <- rownames(tc)
if(requireNamespace("lattice", quietly = TRUE))
    lattice::cloud(blue ~ red + green, data = as.data.frame(tc), col = cNms)
## The 8 corners of the color cube:
isC <- rowSums(tc == 0 | tc == 255) == 3
cNms[isC] # "white" "black" "blue" "cyan" "green" "magenta" "red" "yellow"

table(is.gray <- tc[,1] == tc[,2] & tc[,2] == tc[,3])  # (397, 105)

## Not run: ## Look at the color cube dynamically:
 if(require("rgl")) {
   open3d(windowRect = c(50,50, 950, 950)) # large, so we see details
   plot3d (tc, col = cNms, size = 11) # --> rotate w/ mouse; enlarged corners:
   points3d(tc[isC,], col = cNms[isC], size=22)
   bg3d("darkgray") # (to "see more"); rotate around gray-axis:
   play3d(spin3d(axis = c(1, 1, 1), rpm = 2), duration = 30)
   if(FALSE) # add all names {zoom in with 2nd mouse button!}
     text3d(tc[!is.gray,], texts = cNms[!is.gray],
                             col = cNms[!is.gray], adj=-1/4, cex = 1/2)
   if(FALSE) { ## next version of {rgl}
     hover3d(tc, labels = cNms)
     message("Move mouse over plot to identify points.")
   } else { ##  click on blob to see colors()' name:
     identify3d(tc, labels=cNms)
   }
 }

## End(Not run)
```

---

colorRamp *Color interpolation*

Description

These functions return functions that interpolate a set of given colors to create new color palettes (like `topo.colors`) and color ramps, functions that map the interval $[0, 1]$ to colors (like `grey`).

Usage

```
colorRamp(colors, bias = 1, space = c("rgb", "Lab"),
          interpolate = c("linear", "spline"), alpha = FALSE)
colorRampPalette(colors, ...)
```

Arguments

`colors`	colors to interpolate; must be a valid argument to `col2rgb()`.
`bias`	a positive number. Higher values give more widely spaced colors at the high end.
`space`	a character string; interpolation in RGB or CIE Lab color spaces.
`interpolate`	use spline or linear interpolation.
`alpha`	logical: should alpha channel (opacity) values be returned? It is an error to give a true value if `space` is specified.
`...`	arguments to pass to `colorRamp`.

Details

The CIE Lab color space is approximately perceptually uniform, and so gives smoother and more uniform color ramps. On the other hand, palettes that vary from one hue to another via white may have a more symmetrical appearance in RGB space.

The conversion formulas in this function do not appear to be completely accurate and the color ramp may not reach the extreme values in Lab space. Future changes in the R color model may change the colors produced with `space = "Lab"`.

Value

`colorRamp` returns a `function` with argument a vector of values between 0 and 1 that are mapped to a numeric matrix of RGB color values with one row per color and 3 or 4 columns.

`colorRampPalette` returns a function that takes an integer argument (the required number of colors) and returns a character vector of colors (see `rgb`) interpolating the given sequence (similar to `heat.colors` or `terrain.colors`).

See Also

Good starting points for interpolation are the "sequential" and "diverging" ColorBrewer palettes in the **RColorBrewer** package.

`splinefun` or `approxfun` are used for interpolation.

Examples

```
## Both return a *function* :
colorRamp(c("red", "green"))( (0:4)/4 ) ## (x) , x in [0,1]
colorRampPalette(c("blue", "red"))( 4 ) ## (n)
## a ramp in opacity of blue values
colorRampPalette(c(rgb(0,0,1,1), rgb(0,0,1,0)), alpha = TRUE)(8)

require(graphics)

## Here space="rgb" gives palettes that vary only in saturation,
## as intended.
## With space="Lab" the steps are more uniform, but the hues
## are slightly purple.
filled.contour(volcano,
               color.palette =
                   colorRampPalette(c("red", "white", "blue")),
               asp = 1)
filled.contour(volcano,
               color.palette =
                   colorRampPalette(c("red", "white", "blue"),
                                    space = "Lab"),
               asp = 1)

## Interpolating a 'sequential' ColorBrewer palette
YlOrBr <- c("#FFFFD4", "#FED98E", "#FE9929", "#D95F0E", "#993404")
filled.contour(volcano,
               color.palette = colorRampPalette(YlOrBr, space = "Lab"),
               asp = 1)
filled.contour(volcano,
               color.palette = colorRampPalette(YlOrBr, space = "Lab",
                                                bias = 0.5),
               asp = 1)

## 'jet.colors' is "as in Matlab"
## (and hurting the eyes by over-saturation)
jet.colors <-
  colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan",
                     "#7FFF7F", "yellow", "#FF7F00", "red", "#7F0000"))
filled.contour(volcano, color.palette = jet.colors, asp = 1)

## space="Lab" helps when colors don't form a natural sequence
m <- outer(1:20,1:20,function(x,y) sin(sqrt(x*y)/3))
rgb.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "rgb")
Lab.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "Lab")
filled.contour(m, col = rgb.palette(20))
filled.contour(m, col = Lab.palette(20))
```

---

colors *Color Names*

Description

Returns the built-in color names which R knows about.

Usage

```
colors (distinct = FALSE)
colours(distinct = FALSE)
```

Arguments

distinct	logical indicating if the colors returned should all be distinct; e.g., "snow" and "snow1" are effectively the same point in the $(0 : 255)^3$ RGB space.

Details

These color names can be used with a col= specification in graphics functions.

An even wider variety of colors can be created with primitives rgb, hsv and hcl, or the derived rainbow, heat.colors, etc.

"transparent" is not a color and so not listed, but it is accepted as a color specification.

Value

A character vector containing all the built-in color names.

See Also

palette for setting the 'palette' of colors for col=*index* specifications.

rgb, hsv, hcl, gray; rainbow for a nice example; and heat.colors, topo.colors for images.

col2rgb for translating to RGB numbers and extended examples.

Examples

```
cl <- colors()
length(cl); cl[1:20]

length(cl. <- colors(TRUE))
## only 502 of the 657 named ones

## ----------- Show all named colors and more:
demo("colors")
## -----------
```

`contourLines` *Calculate Contour Lines*

Description

Calculate contour lines for a given set of data.

Usage

```
contourLines(x = seq(0, 1, length.out = nrow(z)),
             y = seq(0, 1, length.out = ncol(z)),
             z, nlevels = 10,
             levels = pretty(range(z, na.rm = TRUE), nlevels))
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
nlevels	number of contour levels desired **iff** levels is not supplied.
levels	numeric vector of levels at which to draw contour lines.

Details

`contourLines` draws nothing, but returns a set of contour lines.

There is currently no documentation about the algorithm. The source code is in '*R_HOME*/src/main/plot3d.c'.

Value

A list of contours, each itself a list with elements:

level	The contour level.
x	The x-coordinates of the contour.
y	The y-coordinates of the contour.

See Also

`options("max.contour.segments")` for the maximal complexity of a single contour line.

`contour`: Its 'Examples' demonstrate how `contourLines()` can be drawn and are the same (as those from `contour()`).

Examples

```
x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
cl <- contourLines(x, y, volcano)
## summarize the sizes of each the contour lines :
cbind(lev = vapply(cl, `[[`, .5, "level"),
      n   = vapply(cl, function(l) length(l$x), 1))

z <- outer(-9:25, -9:25)
pretty(range(z), 10) # -300 -200 ... 600 700
utils::str(c2 <- contourLines(z))
   # no segments for {-300, 700};
   #  2 segments for {-200, -100, 0}
   #  1 segment  for  100:600
```

`convertColor` *Convert between Colour Spaces*

Description

Convert colours between their representations in standard colour spaces.

Usage

```
convertColor(color, from, to, from.ref.white, to.ref.white,
             scale.in = 1, scale.out = 1, clip = TRUE)
```

Arguments

`color`	A matrix whose rows specify colors. The function will also accept a data frame, but will silently convert to a matrix internally.
`from`, `to`	Input and output color spaces. See 'Details' below.
`from.ref.white`, `to.ref.white`	Reference whites or `NULL` if these are built in to the definition, as for RGB spaces. `D65` is the default, see 'Details' for others.
`scale.in`, `scale.out`	Input is divided by `scale.in`, output is multiplied by `scale.out`. Use `NULL` to suppress scaling when input or output is not numeric.
`clip`	If `TRUE`, truncate RGB output to [0,1], `FALSE` return out-of-range RGB, `NA` set out of range colors to `NaN`.

Details

Color spaces are specified by objects of class `colorConverter`, created by `colorConverter` or `make.rgb`. Built-in color spaces may be referenced by strings: `"XYZ"`, `"sRGB"`, `"Apple RGB"`, `"CIE RGB"`, `"Lab"`, `"Luv"`. The converters for these colour spaces are in the object `colorspaces`.

The "sRGB" color space is that used by standard PC monitors. "Apple RGB" is used by Apple monitors. "Lab" and "Luv" are approximately perceptually uniform spaces standardized by the Commission Internationale d'Eclairage. XYZ is a 1931 CIE standard capable of representing all visible colors (and then some), but not in a perceptually uniform way.

The Lab and Luv spaces describe colors of objects, and so require the specification of a reference 'white light' color. Illuminant D65 is a standard indirect daylight, Illuminant D50 is close to direct sunlight, and Illuminant A is the light from a standard incandescent bulb. Other standard CIE illuminants supported are B, C, E and D55. RGB colour spaces are defined relative to a particular reference white, and can be only approximately translated to other reference whites. The von Kries chromatic adaptation algorithm is used for this. Prior to R 3.6, color conversions involving color spaces created with make.rgb were carried out assuming a D65 illuminant, irrespective of the actual illuminant used in the creation of the color space. This affected the built-in "CIE RGB" color space.

The RGB color spaces are specific to a particular class of display. An RGB space cannot represent all colors, and the clip option controls what is done to out-of-range colors.

For the named color spaces color must be a matrix of values in the from color space: in particular opaque colors.

Value

A 3-column matrix whose rows specify the colors.

References

For all the conversion equations http://www.brucelindbloom.com/.

For the white points https://web.archive.org/web/20190613001950/http://efg2.com/Lab/Graphics/Colors/Chromaticity.htm.

See Also

col2rgb and colors for ways to specify colors in graphics.

make.rgb for specifying other colour spaces.

Examples

```
## The displayable colors from four planes of Lab space
ab <- expand.grid(a = (-10:15)*10,
                  b = (-15:10)*10)
require(graphics); require(stats) # for na.omit
par(mfrow = c(2, 2), mar = .1+c(3, 3, 3, .5), mgp = c(2,  .8,  0))

Lab <- cbind(L = 20, ab)
srgb <- convertColor(Lab, from = "Lab", to = "sRGB", clip = NA)
clipped <- attr(na.omit(srgb), "na.action")
srgb[clipped, ] <- 0
cols <- rgb(srgb[, 1], srgb[, 2], srgb[, 3])
image((-10:15)*10, (-15:10)*10, matrix(1:(26*26), ncol = 26), col = cols,
  xlab = "a", ylab = "b", main = "Lab: L=20")

Lab <- cbind(L = 40, ab)
```

```
srgb <- convertColor(Lab, from = "Lab", to = "sRGB", clip = NA)
clipped <- attr(na.omit(srgb), "na.action")
srgb[clipped, ] <- 0
cols <- rgb(srgb[, 1], srgb[, 2], srgb[, 3])
image((-10:15)*10, (-15:10)*10, matrix(1:(26*26), ncol = 26), col = cols,
  xlab = "a", ylab = "b", main = "Lab: L=40")

Lab <- cbind(L = 60, ab)
srgb <- convertColor(Lab, from = "Lab", to = "sRGB", clip = NA)
clipped <- attr(na.omit(srgb), "na.action")
srgb[clipped, ] <- 0
cols <- rgb(srgb[, 1], srgb[, 2], srgb[, 3])
image((-10:15)*10, (-15:10)*10, matrix(1:(26*26), ncol = 26), col = cols,
  xlab = "a", ylab = "b", main = "Lab: L=60")

Lab <- cbind(L = 80, ab)
srgb <- convertColor(Lab, from = "Lab", to = "sRGB", clip = NA)
clipped <- attr(na.omit(srgb), "na.action")
srgb[clipped, ] <- 0
cols <- rgb(srgb[, 1], srgb[, 2], srgb[, 3])
image((-10:15)*10, (-15:10)*10, matrix(1:(26*26), ncol = 26), col = cols,
  xlab = "a", ylab = "b", main = "Lab: L=80")

cols <- t(col2rgb(palette())); rownames(cols) <- palette(); cols
zapsmall(lab <- convertColor(cols, from = "sRGB", to = "Lab", scale.in = 255))
stopifnot(all.equal(cols, # converting back.. getting the original:
   round(convertColor(lab, from = "Lab", to = "sRGB", scale.out = 255)),
                   check.attributes = FALSE))
```

`densCols` *Colors for Smooth Density Plots*

Description

`densCols` produces a vector containing colors which encode the local densities at each point in a scatterplot.

Usage

```
densCols(x, y = NULL, nbin = 128, bandwidth,
         colramp = colorRampPalette(blues9[-(1:3)]))
blues9
```

Arguments

x, y	the x and y arguments provide the x and y coordinates of the points. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details. If supplied separately, they must be of the same length.

nbin	numeric vector of length one (for both directions) or two (for x and y separately) specifying the number of equally spaced grid points for the density estimation; directly used as gridsize in bkde2D().
bandwidth	numeric vector (length 1 or 2) of smoothing bandwidth(s). If missing, a more or less useful default is used. bandwidth is subsequently passed to function bkde2D.
colramp	function accepting an integer n as an argument and returning n colors.

Details

densCols computes and returns the set of colors that will be used in plotting, calling bkde2D(*, bandwidth, gridsize = nbin, ..) from package **KernSmooth**.

blues9 is a set of 9 color shades of blue used as the default in plotting.

Value

densCols returns a vector of length nrow(x) that contains colors to be used in a subsequent scatterplot. Each color represents the local density around the corresponding point.

Author(s)

Florian Hahne at FHCRC, originally

See Also

bkde2D from package **KernSmooth**; further, smoothScatter() (package **graphics**) which builds on the same computations as densCols.

Examples

```
x1  <- matrix(rnorm(1e3), ncol = 2)
x2  <- matrix(rnorm(1e3, mean = 3, sd = 1.5), ncol = 2)
x   <- rbind(x1, x2)

dcols <- densCols(x)
graphics::plot(x, col = dcols, pch = 20, main = "n = 1000")
```

---

dev *Control Multiple Devices*

Description

These functions provide control over multiple graphics devices.

Usage

```
dev.cur()
dev.list()
dev.next(which = dev.cur())
dev.prev(which = dev.cur())
dev.off(which = dev.cur())
dev.set(which = dev.next())
dev.new(..., noRStudioGD = FALSE)
graphics.off()
```

Arguments

`which`	An integer specifying a device number.
`...`	arguments to be passed to the device selected.
`noRStudioGD`	Do not use the RStudio graphics device even if specified as the default device: it does not accept arguments such as `width` and `height`.

Details

Only one device is the 'active' device: this is the device in which all graphics operations occur. There is a `"null device"` which is always open but is really a placeholder: any attempt to use it will open a new device specified by `getOption("device")`.

Devices are associated with a name (e.g., `"X11"` or `"postscript"`) and a number in the range 1 to 63; the `"null device"` is always device 1. Once a device has been opened the null device is not considered as a possible active device. There is a list of open devices, and this is considered as a circular list not including the null device. `dev.next` and `dev.prev` select the next open device in the appropriate direction, unless no device is open.

`dev.off` shuts down the specified (by default the current) device. If the current device is shut down and any other devices are open, the next open device is made current. It is an error to attempt to shut down device 1. `graphics.off()` shuts down all open graphics devices. Normal termination of a session runs the internal equivalent of `graphics.off()`.

`dev.set` makes the specified device the active device. If there is no device with that number, it is equivalent to `dev.next`. If `which = 1` it opens a new device and selects that.

`dev.new` opens a new device. Normally R will open a new device automatically when needed, but this enables you to open further devices in a platform-independent way. (For which device is used see `getOption("device")`.) Note that care is needed with file-based devices such as `pdf` and `postscript` and in that case file names such as '`Rplots.pdf`', '`Rplots1.pdf`', ..., '`Rplots999.pdf`' are tried in turn. Only named arguments are passed to the device, and then only if they match the argument list of the device. Even so, care is needed with the interpretation of e.g. `width`, and for the standard bitmap devices `units = "in"`, `res = 72` is forced if neither is supplied but both `width` and `height` are.

Value

`dev.cur` returns a length-one named integer vector giving the number and name of the active device, or 1, the null device, if none is active.

`dev.list` returns the numbers of all open devices, except device 1, the null device. This is a numeric vector with a `names` attribute giving the device names, or `NULL` is there is no open device.

`dev.next` and `dev.prev` return the number and name of the next / previous device in the list of devices. This will be the null device if and only if there are no open devices.

`dev.off` returns the number and name of the new active device (after the specified device has been shut down).

`dev.set` returns the number and name of the new active device.

`dev.new` returns the return value of the device opened, usually invisible `NULL`.

See Also

`Devices`, such as `postscript`, etc.

`layout` and its links for setting up plotting regions on the current device.

Examples

```
## Not run: ## Unix-specific example
x11()
plot(1:10)
x11()
plot(rnorm(10))
dev.set(dev.prev())
abline(0, 1) # through the 1:10 points
dev.set(dev.next())
abline(h = 0, col = "gray") # for the residual plot
dev.set(dev.prev())
dev.off(); dev.off() #- close the two X devices

## End(Not run)
```

---

dev.capabilities *Query Capabilities of the Current Graphics Device*

Description

Query the capabilities of the current graphics device.

Usage

```
dev.capabilities(what = NULL)
```

Arguments

`what`	a character vector partially matching the names of the components listed in section 'Value', or `NULL` which lists all available capabilities.

Details

The capabilities have to be specified by the author of the graphics device, unless they can be deduced from missing hooks. Thus they will often by returned as NA, and may reflect the maximal capabilities of the underlying device where several output formats are supported by one device.

Most recent devices support semi-transparent colours provided the graphics format does (which PostScript does not). On the other hand, relatively few graphics formats support (fully or semi-) transparent backgrounds: generally the latter is found only in PDF and PNG plots.

Value

A named list with some or all of the following components, any of which may take value NA:

`semiTransparency`
: logical: Does the device support semi-transparent colours?

`transparentBackground`
: character: Does the device support (semi)-transparent backgrounds? Possible values are `"no"`, `"fully"` (only full transparency) and `"semi"` (semi-transparent background colours are supported).

`rasterImage`
: character: To what extent does the device support raster images as used by `rasterImage` and `grid.raster`? Possible values `"no"`, `"yes"` and `"non-missing"` (support only for arrays without any missing values).

`capture`
: logical: Does the current device support raster capture as used by `grid.cap`?

`locator`
: logical: Does the current device support `locator` and `identify`?

`events`
: character: Which events can be generated on this device? Currently this will be a subset of `c("MouseDown", "MouseMove", "MouseUp", "Keybd")`, but other events may be supported in the future.

`patterns`
: character: Does the device support pattern fills? One or more of `c("LinearGradient", "RadialGradient", "TilingPattern")` May also be `FALSE`.

`clippingPaths`
: logical: Does the device support clipping paths?

`masks`
: character: Does the device support masks? One or more of `c("alpha", "luminance")` May also be `FALSE`.

`compositing`
: character: Does the device support compositing operators? There are many possible operators and devices may support any subset. For example the `pdf` device supports a set of "blend modes" whereas Cairo-based devices support Porter-Duff operators as well. May also be `FALSE`.

`transformations`
: logical: Does the devices support affine transformations ?

`paths`
: logical: Does the device support stroking and filling paths?

`glyphs`
: logical: Does the device support rendering glyphs?

See Also

See `getGraphicsEvent` for details on interactive events.

Examples

```
dev.capabilities()
```

---

dev.capture *Capture device output as a raster image*

Description

`dev.capture` captures the current contents of a graphics device as a raster (bitmap) image.

Usage

```
dev.capture(native = FALSE)
```

Arguments

`native`	Logical. If `FALSE` the result is a matrix of R color names, if `TRUE` the output is returned as a `nativeRaster` object which is more efficient for plotting, but not portable.

Details

Not all devices support capture of the output as raster bitmaps. Typically, only image-based devices do and even not all of them.

Value

`NULL` if the device does not support capture, otherwise a matrix of color names (for `native = FALSE`) or a `nativeRaster` object (for `native = TRUE`).

---

dev.flush *Hold or Flush Output on an On-Screen Graphics Device*

Description

This gives a way to hold/flush output on certain on-screen devices, and is ignored by other devices.

Usage

```
dev.hold(level = 1L)
dev.flush(level = 1L)
```

Arguments

`level`	Integer >= 0. The amount by which to change the hold level. Negative values will be silently replaced by zero.

Details

Devices which implement this maintain a stack of hold levels: calling `dev.hold` increases the level and `dev.flush` decreases it. Calling `dev.hold` when the hold level is zero increases the hold level and inhibits graphics display. When calling `dev.flush` clears all pending holds the screen display is refreshed and normal operation is resumed.

This is implemented for the cairo-based `X11` types with buffering. When the hold level is positive the 'watch' cursor is set on the device's window.

It is available on the `quartz` device on macOS.

This is implemented for the `windows` device with buffering selected (the default). When the hold level is positive the 'busy' cursor is set on the device's window.

Value

The current level after the change, invisibly. This is `0` on devices where hold levels are not supported.

`dev.interactive` *Is the Current Graphics Device Interactive?*

Description

Test if the current graphics device (or that which would be opened) is interactive.

Usage

```
dev.interactive(orNone = FALSE)

deviceIsInteractive(name = NULL)
```

Arguments

`orNone`	logical; if `TRUE`, the function also returns `TRUE` when `.Device == "null device"` and `getOption("device")` is among the known interactive devices.
`name`	one or more device names as a character vector, or `NULL` to give the existing list.

Details

The `X11` (Unix), `windows` (Windows) and `quartz` (macOS, on-screen types only) are regarded as interactive, together with `JavaGD` (from the package of the same name) and `CairoWin` and `CairoX11` (from package **Cairo**). Packages can add their devices to the list by calling `deviceIsInteractive`.

Value

`dev.interactive()` returns a logical, `TRUE` if and only if an interactive (screen) device is in use.

`deviceIsInteractive` returns the updated list of known interactive devices, invisibly unless `name = NULL`.

See Also

Devices for the available devices on your platform.

Examples

```
dev.interactive()
print(deviceIsInteractive(NULL))
```

`dev.size` *Find Size of Device Surface*

Description

Find the dimensions of the device surface of the current device.

Usage

```
dev.size(units = c("in", "cm", "px"))
```

Arguments

`units`	the units in which to return the value – inches, cm, or pixels (device units).

Value

A two-element numeric vector giving width and height of the current device; a new device is opened if there is none, similarly to `dev.new()`.

See Also

The size information in inches can be obtained by `par("din")`, but this provides a way to access it independent of the graphics sub-system in use. Note that `par("din")` is only updated when a new plot is started, whereas `dev.size` tracks the size as an on-screen device is resized.

Examples

```
dev.size("cm")
```

dev2 *Copy Graphics Between Multiple Devices*

Description

`dev.copy` copies the graphics contents of the current device to the device specified by `which` or to a new device which has been created by the function specified by `device` (it is an error to specify both `which` and `device`). (If recording is off on the current device, there are no contents to copy: this will result in no plot or an empty plot.) The device copied to becomes the current device.

`dev.print` copies the graphics contents of the current device to a new device which has been created by the function specified by `device` and then shuts the new device.

`dev.copy2eps` is similar to `dev.print` but produces an EPSF output file in portrait orientation (`horizontal = FALSE`). `dev.copy2pdf` is the analogue for PDF output.

`dev.control` allows the user to control the recording of graphics operations in a device. If `displaylist` is `"inhibit"` (`"enable"`) then recording is turned off (on). It is only safe to change this at the beginning of a plot (just before or just after a new page). Initially recording is on for screen devices, and off for print devices.

Usage

```
dev.copy(device, ..., which = dev.next())
dev.print(device = postscript, ...)
dev.copy2eps(...)
dev.copy2pdf(..., out.type = "pdf")
dev.control(displaylist = c("inhibit", "enable"))
```

Arguments

`device`	A device function (e.g., `x11`, `postscript`, ...)
`...`	Arguments to the `device` function above: for `dev.copy2eps` arguments to `postscript` and for `dev.copy2pdf`, arguments to `pdf`. For `dev.print`, this includes `which` and by default any `postscript` arguments.
`which`	A device number specifying the device to copy to.
`out.type`	The name of the output device: can be `"pdf"`, or `"quartz"` (some macOS builds) or `"cairo"` (Windows and some Unix-alikes, see `cairo_pdf`).
`displaylist`	A character string: the only valid values are `"inhibit"` and `"enable"`.

Details

Note that these functions copy the *device region* and not a plot: the background colour of the device surface is part of what is copied. Most screen devices default to a transparent background, which is probably not what is needed when copying to a device such as `png`.

For `dev.copy2eps` and `dev.copy2pdf`, `width` and `height` are taken from the current device unless otherwise specified. If just one of `width` and `height` is specified, the other is adjusted to preserve

the aspect ratio of the device being copied. The default file name is `Rplot.eps` or `Rplot.pdf`, and can be overridden by specifying a `file` argument.

Copying to devices such as `pdf` and `postscript` which need font families pre-specified needs extra care – R is unaware of which families were used in a plot and so they will need to manually specified by the `fonts` argument passed as part of `...`. Similarly, if the device to be copied from was opened with a `family` argument, a suitable `family` argument will need to be included in `...`.

The default for `dev.print` is to produce and print a postscript copy. This will not work unless `options("printcmd")` is set suitably and you have a PostScript printing system: see `postscript` for how to set this up. Windows users may prefer to use `dev.print(win.print)`.

`dev.print` is most useful for producing a postscript print (its default) when the following applies. Unless `file` is specified, the plot will be printed. Unless `width`, `height` and `pointsize` are specified the plot dimensions will be taken from the current device, shrunk if necessary to fit on the paper. (`pointsize` is rescaled if the plot is shrunk.) If `horizontal` is not specified and the plot can be printed at full size by switching its value this is done instead of shrinking the plot region.

If `dev.print` is used with a specified `device` (even `postscript`) it sets the width and height in the same way as `dev.copy2eps`. This will not be appropriate unless the device specifies dimensions in inches, in particular not for `png`, `jpeg`, `tiff` and `bmp` unless `units = "inches"` is specified.

Value

`dev.copy` returns the name and number of the device which has been copied to.

`dev.print`, `dev.copy2eps` and `dev.copy2pdf` return the name and number of the device which has been copied from.

Note

Most devices (including all screen devices) have a display list which records all of the graphics operations that occur in the device. `dev.copy` copies graphics contents by copying the display list from one device to another device. Also, automatic redrawing of graphics contents following the resizing of a device depends on the contents of the display list.

After the command `dev.control("inhibit")`, graphics operations are not recorded in the display list so that `dev.copy` and `dev.print` will not copy anything and the contents of a device will not be redrawn automatically if the device is resized.

The recording of graphics operations is relatively expensive in terms of memory so the command `dev.control("inhibit")` can be useful if memory usage is an issue.

See Also

`dev.cur` and other `dev.xxx` functions.

Examples

```
## Not run:
x11() # on a Unix-alike
plot(rnorm(10), main = "Plot 1")
dev.copy(device = x11)
mtext("Copy 1", 3)
dev.print(width = 6, height = 6, horizontal = FALSE) # prints it
```

```
dev.off(dev.prev())
dev.off()

## End(Not run)
```

---

dev2bitmap *Graphics Device for Bitmap Files via Ghostscript*

Description

bitmap generates a graphics file. dev2bitmap copies the current graphics device to a file in a graphics format.

Usage

```
bitmap(file, type = "png16m", height = 7, width = 7, res = 72,
       units = "in", pointsize, taa = NA, gaa = NA, ...)

dev2bitmap(file, type = "png16m", height = 7, width = 7, res = 72,
           units = "in", pointsize, ...,
           method = c("postscript", "pdf"), taa = NA, gaa = NA)
```

Arguments

file	The output file name, with an appropriate extension.
type	The type of bitmap.
width, height	Dimensions of the display region.
res	Resolution, in dots per inch.
units	The units in which height and width are given. Can be in (inches), px (pixels), cm or mm.
pointsize	The pointsize to be used for text: defaults to something reasonable given the width and height
...	Other parameters passed to postscript or pdf.
method	Should the plot be done by postscript or pdf?
taa, gaa	Number of bits of antialiasing for text and for graphics respectively. Usually 4 (for best effect) or 2. Not supported on all types.

Details

dev2bitmap works by copying the current device to a postscript or pdf device, and post-processing the output file using ghostscript. bitmap works in the same way using a postscript device and post-processing the output as 'printing'.

You will need ghostscript: the full path to the executable can be set by the environment variable R_GSCMD. If this is unset, a GhostScript executable will be looked for by name on your path: on

a Unix alike "gs" is used, and on Windows the setting of the environment variable GSC is used, otherwise commands "gswi64c.exe" then "gswin32c.exe" are tried.

The types available will depend on the version of ghostscript, but are likely to include "jpeg", "jpegcmyk", "jpeggray", "tiffcrle", "tiffg3", "tiffg32d", "tiffg4", "tiffgray", "tifflzw", "tiffpack", "tiff12nc", "tiff24nc", "tiff32nc" "png16", "png16m", "png256", "png48", "pngmono", "pnggray", "pngalpha", "bmp16", "bmp16m" "bmp256", "bmp32b", "bmpgray", "bmpmono".

The default type, "png16m", supports 24-bit colour and anti-aliasing. Type "png256" uses a palette of 256 colours and could give a more compact representation. Monochrome graphs can use "pngmono", or "pnggray" if anti-aliasing is desired. Plots with a transparent background and varying degrees of transparency should use "pngalpha".

Note that for a colour TIFF image you probably want "tiff24nc", which is 8-bit per channel RGB (the most common TIFF format). None of the listed TIFF types support transparency. "tiff32nc" uses 8-bit per channel CMYK, which printers might require.

For formats which contain a single image, a file specification like Rplots%03d.png can be used: this is interpreted by Ghostscript.

For dev2bitmap if just one of width and height is specified, the other is chosen to preserve the aspect ratio of the device being copied. The main reason to prefer method = "pdf" over the default would be to allow semi-transparent colours to be used.

For graphics parameters such as "cra" that need to work in pixels, the default resolution of 72dpi is always used.

On Windows only, paths for file and R_GSCMD which contain spaces are mapped to short names *via* shortPathName.

Value

None.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual. These devices follow the underlying device, so when viewed at the stated res:

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is (for the standard Ghostscript setup) URW Nimbus Sans.
- Line widths are as a multiple of 1/96 inch, with no minimum.
- Circle of any radius are allowed.
- Colours are interpreted by the viewing/printing application.

Note

On Windows, Use of bitmap will leave a temporary file (with file name starting Rbit).

Although using type = "pdfwrite" will work for simple plots, it is not recommended. Either use pdf to produce PDF directly, or call ps2pdf -dAutoRotatePages=/None on the output of postscript: that command is optimized to do the conversion to PDF in ways that these functions are not.

See Also

savePlot, which for windows and X11(type = "cairo") provides a simple way to record a PNG record of the current plot.

postscript, pdf, png, jpeg, tiff and bmp.

To display an array of data, see image.

---

devAskNewPage *Prompt before New Page*

Description

This function can be used to control (for the current device) whether the user is prompted before starting a new page of output.

Usage

```
devAskNewPage(ask = NULL)
```

Arguments

ask	NULL or a logical value. If TRUE, the user will in future be prompted before a new page of output is started.

Details

If the current device is the null device, this will open a graphics device.

The default argument just returns the current setting and does not change it.

The default value when a device is opened is taken from the setting of options("device.ask.default").

The precise circumstances when the user will be asked to confirm a new page depend on the graphics subsystem. Obviously this needs to be an interactive session. In addition 'recording' needs to be in operation, so only when the display list is enabled (see dev.control) which it usually is only on a screen device.

Value

The current prompt setting *before* any new setting is applied. Invisibly if ask is logical.

See Also

plot.new, grid.newpage

`Devices` *List of Graphical Devices*

Description

The following graphics devices are currently available:

`windows`: On Windows only, the graphics device for Windows (on screen, to printer and to Windows metafile).

`pdf`: Write PDF graphics commands to a file.

`postscript`: Writes PostScript graphics commands to a file.

`xfig`: Device for XFig graphics file format. (Of historical interest only, deprecated in R 4.4.0.)

`bitmap`: bitmap pseudo-device via `Ghostscript` (if available).

`pictex`: Writes TeX/PicTeX graphics commands to a file (of historical interest only, deprecated in R 4.4.0).

The following devices will be functional if R was compiled to use them (they exist but will return with a warning on other systems):

`cairo_pdf`, `cairo_ps`: PDF and PostScript devices based on cairo graphics.

`svg`: SVG device based on cairo graphics

`png`: PNG bitmap device

`jpeg`: JPEG bitmap device

`bmp`: BMP bitmap device

`tiff`: TIFF bitmap device

On Unix-alikes (including macOS) only:

`X11`: The graphics device for the X11 windowing system

`quartz`: The graphics device for the macOS native Quartz 2d graphics system. (This is only functional on macOS where it can be used from the `R.app` GUI and from the command line: but it will display on the local screen even for a remote session.)

Details

If no device is open, calling any high-level graphics function will cause a device to be opened. Which device is determined by `options("device")` which is initially set as the most appropriate for each platform: a screen device for most interactive use and `pdf` (or the setting of `R_DEFAULT_DEVICE`) otherwise. The exception is interactive use under Unix if no screen device is known to be available, when `pdf()` is used.

It is possible for an R package (or an R front-end such as RStudio) to provide further graphics devices and several packages on CRAN do so. These include devices outputting SVG (**svglite** and PGF/TiKZ (**tikzDevice**, TeX-based graphics, see `https://pgf.sourceforge.net/`).

See Also

The individual help files for further information on any of the devices listed here;

on Windows: windows.options,

on a Unix-alike: X11.options, quartz.options,

ps.options and pdf.options for how to customize devices.

dev.interactive, dev.cur, dev.print, graphics.off, image, dev2bitmap.

On Unix-alikes only:
capabilities to see if X11, jpeg, png, tiff, quartz and the cairo-based devices are available.

Examples

```
## Not run:
## open the default screen device on this platform if no device is
## open
if(dev.cur() == 1) dev.new()

## End(Not run)
```

---

embedFonts *Embed Fonts in PostScript and PDF*

Description

Runs Ghostscript to process a PDF or PostScript file and embed all fonts in the file.

Use embedGlyphs() if you have drawn typeset glyphs (see glyphInfo), which is only relevant for PDF files.

Usage

```
embedFonts(file, format, outfile = file,
           fontpaths = character(), options = character())

embedGlyphs(file, glyphInfo, outfile = file, options = character())
```

Arguments

file	a character string giving the name of the original file.
format	the format for the new file (with fonts embedded) given as the name of a Ghostscript output device. If not specified, it is guessed from the suffix of file.
outfile	the name of the new file (with fonts embedded).
fontpaths	a character vector giving directories that Ghostscript will search for fonts.
options	a character vector containing further options to Ghostscript.
glyphInfo	typeset glyph information produced by glyphInfo(), or a list of the same.

Details

This function is not necessary if you just use the standard default fonts for PostScript and PDF output.

If you use a special font, this function is useful for embedding that font in your PostScript or PDF document so that it can be shared with others without them having to install your special font (provided the font licence allows this).

If the special font is not installed for Ghostscript, you will need to tell Ghostscript where the font is, using something like `options="-sFONTPATH=path/to/font"`.

You will need `ghostscript`: the full path to the executable can be set by the environment variable `R_GSCMD`. If this is unset, a GhostScript executable will be looked for by name on your path: on a Unix alike `"gs"` is used, and on Windows the setting of the environment variable `GSC` is used, otherwise commands `"gswi64c.exe"` then `"gswin32c.exe"` are tried.

The `format` is by default `"ps2write"`, when the original file has a `.ps` or `.eps` suffix, or `"pdfwrite"` when the original file has a `.pdf` suffix. For versions of Ghostscript before 9.10, `format = "pswrite"` or `format = "epswrite"` can be used: as from 9.14 `format = "eps2write"` is also available. If an invalid device is given, the error message will list the available devices.

Note that Ghostscript may do font substitution, so the font embedded may differ from that specified in the original file.

Some other options which can be useful (see your Ghostscript documentation) are '`-dMaxSubsetPct=100`', '`-dSubsetFonts=true`' and '`-dEmbedAllFonts=true`'.

`embedGlyphs()` is recommended for `pdf()` files that contain typeset glyphs (see `glyphInfo`), but it will only work for TrueType fonts.

Value

The shell command used to invoke Ghostscript is returned invisibly. This may be useful for debugging purposes as you can run the command by hand in a shell to look for problems.

See Also

`postscriptFonts`, `Devices`.

Paul Murrell and Brian Ripley (2006). "Non-standard fonts in PostScript and PDF graphics." *R News*, **6**(2), 41–47. https://www.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

`extendrange` *Extend a Numerical Range by a Small Percentage*

Description

Extends a numerical range by a small percentage, i.e., fraction, *on both sides*.

Usage

```
extendrange(x, r = range(x, na.rm = TRUE), f = 0.05)
```

Arguments

x	numeric vector; not used if r is specified.
r	numeric vector of length 2; defaults to the range of x.
f	positive number(s) specifying the fraction by which the range should be extended. If longer than one, f[1] is used on the left, and f[2] on the right.

Value

A numeric vector of length 2, r + c(-f1, f2) * diff(r), where f1 is f[1] and f2 is f[2] or f if it is of length one.

See Also

range; pretty which can be considered a sophisticated extension of extendrange.

Examples

```
x <- 1:5
(r <- range(x))         # 1    5
extendrange(x)          # 0.8  5.2
extendrange(x, f= 0.01) # 0.96 5.04

## extend more to the right:
extendrange(x, f=c(.01,.03)) # 0.96 5.12

## Use 'r' if you have it already:
stopifnot(identical(extendrange(r = r),
                    extendrange(x)))
```

---

getGraphicsEvent *Wait for a mouse or keyboard event from a graphics window*

Description

This function waits for input from a graphics window in the form of a mouse or keyboard event.

Usage

```
getGraphicsEvent(prompt = "Waiting for input",
                 onMouseDown = NULL, onMouseMove = NULL,
                 onMouseUp = NULL, onKeybd = NULL,
                 onIdle = NULL,
                 consolePrompt = prompt)
setGraphicsEventHandlers(which = dev.cur(), ...)
getGraphicsEventEnv(which = dev.cur())
setGraphicsEventEnv(which = dev.cur(), env)
```

Arguments

prompt	prompt to be displayed to the user in the graphics window
onMouseDown	a function to respond to mouse clicks
onMouseMove	a function to respond to mouse movement
onMouseUp	a function to respond to mouse button releases
onKeybd	a function to respond to key presses
onIdle	a function to call when no events are pending
consolePrompt	prompt to be displayed to the user in the console
which	which graphics device does the call apply to?
...	items including handlers to be placed in the event environment
env	an environment to be used as the event environment

Details

These functions allow user input from some graphics devices (currently only the `windows()`, `X11(type = "Xlib")` and `X11(type = "cairo")` screen displays in base R). Event handlers may be installed to respond to events involving the mouse or keyboard.

The functions are related as follows. If any of the first six arguments to `getGraphicsEvent` are given, then it uses those in a call to `setGraphicsEventHandlers` to replace any existing handlers in the current device. This is for compatibility with pre-2.12.0 R versions. The current normal way to set up event handlers is to set them using `setGraphicsEventHandlers` or `setGraphicsEventEnv` on one or more graphics devices, and then use `getGraphicsEvent()` with no arguments to retrieve event data. `getGraphicsEventEnv()` may be used to save the event environment for use later.

The names of the arguments in `getGraphicsEvent` are special. When handling events, the graphics system will look through the event environment for functions named `onMouseDown`, `onMouseMove`, `onMouseUp`, `onKeybd`, and `onIdle`, and use them as event handlers. It will use `prompt` for a label on the graphics device. Two other special names are `which`, which will identify the graphics device, and `result`, where the result of the last event handler will be stored before being returned by `getGraphicsEvent()`.

The mouse event handlers should be functions with header `function(buttons, x, y)`. The coordinates `x` and `y` will be passed to mouse event handlers in device independent coordinates (i.e., the lower left corner of the window is `(0,0)`, the upper right is `(1,1)`). The `buttons` argument will be a vector listing the buttons that are pressed at the time of the event, with 0 for left, 1 for middle, and 2 for right.

The keyboard event handler should be a function with header `function(key)`. A single element character vector will be passed to this handler, corresponding to the key press. Shift and other modifier keys will have been processed, so `shift-a` will be passed as `"A"`. The following special keys may also be passed to the handler:

- Control keys, passed as `"Ctrl-A"`, etc.
- Navigation keys, passed as one of `"Left"`, `"Up"`, `"Right"`, `"Down"`, `"PgUp"`, `"PgDn"`, `"End"`, `"Home"`
- Edit keys, passed as one of `"Ins"`, `"Del"`
- Function keys, passed as one of `"F1"`, `"F2"`, ...

The idle event handler onIdle should be a function with no arguments. If the function is undefined or NULL, then R will typically call a system function which (efficiently) waits for the next event to appear on a file handle. Otherwise, the idle event handler will be called whenever the event queue of the graphics device was found to be empty, i.e. in an infinite loop. This feature is intended to allow animations to respond to user input, and could be CPU-intensive. Currently, onIdle is only implemented for X11() devices.

Note that calling Sys.sleep() is not recommended within an idle handler - Sys.sleep() removes pending graphics events in order to allow users to move, close, or resize windows while it is executing. Events such as mouse and keyboard events occurring during Sys.sleep() are lost, and currently do not trigger the event handlers registered via getGraphicsEvent or setGraphicsEventHandlers.

The event handlers are standard R functions, and will be executed as though called from the event environment.

In an interactive session, events will be processed until

- one of the event handlers returns a non-NULL value which will be returned as the value of getGraphicsEvent, or
- the user interrupts the function from the console.

Value

When run interactively, getGraphicsEvent returns a non-NULL value returned from one of the event handlers. In a non-interactive session, getGraphicsEvent will return NULL immediately. It will also return NULL if the user closes the last window that has graphics handlers.

getGraphicsEventEnv returns the current event environment for the graphics device, or NULL if none has been set.

setGraphicsEventEnv and setGraphicsEventHandlers return the previous event environment for the graphics device.

Author(s)

Duncan Murdoch

Examples

```
# This currently only works on the Windows, X11(type = "Xlib"), and
# X11(type = "cairo") screen devices...
## Not run:
savepar <- par(ask = FALSE)
dragplot <- function(..., xlim = NULL, ylim = NULL, xaxs = "r", yaxs = "r") {
    plot(..., xlim = xlim, ylim = ylim, xaxs = xaxs, yaxs = yaxs)
    startx <- NULL
    starty <- NULL
    prevx <- NULL
    prevy <- NULL
    usr <- NULL

    devset <- function()
        if (dev.cur() != eventEnv$which) dev.set(eventEnv$which)
```

```
    dragmousedown <- function(buttons, x, y) {
        startx <<- x
        starty <<- y
        prevx <<- 0
        prevy <<- 0
        devset()
        usr <<- par("usr")
        eventEnv$onMouseMove <- dragmousemove
        NULL
    }

    dragmousemove <- function(buttons, x, y) {
        devset()
        deltax <- diff(grconvertX(c(startx, x), "ndc", "user"))
        deltay <- diff(grconvertY(c(starty, y), "ndc", "user"))
 if (abs(deltax-prevx) + abs(deltay-prevy) > 0) {
     plot(..., xlim = usr[1:2]-deltax, xaxs = "i",
        ylim = usr[3:4]-deltay, yaxs = "i")
     prevx <<- deltax
     prevy <<- deltay
 }
        NULL
    }

    mouseup <- function(buttons, x, y) {
     eventEnv$onMouseMove <- NULL
    }

    keydown <- function(key) {
        if (key == "q") return(invisible(1))
        eventEnv$onMouseMove <- NULL
        NULL
    }

    setGraphicsEventHandlers(prompt = "Click and drag, hit q to quit",
                    onMouseDown = dragmousedown,
                    onMouseUp = mouseup,
                    onKeybd = keydown)
    eventEnv <- getGraphicsEventEnv()
}

dragplot(rnorm(1000), rnorm(1000))
getGraphicsEvent()
par(savepar)

## End(Not run)
```

---

glyphInfo *Describe a Set of Typeset Glyphs*

Description

Create an object that contains information about typeset glyphs. This includes glyph identifiers, glyph locations, font and colour information, and metric information.

Usage

```
glyphInfo(id, x, y, font, size, fontList,
          width, height, hAnchor, vAnchor,
          col=NA)

glyphFont(file, index, family, weight, style, PSname=NA)
glyphFontList(...)
glyphAnchor(value, label)
glyphWidth(w, label="width", left="left")
glyphHeight(h, label="height", bottom="bottom")
glyphWidthLeft(w, label)
glyphHeightBottom(h, label)
glyphJust(just, ...)
## S3 method for class 'GlyphJust'
glyphJust(just, ...)
## S3 method for class 'character'
glyphJust(just, ...)
## S3 method for class 'numeric'
glyphJust(just, which=NULL, ...)
```

Arguments

id	Numeric vector of glyph identifiers (index of glyph within font file).
x, y	Numeric locations of glyphs in (big) points (1/72 inches).
font	Integer index into fontList.
size	Numeric size of glyphs (in points).
fontList	List of glyph fonts, as generated by glyphFont().
width	Overall width of glyphs. Can be a single numeric value, but can also be the result from a call to glyphWidth().
height	Overall height of glyphs. Can be a single numeric value, but can also be the result from a call to glyphHeight().
hAnchor	Horizontal anchors for justifying glyphs relative to the (x, y) location. Can be a single numeric value (against which to "left" justify), but can also be result from a call to glyphAnchor().
vAnchor	Vertical anchors for justifying glyphs relative to the (x, y) location. Can be a single numeric value (against which to "bottom" justify), but can also be result from a call to glyphAnchor().
col	An R colour value for each glyph. Can be NA.
file	Character path to font file.
index	Numeric index of font within font file.

family	Character name of font family.
weight	Numeric weight of glyphs (400 is normal, 700 is bold).
style	Character style of glyphs ("normal", "italic", or "oblique").
PSname	The PostScript name for each font. Can be NA.
value, w, h	A numeric value.
label, left, bottom	A character value.
just	A justification value. Either a character value like "left" or a numeric value like 0.
which	When x is numeric, a character value identifying which width metric the numeric value is relative to.
...	Further arguments passed to other methods.

Details

Multiple anchors can be specified so as to allow different character-based justifications of the glyphs relative to the (x, y) location. Horizontal anchors with labels "left", "centre", and "right" are required. It is possible to specify a single numeric hAnchor, which is treated as the "left" anchor, or a single anchor with label "left", in which case the other required anchors will be calculated based on the required width of the glyphs (see below). Vertical anchors with labels "bottom", "centre", and "top" are required. It is possible to specify a single numeric vAnchor, which is treated as the "bottom" anchor, or a single anchor with label "bottom", in which case the other required anchors will be calculated based on the required height of the glyphs (see below). An example of a non-required anchor is a vertical anchor with the label "baseline" so that the glyphs can be placed with their baseline at the y location.

Multiple widths and heights can be specified so as to allow different numeric-based justifications of the glyphs relative to the (x, y) location, e.g., 0 for left-justification and 1 for right-justification, but with any value in between or even outside those limits also possible. A width with label "width", relative to the "left" horizontal anchor, is required, but if a single numeric value is given, that is assumed to be the required width. A height with label "height", relative to the "bottom" vertical anchor is required, but if a single numeric value is given, that is assumed to be the required height. An example of a non-required width is a "tight" width that is relative to a "left-bearing" horizontal anchor, so that the glyphs can be justified relative to a bounding box around the glyph ink, rather than a bounding box that includes left and right bearings.

glyphWidthLeft() and glyphWidthHeight() provide an API for code that needs to access the relevant anchors for width and height metrics.

Value

The result from glyphInfo() is an "RGlyphInfo" object, essentially a data frame with each row containing id, location, font, and colour for a glyph. The metric information (widths and anchors) are stored as attributes of the data frame.

glyphAnchor(), glyphWidth(), and glyphHeight() return values that can be used to specify width, height, hAnchor, and vAnchor values to glyphInfo().

Warning

Any glyph with NA in any of id, x, y, or size is silently dropped.

---

gray *Gray Level Specification*

Description

Create a vector of colors from a vector of gray levels.

Usage

```
gray(level, alpha)
grey(level, alpha)
```

Arguments

level	a vector of desired gray levels between 0 and 1; zero indicates "black" and one indicates "white".
alpha	the opacity, if specified.

Details

The values returned by gray can be used with a col= specification in graphics functions or in par.

grey is an alias for gray.

Value

A vector of colors of the same length as level.

See Also

rainbow, hsv, hcl, rgb.

Examples

```
gray(0:8 / 8)
```

`gray.colors` *Gray Color Palette*

Description

Create a vector of `n` gamma-corrected gray colors.

Usage

```
gray.colors(n, start = 0.3, end = 0.9, gamma = 2.2, alpha, rev = FALSE)
grey.colors(n, start = 0.3, end = 0.9, gamma = 2.2, alpha, rev = FALSE)
```

Arguments

`n`	the number of gray colors (≥ 1) to be in the palette.
`start`	starting gray level in the palette (should be between `0` and `1` where zero indicates `"black"` and one indicates `"white"`).
`end`	ending gray level in the palette.
`gamma`	the gamma correction.
`alpha`	the opacity, if specified.
`rev`	logical indicating whether the ordering of the colors should be reversed.

Details

The function `gray.colors` chooses a series of `n` gamma-corrected gray levels between `start` and `end`: `seq(start^gamma, end^gamma, length = n)^(1/gamma)`. The returned palette contains the corresponding gray colors. This palette is used in `barplot.default`.

`grey.colors` is an alias for `gray.colors`.

Value

A vector of `n` gray colors.

See Also

`gray`, `rainbow`, `palette`.

Examples

```
require(graphics)

pie(rep(1, 12), col = gray.colors(12))
barplot(1:12, col = gray.colors(12))
```

`grSoftVersion` *Report Versions of Graphics Software*

Description

Report versions of third-party graphics software available on the current platform for R's graphics.

Usage

```
grSoftVersion()
```

Value

A named character vector containing at least the elements

`cairo`	the version of cairographics in use, or `""` if cairographics is not available.
`cairoFT`	the FreeType/FontConfig versions if cairographics is using those libraries directly (not *via* pango); otherwise, `""`. Earlier versions of R returned `"yes"` rather than the versions. The FontConfig version is determined when R is built.
`pango`	the version of pango in use, or `""` if pango is not available.

It may also contain the versions of third-party software used by the standard (on Windows), or X11-based (on Unix-alikes) bitmap devices:

`libpng`	the version of `libpng` in use, or `""` if not available.
`jpeg`	the version of the JPEG headers used for compilation, or `""` if JPEG support was not compiled in.
`libtiff`	the version of `libtiff` in use, or `""` if not available.

It is conceivable but unlikely that the cairo-based bitmap devices will use different versions linked *via* cairographics, especially `png(type = "cairo-png")`.

On macOS, if available, the Quartz-based devices will use the system versions of these libraries rather than those reported here.

Unless otherwise stated the reported version is that of the (possibly dynamically-linked) library in use at runtime.

Note that `libjpeg-turbo` as used on some Linux distributions reports its version as `"6.2"`, the IJG version from which it forked.

See Also

`extSoftVersion` for versions of non-graphics software.

Examples

```
grSoftVersion()
```

hcl *HCL Color Specification*

Description

Create a vector of colors from vectors specifying hue, chroma and luminance.

Usage

```
hcl(h = 0, c = 35, l = 85, alpha, fixup = TRUE)
```

Arguments

h	The hue of the color specified as an angle in the range [0,360]. 0 yields red, 120 yields green 240 yields blue, etc.
c	The chroma of the color. The upper bound for chroma depends on hue and luminance.
l	A value in the range [0,100] giving the luminance of the colour. For a given combination of hue and chroma, only a subset of this range is possible.
alpha	numeric vector of values in the range `[0,1]` for alpha transparency channel (0 means transparent and 1 means opaque).
fixup	a logical value which indicates whether the resulting RGB values should be corrected to ensure that a real color results. if `fixup` is `FALSE` RGB components lying outside the range [0,1] will result in an `NA` value.

Details

This function corresponds to polar coordinates in the CIE-LUV color space. Steps of equal size in this space correspond to approximately equal perceptual changes in color. Thus, `hcl` can be thought of as a perceptually based version of `hsv`.

The function is primarily intended as a way of computing colors for filling areas in plots where area corresponds to a numerical value (pie charts, bar charts, mosaic plots, histograms, etc). Choosing colors which have equal chroma and luminance provides a way of minimising the irradiation illusion which would otherwise produce a misleading impression of how large the areas are.

The default values of chroma and luminance make it possible to generate a full range of hues and have a relatively pleasant pastel appearance.

The RGB values produced by this function correspond to the sRGB color space used on most PC computer displays. There are other packages which provide more general color space facilities.

Semi-transparent colors (`0 < alpha < 1`) are supported only on some devices: see `rgb`.

Value

A vector of character strings which can be used as color specifications by R graphics functions.

Missing or infinite values of any of `h`, `c`, `l` result in `NA`: such values of `alpha` are taken as 1 (opaque).

Note

At present there is no guarantee that the colours rendered by R graphics devices will correspond to their sRGB description. It is planned to adopt sRGB as the standard R color description in future.

Author(s)

Ross Ihaka

References

Ihaka, R. (2003). Colour for Presentation Graphics, Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, 2003, Technische Universität Wien, Vienna, Austria. https://www.R-project.org/conferences/DSC-2003/.

See Also

hsv, rgb.

Examples

```
require(graphics)

# The Foley and Van Dam PhD Data.
csd <- matrix(c( 4,2,4,6, 4,3,1,4, 4,7,7,1,
                 0,7,3,2, 4,5,3,2, 5,4,2,2,
                 3,1,3,0, 4,4,6,7, 1,10,8,7,
                 1,5,3,2, 1,5,2,1, 4,1,4,3,
                 0,3,0,6, 2,1,5,5), nrow = 4)

csphd <- function(colors)
  barplot(csd, col = colors, ylim = c(0,30),
          names.arg = 72:85, xlab = "Year", ylab = "Students",
          legend.text = c("Winter", "Spring", "Summer", "Fall"),
          main = "Computer Science PhD Graduates", las = 1)

# The Original (Metaphorical) Colors (Ouch!)
csphd(c("blue", "green", "yellow", "orange"))

# A Color Tetrad (Maximal Color Differences)
csphd(hcl(h = c(30, 120, 210, 300)))

# Same, but lighter and less colorful
# Turn off automatic correction to make sure
# that we have defined real colors.
csphd(hcl(h = c(30, 120, 210, 300),
          c = 20, l = 90, fixup = FALSE))

# Analogous Colors
# Good for those with red/green color confusion
csphd(hcl(h = seq(60, 240, by = 60)))
```

```
# Metaphorical Colors
csphd(hcl(h = seq(210, 60, length.out = 4)))

# Cool Colors
csphd(hcl(h = seq(120, 0, length.out = 4) + 150))

# Warm Colors
csphd(hcl(h = seq(120, 0, length.out = 4) - 30))

# Single Color
hist(stats::rnorm(1000), col = hcl(240))

## Exploring the hcl() color space {in its mapping to R's sRGB colors}:
demo(hclColors)
```

`Hershey` *Hershey Vector Fonts in R*

Description

If the `family` graphical parameter (see `par`) has been set to one of the Hershey fonts (see 'Details') Hershey vector fonts are used to render text.

When using the `text` and `contour` functions Hershey fonts may be selected via the `vfont` argument, which is a character vector of length 2 (see 'Details' for valid values). This allows Cyrillic to be selected, which is not available via the font families.

Usage

```
Hershey
```

Details

The Hershey fonts have two advantages:

1. vector fonts describe each character in terms of a set of points; R renders the character by joining up the points with straight lines. This intimate knowledge of the outline of each character means that R can arbitrarily transform the characters, which can mean that the vector fonts look better for rotated text.
2. this implementation was adapted from the GNU libplot library which provides support for non-ASCII and non-English fonts. This means that it is possible, for example, to produce weird plotting symbols and Japanese characters.

Drawback:
You cannot use mathematical expressions (`plotmath`) with Hershey fonts.

The Hershey characters are organised into a set of fonts. A particular font is selected by specifying one of the following font families via `par(family)` and specifying the desired font face (plain, bold, italic, bold-italic) via `par(font)`.

family	faces available
"HersheySerif"	plain, bold, italic, bold-italic
"HersheySans"	plain, bold, italic, bold-italic
"HersheyScript"	plain, bold
"HersheyGothicEnglish"	plain
"HersheyGothicGerman"	plain
"HersheyGothicItalian"	plain
"HersheySymbol"	plain, bold, italic, bold-italic
"HersheySansSymbol"	plain, italic

In the vfont specification for the text and contour functions, the Hershey font is specified by a typeface (e.g., serif or sans serif) and a fontindex or 'style' (e.g., plain or italic). The first element of vfont specifies the typeface and the second element specifies the fontindex. The first table produced by demo(Hershey) shows the character a produced by each of the different fonts.

The available typeface and fontindex values are available as list components of the variable Hershey. The allowed pairs for (typeface, fontindex) are:

serif	plain
serif	italic
serif	bold
serif	bold italic
serif	cyrillic
serif	oblique cyrillic
serif	EUC
sans serif	plain
sans serif	italic
sans serif	bold
sans serif	bold italic
script	plain
script	italic
script	bold
gothic englisho	plain
gothic german	plain
gothic italian	plain
serif symbol	plain
serif symbol	italic
serif symbol	bold
serif symbol	bold italic
sans serif symbol	plain
sans serif symbol	italic

and the indices of these are available as Hershey$allowed.

Escape sequences: The string to be drawn can include escape sequences, which all begin with a '\'. When R encounters a '\', rather than drawing the '\', it treats the subsequent character(s) as a coded description of what to draw.

One useful escape sequence (in the current context) is of the form: '\123'. The three digits following the '\' specify an octal code for a character. For example, the octal code for p is 160 so the strings "p" and "\160" are equivalent. This is useful for producing characters when there is not an appropriate key on your keyboard.

The other useful escape sequences all begin with '\\'. These are described below. Remember that backslashes have to be doubled in R character strings, so they need to be entered with *four* backslashes.

Symbols: an entire string of Greek symbols can be produced by selecting the HersheySymbol or HersheySansSymbol family or the Serif Symbol or Sans Serif Symbol typeface. To allow Greek symbols to be embedded in a string which uses a non-symbol typeface, there are a set of symbol escape sequences of the form '\\ab'. For example, the escape sequence '*a' produces a Greek alpha. The second table in demo(Hershey) shows all of the symbol escape sequences and the symbols that they produce.

ISO Latin-1: further escape sequences of the form '\\ab' are provided for producing ISO Latin-1 characters. Another option is to use the appropriate octal code. The (non-ASCII) ISO Latin-1 characters are in the range 241...377. For example, '\366' produces the character o with an umlaut. The third table in demo(Hershey) shows all of the ISO Latin-1 escape sequences.

These characters can be used directly. (Characters not in Latin-1 are replaced by a dot.)

Several characters are missing, c-cedilla has no cedilla and 'sharp s' ('U+00DF', also known as 'esszett') is rendered as ss.

Special Characters: a set of characters are provided which do not fall into any standard font. These can only be accessed by escape sequence. For example, '\\LI' produces the zodiac sign for Libra, and '\\JU' produces the astronomical sign for Jupiter. The fourth table in demo(Hershey) shows all of the special character escape sequences.

Cyrillic Characters: cyrillic characters are implemented according to the K018-R encoding, and can be used directly in such a locale using the Serif typeface and Cyrillic (or Oblique Cyrillic) fontindex. Alternatively they can be specified via an octal code in the range 300 to 337 for lower case characters or 340 to 377 for upper case characters. The fifth table in demo(Hershey) shows the octal codes for the available Cyrillic characters.

Cyrillic has to be selected via a ("serif", fontindex) pair rather than via a font family.

Japanese Characters: 83 Hiragana, 86 Katakana, and 603 Kanji characters are implemented according to the EUC-JP (Extended Unix Code) encoding. Each character is identified by a unique hexadecimal code. The Hiragana characters are in the range 0x2421 to 0x2473, Katakana are in the range 0x2521 to 0x2576, and Kanji are (scattered about) in the range 0x3021 to 0x6d55.

When using the Serif typeface and EUC fontindex, these characters can be produced by a *pair* of octal codes. Given the hexadecimal code (e.g., 0x2421), take the first two digits and add 0x80 and do the same to the second two digits (e.g., 0x21 and 0x24 become 0xa4 and 0xa1), then convert both to octal (e.g., 0xa4 and 0xa1 become 244 and 241). For example, the first Hiragana character is produced by '\244\241'.

It is also possible to use the hexadecimal code directly. This works for all non-EUC fonts by specifying an escape sequence of the form '\\#J1234'. For example, the first Hiragana character is produced by '\\#J2421'.

The Kanji characters may be specified in a third way, using the so-called "Nelson Index", by specifying an escape sequence of the form '\\#N1234'. For example, the (obsolete) Kanji for 'one' is produced by '\\#N0001'.

demo(Japanese) shows the available Japanese characters.

Raw Hershey Glyphs: all of the characters in the Hershey fonts are stored in a large array. Some characters are not accessible in any of the Hershey fonts. These characters can only be accessed via an escape sequence of the form '`\\#H1234`'. For example, the fleur-de-lys is produced by '`\\#H0746`'. The sixth and seventh tables of `demo(Hershey)` shows all of the available raw glyphs.

References

https://www.gnu.org/software/plotutils/plotutils.html.

See Also

`demo(Hershey)`, `par`, `text`, `contour`.

`Japanese` for the Japanese characters in the Hershey fonts.

Examples

```
Hershey

## for tables of examples, see demo(Hershey)
```

---

hsv *HSV Color Specification*

Description

Create a vector of colors from vectors specifying hue, saturation and value.

Usage

```
hsv(h = 1, s = 1, v = 1, alpha)
```

Arguments

`h, s, v`	numeric vectors of values in the range `[0, 1]` for 'hue', 'saturation' and 'value' to be combined to form a vector of colors. Values in shorter arguments are recycled.
`alpha`	numeric vector of values in the range `[0, 1]` for alpha transparency channel (0 means transparent and 1 means opaque).

Details

Semi-transparent colors (`0 < alpha < 1`) are supported only on some devices: see `rgb`.

Value

This function creates a vector of colors corresponding to the given values in HSV space. The values returned by `hsv` can be used with a `col=` specification in graphics functions or in `par`.

See Also

hcl for a perceptually based version of hsv(), rgb and rgb2hsv for RGB to HSV conversion; rainbow, gray.

Examples

```
require(graphics)

hsv(.5,.5,.5)

## Red tones:
n <- 20;  y <- -sin(3*pi*((1:n)-1/2)/n)
op <- par(mar = rep(1.5, 4))
plot(y, axes = FALSE, frame.plot = TRUE,
     xlab = "", ylab = "", pch = 21, cex = 30,
     bg = rainbow(n, start = .85, end = .1),
     main = "Red tones")
par(op)
```

Japanese *Japanese characters in R*

Description

The implementation of Hershey vector fonts provides a large number of Japanese characters (Hiragana, Katakana, and Kanji).

Details

Without keyboard support for typing Japanese characters, the only way to produce these characters is to use special escape sequences: see Hershey.

For example, the Hiragana character for the sound 'ka' is produced by '\\#J242b' and the Katakana character for this sound is produced by '\\#J252b'. The Kanji ideograph for "one" is produced by '\\#J306c' or '\\#N0001'.

The output from demo(Japanese) shows tables of the escape sequences for the available Japanese characters.

References

https://www.gnu.org/software/plotutils/plotutils.html

See Also

demo(Japanese), Hershey, text

Examples

```
require(graphics)

plot(1:9, type = "n", axes = FALSE, frame.plot = TRUE, ylab = "",
     main = "example(Japanese)", xlab = "using Hershey fonts")
par(cex = 3)
Vf <- c("serif", "plain")
text(4, 2, "\\#J244b\\#J245b\\#J2473", vfont = Vf)
text(4, 4, "\\#J2538\\#J2563\\#J2551\\#J2573", vfont = Vf)
text(4, 6, "\\#J467c\\#J4b5c", vfont = Vf)
text(4, 8, "Japan", vfont = Vf)
par(cex = 1)
text(8, 2, "Hiragana")
text(8, 4, "Katakana")
text(8, 6, "Kanji")
text(8, 8, "English")
```

---

make.rgb *Create colour spaces*

Description

These functions specify colour spaces for use in convertColor.

Usage

```
make.rgb(red, green, blue, name = NULL, white = "D65",
         gamma = 2.2)

colorConverter(toXYZ, fromXYZ, name, white = NULL, vectorized = FALSE)
```

Arguments

red, green, blue	Chromaticity (xy or xyY) of RGB primaries
name	Name for the colour space
white	Character string specifying the reference white (see 'Details'.)
gamma	Display gamma (nonlinearity). A positive number or the string "sRGB"
fromXYZ	Function to convert from XYZ tristimulus coordinates to this space
toXYZ	Function to convert from this space to XYZ tristimulus coordinates.
vectorized	Whether fromXYZ and toXYZ are vectorized internally to handle input color matrices.

Details

An RGB colour space is defined by the chromaticities of the red, green and blue primaries. These are given as vectors of length 2 or 3 in xyY coordinates (the Y component is not used and may be omitted). The chromaticities are defined relative to a reference white, which must be one of the CIE standard illuminants: "A", "B", "C", "D50", "D55", "D60", "E" (usually "D65").

The display gamma is most commonly 2.2, though 1.8 is used for Apple RGB. The sRGB standard specifies a more complicated function that is close to a gamma of 2.2; `gamma = "sRGB"` uses this function.

Colour spaces other than RGB can be specified directly by giving conversions to and from XYZ tristimulus coordinates. The functions should take two arguments. The first is a vector giving the coordinates for one colour. The second argument is the reference white. If a specific reference white is included in the definition of the colour space (as for the RGB spaces) this second argument should be ignored and may be ....

As of R 3.6.0 the built in color converters along with `convertColor` were vectorized to process three column color matrices in one call, instead of row by row via `apply`. In order to maintain backwards compatibility, `colorConverter` wraps `fromXYZ` and `toXYZ` in a `apply` loop in case they do not also support matrix inputs. If the `fromXYZ` and `toXYZ` functions you are using operate correctly on the whole color matrix at once instead of row by row, you can set `vectorized=TRUE` for a performance improvement.

Value

An object of class `colorConverter`

References

Conversion algorithms from http://www.brucelindbloom.com.

See Also

`convertColor`

Examples

```
(pal <- make.rgb(red =   c(0.6400, 0.3300),
                 green = c(0.2900, 0.6000),
                 blue =  c(0.1500, 0.0600),
                 name = "PAL/SECAM RGB"))

## converter for sRGB in #rrggbb format
hexcolor <- colorConverter(toXYZ = function(hex, ...) {
                           rgb <- t(col2rgb(hex))/255
                           colorspaces$sRGB$toXYZ(rgb, ...) },
                          fromXYZ = function(xyz, ...) {
                             rgb <- colorspaces$sRGB$fromXYZ(xyz, ...)
                             rgb <- round(rgb, 5)
                             if (min(rgb) < 0 || max(rgb) > 1)
                                  as.character(NA)
                             else rgb(rgb[1], rgb[2], rgb[3])},
```

```
                          white = "D65", name = "#rrggbb")

(cols <- t(col2rgb(palette())))
zapsmall(luv <- convertColor(cols, from = "sRGB", to = "Luv", scale.in = 255))
(hex <- convertColor(luv, from = "Luv",  to = hexcolor, scale.out = NULL))

## must make hex a matrix before using it
(cc <- round(convertColor(as.matrix(hex), from = hexcolor, to = "sRGB",
                          scale.in = NULL, scale.out = 255)))
stopifnot(cc == cols)

## Internally vectorized version of hexcolor, notice the use
## of `vectorized = TRUE`:

hexcolorv <- colorConverter(toXYZ = function(hex, ...) {
                            rgb <- t(col2rgb(hex))/255
                            colorspaces$sRGB$toXYZ(rgb, ...) },
                           fromXYZ = function(xyz, ...) {
                              rgb <- colorspaces$sRGB$fromXYZ(xyz, ...)
                              rgb <- round(rgb, 5)
                              oob <- pmin(rgb[,1],rgb[,2],rgb[,3]) < 0 |
                                     pmax(rgb[,1],rgb[,2],rgb[,3]) > 0
                              res <- rep(NA_character_, nrow(rgb))
                              res[!oob] <- rgb(rgb[!oob,,drop=FALSE])},
                           white = "D65", name = "#rrggbb",
                           vectorized=TRUE)
(ccv <- round(convertColor(as.matrix(hex), from = hexcolor, to = "sRGB",
                           scale.in = NULL, scale.out = 255)))
stopifnot(ccv == cols)
```

`msgWindow` *Manipulate a Window*

Description

`msgWindow` sends a message to manipulate the specified screen device's window. With argument `which = -1` it applies to the GUI console (which only accepts the first three actions).

Usage

```
msgWindow(type = c("minimize", "restore", "maximize",
                   "hide", "recordOn", "recordOff"),
          which = dev.cur())
```

Arguments

`type`	action to be taken.
`which`	a device number, or `-1`.

See Also

bringToTop, windows

---

n2mfrow *Compute Default* mfrow *From Number of Plots*

Description

Easy setup for plotting multiple figures (in a rectangular layout) on one page. This computes a sensible default for par(mfrow).

Usage

```
n2mfrow(nr.plots, asp = 1)
```

Arguments

nr.plots	integer; the number of plot figures you'll want to draw.
asp	positive number; the target aspect ratio (columns / rows) in the output. Was implicitly hardwired to 1; because of that and back compatibility, there is a somewhat discontinuous behavior when varying asp around 1, for nr.plots <= 12.

Value

A length-two integer vector (nr, nc) giving the positive number of rows and columns, fulfilling nr * nc >= nr.plots, and currently, for asp = 1, nr >= nc >= 1.

Note

Conceptually, this is a quadratic integer optimization problem, with inequality constraints $nr >= 1$, $nc >= 1$, and $nr.plots >= nr*nc$ (and possibly nr >= asp*nc), and *two* objective functions which would have to be combined via a tuning weight, say w, to, e.g., $(nr.plots - nr * nc) + w|nr/nc - asp|$.

The current algorithm is simple and not trying to solve one of these optimization problems.

Author(s)

Martin Maechler; suggestion of asp by Michael Chirico.

See Also

par, layout.

Examples

```
require(graphics)

n2mfrow(8) # 3 x 3

n <- 5 ; x <- seq(-2, 2, length.out = 51)
## suppose now that 'n' is not known {inside function}
op <- par(mfrow = n2mfrow(n))
for (j in 1:n)
   plot(x, x^j, main = substitute(x^ exp, list(exp = j)), type = "l",
   col = "blue")

sapply(1:14, n2mfrow)
sapply(1:14, n2mfrow, asp=16/9)
```

---

nclass *Compute the Number of Classes for a Histogram*

Description

Compute the number of classes for a histogram, notably `hist()`.

Usage

```
nclass.Sturges(x)
nclass.scott(x)
nclass.FD(x, digits = 5)
```

Arguments

`x`	a data vector.
`digits`	number of *significant* digits to keep when rounding `x` before the `IQR` computation; see 'Details' below.

Details

`nclass.Sturges` uses Sturges' formula, implicitly basing bin sizes on the range of the data.

`nclass.scott` uses Scott's choice for a normal distribution based on the estimate of the standard error, unless that is zero where it returns 1.

`nclass.FD` uses the Freedman-Diaconis choice based on the inter-quartile range (`IQR(signif(x, digits))`) unless that's zero where it uses increasingly more extreme symmetric quantiles up to c(1,511)/512 and if that difference is still zero, reverts to using Scott's choice. The default of `digits = 5` was chosen after a few experiments, but may be too low for some situations, see PR#17274.

Value

The suggested number of classes.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS*. Springer, page 112.

Freedman, D. and Diaconis, P. (1981). On the histogram as a density estimator: L_2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, **57**, 453–476. doi:10.1007/BF01025868.

Scott, D. W. (1979). On optimal and data-based histograms. *Biometrika*, **66**, 605–610. doi:10.2307/2335182.

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice, and Visualization*. Wiley.

Sturges, H. A. (1926). The choice of a class interval. *Journal of the American Statistical Association*, **21**, 65–66. doi:10.1080/01621459.1926.10502161.

See Also

`hist` and `truehist` (package **MASS**); `dpih` (package **KernSmooth**) for a plugin bandwidth proposed by Wand(1995).

Examples

```
set.seed(1)
x <- stats::rnorm(1111)
nclass.Sturges(x)

## Compare them:
NC <- function(x) c(Sturges = nclass.Sturges(x),
      Scott = nclass.scott(x), FD = nclass.FD(x))
NC(x)
onePt <- rep(1, 11)
NC(onePt) # no longer gives NaN
```

---

palette *Set or View the Graphics Palette*

Description

View or manipulate the color palette which is used when `col=` has a numeric index and supporting functions.

Usage

```
palette(value)
palette.pals()
palette.colors(n = NULL, palette = "Okabe-Ito", alpha, recycle = FALSE,
               names = FALSE)
```

Arguments

value	an optional character vector specifying a new palette (see Details).
n	the number of colors to select from a palette. The default NULL selects all colors of the given palette.
palette	a valid palette name (one of palette.pals()). The name is matched to the list of available palettes, ignoring upper vs. lower case, spaces, dashes, etc. in the matching.
alpha	an alpha-transparency level in the range [0,1] (0 means transparent and 1 means opaque).
recycle	logical indicating what happens in case n > length(palette(.)). By default (recycle = FALSE), the result is as for n = NULL, but with a warning.
names	logical indicating whether a named vector of colors should be returned or not (provided that the palette has any names for its colors).

Details

The palette() function gets or sets the current palette, the palette.pals() function lists the available predefined palettes, and the palette.colors() function selects colors from the predefined palettes.

The color palette and referring to colors by number (see e.g. par) was provided for compatibility with S. R extends and improves on the available set of palettes.

If value has length 1, it is taken to be the name of a built-in color palette. The available palette names are returned by palette.pals(). It is also possible to specify "default".

If value has length greater than 1 it is assumed to contain a description of the colors which are to make up the new palette. The maximum size for a palette is 1024 entries.

If value is omitted, no change is made to the current palette.

There is only one palette setting for all devices in an R session. If the palette is changed, the new palette applies to all subsequent plotting.

The current palette also applies to re-plotting (for example if an on-screen device is resized or dev.copy or replayPlot is used). The palette is recorded on the display list at the start of each page and when it is changed.

Value

palette() returns a character vector giving the colors from the palette which *was* in effect. This is invisible unless the argument is omitted.

palette.pals() returns a character vector giving the names of predefined palettes.

palette.colors() returns a vector of R colors. By default (if names = FALSE the vector has no names. If names = TRUE, the function attempts to return a named vector if possible, i.e., for those palettes that provide names for their colors (e.g., "Okabe-Ito", "Tableau 10", or "Alphabet").

See Also

colors for the vector of built-in named colors; hsv, gray, hcl.colors, ... to construct colors.

adjustcolor, e.g., for tweaking existing palettes; colorRamp to interpolate colors, making custom palettes; col2rgb for translating colors to RGB 3-vectors.

Examples

```
require(graphics)

palette()               # obtain the current palette
palette("R3");palette() # old default palette
palette("ggplot2")      # ggplot2-style palette
palette()

palette(hcl.colors(8, "viridis"))

(palette(gray(seq(0,.9,length.out = 25)))) # gray scales; print old palette
matplot(outer(1:100, 1:30), type = "l", lty = 1,lwd = 2, col = 1:30,
        main = "Gray Scales Palette",
        sub = "palette(gray(seq(0, .9, len=25)))")
palette("default")      # reset back to the default

## on a device where alpha transparency is supported,
##  use 'alpha = 0.3' transparency with the default palette :
mycols <- adjustcolor(palette(), alpha.f = 0.3)
opal <- palette(mycols)
x <- rnorm(1000); xy <- cbind(x, 3*x + rnorm(1000))
plot (xy, lwd = 2,
       main = "Alpha-Transparency Palette\n alpha = 0.3")
xy[,1] <- -xy[,1]
points(xy, col = 8, pch = 16, cex = 1.5)
palette("default")

## List available built-in palettes
palette.pals()

## Demonstrate the colors 1:8 in different palettes using a custom matplot()
sinplot <- function(main=NULL, n = 8) {
    x <- outer(
 seq(-pi, pi, length.out = 50),
 seq(  0, pi, length.out = n),
 function(x, y) sin(x - y)
    )
    matplot(x, type = "l", lwd = 4, lty = 1, col = 1:n, ylab = "", main=main)
}
sinplot("default palette")

palette("R3");        sinplot("R3")
palette("Okabe-Ito"); sinplot("Okabe-Ito")
palette("Tableau")  ; sinplot("Tableau", n = 10)
palROB <- colorRampPalette(c("red", "darkorange2", "blue"), space = "Lab")
palette(palROB(16)); sinplot("palROB(16)", n = 16)
palette("default") # reset

## color swatches for palette.colors()
palette.swatch <- function(palette = palette.pals(), n = 8, nrow = 8,
                           border = "black", cex = 1, ...)
{
```

```
    cols <- sapply(palette, palette.colors, n = n, recycle = TRUE)
    ncol <- ncol(cols)
    nswatch <- min(ncol, nrow)
    op <- par(mar = rep(0.1, 4),
              mfrow = c(1, min(5, ceiling(ncol/nrow))),
            cex = cex, ...)
    on.exit(par(op))
    while (length(palette)) {
  subset <- seq_len(min(nrow, ncol(cols)))
  plot.new()
  plot.window(c(0, n), c(0.25, nrow + 0.25))
  y <- rev(subset)
  text(0, y + 0.1, palette[subset], adj = c(0, 0))
  y <- rep(y, each = n)
  rect(rep(0:(n-1), n), y, rep(1:n, n), y - 0.5,
       col = cols[, subset], border = border)
  palette <- palette[-subset]
  cols    <- cols [, -subset, drop = FALSE]
     }
}

palette.swatch()

palette.swatch(n = 26) # show full "Alphabet"; recycle most others
```

Palettes *Color Palettes*

Description

Create a vector of n contiguous colors.

Usage

```
hcl.colors(n, palette = "viridis", alpha = NULL, rev = FALSE, fixup = TRUE)
hcl.pals(type = NULL)

rainbow(n, s = 1, v = 1, start = 0, end = max(1, n - 1)/n,
        alpha, rev = FALSE)
heat.colors(n, alpha, rev = FALSE)
terrain.colors(n, alpha, rev = FALSE)
topo.colors(n, alpha, rev = FALSE)
cm.colors(n, alpha, rev = FALSE)
```

Arguments

n	the number of colors (≥ 1) to be in the palette.
palette	a valid palette name (one of `hcl.pals()`). The name is matched to the list of available palettes, ignoring upper vs. lower case, spaces, dashes, etc. in the matching.

alpha	an alpha-transparency level in the range [0,1] (0 means transparent and 1 means opaque), see argument alpha in hsv and hcl, respectively. A missing, i.e., not explicitly specified alpha is equivalent to alpha = NULL, which does *not* add opacity codes ("FF") to the individual color hex codes.
rev	logical indicating whether the ordering of the colors should be reversed.
fixup	logical indicating whether the resulting color should be corrected to RGB coordinates in [0,1], see hcl.
type	the type of palettes to list: "qualitative", "sequential", "diverging", or "divergingx". NULL lists all palettes.
s, v	the 'saturation' and 'value' to be used to complete the HSV color descriptions.
start	the (corrected) hue in [0,1] at which the rainbow begins.
end	the (corrected) hue in [0,1] at which the rainbow ends.

Details

All of these functions (except the helper function hcl.pals) create a vector of n contiguous colors, either based on the HSV color space (rainbow, heat, terrain, topography, and cyan-magenta colors) or the perceptually-based HCL color space.

HSV (hue-saturation-value) is a simple transformation of the RGB (red-green-blue) space which was therefore a convenient choice for color palettes in many software systems (see also hsv). However, HSV colors capture the perceptual properties hue, colorfulness/saturation/chroma, and lightness/brightness/luminance/value only poorly and consequently the corresponding palettes are typically not a good choice for statistical graphics and data visualization.

In contrast, HCL (hue-chroma-luminance) colors are much more suitable for capturing human color perception (see also hcl) and better color palettes can be derived based on HCL coordinates. Conceptually, three types of palettes are often distinguished:

- Qualitative: For coding categorical information, i.e., where no particular ordering of categories is available and every color should receive the same perceptual weight.
- Sequential: For coding ordered/numeric information, i.e., where colors go from high to low (or vice versa).
- Diverging: Designed for coding numeric information around a central neutral value, i.e., where colors diverge from neutral to two extremes.

The hcl.colors function provides a basic and lean implementation of the pre-specified palettes in the **colorspace** package. In addition to the types above, the functions distinguish "diverging" palettes where the two arms are restricted to be rather balanced as opposed to flexible "divergingx" palettes that combine two sequential palettes without any restrictions. The latter group also includes the cividis palette as it is based on two different hues (blue and yellow) but it is actually a sequential palette (going from dark to light).

The names of all available HCL palettes can be queried with the hcl.pals function and they are also visualized by color swatches in the examples. Many of the palettes closely approximate palettes of the same name from various other packages (including **RColorBrewer**, **rcartocolor**, **viridis**, **scico**, among others).

The default HCL palette is the widely used viridis palette which is a sequential palette with relatively high chroma throughout so that it also works reasonably well as a qualitative palette. However,

while viridis is a rather robust default palette, more suitable HCL palettes are available for most visualizations.

For example, "Dark 3" works well for shading points or lines in up to five groups, "YlGnBu" is a sequential palette similar to "viridis" but with aligned chroma/luminance, and "Green-Brown" or "Blue-Red 3" are colorblind-safe diverging palettes.

Further qualitative palettes are provided in the palette.colors function. While the qualitative palettes in hcl.colors are always based on the same combination of chroma and luminance, the palette.colors vary in chroma and luminance up to a certain degree. The advantage of fixing chroma/luminance is that the perceptual weight of the resulting colors is more balanced. The advantage of allowing variation is that more distinguishable colors can be obtained, especially for viewers with color vision deficiencies.

Note that the rainbow function implements the (in-)famous rainbow (or jet) color palette that was used very frequently in many software packages but has been widely criticized for its many perceptual problems. It is specified by a start and end hue with red = 0, yellow = $\frac{1}{6}$, green = $\frac{2}{6}$, cyan = $\frac{3}{6}$, blue = $\frac{4}{6}$, and magenta = $\frac{5}{6}$. However, these are very flashy and unbalanced with respect to both chroma and luminance which can lead to various optical illusions. Also, the hues that are equispaced in RGB space tend to cluster at the red, green, and blue primaries. Therefore, it is recommended to use a suitable palette from hcl.colors instead of rainbow.

Value

A character vector cv containing either palette names (for hcl.pals) or n hex color codes (for all other functions). The latter can be used either to create a user-defined color palette for subsequent graphics by palette(cv), a col = specification in graphics functions or in par.

References

Wikipedia (2019). HCL color space – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=HCL_color_space&oldid=883465135. Accessed March 26, 2019.

Zeileis, A., Fisher, J. C., Hornik, K., Ihaka, R., McWhite, C. D., Murrell, P., Stauffer, R. and Wilke, C. O. (2020). "colorspace: A toolbox for manipulating and assessing colors and palettes." *Journal of Statistical Software*, **96**(1), 1–49. doi:10.18637/jss.v096.i01

Ihaka, R. (2003). "Colour for presentation graphics." Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, 2003, Technische Universität Wien, Vienna, Austria. https://www.R-project.org/conferences/DSC-2003/.

Zeileis, A., Hornik, K. and Murrell, P. (2009). Escaping RGBland: Selecting colors for statistical graphics. *Computational Statistics & Data Analysis*, **53**, 3259–3270. doi:10.1016/j.csda.2008.11.033.

See Also

colors, palette, gray.colors, hsv, hcl, rgb, gray and col2rgb for translating to RGB numbers.

Examples

```
require("graphics")
```

```
# color wheels in RGB/HSV and HCL space
par(mfrow = c(2, 2))
pie(rep(1, 12), col = rainbow(12), main = "RGB/HSV")
pie(rep(1, 12), col = hcl.colors(12, "Set 2"), main = "HCL")
par(mfrow = c(1, 1))

## color swatches for RGB/HSV palettes
demo.pal <-
  function(n, border = if (n < 32) "light gray" else NA,
           main = paste("color palettes;  n=", n),
           ch.col = c("rainbow(n, start=.7, end=.1)", "heat.colors(n)",
                      "terrain.colors(n)", "topo.colors(n)",
                      "cm.colors(n)"))
{
    nt <- length(ch.col)
    i <- 1:n; j <- n / nt; d <- j/6; dy <- 2*d
    plot(i, i+d, type = "n", yaxt = "n", ylab = "", main = main)
    for (k in 1:nt) {
        rect(i-.5, (k-1)*j+ dy, i+.4, k*j,
             col = eval(str2lang(ch.col[k])), border = border)
        text(2*j,  k * j + dy/4, ch.col[k])
    }
}
demo.pal(16)

## color swatches for HCL palettes
hcl.swatch <- function(type = NULL, n = 5, nrow = 11,
  border = if (n < 15) "black" else NA) {
    palette <- hcl.pals(type)
    cols <- sapply(palette, hcl.colors, n = n)
    ncol <- ncol(cols)
    nswatch <- min(ncol, nrow)

    par(mar = rep(0.1, 4),
        mfrow = c(1, min(5, ceiling(ncol/nrow))),
        pin = c(1, 0.5 * nswatch),
        cex = 0.7)

    while (length(palette)) {
        subset <- 1:min(nrow, ncol(cols))
        plot.new()
        plot.window(c(0, n), c(0, nrow + 1))
        text(0, rev(subset) + 0.1, palette[subset], adj = c(0, 0))
        y <- rep(subset, each = n)
        rect(rep(0:(n-1), n), rev(y), rep(1:n, n), rev(y) - 0.5,
             col = cols[, subset], border = border)
        palette <- palette[-subset]
        cols <- cols[, -subset, drop = FALSE]
    }

    par(mfrow = c(1, 1), mar = c(5.1, 4.1, 4.1, 2.1), cex = 1)
}
hcl.swatch()
```

```
hcl.swatch("qualitative")
hcl.swatch("sequential")
hcl.swatch("diverging")
hcl.swatch("divergingx")

## heat maps with sequential HCL palette (purple)
image(volcano, col = hcl.colors(11, "purples", rev = TRUE))
filled.contour(volcano, nlevels = 10,
               color.palette = function(n, ...)
                   hcl.colors(n, "purples", rev = TRUE, ...))

## list available HCL color palettes
hcl.pals("qualitative")
hcl.pals("sequential")
hcl.pals("diverging")
hcl.pals("divergingx")
```

---

pdf *PDF Graphics Device*

Description

pdf starts the graphics device driver for producing PDF graphics.

Usage

```
pdf(file = if(onefile) "Rplots.pdf" else "Rplot%03d.pdf",
    width, height, onefile, family, title, fonts, version,
    paper, encoding, bg, fg, pointsize, pagecentre, colormodel,
    useDingbats, useKerning, fillOddEven, compress)
```

Arguments

file	a character string giving the file path. See the section 'File specifications' for further details.
width, height	the width and height of the graphics region in inches. The default values are 7.
onefile	logical: if true (the default) allow multiple figures in one file. If false, generate a file with name containing the page number for each page. Defaults to TRUE, and forced to true if file is a pipe.
family	the initial font family to be used, normally as a character string. See the section 'Families'. Defaults to "Helvetica".
title	title string to embed as the '/Title' field in the file. Defaults to "R Graphics Output".
fonts	a character vector specifying R graphics font family names for additional fonts which will be included in the PDF file. Defaults to NULL.

`version` a string describing the PDF version that will be required to view the output. This is a minimum, and will be increased (with a warning) if necessary. Defaults to `"1.4"`, but see 'Details'.

`paper` the target paper size. The choices are `"a4"`, `"letter"`, `"legal"` (or `"us"`) and `"executive"` (and these can be capitalized), or `"a4r"` and `"USr"` for rotated ('landscape'). The default is `"special"`, which means that the `width` and `height` specify the paper size. A further choice is `"default"`; if this is selected, the paper size is taken from the option `"papersize"` if that is set and as `"a4"` if it is unset or empty. Defaults to `"special"`.

`encoding` the name of an encoding file. Defaults to `"default"`. The latter is interpreted

on Unix-alikes as '`"ISOLatin1.enc"`' unless the locale is recognized as corresponding to a language using ISO 8859-{2,5,7,13,15} or KOI8-{R,U}.

on Windows as '`"CP1250.enc"`' (Central European), `"CP1251.enc"` (Cyrillic), `"CP1253.enc"` (Greek) or `"CP1257.enc"` (Baltic) if one of those codepages is in use, otherwise '`"WinAnsi.enc"`' (codepage 1252).

The file is looked for in the 'enc' directory of package **grDevices** if the path does not contain a path separator. An extension `".enc"` can be omitted.

`bg` the initial background color to be used. Defaults to `"transparent"`.

`fg` the initial foreground color to be used. Defaults to `"black"`.

`pointsize` the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points. Defaults to `12`.

`pagecentre` logical: should the device region be centred on the page? – is only relevant for `paper != "special"`. Defaults to `TRUE`.

`colormodel` a character string describing the color model: currently allowed values are `"srgb"`, `"gray"` (or `"grey"`) and `"cmyk"`. Defaults to `"srgb"`. See section 'Color models'.

`useDingbats` logical. Should small circles be rendered *via* the Dingbats font? Defaults to `FALSE`. If `TRUE`, this can produce smaller and better output, but can cause font display problems in broken PDF viewers: although this font is one of the 14 guaranteed to be available in all PDF viewers, that guarantee is not always honoured.

For Unix-alikes (including macOS) see the 'Note' for a possible fix for some viewers.

`useKerning` logical. Should kerning corrections be included in setting text and calculating string widths? Defaults to `TRUE`.

`fillOddEven` logical controlling the polygon fill mode: see `polygon` for details. Defaults to `FALSE`.

`compress` logical. Should PDF streams be generated with Flate compression? Defaults to `TRUE`.

Details

All arguments except `file` default to values given by `pdf.options()`. The ultimate defaults are quoted in the arguments section.

`pdf()` opens the file `file` and the PDF commands needed to plot any graphics requested are sent to that file.

The `family` argument can be used to specify a PDF-specific font family as the initial/default font for the device. If additional font families are to be used they should be included in the `fonts` argument.

If a device-independent R graphics font family is specified (e.g., via `par(family = )` in the graphics package), the PDF device makes use of the PostScript font mappings to convert the R graphics font family to a PDF-specific font family description. (See the documentation for `pdfFonts`.)

This device does *not* embed fonts in the PDF file, so it is only straightforward to use mappings to the font families that can be assumed to be available in any PDF viewer: `"Times"` (equivalently `"serif"`), `"Helvetica"` (equivalently `"sans"`) and `"Courier"` (equivalently `"mono"`). Other families may be specified, but it is the user's responsibility to ensure that these fonts are available on the system and third-party software (e.g., Ghostscript) may be required to embed the fonts so that the PDF can be included in other documents (e.g., LaTeX): see `embedFonts`. The URW-based families described for in section 'Families'.can be used with viewers, platform dependently:

on Unix-alikes viewers set up to use URW fonts, which is usual with those based on `xpdf` or Ghostscript.

on Windows viewers such as GSView which utilise URW fonts.

Since `embedFonts` makes use of Ghostscript, it should be able to embed the URW-based families for use with other viewers.

The PDF produced is fairly simple, with each page being represented as a single stream (by default compressed and possibly with references to raster images). The R graphics model does not distinguish graphics objects at the level of the driver interface.

The `version` argument declares the version of PDF that gets produced. The version must be at least 1.2 when compression is used, 1.4 for semi-transparent output to be understood, and at least 1.3 if CID fonts are to be used: if any of these features are used the version number will be increased (with a warning). (PDF 1.4 was first supported by Acrobat 5 in 2001; it is very unlikely not to be supported in a current viewer.)

Line widths as controlled by `par(lwd = )` are in multiples of 1/96 inch. Multiples less than 1 are allowed. `pch = "."` with `cex = 1` corresponds to a square of side 1/72 inch, which is also the 'pixel' size assumed for graphics parameters such as `"cra"`.

The `paper` argument sets the '`/MediaBox`' entry in the file, which defaults to `width` by `height`. If it is set to something other than `"special"`, a device region of the specified size is (by default) centred on the rectangle given by the paper size: if either `width` or `height` is less than `0.1` or too large to give a total margin of 0.5 inch, it is reset to the corresponding paper dimension minus 0.5. Thus if you want the default behaviour of `postscript` use `pdf(paper = "a4r", width = 0, height = 0)` to centre the device region on a landscape A4 page with 0.25 inch margins.

When the background colour is fully transparent (as is the initial default value), the PDF produced does not paint the background. Most PDF viewers will use a white canvas so the visual effect is if the background were white. This will not be the case when printing onto coloured paper, though.

File specifications

Tilde expansion (see `path.expand`) is done on the `file` argument. An input with a marked encoding is converted to the native encoding or an error is given.

For use with onefile = FALSE, give a C integer format such as "Rplot%03d.pdf" (the default in that case) which is expanded using the page number, so this uses files 'Rplot001.pdf', ..., 'Rplot999.pdf', 'Rplot1000.pdf',

A single integer format matching the regular expression "%[#0 +=-]*[0-9.]*[diouxX]" is allowed in file. The character string should not otherwise contain a %: if it is really necessary, use %% in the string for % in the file path.

For pdf, file can be NULL when no external file is created (effectively, no drawing occurs), but the device may still be queried (e.g., for the size of text by (base graphics) strwidth or (**grid**) stringWidth).

Families

Font families are collections of fonts covering the five font faces, (conventionally plain, bold, italic, bold-italic and symbol) selected by the graphics parameter par(font =) or the grid parameter gpar(fontface =). Font families can be specified either as an initial/default font family for the device via the family argument or after the device is opened by the graphics parameter par(family =) or the grid parameter gpar(fontfamily =). Families which will be used in addition to the initial family must be specified in the fonts argument when the device is opened.

Font families are declared via a call to pdfFonts or postscriptFonts.

The argument family specifies the initial/default font family to be used. In normal use it is one of "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times", and refers to the standard Adobe PostScript fonts families of those names which are included (or cloned) in all common PDF/PostScript renderers.

Many PDF/PostScript renders (including those based on Ghostscript) use the URW equivalents of these fonts, which are "URWGothic", "URWBookman", "NimbusMon", "NimbusSan", "NimbusSanCond", "CenturySch", "URWPalladio" and "NimbusRom" respectively. If your viewer is using URW fonts, you will obtain access to more characters and more appropriate metrics by using these names. To make these easier to remember, "URWHelvetica" == "NimbusSan" and "URWTimes" == "NimbusRom" are also supported. However, if the viewer is not using URW fonts (for example Adobe Acrobat Reader) it may substitute inappropriately or not render at all. (Consider using embedFonts.)

As from R 4.4.0 there is support for URW 2.0 fonts in families "URW2Helvetica" (with 'Oblique' fonts), "URW2HelveticaItalic" (with 'Italic' fonts), "URW2Times" and "NimbusMonoPS". As recent versions of Ghostscript will render with (and embed) these fonts. these families should be used instead of "URWHelvetica", "URWTimes", "NimbusSan", "NimbusRom", and "NimbusMon"

Another type of family makes use of CID-keyed fonts for East Asian languages – see pdfFonts.

The family argument is normally a character string naming a font family, but family objects generated by Type1Font and CIDFont are also accepted. For compatibility with earlier versions of R, the initial family can also be specified as a vector of four or five afm files.

Note that R does not embed the font(s) used in the PostScript output: see embedFonts for a utility to help do so.

Viewers and embedding applications frequently substitute fonts for those specified in the family, and the substitute will often have slightly different font metrics. useKerning = TRUE spaces the letters in the string using kerning corrections for the intended family: this may look uglier than useKerning = FALSE.

Encodings

Encodings describe which glyphs are used to display the character codes (in the range 0–255). Most commonly R uses ISOLatin1 encoding, and the examples for `text` are in that encoding. However, the encoding used on machines running R may well be different, and by using the `encoding` argument the glyphs can be matched to encoding in use. This suffices for European and Cyrillic languages, but not for East Asian languages. For the latter, composite CID fonts are used. These fonts are useful for other languages: for example they may contain Greek glyphs. (The rest of this section applies only when CID fonts are not used.)

None of this will matter if only ASCII characters (codes 32–126) are used as all the encodings (except `"TeXtext"`) agree over that range. Some encodings are supersets of ISOLatin1. However, if accented and special characters do not come out as you expect, you may need to change the encoding. Some other encodings are supplied with R: `"ISOLatin2.enc"` (Central/Eastern Europe), `"ISOLatin7.enc"` (ISO 8859-13, 'Baltic Rim'), `"ISOLatin9.enc"` (ISO 8859-15, including Euro), `"Cyrillic.enc"` (ISO 8859-5), `"KOI8-R.enc"`, `"KOI8-U.enc"`, and the Windows encodings `"WinAnsi.enc"` (also known as `"CP1252.enc"`, `"CP1250.enc"` (Central/Eastern Europe), `"CP1251.enc"` (Cyrillic), `"Greek.enc"` (ISO 8859-7), `"CP1253.enc"` (modern Greek) and `"CP1257.enc"` ('Baltic Rim'). Note that many glyphs in these encodings are not in the fonts corresponding to the standard families. (The Adobe ones for all but Courier, Helvetica and Times cover little more than Latin-1, whereas the URW ones also cover Latin-2, Latin-7, Latin-9 and Cyrillic but no Greek. The Adobe exceptions cover the Latin character sets, but not the Euro.)

NB: support for encodings other than `"ISOLatin1.enc"` (and the Windows ones on Windows) depends on support by the platform's `libiconv` in a UTF-8 locale.

If you specify the encoding, it is your responsibility to ensure that the PostScript font contains the glyphs used. One issue here is the Euro symbol which is in several encodings (including WinAnsi and ISOLatin9 encodings) but may well not be in the PostScript fonts. (It is in the URW variants; it is not in the supplied Adobe Font Metric files so will not be centred correctly.)

There is an exception. Character 45 (`"-"`) is always set as minus (its value in Adobe ISOLatin1) even though it is hyphen in the other encodings. Hyphen is available as character 173 (octal 0255) in all the Latin encodings, Cyrillic and Greek. (This can be entered as `"\u00ad"` in a UTF-8 locale.) There are some discrepancies in accounts of glyphs 39 and 96: the supplied encodings (except CP1250 and CP1251) treat these as 'quoteright' and 'quoteleft' (rather than 'quotesingle'/'acute' and 'grave' respectively), as they are in the Adobe documentation.

Color models

The default color model (`"srgb"`) is sRGB. Model `"gray"` (or `"grey"`) maps sRGB colors to greyscale using perceived luminosity (biased towards green). `"cmyk"` outputs in CMYK colorspace. The simplest possible conversion from sRGB to CMYK is used (https://en.wikipedia.org/wiki/CMYK_color_model#Mapping_RGB_to_CMYK), and raster images are output in RGB.

Also available for backwards compatibility is model `"rgb"` which uses uncalibrated RGB and corresponds to the model used with that name in R prior to 2.13.0. Some viewers may render some plots in that colorspace faster than in sRGB, and the plot files will be smaller.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are as a multiple of 1/96 inch, with a minimum of 0.01 enforced.
- Circles of any radius are allowed. If `useDingbats = TRUE`, opaque circles of less than 10 big points radius are rendered using char 108 in the Dingbats font: all semi-transparent and larger circles using a Bézier curve for each quadrant.
- Colours are by default specified as sRGB.

At very small line widths, the line type may be forced to solid.

Printing

Except on Windows it is possible to print directly from `pdf` by something like (this is appropriate for a CUPS printing system):

```
pdf("|lp -o landscape", paper = "a4r")
```

This forces `onefile = TRUE`.

Note

If you have drawn any typeset glyphs (see `glyphInfo`) then it is *highly* recommended that you use `embedGlyphs` to embed the relevant fonts.

Note

If you see problems with PDF output, do remember that the problem is much more likely to be in your viewer than in R. Try another viewer if possible. Symptoms for which the viewer has been at fault are apparent grids on image plots (turn off graphics anti-aliasing in your viewer if you can) and missing or incorrect glyphs in text (viewers silently doing font substitution).

Unfortunately the default viewers on most Linux and macOS systems have these problems, and no obvious way to turn off graphics anti-aliasing.

Acrobat Reader does not use the fonts specified but rather emulates them from multiple-master fonts. This can be seen in imprecise centering of characters, for example the multiply and divide signs in Helvetica. This can be circumvented by embedding fonts where possible. Most other viewers substitute fonts, e.g. URW fonts for the standard Helvetica and Times fonts, and these too often have different font metrics from the true fonts.

Acrobat Reader can be extended by 'font packs', and these will be needed for the full use of encodings other than Latin-1 (although they may be offered for download as needed).

On some Unix-alike systems: If `useDingbats = TRUE`, the default plotting character `pch = 1` was displayed in some PDF viewers incorrectly as a `"q"` character. (These seem to be viewers based on the 'poppler' PDF rendering library). This may be due to incorrect or incomplete mapping of font names to those used by the system. Adding the following lines to '`~/.fonts.conf`' or '`/etc/fonts/local.conf`' may circumvent this problem, although this has largely been corrected on the affected systems.

```
<fontconfig>
<alias binding="same">
  <family>ZapfDingbats</family>
  <accept><family>Dingbats</family></accept>
</alias>
</fontconfig>
```

Some further workarounds for problems with symbol fonts on viewers using 'fontconfig' are given in the 'Cairo Fonts' section of the help for X11.

On Windows: The TeXworks PDF viewer was one of those which has been seen to fail to display Dingbats (used by e.g. pch = 1) correctly. Whereas on other platforms the problems seen were incorrect output, on Windows points were silently omitted: however recent versions seem to manage to display Dingbats.

There was a different font bug in the pdf.js viewer included in Firefox: that mapped Dingbats to the Symbol font and so displayed symbols such pch = 1 as lambda.

See Also

pdfFonts, pdf.options, embedFonts, glyphInfo, Devices, postscript.

cairo_pdf and (on macOS only) quartz for other devices that can produce PDF.

More details of font families and encodings and especially handling text in a non-Latin-1 encoding and embedding fonts can be found in

Paul Murrell and Brian Ripley (2006). "Non-standard fonts in PostScript and PDF graphics." *R News*, **6**(2), 41–47. https://www.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

Examples

```
## Test function for encodings
TestChars <- function(encoding = "ISOLatin1", ...)
{
    pdf(encoding = encoding, ...)
    par(pty = "s")
    plot(c(-1,16), c(-1,16), type = "n", xlab = "", ylab = "",
         xaxs = "i", yaxs = "i")
         title(paste("Centred chars in encoding", encoding))
    grid(17, 17, lty = 1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch = i)
    }
    dev.off()
}
## there will be many warnings.
TestChars("ISOLatin2")
## this does not view properly in older viewers.
TestChars("ISOLatin2", family = "URWHelvetica")
## works well for viewing in gs-based viewers, and often in xpdf.
```

pdf.options *Auxiliary Function to Set/View Defaults for Arguments of pdf*

Description

The auxiliary function pdf.options can be used to set or view (if called without arguments) the default values for some of the arguments to pdf.

pdf.options needs to be called before calling pdf, and the default values it sets can be overridden by supplying arguments to pdf.

Usage

```
pdf.options(..., reset = FALSE)
```

Arguments

...	arguments width, height, onefile, family, title, fonts, paper, encoding, pointsize, bg, fg, pagecentre, useDingbats, colormodel, fillOddEven and compress can be supplied.
reset	logical: should the defaults be reset to their 'factory-fresh' values?

Details

If both reset = TRUE and ... are supplied the defaults are first reset to the 'factory-fresh' values and then the new values are applied.

Value

A named list of all the defaults. If any arguments are supplied the return values are the old values and the result has the visibility flag turned off.

See Also

pdf, ps.options.

Examples

```
pdf.options(bg = "pink")
utils::str(pdf.options())
pdf.options(reset = TRUE) # back to factory-fresh
```

pictex *A PicTeX Graphics Driver*

Description

This function produces simple graphics suitable for inclusion in TeX and LaTeX documents. It dates from the very early days of R and is for historical interest only. It was deprecated in R 4.4.0. Consider the **tikzDevice** instead.

Usage

```
pictex(file = "Rplots.tex", width = 5, height = 4, debug = FALSE,
       bg = "white", fg = "black")
```

Arguments

file	the file path where output will appear. Tilde expansion (see path.expand) is done. An input with a marked encoding is converted to the native encoding or an error is given.
width	The width of the plot in inches.
height	the height of the plot in inches.
debug	should debugging information be printed.
bg	the background color for the plot. Ignored.
fg	the foreground color for the plot. Ignored.

Details

This driver is much more basic than the other graphics drivers included in R. It does not have any font metric information, so the use of plotmath is not supported.

Line widths are ignored except when setting the spacing of line textures. pch = "." corresponds to a square of side 1pt.

This device does not support colour (nor does the PicTeX package), and all colour settings are ignored.

Note that text is recorded in the file as-is, so annotations involving TeX special characters (such as ampersand and underscore) need to be quoted as they would be when entering TeX.

Multiple plots will be placed as separate environments in the output file.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 5 inches by 4 inches.
- There is no pointsize argument: the default size is interpreted as 10 point.

- The only font family is `cmss10`.
- Line widths are only used when setting the spacing on line textures.
- Circle of any radius are allowed.
- Colour is not supported.

Author(s)

This driver was provided around 1996–7 by Valerio Aimale of the Department of Internal Medicine, University of Genoa, Italy.

References

Knuth, D. E. (1984) *The TeXbook.* Reading, MA: Addison-Wesley.

Lamport, L. (1994) *LATEX: A Document Preparation System.* Reading, MA: Addison-Wesley.

Goossens, M., Mittelbach, F. and Samarin, A. (1994) *The LATEX Companion.* Reading, MA: Addison-Wesley.

See Also

`pdf`, `postscript`, `Devices`.

The `tikzDevice` in the CRAN package of that name for more modern TeX-based graphics (`https://pgf.sourceforge.net/`, although including PDF figures *via* `pdftex` is most common in (La)TeX documents).

Examples

```
require(graphics)


pictex()
plot(1:11, (-5:5)^2, type = "b", main = "Simple Example Plot")
dev.off()
##--------------------
## Not run:
%% LaTeX Example
\documentclass{article}
\usepackage{pictex}
\usepackage{graphics} % for \rotatebox
\begin{document}
%...
\begin{figure}[h]
  \centerline{\input{Rplots.tex}}
  \caption{}
\end{figure}
%...
\end{document}

## End(Not run)
##--------------------
```

```
unlink("Rplots.tex")
```

---

plotmath *Mathematical Annotation in R*

Description

If the text argument to one of the text-drawing functions (text, mtext, axis, legend) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on persp plots).

In most cases other language objects (names and calls, including formulas) are coerced to expressions and so can also be used.

Details

A mathematical expression must obey the normal rules of syntax for any R expression, but it is interpreted according to very different rules than for normal R expressions.

It is possible to produce many different mathematical symbols, generate sub- or superscripts, produce fractions, etc.

The output from demo(plotmath) includes several tables which show the available features. In these tables, the columns of grey text show sample R expressions, and the columns of black text show the resulting output.

The available features are also described in the tables below:

Syntax	Meaning
x + y	x plus y
x - y	x minus y
x*y	juxtapose x and y
x/y	x forwardslash y
x %+-% y	x plus or minus y
x %/% y	x divided by y
x %*% y	x times y
x %.% y	x cdot y
x[i]	x subscript i
x^2	x superscript 2
paste(x, y, z)	juxtapose x, y, and z
sqrt(x)	square root of x
sqrt(x, y)	y-th root of x
x == y	x equals y
x != y	x is not equal to y
x < y	x is less than y
x <= y	x is less than or equal to y
x > y	x is greater than y
x >= y	x is greater than or equal to y

`!x`	not x
`x %~~% y`	x is approximately equal to y
`x %=~% y`	x and y are congruent
`x %==% y`	x is defined as y
`x %prop% y`	x is proportional to y
`x %~% y`	x is distributed as y
`plain(x)`	draw x in normal font
`bold(x)`	draw x in bold font
`italic(x)`	draw x in italic font
`bolditalic(x)`	draw x in bold italic font
`symbol(x)`	draw x in symbol font
`list(x, y, z)`	comma-separated list
`...`	ellipsis (height varies)
`cdots`	ellipsis (vertically centred)
`ldots`	ellipsis (at baseline)
`x %subset% y`	x is a proper subset of y
`x %subseteq% y`	x is a subset of y
`x %notsubset% y`	x is not a subset of y
`x %supset% y`	x is a proper superset of y
`x %supseteq% y`	x is a superset of y
`x %in% y`	x is an element of y
`x %notin% y`	x is not an element of y
`hat(x)`	x with a circumflex
`tilde(x)`	x with a tilde
`dot(x)`	x with a dot
`ring(x)`	x with a ring
`bar(xy)`	xy with bar
`widehat(xy)`	xy with a wide circumflex
`widetilde(xy)`	xy with a wide tilde
`x %<->% y`	x double-arrow y
`x %->% y`	x right-arrow y
`x %<-% y`	x left-arrow y
`x %up% y`	x up-arrow y
`x %down% y`	x down-arrow y
`x %<=>% y`	x is equivalent to y
`x %=>% y`	x implies y
`x %<=% y`	y implies x
`x %dblup% y`	x double-up-arrow y
`x %dbldown% y`	x double-down-arrow y
`alpha – omega`	Greek symbols
`Alpha – Omega`	uppercase Greek symbols
`theta1, phi1, sigma1, omega1`	cursive Greek symbols
`Upsilon1`	capital upsilon with hook
`aleph`	first letter of Hebrew alphabet
`infinity`	infinity symbol
`partialdiff`	partial differential symbol
`nabla`	nabla, gradient symbol
`32*degree`	32 degrees

`60*minute`	60 minutes of angle
`30*second`	30 seconds of angle
`displaystyle(x)`	draw x in normal size (extra spacing)
`textstyle(x)`	draw x in normal size
`scriptstyle(x)`	draw x in small size
`scriptscriptstyle(x)`	draw x in very small size
`underline(x)`	draw x underlined
`x ~~ y`	put extra space between x and y
`x + phantom(0) + y`	leave gap for "0", but don't draw it
`x + over(1, phantom(0))`	leave vertical gap for "0" (don't draw)
`frac(x, y)`	x over y
`over(x, y)`	x over y
`atop(x, y)`	x over y (no horizontal bar)
`sum(x[i], i==1, n)`	sum x[i] for i equals 1 to n
`prod(plain(P)(X==x), x)`	product of P(X=x) for all values of x
`integral(f(x)*dx, a, b)`	definite integral of f(x) wrt x
`union(A[i], i==1, n)`	union of A[i] for i equals 1 to n
`intersect(A[i], i==1, n)`	intersection of A[i]
`lim(f(x), x %->% 0)`	limit of f(x) as x tends to 0
`min(g(x), x > 0)`	minimum of g(x) for x greater than 0
`inf(S)`	infimum of S
`sup(S)`	supremum of S
`x^y + z`	normal operator precedence
`x^(y + z)`	visible grouping of operands
`x^{y + z}`	invisible grouping of operands
`group("(",list(a, b),"]")`	specify left and right delimiters
`bgroup("(",atop(x,y),")")`	use scalable delimiters
`group(lceil, x, rceil)`	special delimiters
`group(lfloor, x, rfloor)`	special delimiters
`group(langle, list(x, y), rangle)`	special delimiters

The supported 'scalable delimiters' are `|` `(` `[` `{` and their right-hand versions. `"."` is equivalent to `""`: the corresponding delimiter will be omitted. Delimiter `||` is supported but has the same effect as `|`. The special delimiters `lceil`, `lfloor`, `langle` (and their right-hand versions) are not scalable.

Note that `paste` does not insert spaces when juxtaposing, unlike (by default) the R function of that name.

The symbol font uses Adobe Symbol encoding so, for example, a lower case mu can be obtained either by the special symbol `mu` or by `symbol("m")`. This provides access to symbols that have no special symbol name, for example, the universal, or forall, symbol is `symbol("\042")`. To see what symbols are available in this way use `TestChars(font=5)` as given in the examples for `points`: some are only available on some devices.

Note to TeX users: TeX's '`\Upsilon`' is `Upsilon1`, TeX's '`\varepsilon`' is close to `epsilon`, and there is no equivalent of TeX's '`\epsilon`'. TeX's '`\varpi`' is close to `omega1`. `vartheta`, `varphi` and `varsigma` are allowed as synonyms for `theta1`, `phi1` and `sigma1`.

`sigma1` is also known as `stigma`, its Unicode name.

Control characters (e.g., '\n') are not interpreted in character strings in plotmath, unlike normal plotting.

The fonts used are taken from the current font family, and so can be set by par(family=) in base graphics, and gpar(fontfamily=) in package **grid**.

Note that bold, italic and bolditalic do not apply to symbols, and hence not to the Greek *symbols* such as mu which are displayed in the symbol font. They also do not apply to numeric constants.

Other symbols

On many OSes and some graphics devices many other symbols are available as part of the standard text font, and all of the symbols in the Adobe Symbol encoding are in principle available *via* changing the font face or (see 'Details') plotmath: see the examples section of points for a function to display them. ('In principle' because some of the glyphs are missing from some implementations of the symbol font.) Unfortunately, pdf and postscript have support for little more than European (not Greek) and CJK characters and the Adobe Symbol encoding (and in a few fonts, also Cyrillic characters).

On Unix-alikes: In a UTF-8 locale any Unicode character can be entered, perhaps as a '\uxxxx' or '\Uxxxxxxxx' escape sequence, but the issue is whether the graphics device is able to display the character. The widest range of characters is likely to be available in the X11 device using cairo: see its help page for how installing additional fonts can help. This can often be used to display Greek *letters* in bold or italic.

On macOS the quartz device and the default system fonts have quite large coverage.

In non-UTF-8 locales there is normally no support for symbols not in the languages for which the current encoding was intended.

On Windows: Any Unicode character can be entered into a text string *via* a '\uxxxx' escape, or used by number in a call to points. The windows family of devices can display such characters if they are available in the font in use. This can often be used to display Greek *letters* in bold or italic.

A good way to both find out which characters are available in a font and to determine the Unicode number is to use the 'Character Map' accessory (usually on the 'Start' menu under 'Accessories->System Tools'). You can also copy-and-paste characters from the 'Character Map' window to the Rgui console (but not to Rterm).

References

Murrell, P. and Ihaka, R. (2000). An approach to providing mathematical annotation in plots. *Journal of Computational and Graphical Statistics*, **9**, 582–599. doi:10.2307/1390947.

A list of the symbol codes can be found in decimal, octal and hex at https://www.stat.auckland.ac.nz/~paul/R/CM/AdobeSym.html.

See Also

demo(plotmath), axis, mtext, text, title, substitute quote, bquote

Examples

```
require(graphics)

x <- seq(-4, 4, length.out = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))


## How to combine "math" and numeric variables :
plot(1:10, type="n", xlab="", ylab="", main = "plot math & numbers")
theta <- 1.23 ; mtext(bquote(hat(theta) == .(theta)), line= .25)
for(i in 2:9)
    text(i, i+1, substitute(list(xi, eta) == group("(",list(x,y),")"),
                            list(x = i, y = i+1)))
## note that both of these use calls rather than expressions.
##
text(1, 10,  "Derivatives:", adj = 0)
text(1, 9.6, expression(
 "             first: {f * minute}(x) " == {f * minute}(x)), adj = 0)
text(1, 9.0, expression(
 "     second: {f * second}(x) "        == {f * second}(x)), adj = 0)


## note the "{ .. }" trick to get "chained" equations:
plot(1:10, 1:10, main = quote(1 <= {1 < 2}))
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .8)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(4, 6.4, "expression(bar(x) == sum(frac(x[i], n), i==1, n))",
     cex = .8)
text(8, 5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
                            plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})),
     cex = 1.2)

## some other useful symbols
plot.new(); plot.window(c(0,4), c(15,1))
text(1, 1, "universal", adj = 0); text(2.5, 1,  "\\042")
text(3, 1, expression(symbol("\042")))
text(1, 2, "existential", adj = 0); text(2.5, 2,  "\\044")
text(3, 2, expression(symbol("\044")))
text(1, 3, "suchthat", adj = 0); text(2.5, 3,  "\\047")
text(3, 3, expression(symbol("\047")))
text(1, 4, "therefore", adj = 0); text(2.5, 4,  "\\134")
text(3, 4, expression(symbol("\134")))
```

```
text(1, 5, "perpendicular", adj = 0); text(2.5, 5,  "\\136")
text(3, 5, expression(symbol("\136")))
text(1, 6, "circlemultiply", adj = 0); text(2.5, 6,  "\\304")
text(3, 6, expression(symbol("\304")))
text(1, 7, "circleplus", adj = 0); text(2.5, 7,  "\\305")
text(3, 7, expression(symbol("\305")))
text(1, 8, "emptyset", adj = 0); text(2.5, 8,  "\\306")
text(3, 8, expression(symbol("\306")))
text(1, 9, "angle", adj = 0); text(2.5, 9,  "\\320")
text(3, 9, expression(symbol("\320")))
text(1, 10, "leftangle", adj = 0); text(2.5, 10,  "\\341")
text(3, 10, expression(symbol("\341")))
text(1, 11, "rightangle", adj = 0); text(2.5, 11,  "\\361")
text(3, 11, expression(symbol("\361")))
```

---

png *BMP, JPEG, PNG and TIFF graphics devices*

Description

Graphics devices for BMP, JPEG, PNG and TIFF format bitmap files.

Usage

```
bmp(filename = "Rplot%03d.bmp",
    width = 480, height = 480, units = "px", pointsize = 12,
    bg = "white", res = NA, ...,
    type = c("cairo", "Xlib", "quartz"), antialias)

jpeg(filename = "Rplot%03d.jpeg",
     width = 480, height = 480, units = "px", pointsize = 12,
     quality = 75,
     bg = "white", res = NA, ...,
     type = c("cairo", "Xlib", "quartz"), antialias)

png(filename = "Rplot%03d.png",
    width = 480, height = 480, units = "px", pointsize = 12,
    bg = "white",  res = NA, ...,
    type = c("cairo", "cairo-png", "Xlib", "quartz"), antialias)

tiff(filename = "Rplot%03d.tiff",
     width = 480, height = 480, units = "px", pointsize = 12,
     compression = c("none", "rle", "lzw", "jpeg", "zip",
                     "lzw+p", "zip+p",
                     "lerc", "lzma",  "zstd", "webp"),
     bg = "white", res = NA,  ...,
     type = c("cairo", "Xlib", "quartz"), antialias)
```

Arguments

filename	the output file path. The page number is substituted if a C integer format is included in the character string, as in the default. (Depending on the platform, the result should be less than PATH_MAX characters long, and may be truncated if not. See pdf for further details.) Tilde expansion is performed where supported by the platform. An input with a marked encoding is converted to the native encoding on an error is given.
width	the width of the device.
height	the height of the device.
units	The units in which height and width are given. Can be px (pixels, the default), in (inches), cm or mm.
pointsize	the default pointsize of plotted text, interpreted as big points (1/72 inch) at res ppi.
bg	the initial background colour: can be overridden by setting par("bg").
quality	the 'quality' of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.
compression	the type of compression to be used. Can also be a numeric value supported by the underlying libtiff library : see its 'tiff.h' header file. Ignored with a warning for type = "quartz".
res	The nominal resolution in ppi which will be recorded in the bitmap file, if a positive integer. Also used for units other than the default, and to convert points to pixels.
...	for type = "Xlib" only, additional arguments to the underlying X11 device such as fonts or family. For types "cairo" and "quartz", the family argument can be supplied. See the 'Cairo fonts' section in the help for X11. For type "cairo", the symbolfamily argument can be supplied. See X11.options.
type	character string, one of "Xlib" or "quartz" (some macOS builds) or "cairo". The latter will only be available if the system was compiled with support for cairo – otherwise "Xlib" will be used. The default is set by getOption("bitmapType") – the 'out of the box' default is "quartz" or "cairo" where available, otherwise "Xlib".
antialias	for type = "cairo", giving the type of anti-aliasing (if any) to be used for fonts and lines (but not fills). See X11. The default is set by X11.options. Also for type = "quartz", where antialiasing is used unless antialias = "none".

Details

Plots in PNG and JPEG format can easily be converted to many other bitmap formats, and both can be displayed in modern web browsers. The PNG format is lossless and is best for line diagrams and blocks of colour. The JPEG format is lossy, but may be useful for image plots, for example. BMP is a standard format on Windows. TIFF is a meta-format: the default format written by tiff is lossless and stores RGB (and alpha where appropriate) values uncompressed—such files are widely accepted, which is their main virtue over PNG.

The JPEG format only supports opaque backgrounds.

png supports transparent backgrounds: use bg = "transparent". (Not all PNG viewers render files with transparency correctly.) When transparency is in use in the type = "Xlib" variant a very light grey is used as the background and so appears as transparent if used in the plot. This allows opaque white to be used, as in the example. The type = "cairo", type = "cairo-png" and type = "quartz" variants allow semi-transparent colours, including on a transparent or semi-transparent background.

tiff with types "cairo" and "quartz" supports semi-transparent colours, including on a transparent or semi-transparent background. Compression type "zip" is 'deflate (Adobe-style)'. Compression types "lzw+p" and "zip+p" use horizontal differencing ('differencing predictor', section 14 of the TIFF specification) in combination with the compression method, which is effective for continuous-tone images, especially colour ones.

The jpeg quality when used for tiff compression is fixed at 75.

R can be compiled without support for some or all of the types for each of these devices: this will be reported if you attempt to use them on a system where they are not supported. For type = "Xlib" they may not be usable unless the X11 display is available to the owner of the R process. type = "cairo" requires cairo 1.2 or later. type = "quartz" uses the quartz device and so is only available where that is (on some macOS builds: see capabilities("aqua")).

By default no resolution is recorded in the file, except for BMP. Viewers will often assume a nominal resolution of 72 ppi when none is recorded. As resolutions in PNG files are recorded in pixels/metre, the reported ppi value will be changed slightly.

For graphics parameters that make use of dimensions in inches (including font sizes in points) the resolution used is res (or 72 ppi if unset).

png will normally use a palette if there are less than 256 colours on the page, and record a 24-bit RGB file otherwise (or a 32-bit ARGB file if type = "cairo" and non-opaque colours are used). However, type = "cairo-png" uses cairographics' PNG backend which will never use a palette and normally creates a larger 32-bit ARGB file—this may work better for specialist uses with semi-transparent colours.

Quartz-produced PNG and TIFF plots with a transparent background are recorded with a dark grey matte which will show up in some viewers, including Preview on macOS.

Unknown resolutions in BMP files are recorded as 72 ppi.

Value

A plot device is opened: nothing is returned to the R interpreter.

Warnings

Note that by default the width and height values are in pixels not inches. A warning will be issued if both are less than 20.

If you plot more than one page on one of these devices and do not include something like %d for the sequence number in file, the file will contain the last page plotted.

Differences between OSes

These functions are interfaces to three or more different underlying devices.

- On Windows, devices based on plotting to a hidden screen using Windows' GDI calls.
- On platforms with support for X11, plotting to a hidden X11 display.
- On macOS when working at the console and when R is compiled with suitable support, using Apple's Quartz plotting system.
- Where support has been compiled in for cairographics, plotting on cairo surfaces. This may use the native platform support for fonts, or it may use `fontconfig` to support a wide range of font formats.

Inevitably there will be differences between the options supported and output produced. Perhaps the most important are support for antialiased fonts and semi-transparent colours: the best results are likely to be obtained with the cairo- or Quartz-based devices where available.

The default extensions are '`.jpg`' and '`.tif`' on Windows, and '`.jpeg`' and '`.tiff`' elsewhere.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is in pixels.
- Font sizes are in big points interpreted at `res` ppi.
- The default font family is Helvetica.
- Line widths in 1/96 inch (interpreted at `res` ppi), minimum one pixel for `type = "Xlib"`, 0.01 for `type = "cairo"`.
- For `type = "Xlib"` circle radii are in pixels with minimum one.
- Colours are interpreted by the viewing application.

For `type = "quartz"` see the help for `quartz`.

Note

For `type = "Xlib"` these devices are based on the `X11` device. The colour model used will be that set up by `X11.options` at the time the first Xlib-based devices was opened (or the first after all such devices have been closed).

Support for compression types depends on the underlying '`libtiff`' library: types `"lerc"`, `"lzma"`, `"zstd"` and `"webp"` are relatively recent additions and may well not be supported. They are also liable to be unsupported in TIFF viewers.

Author(s)

Guido Masarotto and Brian Ripley

References

The PNG specification, https://www.w3.org/TR/png/.

The TIFF specification, https://www.iso.org/standard/34342.html. See also https://en.wikipedia.org/wiki/TIFF.

See Also

Devices, dev.print

capabilities to see if these devices are supported by this build of R, and if type = "cairo" is supported.

bitmap provides an alternative way to generate plots in many bitmap formats that does not depend on accessing the X11 display but does depend on having GhostScript installed.

Ways to write raster images to bitmap formats are available in packages **jpeg**, **png** and **tiff**.

Examples

```
## these examples will work only if the devices are available
## and cairo or an X11 display or a macOS display is available.

## copy current plot to a (large) PNG file
## Not run: dev.print(png, filename = "myplot.png", width = 1024, height = 768)

png("myplot.png", bg = "transparent")
plot(1:10)
rect(1, 5, 3, 7, col = "white")
dev.off()

## will make myplot1.jpeg and myplot2.jpeg
jpeg("myplot%d.jpeg")
example(rect)
dev.off()
```

---

postscript *PostScript Graphics*

Description

postscript starts the graphics device driver for producing PostScript graphics.

Usage

```
postscript(file = if(onefile) "Rplots.ps" else "Rplot%03d.ps",
           onefile, family, title, fonts, encoding, bg, fg,
           width, height, horizontal, pointsize,
           paper, pagecentre, print.it, command,
           colormodel, useKerning, fillOddEven)
```

Arguments

file	a character string giving the file path. If it is "", the output is piped to the command given by the argument command. If it is of the form "|cmd", the output is piped to the command given by cmd.

For use with `onefile = FALSE`, give a C integer format such as `"Rplot%03d.ps"` (the default in that case). The string should not otherwise contain a `%`: if it is really necessary, use `%%` in the string for `%` in the file name. A single integer format matching the regular expression `"%[#0 +=-]*[0-9.]*[diouxX]"` is allowed.

Tilde expansion (see `path.expand`) is done. An input with a marked encoding is converted to the native encoding or an error is given.

See also section 'File specifications' in the help for `pdf` for further details.

`onefile` logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number for each page and use an EPSF header and no `DocumentMedia` comment. Defaults to `TRUE`.

`family` the initial font family to be used, see the section 'Families' in `pdf`. Defaults to `"Helvetica"`.

`title` title string to embed as the `Title` comment in the file. Defaults to `"R Graphics Output"`.

`fonts` a character vector specifying additional R graphics font family names for font families whose declarations will be included in the PostScript file and are available for use with the device. See 'Families' below. Defaults to `NULL`.

`encoding` the name of an encoding file. See `pdf` for details. Defaults to `"default"`.

`bg` the initial background color to be used. If `"transparent"` (or any other non-opaque colour), no background is painted. Defaults to `"transparent"`.

`fg` the initial foreground color to be used. Defaults to `"black"`.

`width, height` the width and height of the graphics region in inches. Default to `0`.

If `paper != "special"` and `width` or `height` is less than `0.1` or too large to give a total margin of 0.5 inch, the graphics region is reset to the corresponding paper dimension minus 0.5.

`horizontal` the orientation of the printed image, a logical. Defaults to true, that is landscape orientation on paper sizes with width less than height.

`pointsize` the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points. Defaults to `12`.

`paper` the size of paper in the printer. The choices are `"a4"`, `"letter"` (or `"us"`), `"legal"` and `"executive"` (and these can be capitalized). Also, `"special"` can be used, when arguments `width` and `height` specify the paper size. A further choice is `"default"` (the default): If this is selected, the paper size is taken from the option `"papersize"` if that is set and to `"a4"` if it is unset or empty.

`pagecentre` logical: should the device region be centred on the page? Defaults to true.

`print.it` logical: should the file be printed when the device is closed? (This only applies if `file` is a real file name.) Defaults to false.

`command` the command to be used for 'printing'. Defaults to `"default"`, the value of option `"printcmd"`. The length limit is `2*PATH_MAX`, typically 8096 bytes on Unix-alikes and 520 bytes on Windows. Recent Windows systems may be configured to use long paths, raising this limit currently to 10000.

`colormodel` a character string describing the color model: currently allowed values as `"srgb"`, `"srgb+gray"`, `"rgb"`, `"rgb-nogray"`, `"gray"` (or `"grey"`) and `"cmyk"`. Defaults to `"srgb"`. See section 'Color models'.

useKerning	logical. Should kerning corrections be included in setting text and calculating string widths? Defaults to TRUE.
fillOddEven	logical controlling the polygon fill mode: see polygon for details. Default FALSE.

Details

All arguments except file default to values given by ps.options(). The ultimate defaults are quoted in the arguments section.

postscript opens the file file and the PostScript commands needed to plot any graphics requested are written to that file. This file can then be printed on a suitable device to obtain hard copy.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.ps', ..., 'Rplot999.ps', 'Rplot1000.ps', ...

The postscript produced for a single R plot is EPS (*Encapsulated PostScript*) compatible, and can be included into other documents, e.g., into LaTeX, using '\includegraphics{<filename>}'. For use in this way you will probably want to use setEPS() to set the defaults as horizontal = FALSE, onefile = FALSE, paper = "special". Note that the bounding box is for the *device* region: if you find the white space around the plot region excessive, reduce the margins of the figure region via par(mar =).

Most of the PostScript prologue used is taken from the R character vector .ps.prolog. This is marked in the output, and can be changed by changing that vector. (This is only advisable for PostScript experts: the standard version is in namespace:grDevices.)

A PostScript device has a default family, which can be set by the user via family. If other font families are to be used when drawing to the PostScript device, these must be declared when the device is created via fonts; the font family names for this argument are R graphics font family names (see the documentation for postscriptFonts).

Line widths as controlled by par(lwd =) are in multiples of 1/96 inch: multiples less than 1 are allowed. pch = "." with cex = 1 corresponds to a square of side 1/72 inch, which is also the 'pixel' size assumed for graphics parameters such as "cra".

When the background colour is fully transparent (as is the initial default value), the PostScript produced does not paint the background. Almost all PostScript viewers will use a white canvas so the visual effect is if the background were white. This will not be the case when printing onto coloured paper, though.

TeX fonts

TeX has traditionally made use of fonts such as Computer Modern which are encoded rather differently, in a 7-bit encoding. This encoding can be specified by encoding = "TeXtext.enc", taking care that the ASCII characters < > \ _ { } are not available in those fonts.

There are supplied families "ComputerModern" and "ComputerModernItalic" which use this encoding, and which are only supported for postscript (and not pdf). They are intended to use with the Type 1 versions of the TeX CM fonts. It will normally be possible to include such output in TeX or LaTeX provided it is processed with dvips -Ppfb -j0 or the equivalent on your system. (-j0 turns off font subsetting.) When family = "ComputerModern" is used, the italic/bold-italic fonts used are slanted fonts (cmsl10 and cmbxsl10). To use text italic fonts instead, set family = "ComputerModernItalic".

These families use the TeX math italic and symbol fonts for a comprehensive but incomplete coverage of the glyphs covered by the Adobe symbol font in other families. This is achieved by special-casing the postscript code generated from the supplied 'CM_symbol_10.afm'.

Color models

The default color model ("srgb") is sRGB.

The alternative "srgb+gray" uses sRGB for colors, but with pure gray colors (including black and white) expressed as greyscales (which results in smaller files and can be advantageous with some printer drivers). Conversely, its files can be rendered much slower on some viewers, and there can be a noticeable discontinuity in color gradients involving gray or white.

Other possibilities are "gray" (or "grey") which used only greyscales (and converts other colours to a luminance), and "cmyk". The simplest possible conversion from sRGB to CMYK is used (https://en.wikipedia.org/wiki/CMYK_color_model#Mapping_RGB_to_CMYK), and raster images are output in RGB.

Color models provided for backwards compatibility are "rgb" (which is RGB+gray) and "rgb-nogray" which use uncalibrated RGB (as used in R prior to 2.13.0). These result in slightly smaller files which may render faster, but do rely on the viewer being properly calibrated.

Printing

A postscript plot can be printed via postscript in two ways.

1. Setting print.it = TRUE causes the command given in argument command to be called with argument "file" when the device is closed. Note that the plot file is not deleted unless command arranges to delete it.
2. file = "" or file = "|cmd" can be used to print using a pipe. Failure to open the command will probably be reported to the terminal but not to R, in which case close the device by dev.off immediately.

On Windows the default "printcmd" is empty and will give an error if print.it = TRUE is used. Suitable commands to spool a PostScript file to a printer can be found in 'RedMon' suite available from http://pages.cs.wisc.edu/~ghost/index.html. The command will be run in a minimized window. GSView 4.x provides 'gsprint.exe' which may be more convenient (it requires Ghostscript version 6.50 or later).

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are as a multiple of 1/96 inch, with a minimum of 0.01 enforced.
- Circle of any radius are allowed.
- Colours are by default specified as sRGB.

At very small line widths, the line type may be forced to solid.

Raster images are currently limited to opaque colours.

Note

If you see problems with postscript output, do remember that the problem is much more likely to be in your viewer than in R. Try another viewer if possible. Symptoms for which the viewer has been at fault are apparent grids on image plots (turn off graphics anti-aliasing in your viewer if you can) and missing or incorrect glyphs in text (viewers silently doing font substitution).

Unfortunately the default viewers on most Linux and macOS systems have these problems, and no obvious way to turn off graphics anti-aliasing.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso <durso@hussle.harvard.edu>.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

postscriptFonts, Devices, and check.options which is called from both ps.options and postscript.

cairo_ps for another device that can produce PostScript.

More details of font families and encodings and especially handling text in a non-Latin-1 encoding and embedding fonts can be found in

Paul Murrell and Brian Ripley (2006). "Non-standard fonts in PostScript and PDF graphics." *R News*, **6**(2), 41–47. https://www.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

Examples

```
require(graphics)
## Not run:
# open the file "foo.ps" for graphics output
postscript("foo.ps")
# produce the desired graph(s)
dev.off()              # turn off the postscript device

## On Unix-alikes only:
postscript("|lp -dlw")
# produce the desired graph(s)
dev.off()              # plot will appear on printer

## On Windows:
options(printcmd = 'redpr -P"\\printhost\lw"')
postscript(file = tempfile("Rps."), print.it = TRUE)
```

```
# produce the desired graph(s)
dev.off()              # send plot file to the printer
## alternative using GSView 4.x :
options(printcmd = '/GhostGum/gsview/gsprint -query')


# for URW PostScript devices
postscript("foo.ps", family = "NimbusSan")

## for inclusion in Computer Modern TeX documents, perhaps
postscript("cm_test.eps", width = 4.0, height = 3.0,
           horizontal = FALSE, onefile = FALSE, paper = "special",
           family = "ComputerModern", encoding = "TeXtext.enc")
## The resultant postscript file can be used by dvips -Ppfb -j0.

## To test out encodings, you can use
TestChars <- function(encoding = "ISOLatin1", family = "URWHelvetica")
{
    postscript(encoding = encoding, family = family)
    par(pty = "s")
    plot(c(-1,16), c(-1,16), type = "n", xlab = "", ylab = "",
         xaxs = "i", yaxs = "i")
    title(paste("Centred chars in encoding", encoding))
    grid(17, 17, lty = 1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch = i)
    }
    dev.off()
}
## there will be many warnings.  We use URW to get a complete enough
## set of font metrics.
TestChars()
TestChars("ISOLatin2")
TestChars("WinAnsi")

## End(Not run)
```

---

postscriptFonts *PostScript and PDF Font Families*

Description

These functions handle the translation of a R graphics font family name to a PostScript or PDF font description, used by the postscript or pdf graphics devices.

Usage

```
postscriptFonts(...)
pdfFonts(...)
```

Arguments

...	either character strings naming mappings to display, or named arguments specifying mappings to add or change.

Details

If these functions are called with no argument they list all the existing mappings, whereas if they are called with named arguments they add (or change) mappings.

A PostScript or PDF device is created with a default font family (see the documentation for `postscript`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `"family"` in `par` and for `"fontfamily"` in `gpar` in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to a set of PostScript fonts. Separate lists of mappings for `postscript` and `pdf` devices are maintained for the current R session and can be added to by the user.

The `postscriptFonts` and `pdfFonts` functions can be used to list existing mappings and to define new mappings. The `Type1Font` and `CIDFont` functions can be used to create new mappings, when the `xxxFonts` function is used to add them to the database. See the examples.

Default mappings are provided for three device-independent family names: `"sans"` for a sans-serif font (to `"Helvetica"`), `"serif"` for a serif font (to `"Times"`) and `"mono"` for a monospaced font (to `"Courier"`).

Mappings for a number of standard Adobe fonts (and URW equivalents) are also provided: `"AvantGarde"`, `"Bookman"`, `"Courier"`, `"Helvetica"`, `"Helvetica-Narrow"`, `"NewCenturySchoolbook"`, `"Palatino"` and `"Times"`; `"URWGothic"`, `"URWBookman"`, `"NimbusMon"`, `"NimbusSan"` (synonym `"URWHelvetica"`), `"NimbusSanCond"`, `"CenturySch"`, `"URWPalladio"` and `"NimbusRom"` (synonym `"URWTimes"`).

There are also mappings for `"ComputerModern"`, `"ComputerModernItalic"` and `"ArialMT"` (Monotype Arial).

Finally, there are some default mappings for East Asian locales described in a separate section.

The specification of font metrics and encodings is described in the help for the `postscript` function.

The fonts are not embedded in the resulting PostScript or PDF file, so software including the PostScript or PDF plot file should either embed the font outlines (usually from '`.pfb`' or '`.pfa`' files) or use DSC comments to instruct the print spooler or including application to do so (see also `embedFonts`).

A font family has both an R-level name, the argument name used when `postscriptFonts` was called, and an internal name, the `family` component. These two names are the same for all the pre-defined font families.

Once a font family is in use it cannot be changed. 'In use' means that it has been specified *via* a `family` or `fonts` argument to an invocation of the same graphics device already in the R session.

(For these purposes `xfig` counts the same as `postscript` but only uses some of the predefined mappings.)

Value

A list of one or more font mappings.

East Asian fonts

There are some default mappings for East Asian locales:
`"Japan1"`, `"Japan1HeiMin"`, `"Japan1GothicBBB"`, and `"Japan1Ryumin"` for Japanese; `"Korea1"` and `"Korea1deb"` for Korean; `"GB1"` (Simplified Chinese) for mainland China and Singapore; `"CNS1"` (Traditional Chinese) for Hong Kong and Taiwan.

These refer to the following fonts

Japan1 (PS)	`HeiseiKakuGo-W5`
	Linotype Japanese printer font
Japan1 (PDF)	`KozMinPro-Regular-Acro`
	from Adobe Reader 7.0 Japanese Font Pack
Japan1HeiMin (PS)	`HeiseiMin-W3`
	Linotype Japanese printer font
Japan1HeiMin (PDF)	`HeiseiMin-W3-Acro`
	from Adobe Reader 7.0 Japanese Font Pack
Japan1GothicBBB	`GothicBBB-Medium`
	Japanese-market PostScript printer font
Japan1Ryumin	`Ryumin-Light`
	Japanese-market PostScript printer font
Korea1 (PS)	`Baekmuk-Batang`
	TrueType font found on some Linux systems
Korea1 (PDF)	`HYSMyeongJoStd-Medium-Acro`
	from Adobe Reader 7.0 Korean Font Pack
Korea1deb (PS)	`Batang-Regular`
	another name for Baekmuk-Batang
Korea1deb (PDF)	`HYGothic-Medium-Acro`
	from Adobe Reader 4.0 Korean Font Pack
GB1 (PS)	`BousungEG-Light-GB`
	TrueType font found on some Linux systems
GB1 (PDF)	`STSong-Light-Acro`
	from Adobe Reader 7.0 Simplified Chinese Font Pack
CNS1 (PS)	`MOESung-Regular`
	Ken Lunde's CJKV resources
CNS1 (PDF)	`MSungStd-Light-Acro`
	from Adobe Reader 7.0 Traditional Chinese Font Pack

`BousungEG-Light-GB` can be found at https://ftp.gnu.org/pub/non-gnu/chinese-fonts-truetype/. These will need to be installed or otherwise made available to the PostScript/PDF interpreter such as Ghostscript (and not all interpreters can handle TrueType fonts).

You may well find that your postscript/PDF interpreters has been set up to provide aliases for many of these fonts. For example, Ghostscript on Windows can optionally be installed to map common East Asian fonts names to Windows TrueType fonts. (You may want to add the `-Acro` versions as well.)

Adding a mapping for a CID-keyed font is for gurus only.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso.

See Also

`postscript` and `pdf`; `Type1Font` and `CIDFont` for specifying new font mappings.

Examples

```
postscriptFonts()
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                         "CM_symbol_10.afm"),
                      encoding = "TeXtext.enc")
postscriptFonts(CMitalic = CMitalic)

## A CID font for Japanese using a different CMap and
## corresponding cmapEncoding.
`Jp_UCS-2` <- CIDFont("TestUCS2",
                  c("Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm"),
                  "UniJIS-UCS2-H", "UCS-2")
pdfFonts(`Jp_UCS-2` = `Jp_UCS-2`)
names(pdfFonts())
```

---

pretty.Date *Pretty Breakpoints for Date-Time Classes*

Description

Compute a sequence of about `n+1` equally spaced 'nice' values which cover the range of the values in `x`, possibly of length one, when `min.n = 0` and there is only one unique `x`.

Usage

```
## S3 method for class 'Date'
pretty(x, n = 5, min.n = n %/% 2, sep = " ", ...)
## S3 method for class 'POSIXt'
pretty(x, n = 5, min.n = n %/% 2, sep = " ", ...)
```

Arguments

x	an object of class "Date" or "POSIXt" (i.e., "POSIXct" or "POSIXlt").
n	integer giving the *desired* number of intervals.
min.n	nonnegative integer giving the *minimal* number of intervals.
sep	character string, serving as a separator for certain formats (e.g., between month and year).
...	further arguments for compatibility with the generic, ignored.

Value

A vector (of the suitable class) of locations, with attribute "labels" giving corresponding formatted character labels and attribute "format" giving the format specification that was used.

See Also

pretty for the default method.

Examples

```
pretty(Sys.Date())
pretty(Sys.time(), n = 10)

pretty(as.Date("2000-03-01")) # R 1.0.0 came in a leap year

## time ranges in diverse scales:% also in ../../../../tests/reg-tests-1c.R
steps <- stats::setNames(,
    c("10 secs", "1 min", "5 mins", "30 mins", "6 hours", "12 hours",
      "1 DSTday", "2 weeks", "1 month", "6 months", "1 year",
      "10 years", "50 years", "1000 years"))
x <- as.POSIXct("2002-02-02 02:02")
lapply(steps,
       function(s) {
           at <- pretty(seq(x, by = s, length.out = 2), n = 5)
           attr(at, "labels")
       })
```

---

ps.options *Auxiliary Function to Set/View Defaults for Arguments of postscript*

Description

The auxiliary function ps.options can be used to set or view (if called without arguments) the default values for some of the arguments to postscript.

ps.options needs to be called before calling postscript, and the default values it sets can be overridden by supplying arguments to postscript.

Usage

```
ps.options(..., reset = FALSE, override.check = FALSE)

setEPS(...)
setPS(...)
```

Arguments

`...`	arguments `onefile`, `family`, `title`, `fonts`, `encoding`, `bg`, `fg`, `width`, `height`, `horizontal`, `pointsize`, `paper`, `pagecentre`, `print.it`, `command`, `colormodel` and `fillOddEven` can be supplied. `onefile`, `horizontal` and `paper` are *ignored* for `setEPS` and `setPS`.
`reset`	logical: should the defaults be reset to their 'factory-fresh' values?
`override.check`	logical argument passed to `check.options`. See the Examples.

Details

If both `reset = TRUE` and `...` are supplied the defaults are first reset to the 'factory-fresh' values and then the new values are applied.

For backwards compatibility argument `append` is accepted but ignored with a warning.

`setEPS` and `setPS` are wrappers to set defaults appropriate for figures for inclusion in documents (the default size is 7 inches square unless `width` or `height` is supplied) and for spooling to a PostScript printer respectively. For historical reasons the latter is the ultimate default.

Value

A named list of all the previous defaults. If `...` or `reset = TRUE` is supplied the result has the visibility flag turned off.

See Also

`postscript`, `pdf.options`

Examples

```
ps.options(bg = "pink")
utils::str(ps.options())

### ---- error checking of arguments: ----
ps.options(width = 0:12, onefile = 0, bg = pi)
# override the check for 'width', but not 'bg':
ps.options(width = 0:12, bg = pi, override.check = c(TRUE,FALSE))
utils::str(ps.options())
ps.options(reset = TRUE) # back to factory-fresh
```

quartz *macOS Quartz Device*

Description

quartz starts a graphics device driver for the macOS system. It supports plotting both to the screen (the default) and to various graphics file formats.

Usage

```
quartz(title, width, height, pointsize, family, antialias, type,
      file = NULL, bg, canvas, dpi)

quartz.options(..., reset = FALSE)

quartz.save(file, type = "png", device = dev.cur(), dpi = 100, ...)
```

Arguments

title	title for the Quartz window (applies to on-screen output only), default "Quartz %d". A C-style format for an integer will be substituted by the device number (see the file argument to pdf for further details).
width	the width of the plotting area in inches. Default 7.
height	the height of the plotting area in inches. Default 7.
pointsize	the default pointsize to be used. Default 12.
family	this is the family name of the font that will be used by the device. Default "Arial". This will be the base name of a font as shown in Font Book.
antialias	whether to use antialiasing. Default TRUE.
type	the type of output to use. See 'Details' for more information. Default "native".
file	an optional target for the graphics device. The default, NULL, selects a default name where one is needed. See 'Details' for more information.
bg	the initial background colour to use for the device. Default "transparent". An opaque colour such as "white" will normally be required on off-screen types that support transparency such as "png" and "tiff".
canvas	canvas colour to use for an on-screen device. Default "white", and will be forced to be an opaque colour.
dpi	resolution of the output. The default (NA_real_) for an on-screen display defaults to the resolution of the main screen, and to 72 dpi otherwise. See 'Details'.
...	Any of the arguments to quartz except file.
reset	logical: should the defaults be reset to their defaults?
device	device number to copy from.

Details

The defaults for all but one of the arguments of quartz are set by quartz.options: the 'Arguments' section gives the 'factory-fresh' defaults.

The Quartz graphics device supports a variety of output types. On-screen output types are "" or "native" or "Cocoa". Off-screen output types produce output files and utilize the file argument. type = "pdf" gives PDF output. The following bitmap formats may be supported (depending on the OS version): "png", "jpeg", "jpg", "jpeg2000", "tif", "tiff", "gif", "psd" (Adobe Photoshop), "bmp" (Windows bitmap), "sgi" and "pict".

The file argument is used for off-screen drawing. The actual file is only created when the device is closed (e.g., using dev.off()). For the bitmap devices, the page number is substituted if a C integer format is included in the character string, e.g. Rplot%03d.png. (Depending on the platform, the result should be less than PATH_MAX characters long, and may be truncated if not. See pdf for further details.) If a file argument is not supplied, the default is Rplots.pdf or Rplot%03d.*type*. Tilde expansion (see path.expand) is done.

If a device-independent R graphics font family is specified (e.g., via par(family =) in the graphics package), the Quartz device makes use of the Quartz font database (see quartzFonts) to convert the R graphics font family to a Quartz-specific font family description. The default conversions are (MonoType TrueType versions of) Helvetica for sans, Times-Roman for serif and Courier for mono.

On-screen devices are launched with a semi-transparent canvas. Once a new plot is created, the canvas is first painted with the canvas colour and then the current background colour (which can be transparent or semi-transparent). Off-screen devices have no canvas colour, and so start with a transparent background where possible (e.g., type = "png" and type = "tiff") – otherwise it appears that a solid white canvas is assumed in the Quartz code. PNG and TIFF files are saved with a dark grey matte which will show up in some viewers, including Preview.

title can be used for on-screen output. It must be a single character string with an optional integer printf-style format that will be substituted by the device number. It is also optionally used (without a format) to give a title to a PDF file.

Calling quartz() sets .Device to "quartz" for on-screen devices and to "quartz_off_screen" otherwise.

The font family chosen needs to cover the characters to be used: characters not in the font are rendered as empty oblongs. For non-Western-European languages something other than the default of "Arial" is likely to be needed—one choice for Chinese is "MingLiU".

quartz.save is a modified version of dev.copy2pdf to copy the plot from the current screen device to a quartz device, by default to a PNG file.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Arial.
- Line widths are a multiple of 1/96 inch with no minimum set by R.

- Circle radii are real-valued with no minimum set by R.
- Colours are specified as sRGB.

Note

For a long time the default font family was documented as "Helvetica" after it had been changed to "Arial" to work around a deficiency in macOS 10.4. It may be changed back in future.

A fairly common Mac problem is no text appearing on plots due to corrupted or duplicated fonts on your system. You may be able to confirm this by using another font family, e.g. family = "serif". Open the Font Book application (in Applications) and check the fonts that you are using.

See Also

quartzFonts, Devices.

png for way to access the bitmap types of this device via R's standard bitmap devices.

Examples

```
## Not run:
## Only on a Mac,
## put something like this is your .Rprofile to customize the defaults
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::quartz.options(width = 8, height = 6,
                                                pointsize = 10))

## End(Not run)
```

---

quartzFonts *Quartz Fonts Setup*

Description

These functions handle the translation of a device-independent R graphics font family name to a quartz font description.

They are only available on Unix-alikes, i.e, not on Windows, and typically used on the Mac.

Usage

```
quartzFont(family)

quartzFonts(...)
```

Arguments

family	a character vector containing the four PostScript font names for plain, bold, italic, and bold italic versions of a font family.
...	either character strings naming mappings to display, or new (named) mappings to define.

Details

A quartz device is created with a default font (see the documentation for quartz), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for gpar in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to quartz fonts. A list of mappings is maintained and can be modified by the user.

The quartzFonts function can be used to list existing mappings and to define new mappings. The quartzFont function can be used to create a new mapping.

Default mappings are provided for three device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font and "mono" for a monospaced font.

See Also

quartz for the default Mac graphics device.

Examples

```
if(.Platform$OS.type == "unix") { # includes macOS

  utils::str( quartzFonts() )  # a list(serif = .., sans = .., mono = ..)
  quartzFonts("mono")          # the list(mono = ..) sublist of quartzFonts()

## Not run:
  ## for East Asian locales you can use something like
  quartzFonts(sans = quartzFont(rep("AppleGothic", 4)),
       serif = quartzFont(rep("AppleMyungjp", 4)))
  ## since the default fonts may well not have the glyphs needed

## End(Not run)
}
```

---

recordGraphics *Record Graphics Operations*

Description

Records arbitrary code on the graphics engine display list. Useful for encapsulating calculations with graphical output that depends on the calculations. Intended *only* for expert use.

Usage

```
recordGraphics(expr, list, env)
```

Arguments

expr	object of mode expression or call or an unevaluated expression.
list	a list defining the environment in which expr is to be evaluated.
env	an environment specifying where R looks for objects not found in list.

Details

The code in `expr` is evaluated in an environment constructed from `list`, with `env` as the parent of that environment.

All three arguments are saved on the graphics engine display list so that on a device resize or copying between devices, the original evaluation environment can be recreated and the code can be re-evaluated to reproduce the graphical output.

Value

The value from evaluating `expr`.

Warning

This function is not intended for general use. Incorrect or improper use of this function could lead to unintended and/or undesirable results.

An example of acceptable use is querying the current state of a graphics device or graphics system setting and then calling a graphics function.

An example of improper use would be calling the `assign` function to performing assignments in the global environment.

See Also

`eval`

Examples

```
require(graphics)

plot(1:10)
# This rectangle remains 1inch wide when the device is resized
recordGraphics(
  {
    rect(4, 2,
         4 + diff(par("usr")[1:2])/par("pin")[1], 3)
  },
  list(),
  getNamespace("graphics"))
```

---

recordPlot *Record and Replay Plots*

Description

Functions to save the current plot in an R variable, and to replay it.

Usage

```
recordPlot(load=NULL, attach=NULL)
replayPlot(x, reloadPkgs=FALSE)
```

Arguments

load	If not NULL, a character vector of package names, which are saved as part of the recorded plot.
attach	If not NULL, a character vector of package names, which are saved as part of the recorded plot.
x	A saved plot.
reloadPkgs	A logical indicating whether to reload and/or reattach any packages that were saved as part of the recorded plot.

Details

These functions record and replay the display list of the current graphics device. The returned object is of class "recordedplot", and replayPlot acts as a print method for that class.

The returned object is stored as a pairlist, but the usual methods for examining R objects such as deparse and str are liable to mislead.

Value

recordPlot returns an object of class "recordedplot".

replayPlot has no return value.

Warning

The format of recorded plots may change between R versions, so recorded plots should **not** be used as a permanent storage format for R plots.

As of R 3.3.0, it is possible (again) to replay a plot from another R session using, for example, saveRDS and readRDS. It is even possible to replay a plot from another R version, however, this will produce warnings, may produce errors, or something worse.

Note

Replay of a recorded plot may not produce the correct result (or may just fail) if the display list contains a call to recordGraphics which in turn contains an expression that calls code from a non-base package other than **graphics** or **grid**. The most well-known example of this is a plot drawn with the package **ggplot2**. One solution is to load the relevant package(s) before replaying the recorded plot. The load and attach arguments to recordPlot can be used to automate this - any packages named in load will be reloaded, via loadNamespace, and any packages named in attach will be reattached, via library, as long as reloadPkgs is TRUE in the call to replayPlot. This is only relevant when attempting to replay in one R session a plot that was recorded in a different R session.

References

Murrell, P., Ooms, J., Allaire, J.J. (2015), "Recording and Replaying the Graphics Engine Display List", https://stattech.wordpress.fos.auckland.ac.nz/2015/12/21/2015-07-recording-and-replaying-the-graphics-engine-display-list/

See Also

The display list can be turned on and off using dev.control. Initially recording is on for screen devices, and off for print devices.

---

rgb *RGB Color Specification*

Description

This function creates colors corresponding to the given intensities (between 0 and max) of the red, green and blue primaries. The colour specification refers to the standard sRGB colorspace (IEC standard 61966).

An alpha transparency value can also be specified (as an opacity, so 0 means fully transparent and max means opaque). If alpha is not specified, an opaque colour is generated.

The names argument may be used to provide names for the colors.

The values returned by these functions can be used with a col= specification in graphics functions or in par.

Usage

```
rgb(red, green, blue, alpha, names = NULL, maxColorValue = 1)
```

Arguments

red, blue, green, alpha	numeric vectors with values in $[0, M]$ where M is maxColorValue. When this is 255, the red, blue, green, and alpha values are coerced to integers in 0:255 and the result is computed most efficiently.
names	character vector. The names for the resulting vector.
maxColorValue	number giving the maximum of the color values range, see above.

Details

The colors may be specified by passing a matrix or data frame as argument red, and leaving blue and green missing. In this case the first three columns of red are taken to be the red, green and blue values.

Semi-transparent colors (0 < alpha < 1) are supported only on some devices: at the time of writing on the pdf, windows, quartz and X11(type = "cairo") devices and associated bitmap devices (jpeg, png, bmp, tiff and bitmap). They are supported by several third-party devices such as those

in packages **Cairo**, **cairoDevice** and **JavaGD**. Only some of these devices support semi-transparent backgrounds.

Most other graphics devices plot semi-transparent colors as fully transparent, usually with a warning when first encountered.

NA values are not allowed for any of red, blue, green or alpha.

Value

A character vector with elements of 7 or 9 characters, beginning with "#" followed by the red, blue, green and optionally alpha values in hexadecimal (after rescaling to 0 ... 255). The optional alpha values range from 0 (fully transparent) to 255 (opaque).

R does **not** use 'premultiplied alpha'.

See Also

col2rgb for translating R colors to RGB vectors; rainbow, hsv, hcl, gray.

Examples

```
rgb(0, 1, 0)

rgb((0:15)/15, green = 0, blue = 0, names = paste("red", 0:15, sep = "."))

rgb(0, 0:12, 0, maxColorValue = 255) # integer input

ramp <- colorRamp(c("red", "white"))
rgb( ramp(seq(0, 1, length.out = 5)), maxColorValue = 255)
```

---

rgb2hsv *RGB to HSV Conversion*

Description

rgb2hsv transforms colors from RGB space (red/green/blue) into HSV space (hue/saturation/value).

Usage

```
rgb2hsv(r, g = NULL, b = NULL, maxColorValue = 255)
```

Arguments

r	vector of 'red' values in $[0, M]$, (M =maxColorValue) or 3-row RGB matrix.
g	vector of 'green' values, or NULL when r is a matrix.
b	vector of 'blue' values, or NULL when r is a matrix.
maxColorValue	number giving the maximum of the RGB color values range. The default 255 corresponds to the typical 0:255 RGB coding as in col2rgb().

Details

Value (brightness) gives the amount of light in the color.
Hue describes the dominant wavelength.
Saturation is the amount of Hue mixed into the color.

An HSV colorspace is relative to an RGB colorspace, which in R is sRGB, which has an implicit gamma correction.

Value

A matrix with a column for each color. The three rows of the matrix indicate hue, saturation and value and are named "h", "s", and "v" accordingly.

Author(s)

R interface by Wolfram Fischer <wolfram@fischer-zim.ch>;
C code mainly by Nicholas Lewin-Koh <nikko@hailmail.net>.

See Also

hsv, col2rgb, rgb.

Examples

```
## These (saturated, bright ones) only differ by hue
(rc <- col2rgb(c("red", "yellow","green","cyan", "blue", "magenta")))
(hc <- rgb2hsv(rc))
6 * hc["h",] # the hues are equispaced


(rgb3 <- floor(256 * matrix(stats::runif(3*12), 3, 12)))
(hsv3 <- rgb2hsv(rgb3))
## Consistency :
stopifnot(rgb3 == col2rgb(hsv(h = hsv3[1,], s = hsv3[2,], v = hsv3[3,])),
          all.equal(hsv3, rgb2hsv(rgb3/255, maxColorValue = 1)))

## A (simplified) pure R version -- originally by Wolfram Fischer --
## showing the exact algorithm:
rgb2hsvR <- function(rgb, gamma = 1, maxColorValue = 255)
{
    if(!is.numeric(rgb)) stop("rgb matrix must be numeric")
    d <- dim(rgb)
    if(d[1] != 3) stop("rgb matrix must have 3 rows")
    n <- d[2]
    if(n == 0) return(cbind(c(h = 1, s = 1, v = 1))[,0])
    rgb <- rgb/maxColorValue
    if(gamma != 1) rgb <- rgb ^ (1/gamma)

    ## get the max and min
    v <- apply( rgb, 2, max)
    s <- apply( rgb, 2, min)
    D <- v - s # range
```

```
    ## set hue to zero for undefined values (gray has no hue)
    h <- numeric(n)
    notgray <- ( s != v )

    ## blue hue
    idx <- (v == rgb[3,] & notgray )
    if (any (idx))
        h[idx] <- 2/3 + 1/6 * (rgb[1,idx] - rgb[2,idx]) / D[idx]
    ## green hue
    idx <- (v == rgb[2,] & notgray )
    if (any (idx))
        h[idx] <- 1/3 + 1/6 * (rgb[3,idx] - rgb[1,idx]) / D[idx]
    ## red hue
    idx <- (v == rgb[1,] & notgray )
    if (any (idx))
        h[idx] <-       1/6 * (rgb[2,idx] - rgb[3,idx]) / D[idx]

    ## correct for negative red
    idx <- (h < 0)
    h[idx] <- 1+h[idx]

    ## set the saturation
    s[! notgray] <- 0;
    s[notgray] <- 1 - s[notgray] / v[notgray]

    rbind( h = h, s = s, v = v )
}

## confirm the equivalence:
all.equal(rgb2hsv (rgb3),
          rgb2hsvR(rgb3), tolerance = 1e-14) # TRUE
```

---

savePlot *Save Cairo X11 Plot to File*

Description

Save the current page of a cairo X11() device to a file.

Usage

```
savePlot(filename = paste0("Rplot.", type),
         type = c("png", "jpeg", "tiff", "bmp"),
         device = dev.cur())
```

Arguments

filename	filename to save to.
type	file type.
device	the device to save from.

Details

Only cairo-based X11 devices are supported.

This works by copying the image surface to a file. For PNG will always be a 24-bit per pixel PNG 'DirectClass' file, for JPEG the quality is 75% and for TIFF there is no compression.

For devices with buffering this copies the buffer's image surface, so works even if dev.hold has been called.

The plot is saved after rendering onto the canvas (default opaque white), so for the default bg = "transparent" the effective background colour is the canvas colour.

Value

Invisible NULL.

Note

There is a similar function of the same name but more types for windows devices on Windows: that has an additional argument restoreConsole which is only supported on Windows.

See Also

recordPlot() which is device independent. Further, X11, dev.copy, dev.print

---

trans3d *3D to 2D Transformation for Perspective Plots*

Description

Projection of 3-dimensional to 2-dimensional points using a 4x4 viewing transformation matrix. Mainly for adding to perspective plots such as persp.

Usage

```
trans3d(x, y, z, pmat, continuous = FALSE, verbose = TRUE)
```

Arguments

x, y, z	numeric vectors of equal length, specifying points in 3D space.
pmat	a 4×4 *viewing transformation matrix*, suitable for projecting the 3D coordinates (x, y, z) into the 2D plane using homogeneous 4D coordinates (x, y, z, t); such matrices are returned by persp().
continuous	logical flag specifying if the transformation should check if the transformed points are continuous in the sense that they do not jump over $a/0$ discontinuity. As these assume (x,y,z) to describe a continuous curve, the default must be false. In case of projecting such a curve however, setting continuous=TRUE may be advisable.
verbose	only for continuous=TRUE, indicates if a warning should be issued when points are cut off.

Value

a list with two components

x, y the projected 2d coordinates of the 3d input (x,y,z).

See Also

persp

Examples

```
## See  help(persp) {after attaching the 'graphics' package}
##      -----------

## Example for 'continuous = TRUE' (vs default):
require(graphics)
x <- -10:10/10 # [-1, 1]
y <- -16:16/16 # [-1, 1] ==> z = fxy := outer(x,y) is also in [-1,1]

p <- persp(x, y, fxy <- outer(x,y), phi = 20, theta = 15, r = 3, ltheta = -75,
           shade = 0.8, col = "green3", ticktype = "detailed")
## 5 axis-parallel auxiliary lines in x-y  and y-z planes :
lines(trans3d(-.5 , y=-1:1, z=min(fxy),  pmat=p), lty=2)
lines(trans3d(  0 , y=-1:1, z=min(fxy),  pmat=p), lty=2)
lines(trans3d(-1:1, y= -.7, z=min(fxy),  pmat=p), lty=2)
lines(trans3d( -1,  y= -.7, z=c(-1,1) ,  pmat=p), lty=2)
lines(trans3d( -1,  y=-1:1, z= -.5    ,  pmat=p), lty=2)
## 2 pillars to carry the horizontals below:
lines(trans3d(-.5 , y= -.7, z=c(-1,-.5), pmat=p), lwd=1.5, col="gray10")
lines(trans3d( 0  , y= -.7, z=c(-1,-.5), pmat=p), lwd=1.5, col="gray10")
## now some "horizontal rays" (going from center to very left or very right):
doHor <- function(x1, x2, z, CNT=FALSE, ...)
    lines(trans3d(x=seq(x1, x2, by=0.5), y= -0.7, z = z, pmat = p, continuous = CNT),
          lwd = 3, type="b", xpd=NA, ...)
doHor(-10,  0, z = -0.5, col = 2)  # x in [-10, 0] -- to the very left : fine
doHor(-.5,  2, z = -0.52,col = 4) # x in [-0.5, 2] only {to the right} --> all fine
## but now, x in [-0.5, 20] -- "too far" ==> "wrap around" problem (without 'continuous=TRUE'):
doHor(-.5, 20, z = -0.58, col = "steelblue", lty=2)
## but it is fixed with continuous = CNT = TRUE:
doHor(-.5, 20, z = -0.55, CNT=TRUE, col = "skyblue")
```

Type1Font *Type 1 and CID Fonts*

Description

These functions are used to define the translation of a R graphics font family name to Type 1 or CID font descriptions, used by the pdf and postscript graphics devices.

Usage

```
Type1Font(family, metrics, encoding = "default")

CIDFont(family, cmap, cmapEncoding, pdfresource = "")
```

Arguments

family	a character string giving the name to be used internally for a Type 1 or CID-keyed font family. This needs to uniquely identify each family, so if you modify a family which is in use (see postscriptFonts) you need to change the family name.
metrics	a character vector of four or five strings giving paths to the afm (Adobe Font Metric) files for the font.
cmap	the name of a CMap file for a CID-keyed font.
encoding	for Type1Font, the name of an encoding file. Defaults to "default", which maps on Unix-alikes to "ISOLatin1.enc" and on Windows to "WinAnsi.enc". Otherwise, a file name in the 'enc' directory of the **grDevices** package, which is used if the path does not contain a path separator. An extension ".enc" can be omitted.
cmapEncoding	The name of a character encoding to be used with the named CMap file: strings will be translated to this encoding when written to the file.
pdfresource	A chunk of PDF code; only required for using a CID-keyed font on pdf; users should not be expected to provide this.

Details

For Type1Fonts, if four '.afm' files are supplied the fifth is taken to be "Symbol.afm". Relative paths are taken relative to the directory '*R_HOME*/library/grDevices/afm'. The fifth (symbol) font must be in AdobeSym encoding. However, the glyphs in the first four fonts are referenced by name and any encoding given within the '.afm' files is not used.

The '.afm' files may be compressed with (or without) final extension '.gz': the files which ship with R are installed as compressed files with this extension.

Glyphs in CID-keyed fonts are accessed by ID (number) and not by name. The CMap file maps encoded strings (usually in a MBCS) to IDs, so cmap and cmapEncoding specifications must match. There are no real bold or italic versions of CID fonts (bold/italic were very rarely used in traditional East Asian topography), and for the pdf device all four font faces will be identical. However, for the postscript device, bold and italic (and bold italic) are emulated.

CID-keyed fonts are intended only for use for the glyphs of East Asian languages, which are all monospaced and are all treated as filling the same bounding box. (Thus plotmath will work with such characters, but the spacing will be less carefully controlled than with Western glyphs.) The CID-keyed fonts do contain other characters, including a Latin alphabet: non-East-Asian glyphs are regarded as monospaced with half the width of East Asian glyphs. This is often the case, but sometimes Latin glyphs designed for proportional spacing are used (and may look odd). We strongly recommend that CID-keyed fonts are **only** used for East Asian glyphs.

Value

A list of class "Type1Font" or "CIDFont".

See Also

pdf, postscript, pdfFonts and postscriptFonts.

Examples

```
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                        "CM_symbol_10.afm"),
                      encoding = "TeXtext.enc")

## Not run:
## This could be used by
postscript(family = CMitalic)
## or
postscriptFonts(CMitalic = CMitalic)  # once in a session
postscript(family = "CMitalic", encoding = "TeXtext.enc")

## End(Not run)
```

---

windows *Windows Graphics Devices*

Description

Available only on Windows. A graphics device is opened. For windows, win.graph, x11 and X11 this is a window on the current Windows display: the multiple names are for compatibility with other systems. win.metafile prints to a file and win.print to the Windows print system.

Usage

```
windows(width, height, pointsize, record, rescale, xpinch, ypinch,
        bg, canvas, gamma, xpos, ypos, buffered, title,
        restoreConsole, clickToConfirm, fillOddEven,
        family, antialias)

win.graph(width, height, pointsize)

win.metafile(filename = "", width = 7, height = 7, pointsize = 12,
             family, restoreConsole = TRUE,
             xpinch = NA_real_, ypinch = NA_real_)

win.print(width = 7, height = 7, pointsize = 12, printer = "",
          family, antialias, restoreConsole = TRUE)
```

Arguments

width, height	the (nominal) width and height of the canvas of the plotting window in inches. Default 7.
pointsize	the default pointsize of plotted text, interpreted as big points (1/72 inch). Values are rounded to the nearest integer: values less than or equal to zero are reset to 12, the default.
record	logical: sets the initial state of the flag for recording plots. Default FALSE.
rescale	character, one of c("R", "fit", "fixed"). Controls the action for resizing of the device. Default "R". See the 'Resizing options' section.
xpinch, ypinch	double. Pixels per inch, horizontally and vertically. Default NA_real_, which means to take the value from Windows.
bg	color. The initial background color. Default "transparent".
canvas	color. The color of the canvas which is visible when the background color is transparent. Should be a solid color (and any alpha value will be ignored). Default "white".
gamma	gamma correction fudge factor. Colours in R are sRGB; if your monitor does not conform to sRGB, you might be able to improve things by tweaking this parameter to apply additional gamma correction to the RGB channels. By default 1 (no additional gamma correction).
xpos, ypos	integer. Position of the top left of the window, in pixels. Negative values are taken from the opposite edge of the monitor. Missing values (the default) mean take the default from the 'Rconsole' file, which in turn defaults to xpos = -25, ypos = 0: this puts the right edge of the window 25 pixels from the right edge of the monitor.
buffered	logical. Should the screen output be double-buffered? Default TRUE.
title	character string, up to 100 bytes. With the default "", a suitable title is created internally. A C-style format for an integer will be substituted by the device number.
filename	the name of the output file: it will be an enhanced Windows metafile, usually given extension '.emf' or '.wmf'. Up to 511 characters are allowed. The page number is substituted if an integer format is included in the character string (see pdf for further details) and tilde-expansion (see path.expand) is performed. (The result must be less than 600 characters long.) The default, "", means the clipboard.
printer	The name of a printer as known to Windows. The default causes a dialog box to come up for the user to choose a printer.
restoreConsole	logical: see the 'Details' below. Defaults to FALSE for screen devices.
clickToConfirm	logical: if true confirmation of a new frame will be by clicking on the device rather than answering a problem in the console. Default TRUE.
fillOddEven	logical controlling the polygon fill mode: see polygon for details. Default TRUE.
family	A length-one character vector specifying the default font family. See section 'Fonts'.
antialias	A length-one character vector, requesting control over font antialiasing. This is partially matched to "default", "none", "cleartype" or "gray". See the 'Fonts' section.

Details

All these devices are implemented as variants of the same device.

All arguments of windows have defaults set by windows.options: the defaults given in the arguments section are the defaults for the defaults. These defaults also apply to the internal values of gamma, xpinch, ypinch, buffered, restoreConsole and antialias for win.graph, x11 and X11.

The size of a window is computed from information provided about the display: it depends on the system being configured accurately. By default a screen device asks Windows for the number of pixels per inch. This can be overridden (it is often wrong) by specifying xpinch and ypinch, most conveniently *via* windows.options. For example, a 13.3 inch 1280x800 screen (a typical laptop display) was reported as 96 dpi even though it is physically about 114 dpi. These arguments may also be useful to match the scale of output to the size of a metafile (which otherwise depends on the system being configured accurately).

The different colours need to be distinguished carefully. Areas outside the device region are coloured in the Windows application background colour. The device region is coloured in the canvas colour. This is over-painted by the background colour of a plot when a new page is called for, but that background colour can be transparent (and is by default). One difference between setting the canvas colour and the background colour is that when a plot is saved the background colour is copied but the canvas colour is not. The argument bg sets the initial value of par("bg") in base graphics and gpar("fill") in grid graphics

Recorded plot histories are of class "SavedPlots". They have a print method, and a subset method. As the individual plots are of class "recordedplot" they can be replayed by printing them: see recordPlot. The active plot history is stored in variable .SavedPlots in the workspace.

When a screen device is double-buffered (the default) the screen is updated 100ms after last plotting call or every 500ms during continuous plotting. These times can be altered by setting options("windowsTimeout") to a vector of two integers before opening the device.

Line widths as controlled by par(lwd =) are in multiples of 1/96inch. Multiples less than 1 are allowed, down to one pixel width.

For win.metafile only one plot is allowed per file, and Windows seems to disallow reusing the file. So the *only* way to allow multiple plots is to use a parametrized filename as in the example. If the filename is omitted (or specified as ""), the output is copied to the clipboard when the device is closed.

The restoreConsole argument is a temporary fix for a problem in the current implementation of several Windows graphics devices, and is likely to be removed in an upcoming release. If set to FALSE, the console will not receive the focus after the new device is opened.

There is support for semi-transparent colours of lines, fills and text on the screen devices. These work for saving (from the 'File' menu) to PDF, PNG, BMP, JPEG and TIFF, but will be ignored if saving to Metafile and PostScript. Limitations in the underlying Windows API mean that a semi-transparent object must be contained strictly within the device region (allowing for line widths and joins).

Value

A plot device is opened: nothing is returned to the R interpreter.

Resizing options

If a screen device is re-sized, the default behaviour (`"R"`) is to redraw the plot(s) as if the new size had been specified originally. Using `"fit"` will rescale the existing plot(s) to fit the new device region, preserving the aspect ratio. Using `"fixed"` will leave the plot size unchanged, adding scrollbars if part of the plot is obscured.

A graphics window will never be created at more than 85% of the screen width or height, but can be resized to a larger size. For the first two `rescale` options the width and height are rescaled proportionally if necessary, and if `rescale = "fit"` the plot(s) are rescaled accordingly. If `rescale = "fixed"` the initially displayed portion is selected within these constraints, separately for width and height. In MDI mode, the limit is 85% of the MDI client region.

Using `strwidth` or `strheight` after a window has been rescaled (when using `"fit"`) gives dimensions in the original units, but only approximately as they are derived from the metrics of the rescaled fonts (which are in integer sizes)

The displayed region may be bigger than the 'paper' size, and area(s) outside the 'paper' are coloured in the Windows application background colour. Graphics parameters such as `"din"` refer to the scaled plot if rescaling is in effect.

Fonts

The fonts used for text drawn in a Windows device may be controlled in two ways. The file `R_HOME\etc\Rdevga` can be used to specify mappings for `par(font =)` (or the **grid** equivalent). Alternatively, a font family can be specified by a non-empty `family` argument (or by e.g. `par(family =)` in the graphics package) and this will be used for fonts 1:4 via the Windows font database (see `windowsFonts`).

How the fonts look depends on the antialiasing settings, both through the `antialias` argument and the machine settings. These are hints to Windows GDI that may not be able to be followed, but `antialias = "none"` should ensure that no antialiasing is used. For a screen device the default depends on the machine settings: it will be `"cleartype"` if that has been enabled. Note that the greyscale antialiasing that is used only for small fonts (below about 9 pixels, around 7 points on a typical display).

When accessing a system through Remote Desktop, both the Remote Desktop settings *and* the user's local account settings are relevant to whether antialiasing is used.

Some fonts are intended only to be used with ClearType antialiasing, for example the `Meiryo` Japanese font.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 7 inches square, although this is often incorrectly implemented by Windows: see 'Details'.
- Font sizes are in big points.
- The default font family is Arial.
- Line widths are as a multiple of 1/96 inch, with a minimum of one pixel.
- The minimum radius of a circle is 1 pixel.

- pch = "." with cex = 1 corresponds to a rectangle of sides the larger of one pixel and 0.01 inch.
- Colours are interpreted via the unprofiled colour mapping of the graphics card – this is *assumed* to conform to sRGB.

Note

x11(), X11() and win.graph() are simple wrappers calling windows(), and mainly exist for compatibility reasons.

Further, x11() and X11() have their own help page for Unix-alikes (where they also have more arguments).

See Also

windowsFonts, savePlot, bringToTop, Devices, pdf, x11 for Unix-alikes.

Examples

```
## Not run: ## A series of plots written to a sequence of metafiles
if(.Platform$OS.type == "windows")
   win.metafile("Rplot%02d.wmf", pointsize = 10)

## End(Not run)
```

---

windows.options *Auxiliary Function to Set/View Defaults for Arguments of windows()*

Description

The auxiliary function windows.options can be used to set or view (if called without arguments) the default values for the arguments of windows.

windows.options needs to be called before calling windows, and the default values it sets can be overridden by supplying arguments to windows.

Usage

```
windows.options(..., reset = FALSE)
```

Arguments

...	arguments width, height, pointsize, record, rescale, xpinch, ypinch, bg, canvas, gamma, xpos, ypos, buffered, restoreConsole, clickToConfirm, title, fillOddEven and antialias can be supplied.
reset	logical: should the defaults be reset to their 'factory-fresh' values?

Details

If both `reset = TRUE` and `...` are supplied the defaults are first reset to the 'factory-fresh' values and then the new values are applied.

Option `antialias` applies to screen devices (`windows`, `win.graph`, `X11` and `x11`). There is a separate option, `bitmap.aa.win`, for bitmap devices with `type = "windows"`.

Value

A named list of all the defaults. If any arguments are supplied the returned values are the old values and the result has the visibility flag turned off.

See Also

`windows`, `ps.options`.

Examples

```
## Not run:
## put something like this is your .Rprofile to customize the defaults
setHook(packageEvent("grDevices", "onLoad"),
        function(...)
            grDevices::windows.options(width = 8, height = 6,
                                       xpos = 0, pointsize = 10,
                                       bitmap.aa.win = "cleartype"))

## End(Not run)
```

---

windowsFonts *Windows Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to a windows font description and are available only on Windows.

Usage

```
windowsFont(family)

windowsFonts(...)
```

Arguments

`family`	a character vector containing the font family name (`"TT"` as the first two characters indicates a TrueType font).
`...`	either character strings naming mappings to display, or new (named) mappings to define.

Details

A windows device is created with a default font (see the documentation for `windows`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `"family"` in `par` and for `"fontfamily"` in `gpar` in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to windows fonts. A list of mappings is maintained and can be modified by the user.

The `windowsFonts` function can be used to list existing mappings and to define new mappings. The `windowsFont` function can be used to create a new mapping.

Default mappings are provided for three device-independent font family names: `"sans"` for a sans-serif font, `"serif"` for a serif font and `"mono"` for a monospaced font.

These mappings will only be used if the current font face is 1 (plain), 2 (bold), 3 (italic), or 4 (bold italic).

See Also

`windows`

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
  windowsFonts()
  windowsFonts("mono")
})

## Not run: ## set up for Japanese: needs the fonts installed
windows()  # make sure we have the right device type (available on Windows only)
Sys.setlocale("LC_ALL", "ja")
windowsFonts(JP1 = windowsFont("MS Mincho"),
             JP2 = windowsFont("MS Gothic"),
             JP3 = windowsFont("Arial Unicode MS"))
plot(1:10)
text(5, 2, "\u{4E10}\u{4E00}\u{4E01}", family = "JP1")
text(7, 2, "\u{4E10}\u{4E00}\u{4E01}", family = "JP1", font = 2)
text(5, 1.5, "\u{4E10}\u{4E00}\u{4E01}", family = "JP2")
text(9, 2, "\u{5100}", family = "JP3")

## End(Not run)
```

---

x11 *X Window System Graphics (X11)*

Description

on Windows, the `X11()` and `x11()` functions are simple wrappers to `windows()` for historical compatibility convenience: Calling `x11()` or `X11()` would work in most cases to open an interactive graphics device.

In R versions before 3.6.0, the Windows version had a shorter list of formal arguments. Consequently, calls to `X11(*)` with arguments should *name* them for back compatibility.

Almost all information below does *not* apply on Windows.

on Unix-alikes `X11` starts a graphics device driver for the X Window System (version 11). This can only be done on machines/accounts that have access to an X server.

`x11` is recognized as a synonym for `X11`.

The R function is a wrapper for two devices, one based on Xlib (https://en.wikipedia.org/wiki/Xlib) and one using cairographics (https://www.cairographics.org).

Usage

```
X11(display = "", width, height, pointsize, gamma, bg, canvas,
    fonts, family, xpos, ypos, title, type, antialias, symbolfamily)

X11.options(..., reset = FALSE)
```

Arguments

`display`	the display on which the graphics window will appear. The default is to use the value in the user's environment variable `DISPLAY`. This is ignored (with a warning) if an X11 device is already open on another display.
`width, height`	the width and height of the plotting window, in inches. If `NA`, taken from the resources and if not specified there defaults to 7 inches. See also 'Resources'.
`pointsize`	the default pointsize to be used. Defaults to 12.
`gamma`	gamma correction fudge factor. Colours in R are sRGB; if your monitor does not conform to sRGB, you might be able to improve things by tweaking this parameter to apply additional gamma correction to the RGB channels. By default 1 (no additional gamma correction).
`bg`	colour, the initial background colour. Default `"transparent"`.
`canvas`	colour. The colour of the canvas, which is visible only when the background colour is transparent. Should be an opaque colour (and any alpha value will be ignored). Default `"white"`.
`fonts`	for `type = "Xlib"` only: X11 font description strings into which weight, slant and size will be substituted. There are two, the first for fonts 1 to 4 and the second for font 5, the symbol font. See section 'Fonts'.
`family`	The default family: a length-one character string. This is primarily intended for cairo-based devices, but for `type = "Xlib"`, the `X11Fonts()` database is used to map family names to `fonts` (and this argument takes precedence over that one).
`xpos, ypos`	integer: initial position of the top left corner of the window, in pixels. Negative values are from the opposite corner, e.g. `xpos = -100` says the top right corner should be 100 pixels from the right edge of the screen. If `NA` (the default), successive devices are cascaded in 20 pixel steps from the top left. See also 'Resources'.
`title`	character string, up to 100 bytes. With the default, `""`, a suitable title is created internally. A C-style format for an integer will be substituted by the device number (see the `file` argument of `df` for further details). How non-ASCII titles are handled is implementation-dependent.

type	character string, one of "Xlib", "cairo", "nbcairo" or "dbcairo". Only the first will be available if the system was compiled without support for cairographics. The default is "cairo" where R was built using pangocairo (often not the case on macOS), otherwise "Xlib".
antialias	for cairo types, the type of anti-aliasing (if any) to be used. One of c("default", "none", "gray", "subpixel").
symbolfamily	for cairo-based devices only: a length-one character string that specifies the font family to be used as the "symbol" font (e.g., for plotmath output). The default value is "default", which means that R will choose a default "symbol" font based on the graphics device capabilities.
reset	logical: should the defaults be reset to their defaults?
...	Any of the arguments to X11, plus colortype and maxcubesize (see section 'Colour Rendering').

Details

The defaults for all of the arguments of X11 are set by X11.options: the 'Arguments' section gives the 'factory-fresh' defaults.

The initial size and position are only hints, and may not be acted on by the window manager. Also, some systems (especially laptops) are set up to appear to have a screen of a different size to the physical screen.

Option type selects between two separate devices: R can be built with support for neither, type = "Xlib" or both. Where both are available, types "cairo", "nbcairo" and "dbcairo" offer

- antialiasing of text and lines.
- translucent colours.
- scalable text, including to sizes like 4.5 pt.
- full support for UTF-8, so on systems with suitable fonts you can plot in many languages on a single figure (and this will work even in non-UTF-8 locales). The output should be locale-independent.

There are three variants of the cairo-based device. type = "nbcairo" has no buffering. type = "cairo" has some buffering, and supports dev.hold and dev.flush. type = "dbcairo" buffers output and updates the screen about every 100ms (by default). The refresh interval can be set (in units of seconds) by e.g. options(X11updates = 0.25): the value is consulted when a device is opened. Updates are only looked for every 50ms (at most), and during heavy graphics computations only every 500ms.

Which version will be fastest depends on the X11 connection and the type of plotting. You will probably want to use a buffered type unless backing store is in use on the X server (which for example it always is on the x86_64 macOS XQuartz server), as otherwise repainting when the window is exposed will be slow. On slow connections type = "dbcairo" will probably give the best performance.

Because of known problems with font selection on macOS without Pango (for example, most CRAN distributions), type = "cairo" is not the default there. These problems have included mixing up bold and italic (since worked around), selecting incorrect glyphs and ugly or missing symbol glyphs.

All devices which use an X11 server (including the `type = "Xlib"` versions of bitmap devices such as png) share internal structures, which means that they must use the same `display` and visual. If you want to change display, first close all such devices.

The cursor shown indicates the state of the device. If quiescent the cursor is an arrow: when the locator is in use it is a crosshair cursor, and when plotting computations are in progress (and this can be detected) it is a watch cursor. (The exact cursors displayed will depend on the window manager in use.)

X11 Fonts

This section applies only to `type = "Xlib"`.

An initial/default font family for the device can be specified via the `fonts` argument, but if a device-independent R graphics font family is specified (e.g., via `par(family =)` in the graphics package), the X11 device makes use of the X11 font database (see `X11Fonts`) to convert the R graphics font family to an X11-specific font family description. If `family` is supplied as an argument, the X11 font database is used to convert that, but otherwise the argument `fonts` (with default given by `X11.options`) is used.

X11 chooses fonts by matching to a pattern, and it is quite possible that it will choose a font in the wrong encoding or which does not contain glyphs for your language (particularly common in `iso10646-1` fonts).

The `fonts` argument is a two-element character vector, and the first element will be crucial in successfully using non-Western-European fonts. Settings that have proved useful include

`"-*-mincho-%s-%s-*-*-%d-*-*-*-*-*-*-*"` for CJK languages and `"-cronyx-helvetica-%s-%s-*-*-%d-*-*-*-*-*-*-*"` for Russian.

For UTF-8 locales, the `XLC_LOCALE` databases provide mappings between character encodings, and you may need to add an entry for your locale (e.g., Fedora Core 3 lacked one for `ru_RU.utf8`).

Cairo Fonts

The cairographics-based devices work directly with font family names such as `"Helvetica"` which can be selected initially by the `family` argument and subsequently by par or gpar. There are mappings for the three device-independent font families, `"sans"` for a sans-serif font (to `"Helvetica"`), `"serif"` for a serif font (to `"Times"`) and `"mono"` for a monospaced font (to `"Courier"`).

The font selection is handled by `Pango` (usually *via* `fontconfig`) or `fontconfig` (on macOS and perhaps elsewhere). The results depend on the fonts installed on the system running R – setting the environment variable `FC_DEBUG` to 1 normally allows some tracing of the selection process.

This works best when high-quality scalable fonts are installed, usually in Type 1 or TrueType formats: see the 'R Installation and Administration' manual for advice on how to obtain and install such fonts. At present the best rendering (including using kerning) will be achieved with TrueType fonts: see https://www.freedesktop.org/software/fontconfig/fontconfig-user.html for ways to set up your system to prefer them. The default family (`"Helvetica"`) is likely not to use kerning: alternatives which should if you have them installed are `"Arial"`, `"DejaVu Sans"` and `"Liberation Sans"` (and perhaps `"FreeSans"`). For those who prefer fonts with serifs, try `"Times New Roman"`, `"DejaVu Serif"` and `"Liberation Serif"`. To match LaTeX text, use something like `"CM Roman"`.

Fedora systems from version 31 on do not like the default "symbol" font family for rendering symbols (e.g., plotmath). For those systems, users should specify a different font via symbolfamily. The default can also be changed via X11.options.

Problems with incorrect rendering of symbols (e.g., of quote(pi) and expression(10^degree)) have been seen on Linux systems which have the Wine symbol font installed – fontconfig then prefers this and misinterprets its encoding. Adding the following lines to '~/.fonts.conf' or '/etc/fonts/local.conf' may circumvent this problem by preferring the URW Type 1 symbol font.

```
<fontconfig>
<match target="pattern">
  <test name="family"><string>Symbol</string></test>
  <edit name="family" mode="prepend" binding="same">
    <string>Standard Symbols L</string>
  </edit>
</match>
</fontconfig>
```

A test for this is to run at the command line fc-match Symbol. If that shows symbol.ttf that may be the Wine symbol font – use locate symbol.ttf to see if it is found from a directory with 'wine' in the name.

Resources

The standard X11 resource geometry can be used to specify the window position and/or size, but will be overridden by values specified as arguments or non-NA defaults set in X11.options. The class looked for is R_x11. Note that the resource specifies the width and height in pixels and not in inches. See for example 'man X' (or https://www.x.org/releases/current/). An example line in '~/.Xresources' might be

```
R_x11*geometry: 900x900-0+0
```

which specifies a 900 x 900 pixel window at the top right of the screen.

Colour Rendering

X11 supports several 'visual' types, and nowadays almost all systems support 'truecolor' which X11 will use by default. This uses a direct specification of any RGB colour up to the depth supported (usually 8 bits per colour). Other visuals make use of a palette to support fewer colours, only grays or even only black/white. The palette is shared between all X11 clients, so it can be necessary to limit the number of colours used by R.

The default for type = "Xlib" is to use the best possible colour model for the visual of the X11 server: these days this will almost always be 'truecolor'. This can be overridden by the colortype argument of X11.options. **Note:** All X11 and type = "Xlib" bmp, jpeg, png and tiff devices share a colortype which is set when the first device to be opened. To change the colortype you need to close *all* open such devices, and then use X11.options(colortype =).

The colortype types are tried in the order "true", "pseudo", "gray" and "mono" (black or white only). The values "pseudo" and "pseudo.cube" provide two colour strategies for a pseudocolor

visual. The first strategy provides on-demand colour allocation which produces exact colours until the colour resources of the display are exhausted (when plotting will fail). The second allocates (if possible) a standard colour cube, and requested colours are approximated by the closest value in the cube.

With `colortype` equal to `"pseudo.cube"` or `"gray"` successively smaller palettes are tried until one is completely allocated. If allocation of the smallest attempt fails the device will revert to `"mono"`. For `"gray"` the search starts at 256 grays for a display with depth greater than 8, otherwise with half the available colours. For `"pseudo.cube"` the maximum cube size is set by `X11.options(maxcolorsize =)` and defaults to 256. With that setting the largest cube tried is 4 levels each for RGB, using 64 colours in the palette.

The cairographics-based devices most likely only work (or work correctly) with 'TrueColor' visuals, although in principle this depends on the cairo installation: a warning is given if any other visual is encountered.

`type = "Xlib"` supports 'TrueColor', 'PseudoColor', 'GrayScale', `StaticGray` and `MonoChrome` visuals: 'StaticColor' and 'DirectColor' visuals are handled only in black/white.

Anti-aliasing

Anti-aliasing is only supported for cairographics-based devices, and applies to both graphics and fonts. It is generally preferable for lines and text, but can lead to undesirable effects for fills, e.g. for `image` plots, and so is never used for fills.

`antialias = "default"` is in principle platform-dependent, but seems most often equivalent to `antialias = "gray"`.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths in 1/96 inch, minimum one pixel for `type = "Xlib"`, 0.01 otherwise.
- For `type = "Xlib"` circle radii are in pixels with minimum one.
- Colours are interpreted by the X11 server, which is *assumed* to conform to sRGB.

Warning

Support for all the Unix devices is optional, so in packages `X11()` should be used conditionally after checking `capabilities("X11")`.

See Also

`Devices`, `X11Fonts`, `savePlot`.

Examples

```
## Not run:
if(.Platform$OS.type == "unix") { # Only on unix-alikes, possibly Mac,
## put something like this is your .Rprofile to customize the defaults
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::X11.options(width = 8, height = 6, xpos = 0,
                                             pointsize = 10))
}
## End(Not run)
```

`X11Fonts` *X11 Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to an X11 font description on Unix-alike platforms.

Usage

```
X11Font(font)

X11Fonts(...)
```

Arguments

`font`	a character string containing an X11 font description.
`...`	either character strings naming mappings to display, or new (named) mappings to define.

Details

These functions apply only to an `X11` device with `type = "Xlib"` – `X11(type = "cairo")` uses a different mechanism to select fonts.

Such a device is created with a default font (see the documentation for `X11`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `"family"` in `par` and for `"fontfamily"` in `gpar` in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to X11 fonts. A list of mappings is maintained and can be modified by the user.

The `X11Fonts` function can be used to list existing mappings and to define new mappings. The `X11Font` function can be used to create a new mapping.

Default mappings are provided for three device-independent font family names: `"sans"` for a sans-serif font, `"serif"` for a serif font and `"mono"` for a monospaced font. Further mappings are provided for `"Helvetica"` (the device default), `"Times"`, `"CyrHelvetica"`, `"CyrTimes"` (versions of these fonts with Cyrillic support, at least on Linux), `"Arial"` (on some platforms including macOS) and `"Mincho"` (a CJK font).

Note

Available only when `capabilities()[["X11"]]` is true.

See Also

`X11`

Examples

```
if(capabilities("X11")) withAutoprint({
X11Fonts()
X11Fonts("mono")
utopia <- X11Font("-*-utopia-*-*-*-*-*-*-*-*-*-*-*")
X11Fonts(utopia = utopia)
})
```

`xfig` *XFig Graphics Device*

Description

`xfig` starts the graphics device driver for producing XFig (version 3.2) graphics.

It was deprecated in R 4.4.0: consider using an SVG device for editable graphics.

The auxiliary function `ps.options` can be used to set and view (if called without arguments) default values for the arguments to `xfig` and `postscript`.

Usage

```
xfig(file = if(onefile) "Rplots.fig" else "Rplot%03d.fig",
     onefile = FALSE, encoding = "none",
     paper = "default", horizontal = TRUE,
     width = 0, height = 0, family = "Helvetica",
     pointsize = 12, bg = "transparent", fg = "black",
     pagecentre = TRUE, defaultfont = FALSE, textspecial = FALSE)
```

Arguments

`file`	a character string giving the file path. For use with `onefile = FALSE`, give a C integer format such as `"Rplot%03d.fig"` (the default in that case). See section 'File specifications' in the help for `pdf` for further details.
`onefile`	logical: if true allow multiple figures in one file. If false, assume only one page per file and generate a file number containing the page number.
`encoding`	The encoding in which to write text strings. The default is not to re-encode. This can be any encoding recognized by `iconv`: in a Western UTF-8 locale you probably want to select an 8-bit encoding such as `latin1`, and in an East Asian locale an `EUC` encoding. If re-encoding fails, the text strings will be written in the current encoding with a warning.

paper	the size of paper region. The choices are "A4", "Letter" and "Legal" (and these can be lowercase). A further choice is "default", which is the default. If this is selected, the paper size is taken from the option "papersize" if that is set to a non-empty value, otherwise "A4".
horizontal	the orientation of the printed image, a logical. Defaults to true, that is landscape orientation.
width, height	the width and height of the graphics region in inches. The default is to use the entire page less a 0.5 inch overall margin in each direction. (See postscript for further details.)
family	the font family to be used. This must be one of "AvantGarde", "Bookman", "Courier", "Helvetica" (the default), "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times". Any other value is replaced by "Helvetica", with a warning.
pointsize	the default point size to be used.
bg	the initial background color to be used.
fg	the initial foreground color to be used.
pagecentre	logical: should the device region be centred on the page?
defaultfont	logical: should the device use XFig's default font?
textspecial	logical: should the device set the textspecial flag for all text elements? This is useful when generating pstex from XFig figures.

Details

Although xfig can produce multiple plots in one file, the XFig format does not say how to separate or view them. So onefile = FALSE is the default.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.fig', ..., 'Rplot999.fig', 'Rplot1000.fig', ...

Line widths as controlled by par(lwd =) are in multiples of 5/6*1/72 inch. Multiples less than 1 are allowed. pch = "." with cex = 1 corresponds to a square of side 1/72 inch.

Windows users could make use of WinFIG (http://winfig.com/, shareware).

Conventions

This section describes the implementation of the conventions for graphics devices set out in the 'R Internals' manual.

- The default device size is the paper size with a 0.25 inch border on all sides.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are integers, multiples of 5/432 inch.
- Circle radii are multiples of 1/1200 inch.
- Colours are interpreted by the viewing/printing application.

Note

Only some line textures (`0 <= lty < 4`) are used. Eventually this may be partially remedied, but the XFig file format does not allow as general line textures as the R model. Unimplemented line textures are displayed as *dash-double-dotted*.

There is a limit of 512 colours (plus white and black) per file.

Author(s)

Brian Ripley. Support for `defaultFont` and `textSpecial` contributed by Sebastian Fischmeister.

See Also

`Devices`, `postscript`, `ps.options`.

---

xy.coords *Extracting Plotting Structures*

Description

`xy.coords` is used by many functions to obtain x and y coordinates for plotting. The use of this common mechanism across all relevant R functions produces a measure of consistency.

Usage

```
xy.coords(x, y = NULL, xlab = NULL, ylab = NULL, log = NULL,
          recycle = FALSE, setLab = TRUE)
```

Arguments

`x`, `y`	the x and y coordinates of a set of points. Alternatively, a single argument `x` can be provided.
`xlab`, `ylab`	names for the x and y variables to be extracted.
`log`	character, `"x"`, `"y"` or both, as for `plot`. Sets negative values to `NA` and gives a warning of class `"log_le_0"`.
`recycle`	logical; if `TRUE`, recycle (`rep`) the shorter of `x` or `y` if their lengths differ.
`setLab`	logical indicating if the resulting `xlab` and `ylab` should be constructed from the "kind" of `(x,y)`; otherwise, the arguments `xlab` and `ylab` are used.

Details

An attempt is made to interpret the arguments `x` and `y` in a way suitable for bivariate plotting (or other bivariate procedures).

If `y` is `NULL` and `x` is a

formula: of the form `yvar ~ xvar`. `xvar` and `yvar` are used as x and y variables.

list: containing components `x` and `y`, these are used to define plotting coordinates.

time series: the x values are taken to be `time(x)` and the y values to be the time series.

matrix or `data.frame` with two or more columns: the first is assumed to contain the x values and the second the y values. *Note* that is also true if `x` has columns named `"x"` and `"y"`; these names will be irrelevant here.

In any other case, the `x` argument is coerced to a vector and returned as **y** component where the resulting `x` is just the index vector `1:n`. In this case, the resulting `xlab` component is set to `"Index"` (if `setLab` is true as by default).

If `x` (after transformation as above) inherits from class `"POSIXt"` it is coerced to class `"POSIXct"`.

Value

A list with the components

`x`	numeric (i.e., `"double"`) vector of abscissa values.
`y`	numeric vector of the same length as `x`.
`xlab`	`character(1)` or `NULL`, the 'label' of `x`.
`ylab`	`character(1)` or `NULL`, the 'label' of `y`.

See Also

`plot.default`, `lines`, `points` and `lowess` are examples of functions which use this mechanism.

Examples

```
ff <- stats::fft(1:9)
xy.coords(ff)
xy.coords(ff, xlab = "fft") # labels "Re(fft)",  "Im(fft)"

with(cars, xy.coords(dist ~ speed, NULL)$xlab ) # = "speed"

xy.coords(1:3, 1:2, recycle = TRUE) # otherwise error "lengths differ"
xy.coords(-2:10, log = "y")
##> xlab: "Index"  \\  warning: 3 y values <= 0 omitted ..
op <- options(warn = 2)# ==> warnings would be errors, we suppress the one "we know":
suppressWarnings(xy.coords(-2:10, log = "y"), classes="log_le_0") -> xy
options(op) # revert
stopifnot(is.list(xy), identical (1:13 +0,  xy$x),
          identical(c(rep(NA, 3), 1:10 +0), xy$y))
```

xyTable *Multiplicities of (x,y) Points, e.g., for a Sunflower Plot*

Description

Given (x,y) points, determine their multiplicity – checking for equality only up to some (crude kind of) noise. Note that this is special kind of 2D binning.

Usage

```
xyTable(x, y = NULL, digits)
```

Arguments

x, y	numeric vectors of the same length; alternatively other (x, y) argument combinations as allowed by `xy.coords(x, y)`.
digits	integer specifying the significant digits to be used for determining equality of coordinates. These are compared after rounding them via `signif(*, digits)`.

Value

A list with three components of same length,

x	x coordinates, rounded and sorted.
y	y coordinates, rounded (and sorted within x).
number	multiplicities (positive integers); i.e., `number[i]` is the multiplicity of `(x[i], y[i])`.

See Also

`sunflowerplot` which typically uses `xyTable()`; `signif`.

Examples

```
xyTable(iris[, 3:4], digits = 6)

## Discretized uncorrelated Gaussian:

xy <- data.frame(x = round(sort(stats::rnorm(100))), y = stats::rnorm(100))
xyTable(xy, digits = 1)
```

xyz.coords *Extracting Plotting Structures*

Description

Utility for obtaining consistent x, y and z coordinates and labels for three dimensional (3D) plots.

Usage

```
xyz.coords(x, y = NULL, z = NULL,
           xlab = NULL, ylab = NULL, zlab = NULL,
           log = NULL, recycle = FALSE, setLab = TRUE)
```

Arguments

x, y, z	the x, y and z coordinates of a set of points. Both y and z can be left at NULL. In this case, an attempt is made to interpret x in a way suitable for plotting. If the argument is a formula zvar ~ xvar + yvar, xvar, yvar and zvar are used as x, y and z variables; if the argument is a list containing components x, y and z, these are assumed to define plotting coordinates; if the argument is a matrix or data.frame with three or more columns, the first is assumed to contain the x values, the 2nd the y ones, and the 3rd the z ones – independently of any column names that x may have. Alternatively two arguments x and y can be provided (leaving z = NULL). One may be real, the other complex; in any other case, the arguments are coerced to vectors and the values plotted against their indices.
xlab, ylab, zlab	names for the x, y and z variables to be extracted.
log	character, "x", "y", "z" or combinations. Sets negative values to NA and gives a warning of class "log_le_0".
recycle	logical; if TRUE, recycle (rep) the shorter ones of x, y or z if their lengths differ.
setLab	logical indicating if the resulting xlab and ylab should be constructed from the "kind" of (x,y); otherwise, the arguments xlab and ylab are used.

Value

A list with the components

x	numeric (i.e., double) vector of abscissa values.
y	numeric vector of the same length as x.
z	numeric vector of the same length as x.
xlab	character(1) or NULL, the axis label of x.
ylab	character(1) or NULL, the axis label of y.
zlab	character(1) or NULL, the axis label of z.

Author(s)

Uwe Ligges and Martin Maechler

See Also

xy.coords for 2D.

Examples

```
xyz.coords(data.frame(10*1:9, -4), y = NULL, z = NULL)

xyz.coords(1:5, stats::fft(1:5), z = NULL, xlab = "X", ylab = "Y")

y <- 2 * (x2 <- 10 + (x1 <- 1:10))
xyz.coords(y ~ x1 + x2, y = NULL, z = NULL)

xyz.coords(data.frame(x = -1:9, y = 2:12, z = 3:13), y = NULL, z = NULL,
           log = "xy")
##> Warning message: 2 x values <= 0 omitted ...
## Suppress this specific warning:
suppressWarnings(xyz.coords(x = -1:9, y = 2:12, z = 3:13, log = "xy"),
                 classes = "log_le_0")
```

Chapter 5

The graphics package

---

graphics-package *The R Graphics Package*

Description

R functions for base graphics

Details

This package contains functions for 'base' graphics. Base graphics are traditional S-like graphics, as opposed to the more recent grid graphics.

For a complete list of functions with individual help pages, use `library(help = "graphics")`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

abline *Add Straight Lines to a Plot*

Description

This function adds one or more straight lines through the current plot.

Usage

```
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, ...)
```

Arguments

a, b	the intercept and slope, single values.
untf	logical asking whether to *untransform*. See 'Details'.
h	the y-value(s) for horizontal line(s).
v	the x-value(s) for vertical line(s).
coef	a vector of length two giving the intercept and slope.
reg	an object with a coef method. See 'Details'.
...	graphical parameters such as col, lty and lwd (possibly as vectors: see 'Details') and xpd and the line characteristics lend, ljoin and lmitre.

Details

Typical usages are

```
abline(a, b, ...)
abline(h =, ...)
abline(v =, ...)
abline(coef =, ...)
abline(reg =, ...)
```

The first form specifies the line in intercept/slope form (alternatively a can be specified on its own and is taken to contain the slope and intercept in vector form).

The h= and v= forms draw horizontal and vertical lines at the specified coordinates.

The coef form specifies the line by a vector containing the slope and intercept.

reg is a regression object with a coef method. If this returns a vector of length 1 then the value is taken to be the slope of a line through the origin, otherwise, the first 2 values are taken to be the intercept and slope.

If untf is true, and one or both axes are log-transformed, then a curve is drawn corresponding to a line in original coordinates, otherwise a line is drawn in the transformed coordinate system. The h and v parameters always refer to original coordinates.

The graphical parameters col, lty and lwd can be specified; see par for details. For the h= and v= usages they can be vectors of length greater than one, recycled as necessary.

Specifying an xpd argument for clipping overrides the global par("xpd") setting used otherwise.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`lines` and `segments` for connected and arbitrary lines given by their *endpoints*. `par`.

Examples

```
## Setup up coordinate system (with x == y aspect ratio):
plot(c(-2,3), c(-1,5), type = "n", xlab = "x", ylab = "y", asp = 1)
## the x- and y-axis, and an integer grid
abline(h = 0, v = 0, col = "gray60")
text(1,0, "abline( h = 0 )", col = "gray60", adj = c(0, -.1))
abline(h = -1:5, v = -2:3, col = "lightgray", lty = 3)
abline(a = 1, b = 2, col = 2)
text(1,3, "abline( 1, 2 )", col = 2, adj = c(-.1, -.1))

## Simple Regression Lines:
require(stats)
sale5 <- c(6, 4, 9, 7, 6, 12, 8, 10, 9, 13)
plot(sale5)
abline(lsfit(1:10, sale5))
abline(lsfit(1:10, sale5, intercept = FALSE), col = 4) # less fitting

z <- lm(dist ~ speed, data = cars)
plot(cars)
abline(z) # equivalent to abline(reg = z) or
abline(coef = coef(z))

## trivial intercept model
abline(mC <- lm(dist ~ 1, data = cars)) ## the same as
abline(a = coef(mC), b = 0, col = "blue")
```

---

arrows *Add Arrows to a Plot*

Description

Draw arrows between pairs of points.

Usage

```
arrows(x0, y0, x1 = x0, y1 = y0, length = 0.25, angle = 30,
       code = 2, col = par("fg"), lty = par("lty"),
       lwd = par("lwd"), ...)
```

Arguments

x0, y0	coordinates of points **from** which to draw.
x1, y1	coordinates of points **to** which to draw. At least one must the supplied
length	length of the edges of the arrow head (in inches).
angle	angle from the shaft of the arrow to the edge of the arrow head.
code	integer code, determining *kind* of arrows to be drawn.
col, lty, lwd	graphical parameters, possible vectors. NA values in col cause the arrow to be omitted.
...	graphical parameters such as xpd and the line characteristics lend, ljoin and lmitre: see par.

Details

For each i, an arrow is drawn between the point (x0[i], y0[i]) and the point (x1[i], y1[i]). The coordinate vectors will be recycled to the length of the longest.

If code = 1 an arrowhead is drawn at (x0[i], y0[i]) and if code = 2 an arrowhead is drawn at (x1[i], y1[i]). If code = 3 a head is drawn at both ends of the arrow. Unless length = 0, when no head is drawn.

The graphical parameters col, lty and lwd can be vectors of length greater than one and will be recycled if necessary.

The direction of a zero-length arrow is indeterminate, and hence so is the direction of the arrowheads. To allow for rounding error, arrowheads are omitted (with a warning) on any arrow of length less than 1/1000 inch.

Note

The first four arguments in the comparable S function are named x1, y1, x2, y2.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

segments to draw segments.

Examples

```
x <- stats::runif(12); y <- stats::rnorm(12)
i <- order(x, y); x <- x[i]; y <- y[i]
plot(x,y, main = "arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)  # one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col = 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col = "pink")
```

assocplot *Association Plots*

Description

Produce a Cohen-Friendly association plot indicating deviations from independence of rows and columns in a 2-dimensional contingency table.

Usage

```
assocplot(x, col = c("black", "red"), space = 0.3,
          main = NULL, xlab = NULL, ylab = NULL)
```

Arguments

x	a two-dimensional contingency table in matrix form.
col	a character vector of length two giving the colors used for drawing positive and negative Pearson residuals, respectively.
space	the amount of space (as a fraction of the average rectangle width and height) left between each rectangle.
main	overall title for the plot.
xlab	a label for the x axis. Defaults to the name (if any) of the row dimension in x.
ylab	a label for the y axis. Defaults to the name (if any) of the column dimension in x.

Details

For a two-way contingency table, the signed contribution to Pearson's χ^2 for cell i, j is $d_{ij} = (f_{ij} - e_{ij})/\sqrt{e_{ij}}$, where f_{ij} and e_{ij} are the observed and expected counts corresponding to the cell. In the Cohen-Friendly association plot, each cell is represented by a rectangle that has (signed) height proportional to d_{ij} and width proportional to $\sqrt{e_{ij}}$, so that the area of the box is proportional to the difference in observed and expected frequencies. The rectangles in each row are positioned relative to a baseline indicating independence ($d_{ij} = 0$). If the observed frequency of a cell is greater than the expected one, the box rises above the baseline and is shaded in the color specified by the first element of col, which defaults to black; otherwise, the box falls below the baseline and is shaded in the color specified by the second element of col, which defaults to red.

A more flexible and extensible implementation of association plots written in the grid graphics system is provided in the function assoc in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2006).

References

Cohen, A. (1980), On the graphical display of the significant components in a two-way contingency table. *Communications in Statistics—Theory and Methods*, **9**, 1025–1041. doi:10.1080/03610928008827940.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://datavis.ca/papers/sugi/sugi17.pdf

Meyer, D., Zeileis, A., and Hornik, K. (2006) The strucplot Framework: Visualizing Multi-Way Contingency Tables with **vcd**. *Journal of Statistical Software*, **17(3)**, 1–48. doi:10.18637/jss.v017.i03.

See Also

mosaicplot, chisq.test.

Examples

```
## Aggregate over sex:
x <- marginSums(HairEyeColor, c(1, 2))
x
assocplot(x, main = "Relation between hair and eye color")
```

Axis *Generic Function to Add an Axis to a Plot*

Description

Generic function to add a suitable axis to the current plot.

Usage

```
Axis(x = NULL, at = NULL, ..., side, labels = NULL)
```

Arguments

x	an object which indicates the range over which an axis should be drawn
at	the points at which tick-marks are to be drawn.
side	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
labels	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tick points. If this is specified as a character or expression vector, at should be supplied and they should be the same length.
...	arguments to be passed to methods and perhaps then to axis.

Details

This is a generic function. It works in a slightly non-standard way: if `x` is supplied and non-NULL it dispatches on `x`, otherwise if `at` is supplied and non-NULL it dispatches on `at`, and the default action is to call `axis`, omitting argument `x`.

The idea is that for plots for which either or both of the axes are numerical but with a special interpretation, the standard plotting functions (including `boxplot`, `contour`, `coplot`, `filled.contour`, `pairs`, `plot.default`, `rug` and `stripchart`) will set up user coordinates and `Axis` will be called to label them appropriately.

There are `"Date"` and `"POSIXt"` methods which can pass an argument `format` on to the appropriate `axis` method (see `axis.POSIXct`).

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see 'Details').

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

See Also

`axis` (which is eventually called from all `Axis()` methods) in package **graphics**.

`axis` *Add an Axis to a Plot*

Description

Adds an axis to the current plot, allowing the specification of the side, position, labels, and other options.

Usage

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
     pos = NA, outer = FALSE, font = NA, lty = "solid",
     lwd = 1, lwd.ticks = lwd, col = NULL, col.ticks = NULL,
     hadj = NA, padj = NA, gap.axis = NA, ...)
```

Arguments

`side`	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
`at`	the points at which tick-marks are to be drawn. Non-finite (infinite, `NaN` or `NA`) values are omitted. By default (when `NULL`) tickmark locations are computed, see 'Details' below.

`labels` this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tick points. (Other objects are coerced by `as.graphicsAnnot`.) If this is not logical, `at` should also be supplied and of the same length. If `labels` is of length zero after coercion, it has the same effect as supplying `TRUE`.

`tick` a logical value specifying whether tickmarks and an axis line should be drawn.

`line` the number of lines into the margin at which the axis line will be drawn, if not `NA`.

`pos` the coordinate at which the axis line is to be drawn: if not `NA` this overrides the value of `line`.

`outer` a logical value indicating whether the axis should be drawn in the outer plot margin, rather than the standard plot margin.

`font` font for text. Defaults to `par("font")`.

`lty` line type for both the axis line and the tick marks.

`lwd`, `lwd.ticks` line widths for the axis line and the tick marks. Zero or negative values will suppress the line or ticks.

`col`, `col.ticks` colors for the axis line and the tick marks respectively. `col = NULL` means to use `par("fg")`, possibly specified inline, and `col.ticks = NULL` means to use whatever color `col` resolved to.

`hadj` adjustment (see `par("adj")`) for all labels *parallel* ('horizontal') to the reading direction. If this is not a finite value, the default is used (centring for strings parallel to the axis, justification of the end nearest the axis otherwise).

`padj` adjustment for each tick label *perpendicular* to the reading direction. For labels parallel to the axes, `padj = 0` means left or bottom alignment, and `padj = 1` means right or top alignment (relative to the line). This can be a vector given a value for each string, and will be recycled as necessary.

If `padj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

`gap.axis` an optional (typically non-negative) numeric factor to be multiplied with the size of an 'm' to determine the minimal gap between labels that are drawn, see 'Details'. The default, `NA`, corresponds to `1` for tick labels drawn *parallel* to the axis and `0.25` otherwise, i.e., the default is equivalent to

```
perpendicular <- function(side, las) {
  is.x <- (side %% 2 == 1) # is horizontal x-axis
  ( is.x && (las %in% 2:3)) ||
  (!is.x && (las %in% 1:2))
}
gap.axis <- if(perpendicular(side, las)) 0.25 else 1
```

`gap.axis` may typically be relevant when `at = .` tick-mark positions are specified explicitly.

`...` other graphical parameters may also be passed as arguments to this function, particularly, `cex.axis`, `col.axis` and `font.axis` for axis annotation, i.e. tick labels, `mgp` and `xaxp` or `yaxp` for positioning, `tck` or `tcl` for tick mark length

and direction, las for vertical/horizontal label orientation, or fg instead of col, and xpd for clipping. See par on these.

Parameters xaxt (sides 1 and 3) and yaxt (sides 2 and 4) control if the axis is plotted at all.

Note that lab will partial match to argument labels unless the latter is also supplied. (Since the default axes have already been set up by plot.window, lab will not be acted on by axis.)

Details

The axis line is drawn from the lowest to the highest value of at, but will be clipped at the plot region. By default, only ticks which are drawn from points within the plot region (up to a tolerance for rounding error) are plotted, but the ticks and their labels may well extend outside the plot region. Use xpd = TRUE or xpd = NA to allow axes to extend further.

When at = NULL, pretty tick mark locations are computed internally (the same way axTicks(side) would) from par("xaxp") or "yaxp" and par("xlog") (or "ylog"). Note that these locations may change if an on-screen plot is resized (for example, if the plot argument asp (see plot.window) is set.)

If labels is not specified, the numeric values supplied or calculated for at are converted to character strings as if they were a numeric vector printed by print.default(digits = 7).

The code tries hard not to draw overlapping tick labels, and so will omit labels where they would abut or overlap previously drawn labels. This can result in, for example, every other tick being labelled. The ticks are drawn left to right or bottom to top, and space at least the size of an 'm', multiplied by gap.axis, is left between labels. In previous R versions, this applied only for labels written *parallel* to the axis direction, hence not for e.g., las = 2. Using gap.axis = -1 restores that (buggy) previous behaviour (in the perpendicular case).

If either line or pos is set, they (rather than par("mgp")[3]) determine the position of the axis line and tick marks, and the tick labels are placed par("mgp")[2] further lines into (or towards for pos) the margin.

Several of the graphics parameters affect the way axes are drawn. The vertical (for sides 1 and 3) positions of the axis and the tick labels are controlled by mgp[2:3] and mex, the size and direction of the ticks is controlled by tck and tcl and the appearance of the tick labels by cex.axis, col.axis and font.axis with orientation controlled by las (but not srt, unlike S which uses srt if at is supplied and las if it is not). Note that adj is not supported and labels are always centered. See par for details.

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see 'Details').

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Axis` for a generic interface.

`axTicks` returns the axis tick locations corresponding to `at = NULL`; `pretty` is more flexible for computing pretty tick coordinates and does *not* depend on (nor adapt to) the coordinate system in use.

Several graphics parameters affecting the appearance are documented in `par`.

Examples

```
require(stats) # for rnorm
plot(1:4, rnorm(4), axes = FALSE)
axis(1, 1:4, LETTERS[1:4])
axis(2)
box() #- to make it look "as usual"

plot(1:7, rnorm(7), main = "axis() examples",
     type = "s", xaxt = "n", frame.plot = FALSE, col = "red")
axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:
axis(4, col = "violet", col.axis = "dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)

# one way to have a custom x axis
plot(1:10, xaxt = "n")
axis(1, xaxp = c(2, 9, 7))

## Changing default gap between labels:
plot(0:100, type="n", axes=FALSE, ann=FALSE)
title(quote("axis(1, .., gap.axis = f)," ~~ f >= 0))
axis(2, at = 5*(0:20), las = 1, gap.axis = 1/4)
gaps <- c(4, 2, 1, 1/2, 1/4, 0.1, 0)
chG <- paste0(ifelse(gaps == 1, "default:  ", ""),
              "gap.axis=", formatC(gaps))
jj <- seq_along(gaps)
linG <- -2.5*(jj-1)
for(j in jj) {
    isD <- gaps[j] == 1 # is default
    axis (1, at=5*(0:20), gap.axis = gaps[j], padj=-1, line = linG[j],
          col.axis = if(isD) "forest green" else 1, font.axis= 1+isD)
}
mtext(chG, side=1, padj=-1, line = linG -1/2, cex=3/4,
      col = ifelse(gaps == 1, "forest green", "blue3"))
## now shrink the window (in x- and y-direction) and observe the axis labels drawn
```

Description

Add a date/time axis to the current plot of an object of class "POSIXt" or "Date", respectively.

Usage

```
axis.POSIXct(side, x, at, format, labels = TRUE, ...)
axis.Date(side, x, at, format, labels = TRUE, ...)
```

Arguments

side	see axis.
x, at	optional date-time or Date objects, or other types of objects that can be converted appropriately.
format	an optional character string specifying the label format, see strftime.
labels	either a logical value specifying whether annotations are to be made at the tick-marks, or a character vector of labels to be placed at the tick points specified by at.
...	further arguments to be passed from or to other methods, typically graphical parameters.

Details

If at is unspecified, axis.POSIXct and axis.Date work quite hard (from R 4.3.0 via pretty for date-time classes) to choose suitable time units (years, months, days, hours, minutes, or seconds) and a sensible label format based on the axis range. par("lab") controls the approximate number of intervals.

If at is supplied it specifies the locations of the ticks and labels. If the label format is unspecified, a good guess is made by looking at the granularity of at. Printing of tick labels can be suppressed with labels = FALSE.

The date-times for a "POSIXct" input are interpreted in the time zone give by the "tzone" attribute if there is one, otherwise the current time zone.

The way the date-times are rendered (especially month names) is controlled by the locale setting of category "LC_TIME" (see Sys.setlocale).

Value

The locations on the axis scale at which tick marks were drawn.

See Also

DateTimeClasses, Dates for details of the classes.

Axis.

Examples

```
with(beaver1, {
    opar <- par(mfrow = c(3,1))
    time <- strptime(paste(1990, day, time %/% 100, time %% 100),
                     "%Y %j %H %M")
    plot(time, temp, type = "l") # axis at 6-hour intervals
    # request more ticks
    olab <- par(lab = c(10, 10, 7))
    plot(time, temp, type = "l")
    par(olab)
    # now label every hour on the time axis
    plot(time, temp, type = "l", xaxt = "n")
    r <- as.POSIXct(round(range(time), "hours"))
    axis.POSIXct(1, at = seq(r[1], r[2], by = "hour"), format = "%H")
    par(opar) # reset changed par settings
})

plot(.leap.seconds, seq_along(.leap.seconds), type = "n", yaxt = "n",
     xlab = "leap seconds", ylab = "", bty = "n")
rug(.leap.seconds)
## or as dates
lps <- as.Date(.leap.seconds)
plot(lps, seq_along(.leap.seconds),
     type = "n", yaxt = "n", xlab = "leap seconds",
     ylab = "", bty = "n")
rug(lps)

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*sort(stats::runif(100))
plot(random.dates, 1:100)
# or for a better axis labelling
plot(random.dates, 1:100, xaxt = "n")
axis.Date(1, at = seq(as.Date("2001/1/1"), max(random.dates)+6, "weeks"))
axis.Date(1, at = seq(as.Date("2001/1/1"), max(random.dates)+6, "days"),
     labels = FALSE, tcl = -0.2)

## axis.Date() with various data types:
x <- seq(as.Date("2022-01-20"), as.Date("2023-03-21"), by = "days")
plot(data.frame(x, y = 1), xaxt = "n")
legend("topleft", title = "input",
       legend = c("character", "Date", "POSIXct", "POSIXlt", "numeric"),
       fill = c("violet", "red", "orange", "coral1", "darkgreen"))
axis.Date(1)
axis.Date(3, at = "2022-04-01", col.axis = "violet")
axis.Date(3, at = as.Date("2022-07-01"), col.axis = "red")
axis.Date(3, at = as.POSIXct(as.Date("2022-10-01")), col.axis = "orange")
axis.Date(3, at = as.POSIXlt(as.Date("2023-01-01")), col.axis = "coral1")
axis.Date(3, at = as.integer(as.Date("2023-04-01")), col.axis = "darkgreen")
## automatically extends the format:
axis.Date(1, at = "2022-02-15", col.axis = "violet",
         col = "violet", tck = -0.05, mgp = c(3,2,0))
```

```
## axis.POSIXct() with various data types (2 minutes):
x <- as.POSIXct("2022-10-01") + c(0, 60, 120)
attributes(x)   # no timezone
plot(data.frame(x, y = 1), xaxt = "n")
legend("topleft", title = "input",
       legend = c("character", "Date", "POSIXct", "POSIXlt", "numeric"),
       fill = c("violet", "red", "orange", "coral1", "darkgreen"))
axis.POSIXct(1)
axis.POSIXct(3, at = "2022-10-01 00:01", col.axis = "violet")
axis.POSIXct(3, at = as.Date("2022-10-01"), col.axis = "red")
axis.POSIXct(3, at = as.POSIXct("2022-10-01 00:01:30"), col.axis = "orange")
axis.POSIXct(3, at = as.POSIXlt("2022-10-01 00:02"), col.axis = "coral1")
axis.POSIXct(3, at = as.numeric(as.POSIXct("2022-10-01 00:00:30")),
                col.axis = "darkgreen")
## automatically extends format (here: subseconds):
axis.POSIXct(3, at = as.numeric(as.POSIXct("2022-10-01 00:00:30")) + 0.25,
                col.axis = "forestgreen", col = "darkgreen", mgp = c(3,2,0))

## axis.POSIXct: 2 time zones
HST <- as.POSIXct("2022-10-01", tz = "HST") + c(0, 60, 60*60)
CET <- HST
attr(CET, "tzone") <- "CET"
plot(data.frame(HST, y = 1), xaxt = "n", xlab = "Hawaii Standard Time (HST)")
axis.POSIXct(1, HST)
axis.POSIXct(1, HST, at = "2022-10-01 00:10", col.axis = "violet")
axis.POSIXct(3, CET)
mtext(3, text = "Central European Time (CET)", line = 3)
axis.POSIXct(3, CET, at="2022-10-01 12:10", col.axis = "violet")
```

`axTicks` *Compute Axis Tickmark Locations*

Description

Compute pretty tickmark locations, the same way as R does internally. This is only non-trivial when **log** coordinates are active. By default, gives the `at` values which `axis(side)` would use.

Usage

```
axTicks(side, axp = NULL, usr = NULL, log = NULL, nintLog = NULL)
```

Arguments

`side`	integer in 1:4, as for `axis`.
`axp`	numeric vector of length three, defaulting to `par("xaxp")` or `par("yaxp")` depending on the `side` argument (`par("xaxp")` if `side` is 1 or 3, `par("yaxp")` if side is 2 or 4).
`usr`	numeric vector of length two giving user coordinate limits, defaulting to the relevant portion of `par("usr")` (`par("usr")[1:2]` or `par("usr")[3:4]` for `side` in (1,3) or (2,4) respectively).

log	logical indicating if log coordinates are active; defaults to par("xlog") or par("ylog") depending on side.
nintLog	(only used when log is true): approximate (lower bound for the) number of tick intervals; defaults to par("lab")[j] where j is 1 or 2 depending on side. Set this to Inf if you want the same behavior as in earlier R versions (than 2.14.x).

Details

The axp, usr, and log arguments must be consistent as their default values (the par(..) results) are. If you specify all three (as non-NULL), the graphics environment is not used at all. Note that the meaning of axp differs significantly when log is TRUE; see the documentation on par(xaxp = .).

axTicks() may be seen as an R implementation of the C function CreateAtVector() in '..../src/main/plot.c' which is called by axis(side, *) when no argument at is specified or directly by axisTicks() (in package **grDevices**).
The delicate case, log = TRUE, now makes use of axisTicks unless nintLog = Inf which exists for back compatibility.

Value

numeric vector of coordinate values at which axis tickmarks can be drawn. By default, when only the first argument is specified, these values should be identical to those that axis(side) would use or has used. Note that the values are decreasing when usr is ("reverse axis" case).

See Also

axis, par. pretty uses the same algorithm (but independently of the graphics environment) and has more options. However it is not available for log = TRUE.

axisTicks() (package **grDevices**).

Examples

```
 plot(1:7, 10*21:27)
 axTicks(1)
 axTicks(2)
 stopifnot(identical(axTicks(1), axTicks(3)),
           identical(axTicks(2), axTicks(4)))

## Show how axTicks() and axis() correspond :
op <- par(mfrow = c(3, 1))
for(x in 9999 * c(1, 2, 8)) {
    plot(x, 9, log = "x")
    cat(formatC(par("xaxp"), width = 5),";", T <- axTicks(1),"\n")
    rug(T, col =  adjustcolor("red", 0.5), lwd = 4)
}
par(op)

x <- 9.9*10^(-3:10)
plot(x, 1:14, log = "x")
axTicks(1) # now length 7
```

```
axTicks(1, nintLog = Inf) # rather too many

## An example using axTicks() without reference to an existing plot
## (copying R's internal procedures for setting axis ranges etc.),
## You do need to supply _all_ of axp, usr, log, nintLog
## standard logarithmic y axis labels
ylims <- c(0.2, 88)
get_axp <- function(x) 10^c(ceiling(x[1]), floor(x[2]))
## mimic par("yaxs") == "i"
usr.i <- log10(ylims)
(aT.i <- axTicks(side = 2, usr = usr.i,
                 axp = c(get_axp(usr.i), n = 3), log = TRUE, nintLog = 5))
## mimic (default) par("yaxs") == "r"
usr.r <- extendrange(r = log10(ylims), f = 0.04)
(aT.r <- axTicks(side = 2, usr = usr.r,
                 axp = c(get_axp(usr.r), 3), log = TRUE, nintLog = 5))

## Prove that we got it right :
plot(0:1, ylims, log = "y", yaxs = "i")
stopifnot(all.equal(aT.i, axTicks(side = 2)))

plot(0:1, ylims, log = "y", yaxs = "r")
stopifnot(all.equal(aT.r, axTicks(side = 2)))
```

---

barplot *Bar Plots*

Description

Creates a bar plot with vertical or horizontal bars.

Usage

```
barplot(height, ...)

## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,
        add = FALSE, ann = !add && par("ann"), args.legend = NULL, ...)

## S3 method for class 'formula'
barplot(formula, data, subset, na.action,
        horiz = FALSE, xlab = NULL, ylab = NULL, ...)
```

Arguments

height	either a vector or matrix of values describing the bars which make up the plot. If height is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If height is a matrix and beside is FALSE then each bar of the plot corresponds to a column of height, with the values in the column giving the heights of stacked sub-bars making up the bar. If height is a matrix and beside is TRUE, then the values in each column are juxtaposed rather than stacked.
width	optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will have no visible effect unless xlim is specified.
space	the amount of space (as a fraction of the average bar width) left before each bar. May be given as a single number or one number per bar. If height is a matrix and beside is TRUE, space may be specified by two numbers, where the first is the space between bars in the same group, and the second the space between the groups. If not given explicitly, it defaults to c(0,1) if height is a matrix and beside is TRUE, and to 0.2 otherwise.
names.arg	a vector of names to be plotted below each bar or group of bars. If this argument is omitted, then the names are taken from the names attribute of height if this is a vector, or the column names if it is a matrix.
legend.text	a vector of text used to construct a legend for the plot, or a logical indicating whether a legend should be included. This is only useful when height is a matrix. In that case given legend labels should correspond to the rows of height; if legend.text is true, the row names of height will be used as labels if they are non-null.
beside	a logical value. If FALSE, the columns of height are portrayed as stacked bars, and if TRUE the columns are portrayed as juxtaposed bars.
horiz	a logical value. If FALSE, the bars are drawn vertically with the first bar to the left. If TRUE, the bars are drawn horizontally with the first at the bottom.
density	a vector giving the density of shading lines, in lines per inch, for the bars or bar components. The default value of NULL means that no shading lines are drawn. Non-positive values of density also inhibit the drawing of shading lines.
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise), for the bars or bar components.
col	a vector of colors for the bars or bar components. By default, "grey" is used if height is a vector, and a gamma-corrected grey palette if height is a matrix; see grey.colors.
border	the color to be used for the border of the bars. Use border = NA to omit borders. If there are shading lines, border = TRUE means use the same colour for the border as for the shading lines.
main, sub	main title and subtitle for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
xlim	limits for the x axis.

ylim	limits for the y axis.
xpd	logical. Should bars be allowed to go outside region?
log	string specifying if axis scales should be logarithmic; see plot.default.
axes	logical. If TRUE, a vertical (or horizontal, if horiz is true) axis is drawn.
axisnames	logical. If TRUE, and if there are names.arg (see above), the other axis is drawn (with lty = 0) and labeled.
cex.axis	expansion factor for numeric axis labels (see par('cex')).
cex.names	expansion factor for axis names (bar labels).
inside	logical. If TRUE, the lines which divide adjacent (non-stacked!) bars will be drawn. Only applies when space = 0 (which it partly is when beside = TRUE).
plot	logical. If FALSE, nothing is plotted.
axis.lty	the graphics parameter lty (see par('lty')) applied to the axis and tick marks of the categorical (default horizontal) axis. Note that by default the axis is suppressed.
offset	a vector indicating how much the bars should be shifted relative to the x axis.
add	logical specifying if bars should be added to an already existing plot; defaults to FALSE.
ann	logical specifying if the default annotation (main, sub, xlab, ylab) should appear on the plot, see title.
args.legend	list of additional arguments to pass to legend(); names of the list are used as argument names. Only used if legend.text is supplied.

formula	a formula where the y variables are numeric data to plot against the categorical x variables. The formula can have one of three forms:
	`y ~ x`
	`y ~ x1 + x2`
	`cbind(y1, y2) ~ x`
	(see the examples).

data	a data frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NA values. The default is to ignore missing values in the given variables.
...	arguments to be passed to/from other methods. For the default method these can include further arguments (such as axes, asp and main) and graphical parameters (see par) which are passed to plot.window(), title() and axis.

Value

A numeric vector (or matrix, when beside = TRUE), say mp, giving the coordinates of *all* the bar midpoints drawn, useful for adding to the graph.

If beside is true, use colMeans(mp) for the midpoints of each *group* of bars, see example.

Author(s)

R Core, with a contribution by Arni Magnusson.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot(..., type = "h")`, `dotchart`; `hist` for bars of a *continuous* variable. `mosaicplot()`, more sophisticated to visualize *several* categorical variables.

Examples

```
# Formula method
barplot(GNP ~ Year, data = longley)
barplot(cbind(Employed, Unemployed) ~ Year, data = longley)

## 3rd form of formula - 2 categories :
op <- par(mfrow = 2:1, mgp = c(3,1,0)/2, mar = .1+c(3,3:1))
summary(d.Titanic <- as.data.frame(Titanic))
barplot(Freq ~ Class + Survived, data = d.Titanic,
        subset = Age == "Adult" & Sex == "Male",
     main = "barplot(Freq ~ Class + Survived, *)", ylab = "# {passengers}", legend.text = TRUE)
# Corresponding table :
(xt <- xtabs(Freq ~ Survived + Class + Sex, d.Titanic, subset = Age=="Adult"))
# Alternatively, a mosaic plot :
mosaicplot(xt[,,"Male"], main = "mosaicplot(Freq ~ Class + Survived, *)", color=TRUE)
par(op)


# Default method
require(grDevices) # for colours
tN <- table(Ni <- stats::rpois(100, lambda = 5))
r <- barplot(tN, col = rainbow(20))
#- type = "h" plotting *is* 'bar'plot
lines(r, tN, type = "h", col = "red", lwd = 2)

barplot(tN, space = 1.5, axisnames = FALSE,
        sub = "barplot(..., space= 1.5, axisnames = FALSE)")

barplot(VADeaths, plot = FALSE)
barplot(VADeaths, plot = FALSE, beside = TRUE)

mp <- barplot(VADeaths) # default
tot <- colMeans(VADeaths)
text(mp, tot + 3, format(tot), xpd = TRUE, col = "blue")
barplot(VADeaths, beside = TRUE,
        col = c("lightblue", "mistyrose", "lightcyan",
```

```
                "lavender", "cornsilk"),
        legend.text = rownames(VADeaths), ylim = c(0, 100))
title(main = "Death Rates in Virginia", font.main = 4)

hh <- t(VADeaths)[, 5:1]
mybarcol <- "gray20"
mp <- barplot(hh, beside = TRUE,
        col = c("lightblue", "mistyrose",
                "lightcyan", "lavender"),
        legend.text = colnames(VADeaths), ylim = c(0,100),
        main = "Death Rates in Virginia", font.main = 4,
        sub = "Faked upper 2*sigma error bars", col.sub = mybarcol,
        cex.names = 1.5)
segments(mp, hh, mp, hh + 2*sqrt(1000*hh/100), col = mybarcol, lwd = 1.5)
stopifnot(dim(mp) == dim(hh))  # corresponding matrices
mtext(side = 1, at = colMeans(mp), line = -2,
      text = paste("Mean", formatC(colMeans(hh))), col = "red")

# Bar shading example
barplot(VADeaths, angle = 15+10*1:5, density = 20, col = "black",
        legend.text = rownames(VADeaths))
title(main = list("Death Rates in Virginia", font = 4))

# Border color
barplot(VADeaths, border = "dark blue")

# Log scales (not much sense here)
barplot(tN, col = heat.colors(12), log = "y")
barplot(tN, col = gray.colors(20), log = "xy")

# Legend location
barplot(height = cbind(x = c(465, 91) / 465 * 100,
                       y = c(840, 200) / 840 * 100,
                       z = c(37, 17) / 37 * 100),
        beside = FALSE,
        width = c(465, 840, 37),
        col = c(1, 2),
        legend.text = c("A", "B"),
        args.legend = list(x = "topleft"))
```

---

box *Draw a Box around a Plot*

Description

This function draws a box around the current plot in the given color and line type. The `bty` parameter determines the type of box drawn. See `par` for details.

Usage

```
box(which = "plot", lty = "solid", ...)
```

Arguments

which	character, one of "plot", "figure", "inner" and "outer".
lty	line type of the box.
...	further graphical parameters, such as bty, col, or lwd, see par. Note that xpd is not accepted as clipping is always to the device region.

Details

The choice of colour is complicated. If col was supplied and is not NA, it is used. Otherwise, if fg was supplied and is not NA, it is used. The final default is par("col").

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

rect for drawing of arbitrary rectangles.

Examples

```
plot(1:7, abs(stats::rnorm(7)), type = "h", axes = FALSE)
axis(1, at = 1:7, labels = letters[1:7])
box(lty = '1373', col = 'red')
```

---

boxplot *Box Plots*

Description

Produce box-and-whisker plot(s) of the given (grouped) values.

Usage

```
boxplot(x, ...)

## S3 method for class 'formula'
boxplot(formula, data = NULL, ..., subset, na.action = NULL,
        xlab = mklab(y_var = horizontal),
        ylab = mklab(y_var =!horizontal),
        add = FALSE, ann = !add, horizontal = FALSE,
        drop = FALSE, sep = ".", lex.order = FALSE)

## Default S3 method:
boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,
        notch = FALSE, outline = TRUE, names, plot = TRUE,
```

```
          border = par("fg"), col = "lightgray", log = "",
          pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5),
           ann = !add, horizontal = FALSE, add = FALSE, at = NULL)
```

Arguments

formula	a formula, such as y ~ grp, where y is a numeric vector of data values to be split into groups according to the grouping variable grp (usually a factor). Note that ~ g1 + g2 is equivalent to g1:g2.
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
xlab, ylab	x- and y-axis annotation, since R 3.6.0 with a non-empty default. Can be suppressed by ann=FALSE.
ann	logical indicating if axes should be annotated (by xlab and ylab).
drop, sep, lex.order	passed to split.default, see there.
x	for specifying data from which the boxplots are to be produced. Either a numeric vector, or a single list containing such vectors. Additional unnamed arguments specify further data as separate vectors (each corresponding to a component boxplot). NAs are allowed in the data.
...	For the formula method, named arguments to be passed to the default method. For the default method, unnamed arguments are additional data vectors (unless x is a list when they are ignored), and named arguments are arguments and graphical parameters to be passed to bxp in addition to the ones given by argument pars (and override those in pars). Note that bxp may or may not make use of graphical parameters it is passed: see its documentation.
range	this determines how far the plot whiskers extend out from the box. If range is positive, the whiskers extend to the most extreme data point which is no more than range times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.
width	a vector giving the relative widths of the boxes making up the plot.
varwidth	if varwidth is TRUE, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
notch	if notch is TRUE, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ (Chambers et al., 1983, p. 62). See boxplot.stats for the calculations used.
outline	if outline is not true, the outliers are not drawn (as points whereas S+ uses lines).
names	group labels which will be printed under each boxplot. Can be a character vector or an expression (see plotmath).
boxwex	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.

staplewex	staple line width expansion, proportional to box width.
outwex	outlier line width expansion, proportional to box width.
plot	if TRUE (the default) then a boxplot is produced. If not, the summaries which the boxplots are based on are returned.
border	an optional vector of colors for the outlines of the boxplots. The values in border are recycled if the length of border is less than the number of plots.
col	if col is non-null it is assumed to contain colors to be used to colour the bodies of the box plots. By default they are in the background colour.
log	character indicating if x or y or both coordinates should be plotted in log scale.
pars	a list of (potentially many) more graphical parameters, e.g., boxwex or outpch; these are passed to bxp (if plot is true); for details, see there.
horizontal	logical indicating if the boxplots should be horizontal; default FALSE means vertical boxes.
add	logical, if true *add* boxplot to current plot.
at	numeric vector giving the locations where the boxplots should be drawn, particularly when add = TRUE; defaults to 1:n where n is the number of boxes.

Details

The generic function boxplot currently has a default method (boxplot.default) and a formula interface (boxplot.formula).

If multiple groups are supplied either as multiple arguments or via a formula, parallel boxplots will be plotted, in the order of the arguments or the order of the levels of the factor (see factor).

Missing values are ignored when forming boxplots.

Value

List with the following components:

stats	a matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot. If all the inputs have the same class attribute, so will this component.
n	a vector with the number of (non-NA) observations in each group.
conf	a matrix where each column contains the lower and upper extremes of the notch.
out	the values of any data points which lie beyond the extremes of the whiskers.
group	a vector of the same length as out whose elements indicate to which group the outlier belongs.
names	a vector of names for the groups.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

Murrell, P. (2005). *R Graphics*. Chapman & Hall/CRC Press.

See also `boxplot.stats`.

See Also

`boxplot.stats` which does the computation, `bxp` for the plotting and more examples; and `stripchart` for an alternative (with small data sets).

Examples

```
## boxplot on a formula:
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
# *add* notches (somewhat funny here <--> warning "notches .. outside hinges"):
boxplot(count ~ spray, data = InsectSprays,
        notch = TRUE, add = TRUE, col = "blue")

boxplot(decrease ~ treatment, data = OrchardSprays, col = "bisque",
        log = "y")
## horizontal=TRUE, switching  y <--> x :
boxplot(decrease ~ treatment, data = OrchardSprays, col = "bisque",
        log = "x", horizontal=TRUE)

rb <- boxplot(decrease ~ treatment, data = OrchardSprays, col = "bisque")
title("Comparing boxplot()s and non-robust mean +/- SD")
mn.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, mean)
sd.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, sd)
xi <- 0.3 + seq(rb$n)
points(xi, mn.t, col = "orange", pch = 18)
arrows(xi, mn.t - sd.t, xi, mn.t + sd.t,
       code = 3, col = "pink", angle = 75, length = .1)

## boxplot on a matrix:
mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
             `5T` = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
boxplot(mat) # directly, calling boxplot.matrix()

## boxplot on a data frame:
df. <- as.data.frame(mat)
par(las = 1) # all axis labels horizontal
boxplot(df., main = "boxplot(*, horizontal = TRUE)", horizontal = TRUE)

## Using 'at = ' and adding boxplots -- example idea by Roger Bivand :
boxplot(len ~ dose, data = ToothGrowth,
        boxwex = 0.25, at = 1:3 - 0.2,
        subset = supp == "VC", col = "yellow",
        main = "Guinea Pigs' Tooth Growth",
        xlab = "Vitamin C dose mg",
        ylab = "tooth length",
        xlim = c(0.5, 3.5), ylim = c(0, 35), yaxs = "i")
boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
        boxwex = 0.25, at = 1:3 + 0.2,
        subset = supp == "OJ", col = "orange")
```

```
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))

## With less effort (slightly different) using factor *interaction*:
boxplot(len ~ dose:supp, data = ToothGrowth,
        boxwex = 0.5, col = c("orange", "yellow"),
        main = "Guinea Pigs' Tooth Growth",
        xlab = "Vitamin C dose mg", ylab = "tooth length",
        sep = ":", lex.order = TRUE, ylim = c(0, 35), yaxs = "i")

## more examples in  help(bxp)
```

`boxplot.matrix` *Draw a Boxplot for each Column (Row) of a Matrix*

Description

Interpreting the columns (or rows) of a matrix as different groups, draw a boxplot for each.

Usage

```
## S3 method for class 'matrix'
boxplot(x, use.cols = TRUE, ...)
```

Arguments

`x`	a numeric matrix.
`use.cols`	logical indicating if columns (by default) or rows (`use.cols = FALSE`) should be plotted.
`...`	Further arguments to `boxplot`.

Value

A list as for `boxplot`.

Author(s)

Martin Maechler, 1995, for S+, then R package **sfsmisc**.

See Also

`boxplot.default` which already works nowadays with data.frames; `boxplot.formula`, `plot.factor` which work with (the more general concept) of a grouping factor.

Examples

```
## Very similar to the example in ?boxplot
mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
             T5 = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
boxplot(mat, main = "boxplot.matrix(...., main = ...)",
        notch = TRUE, col = 1:4)
```

---

bxp *Draw Box Plots from Summaries*

Description

bxp draws box plots based on the given summaries in z. It is usually called from within boxplot, but can be invoked directly.

Usage

```
bxp(z, notch = FALSE, width = NULL, varwidth = FALSE,
    outline = TRUE, notch.frac = 0.5, log = "",
    border = par("fg"), pars = NULL, frame.plot = axes,
    horizontal = FALSE, ann = TRUE,
    add = FALSE, at = NULL, show.names = NULL,
    ...)
```

Arguments

z	a list containing data summaries to be used in constructing the plots. These are usually the result of a call to boxplot, but can be generated in any fashion.
notch	if notch is TRUE, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap then the medians are significantly different at the 5 percent level.
width	a vector giving the relative widths of the boxes making up the plot.
varwidth	if varwidth is TRUE, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
outline	if outline is not true, the outliers are not drawn.
notch.frac	numeric in (0,1). When notch = TRUE, the fraction of the box width that the notches should use.
border	character or numeric (vector), the color of the box borders. Is recycled for multiple boxes. Is used as default for the boxcol, medcol, whiskcol, staplecol, and outcol options (see below).
log	character, indicating if any axis should be drawn in logarithmic scale, as in plot.default.
frame.plot	logical, indicating if a 'frame' (box) should be drawn; defaults to TRUE, unless axes = FALSE is specified.

`horizontal` logical indicating if the boxplots should be horizontal; default `FALSE` means vertical boxes.

`ann` a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.

`add` logical, if true *add* boxplot to current plot.

`at` numeric vector giving the locations where the boxplots should be drawn, particularly when `add = TRUE`; defaults to `1:n` where `n` is the number of boxes.

`show.names` Set to `TRUE` or `FALSE` to override the defaults on whether an x-axis label is printed for each group.

`pars, ...` graphical parameters (etc) can be passed as arguments to this function, either as a list (`pars`) or normally(`...`), see the following. (Those in `...` take precedence over those in `pars`.)

Currently, `yaxs` and `ylim` are used 'along the boxplot', i.e., vertically, when `horizontal` is false, and `xlim` horizontally. `xaxt`, `yaxt`, `las`, `cex.axis`, `gap.axis`, and `col.axis` are passed to `axis`, and `main`, `cex.main`, `col.main`, `sub`, `cex.sub`, `col.sub`, `xlab`, `ylab`, `cex.lab`, and `col.lab` are passed to `title`.

In addition, `axes` is accepted (see `plot.window`), with default `TRUE`.

The following arguments (or `pars` components) allow further customization of the boxplot graphics. Their defaults are typically determined from the non-prefixed version (e.g., `boxlty` from `lty`), either from the specified argument or `pars` component or the corresponding `par` one.

boxwex: a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower. The default depends on `at` and typically is 0.8.

staplewex, outwex: staple and outlier line width expansion, proportional to box width; both default to 0.5.

boxlty, boxlwd, boxcol, boxfill: box outline type, width, color, and fill color (which currently defaults to `col` and will in future default to `par("bg")`).

medlty, medlwd, medpch, medcex, medcol, medbg: median line type, line width, point character, point size expansion, color, and background color. The default `medpch = NA` suppresses the point, and `medlty = "blank"` does so for the line. Note that `medlwd` defaults to $3\times$ the default `lwd`.

whisklty, whisklwd, whiskcol: whisker line type (default: `"dashed"`), width, and color.

staplelty, staplelwd, staplecol: staple (= end of whisker) line type, width, and color.

outlty, outlwd, outpch, outcex, outcol, outbg: outlier line type, line width, point character, point size expansion, color, and background color. The default `outlty = "blank"` suppresses the lines and `outpch = NA` suppresses points.

Value

An invisible vector, actually identical to the `at` argument, with the coordinates ("x" if horizontal is false, "y" otherwise) of box centers, useful for adding to the plot.

Note

When add = FALSE, xlim now defaults to xlim = range(at, *) + c(-0.5, 0.5). It will usually be a good idea to specify xlim if the "x" axis has a log scale or width is far from uniform.

Author(s)

The R Core development team and Arni Magnusson (then at U Washington) who has provided most changes for the box*, med*, whisk*, staple*, and out* arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats)
set.seed(753)
(bx.p <- boxplot(split(rt(100, 4), gl(5, 20))))
op <- par(mfrow =  c(2, 2))
bxp(bx.p, xaxt = "n")
bxp(bx.p, notch = TRUE, axes = FALSE, pch = 4, boxfill = 1:5)
bxp(bx.p, notch = TRUE, boxfill = "lightblue", frame.plot = FALSE,
    outline = FALSE, main = "bxp(*, frame.plot= FALSE, outline= FALSE)")
bxp(bx.p, notch = TRUE, boxfill = "lightblue", border = 2:6,
    ylim = c(-4,4), pch = 22, bg = "green", log = "x",
    main = "... log = 'x', ylim = *")
par(op)
op <- par(mfrow = c(1, 2))

## single group -- no label
boxplot (weight ~ group, data = PlantGrowth, subset = group == "ctrl")
## with label
bx <- boxplot(weight ~ group, data = PlantGrowth,
              subset = group == "ctrl", plot = FALSE)
bxp(bx, show.names=TRUE)
par(op)

## passing gap.axis=* to axis(), PR#18109:
boxplot(matrix(100*rnorm(1e3), 50, 20),
        cex.axis = 1.5, gap.axis = -1)# showing *all* labels

z <- split(rnorm(1000), rpois(1000, 2.2))
boxplot(z, whisklty = 3, main = "boxplot(z, whisklty = 3)")

## Colour support similar to plot.default:
op <- par(mfrow = 1:2, bg = "light gray", fg = "midnight blue")
boxplot(z,   col.axis = "skyblue3", main = "boxplot(*, col.axis=..,main=..)")
plot(z[[1]], col.axis = "skyblue3", main =    "plot(*, col.axis=..,main=..)")
mtext("par(bg=\"light gray\", fg=\"midnight blue\")",
      outer = TRUE, line = -1.2)
par(op)
```

```
## Mimic S-Plus:
splus <- list(boxwex = 0.4, staplewex = 1, outwex = 1, boxfill = "grey40",
              medlwd = 3, medcol = "white", whisklty = 3, outlty = 1, outpch = NA)
boxplot(z, pars = splus)
## Recycled and "sweeping" parameters
op <- par(mfrow = c(1,2))
 boxplot(z, border = 1:5, lty = 3, medlty = 1, medlwd = 2.5)
 boxplot(z, boxfill = 1:3, pch = 1:5, lwd = 1.5, medcol = "white")
par(op)
## too many possibilities
boxplot(z, boxfill = "light gray", outpch = 21:25, outlty = 2,
        bg = "pink", lwd = 2,
        medcol = "dark blue", medcex = 2, medpch = 20)
```

---

cdplot *Conditional Density Plots*

Description

Computes and plots conditional densities describing how the conditional distribution of a categorical variable y changes over a numerical variable x.

Usage

```
cdplot(x, ...)

## Default S3 method:
cdplot(x, y,
  plot = TRUE, tol.ylab = 0.05, ylevels = NULL,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), weights = NULL, ...)

## S3 method for class 'formula'
cdplot(formula, data = list(),
  plot = TRUE, tol.ylab = 0.05, ylevels = NULL,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), ...,
  subset = NULL, weights = NULL)
```

Arguments

x	an object, the default method expects a single numerical variable (or an object coercible to this).
y	a "factor" interpreted to be the dependent variable

formula	a "formula" of type y ~ x with a single dependent "factor" and a single numerical explanatory variable.
data	an optional data frame.
plot	logical. Should the computed conditional densities be plotted?
tol.ylab	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
ylevels	a character or numeric vector specifying in which order the levels of the dependent variable should be plotted.
bw, n, from, to, ...	arguments passed to density
col	a vector of fill colors of the same length as levels(y). The default is to call gray.colors.
border	border color of shaded polygons.
main, xlab, ylab	character strings for annotation
yaxlabels	character vector for annotation of y axis, defaults to levels(y).
xlim, ylim	the range of x and y values with sensible defaults.
subset	an optional vector specifying a subset of observations to be used for plotting.
weights	numeric. A vector of frequency weights for each observation in the data. If NULL all weights are implicitly assumed to be 1.

Details

cdplot computes the conditional densities of x given the levels of y weighted by the marginal distribution of y. The densities are derived cumulatively over the levels of y.

This visualization technique is similar to spinograms (see spineplot) and plots $P(y|x)$ against x. The conditional probabilities are not derived by discretization (as in the spinogram), but using a smoothing approach via density.

Note, that the estimates of the conditional densities are more reliable for high-density regions of x. Conversely, the are less reliable in regions with only few x observations.

Value

The conditional density functions (cumulative over the levels of y) are returned invisibly.

Author(s)

Achim Zeileis <Achim.Zeileis@R-project.org>

References

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*, Unpublished Manuscript.

See Also

spineplot, density

Examples

```
## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1,
                 1, 2, 1, 1, 1, 1, 1),
               levels = 1:2, labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70,
                 70, 70, 72, 73, 75, 75, 76, 76, 78, 79, 81)

## CD plot
cdplot(fail ~ temperature)
cdplot(fail ~ temperature, bw = 2)
cdplot(fail ~ temperature, bw = "SJ")

## compare with spinogram
(spineplot(fail ~ temperature, breaks = 3))

## highlighting for failures
cdplot(fail ~ temperature, ylevels = 2:1)

## scatter plot with conditional density
cdens <- cdplot(fail ~ temperature, plot = FALSE)
plot(I(as.numeric(fail) - 1) ~ jitter(temperature, factor = 2),
     xlab = "Temperature", ylab = "Conditional failure probability")
lines(53:81, 1 - cdens[[1]](53:81), col = 2)
```

---

clip *Set Clipping Region*

Description

Set clipping region in user coordinates

Usage

```
clip(x1, x2, y1, y2)
```

Arguments

x1, x2, y1, y2	user coordinates of clipping rectangle

Details

How the clipping rectangle is set depends on the setting of `par("xpd")`: this function changes the current setting until the next high-level plotting command resets it.

Clipping of lines, rectangles and polygons is done in the graphics engine, but clipping of text is if possible done in the device, so the effect of clipping text is device-dependent (and may result in text not wholly within the clipping region being omitted entirely).

Exactly when the clipping region will be reset can be hard to predict. `plot.new` always resets it. Functions such as `lines` and `text` only reset it if `par("xpd")` has been changed. However, functions such as `box`, `mtext`, `title` and `plot.dendrogram` can manipulate the xpd setting.

See Also

par

Examples

```
x <- rnorm(1000)
hist(x, xlim = c(-4,4))
usr <- par("usr")
clip(usr[1], -2, usr[3], usr[4])
hist(x, col = 'red', add = TRUE)
clip(2, usr[2], usr[3], usr[4])
hist(x, col = 'blue', add = TRUE)
do.call("clip", as.list(usr))  # reset to plot region
```

---

contour *Display Contours*

Description

Create a contour plot, or add contour lines to an existing plot.

Usage

```
contour(x, ...)

## Default S3 method:
contour(x = seq(0, 1, length.out = nrow(z)),
        y = seq(0, 1, length.out = ncol(z)),
        z,
        nlevels = 10, levels = pretty(zlim, nlevels),
        labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont, axes = TRUE, frame.plot = axes,
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.

nlevels	number of contour levels desired **iff** levels is not supplied.
levels	numeric vector of levels at which to draw contour lines.
labels	a vector giving the labels for the contour lines. If NULL then the levels are used as labels, otherwise this is coerced by as.character.
labcex	cex for contour labelling. This is an absolute size, not a multiple of par("cex").
drawlabels	logical. Contours are labelled if TRUE.
method	character string specifying where the labels will be located. Possible values are "simple", "edge" and "flattest" (the default). See the 'Details' section.
vfont	if NULL, the current font family and face are used for the contour labels. If a character vector of length 2 then Hershey vector fonts are used for the contour labels. The first element of the vector selects a typeface and the second element selects a font index (see text for more information). The default is NULL on graphics devices with high-quality rotation of text and c("sans serif", "plain") otherwise.
xlim, ylim, zlim	x-, y- and z-limits for the plot.
axes, frame.plot	logical indicating whether axes or a box should be drawn, see plot.default.
col	colour(s) for the lines drawn.
lty	line type(s) for the lines drawn.
lwd	line width(s) for the lines drawn.
add	logical. If TRUE, add to a current plot.
...	additional arguments to plot.window, title, Axis and box, typically graphical parameters such as cex.axis.

Details

contour is a generic function with only a default method in base R.

The methods for positioning the labels on contours are "simple" (draw at the edge of the plot, overlaying the contour line), "edge" (draw at the edge of the plot, embedded in the contour line, with no labels overlapping) and "flattest" (draw on the flattest section of the contour, embedded in the contour line, with no labels overlapping). The second and third may not draw a label on every contour line.

For information about vector fonts, see the help for text and Hershey.

Notice that contour interprets the z matrix as a table of f(x[i], y[j]) values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree counter-clockwise rotation of the conventional textual layout.

Vector (of length > 1) col, lty, and lwd are applied along levels and recycled, see the Examples.

Alternatively, use contourplot from the **lattice** package where the formula notation allows to use vectors x, y, and z of the same length.

There is limited control over the axes and frame as arguments col, lwd and lty refer to the contour lines (rather than being general graphical parameters). For more control, add contours to a plot, or add axes and frame to a contour plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`options("max.contour.segments")` for the maximal complexity of a single contour line.

`contourLines`, `filled.contour` for color-filled contours, `contourplot` (and `levelplot`) from package **lattice**. Further, `image` and the graphics demo which can be invoked as `demo(graphics)`.

Examples

```
require(grDevices) # for colours
x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
z <- outer(x, sqrt(abs(x)), FUN = `/`)
image(x, x, z)
contour(x, x, z, col = "pink", add = TRUE, method = "edge",
        vfont = c("sans serif", "plain"))
contour(x, x, z, ylim = c(1, 6), method = "simple", labcex = 1,
        xlab = quote(x[1]), ylab = quote(x[2]))
contour(x, x, z, ylim = c(-6, 6), nlevels = 20, lty = 2, method = "simple",
        main = "20 levels; \"simple\" labelling method")
par(op)

## Passing multiple colours / lty / lwd :
op <- par(mfrow = c(1, 2))
z <- outer(-9:25, -9:25)
## Using default levels <- pretty(range(z, finite = TRUE), 10),
##  the first and last of which typically are *not* drawn:
(levs <- pretty(z, n=10)) # -300 -200 ... 600 700
contour(z, col = 1:4)
## Set levels explicitly; show that 'lwd' and 'lty' are recycled as well:
contour(z, levels=levs[-c(1,length(levs))], col = 1:5, lwd = 1:3 *1.5, lty = 1:3)
par(op)

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, length.out = 27)
r <- sqrt(outer(x^2, y^2, `+`))
opar <- par(mfrow = c(2, 2), mar = rep(0, 4))
for(f in pi^(0:3))
  contour(cos(r^2)*exp(-r/f),
          drawlabels = FALSE, axes = FALSE, frame.plot = TRUE)

rx <- range(x <- 10*1:nrow(volcano))
ry <- range(y <- 10*1:ncol(volcano))
ry <- ry + c(-1, 1) * (diff(rx) - diff(ry))/2
tcol <- terrain.colors(12)
par(opar); opar <- par(pty = "s", bg = "lightcyan")
plot(x = 0, y = 0, type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
```

```
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
title("A Topographic Map of Maunga Whau", font = 4)
abline(h = 200*0:4, v = 200*0:4, col = "lightgray", lty = 2, lwd = 0.1)

## contourLines produces the same contour lines as contour
plot(x = 0, y = 0, type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[1], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
line.list <- contourLines(x, y, volcano)
invisible(lapply(line.list, lines, lwd=3, col=adjustcolor(2, .3)))
par(opar)
```

---

convertXY *Convert between Graphics Coordinate Systems*

Description

Convert between graphics coordinate systems.

Usage

```
grconvertX(x, from = "user", to = "user")
grconvertY(y, from = "user", to = "user")
```

Arguments

x, y	numeric vector of coordinates.
from, to	character strings giving the coordinate systems to convert between.

Details

The coordinate systems are

"user" user coordinates.

"inches" inches.

"device" the device coordinate system.

"ndc" normalized device coordinates.

"nfc" normalized figure coordinates.

"npc" normalized plot coordinates.

"nic" normalized inner region coordinates. (The 'inner region' is that inside the outer margins.)

"lines" lines of margin (based on mex).

`"chars"` lines of text (based on `cex`).

(These names can be partially matched.) For the 'normalized' coordinate systems the lower left has value 0 and the top right value 1.

Device coordinates are those in which the device works: they are usually in pixels where that makes sense and in big points (1/72 inch) otherwise (e.g., `pdf` and `postscript`).

Value

A numeric vector of the same length as the input.

Examples

```
op <- par(omd=c(0.1, 0.9, 0.1, 0.9), mfrow = c(1, 2))
plot(1:4)
for(tp in c("in", "dev", "ndc", "nfc", "npc", "nic", "lines", "chars"))
    print(grconvertX(c(1.0, 4.0), "user", tp))
par(op)
```

---

coplot *Conditioning Plots*

Description

This function produces two variants of the **co**nditioning plots discussed in the reference below.

Usage

```
coplot(formula, data, given.values, panel = points, rows, columns,
       show.given = TRUE, col = par("fg"), pch = par("pch"),
       bar.bg = c(num = gray(0.8), fac = gray(0.95)),
       xlab = c(x.name, paste("Given :", a.name)),
       ylab = c(y.name, paste("Given :", b.name)),
       subscripts = FALSE,
       axlabels = function(f) abbreviate(levels(f)),
       number = 6, overlap = 0.5, xlim, ylim, ...)
co.intervals(x, number = 6, overlap = 0.5)
```

Arguments

`formula`	a formula describing the form of conditioning plot. A formula of the form `y ~ x | a` indicates that plots of `y` versus `x` should be produced conditional on the variable `a`. A formula of the form `y ~ x | a * b` indicates that plots of `y` versus `x` should be produced conditional on the two variables `a` and `b`. All three or four variables may be either numeric or factors. When `x` or `y` are factors, the result is almost as if `as.numeric()` was applied, whereas for factor `a` or `b`, the conditioning (and its graphics if `show.given` is true) are adapted.

data	a data frame containing values for any variables in the formula. By default the environment where coplot was called from is used.
given.values	a value or list of two values which determine how the conditioning on a and b is to take place. When there is no b (i.e., conditioning only on a), usually this is a matrix with two columns each row of which gives an interval, to be conditioned on, but is can also be a single vector of numbers or a set of factor levels (if the variable being conditioned on is a factor). In this case (no b), the result of co.intervals can be used directly as given.values argument.
panel	a function(x, y, col, pch, ...) which gives the action to be carried out in each panel of the display. The default is points.
rows	the panels of the plot are laid out in a rows by columns array. rows gives the number of rows in the array.
columns	the number of columns in the panel layout array.
show.given	logical (possibly of length 2 for 2 conditioning variables): should conditioning plots be shown for the corresponding conditioning variables (default TRUE).
col	a vector of colors to be used to plot the points. If too short, the values are recycled.
pch	a vector of plotting symbols or characters. If too short, the values are recycled.
bar.bg	a named vector with components "num" and "fac" giving the background colors for the (shingle) bars, for **num**eric and **fac**tor conditioning variables respectively.
xlab	character; labels to use for the x axis and the first conditioning variable. If only one label is given, it is used for the x axis and the default label is used for the conditioning variable.
ylab	character; labels to use for the y axis and any second conditioning variable.
subscripts	logical: if true the panel function is given an additional (third) argument subscripts giving the subscripts of the data passed to that panel.
axlabels	function for creating axis (tick) labels when x or y are factors.
number	integer; the number of conditioning intervals, for a and b, possibly of length 2. It is only used if the corresponding conditioning variable is not a factor.
overlap	numeric < 1; the fraction of overlap of the conditioning variables, possibly of length 2 for x and y direction. When overlap < 0, there will be *gaps* between the data slices.
xlim	the range for the x axis.
ylim	the range for the y axis.
...	additional arguments to the panel function.
x	a numeric vector.

Details

In the case of a single conditioning variable a, when both rows and columns are unspecified, a 'close to square' layout is chosen with columns >= rows.

In the case of multiple rows, the *order* of the panel plots is from the bottom and from the left (corresponding to increasing a, typically).

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus plot and boxplot are not panel functions.

The rendering of arguments xlab and ylab is not controlled by par arguments cex.lab and font.lab even though they are plotted by mtext rather than title.

Value

co.intervals(., number, .) returns a (number × 2) matrix, say ci, where ci[k,] is the range of x values for the k-th interval.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Cleveland, W. S. (1993) *Visualizing Data.* New Jersey: Summit Press.

See Also

pairs, panel.smooth, points.

Examples

```
## Tonga Trench Earthquakes
coplot(lat ~ long | depth, data = quakes)
given.depth <- co.intervals(quakes$depth, number = 4, overlap = .1)
coplot(lat ~ long | depth, data = quakes, given.values = given.depth, rows = 1)

## Conditioning on 2 variables:
ll.dm <- lat ~ long | depth * mag
coplot(ll.dm, data = quakes)
coplot(ll.dm, data = quakes, number = c(4, 7), show.given = c(TRUE, FALSE))
coplot(ll.dm, data = quakes, number = c(3, 7),
       overlap = c(-.5, .1)) # negative overlap DROPS values

## given two factors
Index <- seq_len(nrow(warpbreaks)) # to get nicer default labels
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       show.given = 0:1)
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       col = "red", bg = "pink", pch = 21,
       bar.bg = c(fac = "light blue"))

## Example with empty panels:
with(data.frame(state.x77), {
coplot(Life.Exp ~ Income | Illiteracy * state.region, number = 3,
       panel = function(x, y, ...) panel.smooth(x, y, span = .8, ...))
## y ~ factor -- not really sensible, but 'show off':
coplot(Life.Exp ~ state.region | Income * state.division,
       panel = panel.smooth)
```

```
})
```

---

curve *Draw Function Plots*

Description

Draws a curve corresponding to a function over the interval `[from, to]`. `curve` can plot also an expression in the variable `xname`, default 'x'.

Usage

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", xname = "x", xlab = xname, ylab = NULL,
      log = NULL, xlim = NULL, ...)

## S3 method for class 'function'
plot(x, y = 0, to = 1, from = y, xlim = NULL, ylab = NULL, ...)
```

Arguments

`expr`	The name of a function, or a call or an expression written as a function of `x` which will evaluate to an object of the same length as `x`.
`x`	a 'vectorizing' numeric R function.
`y`	alias for `from` for compatibility with `plot`
`from, to`	the range over which the function will be plotted.
`n`	integer; the number of x values at which to evaluate.
`add`	logical; if `TRUE` add to an already existing plot; if `NA` start a new plot taking the defaults for the limits and log-scaling of the x-axis from the previous plot. Taken as `FALSE` (with a warning if a different value is supplied) if no graphics device is open.
`xlim`	`NULL` or a numeric vector of length 2; if non-`NULL` it provides the defaults for `c(from, to)` and, unless `add = TRUE`, selects the x-limits of the plot – see `plot.window`.
`type`	plot type: see `plot.default`.
`xname`	character string giving the name to be used for the x axis.
`xlab, ylab, log, ...`	labels and graphical parameters can also be specified as arguments. See 'Details' for the interpretation of the default for `log`. For the `"function"` method of `plot`, `...` can include any of the other arguments of `curve`, except `expr`.

Details

The function or expression expr (for curve) or function x (for plot) is evaluated at n points equally spaced over the range [from, to]. The points determined in this way are then plotted.

If either from or to is NULL, it defaults to the corresponding element of xlim if that is not NULL.

What happens when neither from/to nor xlim specifies both x-limits is a complex story. For plot(<function>) and for curve(add = FALSE) the defaults are $(0, 1)$. For curve(add = NA) and curve(add = TRUE) the defaults are taken from the x-limits used for the previous plot. (This differs from versions of R prior to 2.14.0.)

The value of log is used both to specify the plot axes (unless add = TRUE) and how 'equally spaced' is interpreted: if the x component indicates log-scaling, the points at which the expression or function is plotted are equally spaced on log scale.

The default value of log is taken from the current plot when add = TRUE, whereas if add = NA the x component is taken from the existing plot (if any) and the y component defaults to linear. For add = FALSE the default is ""

This used to be a quick hack which now seems to serve a useful purpose, but can give bad results for functions which are not smooth.

For expensive-to-compute expressions, you should use smarter tools.

The way curve handles expr has caused confusion. It first looks to see if expr is a name (also known as a symbol), in which case it is taken to be the name of a function, and expr is replaced by a call to expr with a single argument with name given by xname. Otherwise it checks that expr is either a call or an expression, and that it contains a reference to the variable given by xname (using all.vars): anything else is an error. Then expr is evaluated in an environment which supplies a vector of name given by xname of length n, and should evaluate to an object of length n. Note that this means that curve(x, ...) is taken as a request to plot a function named x (and it is used as such in the function method for plot).

The plot method can be called directly as plot.function.

Value

A list with components x and y of the points that were drawn is returned invisibly.

Warning

For historical reasons, add is allowed as an argument to the "function" method of plot, but its behaviour may surprise you. It is recommended to use add only with curve.

See Also

splinefun for spline interpolation, lines.

Examples

```
plot(qnorm) # default range c(0, 1) is appropriate here,
            # but end values are -/+Inf and so are omitted.
plot(qlogis, main = "The Inverse Logit : qlogis()")
abline(h = 0, v = 0:2/2, lty = 3, col = "gray")
```

```
curve(sin, -2*pi, 2*pi, xname = "t")
curve(tan, xname = "t", add = NA,
      main = "curve(tan)  --> same x-scale as previous plot")

op <- par(mfrow = c(2, 2))
curve(x^3 - 3*x, -2, 2)
curve(x^2 - 2, add = TRUE, col = "violet")

## simple and advanced versions, quite similar:
plot(cos, -pi,  3*pi)
curve(cos, xlim = c(-pi, 3*pi), n = 1001, col = "blue", add = TRUE)

chippy <- function(x) sin(cos(x)*exp(-x/2))
curve(chippy, -8, 7, n = 2001)
plot (chippy, -8, -5)

for(ll in c("", "x", "y", "xy"))
   curve(log(1+x), 1, 100, log = ll, sub = paste0("log = '", ll, "'"))
par(op)
```

---

dotchart *Cleveland's Dot Plots*

Description

Draw a Cleveland dot plot.

Usage

```
dotchart(x, labels = NULL, groups = NULL, gdata = NULL, offset = 1/8,
         ann = par("ann"), xaxt = par("xaxt"), frame.plot = TRUE, log = "",
         cex = par("cex"), pt.cex = cex,
         pch = 21, gpch = 21, bg = par("bg"),
         color = par("fg"), gcolor = par("fg"), lcolor = "gray",
         xlim = range(x[is.finite(x)]),
         main = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

x	either a vector or matrix of numeric values (NAs are allowed). If x is a matrix the overall plot consists of juxtaposed dotplots for each row. Inputs which satisfy is.numeric(x) but not is.vector(x) || is.matrix(x) are coerced by as.numeric, with a warning.
labels	a vector of labels for each point. For vectors the default is to use names(x) and for matrices the row labels dimnames(x)[[1]].
groups	an optional factor indicating how the elements of x are grouped. If x is a matrix, groups will default to the columns of x.

gdata	data values for the groups. This is typically a summary such as the median or mean of each group.
offset	offset in inches of ylab and labels.
ann	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
xaxt	a string indicating the x-axis style; use "n" to suppress and see also par("xaxt").
frame.plot	a logical indicating whether a box should be drawn around the plot.
log	a character string indicating if one or the other axis should be logarithmic, see plot.default.
cex	the character size to be used. Setting cex to a value smaller than one can be a useful way of avoiding label overlap. Unlike many other graphics functions, this sets the actual size, not a multiple of par("cex").
pt.cex	the cex to be applied to plotting symbols. This behaves like cex in plot().
pch	the plotting character or symbol to be used.
gpch	the plotting character or symbol to be used for group values.
bg	the background color of plotting characters or symbols to be used; use par(bg= *) to set the background color of the whole plot.
color	the color(s) to be used for points and labels.
gcolor	the single color to be used for group labels and values.
lcolor	the color(s) to be used for the horizontal lines.
xlim	horizontal range for the plot, see plot.window, for example.
main	overall title for the plot, see title.
xlab, ylab	axis annotations as in title.
...	graphical parameters can also be specified as arguments.

Value

This function is invoked for its side effect, which is to produce two variants of dotplots as described in Cleveland (1985).

Dot plots are a reasonable substitute for bar plots.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

Examples

```
dotchart(VADeaths, main = "Death Rates in Virginia - 1940")

op <- par(xaxs = "i")  # 0 -- 100%
dotchart(t(VADeaths), xlim = c(0,100), bg = "skyblue",
         main = "Death Rates in Virginia - 1940", xlab = "rate [ % ]",
         ylab = "Grouping:  Age  x   Urbanity . Gender")
par(op)
```

`filled.contour` *Level (Contour) Plots*

Description

This function produces a contour plot with the areas between the contours filled in solid color (Cleveland calls this a level plot). A key showing how the colors map to z values is shown to the right of the plot.

Usage

```
filled.contour(x = seq(0, 1, length.out = nrow(z)),
               y = seq(0, 1, length.out = ncol(z)),
               z,
               xlim = range(x, finite = TRUE),
               ylim = range(y, finite = TRUE),
               zlim = range(z, finite = TRUE),
               levels = pretty(zlim, nlevels), nlevels = 20,
               color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE),
               col = color.palette(length(levels) - 1),
               plot.title, plot.axes, key.title, key.axes, key.border = NULL,
               asp = NA, xaxs = "i", yaxs = "i", las = 1,
               axes = TRUE, frame.plot = axes, ...)

.filled.contour(x, y, z, levels, col)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. (The rest of this description does not apply to `.filled.contour`.) By default, equally spaced values from 0 to 1 are used. If x is a `list`, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a numeric matrix containing the values to be plotted.. Note that x can be used instead of z for convenience.
xlim	x limits for the plot.
ylim	y limits for the plot.

zlim	z limits for the plot.
levels	a set of levels which are used to partition the range of z. Must be **strictly** increasing (and finite). Areas with z values between consecutive levels are painted with the same color.
nlevels	if levels is not specified, the range of z, values is divided into approximately this many levels.
color.palette	a color palette function to be used to assign colors in the plot.
col	an explicit set of colors to be used in the plot. This argument overrides any palette function specification. There should be one less color than levels
plot.title	statements which add titles to the main plot.
plot.axes	statements which draw axes (and a box) on the main plot. This overrides the default axes.
key.title	statements which add titles for the plot key.
key.axes	statements which draw axes on the plot key. This overrides the default axis.
key.border	color for the border of the key rect()angles.
asp	the y/x aspect ratio, see plot.window.
xaxs	the x axis style. The default is to use internal labeling.
yaxs	the y axis style. The default is to use internal labeling.
las	the style of labeling to be used. The default is to use horizontal labeling.
axes, frame.plot	logicals indicating if axes and a box should be drawn, as in plot.default.
...	additional graphical parameters, currently only passed to title().

Details

The values to be plotted can contain NAs. Rectangles with two or more corner values are NA are omitted entirely: where there is a single NA value the triangle opposite the NA is omitted.

Values to be plotted can be infinite: the effect is similar to that described for NA values.

.filled.contour is a 'bare bones' interface to add just the contour plot to an already-set-up plot region. It is is intended for programmatic use, and the programmer is responsible for checking the conditions on the arguments.

Note

filled.contour uses the layout function and so is restricted to a full page display.

The output produced by filled.contour is actually a combination of two plots; one is the filled contour and one is the legend. Two separate coordinate systems are set up for these two plots, but they are only used internally – once the function has returned these coordinate systems are lost. If you want to annotate the main contour plot, for example to add points, you can specify graphics commands in the plot.axes argument. See the examples.

Author(s)

Ross Ihaka and R Core Team

References

Cleveland, W. S. (1993) *Visualizing Data*. Summit, New Jersey: Hobart.

See Also

contour, image, hcl.colors, gray.colors, palette; contourplot and levelplot from package **lattice**.

Examples

```
require("grDevices") # for colours
filled.contour(volcano, asp = 1) # simple

x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
filled.contour(x, y, volcano,
    color.palette = function(n) hcl.colors(n, "terrain"),
    plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
    plot.axes = { axis(1, seq(100, 800, by = 100))
                  axis(2, seq(100, 600, by = 100)) },
    key.title = title(main = "Height\n(meters)"),
    key.axes = axis(4, seq(90, 190, by = 10)))  # maybe also asp = 1
mtext(paste("filled.contour(.) from", R.version.string),
      side = 1, line = 4, adj = 1, cex = .66)

# Annotating a filled contour plot
a <- expand.grid(1:20, 1:20)
b <- matrix(a[,1] + a[,2], 20)
filled.contour(x = 1:20, y = 1:20, z = b,
               plot.axes = { axis(1); axis(2); points(10, 10) })

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, length.out = 27)
r <- sqrt(outer(x^2, y^2, `+`))
## "minimal"
filled.contour(cos(r^2)*exp(-r/(2*pi)), axes = FALSE, key.border=NA)
## rather, the key *should* be labeled (but axes still not):
filled.contour(cos(r^2)*exp(-r/(2*pi)), frame.plot = FALSE,
               plot.axes = {})
```

---

fourfoldplot *Fourfold Plots*

Description

Creates a fourfold display of a 2 by 2 by k contingency table on the current graphics device, allowing for the visual inspection of the association between two dichotomous variables in one or several populations (strata).

Usage

```
fourfoldplot(x, color = c("#99CCFF", "#6699CC"),
             conf.level = 0.95,
             std = c("margins", "ind.max", "all.max"),
             margin = c(1, 2), space = 0.2, main = NULL,
             mfrow = NULL, mfcol = NULL)
```

Arguments

x	a 2 by 2 by k contingency table in array form, or as a 2 by 2 matrix if k is 1.
color	a vector of length 2 specifying the colors to use for the smaller and larger diagonals of each 2 by 2 table.
conf.level	confidence level used for the confidence rings on the odds ratios. Must be a single nonnegative number less than 1; if set to 0, confidence rings are suppressed.
std	a character string specifying how to standardize the table. Must match one of "margins", "ind.max", or "all.max", and can be abbreviated to the initial letter. If set to "margins", each 2 by 2 table is standardized to equate the margins specified by margin while preserving the odds ratio. If "ind.max" or "all.max", the tables are either individually or simultaneously standardized to a maximal cell frequency of 1.
margin	a numeric vector with the margins to equate. Must be one of 1, 2, or c(1, 2) (the default), which corresponds to standardizing the row, column, or both margins in each 2 by 2 table. Only used if std equals "margins".
space	the amount of space (as a fraction of the maximal radius of the quarter circles) used for the row and column labels.
main	character string for the fourfold title.
mfrow	a numeric vector of the form c(nr, nc), indicating that the displays for the 2 by 2 tables should be arranged in an nr by nc layout, filled by rows.
mfcol	a numeric vector of the form c(nr, nc), indicating that the displays for the 2 by 2 tables should be arranged in an nr by nc layout, filled by columns.

Details

The fourfold display is designed for the display of 2 by 2 by k tables.

Following suitable standardization, the cell frequencies f_{ij} of each 2 by 2 table are shown as a quarter circle whose radius is proportional to $\sqrt{f_{ij}}$ so that its area is proportional to the cell frequency. An association (odds ratio different from 1) between the binary row and column variables is indicated by the tendency of diagonally opposite cells in one direction to differ in size from those in the other direction; color is used to show this direction. Confidence rings for the odds ratio allow a visual test of the null of no association; the rings for adjacent quadrants overlap if and only if the observed counts are consistent with the null hypothesis.

Typically, the number k corresponds to the number of levels of a stratifying variable, and it is of interest to see whether the association is homogeneous across strata. The fourfold display visualizes the pattern of association. Note that the confidence rings for the individual odds ratios are not adjusted for multiple testing.

References

Friendly, M. (1994). A fourfold display for 2 by 2 by k tables. Technical Report 217, York University, Psychology Department. http://datavis.ca/papers/4fold/4fold.pdf

See Also

mosaicplot

Examples

```
## Use the Berkeley admission data as in Friendly (1995).
x <- aperm(UCBAdmissions, c(2, 1, 3))
dimnames(x)[[2]] <- c("Yes", "No")
names(dimnames(x)) <- c("Sex", "Admit?", "Department")
stats::ftable(x)

## Fourfold display of data aggregated over departments, with
## frequencies standardized to equate the margins for admission
## and sex.
## Figure 1 in Friendly (1994).
fourfoldplot(marginSums(x, c(1, 2)))

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission and sex.
## Figure 2 in Friendly (1994).
fourfoldplot(x)

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission. but not
## for sex.
## Figure 3 in Friendly (1994).
fourfoldplot(x, margin = 2)
```

---

frame *Create / Start a New Plot Frame*

Description

This function (`frame` is an alias for `plot.new`) causes the completion of plotting in the current plot (if there is one) and an advance to a new graphics frame. This is used in all high-level plotting functions and also useful for skipping plots when a multi-figure region is in use.

Usage

```
plot.new()
frame()
```

Details

The new page is painted with the background colour (`par("bg")`), which is often transparent. For devices with a *canvas* colour (the on-screen devices `X11`, `windows` and `quartz`), the window is first painted with the canvas colour and then the background colour.

There are two hooks called `"before.plot.new"` and `"plot.new"` (see `setHook`) called immediately before and after advancing the frame. The latter is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **graphics** namespace.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`frame`.)

See Also

`plot.window`, `plot.default`.

`grid` *Add Grid to a Plot*

Description

`grid` adds an `nx` by `ny` rectangular grid to an existing plot.

Usage

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",
     lwd = par("lwd"), equilogs = TRUE)
```

Arguments

`nx`, `ny`	number of cells of the grid in x and y direction. When `NULL`, as per default, the grid aligns with the tick marks on the corresponding *default* axis (i.e., tickmarks as computed by `axTicks`). When `NA`, no grid lines are drawn in the corresponding direction.
`col`	character or (integer) numeric; color of the grid lines.
`lty`	character or (integer) numeric; line type of the grid lines.
`lwd`	non-negative numeric giving line width of the grid lines.
`equilogs`	logical, only used when *log* coordinates and alignment with the axis tick marks are active. Setting `equilogs = FALSE` in that case gives *non equidistant* tick aligned grid lines.

Note

If more fine tuning is required, use `abline(h = ., v = .)` directly.

References

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

plot, abline, lines, points.

Examples

```
plot(1:3)
grid(NA, 5, lwd = 2) # grid only in y-direction

## maybe change the desired number of tick marks:  par(lab = c(mx, my, 7))
op <- par(mfcol = 1:2)
with(iris,
     {
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          xlim = c(4, 8), ylim = c(2, 4.5), panel.first = grid(),
          main = "with(iris,  plot(...., panel.first = grid(), ..) )")
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          panel.first = grid(3, lty = 1, lwd = 2),
          main = "... panel.first = grid(3, lty = 1, lwd = 2), ..")
     }
    )
par(op)

plot(1:64)
gr <- grid() # now *invisibly* returns the grid "at" locations
str(gr)
stopifnot(length(gr) == 2, identical(gr[[1]], gr[[2]]),
          gr[["atx"]] == 10*(0:6))

## In log-scale plots :
plot(8:270, log="xy") ; grid() # at (1, 10, 100); if preferring "all" grid lines:
plot(8:270, log="xy") ; grid(equilogs = FALSE) -> grll
stopifnot(identical(grll, list(atx = c(1, 2, 5, 10, 20, 50, 100, 200),
                               aty = c(         10, 20, 50, 100, 200))))
```

---

hist *Histograms*

Description

The generic function hist computes a histogram of the given data values. If plot = TRUE, the resulting object of class "histogram" is plotted by plot.histogram, before it is returned.

Usage

```
hist(x, ...)

## Default S3 method:
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE, fuzz = 1e-7,
     density = NULL, angle = 45, col = "lightgray", border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, warn.unused = TRUE, ...)
```

Arguments

`x`	a vector of values for which the histogram is desired.
`breaks`	one of: • a vector giving the breakpoints between histogram cells, • a function to compute the vector of breakpoints, • a single number giving the number of cells for the histogram, • a character string naming an algorithm to compute the number of cells (see 'Details'), • a function to compute the number of cells. In the last three cases the number is a suggestion only; as the breakpoints will be set to `pretty` values, the number is limited to `1e6` (with a warning if it was larger). If `breaks` is a function, the `x` vector is supplied to it as the only argument (and the number of breaks is only limited by the amount of available memory).
`freq`	logical; if `TRUE`, the histogram graphic is a representation of frequencies, the `counts` component of the result; if `FALSE`, probability densities, component `density`, are plotted (so that the histogram has a total area of one). Defaults to `TRUE` *if and only if* `breaks` are equidistant (and `probability` is not specified).
`probability`	an *alias* for `!freq`, for S compatibility.
`include.lowest`	logical; if `TRUE`, an `x[i]` equal to the `breaks` value will be included in the first (or last, for `right = FALSE`) bar. This will be ignored (with a warning) unless `breaks` is a vector.
`right`	logical; if `TRUE`, the histogram cells are right-closed (left open) intervals.
`fuzz`	non-negative number, for the case when the data is "pretty" and some observations `x[.]` are close but not exactly on a `break`. For counting fuzzy breaks proportional to `fuzz` are used. The default is occasionally suboptimal.
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).

col	a colour to be used to fill the bars.
border	the color of the border around the bars. The default is to use the standard foreground color.
main, xlab, ylab	main title and axis labels: these arguments to title() get "smart" defaults here, e.g., the default ylab is "Frequency" iff freq is true.
xlim, ylim	the range of x and y values with sensible defaults. Note that xlim is *not* used to define the histogram (breaks), but only for plotting (when plot = TRUE).
axes	logical. If TRUE (default), axes are draw if the plot is drawn.
plot	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned. In the latter case, a warning is used if (typically graphical) arguments are specified that only apply to the plot = TRUE case.
labels	logical or character string. Additionally draw labels on top of bars, if not FALSE; see plot.histogram.
nclass	numeric (integer). For S(-PLUS) compatibility only, nclass is equivalent to breaks for a scalar or character argument.
warn.unused	logical. If plot = FALSE and warn.unused = TRUE, a warning will be issued when graphical parameters are passed to hist.default().
...	further arguments and graphical parameters passed to plot.histogram and thence to title and axis (if plot = TRUE).

Details

The definition of *histogram* differs by source (with country-specific biases). R's default with equispaced breaks (also the default) is to plot the counts in the cells defined by breaks. Thus the height of a rectangle is proportional to the number of points falling into the cell, as is the area *provided* the breaks are equally-spaced.

The default with non-equispaced breaks is to give a plot of area one, in which the *area* of the rectangles is the fraction of the data points falling in the cells.

If right = TRUE (default), the histogram cells are intervals of the form $(a, b]$, i.e., they include their right-hand endpoint, but not their left one, with the exception of the first cell when include.lowest is TRUE.

For right = FALSE, the intervals are of the form $[a, b)$, and include.lowest means '*include highest*'.

A numerical tolerance of 10^{-7} times the median bin size (for more than four bins, otherwise the median is substituted) is applied when counting entries on the edges of bins. This is not included in the reported breaks nor in the calculation of density.

The default for breaks is "Sturges": see nclass.Sturges. Other names for which algorithms are supplied are "Scott" and "FD" / "Freedman-Diaconis" (with corresponding functions nclass.scott and nclass.FD). Case is ignored and partial matching is used. Alternatively, a function can be supplied which will compute the intended number of breaks or the actual breakpoints as a function of x.

Value

an object of class `"histogram"` which is a list with components:

`breaks`	the $n+1$ cell boundaries (= `breaks` if that was a vector). These are the nominal breaks, not with the boundary fuzz.
`counts`	n integers; for each cell, the number of `x[]` inside.
`density`	values $\hat{f}(x_i)$, as estimated density values. If `all(diff(breaks) == 1)`, they are the relative frequencies `counts/n` and in general satisfy $\sum_i \hat{f}(x_i)(b_{i+1} - b_i) = 1$, where b_i = `breaks[i]`.
`mids`	the n cell midpoints.
`xname`	a character string with the actual `x` argument name.
`equidist`	logical, indicating if the distances between `breaks` are all the same.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley. B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`nclass.Sturges`, `stem`, `density`, `truehist` in package **MASS**.

Typical plots with vertical bars are *not* histograms. Consider `barplot` or `plot(*, type = "h")` for such bar plots.

Examples

```
op <- par(mfrow = c(2, 2))
hist(islands)
utils::str(hist(islands, col = "gray", labels = TRUE))

hist(sqrt(islands), breaks = 12, col = "lightblue", border = "pink")
##-- For non-equidistant breaks, counts should NOT be graphed unscaled:
r <- hist(sqrt(islands), breaks = c(4*0:5, 10*3:5, 70, 100, 140),
          col = "blue1")
text(r$mids, r$density, r$counts, adj = c(.5, -.5), col = "blue3")
sapply(r[2:3], sum)
sum(r$density * diff(r$breaks)) # == 1
lines(r, lty = 3, border = "purple") # -> lines.histogram(*)
par(op)

require(utils) # for str
str(hist(islands, breaks = 12, plot =  FALSE)) #-> 10 (~= 12) breaks
str(hist(islands, breaks = c(12,20,36,80,200,1000,17000), plot = FALSE))

hist(islands, breaks = c(12,20,36,80,200,1000,17000), freq = TRUE,
     main = "WRONG histogram") # and warning
```

```
## Extreme outliers; the "FD" rule would take very large number of 'breaks':
XXL <- c(1:9, c(-1,1)*1e300)
hh <- hist(XXL, "FD") # did not work in R <= 3.4.1; now gives warning
## pretty() determines how many counts are used (platform dependently!):
length(hh$breaks) ## typically 1 million -- though 1e6 was "a suggestion only"

## R >= 4.2.0: no "*.5" labels on y-axis:
hist(c(2,3,3,5,5,6,6,6,7))

require(stats)
set.seed(14)
x <- rchisq(100, df = 4)

## Histogram with custom x-axis:
hist(x, xaxt = "n")
axis(1, at = 0:17)


## Comparing data with a model distribution should be done with qqplot()!
qqplot(x, qchisq(ppoints(x), df = 4)); abline(0, 1, col = 2, lty = 2)

## if you really insist on using hist() ... :
hist(x, freq = FALSE, ylim = c(0, 0.2))
curve(dchisq(x, df = 4), col = 2, lty = 2, lwd = 2, add = TRUE)
```

`hist.POSIXt` *Histogram of a Date or Date-Time Object*

Description

Methods for `hist` applied to date (class "`Date`") or date-time (class "`POSIXt`") objects.

Usage

```
## S3 method for class 'POSIXt'
hist(x, breaks, ...,
     xlab = deparse1(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format, right = TRUE)

## S3 method for class 'Date'
hist(x, breaks, ...,
     xlab = deparse1(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format, right = TRUE)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
breaks	a vector of cut points *or* number giving the number of intervals which x is to be cut into *or* an interval specification, one of "days", "weeks", "months", "quarters" or "years", plus "secs", "mins", "hours" for date-time objects.
...	graphical parameters, or arguments to hist.default such as include.lowest, density and labels.
xlab	a character string giving the label for the x axis, if plotted.
plot	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned.
freq	logical; if TRUE, the histogram graphic is a representation of frequencies, i.e, the counts component of the result; if FALSE, *relative* frequencies (probabilities) are plotted.
start.on.monday	logical. If breaks = "weeks", should the week start on Mondays or Sundays?
format	for the x-axis labels. See strptime.
right	logical; if TRUE, the histogram cells are right-closed (left open) intervals.

Details

Note that unlike the default method, breaks is a required argument.

Using breaks = "quarters" will create intervals of 3 calendar months, with the intervals beginning on January 1, April 1, July 1 or October 1, based upon min(x) as appropriate.

With the default right = TRUE, breaks will be set on the last day of the previous period when breaks is "months", "quarters" or "years". Use right = FALSE to set them to the first day of the interval shown in each bar.

Value

An object of class "histogram": see hist.

See Also

seq.POSIXt, axis.POSIXct, hist

Examples

```
hist(.leap.seconds, "years", freq = TRUE)
brks <- seq(ISOdate(1970, 1, 1), ISOdate(2030, 1, 1), "5 years")
hist(.leap.seconds, brks)
rug(.leap.seconds, lwd=2)
## show that  'include.lowest' "works"
stopifnot(identical(c(2L, rep(1L,11)),
   hist(brks, brks, plot=FALSE, include.lowest=TRUE )$counts))
tools::assertError(verbose=TRUE, ##--> 'breaks' do not span range of 'x'
   hist(brks, brks, plot=FALSE, include.lowest=FALSE))
## The default fuzz in hist.default()  "kills" this, with a "wrong" message:
```

```
try ( hist(brks[-13] + 1, brks, include.lowest = FALSE) )
## and decreasing 'fuzz' solves the issue:
hb <- hist(brks[-13] + 1, brks, include.lowest = FALSE, fuzz = 1e-10)
stopifnot(hb$counts == 1)

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*stats::runif(100)
hist(random.dates, "weeks", format = "%d %b")
```

`identify` *Identify Points in a Scatter Plot*

Description

`identify` reads the position of the graphics pointer when the (first) mouse button is pressed. It then searches the coordinates given in `x` and `y` for the point closest to the pointer. If this point is close enough to the pointer, its index will be returned as part of the value of the call.

Usage

```
identify(x, ...)

## Default S3 method:
identify(x, y = NULL, labels = seq_along(x), pos = FALSE,
         n = length(x), plot = TRUE, atpen = FALSE, offset = 0.5,
         tolerance = 0.25, order = FALSE, ...)
```

Arguments

`x, y`	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (a plotting structure, time series etc: see `xy.coords`) can be given as `x`, and `y` left missing.
`labels`	an optional character vector giving labels for the points. Will be coerced using `as.character`, and recycled if necessary to the length of `x`. Excess labels will be discarded, with a warning.
`pos`	if `pos` is `TRUE`, a component is added to the return value which indicates where text was plotted relative to each identified point: see Value.
`n`	the maximum number of points to be identified.
`plot`	logical: if `plot` is `TRUE`, the labels are printed near the points and if `FALSE` they are omitted.
`atpen`	logical: if `TRUE` and `plot = TRUE`, the lower-left corners of the labels are plotted at the points clicked rather than relative to the points.
`offset`	the distance (in character widths) which separates the label from identified points. Negative values are allowed. Not used if `atpen = TRUE`.
`tolerance`	the maximal distance (in inches) for the pointer to be 'close enough' to a point.
`order`	if `order` is `TRUE`, a component is added to the return value which indicates the order in which points were identified: see Value.
`...`	further arguments passed to `par` such as `cex`, `col` and `font`.

Details

`identify` is a generic function, and only the default method is described here.

`identify` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Clicking near (as defined by `tolerance`) a point adds it to the list of identified points. Points can be identified only once, and if the point has already been identified or the click is not near any of the points a message is printed immediately on the R console.

If `plot` is `TRUE`, the point is labelled with the corresponding element of `labels`. If `atpen` is false (the default) the labels are placed below, to the left, above or to the right of the identified point, depending on where the pointer was relative to the point. If `atpen` is true, the labels are placed with the bottom left of the string's box at the pointer.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing either the pop-up menu equivalent (usually second mouse button or `Ctrl`-click) or the `ESC` key.

On most devices which support `identify`, successful selection of a point is indicated by a bell sound unless `options(locatorBell = FALSE)` has been set.

If the window is resized or hidden and then exposed before the identification process has terminated, any labels drawn by `identify` will disappear. These will reappear once the identification process has terminated and the window is resized or hidden and exposed again. This is because the labels drawn by `identify` are not recorded in the device's display list until the identification process has terminated.

If you interrupt the `identify` call this leaves the graphics device in an undefined state, with points labelled but labels not recorded in the display list. Copying a device in that state will give unpredictable results.

Value

If both `pos` and `order` are `FALSE`, an integer vector containing the indices of the identified points.

If either of `pos` or `order` is `TRUE`, a list containing a component `ind`, indicating which points were identified and (if `pos` is `TRUE`) a component `pos`, indicating where the labels were placed relative to the identified points (1=below, 2=left, 3=above, 4=right and 0=no offset, used if `atpen = TRUE`) and (if `order` is `TRUE`) a component `order`, indicating the order in which points were identified.

Technicalities

The algorithm used for placing labels is the same as used by `text` if `pos` is specified there, the difference being that the position of the pointer relative the identified point determines `pos` in `identify`.

For labels placed to the left of a point, the right-hand edge of the string's box is placed `offset` units to the left of the point, and analogously for points to the right. The baseline of the text is placed below the point so as to approximately centre string vertically. For labels placed above or below a point, the string is centered horizontally on the point. For labels placed above, the baseline of the text is placed `offset` units above the point, and for those placed below, the baseline is placed so that the top of the string's box is approximately `offset` units below the point. If you want more precise placement (e.g., centering) use `plot = FALSE` and plot via `text` or `points`: see the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`locator`, `text`.

`dev.capabilities` to see if it is supported.

Examples

```
## A function to use identify to select points, and overplot the
## points with another symbol as they are selected
identifyPch <- function(x, y = NULL, n = length(x), plot = FALSE, pch = 19, ...)
{
    xy <- xy.coords(x, y); x <- xy$x; y <- xy$y
    sel <- rep(FALSE, length(x))
    while(sum(sel) < n) {
        ans <- identify(x[!sel], y[!sel], labels = which(!sel), n = 1, plot = plot, ...)
        if(!length(ans)) break
        ans <- which(!sel)[ans]
        points(x[ans], y[ans], pch = pch)
        sel[ans] <- TRUE
    }
    ## return indices of selected points
    which(sel)
}

if(dev.interactive()) { ## use it
  x <- rnorm(50); y <- rnorm(50)
  plot(x,y); identifyPch(x,y) # how fast to get all?
}
```

---

image *Display a Color Image*

Description

Creates a grid of colored or gray-scale rectangles with colors corresponding to the values in `z`. This can be used to display three-dimensional or spatial data aka *images*. This is a generic function.

NOTE: the grid is drawn as a set of rectangles by default; see the `useRaster` argument to draw the grid as a raster image.

The function `hcl.colors` provides a broad range of sequential color palettes that are suitable for displaying ordered data, with `n` giving the number of colors desired.

Usage

```
image(x, ...)

## Default S3 method:
image(x, y, z, zlim, xlim, ylim,
      col = hcl.colors(12, "YlOrRd", rev = TRUE),
      add = FALSE, xaxs = "i", yaxs = "i", xlab, ylab,
      breaks, oldstyle = FALSE, useRaster, ...)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be finite, non-missing and in (strictly) ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a numeric or logical matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
zlim	the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The *midpoints* of the intervals cover the range, so that values just outside the range will be plotted.
xlim, ylim	ranges for the plotted x and y values, defaulting to the ranges of x and y.
col	a list of colors such as that generated by hcl.colors, gray.colors or similar functions.
add	logical; if TRUE, add to current plot (and disregard the following four arguments). This is rarely useful because image 'paints' over existing graphics.
xaxs, yaxs	style of x and y axis. The default "i" is appropriate for images. See par.
xlab, ylab	each a character string giving the labels for the x and y axis. Default to the 'call names' of x or y, or to "" if these were unspecified.
breaks	a set of finite numeric breakpoints for the colours: must have one more breakpoint than colour and be in increasing order. Unsorted vectors will be sorted, with a warning.
oldstyle	logical. If true the midpoints of the colour intervals are equally spaced, and zlim[1] and zlim[2] were taken to be midpoints. The default is to have colour intervals of equal lengths between the limits.
useRaster	logical; if TRUE a bitmap raster is used to plot the image instead of polygons. The grid must be regular in that case, otherwise an error is raised. For the behaviour when this is not specified, see 'Details'.
...	graphical parameters for plot may also be passed as arguments to this function, as can the plot aspect ratio asp and axes (see plot.window).

Details

The length of x should be equal to the nrow(z)+1 or nrow(z). In the first case x specifies the boundaries between the cells: in the second case x specifies the midpoints of the cells. Similar

reasoning applies to y. It probably only makes sense to specify the midpoints of an equally-spaced grid. If you specify just one row or column and a length-one x or y, the whole user area in the corresponding direction is filled. For logarithmic x or y axes the boundaries between cells must be specified.

Rectangles corresponding to missing values are not plotted (and so are transparent and (unless add = TRUE) the default background painted in par("bg") will show through and if that is transparent, the canvas colour will be seen).

If breaks is specified then zlim is unused and the algorithm used follows cut, so intervals are closed on the right and open on the left except for the lowest interval which is closed at both ends.

The axes (where plotted) make use of the classes of xlim and ylim (and hence by default the classes of x and y): this will mean that for example dates are labelled as such.

Notice that image interprets the z matrix as a table of f(x[i], y[j]) values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree counter-clockwise rotation of the conventional printed layout of a matrix.

Images for large z on a regular grid are rendered more efficiently with useRaster = TRUE and can prevent rare anti-aliasing artifacts, but may not be supported by all graphics devices. Some devices (such as postscript and X11(type = "Xlib")) which do not support semi-transparent colours may emit missing values as white rather than transparent, and there may be limitations on the size of a raster image. (Problems with the rendering of raster images have been reported by users of windows() devices under Remote Desktop, at least under its default settings.)

The graphics files in PDF and PostScript can be much smaller under this option.

If useRaster is not specified, raster images are used when the getOption("preferRaster") is true, the grid is regular and either dev.capabilities("rasterImage")$rasterImage is "yes" or it is "non-missing" and there are no missing values.

Note

Originally based on a function by Thomas Lumley.

See Also

filled.contour or heatmap which can look nicer (but are less modular), contour; The **lattice** equivalent of image is levelplot.

hcl.colors, gray.colors, hcl, hsv, par.

dev.capabilities to see if useRaster = TRUE is supported on the current device.

Examples

```
require("grDevices") # for colours
x <- y <- seq(-4*pi, 4*pi, length.out = 27)
r <- sqrt(outer(x^2, y^2, `+`))
image(z = z <- cos(r^2)*exp(-r/6), col = gray.colors(33))
image(z, axes = FALSE, main = "Math can be beautiful ...",
      xlab = expression(cos(r^2) * e^{-r/6}))
contour(z, add = TRUE, drawlabels = FALSE)

# Visualize as matrix.  Need to transpose matrix and then flip it horizontally:
```

```
tf <- function(m) t(m)[, nrow(m):1]
imageM <- function(m, grid = max(dim(m)) <= 25, asp = (nrow(m)-1)/(ncol(m)-1), ...) {
    image(tf(m), asp=asp, axes = FALSE, ...)
    mAxis <- function(side, at, ...) # using 'j'
        axis(side, at=at, labels=as.character(j+1L), col="gray", col.axis=1, ...)
    n <- ncol(m); n1 <- n-1L; j <- 0L:n1; mAxis(1, at= j/n1)
    if(grid) abline(v = (0:n - .5)/n1, col="gray77", lty="dotted")
    n <- nrow(m); n1 <- n-1L; j <- 0L:n1; mAxis(2, at=1-j/n1, las=1)
    if(grid) abline(h = (0:n - .5)/n1, col="gray77", lty="dotted")
}
(m <- outer(1:5, 1:14))
imageM(m, main = "image(<5 x 14 matrix>)   with rows and columns")
imageM(volcano)

# A prettier display of the volcano
x <- 10*(1:nrow(volcano))
y <- 10*(1:ncol(volcano))
image(x, y, volcano, col = hcl.colors(100, "terrain"), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5),
        add = TRUE, col = "brown")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
title(main = "Maunga Whau Volcano", font.main = 4)
```

`layout` *Specifying Complex Plot Arrangements*

Description

`layout` divides the device up into as many rows and columns as there are in matrix `mat`, with the column-widths and the row-heights specified in the respective arguments.

Usage

```
layout(mat, widths = rep.int(1, ncol(mat)),
       heights = rep.int(1, nrow(mat)), respect = FALSE)

layout.show(n = 1)
lcm(x)
```

Arguments

`mat`	a matrix object specifying the location of the next N figures on the output device. Each value in the matrix must be `0` or a positive integer. If N is the largest positive integer in the matrix, then the integers $\{1, \ldots, N-1\}$ must also appear at least once in the matrix.
`widths`	a vector of values for the widths of columns on the device. Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the `lcm()` function (see examples).

heights	a vector of values for the heights of rows on the device. Relative and absolute heights can be specified, see widths above.
respect	either a logical value or a matrix object. If the latter, then it must have the same dimensions as mat and each value in the matrix must be either 0 or 1.
n	number of figures to plot.
x	a dimension to be interpreted as a number of centimetres.

Details

Figure i is allocated a region composed from a subset of these rows and columns, based on the rows and columns in which i occurs in mat.

The respect argument controls whether a unit column-width is the same physical measurement on the device as a unit row-height.

There is a limit (currently 200) for the numbers of rows and columns in the layout, and also for the total number of cells (10007).

layout.show(n) plots (part of) the current layout, namely the outlines of the next n figures.

lcm is a trivial function, to be used as *the* interface for specifying absolute dimensions for the widths and heights arguments of layout().

Value

layout returns the number of figures, N, see above.

Warnings

These functions are totally incompatible with the other mechanisms for arranging plots on a device: par(mfrow), par(mfcol) and split.screen.

Author(s)

Paul R. Murrell

References

Murrell, P. R. (1999). Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, **8**, 121–134. doi:10.2307/1390924.

Chapter 5 of Paul Murrell's Ph.D. thesis.

Murrell, P. (2005). *R Graphics*. Chapman & Hall/CRC Press.

See Also

par with arguments mfrow, mfcol, or mfg.

Examples

```
def.par <- par(no.readonly = TRUE) # save default, for resetting...

## divide the device into two rows and two columns
## allocate figure 1 all of row 1
## allocate figure 2 the intersection of column 2 and row 2
layout(matrix(c(1,1,0,2), 2, 2, byrow = TRUE))
## show the regions that have been allocated to each plot
layout.show(2)

## divide device into two rows and two columns
## allocate figure 1 and figure 2 as above
## respect relations between widths and heights
nf <- layout(matrix(c(1,1,0,2), 2, 2, byrow = TRUE), respect = TRUE)
layout.show(nf)

## create single figure which is 5cm square
nf <- layout(matrix(1), widths = lcm(5), heights = lcm(5))
layout.show(nf)


##-- Create a scatterplot with marginal histograms -----

x <- pmin(3, pmax(-3, stats::rnorm(50)))
y <- pmin(3, pmax(-3, stats::rnorm(50)))
xhist <- hist(x, breaks = seq(-3,3,0.5), plot = FALSE)
yhist <- hist(y, breaks = seq(-3,3,0.5), plot = FALSE)
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3, 3)
yrange <- c(-3, 3)
nf <- layout(matrix(c(2,0,1,3),2,2,byrow = TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)

par(mar = c(3,3,1,1))
plot(x, y, xlim = xrange, ylim = yrange, xlab = "", ylab = "")
par(mar = c(0,3,1,1))
barplot(xhist$counts, axes = FALSE, ylim = c(0, top), space = 0)
par(mar = c(3,0,1,1))
barplot(yhist$counts, axes = FALSE, xlim = c(0, top), space = 0, horiz = TRUE)

par(def.par)  #- reset to default
```

`legend` *Add Legends to Plots*

Description

This function can be used to add legends to plots. Note that a call to the function `locator(1)` can be used in place of the `x` and `y` arguments.

Usage

```
legend(x, y = NULL, legend, fill = NULL, col = par("col"),
       border = "black", lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
       box.lwd = par("lwd"), box.lty = par("lty"), box.col = par("fg"),
       pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
       xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
       adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
       text.font = NULL, merge = do.lines && has.pch, trace = FALSE,
       plot = TRUE, ncol = 1, horiz = FALSE, title = NULL,
       inset = 0, xpd, title.col = text.col[1], title.adj = 0.5,
       title.cex = cex[1], title.font = text.font[1],
       seg.len = 2)
```

Arguments

x, y	the x and y co-ordinates to be used to position the legend. They can be specified by keyword or in any way which is accepted by `xy.coords`: See 'Details'.
legend	a character or expression vector of length ≥ 1 to appear in the legend. Other objects will be coerced by `as.graphicsAnnot`.
fill	if specified, this argument will cause boxes filled with the specified colors (or shaded in the specified colors) to appear beside the legend text.
col	the color of points or lines appearing in the legend.
border	the border color for the boxes (used only if `fill` is specified).
lty, lwd	the line types and widths for lines appearing in the legend. One of these two *must* be specified for line drawing.
pch	the plotting symbols appearing in the legend, as numeric vector or a vector of 1-character strings (see `points`). Unlike `points`, this can all be specified as a single multi-character string. *Must* be specified for symbol drawing.
angle	angle of shading lines.
density	the density of shading lines, if numeric and positive. If `NULL` or negative or `NA` color filling is assumed.
bty	the type of box to be drawn around the legend. The allowed values are `"o"` (the default) and `"n"`.
bg	the background color for the legend box. (Note that this is only used if `bty != "n"`.)
box.lty, box.lwd, box.col	the line type, width and color for the legend box (if `bty = "o"`).
pt.bg	the background color for the `points`, corresponding to its argument `bg`.
cex	character expansion factor **relative** to current `par("cex")`. Used for text, and provides the default for `pt.cex`.
pt.cex	expansion factor(s) for the points.
pt.lwd	line width for the points, defaults to the one for lines, or if that is not set, to `par("lwd")`.

`xjust`	how the legend is to be justified relative to the legend x location. A value of 0 means left justified, 0.5 means centered and 1 means right justified.
`yjust`	the same as `xjust` for the legend y location.
`x.intersp`	character interspacing factor for horizontal (x) spacing between symbol and legend text.
`y.intersp`	vertical (y) distances (in lines of text shared above/below each legend entry). A vector with one element for each row of the legend can be used.
`adj`	numeric of length 1 or 2; the string adjustment for legend text. Useful for y-adjustment when `labels` are plotmath expressions.
`text.width`	the width of the legend text in x (`"user"`) coordinates. (Should be positive even for a reversed x axis.) Can be a single positive numeric value (same width for each column of the legend), a vector (one element for each column of the legend), `NULL` (default) for computing a proper maximum value of `strwidth(legend)`), or `NA` for computing a proper column wise maximum value of `strwidth(legend)`).
`text.col`	the color used for the legend text.
`text.font`	the font used for the legend text, see `text`.
`merge`	logical; if `TRUE`, merge points and lines but not filled boxes. Defaults to `TRUE` if there are points and lines.
`trace`	logical; if `TRUE`, shows how `legend` does all its magical computations.
`plot`	logical. If `FALSE`, nothing is plotted but the sizes are returned.
`ncol`	the number of columns in which to set the legend items (default is 1, a vertical legend).
`horiz`	logical; if `TRUE`, set the legend horizontally rather than vertically (specifying `horiz` overrides the `ncol` specification).
`title`	a character string or length-one expression giving a title to be placed at the top of the legend. Other objects will be coerced by `as.graphicsAnnot`.
`inset`	inset distance(s) from the margins as a fraction of the plot region when legend is placed by keyword.
`xpd`	if supplied, a value of the graphical parameter `xpd` to be used while the legend is being drawn.
`title.col`	color for `title`, defaults to `text.col[1]`.
`title.adj`	horizontal adjustment for `title`: see the help for `par("adj")`.
`title.cex`	expansion factor(s) for the title, defaults to `cex[1]`.
`title.font`	the font used for the legend title, defaults to `text.font[1]`, see `text`.
`seg.len`	the length of lines drawn to illustrate `lty` and/or `lwd` (in units of character widths).

Details

Arguments `x`, `y`, `legend` are interpreted in a non-standard way to allow the coordinates to be specified *via* one or two arguments. If `legend` is missing and `y` is not numeric, it is assumed that the second argument is intended to be `legend` and that the first argument specifies the coordinates.

The coordinates can be specified in any way which is accepted by `xy.coords`. If this gives the coordinates of one point, it is used as the top-left coordinate of the rectangle containing the legend. If it gives the coordinates of two points, these specify opposite corners of the rectangle (either pair of corners, in any order).

The location may also be specified by setting `x` to a single keyword from the list `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` and `"center"`. This places the legend on the inside of the plot frame at the given location. Partial argument matching is used. The optional `inset` argument specifies how far the legend is inset from the plot margins. If a single value is given, it is used for both margins; if two values are given, the first is used for x-distance, the second for y-distance.

Attribute arguments such as `col`, `pch`, `lty`, etc, are recycled if necessary: `merge` is not. Set entries of `lty` to `0` or set entries of `lwd` to `NA` to suppress lines in corresponding legend entries; set `pch` values to `NA` to suppress points.

Points are drawn *after* lines in order that they can cover the line with their background color `pt.bg`, if applicable.

See the examples for how to right-justify labels.

Since they are not used for Unicode code points, values `-31:-1` are silently omitted, as are `NA` and `""` values.

Value

A list with list components

`rect` a list with components

`w, h` positive numbers giving **w**idth and **h**eight of the legend's box.

`left, top` x and y coordinates of upper left corner of the box.

`text` a list with components

`x, y` numeric vectors of length `length(legend)`, giving the x and y coordinates of the legend's text(s).

returned invisibly.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `barplot` which uses `legend()`, and `text` for more examples of math expressions.

Examples

```
## Run the example in '?matplot' or the following:
leg.txt <- c("Setosa     Petals", "Setosa     Sepals",
             "Versicolor Petals", "Versicolor Sepals")
```

```
y.leg <- c(4.5, 3, 2.1, 1.4, .7)
cexv  <- c(1.2, 1, 4/5, 2/3, 1/2)
matplot(c(1, 8), c(0, 4.5), type = "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
for (i in seq(cexv)) {
  text  (1, y.leg[i] - 0.1, paste("cex=", formatC(cexv[i])), cex = 0.8, adj = 0)
  legend(3, y.leg[i], leg.txt, pch = "sSvV", col = c(1, 3), cex = cexv[i])
}
## cex *vector* [in R <= 3.5.1 has 'if(xc < 0)' w/ length(xc) == 2]
legend("right", leg.txt, pch = "sSvV", col = c(1, 3),
       cex = 1+(-1:2)/8, trace = TRUE)# trace: show computed lengths & coords

## 'merge = TRUE' for merging lines & points:
x <- seq(-pi, pi, length.out = 65)
for(reverse in c(FALSE, TRUE)) {  ## normal *and* reverse axes:
  F <- if(reverse) rev else identity
  plot(x, sin(x), type = "l", col = 3, lty = 2,
       xlim = F(range(x)), ylim = F(c(-1.2, 1.8)))
  points(x, cos(x), pch = 3, col = 4)
  lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
  title("legend('top', lty = c(2, -1, 1), pch = c(NA, 3, 4), merge = TRUE)",
        cex.main = 1.1)
  legend("top", c("sin", "cos", "tan"), col = c(3, 4, 6),
       text.col = "green4", lty = c(2, -1, 1), pch = c(NA, 3, 4),
       merge = TRUE, bg = "gray90", trace=TRUE)

} # for(..)

## right-justifying a set of labels: thanks to Uwe Ligges
x <- 1:5; y1 <- 1/x; y2 <- 2/x
plot(rep(x, 2), c(y1, y2), type = "n", xlab = "x", ylab = "y")
lines(x, y1); lines(x, y2, lty = 2)
temp <- legend("topright", legend = c(" ", " "),
               text.width = strwidth("1,000,000"),
               lty = 1:2, xjust = 1, yjust = 1, inset = 1/10,
               title = "Line Types", title.cex = 0.5, trace=TRUE)
text(temp$rect$left + temp$rect$w, temp$text$y,
     c("1,000", "1,000,000"), pos = 2)


##--- log scaled Examples ------------------------------
leg.txt <- c("a one", "a two")

par(mfrow = c(2, 2))
for(ll in c("","x","y","xy")) {
  plot(2:10, log = ll, main = paste0("log = '", ll, "'"))
  abline(1, 1)
  lines(2:3, 3:4, col = 2)
  points(2, 2, col = 3)
  rect(2, 3, 3, 2, col = 4)
  text(c(3,3), 2:3, c("rect(2,3,3,2, col=4)",
                      "text(c(3,3),2:3,\"c(rect(...)\")"), adj = c(0, 0.3))
  legend(list(x = 2,y = 8), legend = leg.txt, col = 2:3, pch = 1:2,
```

```
        lty = 1)  #, trace = TRUE)
} #      ^^^^^^^ to force lines -> automatic merge=TRUE
par(mfrow = c(1,1))

##-- Math expressions:  ------------------------------
x <- seq(-pi, pi, length.out = 65)
plot(x, sin(x), type = "l", col = 2, xlab = expression(phi),
     ylab = expression(f(phi)))
abline(h = -1:1, v = pi/2*(-6:6), col = "gray90")
lines(x, cos(x), col = 3, lty = 2)
ex.cs1 <- expression(plain(sin) * phi,  paste("cos", phi))  # 2 ways
utils::str(legend(-3, .9, ex.cs1, lty = 1:2, plot = FALSE,
          adj = c(0, 0.6)))  # adj y !
legend(-3, 0.9, ex.cs1, lty = 1:2, col = 2:3,  adj = c(0, 0.6))

require(stats)
x <- rexp(100, rate = .5)
hist(x, main = "Mean and Median of a Skewed Distribution")
abline(v = mean(x),   col = 2, lty = 2, lwd = 2)
abline(v = median(x), col = 3, lty = 3, lwd = 2)
ex12 <- expression(bar(x) == sum(over(x[i], n), i == 1, n),
                   hat(x) == median(x[i], i == 1, n))
utils::str(legend(4.1, 30, ex12, col = 2:3, lty = 2:3, lwd = 2))

## 'Filled' boxes -- see also example(barplot) which may call legend(*, fill=)
barplot(VADeaths)
legend("topright", rownames(VADeaths), fill = gray.colors(nrow(VADeaths)))

## Using 'ncol'
x <- 0:64/64
for(R in c(identity, rev)) { # normal *and* reverse x-axis works fine:
  xl <- R(range(x)); x1 <- xl[1]
matplot(x, outer(x, 1:7, function(x, k) sin(k * pi * x)), xlim=xl,
        type = "o", col = 1:7, ylim = c(-1, 1.5), pch = "*")
op <- par(bg = "antiquewhite1")
legend(x1, 1.5, paste("sin(", 1:7, "pi * x)"), col = 1:7, lty = 1:7,
       pch = "*", ncol = 4, cex = 0.8)
legend("bottomright", paste("sin(", 1:7, "pi * x)"), col = 1:7, lty = 1:7,
       pch = "*", cex = 0.8)
legend(x1, -.1, paste("sin(", 1:4, "pi * x)"), col = 1:4, lty = 1:4,
       ncol = 2, cex = 0.8)
legend(x1, -.4, paste("sin(", 5:7, "pi * x)"), col = 4:6,  pch = 24,
       ncol = 2, cex = 1.5, lwd = 2, pt.bg = "pink", pt.cex = 1:3)
par(op)

} # for(..)

## point covering line :
y <- sin(3*pi*x)
plot(x, y, type = "l", col = "blue",
    main = "points with bg & legend(*, pt.bg)")
points(x, y, pch = 21, bg = "white")
legend(.4,1, "sin(c x)", pch = 21, pt.bg = "white", lty = 1, col = "blue")
```

```
## legends with titles at different locations
plot(x, y, type = "n")
legend("bottomright", "(x,y)", pch=1, title= "bottomright")
legend("bottom",      "(x,y)", pch=1, title= "bottom")
legend("bottomleft",  "(x,y)", pch=1, title= "bottomleft")
legend("left",        "(x,y)", pch=1, title= "left")
legend("topleft",     "(x,y)", pch=1, title= "topleft, inset = .05", inset = .05)
legend("top",         "(x,y)", pch=1, title= "top")
legend("topright",    "(x,y)", pch=1, title= "topright, inset = .02",inset = .02)
legend("right",       "(x,y)", pch=1, title= "right")
legend("center",      "(x,y)", pch=1, title= "center")

# using text.font (and text.col):
op <- par(mfrow = c(2, 2), mar = rep(2.1, 4))
c6 <- terrain.colors(10)[1:6]
for(i in 1:4) {
   plot(1, type = "n", axes = FALSE, ann = FALSE); title(paste("text.font =",i))
   legend("top", legend = LETTERS[1:6], col = c6,
          ncol = 2, cex = 2, lwd = 3, text.font = i, text.col = c6)
}
par(op)

# using text.width for several columns
plot(1, type="n")
legend("topleft", c("This legend", "has", "equally sized", "columns."),
       pch = 1:4, ncol = 4)
legend("bottomleft", c("This legend", "has", "optimally sized", "columns."),
       pch = 1:4, ncol = 4, text.width = NA)
legend("right", letters[1:4], pch = 1:4, ncol = 4,
       text.width = 1:4 / 50)
```

`lines` *Add Connected Line Segments to a Plot*

Description

A generic function taking coordinates given in various ways and joining the corresponding points with line segments.

Usage

```
lines(x, ...)

## Default S3 method:
lines(x, y = NULL, type = "l", ...)
```

Arguments

x, y	coordinate vectors of points to join.
type	character indicating the type of plotting; actually any of the types as in plot.default.
...	Further graphical parameters (see par) may also be supplied as arguments, particularly, line type, lty, line width, lwd, color, col and for type = "b", pch. Also the line characteristics lend, ljoin and lmitre.

Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, a time series, ... See xy.coords. If supplied separately, they must be of the same length.

The coordinates can contain NA values. If a point contains NA in either its x or y value, it is omitted from the plot, and lines are not drawn to or from such points. Thus missing values can be used to achieve breaks in lines.

For type = "h", col can be a vector and will be recycled as needed.

lwd can be a vector: its first element will apply to lines but the whole vector to symbols (recycled as necessary).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

lines.formula for the formula method; points, particularly for type %in% c("p","b","o"), plot, and the workhorse function plot.xy.

abline for drawing (single) straight lines.

par for line type (lty) specification and how to specify colors.

Examples

```
# draw a smooth line through a scatter plot
plot(cars, main = "Stopping Distance versus Speed")
lines(stats::lowess(cars))
```

---

locator *Graphical Input*

Description

Reads the position of the graphics cursor when the (first) mouse button is pressed.

Usage

```
locator(n = 512, type = "n", ...)
```

Arguments

`n`	the maximum number of points to locate. Valid values start at 1.
`type`	One of `"n"`, `"p"`, `"l"` or `"o"`. If `"p"` or `"o"` the points are plotted; if `"l"` or `"o"` they are joined by lines.
`...`	additional graphics parameters used if `type != "n"` for plotting the locations.

Details

`locator` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Unless the process is terminated prematurely by the user (see below) at most `n` positions are determined.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

The current graphics parameters apply just as if `plot.default` has been called with the same value of `type`. The plotting of the points and lines is subject to clipping, but locations outside the current clipping rectangle will be returned.

On most devices which support `locator`, successful selection of a point is indicated by a bell sound unless `options(locatorBell = FALSE)` has been set.

If the window is resized or hidden and then exposed before the input process has terminated, any lines or points drawn by `locator` will disappear. These will reappear once the input process has terminated and the window is resized or hidden and exposed again. This is because the points and lines drawn by `locator` are not recorded in the device's display list until the input process has terminated.

Value

A list containing `x` and `y` components which are the coordinates of the identified points in the user coordinate system, i.e., the one specified by `par("usr")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`identify`. `grid.locator` is the corresponding **grid** package function.

`dev.capabilities` to see if it is supported.

matplot *Plot Columns of Matrices*

Description

Plot the columns of one matrix against the columns of another (which often is just a vector treated as 1-column matrix).

Usage

```
matplot(x, y, type = "p", lty = 1:5, lwd = 1, lend = par("lend"),
        pch = NULL,
        col = 1:6, cex = NULL, bg = NA,
        xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
        log = "", ..., add = FALSE, verbose = getOption("verbose"))

matpoints(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)

matlines (x, y, type = "l", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

Arguments

x, y	vectors or matrices of data for plotting. The number of rows should match. If one of them are missing, the other is taken as y and an x vector of 1:n is used. Missing values (NAs) are allowed. Typically, class(.)es of x and y such as "Date" are preserved.
type	character string (length 1 vector) or vector of 1-character strings indicating the type of plot for each column of y, see plot for all possible types. The first character of type defines the first plot, the second character the second, etc. Characters in type are cycled through; e.g., "pl" alternately plots points and lines.
lty, lwd, lend	vector of line types, widths, and end styles. The first element is for the first column, the second element for the second column, etc., even if lines are not plotted for all columns. Line types will be used cyclically until all plots are drawn.
pch	character string or vector of 1-characters or integers for plotting characters, see points for details. The first character is the plotting-character for the first plot, the second for the second, etc. The default is the digits (1 through 9, 0) then the lowercase and uppercase letters.
col	vector of colors. Colors are used cyclically.
cex	vector of character expansion sizes, used cyclically. This works as a multiple of par("cex"). NULL is equivalent to 1.0.

bg	vector of background (fill) colors for the open plot symbols given by pch = 21:25 as in points. The default NA corresponds to the one of the underlying function plot.xy.
xlab, ylab	titles for x and y axes, as in plot.
xlim, ylim	ranges of x and y axes, as in plot.
log, ...	Graphical parameters (see par) and any further arguments of plot, typically plot.default, may also be supplied as arguments to this function; even panel.first etc now work. Hence, the high-level graphics control arguments described under par and the arguments to title may be supplied to this function.
add	logical. If TRUE, plots are added to current one, using points and lines.
verbose	logical. If TRUE, write one line of what is done.

Details

matplot(x,y, ..) is basically a wrapper for

1. calling (the generic function) plot(x[,1], y[,1], ..) for the first columns (only if add = TRUE).
2. calling (the generic) lines(x[,j], y[,j], ..) for subsequent columns.

Care is taken to keep the class(.) of x and y, such that the corresponding plot() and lines() *methods* will be called.

Points involving missing values are not plotted.

The first column of x is plotted against the first column of y, the second column of x against the second column of y, etc. If one matrix has fewer columns, plotting will cycle back through the columns again. (In particular, either x or y may be a vector, against which all columns of the other argument will be plotted.)

The first element of col, cex, lty, lwd is used to plot the axes as well as the first line.

Because plotting symbols are drawn with lines and because these functions may be changing the line style, you should probably specify lty = 1 when using plotting symbols.

Side Effects

Function matplot generates a new plot; matpoints and matlines add to the current one.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

plot, points, lines, matrix, par.

Examples

```
require(grDevices)
matplot((-4:5)^2, main = "Quadratic") # almost identical to plot(*)
sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, pch = 1:4, type = "o", col = rainbow(ncol(sines)))
matplot(sines, type = "b", pch = 21:23, col = 2:5, bg = 2:5,
        main = "matplot(...., pch = 21:23, bg = 2:5)")

x <- 0:50/50
matplot(x, outer(x, 1:8, function(x, k) sin(k*pi * x)),
        ylim = c(-2,2), type = "plobcsSh",
        main= "matplot(,type = \"plobcsSh\" )")
## pch & type =  vector of 1-chars :
matplot(x, outer(x, 1:4, function(x, k) sin(k*pi * x)),
        pch = letters[1:4], type = c("b","p","o"))

lends <- c("round","butt","square")
matplot(matrix(1:12, 4), type="c", lty=1, lwd=10, lend=lends)
text(cbind(2.5, 2*c(1,3,5)-.4), lends, col= 1:3, cex = 1.5)

table(iris$Species) # is data.frame with 'Species' factor
iS <- iris$Species == "setosa"
iV <- iris$Species == "versicolor"
op <- par(bg = "bisque")
matplot(c(1, 8), c(0, 4.5), type =  "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
matpoints(iris[iS,c(1,3)], iris[iS,c(2,4)], pch = "sS", col = c(2,4))
matpoints(iris[iV,c(1,3)], iris[iV,c(2,4)], pch = "vV", col = c(2,4))
legend(1, 4, c("    Setosa Petals", "    Setosa Sepals",
               "Versicolor Petals", "Versicolor Sepals"),
       pch = "sSvV", col = rep(c(2,4), 2))

nam.var <- colnames(iris)[-5]
nam.spec <- as.character(iris[1+50*0:2, "Species"])
iris.S <- array(NA, dim = c(50,4,3),
                dimnames = list(NULL, nam.var, nam.spec))
for(i in 1:3) iris.S[,,i] <- data.matrix(iris[1:50+50*(i-1), -5])

matplot(iris.S[, "Petal.Length",], iris.S[, "Petal.Width",], pch = "SCV",
        col = rainbow(3, start = 0.8, end = 0.1),
        sub = paste(c("S", "C", "V"), dimnames(iris.S)[[3]],
                    sep = "=", collapse= ",  "),
        main = "Fisher's Iris Data")
par(op)

## 'x' a "Date" vector :
nd <- length(dv <- seq(as.Date("1959-02-21"), by = "weeks", length.out = 100))
mSC <- cbind(I=1, sin=sin(pi*(1:nd)/8), cos=cos(pi*(1:nd)/8))
matplot(dv, mSC, type = "b", main = "matplot(<Date>, y)")

## 'x' a "POSIXct" date-time vector :
ct <- seq(c(ISOdate(2000,3,20)), by = "15 mins", length.out = 100)
```

```
matplot(ct, mSC, type = "b", main = "matplot(<POSIXct>, y)")
## or the same with even more axis flexibility:
matplot(ct, mSC, type = "b", main = "matplot(<POSIXct>, y)", xaxt="n")
Axis(ct, side=1, at = ct[1+4*(0:24)])

## Also works for data frame columns:
matplot(iris[1:50,1:4])
```

`mosaicplot` *Mosaic Plots*

Description

Plots a mosaic on the current graphics device.

Usage

```
mosaicplot(x, ...)

## Default S3 method:
mosaicplot(x, main = deparse1(substitute(x)),
           sub = NULL, xlab = NULL, ylab = NULL,
           sort = NULL, off = NULL, dir = NULL,
           color = NULL, shade = FALSE, margin = NULL,
           cex.axis = 0.66, las = par("las"), border = NULL,
           type = c("pearson", "deviance", "FT"), ...)

## S3 method for class 'formula'
mosaicplot(formula, data = NULL, ...,
           main = deparse1(substitute(data)), subset,
           na.action = stats::na.omit)
```

Arguments

`x`	a contingency table in array form, with optional category labels specified in the `dimnames(x)` attribute. The table is best created by the `table()` command.
`main`	character string for the mosaic title.
`sub`	character string for the mosaic sub-title (at bottom).
`xlab, ylab`	x- and y-axis labels used for the plot; by default, the first and second element of `names(dimnames(X))` (i.e., the name of the first and second variable in `X`).
`sort`	vector ordering of the variables, containing a permutation of the integers `1:length(dim(x))` (the default).
`off`	vector of offsets to determine percentage spacing at each level of the mosaic (appropriate values are between 0 and 20, and the default is 20 times the number of splits for 2-dimensional tables, and 10 otherwise). Rescaled to maximally 50, and recycled if necessary.

dir	vector of split directions ("v" for vertical and "h" for horizontal) for each level of the mosaic, one direction for each dimension of the contingency table. The default consists of alternating directions, beginning with a vertical split.
color	logical or (recycling) vector of colors for color shading, used only when shade is FALSE, or NULL (default). By default, grey boxes are drawn. color = TRUE uses grey.colors for a gamma-corrected grey palette. color = FALSE gives empty boxes with no shading.
shade	a logical indicating whether to produce extended mosaic plots, or a numeric vector of at most 5 distinct positive numbers giving the absolute values of the cut points for the residuals. By default, shade is FALSE, and simple mosaics are created. Using shade = TRUE cuts absolute values at 2 and 4.
margin	a list of vectors with the marginal totals to be fit in the log-linear model. By default, an independence model is fitted. See loglin for further information.
cex.axis	The magnification to be used for axis annotation, as a multiple of par("cex").
las	numeric; the style of axis labels, see par.
border	colour of borders of cells: see polygon.
type	a character string indicating the type of residual to be represented. Must be one of "pearson" (giving components of Pearson's χ^2), "deviance" (giving components of the likelihood ratio χ^2), or "FT" for the Freeman-Tukey residuals. The value of this argument can be abbreviated.
formula	a formula, such as y ~ x.
data	a data frame (or list), or a contingency table from which the variables in formula should be taken.
...	further arguments to be passed to or from methods.
subset	an optional vector specifying a subset of observations in the data frame to be used for plotting.
na.action	a function which indicates what should happen when the data contains variables to be cross-tabulated, and these variables contain NAs. The default is to omit cases which have an NA in any variable. Since the tabulation will omit all cases containing missing values, this will only be useful if the na.action function replaces missing values.

Details

This is a generic function. It currently has a default method (mosaicplot.default) and a formula interface (mosaicplot.formula).

Extended mosaic displays visualize standardized residuals of a loglinear model for the table by color and outline of the mosaic's tiles. (Standardized residuals are often referred to a standard normal distribution.) Cells representing negative residuals are drawn in shaded of red and with broken borders; positive ones are drawn in blue with solid borders.

For the formula method, if data is an object inheriting from class "table" or class "ftable" or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be non-negative. In this case the left-hand side of formula should be empty and the variables on the right-hand side should be taken from the names of the dimnames attribute of the contingency table. A marginal table of these variables is computed, and a mosaic plot of that table is produced.

Otherwise, `data` should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables given in `formula`, and a mosaic is produced from this.

See Emerson (1998) for more information and a case study with television viewer data from Nielsen Media Research.

Missing values are not supported except via an `na.action` function when `data` contains variables to be cross-tabulated.

A more flexible and extensible implementation of mosaic plots written in the grid graphics system is provided in the function `mosaic` in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2006).

Author(s)

S-PLUS original by John Emerson <john.emerson@yale.edu>. Originally modified and enhanced for R by Kurt Hornik.

References

Hartigan, J.A., and Kleiner, B. (1984). A mosaic of television ratings. *The American Statistician*, **38**, 32–35. doi:10.2307/2683556.

Emerson, J. W. (1998). Mosaic displays in S-PLUS: A general implementation and a case study. *Statistical Computing and Graphics Newsletter (ASA)*, **9**, 1, 17–23.

Friendly, M. (1994). Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200. doi:10.2307/2291215.

Meyer, D., Zeileis, A., and Hornik, K. (2006) The strucplot Framework: Visualizing Multi-Way Contingency Tables with **vcd**. *Journal of Statistical Software*, **17(3)**, 1–48. doi:10.18637/jss.v017.i03.

See Also

`assocplot`, `loglin`.

Examples

```
require(stats)
mosaicplot(Titanic, main = "Survival on the Titanic", color = TRUE)
## Formula interface for tabulated data:
mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)

mosaicplot(HairEyeColor, shade = TRUE)
## Independence model of hair and eye color and sex.  Indicates that
## there are more blue eyed blonde females than expected in the case
## of independence and too few brown eyed blonde females.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1, 2, 3))
pchisq(fm$pearson, fm$df, lower.tail = FALSE)

mosaicplot(HairEyeColor, shade = TRUE, margin = list(1:2, 3))
## Model of joint independence of sex from hair and eye color.  Males
```

```
## are underrepresented among people with brown hair and eyes, and are
## overrepresented among people with brown hair and blue eyes.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1:2, 3))
pchisq(fm$pearson, fm$df, lower.tail = FALSE)

## Formula interface for raw data: visualize cross-tabulation of numbers
## of gears and carburettors in Motor Trend car data.
mosaicplot(~ gear + carb, data = mtcars, color = TRUE, las = 1)
# color recycling
mosaicplot(~ gear + carb, data = mtcars, color = 2:3, las = 1)
```

`mtext` *Write Text into the Margins of a Plot*

Description

Text is written in one of the four margins of the current figure region or one of the outer margins of the device region.

Usage

```
mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)
```

Arguments

`text`	a character or expression vector specifying the *text* to be written. Other objects are coerced by `as.graphicsAnnot`.
`side`	on which side of the plot (1=bottom, 2=left, 3=top, 4=right).
`line`	on which MARgin line, starting at 0 counting outwards.
`outer`	use outer margins if available.
`at`	give location of each string in user coordinates. If the component of `at` corresponding to a particular text item is not a finite value (the default), the location will be determined by `adj`.
`adj`	adjustment for each string in reading direction. For strings parallel to the axes, `adj = 0` means left or bottom alignment, and `adj = 1` means right or top alignment. If `adj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted parallel to the axis the default is to centre the string.
`padj`	adjustment for each string perpendicular to the reading direction (which is controlled by `adj`). For strings parallel to the axes, `padj = 0` means left or bottom alignment, and `padj = 1` means right or top alignment (relative to the line). If `padj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

cex	character expansion factor. NULL and NA are equivalent to 1.0. This is an absolute measure, not scaled by par("cex") or by setting par("mfrow") or par("mfcol"). Can be a vector.
col	color to use. Can be a vector. NA values (the default) mean use par("col").
font	font for text. Can be a vector. NA values (the default) mean use par("font").
...	Further graphical parameters (see par), including family, las and xpd. (The latter defaults to the figure region unless outer = TRUE, otherwise the device region. It can only be increased.)

Details

The user coordinates in the outer margins always range from zero to one, and are not affected by the user coordinates in the figure region(s) — R differs here from other implementations of S.

All of the named arguments can be vectors, and recycling will take place to plot as many strings as the longest of the vector arguments.

Note that a vector adj has a different meaning from text. adj = 0.5 will centre the string, but for outer = TRUE on the device region rather than the plot region.

Parameter las will determine the orientation of the string(s). For strings plotted perpendicular to the axis the default justification is to place the end of the string nearest the axis on the specified line. (Note that this differs from S, which uses srt if at is supplied and las if it is not. Parameter srt is ignored in R.)

Note that if the text is to be plotted perpendicular to the axis, adj determines the justification of the string *and* the position along the axis unless at is specified.

Graphics parameter "ylbias" (see par) determines how the text baseline is placed relative to the nominal line.

Side Effects

The given text is written onto the current plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

title, text, plot, par; plotmath for details on mathematical annotation.

Examples

```
plot(1:10, (-4:5)^2, main = "Parabola Points", xlab = "xlab")
mtext("10 of them")
for(s in 1:4)
    mtext(paste("mtext(..., line= -1, {side, col, font} = ", s,
          ", cex = ", (1+s)/2, ")"), line = -1,
          side = s, col = s, font = s, cex = (1+s)/2)
mtext("mtext(..., line= -2)", line = -2)
```

```
mtext("mtext(..., line= -2, adj = 0)", line = -2, adj = 0)
##--- log axis :
plot(1:10, exp(1:10), log = "y", main = "log =\"y\"", xlab = "xlab")
for(s in 1:4) mtext(paste("mtext(...,side=", s ,")"), side = s)
```

---

pairs *Scatterplot Matrices*

Description

A matrix of scatterplots is produced.

Usage

```
pairs(x, ...)

## S3 method for class 'formula'
pairs(formula, data = NULL, ..., subset,
      na.action = stats::na.pass)

## Default S3 method:
pairs(x, labels, panel = points, ...,
      horInd = 1:nc, verInd = 1:nc,
      lower.panel = panel, upper.panel = panel,
      diag.panel = NULL, text.panel = textPanel,
      label.pos = 0.5 + has.diag/3, line.main = 3,
      cex.labels = NULL, font.labels = 1,
      row1attop = TRUE, gap = 1, log = "",
      horOdd = !row1attop, verOdd = !row1attop)
```

Arguments

x	the coordinates of points given as numeric columns of a matrix or data frame. Logical and factor columns are converted to numeric in the same way that data.matrix does.
formula	a formula, such as ~ x + y + z. Each term will give a separate variable in the pairs plot, so terms should be numeric vectors. (A response will be interpreted as another variable, but not treated specially, so it is confusing to use one.)
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to pass missing values on to the panel functions, but na.action = na.omit will cause cases with missing values in any of the variables to be omitted entirely.
labels	the names of the variables.

panel	function(x, y, ...) which is used to plot the contents of each panel of the display.
...	arguments to be passed to or from methods. Also, graphical parameters can be given as can arguments to plot such as main. par("oma") will be set appropriately unless specified.
horInd, verInd	The (numerical) indices of the variables to be plotted on the horizontal and vertical axes respectively.
lower.panel, upper.panel	separate panel functions (or NULL) to be used below and above the diagonal respectively.
diag.panel	optional function(x, ...) to be applied on the diagonals.
text.panel	optional function(x, y, labels, cex, font, ...) to be applied on the diagonals.
label.pos	y position of labels in the text panel.
line.main	if main is specified, line.main gives the line argument to mtext() which draws the title. You may want to specify oma when changing line.main.
cex.labels, font.labels	graphics parameters for the text panel.
row1attop	logical. Should the layout be matrix-like with row 1 at the top, or graph-like with row 1 at the bottom? The latter (non default) leads to a basically symmetric scatterplot matrix.
gap	distance between subplots, in margin lines.
log	a character string indicating if logarithmic axes are to be used, see plot.default *or* a numeric vector of indices specifying the indices of those variables where logarithmic axes should be used for both x and y. log = "xy" specifies logarithmic axes for all variables.
horOdd, verOdd	logical (or integer) determining how the horizontal and vertical axis labeling happens. If true, the axis labelling starts at the first (from top left) row or column, respectively.

Details

The ij-th scatterplot contains x[,i] plotted against x[,j]. The scatterplot can be customised by setting panel functions to appear as something completely different. The off-diagonal panel functions are passed the appropriate columns of x as x and y: the diagonal panel function (if any) is passed a single column, and the text.panel function is passed a single (x, y) location and the column name. Setting some of these panel functions to NULL is equivalent to *not* drawing anything there.

The graphical parameters pch and col can be used to specify a vector of plotting symbols and colors to be used in the plots.

The graphical parameter oma will be set by pairs.default unless supplied as an argument.

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus plot and boxplot are not panel functions.

By default, missing values are passed to the panel functions and will often be ignored within a panel. However, for the formula method and `na.action = na.omit`, all cases which contain a missing values for any of the variables are omitted completely (including when the scales are selected).

Arguments `horInd` and `verInd` were introduced in R 3.2.0. If given the same value they can be used to select or re-order variables: with different ranges of consecutive values they can be used to plot rectangular windows of a full pairs plot; in the latter case 'diagonal' refers to the diagonal of the full plot.

Author(s)

Enhancements for R 1.0.0 contributed by Dr. Jens Oehlschlägel-Akiyoshi and R-core members.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])

## formula method, "graph" layout (row 1 at bottom):
pairs(~ Fertility + Education + Catholic, data = swiss, row1attop=FALSE,
      subset = Education < 20, main = "Swiss data, Education < 20")

pairs(USJudgeRatings, gap=1/10) # (gap: not wasting plotting area)
## show only lower triangle (and suppress labeling for whatever reason):
pairs(USJudgeRatings, text.panel = NULL, upper.panel = NULL)

## put histograms on the diagonal
panel.hist <- function(x, ...)
{
    usr <- par("usr")
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col = "cyan", ...)
}
pairs(USJudgeRatings[1:5], panel = panel.smooth,
      cex = 1.5, pch = 24, bg = "light blue", horOdd=TRUE,
      diag.panel = panel.hist, cex.labels = 2, font.labels = 2)

## put (absolute) correlations on the upper panels,
## with size proportional to the correlations.
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits = digits)[1]
    txt <- paste0(prefix, txt)
```

```
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex.cor * r)
}
pairs(USJudgeRatings, lower.panel = panel.smooth, upper.panel = panel.cor,
      gap=0, row1attop=FALSE)

pairs(iris[-5], log = "xy") # plot all variables on log scale
pairs(iris, log = 1:4, # log the first four
      main = "Lengths and Widths in [log]", line.main=1.5, oma=c(2,2,3,2))
```

`panel.smooth` *Simple Panel Plot*

Description

An example of a simple useful `panel` function to be used as argument in e.g., `coplot` or `pairs`.

Usage

```
panel.smooth(x, y, col = par("col"), bg = NA, pch = par("pch"),
             cex = 1, col.smooth = 2, span = 2/3, iter = 3,
             ...)
```

Arguments

`x`, `y`	numeric vectors of the same length
`col`, `bg`, `pch`, `cex`	numeric or character codes for the color(s), point type and size of `points`; see also `par`.
`col.smooth`	color to be used by `lines` for drawing the smooths.
`span`	smoothing parameter `f` for `lowess`, see there.
`iter`	number of robustness iterations for `lowess`.
`...`	further arguments to `lines`.

See Also

`coplot` and `pairs` where `panel.smooth` is typically used; `lowess` which does the smoothing.

Examples

```
pairs(swiss, panel = panel.smooth, pch = ".")  # emphasize the smooths
pairs(swiss, panel = panel.smooth, lwd = 2, cex = 1.5, col = 4)  # hmm...
```

par *Set or Query Graphical Parameters*

Description

par can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to par in tag = value form, or by passing them as a list of tagged values.

Usage

```
par(..., no.readonly = FALSE)

<highlevel plot> (...., <tag> = <value>)
```

Arguments

...	arguments in tag = value form, a single list of tagged values, or character vectors of parameter names. Supported parameters are described in the 'Graphical Parameters' section.
no.readonly	logical; if TRUE and there are no other arguments, only parameters are returned which can be set by a subsequent par() call *on the same device*.

Details

Each device has its own set of graphical parameters. If the current device is the null device, par will open a new device before querying/setting parameters. (What device is controlled by options("device").)

Parameters are queried by giving one or more character vectors of parameter names to par.

par() (no arguments) or par(no.readonly = TRUE) is used to get *all* the graphical parameters (as a named list). Their names are currently taken from the unexported variable graphics:::.Pars.

R.O. indicates ***read-only arguments***: These may only be used in queries and cannot be set. ("cin", "cra", "csi", "cxy", "din" and "page" are always read-only.)

Several parameters can only be set by a call to par():

- "ask",
- "fig", "fin",
- "lheight",
- "mai", "mar", "mex", "mfcol", "mfrow", "mfg",
- "new",
- "oma", "omd", "omi",
- "pin", "plt", "ps", "pty",
- "usr",
- "xlog", "ylog",

- `"ylbias"`

The remaining parameters can also be set as arguments (often via `...`) to high-level plot functions such as `plot.default`, `plot.window`, `points`, `lines`, `abline`, `axis`, `title`, `text`, `mtext`, `segments`, `symbols`, `arrows`, `polygon`, `rect`, `box`, `contour`, `filled.contour` and `image`. Such settings will be active during the execution of the function, only. However, see the comments on `bg`, `cex`, `col`, `lty`, `lwd` and `pch` which may be taken as *arguments* to certain plot functions rather than as graphical parameters.

The meaning of 'character size' is not well-defined: this is set up for the device taking `pointsize` into account but often not the actual font family in use. Internally the corresponding pars (`cra`, `cin`, `cxy` and `csi`) are used only to set the inter-line spacing used to convert `mar` and `oma` to physical margins. (The same inter-line spacing multiplied by `lheight` is used for multi-line strings in `text` and `strheight`.)

Note that graphical parameters are suggestions: plotting functions and devices need not make use of them (and this is particularly true of non-default methods for e.g. `plot`).

Value

When parameters are set, their previous values are returned in an invisible named list. Such a list can be passed as an argument to `par` to restore the parameter values. Use `par(no.readonly = TRUE)` for the full list of parameters that can be restored. However, restoring all of these is not wise: see the 'Note' section.

When just one parameter is queried, the value of that parameter is returned as (atomic) vector. When two or more parameters are queried, their values are returned in a list, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns a vector.

Graphical Parameters

`adj` The value of `adj` determines the way in which text strings are justified in `text`, `mtext` and `title`. A value of `0` produces left-justified text, `0.5` (the default) centered text and `1` right-justified text. (Any value in $[0, 1]$ is allowed, and on most devices values outside that interval will also work.)

Note that the `adj` *argument* of `text` also allows `adj = c(x, y)` for different adjustment in x- and y- directions. Note that whereas for `text` it refers to positioning of text about a point, for `mtext` and `title` it controls placement within the plot or device region.

`ann` If set to `FALSE`, high-level plotting functions calling `plot.default` do not annotate the plots they produce with axis titles and overall titles. The default is to do annotation.

`ask` logical. If `TRUE` (and the R session is interactive) the user is asked for input, before a new figure is drawn. As this applies to the device, it also affects output by packages **grid** and **lattice**. It can be set even on non-screen devices but may have no effect there.

This not really a graphics parameter, and its use is deprecated in favour of `devAskNewPage`.

`bg` The color to be used for the background of the device region. When called from `par()` it also sets `new = FALSE`. See section 'Color Specification' for suitable values. For many devices the initial value is set from the `bg` argument of the device, and for the rest it is normally `"white"`.

Note that some graphics functions such as `plot.default` and `points` have an *argument* of this name with a different meaning.

`bty` A character string which determined the type of `box` which is drawn about plots. If `bty` is one of `"o"` (the default), `"l"`, `"7"`, `"c"`, `"u"`, or `"]"` the resulting box resembles the corresponding upper case letter. A value of `"n"` suppresses the box.

`cex` A numerical value giving the amount by which plotting text and symbols should be magnified relative to the default. This starts as `1` when a device is opened, and is reset when the layout is changed, e.g. by setting `mfrow`.

Note that some graphics functions such as `plot.default` have an *argument* of this name which *multiplies* this graphical parameter, and some functions such as `points` and `text` accept a vector of values which are recycled.

`cex.axis` The magnification to be used for axis annotation relative to the current setting of `cex`.

`cex.lab` The magnification to be used for x and y labels relative to the current setting of `cex`.

`cex.main` The magnification to be used for main titles relative to the current setting of `cex`.

`cex.sub` The magnification to be used for sub-titles relative to the current setting of `cex`.

`cin` ***R.O.***; character size `(width, height)` in inches. These are the same measurements as `cra`, expressed in different units.

`col` A specification for the default plotting color. See section 'Color Specification'.

Some functions such as `lines` and `text` accept a vector of values which are recycled and may be interpreted slightly differently.

`col.axis` The color to be used for axis annotation. Defaults to `"black"`.

`col.lab` The color to be used for x and y labels. Defaults to `"black"`.

`col.main` The color to be used for plot main titles. Defaults to `"black"`.

`col.sub` The color to be used for plot sub-titles. Defaults to `"black"`.

`cra` ***R.O.***; size of default character `(width, height)` in 'rasters' (pixels). Some devices have no concept of pixels and so assume an arbitrary pixel size, usually 1/72 inch. These are the same measurements as `cin`, expressed in different units.

`crt` A numerical value specifying (in degrees) how single characters should be rotated. It is unwise to expect values other than multiples of 90 to work. Compare with `srt` which does string rotation.

`csi` ***R.O.***; height of (default-sized) characters in inches. The same as `par("cin")[2]`.

`cxy` ***R.O.***; size of default character `(width, height)` in user coordinate units. `par("cxy")` is `par("cin")/par("pin")` scaled to user coordinates. Note that `c(strwidth(ch), strheight(ch))` for a given string `ch` is usually much more precise.

`din` ***R.O.***; the device dimensions, `(width, height)`, in inches. See also `dev.size`, which is updated immediately when an on-screen device windows is re-sized.

`err` (*Unimplemented*; R is silent when points outside the plot region are *not* plotted.) The degree of error reporting desired.

`family` The name of a font family for drawing text. The maximum allowed length is 200 bytes. This name gets mapped by each graphics device to a device-specific font description. The default value is `""` which means that the default device fonts will be used (and what those are should be listed on the help page for the device). Standard values are `"serif"`, `"sans"` and

"mono", and the Hershey font families are also available. (Devices may define others, and some devices will ignore this setting completely. Names starting with "Hershey" are treated specially and should only be used for the built-in Hershey font families.) This can be specified inline for text.

`fg` The color to be used for the foreground of plots. This is the default color used for things like axes and boxes around plots. When called from `par()` this also sets parameter `col` to the same value. See section 'Color Specification'. A few devices have an argument to set the initial value, which is otherwise "black".

`fig` A numerical vector of the form `c(x1, x2, y1, y2)` which gives the (NDC) coordinates of the figure region in the display region of the device. If you set this, unlike S, you start a new plot, so to add to an existing plot use `new = TRUE` as well.

`fin` The figure region dimensions, `(width, height)`, in inches. If you set this, unlike S, you start a new plot.

`font` An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic. Also, font 5 is expected to be the symbol font, in Adobe symbol encoding. On some devices font families can be selected by `family` to choose different sets of 5 fonts.

`font.axis` The font to be used for axis annotation.

`font.lab` The font to be used for x and y labels.

`font.main` The font to be used for plot main titles.

`font.sub` The font to be used for plot sub-titles.

`lab` A numerical vector of the form `c(x, y, len)` which modifies the default way that axes are annotated. The values of `x` and `y` give the (approximate) number of tickmarks on the x and y axes and `len` specifies the label length. The default is `c(5, 5, 7)`. `len` *is unimplemented* in R.

`las` numeric in {0,1,2,3}; the style of axis labels.

- **0:** always parallel to the axis [*default*],
- **1:** always horizontal,
- **2:** always perpendicular to the axis,
- **3:** always vertical.

Also supported by mtext. Note that string/character rotation *via* argument `srt` to `par` does *not* affect the axis labels.

`lend` The line end style. This can be specified as an integer or string:

- `0` and "round" mean rounded line caps [*default*];
- `1` and "butt" mean butt line caps;
- `2` and "square" mean square line caps.

`lheight` The line height multiplier. The height of a line of text (used to vertically space multi-line text) is found by multiplying the character height both by the current character expansion and by the line height multiplier. Default value is 1. Used in text and strheight.

`ljoin` The line join style. This can be specified as an integer or string:

- `0` and "round" mean rounded line joins [*default*];
- `1` and "mitre" mean mitred line joins;

`2` and `"bevel"` mean bevelled line joins.

`lmitre` The line mitre limit. This controls when mitred line joins are automatically converted into bevelled line joins. The value must be larger than 1 and the default is 10. Not all devices will honour this setting.

`lty` The line type. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings `"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, or `"twodash"`, where `"blank"` uses 'invisible lines' (i.e., does not draw them).

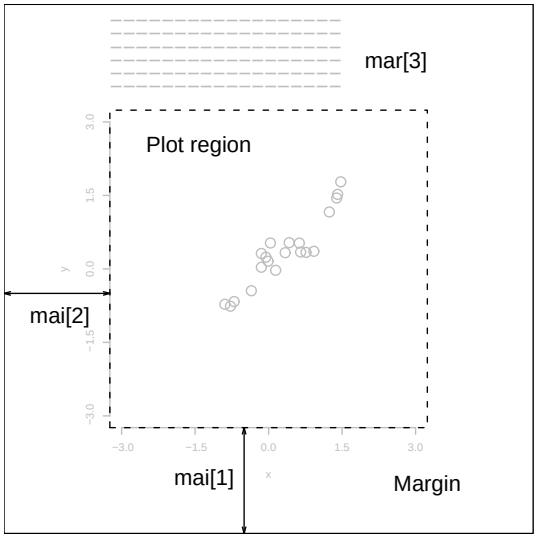
Alternatively, a string of up to 8 characters (from `c(1:9, "A":"F")`) may be given, giving the length of line segments which are alternatively drawn and skipped. See section 'Line Type Specification'.

Functions such as `lines` and `segments` accept a vector of values which are recycled.

`lwd` The line width, a *positive* number, defaulting to `1`. The interpretation is device-specific, and some devices do not implement line widths less than one. (See the help on the device for details of the interpretation.)

Functions such as `lines` and `segments` accept a vector of values which are recycled: in such uses lines corresponding to values `NA` or `NaN` are omitted. The interpretation of `0` is device-specific.

`mai` A numerical vector of the form `c(bottom, left, top, right)` which gives the margin size specified in inches.



`mar` A numerical vector of the form `c(bottom, left, top, right)` which gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) + 0.1`.

`mex` `mex` is a character size expansion factor which is used to describe coordinates in the margins of plots. Note that this does not change the font size, rather specifies the size of font (as a multiple of `csi`) used to convert between `mar` and `mai`, and between `oma` and `omi`.

This starts as `1` when the device is opened, and is reset when the layout is changed (alongside resetting `cex`).

`mfcol, mfrow` A vector of the form `c(nr, nc)`. Subsequent figures will be drawn in an `nr`-by-`nc` array on the device by *columns* (`mfcol`), or *rows* (`mfrow`), respectively.

In a layout with exactly two rows and columns the base value of `"cex"` is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

Setting a layout resets the base value of `cex` and that of `mex` to `1`.

If either of these is queried it will give the current layout, so querying cannot tell you the order in which the array will be filled.

Consider the alternatives, `layout` and `split.screen`.

`mfg` A numerical vector of the form `c(i, j)` where `i` and `j` indicate which figure in an array of figures is to be drawn next (if setting) or is being drawn (if enquiring). The array must already have been set by `mfcol` or `mfrow`.

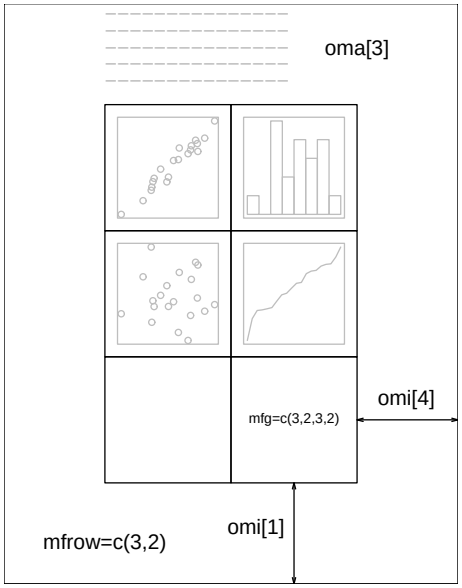
For compatibility with S, the form `c(i, j, nr, nc)` is also accepted, when `nr` and `nc` should be the current number of rows and number of columns. Mismatches will be ignored, with a warning.

`mgp` The margin line (in `mex` units) for the axis title, axis labels and axis line. Note that `mgp[1]` affects `title` whereas `mgp[2:3]` affect `axis`. The default is `c(3, 1, 0)`.

`mkh` The height in inches of symbols to be drawn when the value of `pch` is an integer. *Completely ignored in* R.

`new` logical, defaulting to `FALSE`. If set to `TRUE`, the next high-level plotting command (actually `plot.new`) should *not clean* the frame before drawing *as if it were on a* **new** *device*. It is an error (ignored with a warning) to try to use `new = TRUE` on a device that does not currently contain a high-level plot.

`oma` A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in lines of text.



`omd` A vector of the form `c(x1, x2, y1, y2)` giving the region *inside* outer margins in NDC (= normalized device coordinates), i.e., as a fraction (in $[0, 1]$) of the device region.

`omi` A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in inches.

`page` ***R.O.***; A boolean value indicating whether the next call to `plot.new` is going to start a new page. This value may be `FALSE` if there are multiple figures on the page.

`pch` Either an integer specifying a symbol or a single character to be used as the default in plotting points. See `points` for possible values and their interpretation. Note that only integers and single-character strings can be set as a graphics parameter (and not `NA` nor `NULL`).

Some functions such as `points` accept a vector of values which are recycled.

`pin` The current plot dimensions, `(width, height)`, in inches.

`plt` A vector of the form `c(x1, x2, y1, y2)` giving the coordinates of the plot region as fractions of the current figure region.

`ps` integer; the point size of text (but not symbols). Unlike the `pointsize` argument of most devices, this does not change the relationship between `mar` and `mai` (nor `oma` and `omi`).

What is meant by 'point size' is device-specific, but most devices mean a multiple of 1bp, that is 1/72 of an inch.

`pty` A character specifying the type of plot region to be used; `"s"` generates a square plotting region and `"m"` generates the maximal plotting region.

`smo` (*Unimplemented*) a value which indicates how smooth circles and circular arcs should be.

`srt` The string rotation in degrees. See the comment about `crt`. Only supported by `text`.

`tck` The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck >= 0.5` it is interpreted as a fraction of the relevant side, so if `tck = 1` grid lines are drawn. The default setting (`tck = NA`) is to use `tcl = -0.5`.

`tcl` The length of tick marks as a fraction of the height of a line of text. The default value is `-0.5`; setting `tcl = NA` sets `tck = -0.01` which is S' default.

`usr` A vector of the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region. When a logarithmic scale is in use (i.e., `par("xlog")` is true, see below), then the x-limits will be `10 ^ par("usr")[1:2]`. Similarly for the y-axis.

`xaxp` A vector of the form `c(x1, x2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks when `par("xlog")` is false. Otherwise, when *log* coordinates are active, the three values have a different meaning: For a small range, `n` is *negative*, and the ticks are as in the linear case, otherwise, `n` is in `1:3`, specifying a case number, and `x1` and `x2` are the lowest and highest power of 10 inside the user coordinates, `10 ^ par("usr")[1:2]`. (The `"usr"` coordinates are log10-transformed here!)

n = 1 will produce tick marks at 10^j for integer j,

n = 2 gives marks $k10^j$ with $k \in \{1, 5\}$,

n = 3 gives marks $k10^j$ with $k \in \{1, 2, 5\}$.

See `axTicks()` for a pure R implementation of this.

This parameter is reset when a user coordinate system is set up, for example by starting a new page or by calling `plot.window` or setting `par("usr")`: `n` is taken from `par("lab")`. It affects the default behaviour of subsequent calls to `axis` for sides 1 or 3.

It is only relevant to default numeric axis systems, and not for example to dates.

`xaxs` The style of axis interval calculation to be used for the x-axis. Possible values are `"r"`, `"i"`, `"e"`, `"s"`, `"d"`. The styles are generally controlled by the range of data or `xlim`, if given.
Style `"r"` (regular) first extends the data range by 4 percent at each end and then finds an axis with pretty labels that fits within the extended range.
Style `"i"` (internal) just finds an axis with pretty labels that fits within the original data range.
Style `"s"` (standard) finds an axis with pretty labels within which the original data range fits.
Style `"e"` (extended) is like style `"s"`, except that it is also ensures that there is room for plotting symbols within the bounding box.
Style `"d"` (direct) specifies that the current axis should be used on subsequent plots.
(*Only* `"r"` *and* `"i"` *styles have been implemented in* R.)

`xaxt` A character which specifies the x axis type. Specifying `"n"` suppresses plotting of the axis. The standard value is `"s"`: for compatibility with S values `"l"` and `"t"` are accepted but are equivalent to `"s"`: any value other than `"n"` implies plotting.

`xlog` A logical value (see `log` in `plot.default`). If `TRUE`, a logarithmic scale is in use (e.g., after `plot(*, log = "x")`). For a new device, it defaults to `FALSE`, i.e., linear scale.

`xpd` A logical value or `NA`. If `FALSE`, all plotting is clipped to the plot region, if `TRUE`, all plotting is clipped to the figure region, and if `NA`, all plotting is clipped to the device region. See also `clip`.

`yaxp` A vector of the form `c(y1, y2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks unless for log coordinates, see `xaxp` above.

`yaxs` The style of axis interval calculation to be used for the y-axis. See `xaxs` above.

`yaxt` A character which specifies the y axis type. Specifying `"n"` suppresses plotting.

`ylbias` A positive real value used in the positioning of text in the margins by `axis` and `mtext`. The default is in principle device-specific, but currently `0.2` for all of R's own devices. Set this to `0.2` for compatibility with R < 2.14.0 on `x11` and `windows()` devices.

`ylog` A logical value; see `xlog` above.

Color Specification

Colors can be specified in several different ways. The simplest way is with a character string giving the color name (e.g., `"red"`). A list of the possible colors can be obtained with the function `colors`. Alternatively, colors can be specified directly in terms of their RGB components with a string of the form `"#RRGGBB"` where each of the pairs `RR`, `GG`, `BB` consist of two hexadecimal digits giving a value in the range `00` to `FF`. Hexadecimal colors can be in the long hexadecimal form (e.g., `"#rrggbb"` or `"#rrggbbaa"`) or the short form (e.g, `"#rgb"` or `"#rgba"`). The short form is expanded to the long form by replicating digits (not by adding zeroes), e.g., `"#rgb"` becomes `"#rrggbb"`. Colors can also be specified by giving an index into a small table of colors, the `palette`: indices wrap round so with the default palette of size 8, `10` is the same as `2`. This provides compatibility with S. Index `0` corresponds to the background color. Note that the palette (apart from `0` which is per-device) is a per-session setting.

Negative integer colours are errors.

Additionally, `"transparent"` is *transparent*, useful for filled areas (such as the background!), and just invisible for things like lines or text. In most circumstances (integer) `NA` is equivalent to `"transparent"` (but not for `text` and `mtext`).

Semi-transparent colors are available for use on devices that support them.

The functions `rgb`, `hsv`, `hcl`, `gray` and `rainbow` provide additional ways of generating colors.

Line Type Specification

Line types can either be specified by giving an index into a small built-in table of line types (1 = solid, 2 = dashed, etc, see `lty` above) or directly as the lengths of on/off stretches of line. This is done with a string of an even number (up to eight) of characters, namely *non-zero* (hexadecimal) digits which give the lengths in consecutive positions in the string. For example, the string `"33"` specifies three units on followed by three off and `"3313"` specifies three units on followed by three off followed by one on and finally three off. The 'units' here are (on most devices) proportional to `lwd`, and with `lwd = 1` are in pixels or points or 1/96 inch.

The five standard dash-dot line types (`lty = 2:6`) correspond to `c("44", "13", "1343", "73", "2262")`.

Note that `NA` is not a valid value for `lty`.

Note

The effect of restoring all the (settable) graphics parameters as in the examples is hard to predict if the device has been resized. Several of them are attempting to set the same things in different ways, and those last in the alphabet will win. In particular, the settings of `mai`, `mar`, `pin`, `plt` and `pty` interact, as do the outer margin settings, the figure layout and figure region size.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot.default` for some high-level plotting parameters; `colors`; `clip`; `options` for other setup parameters; graphic devices `x11`, `pdf`, `postscript` and setting up device regions by `layout` and `split.screen`.

Examples

```
op <- par(mfrow = c(2, 2), # 2 x 2 pictures on one plot
          pty = "s")       # square plotting region,
                           # independent of device size

## At end of plotting, reset to previous settings:
par(op)

## Alternatively,
op <- par(no.readonly = TRUE) # the whole list of settable par's.
## do lots of plotting and par(.) calls, then reset:
par(op)
## Note this is not in general good practice

par("ylog") # FALSE
plot(1 : 12, log = "y")
par("ylog") # TRUE
```

```
plot(1:2, xaxs = "i") # 'inner axis' w/o extra space
par(c("usr", "xaxp"))

( nr.prof <-
c(prof.pilots = 16, lawyers = 11, farmers = 10, salesmen = 9, physicians = 9,
  mechanics = 6, policemen = 6, managers = 6, engineers = 5, teachers = 4,
  housewives = 3, students = 3, armed.forces = 1))
par(las = 3)
barplot(rbind(nr.prof)) # R 0.63.2: shows alignment problem
par(las = 0)  # reset to default

require(grDevices) # for gray
## 'fg' use:
plot(1:12, type = "b", main = "'fg' : axes, ticks and box in gray",
     fg = gray(0.7), bty = "7" , sub = R.version.string)

ex <- function() {
   old.par <- par(no.readonly = TRUE) # all par settings which
                                      # could be changed.
   on.exit(par(old.par))
   ## ...
   ## ... do lots of par() settings and plots
   ## ...
   invisible() #-- now,  par(old.par)  will be executed
}
ex()

## Line types
showLty <- function(ltys, xoff = 0, ...) {
   stopifnot((n <- length(ltys)) >= 1)
   op <- par(mar = rep(.5,4)); on.exit(par(op))
   plot(0:1, 0:1, type = "n", axes = FALSE, ann = FALSE)
   y <- (n:1)/(n+1)
   clty <- as.character(ltys)
   mytext <- function(x, y, txt)
      text(x, y, txt, adj = c(0, -.3), cex = 0.8, ...)
   abline(h = y, lty = ltys, ...); mytext(xoff, y, clty)
   y <- y - 1/(3*(n+1))
   abline(h = y, lty = ltys, lwd = 2, ...)
   mytext(1/8+xoff, y, paste(clty," lwd = 2"))
}
showLty(c("solid", "dashed", "dotted", "dotdash", "longdash", "twodash"))
par(new = TRUE)  # the same:
showLty(c("solid", "44", "13", "1343", "73", "2262"), xoff = .2, col = 2)
showLty(c("11", "22", "33", "44",   "12", "13", "14",   "21", "31"))
```

---

persp *Perspective Plots*

Description

This function draws perspective plots of a surface over the x–y plane. `persp` is a generic function.

Usage

```
persp(x, ...)

## Default S3 method:
persp(x = seq(0, 1, length.out = nrow(z)),
      y = seq(0, 1, length.out = ncol(z)),
      z, xlim = range(x), ylim = range(y),
      zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL,
      main = NULL, sub = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1,
      scale = TRUE, expand = 1,
      col = "white", border = NULL, ltheta = -135, lphi = 0,
      shade = NA, box = TRUE, axes = TRUE, nticks = 5,
      ticktype = "simple", ...)
```

Arguments

`x`, `y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`xlim`, `ylim`, `zlim`	x-, y- and z-limits. These should be chosen to cover the range of values of the surface: see 'Details'.
`xlab`, `ylab`, `zlab`	titles for the axes. N.B. These must be character strings; expressions are not accepted. Numbers will be coerced to character strings.
`main`, `sub`	main title and subtitle, as for `title`.
`theta`, `phi`	angles defining the viewing direction. `theta` gives the azimuthal direction and `phi` the colatitude.
`r`	the distance of the eyepoint from the centre of the plotting box.
`d`	a value which can be used to vary the strength of the perspective transformation. Values of `d` greater than 1 will lessen the perspective effect and values less and 1 will exaggerate it.
`scale`	before viewing the x, y and z coordinates of the points defining the surface are transformed to the interval [0,1]. If `scale` is `TRUE` the x, y and z coordinates are transformed separately. If `scale` is `FALSE` the coordinates are scaled so that aspect ratios are retained. This is useful for rendering things like DEM information.
`expand`	a expansion factor applied to the `z` coordinates. Often used with `0 < expand < 1` to shrink the plotting box in the `z` direction.

col	the color(s) of the surface facets. Transparent colours are ignored. This is recycled to the $(nx-1)(ny-1)$ facets.
border	the color of the line drawn around the surface facets. The default, NULL, corresponds to par("fg"). A value of NA will disable the drawing of borders: this is sometimes useful when the surface is shaded.
ltheta, lphi	if finite values are specified for ltheta and lphi, the surface is shaded as though it was being illuminated from the direction specified by azimuth ltheta and colatitude lphi.
shade	the shade at a surface facet is computed as ((1+d)/2)^shade, where d is the dot product of a unit vector normal to the facet and a unit vector in the direction of a light source. Values of shade close to one yield shading similar to a point light source model and values close to zero produce no shading. Values in the range 0.5 to 0.75 provide an approximation to daylight illumination.
box	should the bounding box for the surface be displayed. The default is TRUE.
axes	should ticks and labels be added to the box. The default is TRUE. If box is FALSE then no ticks or labels are drawn.
ticktype	character: "simple" draws just an arrow parallel to the axis to indicate direction of increase; "detailed" draws normal ticks as per 2D plots.
nticks	the (approximate) number of tick marks to draw on the axes. Has no effect if ticktype is "simple".
...	additional graphical parameters (see par).

Details

The plots are produced by first transforming the (x,y,z) coordinates to the interval [0,1] using the limits supplied or computed from the range of the data. The surface is then viewed by looking at the origin from a direction defined by theta and phi. If theta and phi are both zero the viewing direction is directly down the negative y axis. Changing theta will vary the azimuth and changing phi the colatitude.

There is a hook called "persp" (see setHook) called after the plot is completed, which is used in the testing code to annotate the plot page. The hook function(s) are called with no argument.

Notice that persp interprets the z matrix as a table of f(x[i], y[j]) values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, so that with the standard rotation angles, the top left corner of the matrix is displayed at the left hand side, closest to the user.

The sizes and fonts of the axis labels and the annotations for ticktype = "detailed" are controlled by graphics parameters "cex.lab"/"font.lab" and "cex.axis"/"font.axis" respectively.

The bounding box is drawn with edges of faces facing away from the viewer (and hence at the back of the box) with solid lines and other edges dashed and on top of the surface. This (and the plotting of the axes) assumes that the axis limits are chosen so that the surface is within the box, and the function will warn if this is not the case.

Value

persp() returns the *viewing transformation matrix*, say VT, a 4×4 matrix suitable for projecting 3D coordinates (x, y, z) into the 2D plane using homogeneous 4D coordinates (x, y, z, t). It can be

used to superimpose additional graphical elements on the 3D plot, by `lines()` or `points()`, using the function `trans3d()`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`contour` and `image`; `trans3d`.

Rotatable 3D plots can be produced by package **rgl**: other ways to produce static perspective plots are available in packages **lattice** and **scatterplot3d**.

Examples

```
require(grDevices) # for trans3d
## More examples in  demo(persp) !!
##                   -----------

# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length.out = 30)
y <- x
f <- function(x, y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
op <- par(bg = "white")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      ltheta = 120, shade = 0.75, ticktype = "detailed",
      xlab = "X", ylab = "Y", zlab = "Sinc( r )", cex.axis = 0.8
) -> res
round(res, 3)

# (2) Add to existing persp plot - using trans3d() :

xE <- c(-10,10); xy <- expand.grid(xE, xE)
points(trans3d(xy[,1], xy[,2], z = 6,          pmat = res), col = 2, pch = 16)
lines (trans3d(x,      y = 10, z = 6 + sin(x), pmat = res), col = 3)

phi <- seq(0, 2*pi, length.out = 201)
r1 <- 7.725 # radius of 2nd maximum
xr <- r1 * cos(phi)
yr <- r1 * sin(phi)
lines(trans3d(xr,yr, f(xr,yr), res), col = "pink", lwd = 2)
## (no hidden lines)

# (3) Visualizing a simple DEM model

z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
```

```
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
## Don't draw the grid lines :  border = NA
par(bg = "slategray")
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)

# (4) Surface colours corresponding to z-values

par(bg = "white")
x <- seq(-1.95, 1.95, length.out = 30)
y <- seq(-1.95, 1.95, length.out = 35)
z <- outer(x, y, function(a, b) a*b^2)
nrz <- nrow(z)
ncz <- ncol(z)
# Create a function interpolating colors in the range of specified colors
jet.colors <- colorRampPalette( c("blue", "green") )
# Generate the desired number of colors from this palette
nbcol <- 100
color <- jet.colors(nbcol)
# Compute the z-value at the facet centres
zfacet <- z[-1, -1] + z[-1, -ncz] + z[-nrz, -1] + z[-nrz, -ncz]
# Recode facet z-values into color indices
facetcol <- cut(zfacet, nbcol)
persp(x, y, z, col = color[facetcol], phi = 30, theta = -30)

par(op)
```

`pie` *Pie Charts*

Description

Draw a pie chart.

Usage

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Arguments

`x`	a vector of non-negative numerical quantities. The values in `x` are displayed as the areas of pie slices.
`labels`	one or more expressions or character strings giving names for the slices. Other objects are coerced by `as.graphicsAnnot`. For empty or `NA` (after coercion to character) labels, no label nor pointing line is drawn.

edges	the circular outline of the pie is approximated by a polygon with this many edges.
radius	the pie is drawn centered in a square box whose sides range from -1 to 1. If the character strings labeling the slices are long it may be necessary to use a smaller radius.
clockwise	logical indicating if slices are drawn clockwise or counter clockwise (i.e., mathematically positive direction), the latter is default.
init.angle	number specifying the *starting angle* (in degrees) for the slices. Defaults to 0 (i.e., '3 o'clock') unless clockwise is true where init.angle defaults to 90 (degrees), (i.e., '12 o'clock').
density	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. Non-positive values of density also inhibit the drawing of shading lines.
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise).
col	a vector of colors to be used in filling or shading the slices. If missing a set of 6 pastel colours is used, unless density is specified when par("fg") is used.
border, lty	(possibly vectors) arguments passed to polygon which draws each slice.
main	an overall title for the plot.
...	graphical parameters can be given as arguments to pie. They will affect the main title and labels only.

Note

Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.

Cleveland (1985), page 264: "Data that can be shown by pie charts always can be shown by a dot chart. This means that judgements of position along a common scale can be made instead of the less accurate angle judgements." This statement is based on the empirical investigations of Cleveland and McGill as well as investigations by perceptual psychologists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data*. Wadsworth: Monterey, CA, USA.

See Also

dotchart.

Examples

```
require(grDevices)
pie(rep(1, 24), col = rainbow(24), radius = 0.9)
```

```
pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
    "Apple", "Boston Cream", "Other", "Vanilla Cream")
pie(pie.sales) # default colours
pie(pie.sales, col = c("purple", "violetred1", "green3",
                       "cornsilk", "cyan", "white"))
pie(pie.sales, col = gray(seq(0.4, 1.0, length.out = 6)))
pie(pie.sales, density = 10, angle = 15 + 10 * 1:6)
pie(pie.sales, clockwise = TRUE, main = "pie(*, clockwise = TRUE)")
segments(0, 0, 0, 1, col = "red", lwd = 2)
text(0, 1, "init.angle = 90", col = "red")

n <- 200
pie(rep(1, n), labels = "", col = rainbow(n), border = NA,
    main = "pie(*, labels=\"\", col=rainbow(n), border=NA,..")

## Another case showing pie() is rather fun than science:
## (original by FinalBackwardsGlance on http://imgur.com/gallery/wWrpU4X)
pie(c(Sky = 78, "Sunny side of pyramid" = 17, "Shady side of pyramid" = 5),
    init.angle = 315, col = c("deepskyblue", "yellow", "yellow3"), border = FALSE)
```

`plot.data.frame` *Plot Method for Data Frames*

Description

`plot.data.frame`, a method for the `plot` generic. It is designed for a quick look at numeric data frames.

Usage

```
## S3 method for class 'data.frame'
plot(x, ...)
```

Arguments

`x`	object of class `data.frame`.
`...`	further arguments to `stripchart`, `plot.default` or `pairs`.

Details

This is intended for data frames with *numeric* columns. For more than two columns it first calls `data.matrix` to convert the data frame to a numeric matrix and then calls `pairs` to produce a scatterplot matrix. This can fail and may well be inappropriate: for example numerical conversion of dates will lose their special meaning and a warning will be given.

For a two-column data frame it plots the second column against the first by the most appropriate method for the first column.

For a single numeric column it uses `stripchart`, and for other single-column data frames tries to find a plot method for the single column.

See Also

data.frame

Examples

```
plot(OrchardSprays[1], method = "jitter")
plot(OrchardSprays[c(4,1)])
plot(OrchardSprays)

plot(iris)
plot(iris[5:4])
plot(women)
```

---

plot.default *The Default Scatterplot Function*

Description

Draw a scatter plot with decorations such as axes and titles in the active graphics window.

Usage

```
## Default S3 method:
plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA,
     xgap.axis = NA, ygap.axis = NA,
     ...)
```

Arguments

x, y	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function xy.coords for details. If supplied separately, they must be of the same length.
type	1-character string giving the type of plot desired. The following values are possible, for details, see plot: "p" for points, "l" for lines, "b" for both points and lines, "c" for empty points joined by lines, "o" for overplotted points and lines, "s" and "S" for stair steps and "h" for histogram-like vertical lines. Finally, "n" does not produce any points or lines.
xlim	the x limits (x1, x2) of the plot. Note that x1 > x2 is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the finite values to be plotted should be used.
ylim	the y limits of the plot.

log	a character string which contains "x" if the x axis is to be logarithmic, "y" if the y axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.
main	a main title for the plot, see also title.
sub	a subtitle for the plot.
xlab	a label for the x axis, defaults to a description of x.
ylab	a label for the y axis, defaults to a description of y.
ann	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
axes	a logical value indicating whether both axes should be drawn on the plot. Use graphical parameter "xaxt" or "yaxt" to suppress just one of the axes.
frame.plot	a logical indicating whether a box should be drawn around the plot.
panel.first	an 'expression' to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths. Note that this works by lazy evaluation: passing this argument from other plot methods may well not work since it may be evaluated too early.
panel.last	an expression to be evaluated after plotting has taken place but before the axes, title and box are added. See the comments about panel.first.
asp	the y/x aspect ratio, see plot.window.
xgap.axis, ygap.axis	the x/y axis gap factors, passed as gap.axis to the two axis() calls (when axes is true, as per default).
...	other graphical parameters (see par and section 'Details' below).

Details

Commonly used graphical parameters are:

col The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.

bg a vector of background colors for open plot symbols, see points. Note: this is **not** the same setting as par("bg").

pch a vector of plotting characters or symbols: see points.

cex a numerical vector giving the amount by which plotting characters and symbols should be scaled relative to the default. This works as a multiple of par("cex"). NULL and NA are equivalent to 1.0. Note that this does not affect annotation: see below.

lty a vector of line types, see par.

cex.main**,** col.lab**,** font.sub**, etc** settings for main- and sub-title and axis annotation, see title and par.

lwd a vector of line widths, see par.

Note

The presence of `panel.first` and `panel.last` is a historical anomaly: default plots do not have 'panels', unlike e.g. `pairs` plots. For more control, use lower-level plotting functions: `plot.default` calls in turn some of `plot.new`, `plot.window`, `plot.xy`, `axis`, `box` and `title`, and plots can be built up by calling these individually, or by calling `plot(type = "n")` and adding further elements.

The `plot` generic was moved from the **graphics** package to the **base** package in R 4.0.0. It is currently re-exported from the **graphics** namespace to allow packages importing it from there to continue working, but this may change in future versions of R.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `plot.window`, `xy.coords`. For thousands of points, consider using `smoothScatter` instead.

Examples

```
Speed <- cars$speed
Distance <- cars$dist
plot(Speed, Distance, panel.first = grid(8, 8),
     pch = 0, cex = 1.2, col = "blue")
plot(Speed, Distance,
     panel.first = lines(stats::lowess(Speed, Distance), lty = "dashed"),
     pch = 0, cex = 1.2, col = "blue")

## Show the different plot types
x <- 0:12
y <- sin(pi/5 * x)
op <- par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
for (tp in c("p","l","b",  "c","o","h",  "s","S","n")) {
   plot(y ~ x, type = tp, main = paste0("plot(*, type = \"", tp, "\")"))
   if(tp == "S") {
      lines(x, y, type = "s", col = "red", lty = 2)
      mtext("lines(*, type = \"s\", ...)", col = "red", cex = 0.8)
   }
}
par(op)

##--- Log-Log Plot  with  custom axes
lx <- seq(1, 5, length.out = 41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow = c(2,1), mar = par("mar")-c(1,0,2,0), mgp = c(2, .7, 0))
plot(10^lx, y, log = "xy", type = "l", col = "purple",
```

```
     main = "Log-Log plot", ylab = yl, xlab = "x")
plot(10^lx, y, log = "xy", type = "o", pch = ".", col = "forestgreen",
     main = "Log-Log plot with custom axes", ylab = yl, xlab = "x",
     axes = FALSE, frame.plot = TRUE)
my.at <- 10^(1:5)
axis(1, at = my.at, labels = formatC(my.at, format = "fg"))
e.y <- -5:-1 ; at.y <- 10^e.y
axis(2, at = at.y, col.axis = "red", las = 1,
     labels = as.expression(lapply(e.y, function(E) bquote(10^.(E)))))
par(op)
```

`plot.design` *Plot Univariate Effects of a Design or Model*

Description

Plot univariate effects of one or more `factor`s, typically for a designed experiment as analyzed by `aov()`.

Usage

```
plot.design(x, y = NULL, fun = mean, data = NULL, ...,
            ylim = NULL, xlab = "Factors", ylab = NULL,
            main = NULL, ask = NULL, xaxt = par("xaxt"),
            axes = TRUE, xtick = FALSE)
```

Arguments

`x`	either a data frame containing the design factors and optionally the response, or a `formula` or `terms` object.
`y`	the response, if not given in x.
`fun`	a function (or name of one) to be applied to each subset. It must return one number for a numeric (vector) input.
`data`	data frame containing the variables referenced by `x` when that is formula-like.
`...`	graphical parameters such as `col`, see `par`.
`ylim`	range of y values, as in `plot.default`.
`xlab`	x axis label, see `title`.
`ylab`	y axis label with a 'smart' default.
`main`	main title, see `title`.
`ask`	logical indicating if the user should be asked before a new page is started – in the case of multiple y values.
`xaxt`	character giving the type of x axis.
`axes`	logical indicating if axes should be drawn.
`xtick`	logical indicating if ticks (one per factor) should be drawn on the x axis.

Details

The supplied function will be called once for each level of each factor in the design and the plot will show these summary values. The levels of a particular factor are shown along a vertical line, and the overall value of `fun()` for the response is drawn as a horizontal line.

Note

A big effort was taken to make this closely compatible to the S version. However, `col` (and `fg`) specifications have different effects.

In S this was a method of the `plot` generic function for `design` objects.

Author(s)

Roberto Frisullo and Martin Maechler

References

Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S*. Chapman & Hall, London, **the** *white book*, pp. 546–7 (and 163–4).

Freeny, A. E. and Landwehr, J. M. (1990) Displays for data from large designed experiments; Computer Science and Statistics: Proc.\ 22nd Symp\. Interface, 117–126, Springer Verlag.

See Also

`interaction.plot` for a 'standard graphic' of designed experiments.

Examples

```
require(stats)
plot.design(warpbreaks)  # automatic for data frame with one numeric var.

Form <- breaks ~ wool + tension
summary(fm1 <- aov(Form, data = warpbreaks))
plot.design(       Form, data = warpbreaks, col = 2)  # same as above

## More than one y :
utils::str(esoph)
plot.design(esoph) ## two plots; if interactive you are "ask"ed

## or rather, compare mean and median:
op <- par(mfcol = 1:2)
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8))
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8),
            fun = median)
par(op)
```

plot.factor *Plotting Factor Variables*

Description

This functions implements a scatterplot method for factor arguments of the *generic* plot function.

If y is missing barplot is produced. For numeric y a boxplot is used, and for a factor y a spineplot is shown. For any other type of y the next plot method is called, normally plot.default.

Usage

```
## S3 method for class 'factor'
plot(x, y, legend.text = NULL, ...)
```

Arguments

x, y	numeric or factor. y may be missing.
legend.text	character vector for annotation of y axis in the case of a factor y: defaults to levels(y). This sets the yaxlabels argument of spineplot.
...	Further arguments to barplot, boxplot, spineplot or plot as appropriate. All of these accept graphical parameters (see par) and annotation arguments passed to title and axes = FALSE. None accept type.

See Also

plot.default, plot.formula, barplot, boxplot, spineplot.

Examples

```
require(grDevices)
plot(weight ~ group, data = PlantGrowth)           # numeric vector ~ factor
plot(cut(weight, 2) ~ group, data = PlantGrowth)   # factor ~ factor
## passing "..." to spineplot() eventually:
plot(cut(weight, 3) ~ group, data = PlantGrowth,
     col = hcl(c(0, 120, 240), 50, 70))

plot(PlantGrowth$group, axes = FALSE, main = "no axes")  # extremely silly
```

plot.formula *Formula Notation for Scatterplots*

Description

Specify a scatterplot or add points, lines, or text via a formula.

Usage

```
## S3 method for class 'formula'
plot(formula, data = parent.frame(), ..., subset,
             ylab = varnames[response], ask = dev.interactive())

## S3 method for class 'formula'
points(formula, data = parent.frame(), ..., subset)

## S3 method for class 'formula'
lines(formula, data = parent.frame(), ..., subset)

## S3 method for class 'formula'
text(formula, data = parent.frame(), ..., subset)
```

Arguments

formula	a formula, such as y ~ x.
data	a data.frame (or list) from which the variables in formula should be taken. A matrix is converted to a data frame.
...	Arguments to be passed to or from other methods. horizontal = TRUE is also accepted.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
ylab	the y label of the plot(s).
ask	logical, see par.

Details

For the lines, points and text methods the formula should be of the form y ~ x or y ~ 1 with a left-hand side and a single term on the right-hand side. The plot method accepts other forms discussed later in this section.

Both the terms in the formula and the ... arguments are evaluated in data enclosed in parent.frame() if data is a list or a data frame. The terms of the formula and those arguments in ... that are of the same length as data are subjected to the subsetting specified in subset. A plot against the running index can be specified as plot(y ~ 1).

If the formula in the plot method contains more than one term on the right-hand side, a series of plots is produced of the response against each non-response term.

For the plot method the formula can be of the form ~ z + y + z: the variables specified on the right-hand side are collected into a data frame, subsetted if specified, and displayed by plot.data.frame.

Missing values are not considered in these methods, and in particular cases with missing values are not removed.

If y is an object (i.e., has a class attribute) then plot.formula looks for a plot method for that class first. Otherwise, the class of x will determine the type of the plot. For factors this will be a parallel boxplot, and argument horizontal = TRUE can be specified (see boxplot).

Note that some arguments will need to be protected from premature evaluation by enclosing them in quote: currently this is done automatically for main, sub and xlab. For example, it is needed for the panel.first and panel.last arguments passed to plot.default.

Value

These functions are invoked for their side effect of drawing on the active graphics device.

See Also

plot.default, points, lines, plot.factor.

Examples

```
op <- par(mfrow = c(2,1))
plot(Ozone ~ Wind, data = airquality, pch = as.character(Month))
plot(Ozone ~ Wind, data = airquality, pch = as.character(Month),
     subset = Month != 7)
par(op)

## text.formula() can be very natural:
wb <- within(warpbreaks, {
    time <- seq_along(breaks); W.T <- wool:tension })
plot(breaks ~ time, data = wb, type = "b")
text(breaks ~ time, data = wb, labels = W.T, col = 1+as.integer(wool))
```

---

plot.histogram *Plot Histograms*

Description

Plotting methods for objects of class "histogram", typically produced by hist.

Usage

```
## S3 method for class 'histogram'
plot(x, freq = equidist, density = NULL, angle = 45,
               col = "lightgray", border = NULL, lty = NULL,
               main = paste("Histogram of", paste(x$xname, collapse = "\n")),
               sub = NULL, xlab = x$xname, ylab,
               xlim = range(x$breaks), ylim = NULL,
```

```
                axes = TRUE, labels = FALSE, add = FALSE,
                ann = TRUE, ...)

## S3 method for class 'histogram'
lines(x, ...)
```

Arguments

x	a histogram object, or a list with components density, mid, etc, see hist for information about the components of x.
freq	logical; if TRUE, the histogram graphic is to present a representation of frequencies, i.e, x$counts; if FALSE, *relative* frequencies (probabilities), i.e., x$density, are plotted. The default is true for equidistant breaks and false otherwise.
col	a colour to be used to fill the bars. The default has been changed from NULL (unfilled bars) only as from R 4.2.0.
border	the color of the border around the bars.
angle, density	select shading of bars by lines: see rect.
lty	the line type used for the bars, see also lines.
main, sub, xlab, ylab	these arguments to title have useful defaults here.
xlim, ylim	the range of x and y values with sensible defaults.
axes	logical, indicating if axes should be drawn.
labels	logical or character. Additionally draw labels on top of bars, if not FALSE; if TRUE, draw the counts or rounded densities; if labels is a character, draw itself.
add	logical. If TRUE, only the bars are added to the current plot. This is what lines.histogram(*) does.
ann	logical. Should annotations (titles and axis titles) be plotted?
...	further graphical parameters to title and axis.

Details

lines.histogram(*) is the same as plot.histogram(*, add = TRUE).

See Also

hist, stem, density.

Examples

```
(wwt <- hist(women$weight, nclass = 7, plot = FALSE))
plot(wwt, labels = TRUE) # default main & xlab using wwt$xname
plot(wwt, border = "dark blue", col = "light blue",
     main = "Histogram of 15 women's weights", xlab = "weight [pounds]")
```

```
## Fake "lines" example, using non-default labels:
w2 <- wwt; w2$counts <- w2$counts - 1
lines(w2, col = "Midnight Blue", labels = ifelse(w2$counts, "> 1", "1"))
```

`plot.raster` *Plotting Raster Images*

Description

This functions implements a `plot` method for raster images.

Usage

```
## S3 method for class 'raster'
plot(x, y,
     xlim = c(0, ncol(x)), ylim = c(0, nrow(x)),
     xaxs = "i", yaxs = "i",
     asp = 1, add = FALSE, ...)
```

Arguments

`x`, `y`	raster. `y` will be ignored.
`xlim`, `ylim`	Limits on the plot region (default from dimensions of the raster).
`xaxs`, `yaxs`	Axis interval calculation style (default means that raster fills plot region).
`asp`	Aspect ratio (default retains aspect ratio of the raster).
`add`	Logical indicating whether to simply add raster to an existing plot.
`...`	Further arguments to the `rasterImage` function.

See Also

`plot.default`, `rasterImage`.

Examples

```
require(grDevices)
r <- as.raster(c(0.5, 1, 0.5))
plot(r)
# additional arguments to rasterImage()
plot(r, interpolate=FALSE)
# distort
plot(r, asp=NA)
# fill page
op <- par(mar=rep(0, 4))
plot(r, asp=NA)
par(op)
# normal annotations work
plot(r, asp=NA)
```

```
box()
title(main="This is my raster")
# add to existing plot
plot(1)
plot(r, add=TRUE)
```

`plot.table` *Plot Methods for* `table` *Objects*

Description

This is a method of the generic `plot` function for (contingency) `table` objects. Whereas for two- and more dimensional tables, a `mosaicplot` is drawn, one-dimensional ones are plotted as bars.

Usage

```
## S3 method for class 'table'
plot(x, type = "h", ylim = c(0, max(x)), lwd = 2,
     xlab = NULL, ylab = NULL, frame.plot = is.num, ...)
## S3 method for class 'table'
points(x, y = NULL, type = "h", lwd = 2, ...)
## S3 method for class 'table'
lines(x, y = NULL, type = "h", lwd = 2, ...)
```

Arguments

`x`	a `table` (like) object.
`y`	Must be `NULL`: there to protect against incorrect calls.
`type`	plotting type.
`ylim`	range of y-axis.
`lwd`	line width for bars when `type = "h"` is used in the 1D case.
`xlab, ylab`	x- and y-axis labels.
`frame.plot`	logical indicating if a frame (`box`) should be drawn in the 1D case. Defaults to true when `x` has `dimnames` coerce-able to numbers.
`...`	further graphical arguments, see `plot.default`. `axes = FALSE` is accepted.

See Also

`plot.factor`, the `plot` method for factors.

Examples

```
## 1-d tables
(Poiss.tab <- table(N = stats::rpois(200, lambda = 5)))
plot(Poiss.tab, main = "plot(table(rpois(200, lambda = 5)))")

plot(table(state.division))

## 4-D :
plot(Titanic, main ="plot(Titanic, main= *)")
```

`plot.window` *Set up World Coordinates for Graphics Window*

Description

This function sets up the world coordinate system for a graphics window. It is called by higher level functions such as `plot.default` (*after* `plot.new`).

Usage

```
plot.window(xlim, ylim, log = "", asp = NA, ...)
```

Arguments

`xlim`, `ylim`	numeric vectors of length 2, giving the x and y coordinates ranges.
`log`	character; indicating which axes should be in log scale.
`asp`	numeric, giving the **asp**ect ratio y/x, see 'Details'.
`...`	further graphical parameters as in `par`. The relevant ones are `xaxs`, `yaxs` and `lab`.

Details

asp: If `asp` is a finite positive value then the window is set up so that one data unit in the y direction is equal in length to `asp` $\times$ one data unit in the x direction.

Note that in this case, `par("usr")` is no longer determined by, e.g., `par("xaxs")`, but rather by `asp` and the device's aspect ratio. (See what happens if you interactively resize the plot device after running the example below!)

The special case `asp == 1` produces plots where distances between points are represented accurately on screen. Values with `asp > 1` can be used to produce more accurate maps when using latitude and longitude.

Note that the coordinate ranges will be extended by 4% if the appropriate graphical parameter `xaxs` or `yaxs` has value `"r"` (which is the default).

To reverse an axis, use `xlim` or `ylim` of the form `c(hi, lo)`.

The function attempts to produce a plausible set of scales if one or both of `xlim` and `ylim` is of length one or the two values given are identical, but it is better to avoid that case.

Usually, one should rather use the higher-level functions such as plot, hist, image, ..., instead and refer to their help pages for explanation of the arguments.

A side-effect of the call is to set up the usr, xaxp and yaxp graphical parameters. (It is for the latter two that lab is used.)

See Also

xy.coords, plot.xy, plot.default.

par for the graphical parameters mentioned.

Examples

```
##--- An example for the use of 'asp' :
require(stats)  # normally loaded
loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "")
abline(h = pretty(rx, 10), v = pretty(ry, 10), col = "lightgray")
text(x, y, labels(eurodist), cex = 0.8)
```

---

plot.xy *Basic Internal Plot Function*

Description

This is *the* internal function that does the basic plotting of points and lines. Usually, one should rather use the higher level functions instead and refer to their help pages for explanation of the arguments.

Usage

```
plot.xy(xy, type, pch = par("pch"), lty = par("lty"),
        col = par("col"), bg = NA,
        cex = 1, lwd = par("lwd"), ...)
```

Arguments

xy	A four-element list as results from xy.coords.
type	1 character code: see plot.default. NULL is accepted as a synonym for "p".
pch	character or integer code for kind of points, see points.default.
lty	line type code, see lines.
col	color code or name, see colors, palette. Here NULL means colour 0.
bg	background (fill) color for the open plot symbols 21:25: see points.default.
cex	character expansion.
lwd	line width, also used for (non-filled) plot symbols, see lines and points.
...	further graphical parameters such as xpd, lend, ljoin and lmitre.

Details

The arguments pch, col, bg, cex, lwd may be vectors and may be recycled, depending on type: see points and lines for specifics. In particular note that lwd is treated as a vector for points and as a single (first) value for lines.

cex is a numeric factor in addition to par("cex") which affects symbols and characters as drawn by type "p", "o", "b" and "c".

See Also

plot, plot.default, points, lines.

Examples

```
points.default # to see how it calls "plot.xy(xy.coords(x, y), ...)"
```

---

points *Add Points to a Plot*

Description

points is a generic function to draw a sequence of points at the specified coordinates. The specified character(s) are plotted, centered at the coordinates.

Usage

```
points(x, ...)

## Default S3 method:
points(x, y = NULL, type = "p", ...)
```

Arguments

x, y	coordinate vectors of points to plot.
type	character indicating the type of plotting; actually any of the types as in plot.default.
...	Further graphical parameters may also be supplied as arguments. See 'Details'.

Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, a time series, ... See xy.coords. If supplied separately, they must be of the same length.

Graphical parameters commonly used are

`pch` plotting 'character', i.e., symbol to use. This can either be a single character or an integer code for one of a set of graphics symbols. The full set of S symbols is available with `pch = 0:18`, see the examples below. (NB: R uses circles instead of the octagons used in S.)

Value `pch = "."` (equivalently `pch = 46`) is handled specially. It is a rectangle of side 0.01 inch (scaled by `cex`). In addition, if `cex = 1` (the default), each side is at least one pixel (1/72 inch on the `pdf`, `postscript` and `xfig` devices).

For other text symbols, `cex = 1` corresponds to the default font size of the device, often specified by an argument `pointsize`. For `pch` in `0:25` the default size is about 75% of the character height (see `par("cin")`).

`col` color code or name, see `par`.

`bg` background (fill) color for the open plot symbols given by `pch = 21:25`.

`cex` character (or symbol) expansion: a numerical vector. This works as a multiple of `par("cex")`.

`lwd` line width for drawing symbols see `par`.

Others less commonly used are `lty` and `lwd` for types such as `"b"` and `"l"`.

The graphical parameters `pch`, `col`, `bg`, `cex` and `lwd` can be vectors (which will be recycled as needed) giving a value for each point plotted. If lines are to be plotted (e.g., for `type = "b"`) the first element of `lwd` is used.

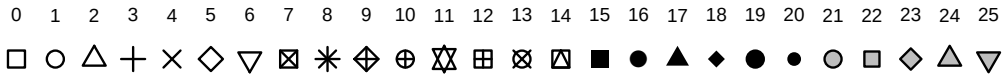
Points whose `x`, `y`, `pch`, `col` or `cex` value is `NA` are omitted from the plot.

`pch` values

Values of `pch` are stored internally as integers. The interpretation is

- `NA_integer_`: no symbol.
- `0:18`: S-compatible vector symbols.
- `19:25`: further R vector symbols.
- `26:31`: unused (and ignored).
- `32:127`: ASCII characters.
- `128:255` native characters *only in a single-byte locale and for the symbol font*. (`128:159` are only used on Windows.)
- `-32 ...` Unicode code point (where supported).

Note that unlike S (which uses octagons), symbols `1`, `10`, `13` and `16` use circles. The filled shapes `15:18` do not include a border.



The following R plotting symbols are can be obtained with `pch = 19:25`: those with `21:25` can be colored and filled with different colors: `col` gives the border color and `bg` the background color

(which is '`"grey"`' in the figure)

- `pch = 19`: solid circle,

- pch = 20: bullet (smaller solid circle, 2/3 the size of 19),
- pch = 21: filled circle,
- pch = 22: filled square,
- pch = 23: filled diamond,
- pch = 24: filled triangle point-up,
- pch = 25: filled triangle point down.

Note that all of these both fill the shape and draw a border. Some care in interpretation is needed when semi-transparent colours are used for both fill and border (and the result might be device-specific and even viewer-specific for pdf).

The difference between pch = 16 and pch = 19 is that the latter uses a border and so is perceptibly larger when lwd is large relative to cex.

Values pch = 26:31 are currently unused and pch = 32:127 give the ASCII characters. In a single-byte locale pch = 128:255 give the corresponding character (if any) in the locale's character set. Where supported by the OS, negative values specify a Unicode code point, so e.g. -0x2642L is a 'male sign' and -0x20ACL is the Euro.

A character string consisting of a single character is converted to an integer: 32:127 for ASCII characters, and usually to the Unicode code point otherwise. (In non-Latin-1 single-byte locales, 128:255 will be used for 8-bit characters.)

If pch supplied is a logical, integer or character NA or an empty character string the point is omitted from the plot.

If pch is NULL or otherwise of length 0, par("pch") is used.

If the symbol font (par(font = 5)) is used, numerical values should be used for pch: the range is c(32:126, 160:254) in all locales (but 240 is not defined (used for 'apple' on macOS) and 160, Euro, may not be present).

Note

A single-byte encoding may include the characters in pch = 128:255, and if it does, a font may not include all (or even any) of them.

Not all negative numbers are valid as Unicode code points, and no check is done. A display device is likely to use a rectangle for (or omit) Unicode code points which are invalid or for which it does not have a glyph in the font used.

What happens for very small or zero values of cex is device-dependent: symbols or characters may become invisible or they may be plotted at a fixed minimum size. Circles of zero radius will not be plotted.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

points.formula for the formula method; plot, lines, and the underlying workhorse function plot.xy.

Examples

```
require(stats) # for rnorm
plot(-4:4, -4:4, type = "n")  # setting up coord. system
points(rnorm(200), rnorm(200), col = "red")
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5)

op <- par(bg = "light blue")
x <- seq(0, 2*pi, length.out = 51)
## something "between type='b' and type='o'":
plot(x, sin(x), type = "o", pch = 21, bg = par("bg"), col = "blue", cex = .6,
 main = 'plot(..., type="o", pch=21, bg=par("bg"))')
par(op)

## Illustration of pch = 0:25 (as in the figure shown above in PDF/HTML help)
## Not run: png("pch.png", height = 0.7, width = 7, res = 100, units = "in")
par(mar = rep(0,4))
plot(c(-1, 26), 0:1, type = "n", axes = FALSE)
text(0:25, 0.6, 0:25, cex = 0.5)
points(0:25, rep(0.3, 26), pch = 0:25, bg = "grey")


##-------- Showing all the extra & some char graphics symbols ---------
pchShow <-
  function(extras = c("*",".", "o","O","0","+","-","|","%","#"),
           cex = 3, ## good for both .Device=="postscript" and "x11"
           col = "red3", bg = "gold", coltext = "brown", cextext = 1.2,
           main = paste("plot symbols :  points (...  pch = *, cex =",
                        cex,")"))
  {
    nex <- length(extras)
    np  <- 26 + nex
    ipch <- 0:(np-1)
    k <- floor(sqrt(np))
    dd <- c(-1,1)/2
    rx <- dd + range(ix <- ipch %/% k)
    ry <- dd + range(iy <- 3 + (k-1)- ipch %% k)
    pch <- as.list(ipch) # list with integers & strings
    if(nex > 0) pch[26+ 1:nex] <- as.list(extras)
    plot(rx, ry, type = "n", axes  =  FALSE, xlab = "", ylab = "", main = main)
    abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
    for(i in 1:np) {
      pc <- pch[[i]]
      ## 'col' symbols with a 'bg'-colored interior (where available) :
      points(ix[i], iy[i], pch = pc, col = col, bg = bg, cex = cex)
      if(cextext > 0)
          text(ix[i] - 0.3, iy[i], pc, col = coltext, cex = cextext)
    }
  }

pchShow()
pchShow(c("o","O","0"), cex = 2.5)
pchShow(NULL, cex = 4, cextext = 0, main = NULL)
```

```
## ------------ test code for various pch specifications -------------
# Try this in various font families (including Hershey)
# and locales.  Use sign = -1 asserts we want Latin-1.
# Standard cases in a MBCS locale will not plot the top half.
TestChars <- function(sign = 1, font = 1, ...)
{
   MB <- l10n_info()$MBCS
   r <- if(font == 5) { sign <- 1; c(32:126, 160:254)
       } else if(MB) 32:126 else 32:255
   if (sign == -1) r <- c(32:126, 160:255)
   par(pty = "s")
   plot(c(-1,16), c(-1,16), type = "n", xlab = "", ylab = "",
        xaxs = "i", yaxs = "i",
        main = sprintf("sign = %d, font = %d", sign, font))
   grid(17, 17, lty = 1) ; mtext(paste("MBCS:", MB))
   for(i in r) try(points(i%%16, i%/%16, pch = sign*i, font = font,...))
}
TestChars()
try(TestChars(sign = -1))
TestChars(font = 5)  # Euro might be at 160 (0+10*16).
                     # macOS has apple at 240 (0+15*16).
try(TestChars(-1, font = 2))  # bold
```

---

polygon *Polygon Drawing*

Description

polygon draws the polygons whose vertices are given in x and y.

Usage

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"),
        ..., fillOddEven = FALSE)
```

Arguments

x, y	vectors containing the coordinates of the vertices of the polygon.
density	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. A zero value of density means no shading nor filling whereas negative values and NA suppress shading (and so allow color filling).
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise).

col	the color for filling the polygon. The default, NA, is to leave polygons unfilled, unless density is specified. (For back-compatibility, NULL is equivalent to NA.) If density is specified with a positive value this gives the color of the shading lines.
border	the color to draw the border. The default, NULL, means to use par("fg"). Use border = NA to omit borders. For compatibility with S, border can also be logical, in which case FALSE is equivalent to NA (borders omitted) and TRUE is equivalent to NULL (use the foreground colour),
lty	the line type to be used, as in par.
...	graphical parameters such as xpd, lend, ljoin and lmitre can be given as arguments.
fillOddEven	logical controlling the polygon shading mode: see below for details. Default FALSE.

Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, ... See xy.coords.

It is assumed that the polygon is to be closed by joining the last point to the first point.

The coordinates can contain missing values. The behaviour is similar to that of lines, except that instead of breaking a line into several lines, NA values break the polygon into several complete polygons (including closing the last point to the first point). See the examples below.

When multiple polygons are produced, the values of density, angle, col, border, and lty are recycled in the usual manner.

Shading of polygons is only implemented for linear plots: if either axis is on log scale then shading is omitted, with a warning.

Bugs

Self-intersecting polygons may be filled using either the "odd-even" or "non-zero" rule. These fill a region if the polygon border encircles it an odd or non-zero number of times, respectively. Shading lines are handled internally by R according to the fillOddEven argument, but device-based solid fills depend on the graphics device. The windows, pdf and postscript devices have their own fillOddEven argument to control this.

Author(s)

The code implementing polygon shading was donated by Kevin Buhr <buhr@stat.wisc.edu>.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

segments for even more flexibility, lines, rect, box, abline.

par for how to specify colors.

Examples

```
x <- c(1:9, 8:1)
y <- c(1, 2*(5:3), 2, -1, 17, 9, 8, 2:9)
op <- par(mfcol = c(3, 1))
for(xpd in c(FALSE, TRUE, NA)) {
  plot(1:10, main = paste("xpd =", xpd))
  box("figure", col = "pink", lwd = 3)
  polygon(x, y, xpd = xpd, col = "orange", lty = 2, lwd = 2, border = "red")
}
par(op)

n <- 100
xx <- c(0:n, n:0)
yy <- c(c(0, cumsum(stats::rnorm(n))), rev(c(0, cumsum(stats::rnorm(n)))))
plot   (xx, yy, type = "n", xlab = "Time", ylab = "Distance")
polygon(xx, yy, col = "gray", border = "red")
title("Distance Between Brownian Motions")

# Multiple polygons from NA values
# and recycling of col, border, and lty
op <- par(mfrow = c(2, 1))
plot(c(1, 9), 1:2, type = "n")
polygon(1:9, c(2,1,2,1,1,2,1,2,1),
        col = c("red", "blue"),
        border = c("green", "yellow"),
        lwd = 3, lty = c("dashed", "solid"))
plot(c(1, 9), 1:2, type = "n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        col = c("red", "blue"),
        border = c("green", "yellow"),
        lwd = 3, lty = c("dashed", "solid"))
par(op)

# Line-shaded polygons
plot(c(1, 9), 1:2, type = "n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        density = c(10, 20), angle = c(-45, 45))
```

---

polypath *Path Drawing*

Description

path draws a path whose vertices are given in x and y.

Usage

```
polypath(x, y = NULL,
         border = NULL, col = NA, lty = par("lty"),
         rule = "winding", ...)
```

Arguments

x, y	vectors containing the coordinates of the vertices of the path.
col	the color for filling the path. The default, NA, is to leave paths unfilled.
border	the color to draw the border. The default, NULL, means to use par("fg"). Use border = NA to omit borders. For compatibility with S, border can also be logical, in which case FALSE is equivalent to NA (borders omitted) and TRUE is equivalent to NULL (use the foreground colour),
lty	the line type to be used, as in par.
rule	character value specifying the path fill mode: either "winding" or "evenodd".
...	graphical parameters such as xpd, lend, ljoin and lmitre can be given as arguments.

Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, ... See xy.coords.

It is assumed that the path is to be closed by joining the last point to the first point.

The coordinates can contain missing values. The behaviour is similar to that of polygon, except that instead of breaking a polygon into several polygons, NA values break the path into several sub-paths (including closing the last point to the first point in each sub-path). See the examples below.

The distinction between a path and a polygon is that the former can contain holes, as interpreted by the fill rule; these fill a region if the path border encircles it an odd or non-zero number of times, respectively.

Hatched shading (as implemented for polygon()) is not (currently) supported.

Not all graphics devices support this function: for example xfig and pictex do not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

segments for even more flexibility, lines, rect, box, polygon.

par for how to specify colors.

Examples

```
plotPath <- function(x, y, col = "grey", rule = "winding") {
    plot.new()
    plot.window(range(x, na.rm = TRUE), range(y, na.rm = TRUE))
    polypath(x, y, col = col, rule = rule)
    if (!is.na(col))
        mtext(paste("Rule:", rule), side = 1, line = 0)
}

plotRules <- function(x, y, title) {
    plotPath(x, y)
    plotPath(x, y, rule = "evenodd")
    mtext(title, side = 3, line = 0)
    plotPath(x, y, col = NA)
}

op <- par(mfrow = c(5, 3), mar = c(2, 1, 1, 1))

plotRules(c(.1, .1, .9, .9, NA, .2, .2, .8, .8),
          c(.1, .9, .9, .1, NA, .2, .8, .8, .2),
          "Nested rectangles, both clockwise")
plotRules(c(.1, .1, .9, .9, NA, .2, .8, .8, .2),
          c(.1, .9, .9, .1, NA, .2, .2, .8, .8),
          "Nested rectangles, outer clockwise, inner anti-clockwise")
plotRules(c(.1, .1, .4, .4, NA, .6, .9, .9, .6),
          c(.1, .4, .4, .1, NA, .6, .6, .9, .9),
          "Disjoint rectangles")
plotRules(c(.1, .1, .6, .6, NA, .4, .4, .9, .9),
          c(.1, .6, .6, .1, NA, .4, .9, .9, .4),
          "Overlapping rectangles, both clockwise")
plotRules(c(.1, .1, .6, .6, NA, .4, .9, .9, .4),
          c(.1, .6, .6, .1, NA, .4, .4, .9, .9),
          "Overlapping rectangles, one clockwise, other anti-clockwise")

par(op)
```

---

rasterImage *Draw One or More Raster Images*

Description

rasterImage draws a raster image at the given locations and sizes.

Usage

```
rasterImage(image,
            xleft, ybottom, xright, ytop,
            angle = 0, interpolate = TRUE, ...)
```

Arguments

image	a raster object, or an object that can be coerced to one by as.raster.
xleft	a vector (or scalar) of left x positions.
ybottom	a vector (or scalar) of bottom y positions.
xright	a vector (or scalar) of right x positions.
ytop	a vector (or scalar) of top y positions.
angle	angle of rotation (in degrees, anti-clockwise from positive x-axis, about the bottom-left corner).
interpolate	a logical vector (or scalar) indicating whether to apply linear interpolation to the image when drawing.
...	graphical parameters.

Details

The positions supplied, i.e., xleft, ..., are relative to the current plotting region. If the x-axis goes from 100 to 200 then xleft should be larger than 100 and xright should be less than 200. The position vectors will be recycled to the length of the longest.

Plotting raster images is not supported on all devices and may have limitations where supported, for example (e.g., for postscript and X11(type = "Xlib") is restricted to opaque colors). Problems with the rendering of raster images have been reported by users of windows() devices under Remote Desktop, at least under its default settings.

You should not expect a raster image to be re-sized when an on-screen device is re-sized: whether it is is device-dependent.

See Also

rect, polygon, and segments and others for flexible ways to draw shapes.

dev.capabilities to see if it is supported.

Examples

```
require(grDevices)
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab = "", ylab = "")
image <- as.raster(matrix(0:1, ncol = 5, nrow = 3))
rasterImage(image, 100, 300, 150, 350, interpolate = FALSE)
rasterImage(image, 100, 400, 150, 450)
rasterImage(image, 200, 300, 200 + xinch(.5), 300 + yinch(.3),
            interpolate = FALSE)
rasterImage(image, 200, 400, 250, 450, angle = 15, interpolate = FALSE)
par(op)
```

`rect` *Draw One or More Rectangles*

Description

`rect` draws a rectangle (or sequence of rectangles) with the given coordinates, fill and border colors.

Usage

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"),
     ...)
```

Arguments

`xleft`	a vector (or scalar) of left x positions.
`ybottom`	a vector (or scalar) of bottom y positions.
`xright`	a vector (or scalar) of right x positions.
`ytop`	a vector (or scalar) of top y positions.
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. A zero value of `density` means no shading lines whereas negative values (and `NA`) suppress shading (and so allow color filling).
`angle`	angle (in degrees) of the shading lines.
`col`	color(s) to fill or shade the rectangle(s) with. The default `NA` (or also `NULL`) means do not fill, i.e., draw transparent rectangles, unless `density` is specified.
`border`	color for rectangle border(s). The default means `par("fg")`. Use `border = NA` to omit borders. If there are shading lines, `border = TRUE` means use the same colour for the border as for the shading lines.
`lty`	line type for borders and shading; defaults to `"solid"`.
`lwd`	line width for borders and shading. Note that the use of `lwd = 0` (as in the examples) is device-dependent.
`...`	graphical parameters such as `xpd`, `lend`, `ljoin` and `lmitre` can be given as arguments.

Details

The positions supplied, i.e., `xleft, ...`, are relative to the current plotting region. If the x-axis goes from 100 to 200 then `xleft` must be larger than 100 and `xright` must be less than 200. The position vectors will be recycled to the length of the longest.

It is a graphics primitive used in `hist`, `barplot`, `legend`, etc.

See Also

`box` for the standard box around the plot; `polygon` and `segments` for flexible line drawing.

`par` for how to specify colors.

Examples

```
require(grDevices)
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab = "", ylab = "",
     main = "2 x 11 rectangles; 'rect(100+i,300+i,  150+i,380+i)'")
i <- 4*(0:10)
## draw rectangles with bottom left (100, 300)+i
## and top right (150, 380)+i
rect(100+i, 300+i, 150+i, 380+i, col = rainbow(11, start = 0.7, end = 0.1))
rect(240-i, 320+i, 250-i, 410+i, col = heat.colors(11), lwd = i/5)
## Background alternating  ( transparent / "bg" ) :
j <- 10*(0:5)
rect(125+j, 360+j,   141+j, 405+j/2, col = c(NA,0),
     border = "gold", lwd = 2)
rect(125+j, 296+j/2, 141+j, 331+j/5, col = c(NA,"midnightblue"))
mtext("+  2 x 6 rect(*, col = c(NA,0)) and  col = c(NA,\"m..blue\")")

## an example showing colouring and shading
plot(c(100, 200), c(300, 450), type= "n", xlab = "", ylab = "")
rect(100, 300, 125, 350) # transparent
rect(100, 400, 125, 450, col = "green", border = "blue") # coloured
rect(115, 375, 150, 425, col = par("bg"), border = "transparent")
rect(150, 300, 175, 350, density = 10, border = "red")
rect(150, 400, 175, 450, density = 30, col = "blue",
     angle = -30, border = "transparent")

legend(180, 450, legend = 1:4, fill = c(NA, "green", par("fg"), "blue"),
       density = c(NA, NA, 10, 30), angle = c(NA, NA, 30, -30))

par(op)
```

---

rug *Add a Rug to a Plot*

Description

Adds a *rug* representation (1-d plot) of the data to the plot.

Usage

```
rug(x, ticksize = 0.03, side = 1, lwd = 0.5, col = par("fg"),
    quiet = getOption("warn") < 0, ...)
```

Arguments

x	A numeric vector
ticksize	The length of the ticks making up the 'rug'. Positive lengths give inwards ticks.

side	On which side of the plot box the rug will be plotted. Normally 1 (bottom) or 3 (top).
lwd	The line width of the ticks. Some devices will round the default width up to 1.
col	The colour the ticks are plotted in.
quiet	logical indicating if there should be a warning about clipped values.
...	further arguments, passed to axis, such as line or pos for specifying the location of the rug.

Details

Because of the way rug is implemented, only values of x that fall within the plot region are included. There will be a warning if any finite values are omitted, but non-finite values are omitted silently.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

jitter which you may want for ties in x.

Examples

```
require(stats)  # both 'density' and its default method
with(faithful, {
    plot(density(eruptions, bw = 0.15))
    rug(eruptions)
    rug(jitter(eruptions, amount = 0.01), side = 3, col = "light blue")
})
```

---

screen *Creating and Controlling Multiple Screens on a Single Device*

Description

split.screen defines a number of regions within the current device which can, to some extent, be treated as separate graphics devices. It is useful for generating multiple plots on a single device. Screens can themselves be split, allowing for quite complex arrangements of plots.

screen is used to select which screen to draw in.

erase.screen is used to clear a single screen, which it does by filling with the background colour.

close.screen removes the specified screen definition(s).

Usage

```
split.screen(figs, screen, erase = TRUE)
screen(n = , new = TRUE)
erase.screen(n = )
close.screen(n, all.screens = FALSE)
```

Arguments

figs	a two-element vector describing the number of rows and the number of columns in a screen matrix *or* a matrix with 4 columns. If a matrix, then each row describes a screen with values for the left, right, bottom, and top of the screen (in that order) in NDC units, that is 0 at the lower left corner of the device surface, and 1 at the upper right corner.
screen	a number giving the screen to be split. It defaults to the current screen if there is one, otherwise the whole device region.
erase	logical: should the selected screen be cleared?
n	a number indicating which screen to prepare for drawing (`screen`), erase (`erase.screen`), or close (`close.screen`). (`close.screen` will accept a vector of screen numbers.)
new	logical value indicating whether the screen should be erased as part of the preparation for drawing in the screen.
all.screens	logical value indicating whether all of the screens should be closed.

Details

The first call to `split.screen` places R into split-screen mode. The other split-screen functions only work within this mode. While in this mode, certain other commands should be avoided (see the Warnings section below). Split-screen mode is exited by the command `close.screen(all = TRUE)`.

If the current screen is closed, `close.screen` sets the current screen to be the next larger screen number if there is one, otherwise to the first available screen.

Value

`split.screen(*)` returns a vector of screen numbers for the newly-created screens. With no arguments, `split.screen()` returns a vector of valid screen numbers.

`screen(n)` invisibly returns `n`, the number of the selected screen. With no arguments, `screen()` returns the number of the current screen.

`close.screen()` returns a vector of valid screen numbers.

`screen`, `erase.screen`, and `close.screen` all return `FALSE` if R is not in split-screen mode.

Warnings

The recommended way to use these functions is to completely draw a plot and all additions (i.e., points and lines) to the base plot, prior to selecting and plotting on another screen. The behavior associated with returning to a screen to add to an existing plot is unpredictable and may result in problems that are not readily visible.

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `layout()`.

The functions are also incompatible with some plotting functions, such as `coplot`, which make use of these other mechanisms.

`erase.screen` will appear not to work if the background colour is transparent (as it is by default on most devices).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

par, layout, Devices, dev.*

Examples

```
if (interactive()) {
par(bg = "white")           # default is likely to be transparent
split.screen(c(2, 1))       # split display into two screens
split.screen(c(1, 3), screen = 2) # now split the bottom half into 3
screen(1) # prepare screen 1 for output
plot(10:1)
screen(4) # prepare screen 4 for output
plot(10:1)
close.screen(all = TRUE)    # exit split-screen mode

split.screen(c(2, 1))       # split display into two screens
split.screen(c(1, 2), 2)    # split bottom half in two
plot(1:10)                  # screen 3 is active, draw plot
erase.screen()              # forgot label, erase and redraw
plot(1:10, ylab = "ylab 3")
screen(1)                   # prepare screen 1 for output
plot(1:10)
screen(4)                   # prepare screen 4 for output
plot(1:10, ylab = "ylab 4")
screen(1, FALSE)            # return to screen 1, but do not clear
plot(10:1, axes = FALSE, lty = 2, ylab = "")  # overlay second plot
axis(4)                     # add tic marks to right-hand axis
title("Plot 1")
close.screen(all = TRUE)    # exit split-screen mode
}
```

---

segments *Add Line Segments to a Plot*

Description

Draw line segments between pairs of points.

Usage

```
segments(x0, y0, x1 = x0, y1 = y0,
         col = par("fg"), lty = par("lty"), lwd = par("lwd"),
         ...)
```

Arguments

x0, y0	coordinates of points **from** which to draw.
x1, y1	coordinates of points **to** which to draw. At least one must be supplied.
col, lty, lwd	graphical parameters as in par, possibly vectors. NA values in col cause the segment to be omitted.
...	further graphical parameters (from par), such as xpd and the line characteristics lend, ljoin and lmitre.

Details

For each i, a line segment is drawn between the point (x0[i], y0[i]) and the point (x1[i], y1[i]). The coordinate vectors will be recycled to the length of the longest.

The graphical parameters col, lty and lwd can be vectors of length greater than one and will be recycled if necessary.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

arrows, polygon for slightly easier and less flexible line drawing, and lines for the usual polygons.

Examples

```
x <- stats::runif(12); y <- stats::rnorm(12)
i <- order(x, y); x <- x[i]; y <- y[i]
plot(x, y, main = "arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)  # one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

smoothScatter *Scatterplots with Smoothed Densities Color Representation*

Description

smoothScatter produces a smoothed color density representation of a scatterplot, obtained through a (2D) kernel density estimate.

Usage

```
smoothScatter(x, y = NULL, nbin = 128, bandwidth,
              colramp = colorRampPalette(c("white", blues9)),
              nrpoints = 100, ret.selection = FALSE,
              pch = ".", cex = 1, col = "black",
              transformation = function(x) x^.25,
              postPlotHook = box,
              xlab = NULL, ylab = NULL, xlim, ylim,
              xaxs = par("xaxs"), yaxs = par("yaxs"), ...)
```

Arguments

x, y	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function xy.coords for details. If supplied separately, they must be of the same length.
nbin	numeric vector of length one (for both directions) or two (for x and y separately) specifying the number of equally spaced grid points for the density estimation; directly used as gridsize in bkde2D().
bandwidth	numeric vector (length 1 or 2) of smoothing bandwidth(s). If missing, a more or less useful default is used. bandwidth is subsequently passed to function bkde2D.
colramp	function accepting an integer n as an argument and returning n colors.
nrpoints	number of points to be superimposed on the density image. The first nrpoints points from those areas of lowest regional densities will be plotted. Adding points to the plot allows for the identification of outliers. If all points are to be plotted, choose nrpoints = Inf.
ret.selection	logical indicating to return the ordered indices of "low density" points if nrpoints > 0.
pch, cex, col	arguments passed to points, when nrpoints > 0: point symbol, character expansion factor and color, see also par.
transformation	function mapping the density scale to the color scale.
postPlotHook	either NULL or a function which will be called (with no arguments) after image.
xlab, ylab	character strings to be used as axis labels, passed to image.
xlim, ylim	numeric vectors of length 2 specifying axis limits.
xaxs, yaxs, ...	further arguments passed to image, e.g., add=TRUE or useRaster=TRUE.

Details

smoothScatter produces a smoothed version of a scatter plot. Two dimensional (kernel density) smoothing is performed by bkde2D from package **KernSmooth**. See the examples for how to use this function together with pairs.

Value

If ret.selection is true, a vector of integers of length nrpoints (or smaller, if there are less finite points inside xlim and ylim) with the indices of the low-density points drawn, ordered with lowest density first.

Author(s)

Florian Hahne at FHCRC, originally

See Also

`bkde2D` from package **KernSmooth**; `densCols` which uses the same smoothing computations and `blues9` in package **grDevices**.

`scatter.smooth` adds a `loess` regression smoother to a scatter plot.

Examples

```
## A largish data set
n <- 10000
x1  <- matrix(rnorm(n), ncol = 2)
x2  <- matrix(rnorm(n, mean = 3, sd = 1.5), ncol = 2)
x   <- rbind(x1, x2)

oldpar <- par(mfrow = c(2, 2), mar=.1+c(3,3,1,1), mgp = c(1.5, 0.5, 0))
smoothScatter(x, nrpoints = 0)
smoothScatter(x)

## a different color scheme:
Lab.palette <- colorRampPalette(c("blue", "orange", "red"), space = "Lab")
i.s <- smoothScatter(x, colramp = Lab.palette,
                     ## pch=NA: do not draw them
                     nrpoints = 250, ret.selection=TRUE)
## label the 20 very lowest-density points,the "outliers" (with obs.number):
i.20 <- i.s[1:20]
text(x[i.20,], labels = i.20, cex= 0.75)

## somewhat similar, using identical smoothing computations,
## but considerably *less* efficient for really large data:
plot(x, col = densCols(x), pch = 20)

## use with pairs:
par(mfrow = c(1, 1))
y <- matrix(rnorm(40000), ncol = 4) + 3*rnorm(10000)
y[, c(2,4)] <-  -y[, c(2,4)]
pairs(y, panel = function(...) smoothScatter(..., nrpoints = 0, add = TRUE),
      gap = 0.2)

par(oldpar)
```

---

spineplot *Spine Plots and Spinograms*

Description

Spine plots are a special cases of mosaic plots, and can be seen as a generalization of stacked (or highlighted) bar plots. Analogously, spinograms are an extension of histograms.

Usage

```
spineplot(x, ...)

## Default S3 method:
spineplot(x, y = NULL,
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          ylevels = NULL, col = NULL,
          main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), axes = TRUE, weights = NULL, ...)

## S3 method for class 'formula'
spineplot(formula, data = NULL,
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          ylevels = NULL, col = NULL,
          main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), axes = TRUE, ...,
          subset = NULL, weights = NULL, drop.unused.levels = FALSE)
```

Arguments

x	an object, the default method expects either a single variable (interpreted to be the explanatory variable) or a 2-way table. See details.
y	a "factor" interpreted to be the dependent variable
formula	a "formula" of type y ~ x with a single dependent "factor" and a single explanatory variable.
data	an optional data frame.
breaks	if the explanatory variable is numeric, this controls how it is discretized. breaks is passed to hist and can be a list of arguments.
tol.ylab	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
off	vertical offset between the bars (in per cent). It is fixed to 0 for spinograms and defaults to 2 for spine plots.
ylevels	a character or numeric vector specifying in which order the levels of the dependent variable should be plotted.
col	a vector of fill colors of the same length as levels(y). The default is to call gray.colors.
main, xlab, ylab	character strings for annotation
xaxlabels, yaxlabels	character vectors for annotation of x and y axis. Default to levels(y) and levels(x), respectively for the spine plot. For xaxlabels in the spinogram, the breaks are used.
xlim, ylim	the range of x and y values with sensible defaults.
axes	logical. If FALSE all axes (including those giving level names) are suppressed.

weights	numeric. A vector of frequency weights for each observation in the data. If NULL all weights are implicitly assumed to be 1. If x is already a 2-way table, the weights are ignored.
...	additional arguments passed to rect.
subset	an optional vector specifying a subset of observations to be used for plotting.
drop.unused.levels	should factors have unused levels dropped? Defaults to FALSE.

Details

spineplot creates either a spinogram or a spine plot. It can be called via spineplot(x, y) or spineplot(y ~ x) where y is interpreted to be the dependent variable (and has to be categorical) and x the explanatory variable. x can be either categorical (then a spine plot is created) or numerical (then a spinogram is plotted). Additionally, spineplot can also be called with only a single argument which then has to be a 2-way table, interpreted to correspond to table(x, y).

Both, spine plots and spinograms, are essentially mosaic plots with special formatting of spacing and shading. Conceptually, they plot $P(y|x)$ against $P(x)$. For the spine plot (where both x and y are categorical), both quantities are approximated by the corresponding empirical relative frequencies. For the spinogram (where x is numerical), x is first discretized (by calling hist with breaks argument) and then empirical relative frequencies are taken.

Thus, spine plots can also be seen as a generalization of stacked bar plots where not the heights but the widths of the bars corresponds to the relative frequencies of x. The heights of the bars then correspond to the conditional relative frequencies of y in every x group. Analogously, spinograms extend stacked histograms.

Value

The table visualized is returned invisibly.

Author(s)

Achim Zeileis <Achim.Zeileis@R-project.org>

References

Friendly, M. (1994). Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200. doi:10.2307/2291215.

Hartigan, J.A., and Kleiner, B. (1984). A mosaic of television ratings. *The American Statistician*, **38**, 32–35. doi:10.2307/2683556.

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*. Unpublished Manuscript.

Hummel, J. (1996). Linked bar charts: Analysing categorical data graphically. *Computational Statistics*, **11**, 23–33.

See Also

mosaicplot, hist, cdplot

Examples

```
## treatment and improvement of patients with rheumatoid arthritis
treatment <- factor(rep(c(1, 2), c(43, 41)), levels = c(1, 2),
                    labels = c("placebo", "treated"))
improved <- factor(rep(c(1, 2, 3, 1, 2, 3), c(29, 7, 7, 13, 7, 21)),
                   levels = c(1, 2, 3),
                   labels = c("none", "some", "marked"))

## (dependence on a categorical variable)
(spineplot(improved ~ treatment))

## applications and admissions by department at UC Berkeley
## (two-way tables)
(spineplot(marginSums(UCBAdmissions, c(3, 2)),
           main = "Applications at UCB"))
(spineplot(marginSums(UCBAdmissions, c(3, 1)),
           main = "Admissions at UCB"))

## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1,
                 1, 1, 1, 2, 1, 1, 1, 1, 1),
               levels = c(1, 2), labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70,
                 70, 70, 72, 73, 75, 75, 76, 76, 78, 79, 81)

## (dependence on a numerical variable)
(spineplot(fail ~ temperature))
(spineplot(fail ~ temperature, breaks = 3))
(spineplot(fail ~ temperature, breaks = quantile(temperature)))

## highlighting for failures
spineplot(fail ~ temperature, ylevels = 2:1)
```

---

stars *Star (Spider/Radar) Plots and Segment Diagrams*

Description

Draw star plots or segment diagrams of a multivariate data set. With one single location, also draws 'spider' (or 'radar') plots.

Usage

```
stars(x, full = TRUE, scale = TRUE, radius = TRUE,
      labels = dimnames(x)[[1]], locations = NULL,
      nrow = NULL, ncol = NULL, len = 1,
      key.loc = NULL, key.labels = dimnames(x)[[2]],
      key.xpd = TRUE,
      xlim = NULL, ylim = NULL, flip.labels = NULL,
```

```
      draw.segments = FALSE,
      col.segments = 1:n.seg, col.stars = NA, col.lines = NA,
      axes = FALSE, frame.plot = axes,
      main = NULL, sub = NULL, xlab = "", ylab = "",
      cex = 0.8, lwd = 0.25, lty = par("lty"), xpd = FALSE,
      mar = pmin(par("mar"),
                 1.1+ c(2*axes+ (xlab != ""),
                 2*axes+ (ylab != ""), 1, 0)),
      add = FALSE, plot = TRUE, ...)
```

Arguments

x	matrix or data frame of data. One star or segment plot will be produced for each row of x. Missing values (NA) are allowed, but they are treated as if they were 0 (after scaling, if relevant).
full	logical flag: if TRUE, the segment plots will occupy a full circle. Otherwise, they occupy the (upper) semicircle only.
scale	logical flag: if TRUE, the columns of the data matrix are scaled independently so that the maximum value in each column is 1 and the minimum is 0. If FALSE, the presumption is that the data have been scaled by some other algorithm to the range $[0, 1]$.
radius	logical flag: in TRUE, the radii corresponding to each variable in the data will be drawn.
labels	vector of character strings for labeling the plots. Unlike the S function stars, no attempt is made to construct labels if labels = NULL.
locations	Either two column matrix with the x and y coordinates used to place each of the segment plots; or numeric of length 2 when all plots should be superimposed (for a 'spider plot'). By default, locations = NULL, the segment plots will be placed in a rectangular grid.
nrow, ncol	integers giving the number of rows and columns to use when locations is NULL. By default, nrow == ncol, a square layout will be used.
len	scale factor for the length of radii or segments.
key.loc	vector with x and y coordinates of the unit key.
key.labels	vector of character strings for labeling the segments of the unit key. If omitted, the second component of dimnames(x) is used, if available.
key.xpd	clipping switch for the unit key (drawing and labeling), see par("xpd").
xlim	vector with the range of x coordinates to plot.
ylim	vector with the range of y coordinates to plot.
flip.labels	logical indicating if the label locations should flip up and down from diagram to diagram. Defaults to a somewhat smart heuristic.
draw.segments	logical. If TRUE draw a segment diagram.
col.segments	color vector (integer or character, see par), each specifying a color for one of the segments (variables). Ignored if draw.segments = FALSE.

col.stars	color vector (integer or character, see par), each specifying a color for one of the stars (cases). Ignored if draw.segments = TRUE.
col.lines	color vector (integer or character, see par), each specifying a color for one of the lines (cases). Ignored if draw.segments = TRUE.
axes	logical flag: if TRUE axes are added to the plot.
frame.plot	logical flag: if TRUE, the plot region is framed.
main	a main title for the plot.
sub	a subtitle for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
cex	character expansion factor for the labels.
lwd	line width used for drawing.
lty	line type used for drawing.
xpd	logical or NA indicating if clipping should be done, see par(xpd = .).
mar	argument to par(mar = *), typically choosing smaller margins than by default.
...	further arguments, passed to the first call of plot(), see plot.default and to box() if frame.plot is true.
add	logical, if TRUE *add* stars to current plot.
plot	logical, if FALSE, nothing is plotted.

Details

Missing values are treated as 0.

Each star plot or segment diagram represents one row of the input x. Variables (columns) start on the right and wind counterclockwise around the circle. The size of the (scaled) column is shown by the distance from the center to the point on the star or the radius of the segment representing the variable.

Only one page of output is produced.

Value

Returns the locations of the plots in a two column matrix, invisibly when plot = TRUE.

Note

This code started life as spatial star plots by David A. Andrews.

Prior to R 1.4.1, scaling only shifted the maximum to 1, although documented as here.

Author(s)

Thomas S. Dye

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`symbols` for another way to draw stars and other symbols.

Examples

```
require(grDevices)
stars(mtcars[, 1:7], key.loc = c(14, 2),
      main = "Motor Trend Cars : stars(*, full = F)", full = FALSE)
stars(mtcars[, 1:7], key.loc = c(14, 1.5),
      main = "Motor Trend Cars : full stars()", flip.labels = FALSE)

## 'Spider' or 'Radar' plot:
stars(mtcars[, 1:7], locations = c(0, 0), radius = FALSE,
      key.loc = c(0, 0), main = "Motor Trend Cars", lty = 2)

## Segment Diagrams:
palette(rainbow(12, s = 0.6, v = 0.75))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
      main = "Motor Trend Cars", draw.segments = TRUE)
stars(mtcars[, 1:7], len = 0.6, key.loc = c(1.5, 0),
      main = "Motor Trend Cars", draw.segments = TRUE,
      frame.plot = TRUE, nrow = 4, cex = .7)

## scale linearly (not affinely) to [0, 1]
USJudge <- apply(USJudgeRatings, 2, function(x) x/max(x))
Jnam <- row.names(USJudgeRatings)
Snam <- abbreviate(substring(Jnam, 1, regexpr("[,.]",Jnam) - 1), 7)
stars(USJudge, labels = Jnam, scale = FALSE,
      key.loc = c(13, 1.5), main = "Judge not ...", len = 0.8)
stars(USJudge, labels = Snam, scale = FALSE,
      key.loc = c(13, 1.5), radius = FALSE)

loc <- stars(USJudge, labels = NULL, scale = FALSE,
             radius = FALSE, frame.plot = TRUE,
             key.loc = c(13, 1.5), main = "Judge not ...", len = 1.2)
text(loc, Snam, col = "blue", cex = 0.8, xpd = TRUE)

## 'Segments':
stars(USJudge, draw.segments = TRUE, scale = FALSE, key.loc = c(13,1.5))

## 'Spider':
stars(USJudgeRatings, locations = c(0, 0), scale = FALSE, radius  =  FALSE,
      col.stars = 1:10, key.loc = c(0, 0), main = "US Judges rated")
## Same as above, but with colored lines instead of filled polygons.
stars(USJudgeRatings, locations = c(0, 0), scale = FALSE, radius  =  FALSE,
      col.lines = 1:10, key.loc = c(0, 0), main = "US Judges rated")
## 'Radar-Segments'
```

```
stars(USJudgeRatings[1:10,], locations = 0:1, scale = FALSE,
      draw.segments = TRUE, col.segments = 0, col.stars = 1:10, key.loc =  0:1,
      main = "US Judges 1-10 ")
palette("default")
stars(cbind(1:16, 10*(16:1)), draw.segments = TRUE,
      main = "A Joke -- do *not* use symbols on 2D data!")
```

`stem` *Stem-and-Leaf Plots*

Description

`stem` produces a stem-and-leaf plot of the values in `x`. The parameter `scale` can be used to expand the scale of the plot. A value of `scale = 2` will cause the plot to be roughly twice as long as the default.

Usage

```
stem(x, scale = 1, width = 80, atom = 1e-08)
```

Arguments

`x`	a numeric vector.
`scale`	This controls the plot length.
`width`	The desired width of plot.
`atom`	a tolerance.

Details

Infinite and missing values in `x` are discarded.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
stem(islands)
stem(log10(islands))
```

stripchart *1-D Scatter Plots*

Description

stripchart produces one dimensional scatter plots (or dot plots) of the given data. These plots are a good alternative to boxplots when sample sizes are small.

Usage

```
stripchart(x, ...)

## S3 method for class 'formula'
stripchart(x, data = NULL, dlab = NULL, ...,
           subset, na.action = NULL)


## Default S3 method:
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
           vertical = FALSE, group.names, add = FALSE,
           at = NULL, xlim = NULL, ylim = NULL,
           ylab = NULL, xlab = NULL, dlab = "", glab = "",
           log = "", pch = 0, col = par("fg"), cex = par("cex"),
           axes = TRUE, frame.plot = axes, ...)
```

Arguments

x	the data from which the plots are to be produced. In the default method the data can be specified as a single numeric vector, or as list of numeric vectors, each corresponding to a component plot. In the formula method, a symbolic specification of the form y ~ g can be given, indicating the observations in the vector y are to be grouped according to the levels of the factor g. NAs are allowed in the data.
data	a data.frame (or list) from which the variables in x should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
...	additional parameters passed to the default method, or by it to plot.window, points, axis and title to control the appearance of the plot.
method	the method to be used to separate coincident points. The default method "overplot" causes such points to be overplotted, but it is also possible to specify "jitter" to jitter the points, or "stack" have coincident points stacked. The last method only makes sense for very granular data.
jitter	when method = "jitter" is used, jitter gives the amount of jittering applied.

offset	when stacking is used, points are stacked this many line-heights (symbol widths) apart.
vertical	when vertical is TRUE the plots are drawn vertically rather than the default horizontal.
group.names	group labels which will be printed alongside (or underneath) each plot.
add	logical, if true *add* the chart to the current plot.
at	numeric vector giving the locations where the charts should be drawn, particularly when add = TRUE; defaults to 1:n where n is the number of boxes.
ylab, xlab	labels: see title.
dlab, glab	alternate way to specify axis labels: see 'Details'.
xlim, ylim	plot limits: see plot.window.
log	on which axes to use a log scale: see plot.default
pch, col, cex	Graphical parameters: see par.
axes, frame.plot	Axis control: see plot.default.

Details

Extensive examples of the use of this kind of plot can be found in Box, Hunter and Hunter (2005) or Wild and Seber (2000).

The dlab and glab labels may be used instead of xlab and ylab if those are not specified. dlab applies to the continuous data axis (the X axis unless vertical is TRUE), glab to the group axis.

References

Box G., Hunter, J. S. and Hunter, W. C. (2005). *Statistics for Experimenters: Design, Innovation, and Discovery*, second edition. New York: Wiley. ISBN: 978-0-471-71813-0.

Wild, C. and Seber, G. (2000). *Chance Encounters: A First Course in Data Analysis and Inference*. John Wiley and Sons. ISBN 0-471-32936-3.

Examples

```
x <- stats::rnorm(50)
xr <- round(x, 1)
stripchart(x) ; m <- mean(par("usr")[1:2])
text(m, 1.04, "stripchart(x, \"overplot\")")
stripchart(xr, method = "stack", add = TRUE, at = 1.2)
text(m, 1.35, "stripchart(round(x,1), \"stack\")")
stripchart(xr, method = "jitter", add = TRUE, at = 0.7)
text(m, 0.85, "stripchart(round(x,1), \"jitter\")")

stripchart(decrease ~ treatment,
    main = "stripchart(OrchardSprays)",
    vertical = TRUE, log = "y", data = OrchardSprays)

stripchart(decrease ~ treatment, at = c(1:8)^2,
    main = "stripchart(OrchardSprays)",
    vertical = TRUE, log = "y", data = OrchardSprays)
```

strwidth *Plotting Dimensions of Character Strings and Math Expressions*

Description

These functions compute the width or height, respectively, of the given strings or mathematical expressions `s[i]` on the current plotting device in *user* coordinates, *inches* or as fraction of the figure width `par("fin")`.

Usage

```
strwidth(s, units = "user", cex = NULL, font = NULL, vfont = NULL, ...)
strheight(s, units = "user", cex = NULL, font = NULL, vfont = NULL, ...)
```

Arguments

`s`	a character or expression vector whose dimensions are to be determined. Other objects are coerced by `as.graphicsAnnot`.
`units`	character indicating in which units `s` is measured; should be one of `"user"`, `"inches"`, `"figure"`; partial matching is performed.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size; the default `NULL` is equivalent to `1`.
`font, vfont, ...`	additional information about the font, possibly including the graphics parameter `"family"`: see `text`.

Details

Note that the 'height' of a string is determined only by the number of linefeeds (`"\n"`, aka "newline"s) it contains: it is the (number of linefeeds - 1) times the line spacing plus the height of `"M"` in the selected font. For an expression it is the height of the bounding box as computed by plotmath. Thus in both cases it is an estimate of how far **above** the final baseline the typeset object extends. (It may also extend below the baseline.) The inter-line spacing is controlled by `cex`, `par("lheight")` and the 'point size' (but not the actual font in use).

Measurements in `"user"` units (the default) are only available after `plot.new` has been called – otherwise an error is thrown.

Value

Numeric vector with the same length as `s`, giving the estimate of width or height for each `s[i]`. `NA` strings are given width and height 0 (as they are not plotted).

See Also

`text`, `nchar`

Examples

```
str.ex <- c("W","w","I",".","WwI.")
op <- par(pty = "s"); plot(1:100, 1:100, type = "n")
sw <- strwidth(str.ex); sw
all.equal(sum(sw[1:4]), sw[5])
#- since the last string contains the others

sw.i <- strwidth(str.ex, "inches"); 25.4 * sw.i  # width in [mm]
unique(sw / sw.i)
# constant factor: 1 value
mean(sw.i / strwidth(str.ex, "fig")) / par('fin')[1]  # = 1: are the same

## See how letters fall in classes
##  -- depending on graphics device and font!
all.lett <- c(letters, LETTERS)
shL <- strheight(all.lett, units = "inches") * 72  # 'big points'
table(shL)  # all have same heights ...
mean(shL)/par("cin")[2] # around 0.6

(swL <- strwidth(all.lett, units = "inches") * 72)  # 'big points'
split(all.lett, factor(round(swL, 2)))

sumex <- expression(sum(x[i], i=1,n), e^{i * pi} == -1)
strwidth(sumex)
strheight(sumex)

par(op)  #- reset to previous setting
```

---

sunflowerplot *Produce a Sunflower Scatter Plot*

Description

Multiple points are plotted as 'sunflowers' with multiple leaves ('petals') such that overplotting is visualized instead of accidental and invisible.

Usage

```
sunflowerplot(x, ...)

## Default S3 method:
sunflowerplot(x, y = NULL, number, log = "", digits = 6,
              xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
              add = FALSE, rotate = FALSE,
              pch = 16, cex = 0.8, cex.fact = 1.5,
              col = par("col"), bg = NA, size = 1/8, seg.col = 2,
              seg.lwd = 1.5, ...)

## S3 method for class 'formula'
```

```
sunflowerplot(formula, data = NULL, xlab = NULL, ylab = NULL, ...,
              subset, na.action = NULL)
```

Arguments

x	numeric vector of x-coordinates of length n, say, or another valid plotting structure, as for plot.default, see also xy.coords.
y	numeric vector of y-coordinates of length n.
number	integer vector of length n. number[i] = number of replicates for (x[i], y[i]), may be 0. Default (missing(number)): compute the exact multiplicity of the points x[], y[], via xyTable().
log	character indicating log coordinate scale, see plot.default.
digits	when number is computed (i.e., not specified), x and y are rounded to digits significant digits before multiplicities are computed.
xlab, ylab	character label for x-, or y-axis, respectively.
xlim, ylim	numeric(2) limiting the extents of the x-, or y-axis.
add	logical; should the plot be added on a previous one ? Default is FALSE.
rotate	logical; if TRUE, randomly rotate the sunflowers (preventing artefacts).
pch	plotting character to be used for points (number[i]==1) and center of sunflowers.
cex	numeric; character size expansion of center points (s. pch).
cex.fact	numeric *shrinking* factor to be used for the center points *when there are flower leaves*, i.e., cex / cex.fact is used for these.
col, bg	colors for the plot symbols, passed to plot.default.
size	of sunflower leaves in inches, 1[in] := 2.54[cm]. Default: 1/8\", approximately 3.2mm.
seg.col	color to be used for the **seg**ments which make the sunflowers leaves, see par(col=); col = "gold" reminds of real sunflowers.
seg.lwd	numeric; the line width for the leaves' segments.
...	further arguments to plot [if add = FALSE], or to be passed to or from another method.
formula	a formula, such as y ~ x.
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore case with missing values.

Details

This is a generic function with default and formula methods.

For `number[i] == 1`, a (slightly enlarged) usual plotting symbol (`pch`) is drawn. For `number[i] > 1`, a small plotting symbol is drawn and `number[i]` equi-angular 'rays' emanate from it.

If `rotate = TRUE` and `number[i] >= 2`, a random direction is chosen (instead of the y-axis) for the first ray. The goal is to `jitter` the orientations of the sunflowers in order to prevent artefactual visual impressions.

Value

A list with three components of same length,

`x`	x coordinates
`y`	y coordinates
`number`	number

Use `xyTable()` (from package **grDevices**) if you are only interested in this return value.

Side Effects

A scatter plot is drawn with 'sunflowers' as symbols.

Author(s)

Andreas Ruckstuhl, Werner Stahel, Martin Maechler, Tim Hesterberg, 1989–1993. Port to R by Martin Maechler <maechler@stat.math.ethz.ch>.

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Wadsworth.

Schilling, M. F. and Watkins, A. E. (1994). A suggestion for sunflower plots. *The American Statistician*, **48**, 303–305. doi:10.2307/2684839.

Murrell, P. (2005). *R Graphics*. Chapman & Hall/CRC Press.

See Also

`density`, `xyTable`

Examples

```
require(stats) # for rnorm
require(grDevices)

## 'number' is computed automatically:
sunflowerplot(iris[, 3:4])
## Imitating Chambers et al, p.109, closely:
sunflowerplot(iris[, 3:4], cex = .2, cex.fact = 1, size = .035, seg.lwd = .8)
## or
```

```
sunflowerplot(Petal.Width ~ Petal.Length, data = iris,
              cex = .2, cex.fact = 1, size = .035, seg.lwd = .8)


sunflowerplot(x = sort(2*round(rnorm(100))), y = round(rnorm(100), 0),
              main = "Sunflower Plot of Rounded N(0,1)")
## Similarly using a "xyTable" argument:
xyT <- xyTable(x = sort(2*round(rnorm(100))), y = round(rnorm(100), 0),
               digits = 3)
utils::str(xyT, vec.len = 20)
sunflowerplot(xyT, main = "2nd Sunflower Plot of Rounded N(0,1)")

## A 'marked point process' {explicit 'number' argument}:
sunflowerplot(rnorm(100), rnorm(100), number = rpois(n = 100, lambda = 2),
              main = "Sunflower plot (marked point process)",
              rotate = TRUE, col = "blue4")
```

`symbols` *Draw Symbols (Circles, Squares, Stars, Thermometers, Boxplots)*

Description

This function draws symbols on a plot. One of six symbols; *circles*, *squares*, *rectangles*, *stars*, *thermometers*, and *boxplots*, can be plotted at a specified set of x and y coordinates. Specific aspects of the symbols, such as relative size, can be customized by additional parameters.

Usage

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = par("col"), bg = NA,
        xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

Arguments

`x, y`	the x and y co-ordinates for the centres of the symbols. They can be specified in any way which is accepted by `xy.coords`.
`circles`	a vector giving the radii of the circles.
`squares`	a vector giving the length of the sides of the squares.
`rectangles`	a matrix with two columns. The first column gives widths and the second the heights of rectangles.
`stars`	a matrix with three or more columns giving the lengths of the rays from the center of the stars. `NA` values are replaced by zeroes.

`thermometers`	a matrix with three or four columns. The first two columns give the width and height of the thermometer symbols. If there are three columns, the third is taken as a proportion: the thermometers are filled (using colour `fg`) from their base to this proportion of their height. If there are four columns, the third and fourth columns are taken as proportions and the thermometers are filled between these two proportions of their heights. The part of the box not filled in `fg` will be filled in the background colour (default transparent) given by `bg`.
`boxplots`	a matrix with five columns. The first two columns give the width and height of the boxes, the next two columns give the lengths of the lower and upper whiskers and the fifth the proportion (with a warning if not in [0,1]) of the way up the box that the median line is drawn.
`inches`	`TRUE`, `FALSE` or a positive number. See 'Details'.
`add`	if `add` is `TRUE`, the symbols are added to an existing plot, otherwise a new plot is created.
`fg`	colour(s) the symbols are to be drawn in.
`bg`	if specified, the symbols are filled with colour(s), the vector `bg` being recycled to the number of symbols. The default is to leave the symbols unfilled.
`xlab`	the x label of the plot if `add` is not true. Defaults to the `deparse`d expression used for `x`.
`ylab`	the y label of the plot. Unused if `add = TRUE`.
`main`	a main title for the plot. Unused if `add = TRUE`.
`xlim`	numeric vector of length 2 giving the x limits for the plot. Unused if `add = TRUE`.
`ylim`	numeric vector of length 2 giving the y limits for the plot. Unused if `add = TRUE`.
`...`	graphics parameters can also be passed to this function, as can the plot aspect ratio `asp` (see `plot.window`).

Details

Observations which have missing coordinates or missing size parameters are not plotted. The exception to this is *stars*. In that case, the length of any ray which is `NA` is reset to zero.

Argument `inches` controls the sizes of the symbols. If `TRUE` (the default), the symbols are scaled so that the largest dimension of any symbol is one inch. If a positive number is given the symbols are scaled to make largest dimension this size in inches (so `TRUE` and `1` are equivalent). If `inches` is `FALSE`, the units are taken to be those of the appropriate axes. (For circles, squares and stars the units of the x axis are used. For boxplots, the lengths of the whiskers are regarded as dimensions alongside width and height when scaling by `inches`, and are otherwise interpreted in the units of the y axis.)

Circles of radius zero are plotted at radius one pixel (which is device-dependent). Circles of a very small non-zero radius may or may not be visible, and may be smaller than circles of radius zero. On `windows` devices circles are plotted at radius at least one pixel as some Windows versions omit smaller circles.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

W. S. Cleveland (1985) *The Elements of Graphing Data.* Monterey, California: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`stars` for drawing *stars* with a bit more flexibility.

If you are thinking about doing 'bubble plots' by `symbols(*, circles=*)`, you should *really* consider using `sunflowerplot` instead.

Examples

```
require(stats); require(grDevices)
x <- 1:10
y <- sort(10*runif(10))
z <- runif(10)
z3 <- cbind(z, 2*runif(10), runif(10))
symbols(x, y, thermometers = cbind(.5, 1, z), inches = .5, fg = 1:10)
symbols(x, y, thermometers = z3, inches = FALSE)
text(x, y, apply(format(round(z3, digits = 2)), 1, paste, collapse = ","),
     adj = c(-.2,0), cex = .75, col = "purple", xpd = NA)

## Note that  example(trees)  shows more sensible plots!
N <- nrow(trees)
with(trees, {
## Girth is diameter in inches
symbols(Height, Volume, circles = Girth/24, inches = FALSE,
        main = "Trees' Girth") # xlab and ylab automatically
## Colours too:
op <- palette(rainbow(N, end = 0.9))
symbols(Height, Volume, circles = Girth/16, inches = FALSE, bg = 1:N,
        fg = "gray30", main = "symbols(*, circles = Girth/16, bg = 1:N)")
palette(op)
})
```

`text` *Add Text to a Plot*

Description

`text` draws the strings given in the vector `labels` at the coordinates given by `x` and `y`. `y` may be missing since `xy.coords(x, y)` is used for construction of the coordinates.

Usage

```
text(x, ...)

## Default S3 method:
 text(x, y = NULL, labels = seq_along(x$x), adj = NULL,
     pos = NULL, offset = 0.5, vfont = NULL,
     cex = 1, col = NULL, font = NULL, ...)
```

Arguments

x, y	numeric vectors of coordinates where the text `labels` should be written. If the length of `x` and `y` differs, the shorter one is recycled.
`labels`	a character vector or expression specifying the *text* to be written. An attempt is made to coerce other language objects (names and calls) to expressions, and vectors and other classed objects to character vectors by `as.character`. If `labels` is longer than `x` and `y`, the coordinates are recycled to the length of `labels`.
`adj`	one or two values in $[0, 1]$ which specify the x (and optionally y) adjustment ('justification') of the labels, with 0 for left/bottom, 1 for right/top, and 0.5 for centered. On most devices values outside $[0, 1]$ will also work. See below.
`pos`	a position specifier for the text. If specified this overrides any `adj` value given. Values of `1`, `2`, `3` and `4`, respectively indicate positions below, to the left of, above and to the right of the specified `(x,y)` coordinates.
`offset`	when `pos` is specified, this value controls the distance ('offset') of the text label from the specified coordinate in fractions of a character width.
`vfont`	`NULL` for the current font family, or a character vector of length 2 for `Hershey` vector fonts. The first element of the vector selects a typeface and the second element selects a style. Ignored if `labels` is an expression.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size. `NULL` and `NA` are equivalent to `1.0`.
col, font	the color and (if `vfont = NULL`) font to be used, possibly vectors. These default to the values of the global graphical parameters in `par()`.
...	further graphical parameters (from `par`), such as `srt`, `family` and `xpd`.

Details

`labels` must be of type `character` or `expression` (or be coercible to such a type). In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, Greek letters, fractions, etc.

`adj` allows *adj*ustment of the text position with respect to `(x, y)`. Values of 0, 0.5, and 1 specify that `(x, y)` should align with the left/bottom, middle and right/top of the text, respectively. A value of `NA` means "centre", which is the same as 0.5 for horizontal justification, but includes descenders for vertical justification (where 0.5 does not). The default is for centered text, although the default horizontal justification is taken from `par(adj)`, i.e., the default is `adj = c(par("adj"), NA)`. If only one value is provided, it is applied to adjust `x` *only*, i.e., when `length(adj) == 1L`, `adj` is applied as `adj = c(adj, NA)`. Accurate vertical centering needs character metric information on individual characters which is only available on some devices. Vertical alignment is done slightly

differently for character strings and for expressions: `adj = c(0,0)` means to left-justify and to align on the baseline for strings but on the bottom of the bounding box for expressions. This also affects vertical centering: for strings the centering excludes any descenders whereas for expressions it includes them.

The `pos` and `offset` arguments can be used in conjunction with values returned by `identify` to recreate an interactively labelled plot.

Text can be rotated by using graphical parameters `srt` (see `par`). When `adj` is specified, a non-zero `srt` rotates the label about `(x, y)`. If `pos` is specified, `srt` rotates the text about the point on its bounding box which is closest to `(x, y)`: top center for `pos = 1`, right center for `pos = 2`, bottom center for `pos = 3`, and left center for `pos = 4`. The `pos` interface is not as useful for rotated text because the result is no longer centered vertically or horizontally with respect to `(x, y)`. At present there is no interface in the **graphics** package for directly rotating text about its center which is achievable however by fiddling with `adj` and `srt` simultaneously.

Graphical parameters `col`, `cex` and `font` can be vectors and will then be applied cyclically to the `labels` (and extra values will be ignored). `NA` values of `font` are replaced by `par("font")`, and similarly for `col`.

Labels whose `x`, `y` or `labels` value is `NA` are omitted from the plot.

What happens when `font = 5` (the symbol font) is selected can be both device- and locale-dependent. Most often `labels` will be interpreted in the Adobe symbol encoding, so e.g. `"d"` is delta, and `"\300"` is aleph.

Euro symbol

The Euro symbol may not be available in older fonts. In current versions of Adobe symbol fonts it is character 160, so `text(x, y, "\xA0", font = 5)` may work. People using Western European locales on Unix-alikes can probably select ISO-8895-15 (Latin-9) which has the Euro as character 165: this can also be used for `pdf` and `postscript`. It is '`\u20ac`' in Unicode, which can be used in UTF-8 locales.

The Euro should be rendered correctly by `X11` in UTF-8 locales, but the corresponding single-byte encoding in `postscript` and `pdf` will need to be selected as `ISOLatin9.enc` (and the font will need to contain the Euro glyph, which for example older printers may not).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`text.formula` for the formula method; `mtext`, `title`, `Hershey` for details on Hershey vector fonts, `plotmath` for details and more examples on mathematical annotation.

Examples

```
plot(-1:1, -1:1, type = "n", xlab = "Re", ylab = "Im")
K <- 16; text(exp(1i * 2 * pi * (1:K) / K), col = 2)
```

```
## The following two examples use latin1 characters: these may not
## appear correctly (or be omitted entirely).
plot(1:10, 1:10, main = "text(...) examples\n~~~~~~~~~~~~~~~",
     sub = "R is GNU ©, but not ® ...")
mtext("«Latin-1 accented chars»: éè øØ å<Å æ<Æ", side = 3)
points(c(6,2), c(2,1), pch = 3, cex = 4, col = "red")
text(6, 2, "the text is CENTERED around (x,y) = (6,2) by default",
     cex = .8)
text(2, 1, "or Left/Bottom - JUSTIFIED at (2,1) by 'adj = c(0,0)'",
     adj = c(0,0))
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .75)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))

## Two more latin1 examples
text(5, 10.2,
     "Le français, c'est facile: Règles, Liberté, Egalité, Fraternité...")
text(5, 9.8,
     "Jetz no chli züritüütsch: (noch ein bißchen Zürcher deutsch)")
```

`title` *Plot Annotation*

Description

This function can be used to add labels to a plot. Its first four principal arguments can also be used as arguments in most high-level plotting functions. They must be of type `character` or `expression`. In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, Greek letters, fractions, etc: see plotmath

Usage

```
title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
      line = NA, outer = FALSE, ...)
```

Arguments

`main`	The main title (on top) using font, size (character expansion) and color `par(c("font.main", "cex.main", "col.main"))`.
`sub`	Sub-title (at bottom) using font, size and color `par(c("font.sub", "cex.sub", "col.sub"))`.
`xlab`	X axis label using font, size and color `par(c("font.lab", "cex.lab", "col.lab"))`.
`ylab`	Y axis label, same font attributes as `xlab`.
`line`	specifying a value for `line` overrides the default placement of labels, and places them this many lines outwards from the plot edge.

outer	a logical value. If TRUE, the titles are placed in the outer margins of the plot.
...	further graphical parameters from par. Use e.g., col.main or cex.sub instead of just col or cex. adj controls the justification of the titles. xpd can be used to set the clipping region: this defaults to the figure region unless outer = TRUE, otherwise the device region and can only be increased. mgp controls the default placing of the axis titles.

Details

The labels passed to title can be character strings or language objects (names, calls or expressions), or a list containing the string to be plotted, and a selection of the optional modifying graphical parameters cex=, col= and font=. Other objects will be coerced by as.graphicsAnnot.

The position of main defaults to being vertically centered in (outer) margin 3 and justified horizontally according to par("adj") on the plot region (device region for outer = TRUE).

The positions of xlab, ylab and sub are line (default for xlab and ylab being par("mgp")[1] and increased by 1 for sub) lines (of height par("mex")) into the appropriate margin, justified in the text direction according to par("adj") on the plot/device region.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

mtext, text; plotmath for details on mathematical annotation.

Examples

```
plot(cars, main = "") # here, could use main directly
title(main = "Stopping Distance versus Speed")

plot(cars, main = "")
title(main = list("Stopping Distance versus Speed", cex = 1.5,
                  col = "red", font = 3))

## Specifying "..." :
plot(1, col.axis = "sky blue", col.lab = "thistle")
title("Main Title", sub = "subtitle",
      cex.main = 2,   font.main= 4, col.main= "blue",
      cex.sub = 0.75, font.sub = 3, col.sub = "red")


x <- seq(-4, 4, length.out = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
```

```
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")
```

---

units *Graphical Units*

Description

xinch and yinch convert the specified number of inches given as their arguments into the correct units for plotting with graphics functions. Usually, this only makes sense when normal coordinates are used, i.e., *no* log scale (see the log argument to par).

xyinch does the same for a pair of numbers xy, simultaneously.

Usage

```
xinch(x = 1, warn.log = TRUE)
yinch(y = 1, warn.log = TRUE)
xyinch(xy = 1, warn.log = TRUE)
```

Arguments

x, y	numeric vector
xy	numeric of length 1 or 2.
warn.log	logical; if TRUE, a warning is printed in case of active log scale.

Examples

```
all(c(xinch(), yinch()) == xyinch()) # TRUE
xyinch()
xyinch #- to see that is really   delta{"usr"} / "pin"

## plot labels offset 0.12 inches to the right
## of plotted symbols in a plot
with(mtcars, {
    plot(mpg, disp, pch = 19, main = "Motor Trend Cars")
    text(mpg + xinch(0.12), disp, row.names(mtcars),
         adj = 0, cex = .7, col = "blue")
    })
```

xspline *Draw an X-spline*

Description

Draw an X-spline, a curve drawn relative to control points.

Usage

```
xspline(x, y = NULL, shape = 0, open = TRUE, repEnds = TRUE,
        draw = TRUE, border = par("fg"), col = NA, ...)
```

Arguments

x, y	vectors containing the coordinates of the vertices of the polygon. See xy.coords for alternatives.
shape	A numeric vector of values between -1 and 1, which control the shape of the spline relative to the control points.
open	A logical value indicating whether the spline is an open or a closed shape.
repEnds	For open X-splines, a logical value indicating whether the first and last control points should be replicated for drawing the curve. Ignored for closed X-splines.
draw	logical: should the X-spline be drawn? If false, a set of line segments to draw the curve is returned, and nothing is drawn.
border	the color to draw the curve. Use border = NA to omit borders.
col	the color for filling the shape. The default, NA, is to leave unfilled.
...	graphical parameters such as lty, xpd, lend, ljoin and lmitre can be given as arguments.

Details

An X-spline is a line drawn relative to control points. For each control point, the line may pass through (interpolate) the control point or it may only approach (approximate) the control point; the behaviour is determined by a shape parameter for each control point.

If the shape parameter is greater than zero, the spline approximates the control points (and is very similar to a cubic B-spline when the shape is 1). If the shape parameter is less than zero, the spline interpolates the control points (and is very similar to a Catmull-Rom spline when the shape is -1). If the shape parameter is 0, the spline forms a sharp corner at that control point.

For open X-splines, the start and end control points must have a shape of 0 (and non-zero values are silently converted to zero).

For open X-splines, by default the start and end control points are replicated before the curve is drawn. A curve is drawn between (interpolating or approximating) the second and third of each set of four control points, so this default behaviour ensures that the resulting curve starts at the first control point you have specified and ends at the last control point. The default behaviour can be turned off via the repEnds argument.

Value

If `draw = TRUE`, `NULL` otherwise a list with elements `x` and `y` which could be passed to `lines`, `polygon` and so on.

Invisible in both cases.

Note

Two-dimensional splines need to be created in an isotropic coordinate system. Device coordinates are used (with an anisotropy correction if needed.)

References

Blanc, C. and Schlick, C. (1995), *X-splines : A Spline Model Designed for the End User*, in *Proceedings of SIGGRAPH 95*, pp. 377–386. https://dept-info.labri.fr/~schlick/DOC/sig1.html

See Also

`polygon`.

`par` for how to specify colors.

Examples

```
## based on examples in ?grid.xspline

xsplineTest <- function(s, open = TRUE,
                        x = c(1,1,3,3)/4,
                        y = c(1,3,3,1)/4, ...) {
    plot(c(0,1), c(0,1), type = "n", axes = FALSE, xlab = "", ylab = "")
    points(x, y, pch = 19)
    xspline(x, y, s, open, ...)
    text(x+0.05*c(-1,-1,1,1), y+0.05*c(-1,1,1,-1), s)
}
op <- par(mfrow = c(3,3), mar = rep(0,4), oma = c(0,0,2,0))
xsplineTest(c(0, -1, -1, 0))
xsplineTest(c(0, -1,  0, 0))
xsplineTest(c(0, -1,  1, 0))
xsplineTest(c(0,  0, -1, 0))
xsplineTest(c(0,  0,  0, 0))
xsplineTest(c(0,  0,  1, 0))
xsplineTest(c(0,  1, -1, 0))
xsplineTest(c(0,  1,  0, 0))
xsplineTest(c(0,  1,  1, 0))
title("Open X-splines", outer = TRUE)

par(mfrow = c(3,3), mar = rep(0,4), oma = c(0,0,2,0))
xsplineTest(c(0, -1, -1, 0), FALSE, col = "grey80")
xsplineTest(c(0, -1,  0, 0), FALSE, col = "grey80")
xsplineTest(c(0, -1,  1, 0), FALSE, col = "grey80")
xsplineTest(c(0,  0, -1, 0), FALSE, col = "grey80")
xsplineTest(c(0,  0,  0, 0), FALSE, col = "grey80")
xsplineTest(c(0,  0,  1, 0), FALSE, col = "grey80")
```

```
xsplineTest(c(0,  1, -1, 0), FALSE, col = "grey80")
xsplineTest(c(0,  1,  0, 0), FALSE, col = "grey80")
xsplineTest(c(0,  1,  1, 0), FALSE, col = "grey80")
title("Closed X-splines", outer = TRUE)

par(op)

x <- sort(stats::rnorm(5))
y <- sort(stats::rnorm(5))
plot(x, y, pch = 19)
res <- xspline(x, y, 1, draw = FALSE)
lines(res)
## the end points may be very close together,
## so use last few for direction
nr <- length(res$x)
arrows(res$x[1], res$y[1], res$x[4], res$y[4], code = 1, length = 0.1)
arrows(res$x[nr-3], res$y[nr-3], res$x[nr], res$y[nr], code = 2, length = 0.1)
```

Chapter 6

The `grid` package

`grid-package` *The Grid Graphics Package*

Description

A rewrite of the graphics layout capabilities, plus some support for interaction.

Details

This package contains a graphics system which supplements S-style graphics (see the **graphics** package).

Further information is available in the following vignettes:

`grid`	Introduction to `grid` (`../doc/grid.pdf`)
`displaylist`	Display Lists in `grid` (`../doc/displaylist.pdf`)
`frame`	Frames and packing grobs (`../doc/frame.pdf`)
`grobs`	Working with `grid` grobs (`../doc/grobs.pdf`)
`interactive`	Editing `grid` Graphics (`../doc/interactive.pdf`)
`locndimn`	Locations versus Dimensions (`../doc/locndimn.pdf`)
`moveline`	Demonstrating move-to and line-to (`../doc/moveline.pdf`)
`nonfinite`	How `grid` responds to non-finite values (`../doc/nonfinite.pdf`)
`plotexample`	Writing `grid` Code (`../doc/plotexample.pdf`)
`rotated`	Rotated Viewports (`../doc/rotated.pdf`)
`saveload`	Persistent representations (`../doc/saveload.pdf`)
`sharing`	Modifying multiple grobs simultaneously (`../doc/sharing.pdf`)
`viewports`	Working with `grid` viewports (`../doc/viewports.pdf`)

For a complete list of functions with individual help pages, use `library(help="grid")`.

Author(s)

Paul Murrell <paul@stat.auckland.ac.nz>

Maintainer: R Core Team <R-core@r-project.org>

References

Murrell, P. (2005). *R Graphics*. Chapman & Hall/CRC Press.

---

absolute.size *Absolute Size of a Grob*

Description

This function converts a unit object into absolute units. Absolute units are unaffected, but non-absolute units are converted into "null" units.

Usage

```
absolute.size(unit)
```

Arguments

unit	An object of class "unit".

Details

Absolute units are things like "inches", "cm", and "lines". Non-absolute units are "npc" and "native".

This function is designed to be used in widthDetails and heightDetails methods.

Value

An object of class "unit".

Author(s)

Paul Murrell

See Also

widthDetails and heightDetails methods.

arrow *Describe arrows to add to a line*

Description

Produces a description of what arrows to add to a line. The result can be passed to a function that draws a line, e.g., grid.lines.

Usage

```
arrow(angle = 30, length = unit(0.25, "inches"),
      ends = "last", type = "open")
```

Arguments

angle	The angle of the arrow head in degrees (smaller numbers produce narrower, pointier arrows). Essentially describes the width of the arrow head.
length	A unit specifying the length of the arrow head (from tip to base).
ends	One of "last", "first", or "both", indicating which ends of the line to draw arrow heads.
type	One of "open" or "closed" indicating whether the arrow head should be a closed triangle.

Examples

```
arrow()
str(arrow(type = "closed"), give.attr=FALSE)
```

as.mask *Define a Soft Mask*

Description

Define either an alpha mask or a luminance mask, based on a grob.

Usage

```
as.mask(x, type=c("alpha", "luminance"))
```

Arguments

x	A grob.
type	The type of mask.

Details

A mask may be specified for a viewport either directly as a grob or using this function. In the former case, the result is an alpha mask. This function allows the user to define a luminance mask instead.

Not all graphics devices support masks and those that do may only support one type of mask: for example `xfig` and `pictex` do not support masks and Cairo-based devices only support alpha masks.

Value

A `"GridMask"` object.

Author(s)

Paul Murrell

See Also

`viewport`

Examples

```
## NOTE: on devices without support for masks normal line segments
##       will be drawn
grid.newpage()
## Alpha mask
grid.segments(y0=1, y1=0, gp=gpar(col=2, lwd=100))
pushViewport(viewport(mask=circleGrob(gp=gpar(fill=rgb(0,0,0,.5)))))
grid.segments(gp=gpar(col=3, lwd=100))
grid.newpage()
## Luminance mask
grid.segments(y0=1, y1=0, gp=gpar(col=2, lwd=100))
pushViewport(viewport(mask=as.mask(circleGrob(gp=gpar(fill="grey50")),
                                   "luminance")))
grid.segments(gp=gpar(col=3, lwd=100))
```

---

calcStringMetric *Calculate Metric Information for Text*

Description

This function returns the ascent, descent, and width metric information for a character or expression vector.

Usage

```
calcStringMetric(text)
```

Arguments

`text`	A character or expression vector.

Value

A list with three numeric components named ascent, descent, and width. All values are in inches.

WARNING

The metric information from this function is based on the font settings that are in effect when this function is called. It will not necessarily correspond to the metric information of any text that is drawn on the page.

Author(s)

Paul Murrell

See Also

stringAscent, stringDescent, grobAscent, and grobDescent.

Examples

```
grid.newpage()
grid.segments(.01, .5, .99, .5, gp=gpar(col="grey"))
metrics <- calcStringMetric(letters)
grid.rect(x=1:26/27,
          width=unit(metrics$width, "inches"),
          height=unit(metrics$ascent, "inches"),
          just="bottom",
          gp=gpar(col="red"))
grid.rect(x=1:26/27,
          width=unit(metrics$width, "inches"),
          height=unit(metrics$descent, "inches"),
          just="top",
          gp=gpar(col="red"))
grid.text(letters, x=1:26/27, just="bottom")

test <- function(x) {
    grid.text(x, just="bottom")
    metric <- calcStringMetric(x)
    if (is.character(x)) {
        grid.rect(width=unit(metric$width, "inches"),
                  height=unit(metric$ascent, "inches"),
                  just="bottom",
                  gp=gpar(col=rgb(1,0,0,.5)))
        grid.rect(width=unit(metric$width, "inches"),
                  height=unit(metric$descent, "inches"),
                  just="top",
                  gp=gpar(col=rgb(1,0,0,.5)))
    } else {
        grid.rect(width=unit(metric$width, "inches"),
                  y=unit(.5, "npc") + unit(metric[2], "inches"),
                  height=unit(metric$ascent, "inches"),
                  just="bottom",
                  gp=gpar(col=rgb(1,0,0,.5)))
```

```
        grid.rect(width=unit(metric$width, "inches"),
                  height=unit(metric$descent, "inches"),
                  just="bottom",
                  gp=gpar(col=rgb(1,0,0,.5)))
    }
}

tests <- list("t",
              "test",
              "testy",
              "test\ntwo",
              expression(x),
              expression(y),
              expression(x + y),
              expression(a + b),
              expression(atop(x + y, 2)))

grid.newpage()
nrowcol <- n2mfrow(length(tests))
pushViewport(viewport(layout=grid.layout(nrowcol[1], nrowcol[2]),
                      gp=gpar(cex=5, lwd=.5)))
for (i in 1:length(tests)) {
    col <- (i - 1) %% nrowcol[2] + 1
    row <- (i - 1) %/% nrowcol[2] + 1
    pushViewport(viewport(layout.pos.row=row, layout.pos.col=col))
    test(tests[[i]])
    popViewport()
}
```

`dataViewport` *Create a Viewport with Scales based on Data*

Description

This is a convenience function for producing a viewport with x- and/or y-scales based on numeric values passed to the function.

Usage

```
dataViewport(xData = NULL, yData = NULL, xscale = NULL,
             yscale = NULL, extension = 0.05, ...)
```

Arguments

`xData`	A numeric vector of data.
`yData`	A numeric vector of data.
`xscale`	A numeric vector (length 2).
`yscale`	A numeric vector (length 2).

extension	A numeric. If length greater than 1, then first value is used to extend the xscale and second value is used to extend the yscale.
...	All other arguments will be passed to a call to the viewport() function.

Details

If xscale is not specified then the values in x are used to generate an x-scale based on the range of x, extended by the proportion specified in extension. Similarly for the y-scale.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

viewport and plotViewport.

---

depth *Determine the number of levels in an object*

Description

Determine the number of levels in a viewport stack or tree, in a viewport path, or in a grob path.

Usage

```
depth(x, ...)
## S3 method for class 'viewport'
depth(x, ...)
## S3 method for class 'path'
depth(x, ...)
```

Arguments

x	Typically a viewport or viewport stack or viewport tree or viewport list, or a viewport path, or a grob path.
...	Arguments used by other methods.

Details

Depths of paths are pretty straightforward because they contain no branchings. The depth of a viewport stack is the sum of the depths of the components of the stack. The depth of a viewport tree is the depth of the parent plus the depth of the children. The depth of a viewport list is the depth of the last component of the list.

Value

An integer value.

See Also

viewport, vpPath, gPath.

Examples

```
vp <- viewport()
depth(vp)
depth(vpStack(vp, vp))
depth(vpList(vpStack(vp, vp), vp))
depth(vpPath("vp"))
depth(vpPath("vp1", "vp2"))
```

---

deviceLoc *Convert Viewport Location to Device Location*

Description

These functions take a pair of unit objects and convert them to a pair of device locations (or dimensions) in inches (or native device coordinates).

Usage

```
deviceLoc(x, y, valueOnly = FALSE, device = FALSE)
deviceDim(w, h, valueOnly = FALSE, device = FALSE)
```

Arguments

x, y, w, h	A unit object.
valueOnly	A logical indicating. If TRUE then the function does not return a unit object, but rather only the converted numeric values.
device	A logical indicating whether the returned values should be in inches or native device units.

Details

These functions differ from the functions like convertX() because they convert from the coordinate systems within a viewport to inches on the device (i.e., from one viewport to another) and because they only deal with pairs of values (locations or dimensions).

The functions like convertX() convert between different units within the same viewport and convert along a single dimension.

Value

A list with two components, both of which are unit object in inches (unless `valueOnly` is `TRUE` in which case both components are numeric).

Warning

The conversion is only valid for the current device size. If the device is resized then at least some conversions will become invalid.

Furthermore, the returned value only makes sense with respect to the entire device (i.e., within the context of the root viewport).

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
## A tautology
grid.newpage()
pushViewport(viewport())
deviceLoc(unit(1, "inches"), unit(1, "inches"))

## Something less obvious
grid.newpage()
pushViewport(viewport(width=.5, height=.5))
grid.rect()
x <- unit(1, "in")
y <- unit(1, "in")
grid.circle(x, y, r=unit(2, "mm"))
loc <- deviceLoc(x, y)
loc
upViewport()
grid.circle(loc$x, loc$y, r=unit(1, "mm"), gp=gpar(fill="black"))

## Something even less obvious
grid.newpage()
pushViewport(viewport(width=.5, height=.5, angle=30))
grid.rect()
x <- unit(.2, "npc")
y <- unit(2, "in")
grid.circle(x, y, r=unit(2, "mm"))
loc <- deviceLoc(x, y)
loc
upViewport()
grid.circle(loc$x, loc$y, r=unit(1, "mm"), gp=gpar(fill="black"))
```

drawDetails *Customising grid Drawing*

Description

These generic hook functions are called whenever a grid grob is drawn. They provide an opportunity for customising the drawing of a new class derived from grob (or gTree).

Usage

```
drawDetails(x, recording)
preDrawDetails(x)
postDrawDetails(x)
```

Arguments

x	A grid grob.
recording	A logical value indicating whether a grob is being added to the display list or redrawn from the display list.

Details

NOTE: these functions have been largely superceded by the makeContent and makeContext functions, though they are still run and may still be useful in some contexts.

These functions are called by the grid.draw methods for grobs and gTrees.

preDrawDetails is called first during the drawing of a grob. This is where any additional viewports should be pushed. Note that the default behaviour for grobs is to push any viewports in the vp slot, and for gTrees is to also push and up any viewports in the childrenvp slot so there is typically nothing to do here.

drawDetails is called next and is where any additional calculations and graphical output should occur. Note that the default behaviour for gTrees is to draw all grobs in the children slot so there is typically nothing to do here.

postDrawDetails is called last and should reverse anything done in preDrawDetails (i.e., pop or up any viewports that were pushed). Note that the default behaviour for grobs is to pop any viewports that were pushed so there is typically nothing to do here.

Note that preDrawDetails and postDrawDetails are also called in the calculation of "grobwidth" and "grobheight" units.

Value

None of these functions are expected to return a value.

Author(s)

Paul Murrell

References

"Changes to grid for R 3.0.0", Paul Murrell, *The R Journal* (2013) 5:2, pages 148-160.

See Also

`grid.draw` and `makeContent`

`editDetails` *Customising grid Editing*

Description

This generic hook function is called whenever a grid grob is edited via `grid.edit` or `editGrob`. This provides an opportunity for customising the editing of a new class derived from grob (or gTree).

Usage

```
editDetails(x, specs)
```

Arguments

`x`	A grid grob.
`specs`	A list of named elements. The names indicate the grob slots to modify and the values are the new values for the slots.

Details

This function is called by `grid.edit` and `editGrob`. A method should be written for classes derived from grob or gTree if a change in a slot has an effect on other slots in the grob or children of a gTree (e.g., see `grid:::editDetails.xaxis`).

Note that the slot already has the new value.

Value

The function MUST return the modified grob.

Author(s)

Paul Murrell

See Also

`grid.edit`

editViewport *Modify a Viewport*

Description

This is a convenience function for producing a new viewport from a copy of an existing viewport (by default the current viewport), with specified modifications.

Usage

```
editViewport(vp=current.viewport(), ...)
```

Arguments

vp	A viewport object.
...	Modification of the viewport (should all be valid arguments to the `viewport()` function.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

`viewport`.

explode *Explode a path into its components*

Description

Explode a viewport path or grob path into its components.

Usage

```
explode(x)
## S3 method for class 'character'
explode(x)
## S3 method for class 'path'
explode(x)
```

Arguments

x	Typically a viewport path or a grob path, but a character vector containing zero or more path separators may also be given.

Value

A character vector.

See Also

vpPath, gPath.

Examples

```
explode("vp1::vp2")
explode(vpPath("vp1", "vp2"))
```

---

gEdit *Create and Apply Edit Objects*

Description

The functions gEdit and gEditList create objects representing an edit operation (essentially a list of arguments to editGrob).

The functions applyEdit and applyEdits apply one or more edit operations to a graphical object.

These functions are most useful for developers creating new graphical functions and objects.

Usage

```
gEdit(...)
gEditList(...)
applyEdit(x, edit)
applyEdits(x, edits)
```

Arguments

...	one or more arguments to the editGrob function (for gEdit) or one or more "gEdit" objects (for gEditList).
x	a grob (grid graphical object).
edit	a "gEdit" object.
edits	either a "gEdit" object or a "gEditList" object.

Value

gEdit returns an object of class "gEdit".

gEditList returns an object of class "gEditList".

applyEdit and applyEditList return the modified grob.

Author(s)

Paul Murrell

See Also

grob editGrob

Examples

```
grid.rect(gp=gpar(col="red"))
# same thing, but more verbose
grid.draw(applyEdit(rectGrob(), gEdit(gp=gpar(col="red"))))
```

---

getNames *List the names of grobs on the display list*

Description

Returns a character vector containing the names of all top-level grobs on the display list.

Usage

```
getNames()
```

Value

A character vector.

Author(s)

Paul Murrell

Examples

```
grid.grill()
getNames()
```

gpar *Handling Grid Graphical Parameters*

Description

gpar() should be used to create a set of graphical parameter settings. It returns an object of class "gpar". This is basically a list of name-value pairs.

get.gpar() can be used to query the current graphical parameter settings.

Usage

```
gpar(...)
get.gpar(names = NULL)
```

Arguments

...	Any number of named arguments.
names	A character vector of valid graphical parameter names.

Details

All grid viewports and (predefined) graphical objects have a slot called gp, which contains a "gpar" object. When a viewport is pushed onto the viewport stack and when a graphical object is drawn, the settings in the "gpar" object are enforced. In this way, the graphical output is modified by the gp settings until the graphical object has finished drawing, or until the viewport is popped off the viewport stack, or until some other viewport or graphical object is pushed or begins drawing.

The default parameter settings are defined by the ROOT viewport, which takes its settings from the graphics device. These defaults may differ between devices (e.g., the default fill setting is different for a PNG device compared to a PDF device).

Valid parameter names are:

col	Colour for lines and borders.
fill	Colour for filling rectangles, polygons, ...
alpha	Alpha channel for transparency
lty	Line type
lwd	Line width
lex	Multiplier applied to line width
lineend	Line end style (round, butt, square)
linejoin	Line join style (round, mitre, bevel)
linemitre	Line mitre limit (number greater than 1)
fontsize	The size of text (in points)
cex	Multiplier applied to fontsize
fontfamily	The font family
fontface	The font face (bold, italic, ...)
lineheight	The height of a line as a multiple of the size of text
font	Font face (alias for fontface; for backward compatibility)

For more details of many of these, see the help for the corresponding graphical parameter par in base graphics. (This may have a slightly different name, e.g. lend, ljoin, lmitre, family.)

Colours can be specified in one of the forms returned by rgb, as a name (see colors) or as a non-negative integer index into the current palette (with zero being taken as transparent). (Negative integer values are now an error.)

The alpha setting is combined with the alpha channel for individual colours by multiplying (with both alpha settings normalised to the range 0 to 1).

The fill setting can also be a linear gradient or a radial gradient or a pattern (see patterns).

The size of text is fontsize*cex. The size of a line is fontsize*cex*lineheight.

The cex setting is cumulative; if a viewport is pushed with a cex of 0.5 then another viewport is pushed with a cex of 0.5, the effective cex is 0.25.

The alpha and lex settings are also cumulative.

Changes to the fontfamily may be ignored by some devices, but is supported by PostScript, PDF, X11, Windows, and Quartz. The fontfamily may be used to specify one of the Hershey Font families (e.g., HersheySerif) and this specification will be honoured on all devices.

The specification of fontface can be an integer or a string. If an integer, then it follows the R base graphics standard: 1 = plain, 2 = bold, 3 = italic, 4 = bold italic. If a string, then valid values are: "plain", "bold", "italic", "oblique", and "bold.italic". For the special case of the HersheySerif font family, "cyrillic", "cyrillic.oblique", and "EUC" are also available.

All parameter values can be vectors of multiple values. (This will not always make sense – for example, viewports will only take notice of the first parameter value.)

get.gpar() returns all current graphical parameter settings.

Value

An object of class "gpar".

Author(s)

Paul Murrell

See Also

Hershey.

Examples

```
gp <- get.gpar()
utils::str(gp)
## These *do* nothing but produce a "gpar" object:
gpar(col = "red")
gpar(col = "blue", lty = "solid", lwd = 3, fontsize = 16)
get.gpar(c("col", "lty"))
grid.newpage()
```

```
vp <- viewport(width = .8, height = .8, gp = gpar(col="blue"))
grid.draw(gTree(children=gList(rectGrob(gp = gpar(col="red")),
                     textGrob(paste("The rect is its own colour (red)",
                                    "but this text is the colour",
                                    "set by the gTree (green)",
                                    sep = "\n"))),
      gp = gpar(col="green"), vp = vp))
grid.text("This text is the colour set by the viewport (blue)",
          y = 1, just = c("center", "bottom"),
          gp = gpar(fontsize=20), vp = vp)
grid.newpage()
## example with multiple values for a parameter
pushViewport(viewport())
grid.points(1:10/11, 1:10/11, gp = gpar(col=1:10))
popViewport()
```

---

gPath *Concatenate Grob Names*

Description

This function can be used to generate a grob path for use in `grid.edit` and friends.

A grob path is a list of nested grob names.

Usage

```
gPath(...)
```

Arguments

...	Character values which are grob names.

Details

Grob names must only be unique amongst grobs which share the same parent in a gTree.

This function can be used to generate a specification for a grob that includes the grob's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A `gPath` object.

See Also

`grob`, `editGrob`, `addGrob`, `removeGrob`, `getGrob`, `setGrob`

Examples

```
gPath("g1", "g2")
```

Grid *Grid Graphics*

Description

General information about the grid graphics package.

Details

Grid graphics provides an alternative to the standard R graphics. The user is able to define arbitrary rectangular regions (called *viewports*) on the graphics device and define a number of coordinate systems for each region. Drawing can be specified to occur in any viewport using any of the available coordinate systems.

Grid graphics and standard R graphics do not mix!

Type `library(help = grid)` to see a list of (public) Grid graphics functions.

Author(s)

Paul Murrell

See Also

`viewport`, `grid.layout`, and `unit`.

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                    heights=unit(rep(1, 4),
                                 c("lines", "lines", "lines", "null")),
                    widths=unit(c(1, 1), "inches")))
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            width=unit(1, "inches"), height=unit(1, "inches")))
## A flash plotting example
grid.multipanel(vp=viewport(0.5, 0.5, 0.8, 0.8))
```

`Grid Viewports` *Create a Grid Viewport*

Description

These functions create viewports, which describe rectangular regions on a graphics device and define a number of coordinate systems within those regions.

Usage

```
viewport(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         default.units = "npc", just = "centre",
         gp = gpar(), clip = "inherit", mask = "inherit",
         xscale = c(0, 1), yscale = c(0, 1),
         angle = 0,
         layout = NULL,
         layout.pos.row = NULL, layout.pos.col = NULL,
         name = NULL)
vpList(...)
vpStack(...)
vpTree(parent, children)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.
`just`	A string or numeric vector specifying the justification of the viewport relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`clip`	One of `"on"`, `"inherit"`, or `"off"`, indicating whether to clip to the extent of this viewport, inherit the clipping region from the parent viewport, or turn clipping off altogether. For back-compatibility, a logical value of `TRUE` corresponds to `"on"` and `FALSE` corresponds to `"inherit"`. May also be a grob (or a gTree) that describes a clipping path or the result of a call to `as.path`.

mask	One of "none" (or FALSE) or "inherit" (or TRUE) or a grob (or a gTree) or the result of call to as.mask. This specifies that the viewport should have no mask, or it should inherit the mask of its parent, or it should have its own mask, as described by the grob.
xscale	A numeric vector of length two indicating the minimum and maximum on the x-scale. The limits may not be identical.
yscale	A numeric vector of length two indicating the minimum and maximum on the y-scale. The limits may not be identical.
angle	A numeric value indicating the angle of rotation of the viewport. Positive values indicate the amount of rotation, in degrees, anticlockwise from the positive x-axis.
layout	A Grid layout object which splits the viewport into subregions.
layout.pos.row	A numeric vector giving the rows occupied by this viewport in its parent's layout.
layout.pos.col	A numeric vector giving the columns occupied by this viewport in its parent's layout.
name	A character value to uniquely identify the viewport once it has been pushed onto the viewport tree.
...	Any number of grid viewport objects.
parent	A grid viewport object.
children	A vpList object.

Details

The location and size of a viewport are relative to the coordinate systems defined by the viewport's parent (either a graphical device or another viewport). The location and size can be specified in a very flexible way by specifying them with unit objects. When specifying the location of a viewport, specifying both layout.pos.row and layout.pos.col as NULL indicates that the viewport ignores its parent's layout and specifies its own location and size (via its locn). If only one of layout.pos.row and layout.pos.col is NULL, this means occupy ALL of the appropriate row(s)/column(s). For example, layout.pos.row = 1 and layout.pos.col = NULL means occupy all of row 1. Specifying non-NULL values for both layout.pos.row and layout.pos.col means occupy the intersection of the appropriate rows and columns. If a vector of length two is specified for layout.pos.row or layout.pos.col, this indicates a range of rows or columns to occupy. For example, layout.pos.row = c(1, 3) and layout.pos.col = c(2, 4) means occupy cells in the intersection of rows 1, 2, and 3, and columns, 2, 3, and 4.

Clipping obeys only the most recent viewport clip setting. For example, if you clip to viewport1, then clip to viewport2, the clipping region is determined wholly by viewport2, the size and shape of viewport1 is irrelevant (until viewport2 is popped of course).

If a viewport is rotated (because of its own angle setting or because it is within another viewport which is rotated) then the clip flag is ignored.

If clip is a grob, then that grob (which may be more than one shape) defines a clipping path. The function as.path may be used to specify a fill rule for the path.

Viewport names need not be unique. When pushed, viewports sharing the same parent must have unique names, which means that if you push a viewport with the same name as an existing viewport,

the existing viewport will be replaced in the viewport tree. A viewport name can be any string, but grid uses the reserved name `"ROOT"` for the top-level viewport. Also, when specifying a viewport name in `downViewport` and `seekViewport`, it is possible to provide a viewport path, which consists of several names concatenated using the separator (currently `::`). Consequently, it is not advisable to use this separator in viewport names.

The viewports in a `vpList` are pushed in parallel. The viewports in a `vpStack` are pushed in series. When a `vpTree` is pushed, the parent is pushed first, then the children are pushed in parallel.

Value

An R object of class `viewport`.

Author(s)

Paul Murrell

See Also

Grid, pushViewport, popViewport, downViewport, seekViewport, upViewport, unit, grid.layout, grid.show.layout.

Examples

```
# Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                   width=unit(1, "inches"), height=unit(1, "inches")))
# Demonstrate viewport clipping
clip.demo <- function(i, j, clip1, clip2) {
  pushViewport(viewport(layout.pos.col=i,
                        layout.pos.row=j))
  pushViewport(viewport(width=0.6, height=0.6, clip=clip1))
  grid.rect(gp=gpar(fill="white"))
  grid.circle(r=0.55, gp=gpar(col="red", fill="pink"))
  popViewport()
  pushViewport(viewport(width=0.6, height=0.6, clip=clip2))
  grid.polygon(x=c(0.5, 1.1, 0.6, 1.1, 0.5, -0.1, 0.4, -0.1),
               y=c(0.6, 1.1, 0.5, -0.1, 0.4, -0.1, 0.5, 1.1),
               gp=gpar(col="blue", fill="light blue"))
  popViewport(2)
}

grid.newpage()
grid.rect(gp=gpar(fill="grey"))
pushViewport(viewport(layout=grid.layout(2, 2)))
clip.demo(1, 1, FALSE, FALSE)
clip.demo(1, 2, TRUE, FALSE)
clip.demo(2, 1, FALSE, TRUE)
clip.demo(2, 2, TRUE, TRUE)
popViewport()
# Demonstrate turning clipping off
grid.newpage()
```

```
pushViewport(viewport(width=.5, height=.5, clip="on"))
grid.rect()
grid.circle(r=.6, gp=gpar(lwd=10))
pushViewport(viewport(clip="inherit"))
grid.circle(r=.6, gp=gpar(lwd=5, col="grey"))
pushViewport(viewport(clip="off"))
grid.circle(r=.6)
popViewport(3)
# Demonstrate vpList, vpStack, and vpTree
grid.newpage()
tree <- vpTree(viewport(width=0.8, height=0.8, name="A"),
               vpList(vpStack(viewport(x=0.1, y=0.1, width=0.5, height=0.5,
                                       just=c("left", "bottom"), name="B"),
                              viewport(x=0.1, y=0.1, width=0.5, height=0.5,
                                       just=c("left", "bottom"), name="C"),
                              viewport(x=0.1, y=0.1, width=0.5, height=0.5,
                                       just=c("left", "bottom"), name="D")),
                      viewport(x=0.5, width=0.4, height=0.9,
                               just="left", name="E")))
pushViewport(tree)
for (i in LETTERS[1:5]) {
  seekViewport(i)
  grid.rect()
  grid.text(current.vpTree(FALSE),
            x=unit(1, "mm"), y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"),
            gp=gpar(fontsize=8))
}
```

---

grid.add *Add a Grid Graphical Object*

Description

Add a grob to a gTree or a descendant of a gTree.

Usage

```
grid.add(gPath, child, strict = FALSE, grep = FALSE,
         global = FALSE, allDevices = FALSE, redraw = TRUE)

addGrob(gTree, child, gPath = NULL, strict = FALSE, grep = FALSE,
        global = FALSE, warn = TRUE)

setChildren(x, children)
```

Arguments

gTree, x	A gTree object.

gPath	A gPath object. For grid.add this specifies a gTree on the display list. For addGrob this specifies a descendant of the specified gTree.
child	A grob object.
children	A gList object.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
warn	A logical to indicate whether failing to find the specified gPath should trigger an error.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
redraw	A logical value to indicate whether to redraw the grob.

Details

addGrob copies the specified grob and returns a modified grob.

grid.add destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

setChildren is a basic function for setting all children of a gTree at once (instead of repeated calls to addGrob).

Value

addGrob returns a grob object; grid.add returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

grid.bezier *Draw a Bezier Curve*

Description

These functions create and draw Bezier Curves (a curve drawn relative to 4 control points).

Usage

```
grid.bezier(...)
bezierGrob(x = c(0, 0.5, 1, 0.5), y = c(0.5, 1, 0.5, 0),
           id = NULL, id.lengths = NULL,
           default.units = "npc", arrow = NULL,
           name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations of spline control points.
y	A numeric vector or unit object specifying y-locations of spline control points.
id	A numeric vector used to separate locations in x and y into multiple beziers. All locations with the same id belong to the same bezier.
id.lengths	A numeric vector used to separate locations in x and y into multiple bezier. Specifies consecutive blocks of locations which make up separate beziers.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
arrow	A list describing arrow heads to place at either end of the bezier, as produced by the arrow function.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to bezierGrob.

Details

Both functions create a beziergrob (a graphical object describing a Bezier curve), but only grid.bezier draws the Bezier curve.

A Bezier curve is a line drawn relative to 4 control points.

Missing values are not allowed for x and y (i.e., it is not valid for a control point to be missing).

The curve is currently drawn using an approximation based on X-splines.

Value

A grob object.

See Also

Grid, viewport, arrow.

grid.xspline.

Examples

```
x <- c(0.2, 0.2, 0.4, 0.4)
y <- c(0.2, 0.4, 0.4, 0.2)

grid.newpage()
grid.bezier(x, y)
grid.bezier(c(x, x + .4), c(y + .4, y + .4),
            id=rep(1:2, each=4))
grid.segments(.4, .6, .6, .6)
grid.bezier(x, y,
            gp=gpar(lwd=3, fill="black"),
            arrow=arrow(type="closed"),
            vp=viewport(x=.9))
```

---

grid.cap *Capture a raster image*

Description

Capture the current contents of a graphics device as a raster (bitmap) image.

Usage

```
grid.cap()
```

Details

This function is only implemented for on-screen graphics devices.

Value

A matrix of R colour names, or NULL if not available.

Author(s)

Paul Murrell

See Also

grid.raster

dev.capabilities to see if it is supported.

Examples

```
dev.new(width=0.5, height=0.5)
grid.rect()
grid.text("hi")
cap <- grid.cap()
dev.off()

if(!is.null(cap))
    grid.raster(cap, width=0.5, height=0.5, interpolate=FALSE)
```

---

grid.circle *Draw a Circle*

Description

Functions to create and draw a circle.

Usage

```
grid.circle(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), draw=TRUE, vp=NULL)
circleGrob(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
r	A numeric vector or unit object specifying radii.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a circle grob (a graphical object describing a circle), but only grid.circle() draws the circle (and then only if draw is TRUE).

The radius may be given in any units; if the units are *relative* (e.g., "npc" or "native") then the radius will be different depending on whether it is interpreted as a width or as a height. In such

cases, the smaller of these two values will be the result. To see the effect, type `grid.circle()` and adjust the size of the window.

What happens for very small radii is device-dependent: the circle may become invisible or be shown at a fixed minimum size. Circles of zero radius will not be plotted.

Value

A circle grob. `grid.circle()` returns the value invisibly.

Warning

Negative values for the radius are silently converted to their absolute value.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

`grid.clip` *Set the Clipping Region*

Description

These functions set the clipping region within the current viewport *without* altering the current coordinate system.

Usage

```
grid.clip(...)
clipGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         just = "centre", hjust = NULL, vjust = NULL,
         default.units = "npc", name = NULL, vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`just`	The justification of the clip rectangle relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.

hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
vp	A Grid viewport object (or NULL).
...	Arguments passed to clipGrob.

Details

Both functions create a clip rectangle (a graphical object describing a clip rectangle), but only grid.clip enforces the clipping.

Pushing or popping a viewport *always* overrides the clip region set by a clip grob, regardless of whether that viewport explicitly enforces a clipping region.

Value

clipGrob returns a clip grob.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
# draw across entire viewport, but clipped
grid.clip(x = 0.3, width = 0.1)
grid.lines(gp=gpar(col="green", lwd=5))
# draw across entire viewport, but clipped (in different place)
grid.clip(x = 0.7, width = 0.1)
grid.lines(gp=gpar(col="red", lwd=5))
# Viewport sets new clip region
pushViewport(viewport(width=0.5, height=0.5, clip=TRUE))
grid.lines(gp=gpar(col="grey", lwd=3))
# Return to original viewport;  get
# clip region from previous grid.clip()
# (NOT from previous viewport clip region)
popViewport()
grid.lines(gp=gpar(col="black"))
```

grid.convert *Convert Between Different grid Coordinate Systems*

Description

These functions take a unit object and convert it to an equivalent unit object in a different coordinate system.

Usage

```
convertX(x, unitTo, valueOnly = FALSE)
convertY(x, unitTo, valueOnly = FALSE)
convertWidth(x, unitTo, valueOnly = FALSE)
convertHeight(x, unitTo, valueOnly = FALSE)
convertUnit(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
```

Arguments

x	A unit object.
unitTo	The coordinate system to convert the unit to. See the unit function for valid coordinate systems.
axisFrom	Either "x" or "y" to indicate whether the unit object represents a value in the x- or y-direction.
typeFrom	Either "location" or "dimension" to indicate whether the unit object represents a location or a length.
axisTo	Same as axisFrom, but applies to the unit object that is to be created.
typeTo	Same as typeFrom, but applies to the unit object that is to be created.
valueOnly	A logical indicating. If TRUE then the function does not return a unit object, but rather only the converted numeric values.

Details

The convertUnit function allows for general-purpose conversions. The other four functions are just more convenient front-ends to it for the most common conversions.

The conversions occur within the current viewport.

It is not currently possible to convert to all valid coordinate systems (e.g., "strwidth" or "grobwidth"). I'm not sure if all of these are impossible, they just seem implausible at this stage.

In normal usage of grid, these functions should not be necessary. If you want to express a location or dimension in inches rather than user coordinates then you should simply do something like unit(1, "inches") rather than something like unit(0.134, "native").

In some cases, however, it is necessary for the user to perform calculations on a unit value and this function becomes necessary. In such cases, please take note of the warning below.

Value

A unit object in the specified coordinate system (unless `valueOnly` is `TRUE` in which case the returned value is a numeric).

Warning

The conversion is only valid for the current device size. If the device is resized then at least some conversions will become invalid. For example, suppose that I create a unit object as follows: `oneinch <- convertUnit(unit(1, "inches"), "native")`. Now if I resize the device, the unit object in `oneinch` no longer corresponds to a physical length of 1 inch.

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
## A tautology
convertX(unit(1, "inches"), "inches")
## The physical units
convertX(unit(2.54, "cm"), "inches")
convertX(unit(25.4, "mm"), "inches")
convertX(unit(72.27, "points"), "inches")
convertX(unit(1/12*72.27, "picas"), "inches")
convertX(unit(72, "bigpts"), "inches")
convertX(unit(1157/1238*72.27, "dida"), "inches")
convertX(unit(1/12*1157/1238*72.27, "cicero"), "inches")
convertX(unit(65536*72.27, "scaledpts"), "inches")
convertX(unit(1/2.54, "inches"), "cm")
convertX(unit(1/25.4, "inches"), "mm")
convertX(unit(1/72.27, "inches"), "points")
convertX(unit(1/(1/12*72.27), "inches"), "picas")
convertX(unit(1/72, "inches"), "bigpts")
convertX(unit(1/(1157/1238*72.27), "inches"), "dida")
convertX(unit(1/(1/12*1157/1238*72.27), "inches"), "cicero")
convertX(unit(1/(65536*72.27), "inches"), "scaledpts")

pushViewport(viewport(width=unit(1, "inches"),
                      height=unit(2, "inches"),
                      xscale=c(0, 1),
                      yscale=c(1, 3)))
  ## Location versus dimension
  convertY(unit(2, "native"), "inches")
  convertHeight(unit(2, "native"), "inches")
  ## From "x" to "y" (the conversion is via "inches")
  convertUnit(unit(1, "native"), "native",
              axisFrom="x", axisTo="y")
```

```
  ## Convert several values at once
  convertX(unit(c(0.5, 2.54), c("npc", "cm")),
               c("inches", "native"))
popViewport()
## Convert a complex unit
convertX(unit(1, "strwidth", "Hello"), "native")
```

---

grid.copy *Make a Copy of a Grid Graphical Object*

Description

This function is redundant and will disappear in future versions.

Usage

```
grid.copy(grob)
```

Arguments

grob	A grob object.

Value

A copy of the grob object.

Author(s)

Paul Murrell

See Also

grid.grob.

---

grid.curve *Draw a Curve Between Locations*

Description

These functions create and draw a curve from one location to another.

Usage

```
grid.curve(...)
curveGrob(x1, y1, x2, y2, default.units = "npc",
          curvature = 1, angle = 90, ncp = 1, shape = 0.5,
          square = TRUE, squareShape = 1,
          inflect = FALSE, arrow = NULL, open = TRUE,
          debug = FALSE,
          name = NULL, gp = gpar(), vp = NULL)
arcCurvature(theta)
```

Arguments

x1	A numeric vector or unit object specifying the x-location of the start point.
y1	A numeric vector or unit object specifying the y-location of the start point.
x2	A numeric vector or unit object specifying the x-location of the end point.
y2	A numeric vector or unit object specifying the y-location of the end point.
default.units	A string indicating the default units to use if x1, y1, x2 or y2 are only given as numeric values.
curvature	A numeric value giving the amount of curvature. Negative values produce left-hand curves, positive values produce right-hand curves, and zero produces a straight line.
angle	A numeric value between 0 and 180, giving an amount to skew the control points of the curve. Values less than 90 skew the curve towards the start point and values greater than 90 skew the curve towards the end point.
ncp	The number of control points used to draw the curve. More control points creates a smoother curve.
shape	A numeric vector of values between -1 and 1, which control the shape of the curve relative to its control points. See grid.xspline for more details.
square	A logical value that controls whether control points for the curve are created city-block fashion or obliquely. When ncp is 1 and angle is 90, this is typically TRUE, otherwise this should probably be set to FALSE (see Examples below).
squareShape	A shape value to control the behaviour of the curve relative to any additional control point that is inserted if square is TRUE.
inflect	A logical value specifying whether the curve should be cut in half and inverted (see Examples below).
arrow	A list describing arrow heads to place at either end of the curve, as produced by the arrow function.
open	A logical value indicating whether to close the curve (connect the start and end points).
debug	A logical value indicating whether debugging information should be drawn.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.

vp	A Grid viewport object (or NULL).
...	Arguments to be passed to curveGrob.
theta	An angle (in degrees).

Details

Both functions create a curve grob (a graphical object describing an curve), but only grid.curve draws the curve.

The arcCurvature function can be used to calculate a curvature such that control points are generated on an arc corresponding to angle theta. This is typically used in conjunction with a large ncp to produce a curve corresponding to the desired arc.

Value

A grob object.

See Also

Grid, viewport, grid.xspline, arrow

Examples

```
curveTest <- function(i, j, ...) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  do.call("grid.curve", c(list(x1=.25, y1=.25, x2=.75, y2=.75), list(...)))
  grid.text(sub("list\\((.*)\\)", "\\1",
                deparse(substitute(list(...)))),
            y=unit(1, "npc"))
  popViewport()
}
# grid.newpage()
pushViewport(plotViewport(c(0, 0, 1, 0),
                          layout=grid.layout(2, 1, heights=c(2, 1))))
pushViewport(viewport(layout.pos.row=1,
                      layout=grid.layout(3, 3, respect=TRUE)))
curveTest(1, 1)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport()
pushViewport(viewport(layout.pos.row=2,
                      layout=grid.layout(3, 3)))
curveTest(1, 1)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
```

```
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport(2)
```

`grid.delay` *Encapsulate calculations and generating a grob*

Description

Evaluates an expression that includes both calculations and generating a grob that depends on the calculations so that both the calculations and the grob generation will be rerun when the scene is redrawn (e.g., device resize or editing).

Intended *only* for expert use.

Usage

```
delayGrob(expr, list, name=NULL, gp=NULL, vp=NULL)
grid.delay(expr, list, name=NULL, gp=NULL, vp=NULL)
```

Arguments

`expr`	object of mode `expression` or `call` or an unevaluated expression.
`list`	a list defining the environment in which `expr` is to be evaluated.
`name`	A character identifier.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	A Grid viewport object (or NULL).

Details

A grob is created of special class `"delayedgrob"` (and drawn, in the case of `grid.delay`). The `makeContent` method for this class evaluates the expression with the list as the evaluation environment (and the grid Namespace as the parent of that environment).

The `expr` argument should return a grob as its result.

These functions are analogues of the `grid.record()` and `recordGrob()` functions; the difference is that these functions are based on the `makeContent()` hook, while those functions are based on the `drawDetails()` hook.

Note

This function *must* be used instead of the function `recordGraphics`; all of the dire warnings about using `recordGraphics` responsibly also apply here.

Author(s)

Paul Murrell

See Also

recordGraphics

Examples

```
grid.delay({
              w <- convertWidth(unit(1, "inches"), "npc")
              rectGrob(width=w)
            },
            list())
```

---

grid.display.list *Control the Grid Display List*

Description

Turn the Grid display list on or off.

Usage

```
grid.display.list(on=TRUE)
engine.display.list(on=TRUE)
```

Arguments

on	A logical value to indicate whether the display list should be on or off.

Details

All drawing and viewport-setting operations are (by default) recorded in the Grid display list. This allows redrawing to occur following an editing operation.

This display list could get very large so it may be useful to turn it off in some cases; this will of course disable redrawing.

All graphics output is also recorded on the main display list of the R graphics engine (by default). This supports redrawing following a device resize and allows copying between devices.

Turning off this display list means that grid will redraw from its own display list for device resizes and copies. This will be slower than using the graphics engine display list.

Value

None.

WARNING

Turning the display list on causes the display list to be erased!

Turning off both the grid display list and the graphics engine display list will result in no redrawing whatsoever.

Author(s)

Paul Murrell

`grid.DLapply` *Modify the Grid Display List*

Description

Call a function on each element of the current display list.

Usage

```
grid.DLapply(FUN, ...)
```

Arguments

`FUN`	A function; the first argument to this function is passed each element of the display list.
`...`	Further arguments to pass to `FUN` .

Details

This function is insanely dangerous (for the grid display list).

Two token efforts are made to try to avoid ending up with complete garbage on the display list:

1. The display list is only replaced once all new elements have been generated (so an error during generation does not result in a half-finished display list).
2. All new elements must be either `NULL` or inherit from the class of the element that they are replacing.

Value

The side effect of these functions is usually to modify the grid display list.

See Also

Grid.

Examples

```
grid.newpage()
grid.rect(width=.4, height=.4, x=.25, y=.75, gp=gpar(fill="black"), name="r1")
grid.rect(width=.4, height=.4, x=.5, y=.5, gp=gpar(fill="grey"), name="r2")
grid.rect(width=.4, height=.4, x=.75, y=.25, gp=gpar(fill="white"), name="r3")
grid.DLapply(function(x) { if (is.grob(x)) x$gp <- gpar(); x })
grid.refresh()
```

`grid.draw` *Draw a grid grob*

Description

Produces graphical output from a graphical object.

Usage

```
grid.draw(x, recording=TRUE)
```

Arguments

`x`	An object of class `"grob"` or NULL.
`recording`	A logical value to indicate whether the drawing operation should be recorded on the Grid display list.

Details

This is a generic function with methods for grob and gTree objects.

The grob and gTree methods automatically push any viewports in a `vp` slot and automatically apply any `gpar` settings in a `gp` slot. In addition, the gTree method pushes and ups any viewports in a `childrenvp` slot and automatically calls `grid.draw` for any grobs in a `children` slot.

The methods for grob and gTree call the generic hook functions `preDrawDetails`, `drawDetails`, and `postDrawDetails` to allow classes derived from grob or gTree to perform additional viewport pushing/popping and produce additional output beyond the default behaviour for grobs and gTrees.

Value

None.

Author(s)

Paul Murrell

See Also

`grob`.

Examples

```
grid.newpage()
## Create a graphical object, but don't draw it
l <- linesGrob()
## Draw it
grid.draw(l)
```

`grid.edit` *Edit the Description of a Grid Graphical Object*

Description

Changes the value of one of the slots of a grob and redraws the grob.

Usage

```
grid.edit(gPath, ..., strict = FALSE, grep = FALSE,
          global = FALSE, allDevices = FALSE, redraw = TRUE)

grid.gedit(..., grep = TRUE, global = TRUE)

editGrob(grob, gPath = NULL, ..., strict = FALSE, grep = FALSE,
         global = FALSE, warn = TRUE)
```

Arguments

`grob`	A grob object.
`...`	Zero or more named arguments specifying new slot values.
`gPath`	A gPath object. For `grid.edit` this specifies a grob on the display list. For `editGrob` this specifies a descendant of the specified grob.
`strict`	A boolean indicating whether the gPath must be matched exactly.
`grep`	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).
`global`	A boolean indicating whether the function should affect just the first match of the `gPath`, or whether all matches should be affected.
`warn`	A logical to indicate whether failing to find the specified gPath should trigger an error.
`allDevices`	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
`redraw`	A logical value to indicate whether to redraw the grob.

Details

editGrob copies the specified grob and returns a modified grob.

grid.edit destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

Both functions call editDetails to allow a grob to perform custom actions and validDetails to check that the modified grob is still coherent.

grid.gedit (g for global) is just a convenience wrapper for grid.edit with different defaults.

Value

editGrob returns a grob object; grid.edit returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

Examples

```
grid.newpage()
grid.xaxis(name = "xa", vp = viewport(width=.5, height=.5))
grid.edit("xa", gp = gpar(col="red"))
# won't work because no ticks (at is NULL)
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
grid.edit("xa", at = 1:4/5)
# Now it should work
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
```

---

grid.force *Force a grob into its components*

Description

Some grobs only generate their content to draw at drawing time; this function replaces such grobs with their at-drawing-time content.

Usage

```
grid.force(x, ...)
## Default S3 method:
grid.force(x, redraw = FALSE, ...)
## S3 method for class 'gPath'
grid.force(x, strict = FALSE, grep = FALSE, global = FALSE,
                            redraw = FALSE, ...)
```

```
## S3 method for class 'grob'
grid.force(x, draw = FALSE, ...)
forceGrob(x)
grid.revert(x, ...)
## S3 method for class 'gPath'
grid.revert(x, strict = FALSE, grep = FALSE, global = FALSE,
                           redraw = FALSE, ...)
## S3 method for class 'grob'
grid.revert(x, draw = FALSE, ...)
```

Arguments

x	For the default method, x should not be specified. Otherwise, x should be a grob or a gPath. If x is character, it is assumed to be a gPath.
strict	A boolean indicating whether the path must be matched exactly.
grep	Whether the path should be treated as a regular expression.
global	A boolean indicating whether the function should affect just the first match of the path, or whether all matches should be affected.
draw	logical value indicating whether a grob should be drawn after it is forced.
redraw	logical value indicating whether to redraw the **grid** scene after the forcing operation.
...	Further arguments for use by methods.

Details

Some grobs wait until drawing time to generate what content will actually be drawn (an axis, as produced by grid.xaxis(), with an at or NULL is a good example because it has to see what viewport it is going to be drawn in before it can decide what tick marks to draw).

The content of such grobs (e.g., the tick marks) are not usually visible to grid.ls() or accessible to grid.edit().

The grid.force() function *replaces* a grob with its at-drawing-time contents. For example, an axis will be replaced by a vanilla gTree with lines and text representing the axis tick marks that were actually drawn. This makes the tick marks visible to grid.ls() and accessible to grid.edit().

The forceGrob() function is the internal work horse for grid.force(), so will not normally be called directly by the user. It is exported so that methods can be written for custom grob classes if necessary.

The grid.revert() function reverses the effect of grid.force(), replacing forced content with the original grob.

Warning

Forcing an explicit grob produces a result as if the grob were drawn in the *current* drawing context. It may not make sense to draw the result in a different drawing context.

Note

These functions only have an effect for grobs that generate their content at drawing time using `makeContext()` and `makeContent()` methods (*not* for grobs that generate their content at drawing time using `preDrawDetails()` and `drawDetails()` methods).

Author(s)

Paul Murrell

Examples

```
grid.newpage()
pushViewport(viewport(width=.5, height=.5))
# Draw xaxis
grid.xaxis(name="xax")
grid.ls()
# Force xaxis
grid.force()
grid.ls()
# Revert xaxis
grid.revert()
grid.ls()
# Draw and force yaxis
grid.force(yaxisGrob(), draw=TRUE)
grid.ls()
# Revert yaxis
grid.revert()
grid.ls()
# Force JUST xaxis
grid.force("xax")
grid.ls()
# Force ALL
grid.force()
grid.ls()
# Revert JUST xaxis
grid.revert("xax")
grid.ls()
```

---

grid.frame *Create a Frame for Packing Objects*

Description

These functions, together with `grid.pack`, `grid.place`, `packGrob`, and `placeGrob` are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with this function then use `grid.pack` or whatever to pack/place objects into the frame.

Usage

```
grid.frame(layout=NULL, name=NULL, gp=gpar(), vp=NULL, draw=TRUE)
frameGrob(layout=NULL, name=NULL, gp=gpar(), vp=NULL)
```

Arguments

layout	A Grid layout, or NULL. This can be used to initialise the frame with a number of rows and columns, with initial widths and heights, etc.
name	A character identifier.
vp	An object of class viewport, or NULL.
gp	An object of class "gpar"; typically the output from a call to the function gpar.
draw	Should the frame be drawn.

Details

Both functions create a frame grob (a graphical object describing a frame), but only grid.frame() draws the frame (and then only if draw is TRUE). Nothing will actually be drawn, but it will put the frame on the display list, which means that the output will be dynamically updated as objects are packed into the frame. Possibly useful for debugging.

Value

A frame grob. grid.frame() returns the value invisibly.

Author(s)

Paul Murrell

See Also

grid.pack

Examples

```
grid.newpage()
grid.frame(name="gf", draw=TRUE)
grid.pack("gf", rectGrob(gp=gpar(fill="grey")), width=unit(1, "null"))
grid.pack("gf", textGrob("hi there"), side="right")
```

grid.function *Draw a curve representing a function*

Description

Draw a curve representing a function.

Usage

```
grid.function(...)
functionGrob(f, n = 101, range = "x", units = "native",
             name = NULL, gp=gpar(), vp = NULL)

grid.abline(intercept, slope, ...)
```

Arguments

f	A function that must take a single argument and return a list with two numeric components named x and y.
n	The number values that will be generated as input to the function f.
range	Either "x", "y", or a numeric vector. See the 'Details' section.
units	A string indicating the units to use for the x and y values generated by the function.
intercept	Numeric.
slope	Numeric.
...	Arguments passed to grid.function()
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

n values are generated and passed to the function f and a series of lines are drawn through the resulting x and y values.

The generation of the n values depends on the value of range. In the default case, dim is "x", which means that a set of x values are generated covering the range of the current viewport scale in the x-dimension. If dim is "y" then values are generated from the current y-scale instead. If range is a numeric vector, then values are generated from that range.

grid.abline() provides a simple front-end for a straight line parameterized by intercept and slope.

Value

A functiongrob grob.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
# abline
# NOTE: in ROOT viewport on screen, (0, 0) at top-left
#       and "native" is pixels!
grid.function(function(x) list(x=x, y=0 + 1*x))
# a more "normal" viewport with default normalized "native" coords
grid.newpage()
pushViewport(viewport())
grid.function(function(x) list(x=x, y=0 + 1*x))
# slightly simpler
grid.newpage()
pushViewport(viewport())
grid.abline()
# sine curve
grid.newpage()
pushViewport(viewport(xscale=c(0, 2*pi), yscale=c(-1, 1)))
grid.function(function(x) list(x=x, y=sin(x)))
# constrained sine curve
grid.newpage()
pushViewport(viewport(xscale=c(0, 2*pi), yscale=c(-1, 1)))
grid.function(function(x) list(x=x, y=sin(x)),
              range=0:1)
# inverse sine curve
grid.newpage()
pushViewport(viewport(xscale=c(-1, 1), yscale=c(0, 2*pi)))
grid.function(function(y) list(x=sin(y), y=y),
              range="y")
# parametric function
grid.newpage()
pushViewport(viewport(xscale=c(-1, 1), yscale=c(-1, 1)))
grid.function(function(t) list(x=cos(t), y=sin(t)),
              range=c(0, 9*pi/5))
# physical abline
grid.newpage()
grid.function(function(x) list(x=x, y=0 + 1*x),
              units="in")
```

---

grid.get *Get a Grid Graphical Object*

Description

Retrieve a grob or a descendant of a grob.

Usage

```
grid.get(gPath, strict = FALSE, grep = FALSE, global = FALSE,
         allDevices = FALSE)

grid.gget(..., grep = TRUE, global = TRUE)

getGrob(gTree, gPath, strict = FALSE, grep = FALSE, global = FALSE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.get this specifies a grob on the display list. For getGrob this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
...	Arguments that are passed to grid.get.

Details

grid.gget (g for global) is just a convenience wrapper for grid.get with different defaults.

Value

A grob object.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

Examples

```
grid.xaxis(name="xa")
grid.get("xa")
grid.get(gPath("xa", "ticks"))

grid.draw(gTree(name="gt", children=gList(xaxisGrob(name="axis"))))
grid.get(gPath("gt", "axis", "ticks"))
```

---

grid.glyph *Draw Typeset Glyphs*

Description

These functions create and draw a set of typeset glyphs.

Usage

```
grid.glyph(...)
glyphGrob(glyphInfo,
          x=.5, y=.5, default.units="npc",
          hjust="centre", vjust="centre",
          gp=gpar(), vp=NULL, name=NULL)
```

Arguments

glyphInfo	An "RGlyphInfo" object from a call to glyphInfo.
x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
hjust, vjust	The justification of the glyphs relative to the (x, y) location. Can be character (e.g., "left"), numeric (e.g., 0), or the result of a call to glyphJust.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments passed to glyphGrob().

Details

Both functions create a glyph grob (a graphical object describing glyphs), but only grid.glyph draws the glyphs.

Value

A glyph grob.

Author(s)

Paul Murrell

See Also

Grid, glyphInfo

`grid.grab` *Grab the current grid output*

Description

Creates a gTree object from the current grid display list or from a scene generated by user-specified code.

Usage

```
grid.grab(warn = 2, wrap = wrap.grobs, wrap.grobs = FALSE, ...)
grid.grabExpr(expr, warn = 2, wrap = wrap.grobs, wrap.grobs = FALSE,
              width = 7, height = 7, device = offscreen, ...)
```

Arguments

`expr`	An expression to be evaluated. Typically, some calls to grid drawing functions.
`warn`	An integer specifying the amount of warnings to emit. 0 means no warnings, 1 means warn when it is certain that the grab will not faithfully represent the original scene. 2 means warn if there's any possibility that the grab will not faithfully represent the original scene.
`wrap`	A logical indicating how the output should be captured. If `TRUE`, each non-grob element on the display list is captured by wrapping it in a grob.
`wrap.grobs`	A logical indicating whether, if we are wrapping elements (`wrap=TRUE`), we should wrap grobs (or just wrap viewports).
`width, height`	Size of the device used for temporary rendering.
`device`	A function that opens a graphics device for temporary rendering. By default this is an off-screen, in-memory device based on the `pdf` device, but this default device may not be satisfactory when using custom fonts.
`...`	arguments passed to gTree, for example, a name and/or class for the gTree that is created.

Details

There are four ways to capture grid output as a gTree.

There are two functions for capturing output: use `grid.grab` to capture an existing drawing and `grid.grabExpr` to capture the output from an expression (without drawing anything).

For each of these functions, the output can be captured in two ways. One way tries to be clever and make a gTree with a childrenvp slot containing all viewports on the display list (including those that are popped) and every grob on the display list as a child of the new gTree; each child has a vpPath in the vp slot so that it is drawn in the appropriate viewport. In other words, the gTree contains all elements on the display list, but in a slightly altered form.

The other way, `wrap=TRUE`, is to create a grob for every element on the display list (and make all of those grobs children of the gTree). Only viewports are wrapped unless `wrap.grobs` is also `TRUE`.

The first approach creates a more compact and elegant gTree, which is more flexible to work with, but is not guaranteed to faithfully replicate all possible grid output. The second approach is more brute force, and harder to work with, but is more likely to replicate the original output.

An example of a case that will NOT be replicated by wrapping, with `wrap.grobs=TRUE`, is a scene where the placement of one grob is dependent on another grob (e.g., via `grobX` or `grobWidth`).

Value

A gTree object.

See Also

gTree

Examples

```
pushViewport(viewport(width=.5, height=.5))
grid.rect()
grid.points(stats::runif(10), stats::runif(10))
popViewport()
grab <- grid.grab()
grid.newpage()
grid.draw(grab)
```

---

grid.grep *Search for Grobs and/or Viewports*

Description

Given a path, find all matching grobs and/or viewports on the display list or within a given grob.

Usage

```
grid.grep(path, x = NULL, grobs = TRUE, viewports = FALSE,
          strict = FALSE, grep = FALSE, global = FALSE,
          no.match = character(), vpPath = viewports)
```

Arguments

path	a gPath or a vpPath or a character value that could be interpreted as either.
x	a grob or NULL. If NULL, the display list is searched.
grobs	A logical value indicating whether to search for grobs.
viewports	A logical value indicating whether to search for viewports.
strict	A boolean indicating whether the path must be matched exactly.
grep	Whether the path should be treated as a regular expression.
global	A boolean indicating whether the function should affect just the first match of the path, or whether all matches should be affected.
no.match	The value to return if no matches are found.
vpPath	A logical value indicating whether to return the vpPath for each grob as an attribute of the result.

Value

Either a gPath or a vpPath or, if global is TRUE a list of gPaths and/or vpPaths.

If vpPath is TRUE, each gPath result will have an attribute "vpPath".

If there are no matches, no.match is returned.

See Also

grid.ls()

Examples

```
# A gTree, called "grandparent", with child gTree,
# called "parent", with childrenvp vpStack (vp2 within vp1)
# and child grob, called "child", with vp vpPath (down to vp2)
sampleGTree <- gTree(name="grandparent",
                     children=gList(gTree(name="parent",
                          children=gList(grob(name="child", vp="vp1::vp2")),
                          childrenvp=vpStack(viewport(name="vp1"),
                                             viewport(name="vp2")))))
# Searching for grobs
grid.grep("parent", sampleGTree)
grid.grep("parent", sampleGTree, strict=TRUE)
grid.grep("grandparent", sampleGTree, strict=TRUE)
grid.grep("grandparent::parent", sampleGTree)
grid.grep("parent::child", sampleGTree)
grid.grep("[a-z]", sampleGTree, grep=TRUE)
grid.grep("[a-z]", sampleGTree, grep=TRUE, global=TRUE)
# Searching for viewports
grid.grep("vp1", sampleGTree, viewports=TRUE)
grid.grep("vp2", sampleGTree, viewports=TRUE)
grid.grep("vp", sampleGTree, viewports=TRUE, grep=TRUE)
grid.grep("vp2", sampleGTree, viewports=TRUE, strict=TRUE)
grid.grep("vp1::vp2", sampleGTree, viewports=TRUE)
```

```
# Searching for both
grid.grep("[a-z]", sampleGTree, viewports=TRUE, grep=TRUE, global=TRUE)
```

`grid.grill` *Draw a Grill*

Description

This function draws a grill within a Grid viewport.

Usage

```
grid.grill(h = unit(seq(0.25, 0.75, 0.25), "npc"),
           v = unit(seq(0.25, 0.75, 0.25), "npc"),
           default.units = "npc", gp=gpar(col = "grey"), vp = NULL)
```

Arguments

`h`	A numeric vector or unit object indicating the horizontal location of the vertical grill lines.
`v`	A numeric vector or unit object indicating the vertical location of the horizontal grill lines.
`default.units`	A string indicating the default units to use if `h` or `v` are only given as numeric vectors.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	A Grid viewport object.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, `viewport`.

grid.grob *Create Grid Graphical Objects, aka "Grob"s*

Description

Creating grid graphical objects, short ("grob"s).

`grob()` and `gTree()` are the basic creators, `grobTree()` and `gList()` take several grobs to build a new one.

Usage

```
## Grob Creation:

grob (..., name = NULL, gp = NULL, vp = NULL, cl = NULL)
gTree(..., name = NULL, gp = NULL, vp = NULL, children = NULL,
      childrenvp = NULL, cl = NULL)
grobTree(..., name = NULL, gp = NULL, vp = NULL,
         childrenvp = NULL, cl = NULL)
gList(...)

## Grob Properties:
childNames(gTree)
is.grob(x)
```

Arguments

`...`	For `grob` and `gTree`, the named slots describing important features of the graphical object. For `gList` and `grobTree`, a series of grob objects.
`name`	a character identifier for the grob. Used to find the grob on the display list and/or as a child of another grob.
`children`	a `"gList"` object.
`childrenvp`	a `viewport` object (or `NULL`).
`gp`	A `"gpar"` object, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	a `viewport` object (or `NULL`).
`cl`	string giving the class attribute for the new class.
`gTree`	a `"gTree"` object.
`x`	An R object.

Details

These functions can be used to create a basic "grob", "gTree", or "gList" object, or a new class derived from one of these.

A grid graphical object ("grob") is a description of a graphical item. These basic classes provide default behaviour for validating, drawing, and modifying graphical objects. Both grob() and gTree() call the function validDetails to check that the object returned is internally coherent.

A "gTree" can have other grobs as children; when a gTree is drawn, it draws all of its children. Before drawing its children, a gTree pushes its childrenvp slot and then navigates back up (calls upViewport) so that the children can specify their location within the childrenvp via a vpPath.

Grob names need not be unique in general, but all children of a gTree must have different names. A grob name can be any string, though it is not advisable to use the gPath separator (currently ::) in grob names.

The function childNames returns the names of the grobs which are children of a gTree.

All grid primitives (grid.lines, grid.rect, ...) and some higher-level grid components (e.g., grid.xaxis and grid.yaxis) are derived from these classes.

grobTree is just a convenient wrapper for gTree when the only components of the gTree are grobs (so all unnamed arguments become children of the gTree).

The grid.grob function is defunct.

Value

An R object of class "grob", a **gr**aphical **ob**ject.

Author(s)

Paul Murrell

See Also

grid.draw, grid.edit, grid.get.

---

grid.group *Draw a Group*

Description

These functions define and draw one or more *groups*, where a group is a grob that is drawn in isolation before being combined with the main image. The concept of groups allows for compositing operators, object reuse, and affine transformations (see the Details section).

Usage

```
groupGrob(src, op = "over", dst = NULL, coords = TRUE,
          name = NULL, gp=gpar(), vp=NULL)
grid.group(src, op = "over", dst = NULL, coords = TRUE,
           name = NULL, gp=gpar(), vp=NULL)

defineGrob(src, op = "over", dst = NULL, coords = TRUE,
           name = NULL, gp=gpar(), vp=NULL)
grid.define(src, op = "over", dst = NULL, coords = TRUE,
            name = NULL, gp=gpar(), vp=NULL)

useGrob(group, transform=viewportTransform,
        name=NULL, gp=gpar(), vp=NULL)

grid.use(group, transform=viewportTransform,
         name=NULL, gp=gpar(), vp=NULL)
```

Arguments

src	A grob.
op	The name of a compositing operator (see Details).
dst	A grob.
coords	A logical indicating whether grob coordinates should be calculated for the group.
group	A character identified referring to the name of a defined group.
transform	A function that returns an affine transformation matrix; see viewportTransform.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

In the simplest usage, we can use grid.group() to specify a grob to be drawn in isolation before being combined with the main image. This can be different from normal drawing if, for example, the grob draws more than one shape and there is a mask currently in effect.

Another possible use of grid.group() is to specify both src and dst and combine them using a compositing operator other than the default "over", before combining the result with the main image. For example, if we use the "dest.out" operator then dst is only drawn where it is NOT overlapped by src. The following (extended) Porter-Duff operators are available: "clear", "source", "over", "in", "out", "atop", "dest", "dest.over", "dest.in", "dest.out", "dest.atop", "xor", "add", and "saturate". In addition, there are operators corresponding to PDF blend modes: "multiply", "screen", "overlay", "darken", "lighten", "color.dodge", "color.burn", "hard.light", "soft.light", "difference", and "exclusion". However,

even if a graphics device supports groups, it may not support all compositing operators; see dev.capabilities.

It is also possible to break the process into two steps by first using grid.define() to define a group and then grid.use() to draw the group. This allows for reuse of a group (define the group once and use it several times).

If a group is defined in one viewport and used in a different viewport, an implicit transformation is applied. This could be a simple transformation (if the viewports are in different locations, but are the same size), or it could be more complex if the viewports are also different sizes or at different orientations.

NOTE: transformations occur on the graphics device so affect all aspects of drawing. For example, text and line widths are transformed as well as locations.

See viewportTransform for more information about transformations and how to customise them.

Not all graphics devices support these functions: for example xfig and pictex do not. For devices that do provide support, that support may only be partial (e.g., the Cairo-based devices support more compositing operators than the pdf() device).

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid

Examples

```
## NOTE: on devices without support for groups (or masks or patterns),
##       there will only be two overlapping opaque circles
grid.newpage()
pat <- pattern(rasterGrob(matrix(c(.5, 1, 1, .5), nrow=2),
                          width=unit(1, "cm"),
                          height=unit(1, "cm"),
                          interpolate=FALSE),
               width=unit(1, "cm"), height=unit(1, "cm"),
               extend="repeat")
grid.rect(gp=gpar(col=NA, fill=pat))
masks <- dev.capabilities()$masks
if (is.character(masks) && "luminance" %in% masks) {
  mask <- as.mask(rectGrob(gp=gpar(col=NA, fill="grey50")), type="luminance")
} else {
  mask <- rectGrob(gp=gpar(col=NA, fill=rgb(0,0,0,.5)))
}
pushViewport(viewport(mask=mask))
pushViewport(viewport(y=.5, height=.5, just="bottom"))
grid.circle(1:2/3, r=.45, gp=gpar(fill=2:3))
```

```
popViewport()
pushViewport(viewport(y=0, height=.5, just="bottom"))
grid.group(circleGrob(1:2/3, r=.45, gp=gpar(fill=2:3)))
popViewport()
```

`grid.layout` *Create a Grid Layout*

Description

This function returns a Grid layout, which describes a subdivision of a rectangular region.

Usage

```
grid.layout(nrow = 1, ncol = 1,
        widths = unit(rep_len(1, ncol), "null"),
        heights = unit(rep_len(1, nrow), "null"),
        default.units = "null", respect = FALSE,
        just="centre")
```

Arguments

`nrow`	An integer describing the number of rows in the layout.
`ncol`	An integer describing the number of columns in the layout.
`widths`	A numeric vector or unit object describing the widths of the columns in the layout.
`heights`	A numeric vector or unit object describing the heights of the rows in the layout.
`default.units`	A string indicating the default units to use if `widths` or `heights` are only given as numeric vectors.
`respect`	A logical value or a numeric matrix. If a logical, this indicates whether row heights and column widths should respect each other. If a matrix, non-zero values indicate that the corresponding row and column should be respected (see examples below).
`just`	A string or numeric vector specifying how the layout should be justified if it is not the same size as its parent viewport. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment. NOTE that in this context, `"left"`, for example, means align the left edge of the left-most layout column with the left edge of the parent viewport.

Details

The unit objects given for the `widths` and `heights` of a layout may use a special `units` that only has meaning for layouts. This is the `"null"` unit, which indicates what relative fraction of the available width/height the column/row occupies. See the reference for a better description of relative widths and heights in layouts.

Value

A Grid layout object.

WARNING

This function must NOT be confused with the base R graphics function `layout`. In particular, do not use `layout` in combination with Grid graphics. The documentation for `layout` may provide some useful information and this function should behave identically in comparable situations. The `grid.layout` function has *added* the ability to specify a broader range of units for row heights and column widths, and allows for nested layouts (see `viewport`).

Author(s)

Paul Murrell

References

Murrell, P. R. (1999). Layouts: A Mechanism for Arranging Plots on a Page. *Journal of Computational and Graphical Statistics*, **8**, 121–134. doi:10.2307/1390924.

See Also

Grid, `grid.show.layout`, `viewport`, `layout`

Examples

```
## A variety of layouts (some a bit mid-bending ...)
layout.torture()
## Demonstration of layout justification
grid.newpage()
testlay <- function(just="centre") {
  pushViewport(viewport(layout=grid.layout(1, 1, widths=unit(1, "inches"),
                          heights=unit(0.25, "npc"),
                          just=just)))
  pushViewport(viewport(layout.pos.col=1, layout.pos.row=1))
  grid.rect()
  grid.text(paste(just, collapse="-"))
  popViewport(2)
}
testlay()
testlay(c("left", "top"))
testlay(c("right", "top"))
testlay(c("right", "bottom"))
testlay(c("left", "bottom"))
testlay(c("left"))
testlay(c("right"))
testlay(c("bottom"))
testlay(c("top"))
```

grid.lines *Draw Lines in a Grid Viewport*

Description

These functions create and draw a series of lines.

Usage

```
grid.lines(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), draw = TRUE, vp = NULL)
linesGrob(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), vp = NULL)
grid.polyline(...)
polylineGrob(x = unit(c(0, 1), "npc"),
             y = unit(c(0, 1), "npc"),
             id=NULL, id.lengths=NULL,
             default.units = "npc",
             arrow = NULL, name = NULL,
             gp=gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
arrow	A list describing arrow heads to place at either end of the line, as produced by the arrow function.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).
id	A numeric vector used to separate locations in x and y into multiple lines. All locations with the same id belong to the same line.
id.lengths	A numeric vector used to separate locations in x and y into multiple lines. Specifies consecutive blocks of locations which make up separate lines.
...	Arguments passed to polylineGrob.

Details

The first two functions create a lines grob (a graphical object describing lines), and `grid.lines` draws the lines (if `draw` is `TRUE`).

The second two functions create or draw a polyline grob, which is just like a lines grob, except that there can be multiple distinct lines drawn.

Value

A lines grob or a polyline grob. `grid.lines` returns a lines grob invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `arrow`

Examples

```
grid.lines()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polyline(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
              y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
              id=rep(1:5, 4),
              gp=gpar(col=1:5, lwd=3))
# Using id.lengths
grid.newpage()
grid.polyline(x=outer(c(0, .5, 1, .5), 5:1/5),
              y=outer(c(.5, 1, .5, 0), 5:1/5),
              id.lengths=rep(4, 5),
              gp=gpar(col=1:5, lwd=3))
```

---

grid.locator *Capture a Mouse Click*

Description

Allows the user to click the mouse once within the current graphics device and returns the location of the mouse click within the current viewport, in the specified coordinate system.

Usage

```
grid.locator(unit = "native")
```

Arguments

unit	The coordinate system in which to return the location of the mouse click. See the unit function for valid coordinate systems.

Details

This function is modal (like the graphics package function locator) so the command line and graphics drawing is blocked until the use has clicked the mouse in the current device.

Value

A unit object representing the location of the mouse click within the current viewport, in the specified coordinate system.

If the user did not click mouse button 1, the function (invisibly) returns NULL.

Author(s)

Paul Murrell

See Also

viewport, unit, locator in package **graphics**, and for an application see trellis.focus and panel.identify in package **lattice**.

Examples

```
if (dev.interactive()) {
  ## Need to write a more sophisticated unit as.character method
  unittrim <- function(unit) {
    sub("^([0-9]+|[0-9]+[.][0-9])[0-9]*", "\\1", as.character(unit))
  }
  do.click <- function(unit) {
    click.locn <- grid.locator(unit)
    grid.segments(unit.c(click.locn$x, unit(0, "npc")),
                  unit.c(unit(0, "npc"), click.locn$y),
                  click.locn$x, click.locn$y,
                  gp=gpar(lty="dashed", col="grey"))
    grid.points(click.locn$x, click.locn$y, pch=16, size=unit(1, "mm"))
    clickx <- unittrim(click.locn$x)
    clicky <- unittrim(click.locn$y)
    grid.text(paste0("(", clickx, ", ", clicky, ")"),
              click.locn$x + unit(2, "mm"), click.locn$y,
              just="left")
  }

  grid.newpage() # (empty slate)
  ## device
  do.click("inches")
  Sys.sleep(1)

  pushViewport(viewport(width=0.5, height=0.5,
```

```
                        xscale=c(0, 100), yscale=c(0, 10)))
  grid.rect()
  grid.xaxis()
  grid.yaxis()
  do.click("native")
  popViewport()
}
```

---

grid.ls *List the names of grobs or viewports*

Description

Return a listing of the names of grobs or viewports.

This is a generic function with methods for grobs (including gTrees) and viewports (including vpTrees).

Usage

```
grid.ls(x=NULL, grobs=TRUE, viewports=FALSE, fullNames=FALSE,
        recursive=TRUE, print=TRUE, flatten=TRUE, ...)

nestedListing(x, gindent="  ", vpindent=gindent)
pathListing(x, gvpSep=" | ", gAlign=TRUE)
grobPathListing(x, ...)
```

Arguments

x	A grob or viewport or NULL. If NULL, the current grid display list is listed. For print functions, this should be the result of a call to grid.ls.
grobs	A logical value indicating whether to list grobs.
viewports	A logical value indicating whether to list viewports.
fullNames	A logical value indicating whether to embellish object names with information about the object type.
recursive	A logical value indicating whether recursive structures should also list their children.
print	A logical indicating whether to print the listing or a function that will print the listing.
flatten	A logical value indicating whether to flatten the listing. Otherwise a more complex hierarchical object is produced.
gindent	The indent used to show nesting in the output for grobs.
vpindent	The indent used to show nesting in the output for viewports.
gvpSep	The string used to separate viewport paths from grob paths.
gAlign	Logical indicating whether to align the left hand edge of all grob paths.
...	Arguments passed to the print function.

Details

If the argument x is NULL, the current contents of the grid display list are listed (both viewports and grobs). In other words, all objects representing the current scene are listed.

Otherwise, x should be a grob or a viewport.

The default behaviour of this function is to print information about the grobs in the current scene. It is also possible to add information about the viewports in the scene. By default, the listing is recursive, so all children of gTrees and all nested viewports are reported.

The format of the information can be controlled via the print argument, which can be given a function to perform the formatting. The nestedListing function produces a line per grob or viewport, with indenting used to show nesting. The pathListing function produces a line per grob or viewport, with viewport paths and grob paths used to show nesting. The grobPathListing is a simple derivation that only shows lines for grobs. The user can define new functions.

Value

The result of this function is either a "gridFlatListing" object (if flatten is TRUE) or a "gridListing" object.

The former is a simple (flat) list of vectors. This is convenient, for example, for working programmatically with the list of grob and viewport names, or for writing a new display function for the listing.

The latter is a more complex hierarchical object (list of lists), but it does contain more detailed information so may be of use for more advanced customisations.

Author(s)

Paul Murrell

See Also

grob viewport

Examples

```
# A gTree, called "parent", with childrenvp vpTree (vp2 within vp1)
# and child grob, called "child", with vp vpPath (down to vp2)
sampleGTree <- gTree(name="parent",
                     children=gList(grob(name="child", vp="vp1::vp2")),
                     childrenvp=vpTree(parent=viewport(name="vp1"),
                                       children=vpList(viewport(name="vp2"))))
grid.ls(sampleGTree)
# Show viewports too
grid.ls(sampleGTree, viewports=TRUE)
# Only show viewports
grid.ls(sampleGTree, viewports=TRUE, grobs=FALSE)
# Alternate displays
# nested listing, custom indent
grid.ls(sampleGTree, viewports=TRUE, print=nestedListing, gindent="--")
# path listing
grid.ls(sampleGTree, viewports=TRUE, print=pathListing)
```

```
# path listing, without grobs aligned
grid.ls(sampleGTree, viewports=TRUE, print=pathListing, gAlign=FALSE)
# grob path listing
grid.ls(sampleGTree, viewports=TRUE, print=grobPathListing)
# path listing, grobs only
grid.ls(sampleGTree, print=pathListing)
# path listing, viewports only
grid.ls(sampleGTree, viewports=TRUE, grobs=FALSE, print=pathListing)
# raw flat listing
str(grid.ls(sampleGTree, viewports=TRUE, print=FALSE))
```

`grid.move.to` *Move or Draw to a Specified Position*

Description

Grid has the notion of a current location. These functions sets that location.

Usage

```
grid.move.to(x = 0, y = 0, default.units = "npc", name = NULL,
             draw = TRUE, vp = NULL)

moveToGrob(x = 0, y = 0, default.units = "npc", name = NULL,
           vp = NULL)

grid.line.to(x = 1, y = 1, default.units = "npc",
             arrow = NULL, name = NULL,
             gp = gpar(), draw = TRUE, vp = NULL)

lineToGrob(x = 1, y = 1, default.units = "npc", arrow = NULL,
           name = NULL, gp = gpar(), vp = NULL)
```

Arguments

`x`	A numeric value or a unit object specifying an x-value.
`y`	A numeric value or a unit object specifying a y-value.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric values.
`arrow`	A list describing arrow heads to place at either end of the line, as produced by the `arrow` function.
`name`	A character identifier.
`draw`	A logical value indicating whether graphics output should be produced.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	A Grid viewport object (or NULL).

Details

Both functions create a move.to/line.to grob (a graphical object describing a move-to/line-to), but only `grid.move.to/line.to()` draws the move.to/line.to (and then only if `draw` is `TRUE`).

Value

A move.to/line.to grob. `grid.move.to/line.to()` returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `arrow`

Examples

```
grid.newpage()
grid.move.to(0.5, 0.5)
grid.line.to(1, 1)
grid.line.to(0.5, 0)
pushViewport(viewport(x=0, y=0, width=0.25, height=0.25, just=c("left", "bottom")))
grid.rect()
grid.grill()
grid.line.to(0.5, 0.5)
popViewport()
```

---

grid.newpage *Move to a New Page on a Grid Device*

Description

This function erases the current device or moves to a new page.

Usage

```
grid.newpage(recording = TRUE, clearGroups = TRUE)
```

Arguments

`recording`	A logical value to indicate whether the new-page operation should be saved onto the Grid display list.
`clearGroups`	A logical value indicating whether any groups that have been defined on the current page should be released (see `grid.group`).

Details

The new page is painted with the fill colour (gpar("fill")), which is often transparent. For devices with a *canvas* colour (the on-screen devices X11, windows and quartz), the page is first painted with the canvas colour and then the background colour.

There are two hooks called "before.grid.newpage" and "grid.newpage" (see setHook). The latter is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, get is called on it from within the **grid** namespace.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid

---

grid.null *Null Graphical Object*

Description

These functions create a NULL graphical object, which has zero width, zero height, and draw nothing. It can be used as a place-holder or as an invisible reference point for other drawing.

Usage

```
nullGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         default.units = "npc",
         name = NULL, vp = NULL)
grid.null(...)
```

Arguments

x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
vp	A Grid viewport object (or NULL).
...	Arguments passed to nullGrob().

Value

A null grob.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.newpage()
grid.null(name="ref")
grid.rect(height=grobHeight("ref"))
grid.segments(0, 0, grobX("ref", 0), grobY("ref", 0))
```

---

grid.pack *Pack an Object within a Frame*

Description

these functions, together with grid.frame and frameGrob are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with grid.frame or frameGrob then use these functions to pack objects into the frame.

Usage

```
grid.pack(gPath, grob, redraw = TRUE, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
          width = NULL, height = NULL,
          force.width = FALSE, force.height = FALSE, border = NULL,
          dynamic = FALSE)

packGrob(frame, grob, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
          width = NULL, height = NULL,
          force.width = FALSE, force.height = FALSE, border = NULL,
          dynamic = FALSE)
```

Arguments

gPath	A gPath object, which specifies a frame on the display list.
frame	An object of class frame, typically the output from a call to grid.frame.
grob	An object of class grob. The object to be packed.
redraw	A boolean indicating whether the output should be updated.
side	One of "left", "top", "right", "bottom" to indicate which side to pack the object on.
row	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame + 1, or NULL in which case the object occupies all rows.
row.before	Add the object to a new row just before this row.
row.after	Add the object to a new row just after this row.
col	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame + 1, or NULL in which case the object occupies all cols.
col.before	Add the object to a new col just before this col.
col.after	Add the object to a new col just after this col.
width	Specifies the width of the column that the object is added to (rather than allowing the width to be taken from the object).
height	Specifies the height of the row that the object is added to (rather than allowing the height to be taken from the object).
force.width	A logical value indicating whether the width of the column that the grob is being packed into should be EITHER the width specified in the call to grid.pack OR the maximum of that width and the pre-existing width.
force.height	A logical value indicating whether the height of the column that the grob is being packed into should be EITHER the height specified in the call to grid.pack OR the maximum of that height and the pre-existing height.
border	A unit object of length 4 indicating the borders around the object.
dynamic	If the width/height is taken from the grob being packed, this boolean flag indicates whether the "grobwidth"/"grobheight" unit refers directly to the grob, or uses a gPath to the grob. In the latter case, changes to the grob will trigger a recalculation of the width/height.

Details

packGrob modifies the given frame grob and returns the modified frame grob.

grid.pack destructively modifies a frame grob on the display list (and redraws the display list if redraw is TRUE).

These are (meant to be) very flexible functions. There are many different ways to specify where the new object is to be added relative to the objects already in the frame. The function checks that the specification is not self-contradictory.

NOTE that the width/height of the row/col that the object is added to is taken from the object itself unless the width/height is specified.

Value

packGrob returns a frame grob, but grid.pack returns NULL.

Author(s)

Paul Murrell

See Also

grid.frame, grid.place, grid.edit, and gPath.

---

grid.path *Draw a Path*

Description

These functions create and draw one or more paths. The final point of a path will automatically be connected to the initial point.

Usage

```
pathGrob(x, y,
         id=NULL, id.lengths=NULL,
         pathId=NULL, pathId.lengths=NULL,
         rule="winding",
         default.units="npc",
         name=NULL, gp=gpar(), vp=NULL)
grid.path(...)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
id	A numeric vector used to separate locations in x and y into sub-paths. All locations with the same id belong to the same sub-path.
id.lengths	A numeric vector used to separate locations in x and y into sub-paths. Specifies consecutive blocks of locations which make up separate sub-paths.
pathId	A numeric vector used to separate locations in x and y into distinct paths. All locations with the same pathId belong to the same path.
pathId.lengths	A numeric vector used to separate locations in x and y into paths. Specifies consecutive blocks of locations which make up separate paths.
rule	A character value specifying the fill rule: either "winding" or "evenodd".
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
name	A character identifier.

gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments passed to pathGrob().

Details

Both functions create a path grob (a graphical object describing a path), but only grid.path draws the path (and then only if draw is TRUE).

A path is like a polygon except that the former can contain holes, as interpreted by the fill rule; these fill a region if the path border encircles it an odd or non-zero number of times, respectively.

Not all graphics devices support this function: for example xfig and pictex do not.

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
pathSample <- function(x, y, rule, gp = gpar()) {
    if (is.na(rule))
        grid.path(x, y, id = rep(1:2, each = 4), gp = gp)
    else
        grid.path(x, y, id = rep(1:2, each = 4), rule = rule, gp = gp)
    if (!is.na(rule))
        grid.text(paste("Rule:", rule), y = 0, just = "bottom")
}

pathTriplet <- function(x, y, title) {
    pushViewport(viewport(height = 0.9, layout = grid.layout(1, 3),
                          gp = gpar(cex = .7)))
    grid.rect(y = 1, height = unit(1, "char"), just = "top",
              gp = gpar(col = NA, fill = "grey"))
    grid.text(title, y = 1, just = "top")
    pushViewport(viewport(layout.pos.col = 1))
    pathSample(x, y, rule = "winding",
               gp = gpar(fill = "grey"))
    popViewport()
    pushViewport(viewport(layout.pos.col = 2))
    pathSample(x, y, rule = "evenodd",
               gp = gpar(fill = "grey"))
    popViewport()
```

```
    pushViewport(viewport(layout.pos.col = 3))
    pathSample(x, y, rule = NA)
    popViewport()
    popViewport()
}

pathTest <- function() {
    grid.newpage()
    pushViewport(viewport(layout = grid.layout(5, 1)))
    pushViewport(viewport(layout.pos.row = 1))
    pathTriplet(c(.1, .1, .9, .9, .2, .2, .8, .8),
                c(.1, .9, .9, .1, .2, .8, .8, .2),
                "Nested rectangles, both clockwise")
    popViewport()
    pushViewport(viewport(layout.pos.row = 2))
    pathTriplet(c(.1, .1, .9, .9, .2, .8, .8, .2),
                c(.1, .9, .9, .1, .2, .2, .8, .8),
                "Nested rectangles, outer clockwise, inner anti-clockwise")
    popViewport()
    pushViewport(viewport(layout.pos.row = 3))
    pathTriplet(c(.1, .1, .4, .4, .6, .9, .9, .6),
                c(.1, .4, .4, .1, .6, .6, .9, .9),
                "Disjoint rectangles")
    popViewport()
    pushViewport(viewport(layout.pos.row = 4))
    pathTriplet(c(.1, .1, .6, .6, .4, .4, .9, .9),
                c(.1, .6, .6, .1, .4, .9, .9, .4),
                "Overlapping rectangles, both clockwise")
    popViewport()
    pushViewport(viewport(layout.pos.row = 5))
    pathTriplet(c(.1, .1, .6, .6, .4, .9, .9, .4),
                c(.1, .6, .6, .1, .4, .4, .9, .9),
                "Overlapping rectangles, one clockwise, other anti-clockwise")
    popViewport()
    popViewport()
}

pathTest()

# Drawing multiple paths at once
holed_rect <- cbind(c(.15, .15, -.15, -.15, .1, .1, -.1, -.1),
                    c(.15, -.15, -.15, .15, .1, -.1, -.1, .1))
holed_rects <- rbind(
    holed_rect + matrix(c(.7, .2), nrow = 8, ncol = 2, byrow = TRUE),
    holed_rect + matrix(c(.7, .8), nrow = 8, ncol = 2, byrow = TRUE),
    holed_rect + matrix(c(.2, .5), nrow = 8, ncol = 2, byrow = TRUE)
)
grid.newpage()
grid.path(x = holed_rects[, 1], y = holed_rects[, 2],
          id = rep(1:6, each = 4), pathId = rep(1:3, each = 8),
          gp = gpar(fill = c('red', 'blue', 'green')),
          rule = 'evenodd')
```

```
# Not specifying pathId will treat all points as part of the same path, thus
# having same fill
grid.newpage()
grid.path(x = holed_rects[, 1], y = holed_rects[, 2],
          id = rep(1:6, each = 4),
          gp = gpar(fill = c('red', 'blue', 'green')),
          rule = 'evenodd')
```

`grid.place` *Place an Object within a Frame*

Description

These functions provide a simpler (and faster) alternative to the `grid.pack()` and `packGrob` functions. They can be used to place objects within the existing rows and columns of a frame layout. They do not provide the ability to add new rows and columns nor do they affect the heights and widths of the rows and columns.

Usage

```
grid.place(gPath, grob, row = 1, col = 1, redraw = TRUE)
placeGrob(frame, grob, row = NULL, col = NULL)
```

Arguments

`gPath`	A gPath object, which specifies a frame on the display list.
`frame`	An object of class `frame`, typically the output from a call to `grid.frame`.
`grob`	An object of class `grob`. The object to be placed.
`row`	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame.
`col`	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame.
`redraw`	A boolean indicating whether the output should be updated.

Details

`placeGrob` modifies the given frame grob and returns the modified frame grob.

`grid.place` destructively modifies a frame grob on the display list (and redraws the display list if `redraw` is `TRUE`).

Value

`placeGrob` returns a frame grob, but `grid.place` returns `NULL`.

Author(s)

Paul Murrell

See Also

grid.frame, grid.pack, grid.edit, and gPath.

---

grid.plot.and.legend *A Simple Plot and Legend Demo*

Description

This function is just a wrapper for a simple demonstration of how a basic plot and legend can be drawn from scratch using grid.

Usage

```
grid.plot.and.legend()
```

Author(s)

Paul Murrell

Examples

```
grid.plot.and.legend()
```

---

grid.points *Draw Data Symbols*

Description

These functions create and draw data symbols.

Usage

```
grid.points(x = stats::runif(10),
            y = stats::runif(10),
            pch = 1, size = unit(1, "char"),
            default.units = "native", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)
pointsGrob(x = stats::runif(10),
           y = stats::runif(10),
           pch = 1, size = unit(1, "char"),
           default.units = "native", name = NULL,
           gp = gpar(), vp = NULL)
```

Arguments

x	numeric vector or unit object specifying x-values.
y	numeric vector or unit object specifying y-values.
pch	numeric or character vector indicating what sort of plotting symbol to use. See points for the interpretation of these values, and note fill below.
size	unit object specifying the size of the plotting symbols.
default.units	string indicating the default units to use if x or y are only given as numeric vectors.
name	character identifier.
gp	an R object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings; note that fill (and not bg as in package **graphics** points) is used to "fill", i.e., color the background of symbols with pch = 21:25.
draw	logical indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a points grob (a graphical object describing points), but only grid.points draws the points (and then only if draw is TRUE).

Value

A points grob. grid.points returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

---

grid.polygon *Draw a Polygon*

Description

These functions create and draw a polygon. The final point will automatically be connected to the initial point.

Usage

```
grid.polygon(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), draw=TRUE, vp=NULL)
polygonGrob(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
id	A numeric vector used to separate locations in x and y into multiple polygons. All locations with the same id belong to the same polygon.
id.lengths	A numeric vector used to separate locations in x and y into multiple polygons. Specifies consecutive blocks of locations which make up separate polygons.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a polygon grob (a graphical object describing a polygon), but only grid.polygon draws the polygon (and then only if draw is TRUE).

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.polygon()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polygon(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
             y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
             id=rep(1:5, 4),
             gp=gpar(fill=1:5))
# Using id.lengths
grid.newpage()
grid.polygon(x=outer(c(0, .5, 1, .5), 5:1/5),
             y=outer(c(.5, 1, .5, 0), 5:1/5),
             id.lengths=rep(4, 5),
             gp=gpar(fill=1:5))
```

---

grid.pretty *Generate a Sensible ("Pretty") Set of Breakpoints*

Description

Produces a pretty set of approximately n breakpoints within the range given.

This is a direct interface to R's graphical engine `GEpretty()` function, which also underlies base **graphics**' package `axis()`, `axTicks()`, etc.

Usage

```
grid.pretty(range, n = 5)
```

Arguments

range	a numeric vector of length at least two, as e.g., returned by `range()`.
n	a non-negative integer specifying the *approximate* number of breakpoints to be produced.

Value

A numeric vector of "pretty" breakpoints.

Author(s)

Paul Murrell

grid.raster *Render a raster object*

Description

Render a raster object (bitmap image) at the given location, size, and orientation.

Usage

```
grid.raster(image,
            x = unit(0.5, "npc"), y = unit(0.5, "npc"),
            width = NULL, height = NULL,
            just = "centre", hjust = NULL, vjust = NULL,
            interpolate = TRUE, default.units = "npc",
            name = NULL, gp = gpar(), vp = NULL)

rasterGrob(image,
           x = unit(0.5, "npc"), y = unit(0.5, "npc"),
           width = NULL, height = NULL,
           just = "centre", hjust = NULL, vjust = NULL,
           interpolate = TRUE, default.units = "npc",
           name = NULL, gp = gpar(), vp = NULL)
```

Arguments

image	Any R object that can be coerced to a raster object.
x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
width	A numeric vector or unit object specifying width.
height	A numeric vector or unit object specifying height.
just	The justification of the rectangle relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.

vp	A Grid viewport object (or NULL).
interpolate	A logical value indicating whether to linearly interpolate the image (the alternative is to use nearest-neighbour interpolation, which gives a more blocky result).

Details

Neither width nor height needs to be specified, in which case, the aspect ratio of the image is preserved. If both width and height are specified, it is likely that the image will be distorted.

Not all graphics devices are capable of rendering raster images and some may not be able to produce rotated images (i.e., if a raster object is rendered within a rotated viewport). See also the comments under rasterImage.

All graphical parameter settings in gp will be ignored, including alpha.

Value

A rastergrob grob.

Author(s)

Paul Murrell

See Also

as.raster.

dev.capabilities to see if it is supported.

Examples

```
redGradient <- matrix(hcl(0, 80, seq(50, 80, 10)),
                      nrow=4, ncol=5)
# interpolated
grid.newpage()
grid.raster(redGradient)
# blocky
grid.newpage()
grid.raster(redGradient, interpolate=FALSE)
# blocky and stretched
grid.newpage()
grid.raster(redGradient, interpolate=FALSE, height=unit(1, "npc"))

# The same raster drawn several times
grid.newpage()
grid.raster(0, x=1:3/4, y=1:3/4, width=.1, interpolate=FALSE)
```

grid.record *Encapsulate calculations and drawing*

Description

Evaluates an expression that includes both calculations and drawing that depends on the calculations so that both the calculations and the drawing will be rerun when the scene is redrawn (e.g., device resize or editing).

Intended *only* for expert use.

Usage

```
recordGrob(expr, list, name=NULL, gp=NULL, vp=NULL)
grid.record(expr, list, name=NULL, gp=NULL, vp=NULL)
```

Arguments

expr	object of mode expression or call or an unevaluated expression.
list	a list defining the environment in which expr is to be evaluated.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

A grob is created of special class "recordedGrob" (and drawn, in the case of grid.record). The drawDetails method for this class evaluates the expression with the list as the evaluation environment (and the grid Namespace as the parent of that environment).

Note

This function *must* be used instead of the function recordGraphics; all of the dire warnings about using recordGraphics responsibly also apply here.

Author(s)

Paul Murrell

See Also

recordGraphics

Examples

```
grid.record({
              w <- convertWidth(unit(1, "inches"), "npc")
              grid.rect(width=w)
            },
            list())
```

`grid.rect` *Draw rectangles*

Description

These functions create and draw rectangles.

Usage

```
grid.rect(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          width = unit(1, "npc"), height = unit(1, "npc"),
          just = "centre", hjust = NULL, vjust = NULL,
          default.units = "npc", name = NULL,
          gp=gpar(), draw = TRUE, vp = NULL)
rectGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         just = "centre", hjust = NULL, vjust = NULL,
         default.units = "npc", name = NULL,
         gp=gpar(), vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`just`	The justification of the rectangle relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.
`hjust`	A numeric vector specifying horizontal justification. If specified, overrides the `just` setting.
`vjust`	A numeric vector specifying vertical justification. If specified, overrides the `just` setting.
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.

name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a rect grob (a graphical object describing rectangles), but only grid.rect draws the rectangles (and then only if draw is TRUE).

Value

A rect grob. grid.rect returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

---

grid.refresh *Refresh the current grid scene*

Description

Replays the current grid display list.

Usage

```
grid.refresh()
```

Author(s)

Paul Murrell

grid.remove *Remove a Grid Graphical Object*

Description

Remove a grob from a gTree or a descendant of a gTree.

Usage

```
grid.remove(gPath, warn = TRUE, strict = FALSE, grep = FALSE,
            global = FALSE, allDevices = FALSE, redraw = TRUE)

grid.gremove(..., grep = TRUE, global = TRUE)

removeGrob(gTree, gPath, strict = FALSE, grep = FALSE,
           global = FALSE, warn = TRUE)
```

Arguments

gTree	A gTree object.
gPath	a gPath object. For grid.remove this specifies a gTree on the display list. For removeGrob this specifies a descendant of the specified gTree.
strict	a logical indicating whether the gPath must be matched exactly.
grep	a logical indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	a logical indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	a logical indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
warn	A logical to indicate whether failing to find the specified grob should trigger an error.
redraw	A logical value to indicate whether to redraw the grob.
...	arguments passed to grid.get.

Details

removeGrob copies the specified grob and returns a modified grob.

grid.remove destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

grid.gremove (g for global) is just a convenience wrapper for grid.remove with different defaults.

Value

removeGrob returns a grob object; grid.remove returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob.

---

grid.reorder *Reorder the children of a gTree*

Description

Change the order in which the children of a gTree get drawn.

Usage

```
grid.reorder(gPath, order, back=TRUE, grep=FALSE, redraw=TRUE)
reorderGrob(x, order, back=TRUE)
```

Arguments

gPath	A gPath object specifying a gTree within the current scene.
x	A gTree object to be modified.
order	A character vector or a numeric vector that specifies the new drawing order for the children of the gTree. May not refer to all children of the gTree (see Details).
back	Controls what happens when the order does not specify all children of the gTree (see Details).
grep	Should the gPath be treated as a regular expression?
redraw	Should the modified scene be redrawn?

Details

In the simplest case, order specifies a new ordering for all of the children of the gTree. The children may be specified either by name or by existing numerical order.

If the order does not specify all children of the gTree then, by default, the children specified by order are drawn first and then all remaining children are drawn. If back=FALSE then the children not specified in order are drawn first, followed by the specified children. This makes it easy to specify a send-to-back or bring-to-front reordering. The order argument is *always* in back-to-front order.

It is not possible to reorder the grid display list (the top-level grobs in the current scene) because the display list is a mixture of grobs and viewports (so it is not clear what reordering even means and it would be too easy to end up with a scene that would not draw). If you want to reorder the grid display list, try grid.grab() to create a gTree and then reorder (and redraw) that gTree.

Value

`grid.reorder()` is called for its side-effect of modifying the current scene. `reorderGrob()` returns the modified gTree.

Warning

This function may return a gTree that will not draw. For example, a gTree has two children, A and B (in that order), and the width of child B depends on the width of child A (e.g., a box around a piece of text). Switching the order so that B is drawn before A will not allow B to be drawn. If this happens with `grid.reorder()`, the modification will not be performed. If this happens with `reorderGrob()` it should be possible simply to restore the original order.

Author(s)

Paul Murrell

Examples

```
# gTree with two children, "red-rect" and "blue-rect" (in that order)
gt <- gTree(children=gList(
                rectGrob(gp=gpar(col=NA, fill="red"),
                         width=.8, height=.2, name="red-rect"),
                rectGrob(gp=gpar(col=NA, fill="blue"),
                         width=.2, height=.8, name="blue-rect")),
            name="gt")
grid.newpage()
grid.draw(gt)
# Spec entire order as numeric (blue-rect, red-rect)
grid.reorder("gt", 2:1)
# Spec entire order as character
grid.reorder("gt", c("red-rect", "blue-rect"))
# Only spec the one I want behind as character
grid.reorder("gt", "blue-rect")
# Only spec the one I want in front as character
grid.reorder("gt", "blue-rect", back=FALSE)
```

---

grid.segments *Draw Line Segments*

Description

These functions create and draw line segments.

Usage

```
grid.segments(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
```

```
              arrow = NULL,
              name = NULL, gp = gpar(), draw = TRUE, vp = NULL)
segmentsGrob(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
              arrow = NULL, name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x0	Numeric indicating the starting x-values of the line segments.
y0	Numeric indicating the starting y-values of the line segments.
x1	Numeric indicating the stopping x-values of the line segments.
y1	Numeric indicating the stopping y-values of the line segments.
default.units	A string.
arrow	A list describing arrow heads to place at either end of the line segments, as produced by the arrow function.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a segments grob (a graphical object describing segments), but only grid.segments draws the segments (and then only if draw is TRUE).

Value

A segments grob. grid.segments returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, arrow

grid.set *Set a Grid Graphical Object*

Description

Replace a grob or a descendant of a grob.

Usage

```
grid.set(gPath, newGrob, strict = FALSE, grep = FALSE,
         redraw = TRUE)

setGrob(gTree, gPath, newGrob, strict = FALSE, grep = FALSE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.set this specifies a grob on the display list. For setGrob this specifies a descendant of the specified gTree.
newGrob	A grob object.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
redraw	A logical value to indicate whether to redraw the grob.

Details

setGrob copies the specified grob and returns a modified grob.

grid.set destructively replaces a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

These functions should not normally be called by the user.

Value

setGrob returns a grob object; grid.set returns NULL.

Author(s)

Paul Murrell

See Also

grid.grob.

`grid.show.layout` *Draw a Diagram of a Grid Layout*

Description

This function uses Grid graphics to draw a diagram of a Grid layout.

Usage

```
grid.show.layout(l, newpage=TRUE, vp.ex = 0.8, bg = "light grey",
             cell.border = "blue", cell.fill = "light blue",
             cell.label = TRUE, label.col = "blue",
             unit.col = "red", vp = NULL, ...)
```

Arguments

`l`	A Grid layout object.
`newpage`	A logical value indicating whether to move on to a new page before drawing the diagram.
`vp.ex`	positive number, typically in $(0, 1]$, specifying the scaling of the layout.
`bg`	The colour used for the background.
`cell.border`	The colour used to draw the borders of the cells in the layout.
`cell.fill`	The colour used to fill the cells in the layout.
`cell.label`	A logical indicating whether the layout cells should be labelled.
`label.col`	The colour used for layout cell labels.
`unit.col`	The colour used for labelling the widths/heights of columns/rows.
`vp`	A Grid viewport object (or NULL).
`...`	Arguments passed to `format` for formatting the layout width and height annotations.

Details

A viewport is created within `vp` to provide a margin for annotation, and the layout is drawn within that new viewport. The margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the layout regions are filled with light blue and framed with a blue border. The diagram is annotated with the widths and heights (including units) of the columns and rows of the layout using red text. (All colours are defaults and may be customised via function arguments.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.layout

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                     heights=unit(rep(1, 4),
                                  c("lines", "lines", "lines", "null")),
                     widths=unit(c(1, 1), "inches")))
```

---

grid.show.viewport *Draw a Diagram of a Grid Viewport*

Description

This function uses Grid graphics to draw a diagram of a Grid viewport.

Usage

```
grid.show.viewport(v, parent.layout = NULL, newpage = TRUE,
                   vp.ex = 0.8, border.fill="light grey",
                   vp.col="blue", vp.fill="light blue",
                   scale.col="red",
                   vp = NULL)
```

Arguments

v	A Grid viewport object.
parent.layout	A grid layout object. If this is not NULL and the viewport given in v has its location specified relative to the layout, then the diagram shows the layout and which cells v occupies within the layout.
newpage	A logical value to indicate whether to move to a new page before drawing the diagram.
vp.ex	positive number, typically in $(0, 1]$, specifying the scaling of the layout.
border.fill	Colour to fill the border margin.
vp.col	Colour for the border of the viewport region.
vp.fill	Colour to fill the viewport region.
scale.col	Colour to draw the viewport axes.
vp	A Grid viewport object (or NULL).

Details

A viewport is created within vp to provide a margin for annotation, and the diagram is drawn within that new viewport. By default, the margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the viewport region is filled with light blue and framed with a blue border. The diagram is annotated with the width and height (including units) of the viewport, the (x, y) location of the viewport, and the x- and y-scales of the viewport, using red lines and text.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            width=unit(1, "inches"), height=unit(1, "inches")))
grid.show.viewport(viewport(layout.pos.row=2, layout.pos.col=2:3),
                   grid.layout(3, 4))
```

---

grid.stroke *Stroke or Fill a Path*

Description

These functions stroke (draw a line along the border) or fill (or both) a path, where the path is defined by a grob.

Usage

```
strokeGrob(x, ...)
## S3 method for class 'grob'
strokeGrob(x, name=NULL, gp=gpar(), vp=NULL, ...)
## S3 method for class 'GridPath'
strokeGrob(x, name=NULL, vp=NULL, ...)
grid.stroke(...)
fillGrob(x, ...)
## S3 method for class 'grob'
fillGrob(x, rule=c("winding", "evenodd"),
         name=NULL, gp=gpar(), vp=NULL, ...)
```

```
## S3 method for class 'GridPath'
fillGrob(x, name=NULL, vp=NULL, ...)
grid.fill(...)
fillStrokeGrob(x, ...)
## S3 method for class 'grob'
fillStrokeGrob(x, rule=c("winding", "evenodd"),
               name=NULL, gp=gpar(), vp=NULL, ...)
## S3 method for class 'GridPath'
fillStrokeGrob(x, name=NULL, vp=NULL, ...)
grid.fillStroke(...)
as.path(x, gp=gpar(), rule=c("winding", "evenodd"))
```

Arguments

x	A grob or the result of a call to as.path().
rule	A fill rule.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to grid.*() passed on to *Grob(), or additional arguments passed on to methods.

Details

A path is defined by the shapes that the grob given in x would draw. The grob only contributes to the outline of the path; graphical parameter settings such as line colour and fill are ignored.

grid.stroke() will only ever draw the border (even when a fill is specified).

grid.fill() will only ever fill the path (even when a line colour is specified).

A stroke will only ever happen if a non-transparent line colour is specified and a fill will only ever happen if a non-transparent fill is specified.

as.path() allows graphical parameter settings and a fill rule to be associated with a grob. This can be useful when specifying a clipping path for a viewport (see viewport).

Not all graphics devices support these functions: for example xfig and pictex do not.

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid

Examples

```
## NOTE: on devices without support for stroking and filling
##       nothing will be drawn
grid.newpage()
grid.stroke(textGrob("hello", gp=gpar(cex=10)))
grid.fill(circleGrob(1:2/3, r=.3), gp=gpar(fill=rgb(1,0,0,.5)))
```

`grid.text` *Draw Text*

Description

These functions create and draw text and plotmath expressions.

Usage

```
grid.text(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), draw = TRUE, vp = NULL)

textGrob(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), vp = NULL)
```

Arguments

`label`	A character or expression vector. Other objects are coerced by `as.graphicsAnnot`.
`x`	A numeric vector or unit object specifying x-values.
`y`	A numeric vector or unit object specifying y-values.
`just`	The justification of the text relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left (bottom) alignment and 1 means right (top) alignment.
`hjust`	A numeric vector specifying horizontal justification. If specified, overrides the `just` setting.
`vjust`	A numeric vector specifying vertical justification. If specified, overrides the `just` setting.
`rot`	The angle to rotate the text.
`check.overlap`	A logical value to indicate whether to check for and omit overlapping text (within that grob).

default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a text grob (a graphical object describing text), but only grid.text draws the text (and then only if draw is TRUE).

If the label argument is an expression, the output is formatted as a mathematical annotation, as for base graphics text.

Value

A text grob. grid.text() returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.newpage()
x <- stats::runif(20)
y <- stats::runif(20)
rot <- stats::runif(20, 0, 360)
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20, col="grey"))
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20), check.overlap=TRUE)

grid.newpage() ## plotmath example
grid.text(quote(frac(e^{-x^2/2}, sqrt(2*pi))), x=x, y=y, rot=stats::runif(20, -45,45),
          gp=gpar(fontsize=17, col=4), check.overlap=TRUE)

grid.newpage()
draw.text <- function(just, i, j) {
  grid.text("ABCD", x=x[j], y=y[i], just=just)
  grid.text(deparse(substitute(just)), x=x[j], y=y[i] + unit(2, "lines"),
            gp=gpar(col="grey", fontsize=8))
}
x <- unit(1:4/5, "npc")
y <- unit(1:4/5, "npc")
```

```
grid.grill(h=y, v=x, gp=gpar(col="grey"))
draw.text(c("bottom"), 1, 1)
draw.text(c("left", "bottom"), 2, 1)
draw.text(c("right", "bottom"), 3, 1)
draw.text(c("centre", "bottom"), 4, 1)
draw.text(c("centre"), 1, 2)
draw.text(c("left", "centre"), 2, 2)
draw.text(c("right", "centre"), 3, 2)
draw.text(c("centre", "centre"), 4, 2)
draw.text(c("top"), 1, 3)
draw.text(c("left", "top"), 2, 3)
draw.text(c("right", "top"), 3, 3)
draw.text(c("centre", "top"), 4, 3)
draw.text(c(), 1, 4)
draw.text(c("left"), 2, 4)
draw.text(c("right"), 3, 4)
draw.text(c("centre"), 4, 4)
```

`grid.xaxis` *Draw an X-Axis*

Description

These functions create and draw an x-axis.

Usage

```
grid.xaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

xaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

`at`	A numeric vector of x-value locations for the tick marks.
`label`	A logical value indicating whether to draw the labels on the tick marks, or an expression or character vector which specify the labels to use. If not logical, must be the same length as the `at` argument.
`main`	A logical value indicating whether to draw the axis at the bottom (`TRUE`) or at the top (`FALSE`) of the viewport.
`edits`	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever `at` is `NULL`.
`name`	A character identifier.

gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create an xaxis grob (a graphical object describing an xaxis), but only grid.xaxis draws the xaxis (and then only if draw is TRUE).

Value

An xaxis grob. grid.xaxis returns the value invisibly.

Children

If the at slot of an xaxis grob is not NULL then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the at slot is NULL then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.yaxis

---

grid.xspline *Draw an Xspline*

Description

These functions create and draw an xspline, a curve drawn relative to control points.

Usage

```
grid.xspline(...)
xsplineGrob(x = c(0, 0.5, 1, 0.5), y = c(0.5, 1, 0.5, 0),
            id = NULL, id.lengths = NULL,
            default.units = "npc",
            shape = 0, open = TRUE, arrow = NULL, repEnds = TRUE,
            name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations of spline control points.
y	A numeric vector or unit object specifying y-locations of spline control points.
id	A numeric vector used to separate locations in x and y into multiple xsplines. All locations with the same id belong to the same xspline.
id.lengths	A numeric vector used to separate locations in x and y into multiple xsplines. Specifies consecutive blocks of locations which make up separate xsplines.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
shape	A numeric vector of values between -1 and 1, which control the shape of the spline relative to the control points.
open	A logical value indicating whether the spline is a line or a closed shape.
arrow	A list describing arrow heads to place at either end of the xspline, as produced by the arrow function.
repEnds	A logical value indicating whether the first and last control points should be replicated for drawing the curve (see Details below).
name	A character identifier.
gp	An object of class "gpar", typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to xsplineGrob.

Details

Both functions create an xspline grob (a graphical object describing an xspline), but only grid.xspline draws the xspline.

An xspline is a line drawn relative to control points. For each control point, the line may pass through (interpolate) the control point or it may only approach (approximate) the control point; the behaviour is determined by a shape parameter for each control point.

If the shape parameter is greater than zero, the spline approximates the control points (and is very similar to a cubic B-spline when the shape is 1). If the shape parameter is less than zero, the spline interpolates the control points (and is very similar to a Catmull-Rom spline when the shape is -1). If the shape parameter is 0, the spline forms a sharp corner at that control point.

For open xsplines, the start and end control points must have a shape of 0 (and non-zero values are silently converted to zero without warning).

For open xsplines, by default the start and end control points are actually replicated before the curve is drawn. A curve is drawn between (interpolating or approximating) the second and third of each set of four control points, so this default behaviour ensures that the resulting curve starts at the first control point you have specified and ends at the last control point. The default behaviour can be turned off via the repEnds argument, in which case the curve that is drawn starts (approximately) at the second control point and ends (approximately) at the first and second-to-last control point.

The repEnds argument is ignored for closed xsplines.

Missing values are not allowed for x and y (i.e., it is not valid for a control point to be missing).

For closed xsplines, a curve is automatically drawn between the final control point and the initial control point.

Value

A grob object.

References

Blanc, C. and Schlick, C. (1995), "X-splines : A Spline Model Designed for the End User", in *Proceedings of SIGGRAPH 95*, pp. 377–386. https://dept-info.labri.fr/~schlick/DOC/sig1.html

See Also

Grid, viewport, arrow.

xspline.

Examples

```
x <- c(0.25, 0.25, 0.75, 0.75)
y <- c(0.25, 0.75, 0.75, 0.25)

xsplineTest <- function(s, i, j, open) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  grid.points(x, y, default.units="npc", pch=16, size=unit(2, "mm"))
  grid.xspline(x, y, shape=s, open=open, gp=gpar(fill="grey"))
  grid.text(s, gp=gpar(col="grey"),
            x=unit(x, "npc") + unit(c(-1, -1, 1, 1), "mm"),
            y=unit(y, "npc") + unit(c(-1, 1, 1, -1), "mm"),
            hjust=c(1, 1, 0, 0),
            vjust=c(1, 0, 0, 1))
  popViewport()
}

pushViewport(viewport(width=.5, x=0, just="left",
                      layout=grid.layout(3, 3, respect=TRUE)))
pushViewport(viewport(layout.pos.row=1))
grid.text("Open Splines", y=1, just="bottom")
popViewport()
xsplineTest(c(0, -1, -1, 0), 1, 1, TRUE)
xsplineTest(c(0, -1,  0, 0), 1, 2, TRUE)
xsplineTest(c(0, -1,  1, 0), 1, 3, TRUE)
xsplineTest(c(0,  0, -1, 0), 2, 1, TRUE)
xsplineTest(c(0,  0,  0, 0), 2, 2, TRUE)
xsplineTest(c(0,  0,  1, 0), 2, 3, TRUE)
xsplineTest(c(0,  1, -1, 0), 3, 1, TRUE)
xsplineTest(c(0,  1,  0, 0), 3, 2, TRUE)
xsplineTest(c(0,  1,  1, 0), 3, 3, TRUE)
popViewport()
pushViewport(viewport(width=.5, x=1, just="right",
                      layout=grid.layout(3, 3, respect=TRUE)))
```

```
pushViewport(viewport(layout.pos.row=1))
grid.text("Closed Splines", y=1, just="bottom")
popViewport()
xsplineTest(c(-1, -1, -1, -1), 1, 1, FALSE)
xsplineTest(c(-1, -1,  0, -1), 1, 2, FALSE)
xsplineTest(c(-1, -1,  1, -1), 1, 3, FALSE)
xsplineTest(c( 0,  0, -1,  0), 2, 1, FALSE)
xsplineTest(c( 0,  0,  0,  0), 2, 2, FALSE)
xsplineTest(c( 0,  0,  1,  0), 2, 3, FALSE)
xsplineTest(c( 1,  1, -1,  1), 3, 1, FALSE)
xsplineTest(c( 1,  1,  0,  1), 3, 2, FALSE)
xsplineTest(c( 1,  1,  1,  1), 3, 3, FALSE)
popViewport()
```

`grid.yaxis` *Draw a Y-Axis*

Description

These functions create and draw a y-axis.

Usage

```
grid.yaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

yaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

`at`	A numeric vector of y-value locations for the tick marks.
`label`	A logical value indicating whether to draw the labels on the tick marks, or an expression or character vector which specify the labels to use. If not logical, must be the same length as the `at` argument.
`main`	A logical value indicating whether to draw the axis at the left (`TRUE`) or at the right (`FALSE`) of the viewport.
`edits`	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever `at` is `NULL`.
`name`	A character identifier.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport object (or `NULL`).

Details

Both functions create a yaxis grob (a graphical object describing a yaxis), but only `grid.yaxis` draws the yaxis (and then only if `draw` is `TRUE`).

Value

A yaxis grob. `grid.yaxis` returns the value invisibly.

Children

If the `at` slot of an xaxis grob is not `NULL` then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the `at` slot is `NULL` then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `grid.xaxis`

`gridCoords` *Create Sets of Coordinates for Grid Grobs*

Description

These functions support the development of `grobPoints` methods for custom grobs.

Usage

```
gridCoords(x, y)
gridGrobCoords(x, name, rule = NULL)
gridGTreeCoords(x, name)
emptyCoords
emptyGrobCoords(name)
emptyGTreeCoords(name)
isEmptyCoords(coords)
```

Arguments

x	For gridCoords a numeric vector. For gridGrobCoords a list of "GridCoords" objects. For gridGTreeCoords a list of either "GridGrobCoords" or "GridGTreeCoords" objects.
y	A numeric vector.
name	A character value.
rule	A fill rule, either "winding" or "evenodd", or NULL.
coords	A set of grob coordinates (as generated by grobCoords).

Details

These functions help the developer of a grobPoints method to generate the coordinates from a custom grob.

The emptyCoords object can be used to return a "null" result (e.g., when asking for closed coordinates on an open line) and the isEmptyCoords function can be used to check for "null" results.

Value

For gridCoords a "GridCoords" object. For gridGrobCoords a "GridGrobCoords" object. For gridGTreeCoords a "GridGTreeCoords" object.

Author(s)

Paul Murrell

---

grobCoords *Calculate Points on the Perimeter of a Grob*

Description

These functions calculate points along the perimeter (or length) of a grob.

Usage

```
grobCoords(x, closed, ...)
grobPoints(x, closed, ...)
isClosed(x, ...)
```

Arguments

x	A grob object.
closed	Whether we are asking for points along the perimeter of a closed object or points along the length of an open object. Some grobs (e.g., X-splines) can do both. This defaults to TRUE except for known cases that are not closed (e.g., lines and segments).
...	Arguments to be used by methods.

Details

The difference between grobCoords and grobPoints is that grobCoords performs all pre- and post-drawing operations on the grob that would normally occur if the grob was being drawn, then calls grobPoints. So the former takes into account any vp and gp settings on the grob. This means that users should usually only want to call grobCoords; only (expert) developers may have a need to call grobPoints.

Custom grobs can write their own methods for grobPoints (see gridCoords).

The isClosed function returns TRUE or FALSE to indicate whether a grob is a closed shape. The default response is TRUE, unless a method has been defined otherwise (e.g., for lines and line segments).

Value

Either a "GridGrobCoords" object (a list of lists with components x and y) or a "GridGTreeCoords" object (a list of "GridGrobCoords" and/or "GridGTreeCoords" objects).

All locations are in inches relative to the current **grid** viewport.

Author(s)

Paul Murrell

---

grobName *Generate a Name for a Grob*

Description

This function generates a unique (within-session) name for a grob, based on the grob's class.

Usage

```
grobName(grob = NULL, prefix = "GRID")
```

Arguments

grob	A grob object or NULL.
prefix	The prefix part of the name.

Value

A character string of the form prefix.class(grob).index

Author(s)

Paul Murrell

grobWidth *Create a Unit Describing the Width of a Grob*

Description

These functions create a unit object describing the width or height of a grob. They are generic.

Usage

```
grobWidth(x)
grobHeight(x)
grobAscent(x)
grobDescent(x)
```

Arguments

x	A grob object.

Value

A unit object.

Author(s)

Paul Murrell

See Also

unit and stringWidth

grobX *Create a Unit Describing a Grob Boundary Location*

Description

These functions create a unit object describing a location somewhere on the boundary of a grob. They are generic.

Usage

```
grobX(x, theta)
grobY(x, theta)
```

Arguments

x	A grob, or gList, or gTree, or gPath.
theta	An angle indicating where the location is on the grob boundary. Can be one of "east", "north", "west", or "south", which correspond to angles 0, 90, 180, and 270, respectively.

Details

The angle is anti-clockwise with zero corresponding to a line with an origin centred between the extreme points of the shape, and pointing at 3 o'clock.

If the grob describes a single shape, the boundary value should correspond to the exact edge of the shape.

If the grob describes multiple shapes, the boundary value will either correspond to the edge of a bounding box around all of the shapes described by the grob (for multiple rectangles, circles, xsplines, or text), or to a convex hull around all vertices of all shapes described by the grob (for multiple polygons, points, lines, polylines, and segments).

Points grobs are currently a special case because the convex hull is based on the data symbol *locations* and does not take into account the extent of the data symbols themselves.

The extents of any arrow heads are currently *not* taken into account.

Value

A unit object.

Author(s)

Paul Murrell

See Also

unit and grobWidth

---

legendGrob *Constructing a Legend Grob*

Description

Constructing a legend grob (in progress)

Usage

```
legendGrob(labels, nrow, ncol, byrow = FALSE,
           do.lines = has.lty || has.lwd, lines.first = TRUE,
           hgap = unit(1, "lines"), vgap = unit(1, "lines"),
           default.units = "lines", pch, gp = gpar(), vp = NULL)

grid.legend(..., draw=TRUE)
```

Arguments

`labels`	legend labels (expressions)
`nrow`, `ncol`	integer; the number of rows or columns, respectively of the legend "layout". `nrow` is optional and typically computed from the number of labels and `ncol`.
`byrow`	logical indicating whether rows of the legend are filled first.
`do.lines`	logical indicating whether legend lines are drawn.
`lines.first`	logical indicating whether legend lines are drawn first and hence in a plain "below" legend symbols.
`hgap`	horizontal space between the legend entries
`vgap`	vertical space between the legend entries
`default.units`	default units, see `unit`.
`pch`	legend symbol, numeric or character, passed to `pointsGrob()`; see also `points` for interpretation of the numeric codes.
`gp`	an R object of class `"gpar"`, typically the output from a call to the function `gpar`, is basically a list of graphical parameter settings.
`vp`	a Grid `viewport` object (or `NULL`).
`...`	for `grid.legend()`: all the arguments above are passed to `legendGrob()`.
`draw`	logical indicating whether graphics output should be produced.

Value

Both functions create a legend `grob` (a graphical object describing a plot legend), but only `grid.legend` draws it (only if `draw` is `TRUE`).

See Also

Grid, `viewport`; `pointsGrob`, `linesGrob`.

`grid.plot.and.legend` contains a simple example.

Examples

```
## Data:
n <- 10
x <- stats::runif(n) ; y1 <- stats::runif(n) ; y2 <- stats::runif(n)
## Construct the grobs :
plot <- gTree(children=gList(rectGrob(),
                pointsGrob(x, y1, pch=21, gp=gpar(col=2, fill="gray")),
                pointsGrob(x, y2, pch=22, gp=gpar(col=3, fill="gray")),
                xaxisGrob(),
                yaxisGrob()))
legd <- legendGrob(c("Girls", "Boys", "Other"), pch=21:23,
                   gp=gpar(col = 2:4, fill = "gray"))
gg <- packGrob(packGrob(frameGrob(), plot),
               legd, height=unit(1,"null"), side="right")

## Now draw it on a new device page:
```

```
grid.newpage()
pushViewport(viewport(width=0.8, height=0.8))
grid.draw(gg)
```

---

makeContent *Customised grid Grobs*

Description

These generic hook functions are called whenever a grid grob is drawn. They provide an opportunity for customising the drawing context and drawing content of a new class derived from grob (or gTree).

Usage

```
makeContext(x)
makeContent(x)
```

Arguments

x	A grid grob.

Details

These functions are called by the `grid.draw` methods for grobs and gTrees.

`makeContext` is called first during the drawing of a grob. This function should be used to *modify* the `vp` slot of `x` (and/or the `childrenvp` slot if `x` is a gTree). The function *must* return the modified `x`. Note that the default behaviour for grobs is to push any viewports in the `vp` slot, and for gTrees is to also push and up any viewports in the `childrenvp` slot, so this function is used to customise the drawing context for a grob or gTree.

`makeContent` is called next and is where any additional calculations should occur and graphical content should be generated (see, for example, `grid:::makeContent.xaxis`). This function should be used to *modify* the `children` of a gTree. The function *must* return the modified `x`. Note that the default behaviour for gTrees is to draw all grobs in the `children` slot, so this function is used to customise the drawing content for a gTree. It is also possible to customise the drawing content for a simple grob, but more care needs to be taken; for example, the function should return a standard grid primitive with a `drawDetails()` method in this case.

Note that these functions should be *cumulative* in their effects, so that the `x` returned by `makeContent()` *includes* any changes made by `makeContext()`.

Note that `makeContext` is also called in the calculation of `"grobwidth"` and `"grobheight"` units.

Value

Both functions are expected to return a grob or gTree (a modified version of `x`).

Author(s)

Paul Murrell

References

"Changes to grid for R 3.0.0", Paul Murrell, *The R Journal* (2013) 5:2, pages 148-160.

See Also

grid.draw

---

patterns *Define Gradient and Pattern Fills*

Description

Functions to define gradient fills and pattern fills.

Usage

```
linearGradient(colours = c("black", "white"),
               stops = seq(0, 1, length.out = length(colours)),
               x1 = unit(0, "npc"), y1 = unit(0, "npc"),
               x2 = unit(1, "npc"), y2 = unit(1, "npc"),
               default.units = "npc",
               extend = c("pad", "repeat", "reflect", "none"),
               group = TRUE)
radialGradient(colours = c("black", "white"),
               stops = seq(0, 1, length.out = length(colours)),
               cx1 = unit(.5, "npc"), cy1 = unit(.5, "npc"),
               r1 = unit(0, "npc"),
               cx2 = unit(.5, "npc"), cy2 = unit(.5, "npc"),
               r2 = unit(.5, "npc"),
               default.units = "npc",
               extend = c("pad", "repeat", "reflect", "none"),
               group = TRUE)
pattern(grob,
        x = 0.5, y = 0.5, width = 1, height = 1,
        default.units = "npc",
        just="centre", hjust=NULL, vjust=NULL,
        extend = c("pad", "repeat", "reflect", "none"),
        gp = gpar(fill="transparent"),
        group = TRUE)
```

Arguments

colours	Two or more colours for the gradient to transition between.
stops	Locations of the gradient colours between the start and end points of the gradient (as a proportion of the distance from the start point to the end point).
x1, y1, x2, y2	The start and end points for a linear gradient.
default.units	The coordinate system to use if any location or dimension is specified as just a numeric value.
extend	What happens outside the start and end of the gradient (see Details).
cx1, cy1, r1, cx2, cy2, r2	The centre and radius of the start and end circles for a radial gradient.
grob	A grob (or a gTree) that will be drawn as the tile in a pattern fill.
x, y, width, height	The size of the tile for a pattern fill.
just, hjust, vjust	The justification of the tile relative to its location.
gp	Default graphical parameter settings for the tile.
group	A logical indicating whether the gradient or pattern is relative to the bounding box of the grob or whether it is relative to individual shapes within the grob.

Details

Use these functions to define a gradient fill or pattern fill and then use the resulting object as the value for `fill` in a call to the `gpar()` function.

The possible values of extend, and their meanings, are:

- [pad:] propagate the value of the gradient at its boundary.
- [none:] produce no fill beyond the limits of the gradient.
- [repeat:] repeat the fill.
- [reflect:] repeat the fill in reverse.

To create a tiling pattern, provide a simple grob (like a circle), specify the location and size of the pattern to include the simple grob, and specify `extend="repeat"`.

On viewports, gradients and patterns are relative to the entire viewport, unless `group = FALSE`, in which case they are relative to individual grobs as they are drawn. On gTrees, gradients and patterns are relative to a bounding box around all of the children of the gTree, unless `group = FALSE`, in which case they are relative to individual children as they are drawn. On grobs, gradients and patterns are relative to a bounding box around all of the shapes that are drawn by the grob, unless `group = FALSE`, in which case they are relative to individual shapes.

Value

A linear gradient or radial gradient or pattern object.

Warning

Gradient fills and pattern fills are not supported on all graphics devices. Where they are not supported, closed shapes will be rendered with a transparent fill. Where they are supported, not all values of `extend` are supported.

On Cairo devices, use of clipping in the pattern definition should be avoided because it is very likely to result in distortion of the pattern tile.

Author(s)

Paul Murrell

See Also

`gpar`

`plotViewport` *Create a Viewport with a Standard Plot Layout*

Description

This is a convenience function for producing a viewport with the common S-style plot layout – i.e., a central plot region surrounded by margins given in terms of a number of lines of text.

Usage

```
plotViewport(margins=c(5.1, 4.1, 4.1, 2.1), ...)
```

Arguments

`margins`	A numeric vector interpreted in the same way as `par(mar)` in base graphics.
`...`	All other arguments will be passed to a call to the `viewport()` function.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

`viewport` and `dataViewport`.

Querying the Viewport Tree

Get the Current Grid Viewport (Tree)

Description

`current.viewport()` returns the viewport that Grid is going to draw into.

`current.parent` returns the parent of the current viewport.

`current.vpTree` returns the entire Grid viewport tree.

`current.vpPath` returns the viewport path to the current viewport.

`current.transform` returns the transformation matrix for the current viewport.

`current.rotation` returns the (total) rotation for the current viewport.

Usage

```
current.viewport()
current.parent(n=1)
current.vpTree(all=TRUE)
current.vpPath()
current.transform()
```

Arguments

`n`	The number of generations to go up.
`all`	A logical value indicating whether the entire viewport tree should be returned.

Details

It is possible to get the grandparent of the current viewport (or higher) using the `n` argument to `current.parent()`.

The parent of the ROOT viewport is `NULL`. It is an error to request the grandparent of the ROOT viewport.

If `all` is `FALSE` then `current.vpTree` only returns the subtree below the current viewport.

Value

A Grid viewport object from `current.viewport` or `current.vpTree`.

`current.transform` returns a 4x4 transformation matrix.

The viewport path returned by `current.vpPath` is `NULL` if the current viewport is the `ROOT` viewport

Author(s)

Paul Murrell

See Also

viewport

Examples

```
grid.newpage()
pushViewport(viewport(width=0.8, height=0.8, name="A"))
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
upViewport(1)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
pushViewport(viewport(width=0.8, height=0.8, name="D"))
current.vpPath()
upViewport(1)
current.vpPath()
current.vpTree()
current.viewport()
current.vpTree(all=FALSE)
popViewport(0)
```

---

resolveRasterSize *Utility function to resolve the size of a raster grob*

Description

Determine the width and height of a raster grob when one or both are not given explicitly.

The result depends on both the aspect ratio of the raster image and the aspect ratio of the physical drawing context, so the result is only valid for the drawing context in which this function is called.

Usage

```
resolveRasterSize(x)
```

Arguments

x	A raster grob

Details

A raster grob can be specified with width and/or height of NULL, which means that the size at which the raster is drawn will be decided at drawing time.

Value

A raster grob, with explicit width and height.

See Also

grid.raster

Examples

```
# Square raster
rg <- rasterGrob(matrix(0))
# Fill the complete page (if page is square)
grid.newpage()
resolveRasterSize(rg)$height
grid.draw(rg)
# Forced to fit tall thin region
grid.newpage()
pushViewport(viewport(width=.1))
resolveRasterSize(rg)$height
grid.draw(rg)
```

---

roundrect *Draw a rectangle with rounded corners*

Description

Draw a *single* rectangle with rounded corners.

Usage

```
roundrectGrob(x=0.5, y=0.5, width=1, height=1,
              default.units="npc",
              r=unit(0.1, "snpc"),
              just="centre",
              name=NULL, gp=NULL, vp=NULL)
grid.roundrect(...)
```

Arguments

x, y, width, height	The location and size of the rectangle.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
r	The radius of the rounded corners.
just	The justification of the rectangle relative to its location.
name	A name to identify the grob.
gp	Graphical parameters to apply to the grob.
vp	A viewport object or NULL.
...	Arguments to be passed to roundrectGrob().

Details

At present, this function can only be used to draw *one* rounded rectangle.

Examples

```
grid.roundrect(width=.5, height=.5, name="rr")
theta <- seq(0, 360, length.out=50)
for (i in 1:50)
    grid.circle(x=grobX("rr", theta[i]),
                y=grobY("rr", theta[i]),
                r=unit(1, "mm"),
                gp=gpar(fill="black"))
```

---

showGrob *Label grid grobs*

Description

Produces a graphical display of (by default) the current grid scene, with labels showing the names of each grob in the scene. It is also possible to label only specific grobs in the scene.

Usage

```
showGrob(x = NULL,
         gPath = NULL, strict = FALSE, grep = FALSE,
         recurse = TRUE, depth = NULL,
         labelfun = grobLabel, ...)
```

Arguments

x	If NULL, the current grid scene is labelled. Otherwise, a grob (or gTree) to draw and then label.
gPath	A path identifying a subset of the current scene or grob to be labelled.
strict	Logical indicating whether the gPath is strict.
grep	Logical indicating whether the gPath is a regular expression.
recurse	Should the children of gTrees also be labelled?
depth	Only display grobs at the specified depth (may be a vector of depths).
labelfun	Function used to generate a label from each grob.
...	Arguments passed to labelfun to control fine details of the generated label.

Details

None of the labelling is recorded on the grid display list so the original scene can be reproduced by calling grid.refresh.

See Also

grob and gTree

Examples

```
grid.newpage()
gt <- gTree(childrenvp=vpStack(
              viewport(x=0, width=.5, just="left", name="vp"),
              viewport(y=.5, height=.5, just="bottom", name="vp2")),
            children=gList(rectGrob(vp="vp::vp2", name="child")),
            name="parent")
grid.draw(gt)
showGrob()
showGrob(gPath="child")
showGrob(recurse=FALSE)
showGrob(depth=1)
showGrob(depth=2)
showGrob(depth=1:2)
showGrob(gt)
showGrob(gt, gPath="child")
showGrob(just="left", gp=gpar(col="red", cex=.5), rot=45)
showGrob(labelfun=function(grob, ...) {
    x <- grobX(grob, "west")
    y <- grobY(grob, "north")
    gTree(children=gList(rectGrob(x=x, y=y,
            width=stringWidth(grob$name) + unit(2, "mm"),
            height=stringHeight(grob$name) + unit(2, "mm"),
            gp=gpar(col=NA, fill=rgb(1, 0, 0, .5)),
            just=c("left", "top")),
            textGrob(grob$name,
                     x=x + unit(1, "mm"), y=y - unit(1, "mm"),
                     just=c("left", "top"))))
})

## Not run:
    # Examples from higher-level packages

    library(lattice)
    # Ctrl-c after first example
    example(histogram)
    showGrob()
    showGrob(gPath="plot_01.ylab")

    library(ggplot2)
    # Ctrl-c after first example
    example(qplot)
    showGrob()
    showGrob(recurse=FALSE)
    showGrob(gPath="panel-3-3")
    showGrob(gPath="axis.title", grep=TRUE)
    showGrob(depth=2)
```

```
## End(Not run)
```

---

showViewport *Display grid viewports*

Description

Produces a graphical display of (by default) the current grid viewport tree. It is also possible to display only specific viewports. Each viewport is drawn as a rectangle and the leaf viewports are labelled with the viewport name.

Usage

```
showViewport(vp = NULL, recurse = TRUE, depth = NULL,
             newpage = FALSE, leaves = FALSE,
             col = rgb(0, 0, 1, 0.2), fill = rgb(0, 0, 1, 0.1),
             label = TRUE, nrow = 3, ncol = nrow)
```

Arguments

vp	If NULL, the current viewport tree is displayed. Otherwise, a viewport (or vpList, or vpStack, or vpTree) or a vpPath that specifies which viewport to display.
recurse	Should the children of the specified viewport also be displayed?
depth	Only display viewports at the specified depth (may be a vector of depths).
newpage	Start a new page for the display? Otherwise, the viewports are displayed on top of the current plot.
leaves	Produce a matrix of smaller displays, with each leaf viewport in its own display.
col	The colour used to draw the border of the rectangle for each viewport *and* to draw the label for each viewport. If a vector, then the first colour is used for the top-level viewport, the second colour is used for its children, the third colour for their children, and so on.
fill	The colour used to fill each viewport. May be a vector as per col.
label	Should the viewports be labelled (with the viewport name)?
nrow, ncol	The number of rows and columns when leaves is TRUE. Otherwise ignored.

See Also

viewport and grid.show.viewport

Examples

```
showViewport(viewport(width=.5, height=.5, name="vp"))

grid.newpage()
pushViewport(viewport(width=.5, height=.5, name="vp"))
upViewport()
showViewport(vpPath("vp"))

showViewport(vpStack(viewport(width=.5, height=.5, name="vp1"),
                     viewport(width=.5, height=.5, name="vp2")),
             newpage=TRUE)

showViewport(vpStack(viewport(width=.5, height=.5, name="vp1"),
                     viewport(width=.5, height=.5, name="vp2")),
             fill=rgb(1:0, 0:1, 0, .1),
             newpage=TRUE)
```

---

stringWidth *Create a Unit Describing the Width and Height of a String or Math Expression*

Description

These functions create a unit object describing the width or height of a string.

Usage

```
stringWidth(string)
stringHeight(string)
stringAscent(string)
stringDescent(string)
```

Arguments

string	A character vector or a language object (as used for 'plotmath' calls).

Value

A unit object.

Author(s)

Paul Murrell

See Also

unit and grobWidth

strwidth in the **graphics** package for more details of the typographic concepts behind the computations.

`unit` *Function to Create a Unit Object*

Description

This function creates a unit object — a vector of unit values. A unit value is typically just a single numeric value with an associated unit.

Usage

```
unit(x, units, data=NULL)
is.unit(x)
```

Arguments

`x`	A numeric vector. For `is.unit`, any R object.
`units`	A character vector specifying the units for the corresponding numeric values.
`data`	This argument is used to supply extra information for special `unit` types.

Details

Unit objects allow the user to specify locations and dimensions in a large number of different coordinate systems. All drawing occurs relative to a viewport and the `units` specifies what coordinate system to use within that viewport.

Possible `units` (coordinate systems) are:

`"npc"` Normalised Parent Coordinates (the default). The origin of the viewport is (0, 0) and the viewport has a width and height of 1 unit. For example, (0.5, 0.5) is the centre of the viewport.

`"cm"` Centimetres.

`"inches"` Inches. 1 in = 2.54 cm.

`"mm"` Millimetres. 10 mm = 1 cm.

`"points"` Points. 72.27 pt = 1 in.

`"picas"` Picas. 1 pc = 12 pt.

`"bigpts"` Big Points. 72 bp = 1 in.

`"dida"` Dida. 1157 dd = 1238 pt.

`"cicero"` Cicero. 1 cc = 12 dd.

`"scaledpts"` Scaled Points. 65536 sp = 1 pt.

`"lines"` Lines of text. Locations and dimensions are in terms of multiples of the default text size of the viewport (as specified by the viewport's `fontsize` and `lineheight`).

`"char"` Multiples of nominal font height of the viewport (as specified by the viewport's `fontsize`).

`"native"` Locations and dimensions are relative to the viewport's `xscale` and `yscale`.

"snpc" Square Normalised Parent Coordinates. Same as Normalised Parent Coordinates, except gives the same answer for horizontal and vertical locations/dimensions. It uses the *lesser* of npc-width and npc-height. This is useful for making things which are a proportion of the viewport, but have to be square (or have a fixed aspect ratio).

"strwidth" Multiples of the width of the string specified in the data argument. The font size is determined by the pointsize of the viewport.

"strheight" Multiples of the height of the string specified in the data argument. The font size is determined by the pointsize of the viewport.

"grobwidth" Multiples of the width of the grob specified in the data argument.

"grobheight" Multiples of the height of the grob specified in the data argument.

A number of variations are also allowed for the most common units. For example, it is possible to use "in" or "inch" instead of "inches" and "centimetre" or "centimeter" instead of "cm".

A special units value of "null" is also allowed, but only makes sense when used in specifying widths of columns or heights of rows in grid layouts (see grid.layout).

The data argument must be a list when the unit.length() is greater than 1. For example,

```
unit(rep(1, 3), c("npc", "strwidth", "inches"),
data = list(NULL, "my string", NULL))
```

.

It is possible to subset unit objects in the normal way and to perform subassignment (see the examples), but a special function unit.c is provided for combining unit objects.

Certain arithmetic and summary operations are defined for unit objects. In particular, it is possible to add and subtract unit objects (e.g., unit(1, "npc") - unit(1, "inches")), and to specify the minimum or maximum of a list of unit objects (e.g., min(unit(0.5, "npc"), unit(1, "inches"))).

There is a format method for units, which should respond to the arguments for the default format method, e.g., digits to control the number of significant digits printed for numeric values.

The is.unit() function is a convenience for checking whether x inherits from the "unit" class.

Value

An object of class "unit".

WARNING

There is a special function unit.c for concatenating several unit objects.

The c function will not give the right answer.

There used to be "mylines", "mychar", "mystrwidth", "mystrheight" units. These will still be accepted, but work exactly the same as "lines", "char", "strwidth", "strheight".

Author(s)

Paul Murrell

See Also

unit.c

Examples

```
unit(1, "npc")
unit(1:3/4, "npc")
unit(1:3/4, "npc") + unit(1, "inches")
min(unit(0.5, "npc"), unit(1, "inches"))
unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
       unit(1, "strwidth", "hi there"))
x <- unit(1:5, "npc")
x[2:4]
x[2:4] <- unit(1, "mm")
x
```

---

unit.c *Combine Unit Objects*

Description

This function produces a new unit object by combining the unit objects specified as arguments.

Usage

```
unit.c(..., check = TRUE)
```

Arguments

...	An arbitrary number of unit objects.
check	Should input be checked? If you are certain all arguments are unit objects this can be set to FALSE

Value

An object of class unit.

Author(s)

Paul Murrell

See Also

unit.

unit.length *Length of a Unit Object*

Description

The length of a unit object is defined as the number of unit values in the unit object.

This function has been deprecated in favour of a unit method for the generic `length` function.

Usage

```
unit.length(unit)
```

Arguments

`unit`	A unit object.

Value

An integer value.

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
length(unit(1:3, "npc"))
length(unit(1:3, "npc") + unit(1, "inches"))
length(max(unit(1:3, "npc") + unit(1, "inches")))
length(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4)
length(unit(1:3, "npc") + unit(1, "strwidth", "a")*4)
```

unit.pmin *Parallel Unit Minima and Maxima*

Description

Returns a unit object whose i-th value is the minimum (or maximum) of the i-th values of the arguments.

Usage

```
unit.pmin(...)
unit.pmax(...)
unit.psum(...)
```

Arguments

...	One or more unit objects.

Details

The length of the result is the maximum of the lengths of the arguments; shorter arguments are recycled in the usual manner.

Value

A unit object.

Author(s)

Paul Murrell

Examples

```
max(unit(1:3, "cm"), unit(0.5, "npc"))
unit.pmax(unit(1:3, "cm"), unit(0.5, "npc"))
```

---

unit.rep *Replicate Elements of Unit Objects*

Description

Replicates the units according to the values given in `times` and `length.out`.

This function has been deprecated in favour of a unit method for the generic `rep` function.

Usage

```
unit.rep(x, ...)
```

Arguments

x	An object of class `"unit"`.
...	arguments to be passed to `rep` such as `times` and `length.out`.

Value

An object of class `"unit"`.

Author(s)

Paul Murrell

See Also

rep

Examples

```
rep(unit(1:3, "npc"), 3)
rep(unit(1:3, "npc"), 1:3)
rep(unit(1:3, "npc") + unit(1, "inches"), 3)
rep(max(unit(1:3, "npc") + unit(1, "inches")), 3)
rep(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4, 3)
rep(unit(1:3, "npc") + unit(1, "strwidth", "a")*4, 3)
```

---

unitType *Return the Units of a Unit Object*

Description

This function returns the units of a unit object.

Usage

```
unitType(x, recurse = FALSE)
```

Arguments

x	A unit object.
recurse	Whether to recurse into complex units.

Value

For simple units, this will be just a vector of coordinate systems, like "inches" or "npc".

More complex units that involve an operation on units return an operator, like "sum", "min", or "max".

When recurse = TRUE, the result is always a list and more complex units generate sublists (see the Examples below).

Author(s)

Thomas Lin Pedersen and Paul Murrell

See Also

unit

Examples

```
u <- unit(1:5, c("cm", "mm", "in", "pt", "null"))

unitType(u)
unitType(unit(1, "npc"))
unitType(unit(1:3/4, "npc"))
unitType(unit(1:3/4, "npc") + unit(1, "inches"))
unitType(min(unit(0.5, "npc"), unit(1, "inches")))
unitType(unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
                unit(1, "strwidth", "hi there")))
unitType(min(unit(1, "in"), unit(1, "npc") + unit(1, "mm")))

unitType(u, recurse=TRUE)
unitType(unit(1, "npc"), recurse=TRUE)
unitType(unit(1:3/4, "npc"), recurse=TRUE)
unitType(unit(1:3/4, "npc") + unit(1, "inches"), recurse=TRUE)
unitType(min(unit(0.5, "npc"), unit(1, "inches")), recurse=TRUE)
unitType(unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
                unit(1, "strwidth", "hi there")), recurse=TRUE)
unitType(min(unit(1, "in"), unit(1, "npc") + unit(1, "mm")), recurse=TRUE)
unlist(unitType(min(unit(1, "in"), unit(1, "npc") + unit(1, "mm")),
                recurse=TRUE))
```

---

valid.just *Validate a Justification*

Description

Utility functions for determining whether a justification specification is valid and for resolving a single justification value from a combination of character and numeric values.

Usage

```
valid.just(just)
resolveHJust(just, hjust)
resolveVJust(just, vjust)
```

Arguments

just	A justification either as a character value, e.g., "left", or as a numeric value, e.g., 0.
hjust	A numeric horizontal justification
vjust	A numeric vertical justification

Details

These functions may be useful within a validDetails method when writing a new grob class.

Value

A numeric representation of the justification (e.g., "left" becomes 0, "right" becomes 1, etc, ...). An error is given if the justification is not valid.

Author(s)

Paul Murrell

---

validDetails *Customising grid grob Validation*

Description

This generic hook function is called whenever a grid grob is created or edited via grob, gTree, grid.edit or editGrob. This provides an opportunity for customising the validation of a new class derived from grob (or gTree).

Usage

```
validDetails(x)
```

Arguments

x	A grid grob.

Details

This function is called by grob, gTree, grid.edit and editGrob. A method should be written for classes derived from grob or gTree to validate the values of slots specific to the new class. (e.g., see grid:::validDetails.axis).

Note that the standard slots for grobs and gTrees are automatically validated (e.g., vp, gp slots for grobs and, in addition, children, and childrenvp slots for gTrees) so only slots specific to a new class need to be addressed.

Value

The function MUST return the validated grob.

Author(s)

Paul Murrell

See Also

grid.edit

viewportTransform *Define a Group Transformation*

Description

These functions define the transformation that will be applied when a `grid.define()`d group is `grid.use()`d.

Usage

```
viewportTransform(group, shear=groupShear(), flip=groupFlip(), device=TRUE)
viewportTranslate(group, device=TRUE)
viewportScale(group, device=TRUE)
viewportRotate(group, device=TRUE)
defnTranslate(group, inverse=FALSE, device=TRUE)
defnScale(group, inverse=FALSE)
defnRotate(group, inverse=FALSE, device=TRUE)
useTranslate(inverse=FALSE, device=TRUE)
useScale(inverse=FALSE)
useRotate(inverse=FALSE, device=TRUE)
groupTranslate(dx=0, dy=0)
groupRotate(r=0, device=TRUE)
groupScale(sx=1, sy=1)
groupShear(sx=0, sy=0)
groupFlip(flipX=FALSE, flipY=FALSE)
```

Arguments

`group`	The group that is being transformed.
`inverse`	A logical indicating whether we want the forward or backward transformation.
`shear`	An affine transformation matrix that describes a shear transformation.
`flip`	An affine transformation matrix that describes a scaling inversion.
`dx`, `dy`	The translation to apply.
`r`	The rotation to apply.
`sx`, `sy`	The scaling (or shear) to apply.
`flipX`, `flipY`	Whether to negate the x-scaling or y-scaling (logical).
`device`	A logical indicating whether transformation should be relative to the device or relative to the current viewport.

Details

The `viewport*()` functions are not called directly. They are passed as the `transform` argument to `grid.use`.

The `defn*()` and `use*()` functions are also not called directly, but can be useful to create custom transformation functions. For example, see the source code for `viewportTransform`.

The `group*()` functions generate basic affine transformation matrices and may also be useful to create custom transformation functions. For example, the `groupShear()` function can be used to specify a shear transform to `viewportTransform()`.

It is also possible to define any function that returns a 3x3 matrix (as long as the last column contains 0, 0, and 1) and use it as the `transform` argument to `grid.use`, but the results will probably be device-dependent, and may be *very* difficult to predict. The function will be called with two arguments: `group` and `device`.

Value

An affine transformation matrix.

Author(s)

Paul Murrell

See Also

Grid

Examples

```
## NOTE: on devices without support for groups nothing will be drawn
grid.newpage()
## Define and use group in same viewport
pushViewport(viewport(width=.2, height=.2))
grid.define(circleGrob(gp=gpar(lwd=5)), name="circle")
grid.use("circle")
popViewport()
## Use group in viewport that is translated and scaled
pushViewport(viewport(x=.2, y=.2, width=.1, height=.1))
grid.use("circle")
popViewport()
## Use group in viewport that is translated and scaled
## BUT only make use of the translation
pushViewport(viewport(x=.2, y=.8, width=.1, height=.1))
grid.use("circle", transform=viewportTranslate)
popViewport()
## Use group in viewport that is translated and scaled
## unevenly (distorted)
pushViewport(viewport(x=.8, y=.7, width=.2, height=.4))
grid.use("circle")
popViewport()
```

vpPath *Concatenate Viewport Names*

Description

This function can be used to generate a viewport path for use in `downViewport` or `seekViewport`.

A viewport path is a list of nested viewport names.

Usage

```
vpPath(...)
```

Arguments

...	Character values which are viewport names.

Details

Viewport names must only be unique amongst viewports which share the same parent in the viewport tree.

This function can be used to generate a specification for a viewport that includes the viewport's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A `vpPath` object.

See Also

`viewport`, `pushViewport`, `popViewport`, `downViewport`, `seekViewport`, `upViewport`

Examples

```
vpPath("vp1", "vp2")
```

`widthDetails` *Width and Height of a grid grob*

Description

These generic functions are used to determine the size of grid grobs.

Usage

```
widthDetails(x)
heightDetails(x)
ascentDetails(x)
descentDetails(x)
```

Arguments

`x`	A grid grob.

Details

These functions are called in the calculation of `"grobwidth"` and `"grobheight"` units. Methods should be written for classes derived from grob or gTree where the size of the grob can be determined (see, for example `grid:::widthDetails.frame`).

The ascent of a grob is the height of the grob by default and the descent of a grob is zero by default, except for text grobs where the label is a single character value or expression.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`absolute.size`.

`Working with Viewports`

Maintaining and Navigating the Grid Viewport Tree

Description

Grid maintains a tree of viewports — nested drawing contexts.

These functions provide ways to add or remove viewports and to navigate amongst viewports in the tree.

Usage

```
pushViewport(..., recording=TRUE)
popViewport(n = 1, recording=TRUE)
downViewport(name, strict=FALSE, recording=TRUE)
seekViewport(name, recording=TRUE)
upViewport(n = 1, recording=TRUE)
```

Arguments

`...`	One or more objects of class `"viewport"`.
`n`	An integer value indicating how many viewports to pop or navigate up. The special value `0` indicates to pop or navigate viewports right up to the root viewport.
`name`	A character value to identify a viewport in the tree.
`strict`	A boolean indicating whether the vpPath must be matched exactly.
`recording`	A logical value to indicate whether the viewport operation should be recorded on the Grid display list.

Details

Objects created by the `viewport()` function are only descriptions of a drawing context. A viewport object must be pushed onto the viewport tree before it has any effect on drawing.

The viewport tree always has a single root viewport (created by the system) which corresponds to the entire device (and default graphical parameter settings). Viewports may be added to the tree using `pushViewport()` and removed from the tree using `popViewport()`.

There is only ever one current viewport, which is the current position within the viewport tree. All drawing and viewport operations are relative to the current viewport. When a viewport is pushed it becomes the current viewport. When a viewport is popped, the parent viewport becomes the current viewport. Use `upViewport` to navigate to the parent of the current viewport, without removing the current viewport from the viewport tree. Use `downViewport` to navigate to a viewport further down the viewport tree and `seekViewport` to navigate to a viewport anywhere else in the tree.

If a viewport is pushed and it has the same `name` as a viewport at the same level in the tree, then it replaces the existing viewport in the tree.

Value

`downViewport` returns the number of viewports it went down.

This can be useful for returning to your starting point by doing something like `depth <- downViewport()` then `upViewport(depth)`.

Author(s)

Paul Murrell

See Also

`viewport` and `vpPath`.

Examples

```
# push the same viewport several times
grid.newpage()
vp <- viewport(width=0.5, height=0.5)
pushViewport(vp)
grid.rect(gp=gpar(col="blue"))
grid.text("Quarter of the device",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
pushViewport(vp)
grid.rect(gp=gpar(col="red"))
grid.text("Quarter of the parent viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
popViewport(2)
# push several viewports then navigate amongst them
grid.newpage()
grid.rect(gp=gpar(col="grey"))
grid.text("Top-level viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.7, name="A"))
grid.rect(gp=gpar(col="blue"))
grid.text("1. Push Viewport A",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
grid.rect(gp=gpar(col="red"))
grid.text("2. Push Viewport B (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
if (interactive()) Sys.sleep(1.0)
upViewport(1)
grid.text("3. Up from B to A",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
grid.rect(gp=gpar(col="green"))
```

```
grid.text("4. Push Viewport C (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="green"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.6, name="D"))
grid.rect()
grid.text("5. Push Viewport D (in C)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
upViewport(0)
grid.text("6. Up from D to top-level",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
downViewport("D")
grid.text("7. Down from top-level to D",
  y=unit(1, "npc") - unit(2, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("B")
grid.text("8. Seek from D to B",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="red"))
pushViewport(viewport(width=0.9, height=0.5, name="A"))
grid.rect()
grid.text("9. Push Viewport A (in B)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("A")
grid.text("10. Seek from B to A (in ROOT)",
  y=unit(1, "npc") - unit(3, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
seekViewport(vpPath("B", "A"))
grid.text("11. Seek from\nA (in ROOT)\nto A (in B)")
popViewport(0)
```

`xDetails` *Boundary of a grid grob*

Description

These generic functions are used to determine a location on the boundary of a grid grob.

Usage

```
xDetails(x, theta)
yDetails(x, theta)
```

Arguments

`x`	A grid grob.
`theta`	A numeric angle, in degrees, measured anti-clockwise from the 3 o'clock *or* one of the following character strings: `"north"`, `"east"`, `"west"`, `"south"`.

Details

The location on the grob boundary is determined by taking a line from the centre of the grob at the angle `theta` and intersecting it with the convex hull of the grob (for the basic grob primitives, the centre is determined as half way between the minimum and maximum values in x and y directions).

These functions are called in the calculation of `"grobx"` and `"groby"` units as produced by the `grobX` and `grobY` functions. Methods should be written for classes derived from grob or gTree where the boundary of the grob can be determined.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`grobX`, `grobY`.

`xsplinePoints` *Return the points that would be used to draw an xspline (or a Bezier curve)*

Description

Rather than drawing an xspline (or Bezier curve), this function returns the points that would be used to draw the series of line segments for the xspline. This may be useful to post-process the xspline curve, for example, to clip the curve.

Usage

```
xsplinePoints(x)
bezierPoints(x)
```

Arguments

`x`	An xspline grob, as produced by the `xsplineGrob()` function (or a beziergrob, as produced by the `bezierGrob()` function).

Details

The points returned by this function will only be relevant for the drawing context in force when this function was called.

Value

Depends on how many xsplines would be drawn. If only one, then a list with two components, named x and y, both of which are unit objects (in inches). If several xsplines would be drawn then the result of this function is a list of lists.

Author(s)

Paul Murrell

See Also

xsplineGrob and bezierGrob

Examples

```
grid.newpage()
xsg <- xsplineGrob(c(.1, .1, .9, .9), c(.1, .9, .9, .1), shape=1)
grid.draw(xsg)
trace <- xsplinePoints(xsg)
grid.circle(trace$x, trace$y, default.units="inches", r=unit(.5, "mm"))

grid.newpage()
vp <- viewport(width=.5)
xg <- xsplineGrob(x=c(0, .2, .4, .2, .5, .7, .9, .7),
                  y=c(.5, 1, .5, 0, .5, 1, .5, 0),
                  id=rep(1:2, each=4),
                  shape=1,
                  vp=vp)
grid.draw(xg)
trace <- xsplinePoints(xg)
pushViewport(vp)
invisible(lapply(trace, function(t) grid.lines(t$x, t$y, gp=gpar(col="red"))))
popViewport()

grid.newpage()
bg <- bezierGrob(c(.2, .2, .8, .8), c(.2, .8, .8, .2))
grid.draw(bg)
trace <- bezierPoints(bg)
grid.circle(trace$x, trace$y, default.units="inches", r=unit(.5, "mm"))
```

Chapter 7

The `methods` package

`methods-package` *Formal Methods and Classes*

Description

Formally defined methods and classes for R objects, plus other programming tools, as described in the references.

Details

This package provides the "S4" or "S version 4" approach to methods and classes in a functional language.

For basic use of the techniques, start with Introduction and follow the links there to the key functions for programming, notably `setClass` and `setMethod`.

Some specific topics:

Classes: Creating one, see `setClass`; examining definitions, see `getClassDef` and classRepresentation; inheritance and coercing, see `is` and `as`

Generic functions: Basic programming, see `setGeneric`; the class of objects, see genericFunction; other functions to examine or manipulate them, see GenericFunctions.

S3: Using classes, see `setOldClass`; methods, see Methods_for_S3.

Reference classes: See ReferenceClasses.

Class unions; virtual classes See `setClassUnion`.

These pages will have additional links to related topics.

For a complete list of functions and classes, use `library(help="methods")`.

Author(s)

R Core Team

Maintainer: R Core Team <R-core@r-project.org>

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Chapter 10 has some additional details.)

`.BasicFunsList` *List of Builtin and Special Functions*

Description

A named list providing instructions for turning builtin and special functions into generic functions.

Functions in R that are defined as `.Primitive(<name>)` are not suitable for formal methods, because they lack the basic reflectance property. You can't find the argument list for these functions by examining the function object itself.

Future versions of R may fix this by attaching a formal argument list to the corresponding function. While generally the names of arguments are not checked by the internal code implementing the function, the number of arguments frequently is.

In any case, some definition of a formal argument list is needed if users are to define methods for these functions. In particular, if methods are to be merged from multiple packages, the different sets of methods need to agree on the formal arguments.

In the absence of reflectance, this list provides the relevant information via a dummy function associated with each of the known specials for which methods are allowed.

At the same, the list flags those specials for which methods are meaningless (e.g., `for`) or just a very bad idea (e.g., `.Primitive`).

A generic function created via `setMethod`, for example, for one of these special functions will have the argument list from `.BasicFunsList`. If no entry exists, the argument list `(x, ...)` is assumed.

`as` *Force an Object to Belong to a Class*

Description

Coerce an object to a given class.

Usage

```
as(object, Class, strict=TRUE, ext)

as(object, Class) <- value
```

Arguments

object	any R object.
Class	the name of the class to which object should be coerced.
strict	logical flag. If TRUE, the returned object must be strictly from the target class (unless that class is a virtual class, in which case the object will be from the closest actual class, in particular the original object, if that class extends the virtual class directly). If strict = FALSE, any simple extension of the target class will be returned, without further change. A simple extension is, roughly, one that just adds slots to an existing class.
value	The value to use to modify object (see the discussion below). You should supply an object with class Class; some coercion is done, but you're unwise to rely on it.
ext	an optional object defining how Class is extended by the class of the object (as returned by possibleExtends). This argument is used internally; do not use it directly.

Description

as(object) returns the version of this object coerced to be the given Class. When used in the replacement form on the left of an assignment, the portion of the object corresponding to Class is replaced by value.

The operation of as() in either form depends on the definition of coerce methods. Methods are defined automatically when the two classes are related by inheritance; that is, when one of the classes is a subclass of the other.

Coerce methods are also predefined for basic classes (including all the types of vectors, functions and a few others).

Beyond these two sources of methods, further methods are defined by calls to the setAs function. See that documentation also for details of how coerce methods work. Use showMethods(coerce) for a list of all currently defined methods, as in the example below.

Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, as(x, "numeric") uses the existing as.numeric function. These and all other existing methods can be listed as shown in the example.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

If you think of using try(as(x, cl)), consider canCoerce(x, cl) instead.

Examples

```
## Show all the existing methods for as()
showMethods("coerce")
```

BasicClasses *Classes Corresponding to Basic Data Types*

Description

Formal classes exist corresponding to the basic R object types, allowing these types to be used in method signatures, as slots in class definitions, and to be extended by new classes.

Usage

```
### The following are all basic vector classes.
### They can appear as class names in method signatures,
### in calls to as(), is(), and new().
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"
"raw"

### the class
"vector"
### is a virtual class, extended by all the above

### the class
"S4"
### is an object type for S4 objects that do not extend
### any of the basic vector classes.  It is a virtual class.

### The following are additional basic classes
"NULL"     #  NULL objects
"function" #  function objects, including primitives
"externalptr" # raw external pointers for use in C code

"ANY"  # virtual classes used by the methods package itself
"VIRTUAL"
"missing"
```

```
"namedList" # the alternative to "list" that preserves
            # the names attribute
```

Objects from the Classes

If a class is not virtual (see section in Classes_Details), objects can be created by calls of the form new(Class, ...), where Class is the quoted class name, and the remaining arguments if any are objects to be interpreted as vectors of this class. Multiple arguments will be concatenated.

The class "expression" is slightly odd, in that the ... arguments will *not* be evaluated; therefore, don't enclose them in a call to quote().

Note that class "list" is a pure vector. Although lists with names go back to the earliest versions of S, they are an extension of the vector concept in that they have an attribute (which can now be a slot) and which is either NULL or a character vector of the same length as the vector. If you want to guarantee that list names are preserved, use class "namedList", rather than "list". Objects from this class must have a names attribute, corresponding to slot "names", of type "character". Internally, R treats names for lists specially, which makes it impractical to have the corresponding slot in class "namedList" be a union of character names and NULL.

Classes and Types

The basic classes include classes for the basic R types. Note that objects of these types will not usually be S4 objects (isS4 will return FALSE), although objects from classes that contain the basic class will be S4 objects, still with the same type. The type as returned by typeof will sometimes differ from the class, either just from a choice of terminology (type "symbol" and class "name", for example) or because there is not a one-to-one correspondence between class and type (most of the classes that inherit from class "language" have type "language", for example).

Extends

The vector classes extend "vector", directly.

Methods

coerce Methods are defined to coerce arbitrary objects to the vector classes, by calling the corresponding basic function, for example, as(x, "numeric") calls as.numeric(x).

---

callGeneric *Call the Current Generic Function from a Method*

Description

A call to callGeneric can only appear inside a method definition. It then results in a call to the current generic function. The value of that call is the value of callGeneric. While it can be called from any method, it is useful and typically used in methods for group generic functions.

Usage

```
callGeneric(...)
```

Arguments

...	Optionally, the arguments to the function in its next call. If no arguments are included in the call to `callGeneric`, the effect is to call the function with the current arguments. See the detailed description for what this really means.

Details

The name and package of the current generic function is stored in the environment of the method definition object. This name is looked up and the corresponding function called.

The statement that passing no arguments to `callGeneric` causes the generic function to be called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that `"missing"` is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the generated call equivalent to `x = x`. In effect, this means that the generic function sees the same actual arguments, but arguments are evaluated only once.

Using `callGeneric` with no arguments is prone to creating infinite recursion, unless one of the arguments in the signature has been modified in the current method so that a different method is selected.

Value

The value returned by the new call.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.4 for some details.)

See Also

GroupGenericFunctions for other information about group generic functions; Methods_Details for the general behavior of method dispatch

Examples

```
## the method for group generic function Ops
## for signature( e1="structure", e2="vector")
function (e1, e2)
{
    value <- callGeneric(e1@.Data, e2)
    if (length(value) == length(e1)) {
        e1@.Data <- value
        e1
    }
    else value
}
```

```
## For more examples
## Not run:
showMethods("Ops", includeDefs = TRUE)

## End(Not run)
```

`callNextMethod` *Call an Inherited Method*

Description

A call to `callNextMethod` can only appear inside a method definition. It then results in a call to the first inherited method after the current method, with the arguments to the current method passed down to the next method. The value of that method call is the value of `callNextMethod`.

Usage

```
callNextMethod(...)
```

Arguments

`...`	Optionally, the arguments to the function in its next call (but note that the dispatch is as in the detailed description below; the arguments have no effect on selecting the next method.) If no arguments are included in the call to `callNextMethod`, the effect is to call the method with the current arguments. See the detailed description for what this really means. Calling with no arguments is often the natural way to use `callNextMethod`; see the examples.

Details

The 'next' method (i.e., the first inherited method) is defined to be that method which *would* have been called if the current method did not exist. This is more-or-less literally what happens: The current method (to be precise, the method with signature given by the `defined` slot of the method from which `callNextMethod` is called) is deleted from a copy of the methods for the current generic, and `selectMethod` is called to find the next method (the result is cached in the method object where the call occurred, so the search typically happens only once per session per combination of argument classes).

The next method is defined from the *signature* of the current method, not from the actual classes of the arguments. In particular, modifying any of the arguments has no effect on the selection. As a result, the selected next method can be called with invalid arguments if the calling function assigns objects of a different class before the `callNextMethod()` call. Be careful of any assignments to such arguments.

It is possible for the selection of the next method to be ambiguous, even though the original set of methods was consistent. See the section "Ambiguous Selection".

The statement that the method is called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that `"missing"` is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the next method call equivalent to `x = x`. In effect, this means that the next method sees the same actual arguments, but arguments are evaluated only once.

Value

The value returned by the selected method.

Ambiguous Selection

There are two fairly common situations in which the choice of a next method is ambiguous, even when the original set of methods uniquely defines all method selection unambiguously. In these situations, `callNextMethod()` should be replaced, either by a call to a specific function or by recalling the generic with different arguments.

The most likely situation arises with methods for binary operators, typically through one of the group generic functions. See the example for class `"rnum"` below. Examples of this sort usually require three methods: two for the case that the first or the second argument comes from the class, and a third for the case that both arguments come from the class. If that last method uses `callNextMethod`, the other two methods are equally valid. The ambiguity is exactly the same that required defining the two-argument method in the first place.

In fact, the two possibilities are equally valid conceptually as well as formally. As in the example below, the logic of the application usually requires selecting a computation explicitly or else calling the generic function with modified arguments to select an appropriate method.

The other likely source of ambiguity arises from a class that inherits directly from more than one other class (a "mixin" in standard terminology). If the generic has methods corresponding to both superclasses, a method for the current class is again needed to resolve ambiguity. Using `callNextMethod` will again reimpose the ambiguity. Again, some explicit choice has to be made in the calling method instead.

These ambiguities are not the result of bad design, but they do require workarounds. Other ambiguities usually reflect inconsistencies in the tree of inheritances, such as a class appearing in more than one place among the superclasses. Such cases should be rare, but with the independent definition of classes in multiple packages, they can't be ruled out.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

`callGeneric` to call the generic function with the current dispatch rules (typically for a group generic function); Methods_Details for the general behavior of method dispatch.

Examples

```
## callNextMethod() used for the Math, Math2 group generic functions

## A class to automatically round numeric results to "d" digits
```

```
rnum <- setClass("rnum", slots = c(d = "integer"), contains = "numeric")

## Math functions operate on the rounded numbers, return a plain
## vector.  The next method will always be the default, usually a primitive.
setMethod("Math", "rnum",
          function(x)
              callNextMethod(round(as.numeric(x), x@d)))
setMethod("Math2", "rnum",
          function(x, digits)
              callNextMethod(round(as.numeric(x), x@d), digits))

## Examples of callNextMethod with two arguments in the signature.

## For arithmetic and one rnum with anything, callNextMethod with no arguments
## round the full accuracy result, and return as plain vector
setMethod("Arith", c(e1 ="rnum"),
          function(e1, e2)
              as.numeric(round(callNextMethod(), e1@d)))
setMethod("Arith", c(e2 ="rnum"),
          function(e1, e2)
              as.numeric(round(callNextMethod(), e2@d)))

## A method for BOTH arguments from "rnum" would be ambiguous
## for callNextMethod(): the two methods above are equally valid.
## The method chooses the smaller number of digits,
## and then calls the generic function, postponing the method selection
## until it's not ambiguous.
setMethod("Arith", c(e1 ="rnum", e2 = "rnum"),
          function(e1, e2) {
              if(e1@d <= e2@d)
                  callGeneric(e1, as.numeric(e2))
              else
                  callGeneric(as.numeric(e1), e2)
          })

## For comparisons, callNextMethod with the rounded arguments
setMethod("Compare", c(e1 = "rnum"),
          function(e1, e2)
              callNextMethod(round(e1, e1@d), round(e2, e1@d)))
setMethod("Compare", c(e2 = "rnum"),
          function(e1, e2)
              callNextMethod(round(e1, e2@d), round(e2, e2@d)))

## similarly to the Arith case, the method for two "rnum" objects
## can not unambiguously use callNextMethod().  Instead, we rely on
## The rnum() method inhertited from Math2 to return plain vectors.
setMethod("Compare", c(e1 ="rnum", e2 = "rnum"),
          function(e1, e2) {
              d <- min(e1@d, e2@d)
              callGeneric(round(e1, d), round(e2, d))
          })
```

```
set.seed(867)

x1 <- rnum(10*runif(5), d=1L)
x2 <- rnum(10*runif(5), d=2L)

x1+1
x2*2
x1-x2

## Simple examples to illustrate callNextMethod with and without arguments
B0 <- setClass("B0", slots = c(s0 = "numeric"))

## and a function to illustrate callNextMethod

f <- function(x, text = "default") {
    str(x) # print a summary
    paste(text, ":", class(x))
}

setGeneric("f")
setMethod("f", "B0", function(x, text = "B0") {
    cat("B0 method called with s0 =", x@s0, "\n")
    callNextMethod()
})

b0 <- B0(s0 = 1)

## call f() with 2 arguments: callNextMethod passes both to the default method
f(b0, "first test")

## call f() with 1 argument:  the default "B0" is not passed by callNextMethod
f(b0)

## Now, a class that extends B0, with no methods for f()
B1 <- setClass("B1", slots = c(s1 = "character"), contains = "B0")
b1 <- B1(s0 = 2, s1 = "Testing B1")

## the two cases work as before, by inheriting the "B0" method

f(b1, b1@s1)

f(b1)

B2 <- setClass("B2", contains = "B1")

## And, a method for "B2" that calls with explicit arguments.
## Note that the method selection in callNextMethod
## uses the class of the *argument* to consistently select the "B0" method

setMethod("f", "B2", function(x, text = "B1 method") {
```

```
    y <- B1(s0 = -x@s0, s1 ="Modified x")
    callNextMethod(y, text)
})

b2 <- B2(s1 = "Testing B2", s0 = 10)

f(b2, b2@s1)

f(b2)


## Be careful:  the argument passed must be legal for the method selected
## Although the argument here is numeric, it's still the "B0" method that's called
setMethod("f", "B2", function(x, text = "B1 method") {
    callNextMethod(x@s0, text)
})

##  Now the call will cause an error:

tryCatch(f(b2), error = function(e) cat(e$message,"\n"))
```

---

canCoerce *Can an Object be Coerced to a Certain S4 Class?*

Description

Test if an object can be coerced to a given S4 class. Maybe useful inside `if()` to ensure that calling `as(object, Class)` will find a method.

Usage

```
canCoerce(object, Class)
```

Arguments

`object`	any R object, typically of a formal S4 class.
`Class`	an S4 class (see `isClass`).

Value

a scalar logical, `TRUE` if there is a `coerce` method (as defined by e.g. `setAs`) for the signature `(from = class(object), to = Class)`.

See Also

`as`, `setAs`, `selectMethod`, `setClass`,

Examples

```
m <- matrix(pi, 2,3)
canCoerce(m, "numeric") # TRUE
canCoerce(m, "array")   # TRUE
```

---

cbind2 *Combine two Objects by Columns or Rows*

Description

Combine two matrix-like R objects by columns (cbind2) or rows (rbind2). These are (S4) generic functions with default methods.

Usage

```
cbind2(x, y, ...)
rbind2(x, y, ...)
```

Arguments

x	any R object, typically matrix-like.
y	any R object, typically similar to x, or missing completely.
...	optional arguments for methods.

Details

The main use of cbind2 (rbind2) is to be called recursively by cbind() (rbind()) when both of these requirements are met:

- There is at least one argument that is an S4 object, and
- S3 dispatch fails (see the Dispatch section under cbind).

The methods on cbind2 and rbind2 effectively define the type promotion policy when combining a heterogeneous set of arguments. The homogeneous case, where all objects derive from some S4 class, can be handled via S4 dispatch on the ... argument via an externally defined S4 cbind (rbind) generic.

Since (for legacy reasons) S3 dispatch is attempted first, it is generally a good idea to additionally define an S3 method on cbind (rbind) for the S4 class. The S3 method will be invoked when the arguments include objects of the S4 class, along with arguments of classes for which no S3 method exists. Also, in case there is an argument that selects a different S3 method (like the one for data.frame), this S3 method serves to introduce an ambiguity in dispatch that triggers the recursive fallback to cbind2 (rbind2). Otherwise, the other S3 method would be called, which may not be appropriate.

Value

A matrix (or matrix like object) combining the columns (or rows) of `x` and `y`. Note that methods must construct `colnames` and `rownames` from the corresponding column and row names of `x` and `y` (but not from deparsing argument names such as in `cbind(..., deparse.level = d)` for $d \geq 1$).

Methods

`signature(x = "ANY", y = "ANY")` the default method using R's internal code.

`signature(x = "ANY", y = "missing")` the default method for one argument using R's internal code.

See Also

`cbind`, `rbind`; further, `cBind`, `rBind` in the **Matrix** package.

Examples

```
cbind2(1:3, 4)
m <- matrix(3:8, 2,3, dimnames=list(c("a","b"), LETTERS[1:3]))
cbind2(1:2, m) # keeps dimnames from m

## rbind() and cbind() now make use of rbind2()/cbind2() methods
setClass("Num", contains="numeric")
setMethod("cbind2", c("Num", "missing"),
          function(x,y, ...) { cat("Num-miss--meth\n"); as.matrix(x)})
setMethod("cbind2", c("Num","ANY"), function(x,y, ...) {
    cat("Num-A.--method\n") ; cbind(getDataPart(x), y, ...) })
setMethod("cbind2", c("ANY","Num"), function(x,y, ...) {
    cat("A.-Num--method\n") ; cbind(x, getDataPart(y), ...) })

a <- new("Num", 1:3)
trace("cbind2")
cbind(a)
cbind(a, four=4, 7:9)# calling cbind2() twice

cbind(m,a, ch=c("D","E"), a*3)
cbind(1,a, m) # ok with a warning
untrace("cbind2")
```

`Classes` *S4 Class Documentation*

Description

You have navigated to an old link to documentation of S4 classes.

For basic use of classes and methods, see Introduction; to create new class definitions, see `setClass`; for technical details on S4 classes, see Classes_Details.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`classesToAM` *Compute an Adjacency Matrix for Superclasses of Class Definitions*

Description

Given a vector of class names or a list of class definitions, the function returns an adjacency matrix of the superclasses of these classes; that is, a matrix with class names as the row and column names and with element [i, j] being 1 if the class in column j is a direct superclass of the class in row i, and 0 otherwise.

The matrix has the information implied by the `contains` slot of the class definitions, but in a form that is often more convenient for further analysis; for example, an adjacency matrix is used in packages and other software to construct graph representations of relationships.

Usage

```
classesToAM(classes, includeSubclasses = FALSE,
       abbreviate = 2)
```

Arguments

`classes`	Either a character vector of class names or a list, whose elements can be either class names or class definitions. The list is convenient, for example, to include the package slot for the class name. See the examples.
`includeSubclasses`	A logical flag; if `TRUE`, then the matrix will include all the known subclasses of the specified classes as well as the superclasses. The argument can also be a logical vector of the same length as `classes`, to include subclasses for some but not all the classes.
`abbreviate`	Control of the abbreviation of the row and/or column labels of the matrix returned: values 0, 1, 2, or 3 abbreviate neither, rows, columns or both. The default, 2, is useful for printing the matrix, since class names tend to be more than one character long, making for spread-out printing. Values of 0 or 3 would be appropriate for making a graph (3 avoids the tendency of some graph plotting software to produce labels in minuscule font size).

Details

For each of the classes, the calculation gets all the superclass names from the class definition, and finds the edges in those classes' definitions; that is, all the superclasses at distance 1. The corresponding elements of the adjacency matrix are set to 1.

The adjacency matrices for the individual class definitions are merged. Note two possible kinds of inconsistency, neither of which should cause problems except possibly with identically named

classes from different packages. Edges are computed from each superclass definition, so that information overrides a possible inference from extension elements with distance > 1 (and it should). When matrices from successive classes in the argument are merged, the computations do not currently check for inconsistencies—this is the area where possible multiple classes with the same name could cause confusion. A later revision may include consistency checks.

Value

As described, a matrix with entries 0 or 1, non-zero values indicating that the class corresponding to the column is a direct superclass of the class corresponding to the row. The row and column names are the class names (without package slot).

See Also

`extends` and classRepresentation for the underlying information from the class definition.

Examples

```
## the super- and subclasses of "standardGeneric"
## and "derivedDefaultMethod"
am <- classesToAM(list(class(show), class(getMethod(show))), TRUE)
am

## Not run:
## the following function depends on the Bioconductor package Rgraphviz
plotInheritance <- function(classes, subclasses = FALSE, ...) {
    if(!require("Rgraphviz", quietly=TRUE))
      stop("Only implemented if Rgraphviz is available")
    mm <- classesToAM(classes, subclasses)
    classes <- rownames(mm); rownames(mm) <- colnames(mm)
    graph <-  new("graphAM", mm, "directed", ...)
    plot(graph)
    cat("Key:\n", paste(abbreviate(classes), " = ", classes, ", ",
        sep = ""),  sep = "", fill = TRUE)
    invisible(graph)
}

## The plot of the class inheritance of the package "graph"
require(graph)
plotInheritance(getClasses("package:graph"))


## End(Not run)
```

`Classes_Details` *Class Definitions*

Description

Class definitions are objects that contain the formal definition of a class of R objects, usually referred to as an S4 class, to distinguish them from the informal S3 classes. This document gives an overview of S4 classes; for details of the class representation objects, see help for the class `classRepresentation`.

Metadata Information

When a class is defined, an object is stored that contains the information about that class. The object, known as the *metadata* defining the class, is not stored under the name of the class (to allow programmers to write generating functions of that name), but under a specially constructed name. To examine the class definition, call `getClass`. The information in the metadata object includes:

Slots: The data contained in an object from an S4 class is defined by the *slots* in the class definition.

Each slot in an object is a component of the object; like components (that is, elements) of a list, these may be extracted and set, using the function `slot()` or more often the operator `"@"`. However, they differ from list components in important ways. First, slots can only be referred to by name, not by position, and there is no partial matching of names as with list elements.

All the objects from a particular class have the same set of slot names; specifically, the slot names that are contained in the class definition. Each slot in each object always is an object of the class specified for this slot in the definition of the current class. The word "is" corresponds to the R function of the same name (`is`), meaning that the class of the object in the slot must be the same as the class specified in the definition, or some class that extends the one in the definition (a *subclass*).

A special slot name, `.Data`, stands for the 'data part' of the object. An object from a class with a data part is defined by specifying that the class contains one of the R object types or one of the special pseudo-classes, `matrix` or `array`, usually because the definition of the class, or of one of its superclasses, has included the type or pseudo-class in its `contains` argument. A second special slot name, `.xData`, is used to enable inheritance from abnormal types such as `"environment"` See the section on inheriting from non-S4 classes for details on the representation and for the behavior of S3 methods with objects from these classes.

Some slot names correspond to attributes used in old-style S3 objects and in R objects without an explicit class, for example, the `names` attribute. If you define a class for which that attribute will be set, such as a subclass of named vectors, you should include `"names"` as a slot. See the definition of class `"namedList"` for an example. Using the `names()` assignment to set such names will generate a warning if there is no names slot and an error if the object in question is not a vector type. A slot called `"names"` can be used anywhere, but only if it is assigned as a slot, not via the default `names()` assignment.

Superclasses: The definition of a class includes the *superclasses* —the classes that this class extends. A class `Fancy`, say, extends a class `Simple` if an object from the `Fancy` class has all the capabilities of the `Simple` class (and probably some more as well). In particular, and very usefully, any method defined to work for a `Simple` object can be applied to a `Fancy` object as well.

This relationship is expressed equivalently by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The direct superclasses of a class are those superclasses explicitly defined. Direct superclasses can be defined in three ways. Most commonly, the superclasses are listed in the `contains=`

argument in the call to setClass that creates the subclass. In this case the subclass will contain all the slots of the superclass, and the relation between the class is called *simple*, as it in fact is. Superclasses can also be defined explicitly by a call to setIs; in this case, the relation requires methods to be specified to go from subclass to superclass. Thirdly, a class union is a superclass of all the members of the union. In this case too the relation is simple, but notice that the relation is defined when the superclass is created, not when the subclass is created as with the contains= mechanism.

The definition of a superclass will also potentially contain its own direct superclasses. These are considered (and shown) as superclasses at distance 2 from the original class; their direct superclasses are at distance 3, and so on. All these are legitimate superclasses for purposes such as method selection.

When superclasses are defined by including the names of superclasses in the contains= argument to setClass, an object from the class will have all the slots defined for its own class *and* all the slots defined for all its superclasses as well.

The information about the relation between a class and a particular superclass is encoded as an object of class SClassExtension. A list of such objects for the superclasses (and sometimes for the subclasses) is included in the metadata object defining the class. If you need to compute with these objects (for example, to compare the distances), call the function extends with argument fullInfo=TRUE.

Prototype: The objects from a class created by a call to new are defined by the *prototype* object for the class and by additional arguments in the call to new, which are passed to a method for that class for the function initialize.

Each class representation object contains a prototype object for the class (although for a virtual class the prototype may be NULL). The prototype object must have values for all the slots of the class. By default, these are the prototypes of the corresponding slot classes. However, the definition of the class can specify any valid object for any of the slots.

Basic classes

There are a number of 'basic' classes, corresponding to the ordinary kinds of data occurring in R. For example, "numeric" is a class corresponding to numeric vectors. The other vector basic classes are "logical", "integer", "complex", "character", "raw", "list" and "expression". The prototypes for the vector classes are vectors of length 0 of the corresponding type. Notice that basic classes are unusual in that the prototype object is from the class itself.

In addition to the vector classes there are also basic classes corresponding to objects in the language, such as "function" and "call". These classes are subclasses of the virtual class "language". Finally, there are object types and corresponding basic classes for "abnormal" objects, such as "environment" and "externalptr". These objects do not follow the functional behavior of the language; in particular, they are not copied and so cannot have attributes or slots defined locally.

All these classes can be used as slots or as superclasses for any other class definitions, although they do not themselves come with an explicit class. For the abnormal object types, a special mechanism is used to enable inheritance as described below.

Inheriting from non-S4 Classes

A class definition can extend classes other than regular S4 classes, usually by specifying them in the contains= argument to setClass. Three groups of such classes behave distinctly:

1. S3 classes, which must have been registered by a previous call to setOldClass (you can check that this has been done by calling getClass, which should return a class that extends oldClass);
2. One of the R object types, typically a vector type, which then defines the type of the S4 objects, but also a type such as environment that can not be used directly as a type for an S4 object. See below.
3. One of the pseudo-classes matrix and array, implying objects with arbitrary vector types plus the dim and dimnames attributes.

This section describes the approach to combining S4 computations with older S3 computations by using such classes as superclasses. The design goal is to allow the S4 class to inherit S3 methods and default computations in as consistent a form as possible.

As part of a general effort to make the S4 and S3 code in R more consistent, when objects from an S4 class are used as the first argument to a non-default S3 method, either for an S3 generic function (one that calls UseMethod) or for one of the primitive functions that dispatches S3 methods, an effort is made to provide a valid object for that method. In particular, if the S4 class extends an S3 class or matrix or array, and there is an S3 method matching one of these classes, the S4 object will be coerced to a valid S3 object, to the extent that is possible given that there is no formal definition of an S3 class.

For example, suppose "myFrame" is an S4 class that includes the S3 class "data.frame" in the contains= argument to setClass. If an object from this S4 class is passed to a function, say as.matrix, that has an S3 method for "data.frame", the internal code for UseMethod will convert the object to a data frame; in particular, to an S3 object whose class attribute will be the vector corresponding to the S3 class (possibly containing multiple class names). Similarly for an S4 object inheriting from "matrix" or "array", the S4 object will be converted to a valid S3 matrix or array.

Note that the conversion is *not* applied when an S4 object is passed to the default S3 method. Some S3 generics attempt to deal with general objects, including S4 objects. Also, no transformation is applied to S4 objects that do not correspond to a selected S3 method; in particular, to objects from a class that does not contain either an S3 class or one of the basic types. See asS4 for the transformation details.

In addition to explicit S3 generic functions, S3 methods are defined for a variety of operators and functions implemented as primitives. These methods are dispatched by some internal C code that operates partly through the same code as real S3 generic functions and partly via special considerations (for example, both arguments to a binary operator are examined when looking for methods). The same mechanism for adapting S4 objects to S3 methods has been applied to these computations as well, with a few exceptions such as generating an error if an S4 object that does not extend an appropriate S3 class or type is passed to a binary operator.

The remainder of this section discusses the mechanisms for inheriting from basic object types. See matrix or array for inhering from the matrix and array pseudo-classes, or from time-series. For the corresponding details for inheritance from S3 classes, see setOldClass.

An object from a class that directly and simply contains one of the basic object types in R, has implicitly a corresponding .Data slot of that type, allowing computations to extract or replace the data part while leaving other slots unchanged. If the type is one that can accept attributes and is duplicated normally, the inheritance also determines the type of the object; if the class definition has a .Data slot corresponding to a normal type, the class of the slot determines the type of the object (that is, the value of typeof(x)). For such classes, .Data is a pseudo-slot; that is, extracting

or setting it modifies the non-slot data in the object. The functions getDataPart and setDataPart are a cleaner, but essentially equivalent way to deal with the data part.

Extending a basic type this way allows objects to use old-style code for the corresponding type as well as S4 methods. Any basic type can be used for .Data, but a few types are treated differently because they do not behave like ordinary objects; for example, "NULL", environments, and external pointers. Classes extend these types by having a slot, .xData, itself inherited from an internally defined S4 class. This slot actually contains an object of the inherited type, to protect computations from the reference semantics of the type. Coercing to the nonstandard object type then requires an actual computation, rather than the "simple" inclusion for other types and classes. The intent is that programmers will not need to take account of the mechanism, but one implication is that you should *not* explicitly use the type of an S4 object to detect inheritance from an arbitrary object type. Use is and similar functions instead.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Methods_Details for analogous discussion of methods, setClass for details of specifying class definitions, is, as, new, slot

---

className *Class names including the corresponding package*

Description

The function className() generates a valid references to a class, including the name of the package containing the class definition. The object returned, from class "className", is the unambiguous way to refer to a class, for example when calling setMethod, just in case multiple definitions of the class exist.

Function "multipleClasses" returns information about multiple definitions of classes with the same name from different packages.

Usage

```
className(class, package)

multipleClasses(details = FALSE)
```

Arguments

class, package	The character string name of a class and, optionally, of the package to which it belongs. If argument package is missing and the class argument has a package slot, that is used (in particular, passing in an object from class "className" returns itself in this case, but changes the package slot if the second argument is supplied).

If there is no package argument or slot, a definition for the class must exist and will be used to define the package. If there are multiple definitions, one will be chosen and a warning printed giving the other possibilities.

`details` If `FALSE`, the default, `multipleClasses()` returns a character vector of those classes currently known with multiple definitions.

If `TRUE`, a named list of those class definitions is returned. Each element of the list is itself a list of the corresponding class definitions, with the package names as the names of the list. Note that identical class definitions will not be considered "multiple" definitions (see the discussion of the details below).

Details

The table of class definitions used internally can maintain multiple definitions for classes with the same name but coming from different packages. If identical class definitions are encountered, only one class definition is kept; this occurs most often with S3 classes that have been specified in calls to `setOldClass`. For true classes, multiple class definitions are unavoidable in general if two packages happen to have used the same name, independently.

Overriding a class definition in another package with the same name deliberately is usually a bad idea. Although R attempts to keep and use the two definitions (as of version 2.14.0), ambiguities are always possible. It is more sensible to define a new class that extends an existing class but has a different name.

Value

A call to `className()` returns an object from class `"className"`.

A call to `multipleClasses()` returns either a character vector or a named list of class definitions. In either case, testing the length of the returned value for being greater than `0` is a check for the existence of multiply defined classes.

Objects from the Class

The class `"className"` extends `"character"` and has a slot `"package"`, also of class `"character"`.

Examples

```
## Not run:
className("vector") # will be found, from package "methods"
className("vector", "magic") # OK, even though the class doesn't exist


className("An unknown class") # Will cause an error

## End(Not run)
```

`classRepresentation-class` *Class Objects*

Description

These are the objects that hold the definition of classes of objects. They are constructed and stored as meta-data by calls to the function `setClass`. Don't manipulate them directly, except perhaps to look at individual slots.

Details

Class definitions are stored as metadata in various packages. Additional metadata supplies information on inheritance (the result of calls to `setIs`). Inheritance information implied by the class definition itself (because the class contains one or more other classes) is also constructed automatically.

When a class is to be used in an R session, this information is assembled to complete the class definition. The completion is a second object of class `"classRepresentation"`, cached for the session or until something happens to change the information. A call to `getClass` returns the completed definition of a class; a call to `getClassDef` returns the stored definition (uncompleted).

In particular, completion fills in the upward- and downward-pointing inheritance information for the class, in slots `contains` and `subclasses` respectively. It's in principle important to note that this information can depend on which packages are installed, since these may define additional subclasses or superclasses.

Slots

`slots`: A named list of the slots in this class; the elements of the list are the classes to which the slots must belong (or extend), and the names of the list gives the corresponding slot names.

`contains`: A named list of the classes this class 'contains'; the elements of the list are objects of `SClassExtension`. The list may be only the direct extensions or all the currently known extensions (see the details).

`virtual`: Logical flag, set to `TRUE` if this is a virtual class.

`prototype`: The object that represents the standard prototype for this class; i.e., the data and slots returned by a call to `new` for this class with no special arguments. Don't mess with the prototype object directly.

`validity`: Optionally, a function to be used to test the validity of objects from this class. See `validObject`.

`access`: Access control information. Not currently used.

`className`: The character string name of the class.

`package`: The character string name of the package to which the class belongs. Nearly always the package on which the metadata for the class is stored, but in operations such as constructing inheritance information, the internal package name rules.

`subclasses`: A named list of the classes known to extend this class'; the elements of the list are objects of class `SClassExtension`. The list is currently only filled in when completing the class definition (see the details).

`versionKey`: Object of class `"externalptr"`; eventually will perhaps hold some versioning information, but not currently used.

`sealed`: Object of class `"logical"`; is this class sealed? If so, no modifications are allowed.

See Also

See function `setClass` to supply the information in the class definition. See Classes_Details for a more basic discussion of class information.

`Documentation` *Using and Creating On-line Documentation for Classes and Methods*

Description

Special documentation can be supplied to describe the classes and methods that are created by the software in the methods package. Techniques to access this documentation and to create it in R help files are described here.

Getting documentation on classes and methods

You can ask for on-line help for class definitions, for specific methods for a generic function, and for general discussion of methods for a generic function. These requests use the ? operator (see `help` for a general description of the operator). Of course, you are at the mercy of the implementer as to whether there *is* any documentation on the corresponding topics.

Documentation on a class uses the argument `class` on the left of the ?, and the name of the class on the right; for example,

```
class ? genericFunction
```

to ask for documentation on the class `"genericFunction"`.

When you want documentation for the methods defined for a particular function, you can ask either for a general discussion of the methods or for documentation of a particular method (that is, the method that would be selected for a particular set of actual arguments).

Overall methods documentation is requested by calling the ? operator with `methods` as the left-side argument and the name of the function as the right-side argument. For example,

```
methods ? initialize
```

asks for documentation on the methods for the `initialize` function.

Asking for documentation on a particular method is done by giving a function call expression as the right-hand argument to the `"?"` operator. There are two forms, depending on whether you prefer to give the class names for the arguments or expressions that you intend to use in the actual call.

If you planned to evaluate a function call, say `myFun(x, sqrt(wt))` and wanted to find out something about the method that would be used for this call, put the call on the right of the `"?"` operator:

```
?myFun(x, sqrt(wt))
```

A method will be selected, as it would be for the call itself, and documentation for that method will be requested. If `myFun` is not a generic function, ordinary documentation for the function will be requested.

If you know the actual classes for which you would like method documentation, you can supply these explicitly in place of the argument expressions. In the example above, if you want method documentation for the first argument having class `"maybeNumber"` and the second `"logical"`, call the `"?"` operator, this time with a left-side argument `method`, and with a function call on the right using the class names as arguments:

```
method ? myFun("maybeNumber", "logical")
```

Once again, a method will be selected, this time corresponding to the specified classes, and method documentation will be requested. This version only works with generic functions.

The two forms each have advantages. The version with actual arguments doesn't require you to figure out (or guess at) the classes of the arguments. On the other hand, evaluating the arguments may take some time, depending on the example. The version with class names does require you to pick classes, but it's otherwise unambiguous. It has a subtler advantage, in that the classes supplied may be virtual classes, in which case no actual argument will have specifically this class. The class `"maybeNumber"`, for example, might be a class union (see the example for `setClassUnion`).

In either form, methods will be selected as they would be in actual computation, including use of inheritance and group generic functions. See `selectMethod` for the details, since it is the function used to find the appropriate method.

Writing Documentation for Methods

The on-line documentation for methods and classes uses some extensions to the R documentation format to implement the requests for class and method documentation described above. See the document *Writing R Extensions* for the available markup commands (you should have consulted this document already if you are at the stage of documenting your software).

In addition to the specific markup commands to be described, you can create an initial, overall file with a skeleton of documentation for the methods defined for a particular generic function:

```
promptMethods("myFun")
```

will create a file, '`myFun-methods.Rd`' with a skeleton of documentation for the methods defined for function `myFun`. The output from `promptMethods` is suitable if you want to describe all or most of the methods for the function in one file, separate from the documentation of the generic function itself. Once the file has been filled in and moved to the '`man`' subdirectory of your source package, requests for methods documentation will use that file, both for specific methods documentation as described above, and for overall documentation requested by

```
methods ? myFun
```

You are not required to use `promptMethods`, and if you do, you may not want to use the entire file created:

- If you want to document the methods in the file containing the documentation for the generic function itself, you can cut-and-paste to move the `\alias` lines and the `Methods` section from the file created by `promptMethods` to the existing file.

- On the other hand, if these are auxiliary methods, and you only want to document the added or modified software, you should strip out all but the relevant `\alias` lines for the methods of interest, and remove all but the corresponding `\item` entries in the `Methods` section. Note that in this case you will usually remove the first `\alias` line as well, since that is the marker for general methods documentation on this function (in the example, '`\alias{myfun-methods}`').

If you simply want to direct documentation for one or more methods to a particular R documentation file, insert the appropriate alias.

`dotsMethods` *The Use of . . . in Method Signatures*

Description

The "..." argument in R functions is treated specially, in that it matches zero, one or more actual arguments (and so, objects). A mechanism has been added to R to allow "..." as the signature of a generic function. Methods defined for such functions will be selected and called when *all* the arguments matching "..." are from the specified class or from some subclass of that class.

Using "..." in a Signature

Beginning with version 2.8.0 of R, S4 methods can be dispatched (selected and called) corresponding to the special argument "...". Currently, "..." cannot be mixed with other formal arguments: either the signature of the generic function is "..." only, or it does not contain "...". (This restriction may be lifted in a future version.)

Given a suitable generic function, methods are specified in the usual way by a call to `setMethod`. The method definition must be written expecting all the arguments corresponding to "..." to be from the class specified in the method's signature, or from a class that extends that class (i.e., a subclass of that class).

Typically the methods will pass "..." down to another function or will create a list of the arguments and iterate over that. See the examples below.

When you have a computation that is suitable for more than one existing class, a convenient approach may be to define a union of these classes by a call to `setClassUnion`. See the example below.

Method Selection and Dispatch for "..."

See Methods_Details for a general discussion. The following assumes you have read the "Method Selection and Dispatch" section of that documentation.

A method selecting on "..." is specified by a single class in the call to `setMethod`. If all the actual arguments corresponding to "..." have this class, the corresponding method is selected directly.

Otherwise, the class of each argument and that class' superclasses are computed, beginning with the first "..." argument. For the first argument, eligible methods are those for any of the classes. For each succeeding argument that introduces a class not considered previously, the eligible methods are further restricted to those matching the argument's class or superclasses. If no further eligible classes exist, the iteration breaks out and the default method, if any, is selected.

At the end of the iteration, one or more methods may be eligible. If more than one, the selection looks for the method with the least distance to the actual arguments. For each argument, any inherited method corresponds to a distance, available from the `contains` slot of the class definition. Since the same class can arise for more than one argument, there may be several distances associated with it. Combining them is inevitably arbitrary: the current computation uses the minimum distance. Thus, for example, if a method matched one argument directly, one as first generation superclass and another as a second generation superclass, the distances are 0, 1 and 2. The current selection computation would use distance 0 for this method. In particular, this selection criterion tends to use a method that matches exactly one or more of the arguments' class.

As with ordinary method selection, there may be multiple methods with the same distance. A warning message is issued and one of the methods is chosen (the first encountered, which in this case is rather arbitrary).

Notice that, while the computation examines all arguments, the essential cost of dispatch goes up with the number of *distinct* classes among the arguments, likely to be much smaller than the number of arguments when the latter is large.

Implementation Details

Methods dispatching on "..." were introduced in version 2.8.0 of R. The initial implementation of the corresponding selection and dispatch is in an R function, for flexibility while the new mechanism is being studied. In this implementation, a local version of `standardGeneric` is inserted in the generic function's environment. The local version selects a method according to the criteria above and calls that method, from the environment of the generic function. This is slightly different from the action taken by the C implementation when "..." is not involved. Aside from the extra computing time required, the method is evaluated in a true function call, as opposed to the special context constructed by the C version (which cannot be exactly replicated in R code.) However, situations in which different computational results would be obtained have not been encountered so far, and seem very unlikely.

Methods dispatching on arguments other than "..." are *cached* by storing the inherited method in the table of all methods, where it will be found on the next selection with the same combination of classes in the actual arguments (but not used for inheritance searches). Methods based on "..." are also cached, but not found quite as immediately. As noted, the selected method depends only on the set of classes that occur in the "..." arguments. Each of these classes can appear one or more times, so many combinations of actual argument classes will give rise to the same effective signature. The selection computation first computes and sorts the distinct classes encountered. This gives a label that will be cached in the table of all methods, avoiding any further search for inherited classes after the first occurrence. A call to `showMethods` will expose such inherited methods.

The intention is that the "..." features will be added to the standard C code when enough experience with them has been obtained. It is possible that at the same time, combinations of "..." with other arguments in signatures may be supported.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

For the general discussion of methods, see Methods_Details and links from there.

Examples

```
cc <- function(...)c(...)

setGeneric("cc")

setMethod("cc", "character", function(...)paste(...))

setClassUnion("Number", c("numeric", "complex"))

setMethod("cc", "Number", function(...) sum(...))

setClass("cdate", contains = "character", slots = c(date = "Date"))

setClass("vdate", contains = "vector", slots = c(date = "Date"))

cd1 <- new("cdate", "abcdef", date = Sys.Date())

cd2 <- new("vdate", "abcdef", date = Sys.Date())

stopifnot(identical(cc(letters, character(), cd1),
           paste(letters, character(), cd1))) # the "character" method

stopifnot(identical(cc(letters, character(), cd2),
                    c(letters, character(), cd2)))
# the default, because "vdate" doesn't extend "character"

stopifnot(identical(cc(1:10, 1+1i), sum(1:10, 1+1i))) # the "Number" method

stopifnot(identical(cc(1:10, 1+1i, TRUE), c(1:10, 1+1i, TRUE))) # the default

stopifnot(identical(cc(), c())) # no arguments implies the default method

setGeneric("numMax", function(...)standardGeneric("numMax"))

setMethod("numMax", "numeric", function(...)max(...))
# won't work for complex data
setMethod("numMax", "Number", function(...) paste(...))
# should not be selected w/o complex args

stopifnot(identical(numMax(1:10, pi, 1+1i), paste(1:10, pi, 1+1i)))
stopifnot(identical(numMax(1:10, pi, 1), max(1:10, pi, 1)))

try(numMax(1:10, pi, TRUE)) # should be an error:  no default method

## A generic version of paste(), dispatching on the "..." argument:
setGeneric("paste", signature = "...")

setMethod("paste", "Number", function(..., sep, collapse) c(...))
```

```
stopifnot(identical(paste(1:10, pi, 1), c(1:10, pi, 1)))
```

`environment-class` *Class* `"environment"`

Description

A formal class for R environments.

Objects from the Class

Objects can be created by calls of the form `new("environment", ...)`. The arguments in `...`, if any, should be named and will be assigned to the newly created environment.

Methods

coerce `signature(from = "ANY", to = "environment")`: calls `as.environment`.

initialize `signature(object = "environment")`: Implements the assignments in the new environment. Note that the `object` argument is ignored; a new environment is *always* created, since environments are not protected by copying.

See Also

`new.env`

`envRefClass-class` *Class* `"envRefClass"`

Description

Support Class to Implement R Objects using Reference Semantics

NOTE:

The software described here is an initial version. The eventual goal is to support reference-style classes with software in R itself or using inter-system interfaces. The current implementation (R version 2.12.0) is preliminary and subject to change, and currently includes only the R-only implementation. Developers are encouraged to experiment with the software, but the description here is more than usually subject to change.

Purpose of the Class

This class implements basic reference-style semantics for R objects. Objects normally do not come directly from this class, but from subclasses defined by a call to `setRefClass`. The documentation below is technical background describing the implementation, but applications should use the interface documented under `setRefClass`, in particular the `$` operator and field accessor functions as described there.

A Basic Reference Class

The design of reference classes for R divides those classes up according to the mechanism used for implementing references, fields, and class methods. Each version of this mechanism is defined by a *basic reference class*, which must implement a set of methods and provide some further information used by `setRefClass`.

The required methods are for operators `$` and `$<-` to get and set a field in an object, and for `initialize` to initialize objects.

To support these methods, the basic reference class needs to have some implementation mechanism to store and retrieve data from fields in the object. The mechanism needs to be consistent with reference semantics; that is, changes made to the contents of an object are global, seen by any code accessing that object, rather than only local to the function call where the change takes place. As described below, class `envRefClass` implements reference semantics through specialized use of environment objects. Other basic reference classes may use an interface to a language such as Java or C++ using reference semantics for classes.

Usually, the R user will be able to invoke class methods on the class, using the `$` operator. The basic reference class method for `$` needs to make this possible. Essentially, the operator must return an R function corresponding to the object and the class method name.

Class methods may include an implementation of data abstraction, in the sense that fields are accessed by "get" and "set" methods. The basic reference class provides this facility by setting the `"fieldAccessorGenerator"` slot in its definition to a function of one variable. This function will be called by `setRefClass` with the vector of field names as arguments. The generator function must return a list of defined accessor functions. An element corresponding to a get operation is invoked with no arguments and should extract the corresponding field; an element for a set operation will be invoked with a single argument, the value to be assigned to the field. The implementation needs to supply the object, since that is not an argument in the method invocation. The mechanism used currently by `envRefClass` is described below.

Support Classes

Two virtual classes are supplied to test for reference objects: `is(x, "refClass")` tests whether `x` comes from a class defined using the reference class mechanism described here; `is(x, "refObject")` tests whether the object has reference semantics generally, including the previous classes and also classes inheriting from the R types with reference semantics, such as `"environment"`.

Installed class methods are `"classMethodDefinition"` objects, with slots that identify the name of the function as a class method and the other class methods called from this method. The latter information is determined heuristically when the class is defined by using the `codetools` recommended package. This package must be installed when reference classes are defined, but is not needed in order to use existing reference classes.

Author(s)

John Chambers

`evalSource` *Use Function Definitions from a Source File without Reinstalling a Package*

Description

Definitions of functions and/or methods from a source file are inserted into a package, using the `trace` mechanism. Typically, this allows testing or debugging modified versions of a few functions without reinstalling a large package.

Usage

```
evalSource(source, package = "", lock = TRUE, cache = FALSE)

insertSource(source, package = "", functions = , methods = ,
           force = )
```

Arguments

`source`	A file to be parsed and evaluated by `evalSource` to find the new function and method definitions. The argument to `insertSource` can be an object of class `"sourceEnvironment"` returned from a previous call to `evalSource` If a file name is passed to `insertSource` it calls `evalSource` to obtain the corresponding object. See the section on the class for details.
`package`	Optionally, the name of the package to which the new code corresponds and into which it will be inserted. Although the computations will attempt to infer the package if it is omitted, the safe approach is to supply it. In the case of a package that is not attached to the search list, the package name must be supplied.
`functions, methods`	Optionally, the character-string names of the functions to be used in the insertion. Names supplied in the `functions` argument are expected to be defined as functions in the source. For names supplied in the `methods` argument, a table of methods is expected (as generated by calls to `setMethod`, see the details section); methods from this table will be inserted by `insertSource`. In both cases, the revised function or method is inserted only if it differs from the version in the corresponding package as loaded. If `what` is omitted, the results of evaluating the source file will be compared to the contents of the package (see the details section).

`lock`, `cache` Optional arguments to control the actions taken by `evalSource`. If `lock` is `TRUE`, the environment in the object returned will be locked, and so will all its bindings. If `cache` is `FALSE`, the normal caching of method and class definitions will be suppressed during evaluation of the `source` file.

The default settings are generally recommended, the `lock` to support the credibility of the object returned as a snapshot of the source file, and the second so that method definitions can be inserted later by `insertSource` using the trace mechanism.

`force` If `FALSE`, only functions currently in the environment will be redefined, using `trace`. If `TRUE`, other objects/functions will be simply assigned. By default, `TRUE` if neither the `functions` nor the `methods` argument is supplied.

Details

The `source` file is parsed and evaluated, suppressing by default the actual caching of method and class definitions contained in it, so that functions and methods can be tested out in a reversible way. The result, if all goes well, is an environment containing the assigned objects and metadata corresponding to method and class definitions in the source file.

From this environment, the objects are inserted into the package, into its namespace if it has one, for use during the current session or until reverting to the original version by a call to `untrace`. The insertion is done by calls to the internal version of `trace`, to make reversion possible.

Because the trace mechanism is used, only function-type objects will be inserted, functions themselves or S4 methods.

When the `functions` and `methods` arguments are both omitted, `insertSource` selects all suitable objects from the result of evaluating the `source` file.

In all cases, only objects in the source file that differ from the corresponding objects in the package are inserted. The definition of "differ" is that either the argument list (including default expressions) or the body of the function is not identical. Note that in the case of a method, there need be no specific method for the corresponding signature in the package: the comparison is made to the method that would be selected for that signature.

Nothing in the computation requires that the source file supplied be the same file as in the original package source, although that case is both likely and sensible if one is revising the package. Nothing in the computations compares source files: the objects generated by evaluating `source` are compared as objects to the content of the package.

Value

An object from class `"sourceEnvironment"`, a subclass of `"environment"` (see the section on the class) The environment contains the versions of *all* object resulting from evaluation of the source file. The class also has slots for the time of creation, the source file and the package name. Future extensions may use these objects for versioning or other code tools.

The object returned can be used in debugging (see the section on that topic) or as the `source` argument in a future call to `insertSource`. If only some of the revised functions were inserted in the first call, others can be inserted in a later call without re-evaluating the source file, by supplying the environment and optionally suitable `functions` and/or `methods` argument.

Debugging

Once a function or method has been inserted into a package by `insertSource`, it can be studied by the standard debugging tools; for example, `debug` or the various versions of `trace`.

Calls to `trace` should take the extra argument `edit = env`, where `env` is the value returned by the call to `evalSource`. The trace mechanism has been used to install the revised version from the source file, and supplying the argument ensures that it is this version, not the original, that will be traced. See the example below.

To turn tracing off, but retain the source version, use `trace(x,edit = env)` as in the example. To return to the original version from the package, use `untrace(x)`.

Class `"sourceEnvironment"`

Objects from this class can be treated as environments, to extract the version of functions and methods generated by `evalSource`. The objects also have the following slots:

`packageName`**:** The character-string name of the package to which the source code corresponds.

`dateCreated`**:** The date and time that the source file was evaluated (usually from a call to `Sys.time`).

`sourceFile`**:** The character-string name of the source file used.

Note that using the environment does not change the `dateCreated`.

See Also

`trace` for the underlying mechanism, and also for the `edit=` argument that can be used for somewhat similar purposes; that function and also `debug` and `setBreakpoint`, for techniques more oriented to traditional debugging styles. The present function is directly intended for the case that one is modifying some of the source for an existing package, although it can be used as well by inserting debugging code in the source (more useful if the debugging involved is non-trivial). As noted in the details section, the source file need not be the same one in the original package source.

Examples

```
## Not run:
## Suppose package P0 has a source file "all.R"
## First, evaluate the source, and from it
## insert the revised version of methods for summary()
  env <- insertSource("./P0/R/all.R", package = "P0",
     methods = "summary")
## now test one of the methods, tracing  the version from the source
  trace("summary", signature = "myMat", browser, edit = env)
## After testing, remove the browser() call but keep the source
  trace("summary", signature = "myMat", edit = env)
## Now insert all the (other) revised functions and methods
## without re-evaluating the source file.
## The package name is included in the object env.
  insertSource(env)

## End(Not run)
```

findClass *Find Class Definitions*

Description

Functions to find classes: `isClass` tests for a class; `findClass` returns the name(s) of packages containing the class; `getClasses` returns the names of all the classes in an environment, typically a namespace. To examine the definition of a class, use `getClass`.

Usage

```
isClass(Class, formal=TRUE, where)

getClasses(where, inherits = missing(where))

findClass(Class, where, unique = "")

## The remaining functions are retained for compatibility
## but not generally recommended

removeClass(Class, where)

resetClass(Class, classDef, where)

sealClass(Class, where)
```

Arguments

`Class`	character string name for the class. The functions will usually take a class definition instead of the string. To restrict the class to those defined in a particular package, set the `packageSlot` of the character string.
`where`	the `environment` in which to search for the class definition. Defaults to the top-level environment of the calling function. When called from the command line, this has the effect of using all the package environments in the search list. To restrict the search to classes in a particular package, use `where = asNamespace(pkg)` with `pkg` the package name; to restrict it to the *exported* classes, use `where = "package:pkg"` after the package is attached to the search list.
`formal`	`logical` is a formal definition required? For S compatibility, and always treated as `TRUE`.
`unique`	if `findClass` expects a unique location for the class, `unique` is a character string explaining the purpose of the search (and is used in warning and error messages). By default, multiple locations are possible and the function always returns a list.

`inherits` in a call to `getClasses`, should the value returned include all parent environments of `where`, or that environment only? Defaults to `TRUE` if `where` is omitted, and to `FALSE` otherwise.

`classDef` For `resetClass`, the optional class definition.

Functions

`isClass`**:** Is this the name of a formally defined class?

`getClasses`**:** The names of all the classes formally defined on `where`. If called with no argument, all the classes visible from the calling function (if called from the top-level, all the classes in any of the environments on the search list). The `where` argument is used to search only in a particular package.

`findClass`**:** The list of environments in which a class definition of `Class` is found. If `where` is supplied, a list is still returned, either empty or containing the environment corresponding to `where`. By default when called from the R session, the global environment and all the currently attached packages are searched.

If `unique` is supplied as a character string, `findClass` will warn if there is more than one definition visible (using the string to identify the purpose of the call), and will generate an error if no definition can be found.

The remaining functions are retained for back-compatibility and internal use, but not generally recommended.

`removeClass`**:** Remove the definition of this class. This can't be used if the class is in another package, and would rarely be needed in source code defining classes in a package.

`resetClass`**:** Reset the internal definition of a class. Not legitimate for a class definition not in this package and rarely needed otherwise.

`sealClass`**:** Seal the current definition of the specified class, to prevent further changes, by setting the corresponding slot in the class definition. This is rarely used, since classes in loaded packages are sealed by locking their namespace.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Chapter 9 has some details not in the later reference.)

See Also

`getClass`, `Classes_Details`, `Methods_Details`, `makeClassRepresentation`

findMethods *Description of the Methods Defined for a Generic Function*

Description

The function findMethods converts the methods defined in a table for a generic function (as used for selection of methods) into a list, for study or display. The list is actually from the class listOfMethods (see the section describing the class, below).

The list will be limited to the methods defined in environment where if that argument is supplied and limited to those including one or more of the specified classes in the method signature if that argument is supplied.

To see the actual table (an environment) used for methods dispatch, call getMethodsForDispatch. The names of the list returned by findMethods are the names of the objects in the table.

The function findMethodSignatures returns a character matrix whose rows are the class names from the signature of the corresponding methods; it operates either from a list returned by findMethods, or by computing such a list itself, given the same arguments as findMethods .

The function hasMethods returns TRUE or FALSE according to whether there is a non-empty table of methods for function f in the environment or search position where (or for the generic function generally if where is missing).

The defunct function getMethods is an older alternative to findMethods , returning information in the form of an object of class MethodsList, previously used for method dispatch. This class of objects is deprecated generally and will disappear in a future version of R.

Usage

```
findMethods(f, where, classes = character(), inherited = FALSE,
      package = "")

findMethodSignatures(..., target = TRUE, methods = )

hasMethods(f, where, package)

## Deprecated in 2010 and defunct in 2015 for 'table = FALSE':
getMethods(f, where, table = FALSE)
```

Arguments

f	A generic function or the character-string name of one.
where	Optionally, an environment or position on the search list to look for methods metadata. If where is missing, findMethods uses the current table of methods in the generic function itself, and hasMethods looks for metadata anywhere in the search list.

table	If TRUE in a call to getMethods the returned value is the table used for dispatch, including inherited methods discovered to date. Used internally, but since the default result is the now unused mlist object, the default will likely be changed at some point.
classes	If supplied, only methods whose signatures contain at least one of the supplied classes will be included in the value returned.
inherited	Logical flag; if TRUE, the table of all methods, inherited or defined directly, will be used; otherwise, only the methods explicitly defined. Option TRUE is meaningful only if where is missing.
...	In the call to findMethodSignatures, any arguments that might be given to findMethods.
target	Optional flag to findMethodSignatures; if TRUE, the signatures used are the target signatures (the classes for which the method will be selected); if FALSE, they will be the signatures are defined. The difference is only meaningful if inherited is TRUE.
methods	In the call to findMethodSignatures, an optional list of methods, presumably returned by a previous call to findMethods. If missing, that function will be call with the ... arguments.
package	In a call to hasMethods, the package name for the generic function (e.g., "base" for primitives). If missing this will be inferred either from the "package" attribute of the function name, if any, or from the package slot of the generic function. See 'Details'.

Details

The functions obtain a table of the defined methods, either from the generic function or from the stored metadata object in the environment specified by where. In a call to getMethods, the information in the table is converted as described above to produce the returned value, except with the table argument.

Note that hasMethods, but not the other functions, can be used even if no generic function of this name is currently found. In this case package must either be supplied as an argument or included as an attribute of f, since the package name is part of the identification of the methods tables.

The Class for lists of methods

The class "listOfMethods" returns the methods as a named list of method definitions (or a primitive function, see the slot documentation below). The names are the strings used to store the corresponding objects in the environment from which method dispatch is computed. The current implementation uses the names of the corresponding classes in the method signature, separated by "#" if more than one argument is involved in the signature.

Slots

.Data: Object of class "list" The method definitions.

Note that these may include the primitive function itself as default method, when the generic corresponds to a primitive. (Basically, because primitive functions are abnormal R objects, which cannot currently be extended as method definitions.) Computations that use the returned

list to derive other information need to take account of this possibility. See the implementation of `findMethodSignatures` for an example.

`arguments`: Object of class `"character"`. The names of the formal arguments in the signature of the generic function.

`signatures`: Object of class `"list"`. A list of the signatures of the individual methods. This is currently the result of splitting the `names` according to the `"#"` separator.

If the object has been constructed from a table, as when returned by `findMethods`, the signatures will all have the same length. However, a list rather than a character matrix is used for generality. Calling `findMethodSignatures` as in the example below will always convert to the matrix form.

`generic`: Object of class `"genericFunction"`. The generic function corresponding to these methods. There are plans to generalize this slot to allow reference to the function.

`names`: Object of class `"character"`. The names as noted are the class names separated by `"#"` .

Extends

Class `"namedList"`, directly.

Class `"list"`, by class `"namedList"`, distance 2.

Class `"vector"`, by class `"namedList"`, distance 3.

See Also

`showMethods`, `selectMethod`, Methods_Details

Examples

```
mm <-  findMethods("Ops")
findMethodSignatures(methods = mm)
```

`fixPre1.8` *Fix Objects Saved from R Versions Previous to 1.8*

Description

Beginning with R version 1.8.0, the class of an object contains the identification of the package in which the class is defined. The function `fixPre1.8` fixes and re-assigns objects missing that information (typically because they were loaded from a file saved with a previous version of R.)

Usage

```
fixPre1.8(names, where)
```

Arguments

`names`	Character vector of the names of all the objects to be fixed and re-assigned.
`where`	The environment from which to look for the objects, and for class definitions. Defaults to the top environment of the call to `fixPre1.8`, the global environment if the function is used interactively.

Details

The named object will be saved where it was found. Its class attribute will be changed to the full form required by R 1.8; otherwise, the contents of the object should be unchanged.

Objects will be fixed and re-assigned only if all the following conditions hold:

1. The named object exists.
2. It is from a defined class (not a basic datatype which has no actual class attribute).
3. The object appears to be from an earlier version of R.
4. The class is currently defined.
5. The object is consistent with the current class definition.

If any condition except the second fails, a warning message is generated.

Note that `fixPre1.8` currently fixes *only* the change in class attributes. In particular, it will not fix binary versions of packages installed with earlier versions of R if these use incompatible features. Such packages must be re-installed from source, which is the wise approach always when major version changes occur in R.

Value

The names of all the objects that were in fact re-assigned.

`genericFunction-class` *Generic Function Objects*

Description

Generic functions (objects from or extending class `genericFunction`) are extended function objects, containing information used in creating and dispatching methods for this function. They also identify the package associated with the function and its methods.

Objects from the Class

Generic functions are created and assigned by `setGeneric` or `setGroupGeneric` and, indirectly, by `setMethod`.

As you might expect `setGeneric` and `setGroupGeneric` create objects of class `"genericFunction"` and `"groupGenericFunction"` respectively.

Slots

`.Data`**:** Object of class `"function"`, the function definition of the generic, usually created automatically as a call to `standardGeneric`.

`generic`**:** Object of class `"character"`, the name of the generic function.

`package`**:** Object of class `"character"`, the name of the package to which the function definition belongs (and *not* necessarily where the generic function is stored). If the package is not specified explicitly in the call to `setGeneric`, it is usually the package on which the corresponding non-generic function exists.

`group`: Object of class `"list"`, the group or groups to which this generic function belongs. Empty by default.

`valueClass`: Object of class `"character"`; if not an empty character vector, identifies one or more classes. It is asserted that all methods for this function return objects from these class (or from classes that extend them).

`signature`: Object of class `"character"`, the vector of formal argument names that can appear in the signature of methods for this generic function. By default, it is all the formal arguments, except for `...`. Order matters for efficiency: the most commonly used arguments in specifying methods should come first.

`default`: Object of class `"optionalMethod"` (a union of classes `"function"` and `"NULL"`), containing the default method for this function if any. Generated automatically and used to initialize method dispatch.

`skeleton`: Object of class `"call"`, a slot used internally in method dispatch. Don't expect to use it directly.

Extends

Class `"function"`, from data part.
Class `"OptionalMethods"`, by class `"function"`.
Class `"PossibleMethod"`, by class `"function"`.

Methods

Generic function objects are used in the creation and dispatch of formal methods; information from the object is used to create methods list objects and to merge or update the existing methods for this generic.

GenericFunctions *Tools for Managing Generic Functions*

Description

The functions documented here manage collections of methods associated with a generic function, as well as providing information about the generic functions themselves.

Usage

```
isGeneric(f, where, fdef, getName = FALSE)
isGroup(f, where, fdef)
removeGeneric(f, where)

dumpMethod(f, signature, file, where, def)
findFunction(f, generic = TRUE, where = topenv(parent.frame()))
dumpMethods(f, file, signature, methods, where)
signature(...)
```

```
removeMethods(f, where = topenv(parent.frame()), all = missing(where))

setReplaceMethod(f, ..., where = topenv(parent.frame()))

getGenerics(where, searchForm = FALSE)
```

Arguments

f	The character string naming the function.
where	The environment, namespace, or search-list position from which to search for objects. By default, start at the top-level environment of the calling function, typically the global environment (i.e., use the search list), or the namespace of a package from which the call came. It is important to supply this argument when calling any of these functions indirectly. With package namespaces, the default is likely to be wrong in such calls.
signature	The class signature of the relevant method. A signature is a named or unnamed vector of character strings. If named, the names must be formal argument names for the generic function. Signatures are matched to the arguments specified in the signature slot of the generic function (see the Details section of the setMethod documentation). The signature argument to dumpMethods is ignored (it was used internally in previous implementations).
file	The file or connection on which to dump method definitions.
def	The function object defining the method; if omitted, the current method definition corresponding to the signature.
...	Named or unnamed arguments to form a signature.
generic	In testing or finding functions, should generic functions be included. Supply as FALSE to get only non-generic functions.
fdef	Optional, the generic function definition. Usually omitted in calls to isGeneric
getName	If TRUE, isGeneric returns the name of the generic. By default, it returns TRUE.
methods	The methods object containing the methods to be dumped. By default, the methods defined for this generic (optionally on the specified where location).
all	in removeMethods, logical indicating if all (default) or only the first method found should be removed.
searchForm	In getGenerics, if TRUE, the package slot of the returned result is in the form used by search(), otherwise as the simple package name (e.g, "package:base" vs "base").

Summary of Functions

isGeneric**:** Is there a function named f, and if so, is it a generic?

The getName argument allows a function to find the name from a function definition. If it is TRUE then the name of the generic is returned, or FALSE if this is not a generic function definition.

The behavior of isGeneric and getGeneric for primitive functions is slightly different. These functions don't exist as formal function objects (for efficiency and historical reasons), regardless of whether methods have been defined for them. A call to isGeneric tells you whether methods have been defined for this primitive function, anywhere in the current search list, or in the specified position where. In contrast, a call to getGeneric will return what the generic for that function would be, even if no methods have been currently defined for it.

removeGeneric, removeMethods: Remove all the methods for the generic function of this name. In addition, removeGeneric removes the function itself; removeMethods restores the non-generic function which was the default method. If there was no default method, removeMethods leaves a generic function with no methods.

standardGeneric: Dispatches a method from the current function call for the generic function f. It is an error to call standardGeneric anywhere except in the body of the corresponding generic function.

Note that standardGeneric is a primitive function in the **base** package for efficiency reasons, but rather documented here where it belongs naturally.

dumpMethod: Dump the method for this generic function and signature.

findFunction: return a list of either the positions on the search list, or the current top-level environment, on which a function object for name exists. The returned value is *always* a list, use the first element to access the first visible version of the function. See the example.

NOTE: Use this rather than find with mode="function", which is not as meaningful, and has a few subtle bugs from its use of regular expressions. Also, findFunction works correctly in the code for a package when attaching the package via a call to library.

dumpMethods: Dump all the methods for this generic.

signature: Returns a named list of classes to be matched to arguments of a generic function.

getGenerics: returns the names of the generic functions that have methods defined on where; this argument can be an environment or an index into the search list. By default, the whole search list is used.

The methods definitions are stored with package qualifiers; for example, methods for function "initialize" might refer to two different functions of that name, on different packages. The package names corresponding to the method list object are contained in the slot package of the returned object. The form of the returned name can be plain (e.g., "base"), or in the form used in the search list ("package:base") according to the value of searchForm

Details

isGeneric: If the fdef argument is supplied, take this as the definition of the generic, and test whether it is really a generic, with f as the name of the generic. (This argument is not available in S-Plus.)

removeGeneric: If where supplied, just remove the version on this element of the search list; otherwise, removes the first version encountered.

standardGeneric: Generic functions should usually have a call to standardGeneric as their entire body. They can, however, do any other computations as well.

The usual setGeneric (directly or through calling setMethod) creates a function with a call to standardGeneric.

dumpMethod: The resulting source file will recreate the method.

`findFunction`: If `generic` is `FALSE`, ignore generic functions.

`dumpMethods`: If `signature` is supplied only the methods matching this initial signature are dumped. (This feature is not found in S-Plus: don't use it if you want compatibility.)

`signature`: The advantage of using `signature` is to provide a check on which arguments you meant, as well as clearer documentation in your method specification. In addition, `signature` checks that each of the elements is a single character string.

`removeMethods`: Returns `TRUE` if `f` was a generic function, `FALSE` (silently) otherwise.

If there is a default method, the function will be re-assigned as a simple function with this definition. Otherwise, the generic function remains but with no methods (so any call to it will generate an error). In either case, a following call to `setMethod` will consistently re-establish the same generic function as before.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

`getMethod` (also for `selectMethod`), `setGeneric`, `setClass`, `showMethods`

Examples

```
require(stats) # for lm

## get the function "myFun" -- throw an error if 0 or > 1 versions visible:
findFuncStrict <- function(fName) {
  allF <- findFunction(fName)
  if(length(allF) == 0)
    stop("No versions of ",fName," visible")
  else if(length(allF) > 1)
    stop(fName," is ambiguous: ", length(allF), " versions")
  else
    get(fName, allF[[1]])
}

try(findFuncStrict("myFun"))# Error: no version
lm <- function(x) x+1
try(findFuncStrict("lm"))#    Error: 2 versions
findFuncStrict("findFuncStrict")# just 1 version
rm(lm)



## method dumping -----------------------------------

setClass("A", slots = c(a="numeric"))
setMethod("plot", "A", function(x,y,...){ cat("A meth\n") })
dumpMethod("plot","A", file="")
## Not run:
setMethod("plot", "A",
```

```
function (x, y, ...)
{
    cat("AAAAA\n")
}
)

## End(Not run)
tmp <- tempfile()
dumpMethod("plot","A", file=tmp)
## now remove, and see if we can parse the dump
stopifnot(removeMethod("plot", "A"))
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"))

## same with dumpMethods() :
setClass("B", contains="A")
setMethod("plot", "B", function(x,y,...){ cat("B ...\n") })
dumpMethods("plot", file=tmp)
stopifnot(removeMethod("plot", "A"),
          removeMethod("plot", "B"))
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"),
          is(getMethod("plot", "B"), "MethodDefinition"))
```

`getClass` *Get Class Definition*

Description

Get the definition of a class.

Usage

```
getClass   (Class, .Force = FALSE, where)
getClassDef(Class, where, package, inherits = TRUE)
```

Arguments

`Class`	the character-string name of the class, often with a `"package"` attribute as noted below under `package`.
`.Force`	if `TRUE`, return `NULL` if the class is undefined; otherwise, an undefined class results in an error.
`where`	environment from which to begin the search for the definition; by default, start at the top-level (global) environment and proceed through the search list.
`package`	the name or environment of the package asserted to hold the definition. If it is a non-empty string it is used instead of `where`, as the first place to look for the class. Note that the package must be loaded but need not be attached. By default, the package attribute of the `Class` argument is used, if any. There will usually be a package attribute if `Class` comes from `class(x)` for some object.

`inherits` logical; should the class definition be retrieved from any enclosing environment and also from the cache? If `FALSE` only a definition in the environment `where` will be returned.

Details

Class definitions are stored in metadata objects in a package namespace or other environment where they are defined. When packages are loaded, the class definitions in the package are cached in an internal table. Therefore, most calls to `getClassDef` will find the class in the cache or fail to find it at all, unless `inherits` is `FALSE`, in which case only the environment(s) defined by `package` or `where` are searched.

The class cache allows for multiple definitions of the same class name in separate environments, with of course the limitation that the package attribute or package name must be provided in the call to

Value

The object defining the class. If the class definition is not found, `getClassDef` returns `NULL`, while `getClass`, which calls `getClassDef`, either generates an error or, if `.Force` is `TRUE`, returns a simple definition for the class. The latter case is used internally, but is not typically sensible in user code.

The non-null returned value is an object of class `classRepresentation`.

Use functions such as `setClass` and `setClassUnion` to create class definitions.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

classRepresentation, `setClass`, `isClass`.

Examples

```
getClass("numeric") ## a built in class

cld <- getClass("thisIsAnUndefinedClass", .Force = TRUE)
cld ## a NULL prototype
## If you are really curious:
utils::str(cld)
## Whereas these generate errors:
try(getClass("thisIsAnUndefinedClass"))
try(getClassDef("thisIsAnUndefinedClass"))
```

getMethod *Get or Test for the Definition of a Method*

Description

The function selectMethod() returns the method that would be selected for a call to function f if the arguments had classes as specified by signature. Failing to find a method is an error, unless argument optional = TRUE, in which case NULL is returned.

The function findMethod() returns a list of environments that contain a method for the specified function and signature; by default, these are a subset of the packages in the current search list. See section "Using findMethod()" for details.

The function getMethod() returns the method corresponding to the function and signature supplied similarly to selectMethod, but without using inheritance or group generics.

The functions hasMethod() and existsMethod() test whether selectMethod() or getMethod(), respectively, finds a matching method.

Usage

```
selectMethod(f, signature, optional = FALSE, useInherited =,
           mlist = , fdef = , verbose = , doCache = )

findMethod(f, signature, where)

getMethod(f, signature = character(), where, optional = FALSE,
           mlist, fdef)

existsMethod(f, signature = character(), where)

hasMethod(f, signature = character(), where)
```

Arguments

f	a generic function or the character-string name of one.
signature	the signature of classes to match to the arguments of f. See the details below.
where	the environment in which to look for the method(s). By default, if the call comes from the command line, the table of methods defined in the generic function itself is used, except for findMethod (see the section below).
optional	if the selection in selectMethod does not find a valid method an error is generated, unless optional is TRUE, in which case the value returned is NULL.
mlist, fdef, useInherited, verbose, doCache	optional arguments to getMethod and selectMethod for internal use. Avoid these: some will work as expected and others will not, and none of them is required for normal use of the functions. But see the section "Methods for as()" for nonstandard inheritance.

Details

The signature argument specifies classes, corresponding to formal arguments of the generic function; to be precise, to the signature slot of the generic function object. The argument may be a vector of strings identifying classes, and may be named or not. Names, if supplied, match the names of those formal arguments included in the signature of the generic. That signature is normally all the arguments except .... However, generic functions can be specified with only a subset of the arguments permitted, or with the signature taking the arguments in a different order.

It's a good idea to name the arguments in the signature to avoid confusion, if you're dealing with a generic that does something special with its signature. In any case, the elements of the signature are matched to the formal signature by the same rules used in matching arguments in function calls (see match.call).

The strings in the signature may be class names, "missing" or "ANY". See Methods_Details for the meaning of these in method selection. Arguments not supplied in the signature implicitly correspond to class "ANY"; in particular, giving an empty signature means to look for the default method.

A call to getMethod returns the method for a particular function and signature. The search for the method makes no use of inheritance.

The function selectMethod also looks for a method given the function and signature, but makes full use of the method dispatch mechanism; i.e., inherited methods and group generics are taken into account just as they would be in dispatching a method for the corresponding signature, with the one exception that conditional inheritance is not used. Like getMethod, selectMethod returns NULL or generates an error if the method is not found, depending on the argument optional.

Both selectMethod and getMethod will normally use the current version of the generic function in the R session, which has a table of the methods obtained from all the packages loaded in the session. Optional arguments can cause a search for the generic function from a specified environment, but this is rarely a useful idea. In contrast, findMethod has a different default and the optional where= argument may be needed. See the section "Using findMethod()".

The functions existsMethod and hasMethod return TRUE or FALSE according to whether a method is found, the first corresponding to getMethod (no inheritance) and the second to selectMethod.

Value

The call to selectMethod or getMethod returns the selected method, if one is found. (This class extends function, so you can use the result directly as a function if that is what you want.) Otherwise an error is thrown if optional is FALSE and NULL is returned if optional is TRUE.

The returned method object is a MethodDefinition object, *except* that the default method for a primitive function is required to be the primitive itself. Note therefore that the only reliable test that the search failed is is.null().

The returned value of findMethod is a list of environments in which a corresponding method was found; that is, a table of methods including the one specified.

Using findMethod()

As its name suggests, this function is intended to behave like find, which produces a list of the packages on the current search list which have, and have exported, the object named. That's what findMethod does also, by default. The "exported" part in this case means that the package's namespace has an exportMethods directive for this generic function.

An important distinction is that the absence of such a directive does not prevent methods from the package from being called once the package is loaded. Otherwise, the code in the package could not use un-exported methods.

So, if your question is whether loading package `thisPkg` will define a method for this function and signature, you need to ask that question about the namespace of the package:

`findMethod(f, signature, where = asNamespace("thisPkg"))`

If the package did not export the method, attaching it and calling `findMethod` with no `where` argument will not find the method.

Notice also that the length of the signature must be what the corresponding package used. If `thisPkg` had only methods for one argument, only length-1 signatures will match (no trailing `"ANY"`), even if another currently loaded package had signatures with more arguments.

Methods for `as()`

The function `setAs` allows packages to define methods for coercing one class of objects to another class. This works internally by defining methods for the generic function `coerce(from,to)`, which can not be called directly.

The R evaluator selects methods for this purpose using a different form of inheritance. While methods can be inherited for the object being coerced, they cannot inherit for the target class, since the result would not be a valid object from that class. If you want to examine the selection procedure, you must supply the optional argument `useInherited = c(TRUE, FALSE)` to `selectMethod`.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.6 for some details of method selection.)

See Also

`Methods_Details` for the details of method selection; `GenericFunctions` for other functions manipulating methods and generic function objects; `MethodDefinition` for the class that represents method definitions.

Examples

```
testFun <-  function(x)x
setGeneric("testFun")
setMethod("testFun", "numeric", function(x)x+1)

hasMethod("testFun", "numeric") # TRUE

hasMethod("testFun", "integer") #TRUE, inherited

existsMethod("testFun", "integer") #FALSE

hasMethod("testFun") # TRUE, default method

hasMethod("testFun", "ANY")
```

getPackageName *The Name associated with a Given Package*

Description

The functions below produce the package associated with a particular environment or position on the search list, or of the package containing a particular function. They are primarily used to support computations that need to differentiate objects on multiple packages.

Usage

```
getPackageName(where, create = TRUE)
setPackageName(pkg, env)

packageSlot(object)
packageSlot(object) <- value
```

Arguments

where	the environment or position on the search list associated with the desired package.
object	object providing a character string name, plus the package in which this object is to be found.
value	the name of the package.
create	flag, should a package name be created if none can be inferred? If TRUE and no non-empty package name is found, the current date and time are used as a package name, and a warning is issued. The created name is stored in the environment if that environment is not locked.
pkg, env	make the string in pkg the internal package name for all computations that set class and method definitions in environment env.

Details

Package names are normally installed during loading of the package, by the INSTALL script or by the library function. (Currently, the name is stored as the object .packageName but don't trust this for the future.)

Value

getPackageName returns the character-string name of the package (without the extraneous "package:" found in the search list).

packageSlot returns or sets the package name slot (currently an attribute, not a formal slot, but this may change someday).

setPackageName can be used to establish a package name in an environment that would otherwise not have one. This allows you to create classes and/or methods in an arbitrary environment, but it is usually preferable to create packages by the standard R programming tools (package.skeleton, etc.)

See Also

search, packageName

Examples

```
## all the following usually return "base"
getPackageName(length(search()))
getPackageName(baseenv())
getPackageName(asNamespace("base"))
getPackageName("package:base")
```

---

hasArg *Look for an Argument in the Call*

Description

Returns TRUE if name corresponds to an argument in the call, either a formal argument to the function, or a component of ..., and FALSE otherwise.

Usage

```
hasArg(name)
```

Arguments

name	The name of a potential argument, as an unquoted name or character string.

Details

The expression hasArg(x), for example, is similar to !missing(x), with two exceptions. First, hasArg will look for an argument named x in the call if x is not a formal argument to the calling function, but ... is. Second, hasArg never generates an error if given a name as an argument, whereas missing(x) generates an error if x is not a formal argument.

Value

Always TRUE or FALSE as described above.

See Also

missing

Examples

```
ftest <- function(x1, ...) c(hasArg(x1), hasArg("y2"))

ftest(1) ## c(TRUE, FALSE)
ftest(1, 2)  ## c(TRUE, FALSE)
ftest(y2 = 2)   ## c(FALSE, TRUE)
ftest(y = 2)    ## c(FALSE, FALSE) (no partial matching)
ftest(y2 = 2, x = 1)  ## c(TRUE, TRUE) partial match x1
```

`implicitGeneric` *Manage Implicit Versions of Generic Functions*

Description

The implicit generic mechanism stores generic versions of functions in a table in a package. The package does not want the current version of the function to be a generic, however, and retains the non-generic version.

When a call to `setMethod` or `setGeneric` creates a generic version for one of these functions, the object in the table is used. This mechanism is only needed if special arguments were used to create the generic; e.g., the `signature` or the `valueClass` options.

Function `implicitGeneric()` returns the implicit generic version, `setGenericImplicit()` turns a generic implicit, `prohibitGeneric()` prevents your function from being made generic, and `registerImplicitGenerics()` saves a set of implicit generic definitions in the cached table of the current session.

Usage

```
implicitGeneric(name, where, generic)
setGenericImplicit(name, where, restore = TRUE)
prohibitGeneric(name, where)
registerImplicitGenerics(what, where)
```

Arguments

`name`	Character string name of the function.
`where`	Package or environment in which to register the implicit generics. When using the functions from the top level of your own package source, this argument should be omitted.
`generic`	Obsolete, and likely to be deprecated.
`restore`	Should the non-generic version of the function be restored?.
`what`	Optional table of the implicit generics to register, but nearly always omitted, when it defaults to a standard metadata name.

Details

Multiple packages may define methods for the same function, to apply to classes defined in that package. Arithmetic and other operators, `plot()` and many other basic computations are typical examples. It's essential that all such packages write methods for the *same* definition of the generic function. So long as that generic uses the default choice for signature and other parameters, nothing needs to be done.

If the generic has special properties, these need to be ensured for all packages creating methods for it. The simplest solution is just to make the function generic in the package that originally owned it. If for some reason the owner(s) of that package are unwilling to do this, the alternative is to define the correct generic, save it in a special table and restore the non-generic version by calling `setGenericImplicit`.

Note that the package containing the function can define methods for the implicit generic as well; when the implicit generic is made a real generic, those methods will be included.

The usual reason for having a non-default implicit generic is to provide a non-default signature, and the usual reason for *that* is to allow lazy evaluation of some arguments. All arguments in the signature of a generic function must be evaluated at the time the function needs to select a method. In the base function `with()` in the example below, evaluation of the argument `expr` must be delayed; therefore, it is excluded from the signature.

If you want to completely prohibit anyone from turning your function into a generic, call `prohibitGeneric()`.

Function `implicitGeneric()` returns the implicit generic version of the named function. If there is no table of these or if this function is not in the table, the result of a simple call `setGeneric(name)` is returned.

Value

Function `implicitGeneric()` returns the implicit generic definition (and caches that definition the first time if it has to construct it).

The other functions exist for their side effect and return nothing useful.

Implicit Generics for Base Functions

Implicit generic versions exist for some functions in the packages supplied in the distribution of R itself. These are stored in the 'methods' package itself and will always be available.

As emphasized repeatedly in the documentation, `setGeneric()` calls for a function in another package should never have non-default settings for arguments such as `signature`. The reasoning applies specially to functions in supplied packages, since methods for these are likely to exist in multiple packages. A call to `implicitGeneric()` will show the generic version.

See Also

`setGeneric`

Examples

```
### How we would make the function with() into a generic:
```

```
## Since the second argument, 'expr' is used literally, we want
## with() to only have "data" in the signature.

## Not run:
setGeneric("with", signature = "data")
## Now we could predefine methods for "with" if we wanted to.

## When ready, we store the generic as implicit, and restore the
original

setGenericImplicit("with")

## End(Not run)

implicitGeneric("with")

# (This implicit generic is stored in the 'methods' package.)
```

`inheritedSlotNames` *Names of Slots Inherited From a Super Class*

Description

For a class (or class definition, see `getClass` and the description of class `classRepresentation`), give the names which are inherited from "above", i.e., super classes, rather than by this class' definition itself.

Usage

```
inheritedSlotNames(Class, where = topenv(parent.frame()))
```

Arguments

`Class`	character string or `classRepresentation`, i.e., resulting from `getClass`.
`where`	environment, to be passed further to `isClass` and `getClass`.

Value

character vector of slot names, or `NULL`.

See Also

`slotNames`, `slot`, `setClass`, etc.

Examples

```
.srch <- search()
library(stats4)
inheritedSlotNames("mle")

if(require("Matrix", quietly = TRUE)) withAutoprint({
  inheritedSlotNames("Matrix")       # NULL
  ## whereas
  inheritedSlotNames("sparseMatrix") # --> Dim & Dimnames
  ##  i.e. inherited from "Matrix" class
  cl <- getClass("dgCMatrix")        # six slots, etc
  inheritedSlotNames(cl) # *all* six slots are inherited
})
## Not run:
## detach package we've attached above:
for(n in rev(which(is.na(match(search(), .srch)))))
   try( detach(pos = n) )

## End(Not run)
```

---

initialize-methods *Methods to Initialize New Objects from a Class*

Description

The arguments to function `new` to create an object from a particular class can be interpreted specially for that class, by the definition of a method for function `initialize` for the class. This documentation describes some existing methods, and also outlines how to write new ones.

Methods

`signature(.Object = "ANY")` The default method for `initialize` takes either named or unnamed arguments. Argument names must be the names of slots in this class definition, and the corresponding arguments must be valid objects for the slot (that is, have the same class as specified for the slot, or some superclass of that class). If the object comes from a superclass, it is not coerced strictly, so normally it will retain its current class (specifically, `as(object, Class, strict = FALSE)`).

Unnamed arguments must be objects of this class, of one of its superclasses, or one of its subclasses (from the class, from a class this class extends, or from a class that extends this class). If the object is from a superclass, this normally defines some of the slots in the object. If the object is from a subclass, the new object is that argument, coerced to the current class.

Unnamed arguments are processed first, in the order they appear. Then named arguments are processed. Therefore, explicit values for slots always override any values inferred from superclass or subclass arguments.

`signature(.Object = "traceable")` Objects of a class that extends `traceable` are used to implement debug tracing (see class traceable and `trace`).

The `initialize` method for these classes takes special arguments `def`, `tracer`, `exit`, `at`, `print`. The first of these is the object to use as the original definition (e.g., a function). The others correspond to the arguments to `trace`.

`signature(.Object = "environment")`, `signature(.Object = ".environment")` The `initialize` method for environments takes a named list of objects to be used to initialize the environment. Subclasses of `"environment"` inherit an initialize method through `".environment"`, which has the additional effect of allocating a new environment. If you define your own method for such a subclass, be sure either to call the existing method via `callNextMethod` or allocate an environment in your method, since environments are references and are not duplicated automatically.

`signature(.Object = "signature")` This is a method for internal use only. It takes an optional `functionDef` argument to provide a generic function with a `signature` slot to define the argument names. See Methods_Details for details.

Writing Initialization Methods

Initialization methods provide a general mechanism corresponding to generator functions in other languages.

The arguments to `initialize` are `.Object` and `....` Nearly always, `initialize` is called from `new`, not directly. The `.Object` argument is then the prototype object from the class.

Two techniques are often appropriate for `initialize` methods: special argument names and `callNextMethod`.

You may want argument names that are more natural to your users than the (default) slot names. These will be the formal arguments to your method definition, in addition to `.Object` (always) and `...` (optionally). For example, the method for class `"traceable"` documented above would be created by a call to `setMethod` of the form:

```
setMethod("initialize", "traceable",
  function(.Object, def, tracer, exit, at, print) { .... }
)
```

In this example, no other arguments are meaningful, and the resulting method will throw an error if other names are supplied.

When your new class extends another class, you may want to call the initialize method for this superclass (either a special method or the default). For example, suppose you want to define a method for your class, with special argument `x`, but you also want users to be able to set slots specifically. If you want `x` to override the slot information, the beginning of your method definition might look something like this:

```
function(.Object, x, ...) {
  Object <- callNextMethod(.Object, ...)
  if(!missing(x)) { # do something with x
```

You could also choose to have the inherited method override, by first interpreting `x`, and then calling the next method.

`Introduction` *Basic use of S4 Methods and Classes*

Description

The majority of applications using methods and classes will be in R packages implementing new computations for an application, using new *classes* of objects that represent the data and results. Computations will be implemented using *methods* that implement functional computations when one or more of the arguments is an object from these classes.

Calls to the functions `setClass()` define the new classes; calls to `setMethod` define the methods. These, along with ordinary R computations, are sufficient to get started for most applications.

Classes are defined in terms of the data in them and what other classes of data they inherit from. Section 'Defining Classes' outlines the basic design of new classes.

Methods are R functions, often implementing basic computations as they apply to the new classes of objects. Section 'Defining Methods' discusses basic requirements and special tools for defining methods.

The classes discussed here are the original functional classes. R also supports formal classes and methods similar to those in other languages such as Python, in which methods are part of class definitions and invoked on an object. These are more appropriate when computations expect references to objects that are persistent, making changes to the object over time. See ReferenceClasses and Chapter 9 of the reference for the choice between these and S4 classes.

Defining Classes

All objects in R belong to a class; ordinary vectors and other basic objects are built-in (builtin-class). A new class is defined in terms of the named *slots* that is has and/or in terms of existing classes that it inherits from, or *contains* (discussed in 'Class Inheritance' below). A call to `setClass()` names a new class and uses the corresponding arguments to define it.

For example, suppose we want a class of objects to represent a collection of positions, perhaps from GPS readings. A natural way to think of these in R would have vectors of numeric values for latitude, longitude and altitude. A class with three corresponding slots could be defined by:

```
Pos <- setClass("Pos", slots = c(latitude = "numeric", longitude = "numeric",
altitude = "numeric"))
```

The value returned is a function, typically assigned as here with the name of the class. Calling this function returns an object from the class; its arguments are named with the slot names. If a function in the class had read the corresponding data, perhaps from a CSV file or from a data base, it could return an object from the class by:

```
Pos(latitude = x, longitude = y, altitude = z)
```

The slots are accessed by the `@` operator; for example, if g is an object from the class, `g@latitude`.

In addition to returning a generator function the call to `setClass()` assigns a definition of the class in a special metadata object in the package's namespace. When the package is loaded into an R session, the class definition is added to a table of known classes.

To make the class and the generating function publicly available, the package should include `POS` in `exportClasses()` and `export()` directives in its `NAMESPACE` file:

```
exportClasses(Pos); export(Pos)
```

Defining Methods

Defining methods for an R function makes that function *generic*. Instead of a call to the function always being carried out by the same method, there will be several alternatives. These are selected by matching the classes of the arguments in the call to a table in the generic function, indexed by classes for one or more formal arguments to the function, known as the *signatures* for the methods.

A method definition then specifies three things: the name of the function, the signature and the method definition itself. The definition must be a function with the same formal arguments as the generic.

For example, a method to make a plot of an object from class `"Pos"` could be defined by:

```
setMethod("plot", c("Pos", "missing"), function(x, y, ...) { plotPos(x, y) })
```

This method will match a call to `plot()` if the first argument is from class `"Pos"` or a subclass of that. The second argument must be missing; only a missing argument matches that class in the signature. Any object will match class `"ANY"` in the corresponding position of the signature.

Class Inheritance

A class may inherit all the slots and methods of one or more existing classes by specifying the names of the inherited classes in the `contains` = argument to `setClass()`.

To define a class that extends class `"Pos"` to a class `"GPS"` with a slot for the observation times:

```
GPS <- setClass("GPS", slots = c(time = "POSIXt"), contains = "Pos")
```

The inherited classes may be S4 classes, S3 classes or basic data types. S3 classes need to be identified as such by a call to `setOldClass()`; most S3 classes in the base package and many in the other built-in packages are already declared, as is `"POSIXt"`. If it had not been, the application package should contain:

```
setOldClass("POSIXt")
```

Inheriting from one of the R types is special. Objects from the new class will have the same type. A class `Currency` that contains numeric data plus a slot `"unit"` would be created by

```
Currency <- setClass("Currency", slots = c(unit = "character"), contains =
"numeric")
```

Objects created from this class will have type `"numeric"` and inherit all the builtin arithmetic and other computations for that type. Classes can only inherit from at most one such type; if the class does not inherit from a type, objects from the class will have type `"S4"`.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`is` *Is an Object from a Class?*

Description

Functions to test inheritance relationships between an object and a class or between two classes (`extends`).

Usage

```
is(object, class2)

extends(class1, class2, maybe = TRUE, fullInfo = FALSE)
```

Arguments

`object`	any R object.
`class1`, `class2`	character strings giving the names of each of the two classes between which `is` relations are to be examined, or (more efficiently) the class definition objects for the classes.
`fullInfo`	In a call to `extends`, with `class2` missing, `fullInfo` is a flag, which if `TRUE` causes a list of objects of class `SClassExtension` to be returned, rather than just the names of the classes. Only the distance slot is likely to be useful in practice; see the 'Selecting Superclasses' section;
`maybe`	What to return for conditional inheritance. But such relationships are rarely used and not recommended, so this argument should not be needed.

Selecting Superclasses

A call to `selectSuperClasses(cl)` returns a list of superclasses, similarly to `extends(cl)`. Additional arguments restrict the class names returned to direct superclasses and/or to non-virtual classes.

Either way, programming with the result, particularly using `sapply`, can be useful.

To find superclasses with more generally defined properties, one can program with the result returned by `extends` when called with one class as argument. By default, the call returns a character vector including the name of the class itself and of all its superclasses. Alternatively, if `extends` is called with `fullInfo = TRUE`, the return value is a named list, its names being the previous character vector. The elements of the list corresponding to superclasses are objects of class `SClassExtension`. Of the information in these objects, one piece can be useful: the number of generations between the classes, given by the `"distance"` slot.

Programming with the result of the call to `extends`, particularly using `sapply`, can select superclasses. The programming technique is to define a test function that returns `TRUE` for superclasses or relationships obeying some requirement. For example, to find only next-to-direct superclasses, use this function with the list of extension objects:

```
function(what) is(what, "SClassExtension") && what@distance == 2
```

or, to find only superclasses from "myPkg", use this function with the simple vector of names:

`function(what) getClassDef(what)@package == "myPkg"`

Giving such functions as an argument to `sapply` called on the output of `extends` allows you to find superclasses with desired properties. See the examples below.

Note that the function using extension objects must test the class of its argument since, unfortunately for this purpose, the list returned by `extends` includes `class1` itself, as the object `TRUE`.

Note

Prior to R 4.2.0 the code used the first elements of `class1` and `class2`, silently, These are now required to be length-one character vectors.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Although `inherits` is defined for S3 classes, it has been modified so that the result returned is nearly always equivalent to `is`, both for S4 and non-S4 objects. Since it is implemented in C, it is somewhat faster. The only non-equivalences arise from use of `setIs`, which should rarely be encountered.

Examples

```
## Not run:
## this example can be run if package XRPython from CRAN is installed.
supers <- extends("PythonInterface")
## find all the superclasses from package XR
fromXR <- sapply(supers,
    function(what) getClassDef(what)@package == "XR")
## print them
supers[fromXR]

## find all the superclasses at distance 2
superRelations <- extends("PythonInterface", fullInfo = TRUE)
dist2 <- sapply(superRelations,
    function(what) is(what, "SClassExtension") && what@distance == 2)
## print them
names(superRelations)[dist2]


## End(Not run)
```

`isSealedMethod` *Check for a Sealed Method or Class*

Description

These functions check for either a method or a class that has been *sealed* when it was defined, and which therefore cannot be re-defined.

Usage

```
isSealedMethod(f, signature, fdef, where)
isSealedClass(Class, where)
```

Arguments

`f`	The quoted name of the generic function.
`signature`	The class names in the method's signature, as they would be supplied to `setMethod`.
`fdef`	Optional, and usually omitted: the generic function definition for `f`.
`Class`	The quoted name of the class.
`where`	where to search for the method or class definition. By default, searches from the top environment of the call to `isSealedMethod` or `isSealedClass`, typically the global environment or the namespace of a package containing a call to one of the functions.

Details

In the R implementation of classes and methods, it is possible to seal the definition of either a class or a method. The basic classes (numeric and other types of vectors, matrix and array data) are sealed. So also are the methods for the primitive functions on those data types. The effect is that programmers cannot re-define the meaning of these basic data types and computations. More precisely, for primitive functions that depend on only one data argument, methods cannot be specified for basic classes. For functions (such as the arithmetic operators) that depend on two arguments, methods can be specified if *one* of those arguments is a basic class, but not if both are.

Programmers can seal other class and method definitions by using the `sealed` argument to `setClass` or `setMethod`.

Value

The functions return `FALSE` if the method or class is not sealed (including the case that it is not defined); `TRUE` if it is.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Examples

```
## these are both TRUE
isSealedMethod("+", c("numeric", "character"))
isSealedClass("matrix")

setClass("track", slots = c(x="numeric", y="numeric"))
## but this is FALSE
isSealedClass("track")
## and so is this
isSealedClass("A Name for an undefined Class")
## and so are these, because only one of the two arguments is basic
isSealedMethod("+", c("track", "numeric"))
isSealedMethod("+", c("numeric", "track"))
```

---

language-class *Classes to Represent Unevaluated Language Objects*

Description

The virtual class "language" and the specific classes that extend it represent unevaluated objects, as produced for example by the parser or by functions such as quote.

Usage

```
### each of these classes corresponds to an unevaluated object
### in the S language.
### The class name can appear in method signatures,
### and in a few other contexts (such as some calls to as()).

"("
"<-"
"call"
"for"
"if"
"repeat"
"while"
"name"
"{"

### Each of the classes above extends the virtual class
"language"
```

Objects from the Class

"language" is a virtual class; no objects may be created from it.

Objects from the other classes can be generated by a call to `new(Class, ...)`, where `Class` is the quoted class name, and the ... arguments are either empty or a *single* object that is from this class (or an extension).

Methods

coerce `signature(from = "ANY", to = "call")`. A method exists for `as(object, "call")`, calling `as.call()`.

Examples

```
showClass("language")

is( quote(sin(x)) ) # "call"  "language"

(ff <- new("if"))  ; is(ff) # "if" "language"
(ff <- new("for")) ; is(ff) # "for" "language"
```

LinearMethodsList-class

Class `"LinearMethodsList"`

Description

A version of methods lists that has been 'linearized' for producing summary information. The actual objects from class `"MethodsList"` used for method dispatch are defined recursively over the arguments involved.

Objects from the Class

The function `linearizeMlist` converts an ordinary methods list object into the linearized form.

Slots

`methods`**:** Object of class `"list"`, the method definitions.

`arguments`**:** Object of class `"list"`, the corresponding formal arguments, namely as many of the arguments in the signature of the generic function as are active in the relevant method table.

`classes`**:** Object of class `"list"`, the corresponding classes in the signatures.

`generic`**:** Object of class `"genericFunction"`; the generic function to which the methods correspond.

Future Note

The current version of `linearizeMlist` does not take advantage of the `MethodDefinition` class, and therefore does more work for less effect than it could. In particular, we may move to redefine both the function and the class to take advantage of the stored signatures. Don't write code depending precisely on the present form, although all the current information will be obtainable in the future.

See Also

Function `linearizeMlist` for the computation, and class `MethodsList` for the original, recursive form.

`LocalReferenceClasses` *Localized Objects based on Reference Classes*

Description

Local reference classes are modified ReferenceClasses that isolate the objects to the local frame. Therefore, they do *not* propagate changes back to the calling environment. At the same time, they use the reference field semantics locally, avoiding the automatic duplication applied to standard R objects.

The current implementation has no special construction. To create a local reference class, call `setRefClass()` with a `contains=` argument that includes `"localRefClass"`. See the example below.

Local reference classes operate essentially as do regular, functional classes in R; that is, changes are made by assignment and take place in the local frame. The essential difference is that replacement operations (like the change to the `twiddle` field in the example) do not cause duplication of the entire object, as would be the case for a formal class or for data with attributes or in a named list. The purpose is to allow large objects in some fields that are not changed along with potentially frequent changes to other fields, but without copying the large fields.

Usage

```
setRefClass(Class, fields = , contains = c("localRefClass",....),
     methods =, where =, ...)
```

Details

Localization of objects is only partially automated in the current implementation. Replacement expressions using the `$<-` operator are safe.

However, if reference methods for the class themselves modify fields, using `<<-`, for example, then one must ensure that the object is local to the relevant frame before any such method is called. Otherwise, standard reference class behavior still prevails.

There are two ways to ensure locality. The direct way is to invoke the special method `x$ensureLocal()` on the object. The other way is to modify a field explicitly by `x$field <- ...` It's only necessary that one or the other of these happens once for each object, in order to trigger the shallow copy that provides locality for the references. In the example below, we show both mechanisms.

However it's done, localization must occur *before* any methods make changes. (Eventually, some use of code tools should at least largely automate this process, although it may be difficult to guarantee success under arbitrary circumstances.)

Author(s)

John Chambers

Examples

```
## class "myIter" has a BigData field for the real (big) data
## and a "twiddle" field for some parameters that it twiddles
## ( for some reason)

myIter <- setRefClass("myIter", contains = "localRefClass",
  fields = list(BigData = "numeric", twiddle = "numeric"))

tw <- rnorm(3)
x1 <- myIter(BigData = rnorm(1000), twiddle = tw) # OK, not REALLY big

twiddler <- function(x, n) {
  x$ensureLocal() # see the Details.  Not really needed in this example
  for(i in seq_len(n)) {
      x$twiddle <- x$twiddle + rnorm(length(x$twiddle))
      ## then do something ....
      ## Snooping in gdb, etc will show that x$BigData is not copied
  }
  return(x)
}

x2 <- twiddler(x1, 10)

stopifnot(identical(x1$twiddle, tw), !identical(x1$twiddle, x2$twiddle))
```

---

makeClassRepresentation

Create a Class Definition

Description

Constructs an object of class `classRepresentation` to describe a particular class. Mostly a utility function, but you can call it to create a class definition without assigning it, as `setClass` would do.

Usage

```
makeClassRepresentation(name, slots=list(), superClasses=character(),
                        prototype=NULL, package, validity, access,
                        version, sealed, virtual=NA, where)
```

Arguments

name	character string name for the class
slots	named list of slot classes as would be supplied to setClass, but *without* the unnamed arguments for superClasses if any.
superClasses	what classes does this class extend
prototype	an object providing the default data for the class, e.g., the result of a call to prototype.
package	The character string name for the package in which the class will be stored; see getPackageName.
validity	Optional validity method. See validObject, and the discussion of validity methods in the reference.
access	Access information. Not currently used.
version	Optional version key for version control. Currently generated, but not used.
sealed	Is the class sealed? See setClass.
virtual	Is this known to be a virtual class?
where	The environment from which to look for class definitions needed (e.g., for slots or superclasses). See the discussion of this argument under GenericFunctions.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

setClass

---

method.skeleton *Create a Skeleton File for a New Method*

Description

This function writes a source file containing a call to setMethod to define a method for the generic function and signature supplied. By default the method definition is in line in the call, but can be made an external (previously assigned) function.

Usage

```
method.skeleton(generic, signature, file, external = FALSE, where)
```

Arguments

generic	the character string name of the generic function, or the generic function itself. In the first case, the function need not currently be a generic, as it would not for the resulting call to setMethod.
signature	the method signature, as it would be given to setMethod
file	a character string name for the output file, or a writable connection. By default the generic function name and the classes in the signature are concatenated, with separating underscore characters. The file name should normally end in ".R". To write multiple method skeletons to one file, open the file connection first and then pass it to method.skeleton() in multiple calls.
external	flag to control whether the function definition for the method should be a separate external object assigned in the source file, or included in line in the call to setMethod. If supplied as a character string, this will be used as the name for the external function; by default the name concatenates the generic and signature names, with separating underscores.
where	The environment in which to look for the function; by default, the top-level environment of the call to method.skeleton.

Value

The file argument, invisibly, but the function is used for its side effect.

See Also

setMethod, package.skeleton

Examples

```
setClass("track", slots = c(x ="numeric", y="numeric"))
method.skeleton("show", "track")            ## writes show_track.R
method.skeleton("Ops", c("track", "track")) ## writes "Ops_track_track.R"

## write multiple method skeletons to one file
con <- file("./Math_track.R", "w")
method.skeleton("Math", "track", con)
method.skeleton("exp", "track", con)
method.skeleton("log", "track", con)
close(con)
```

MethodDefinition-class

Classes to Represent Method Definitions

Description

These classes extend the basic class "function" when functions are to be stored and used as method definitions.

Details

Method definition objects are functions with additional information defining how the function is being used as a method. The target slot is the class signature for which the method will be dispatched, and the defined slot the signature for which the method was originally specified (that is, the one that appeared in some call to setMethod).

Objects from the Class

The action of setting a method by a call to setMethod creates an object of this class. It's unwise to create them directly.

The class "SealedMethodDefinition" is created by a call to setMethod with argument sealed = TRUE. It has the same representation as "MethodDefinition".

Slots

.Data: Object of class "function"; the data part of the definition.

target: Object of class "signature"; the signature for which the method was wanted.

defined: Object of class "signature"; the signature for which a method was found. If the method was inherited, this will not be identical to target.

generic: Object of class "character"; the function for which the method was created.

Extends

Class "function", from data part.
Class "PossibleMethod", directly.
Class "OptionalMethods", by class "function".

See Also

class MethodsList for the objects defining sets of methods associated with a particular generic function. The individual method definitions stored in these objects are from class MethodDefinition, or an extension. Class MethodWithNext for an extension used by callNextMethod.

Methods *S4 Class Documentation*

Description

You have navigated to an old link to documentation of S4 methods.

For basic use of classes and methods, see Introduction; to create new method definitions, see `setMethod`; for technical details on S4 methods, see Methods_Details.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

MethodsList-class *Class "MethodsList", Defunct Representation of Methods*

Description

This class of objects was used in the original implementation of the package to control method dispatch. Its use is now defunct, but object appear as the default method slot in generic functions. This and any other remaining uses will be removed in the future.

For the modern alternative, see listOfMethods.

The details in this documentation are retained to allow analysis of old-style objects.

Details

Suppose a function `f` has formal arguments `x` and `y`. The methods list object for that function has the object `as.name("x")` as its `argument` slot. An element of the methods named `"track"` is selected if the actual argument corresponding to `x` is an object of class `"track"`. If there is such an element, it can generally be either a function or another methods list object.

In the first case, the function defines the method to use for any call in which `x` is of class `"track"`. In the second case, the new methods list object defines the available methods depending on the remaining formal arguments, in this example, `y`.

Each method corresponds conceptually to a *signature*; that is a named list of classes, with names corresponding to some or all of the formal arguments. In the previous example, if selecting class `"track"` for `x`, finding that the selection was another methods list and then selecting class `"numeric"` for `y` would produce a method associated with the signature `x = "track", y = "numeric"`.

Slots

`argument`: Object of class `"name"`. The name of the argument being used for dispatch at this level.

`methods`: A named list of the methods (and method lists) defined *explicitly* for this argument. The names are the names of classes, and the corresponding element defines the method or methods to be used if the corresponding argument has that class. See the details below.

`allMethods`: A named list, contains all the directly defined methods from the `methods` slot, plus any inherited methods. Ignored when methods tables are used for dispatch (see Methods_Details).

Extends

Class `"OptionalMethods"`, directly.

Methods_Details *General Information on Methods*

Description

This documentation covers some general topics on how methods work and how the **methods** package interacts with the rest of R. The information is usually not needed to get started with methods and classes, but may be helpful for moderately ambitious projects, or when something doesn't work as expected.

For additional information see documentation for the important steps: (`setMethod()`, `setClass()` and `setGeneric()`). Also `Methods_for_Nongenerics` on defining formal methods for functions that are not currently generic functions; Methods_for_S3 for the relation to S3 classes and methods; `Classes_Details` for class definitions and Chapters 9 and 10 of the reference.

How Methods Work

A call to a generic function selects a method matching the actual arguments in the call. The body of the method is evaluated in the frame of the call to the generic function. A generic function is identified by its name and by the package to which it correspond. Unlike ordinary functions, the generic has a slot that specifies its package.

In an R session, there is one version of each such generic, regardless of where the call to that generic originated, and the generic function has a table of all the methods currently available for it; that is, all the methods in packages currently loaded into the session.

Methods are frequently defined for functions that are non-generic in their original package,. for example, for function `plot()` in package **graphics**. An identical version of the corresponding generic function may exist in several packages. All methods will be dispatched consistently from the R session.

Each R package with a call to `setMethod` in its source code will include a methods metadata object for that generic. When the package is loaded into an R session, the methods for each generic function are *cached*, that is, added to the environment of the generic function. This merged table of methods is used to dispatch or select methods from the generic, using class inheritance and possibly group generic functions (see `GroupGenericFunctions`) to find an applicable method. See

the "Method Selection and Dispatch" section below. The caching computations ensure that only one version of each generic function is visible globally; although different attached packages may contain a copy of the generic function, these behave identically with respect to method selection.

In contrast, it is possible for the same function name to refer to more than one generic function, when these have different `package` slots. In the latter case, R considers the functions unrelated: A generic function is defined by the combination of name and package. See the "Generic Functions" section below.

The methods for a generic are stored according to the corresponding `signature` in the call to `setMethod` that defined the method. The signature associates one class name with each of a subset of the formal arguments to the generic function. Which formal arguments are available, and the order in which they appear, are determined by the `"signature"` slot of the generic function itself. By default, the signature of the generic consists of all the formal arguments except ..., in the order they appear in the function definition.

Trailing arguments in the signature of the generic will be *inactive* if no method has yet been specified that included those arguments in its signature. Inactive arguments are not needed or used in labeling the cached methods. (The distinction does not change which methods are dispatched, but ignoring inactive arguments improves the efficiency of dispatch.)

All arguments in the signature of the generic function will be evaluated when the function is called, rather than using lazy evaluation. Therefore, it's important to *exclude* from the signature any arguments that need to be dealt with symbolically (such as the `expr` argument to function `with`). Note that only actual arguments are evaluated, not default expressions. A missing argument enters into the method selection as class `"missing"`.

The cached methods are stored in an environment object. The names used for assignment are a concatenation of the class names for the active arguments in the method signature.

Method Selection: Details

When a call to a generic function is evaluated, a method is selected corresponding to the classes of the actual arguments in the signature. First, the cached methods table is searched for an exact match; that is, a method stored under the signature defined by the string value of `class(x)` for each non-missing argument, and `"missing"` for each missing argument. If no method is found directly for the actual arguments in a call to a generic function, an attempt is made to match the available methods to the arguments by using the superclass information about the actual classes. A method found by this search is cached in the generic function so that future calls with the same argument classes will not require repeating the search. In any likely application, the search for inherited methods will be a negligible overhead.

Each class definition may include a list of one or more direct *superclasses* of the new class. The simplest and most common specification is by the `contains=` argument in the call to `setClass`. Each class named in this argument is a superclass of the new class. A class will also have as a direct superclass any class union to which it is a member. Class unions are created by a call to `setClassUnion`. Additional members can be added to the union by a simple call to `setIs`. Superclasses specified by either mechanism are the *direct* superclasses.

Inheritance specified in either of these forms is *simple* in the sense that all the information needed for the superclass is asserted to be directly available from the object. R inherited from S a more general form of inheritance in which inheritance may require some transformation or be conditional on a test. This more general form has not proved to be useful in general practical situations. Since

it also adds some computational costs non-simple inheritance is not recommended. See `setIs` for the general version.

The direct superclasses themselves may have direct superclasses and similarly through further generations. Putting all this information together produces the full list of superclasses for this class. The superclass list is included in the definition of the class that is cached during the R session. The *distance* between the two classes is defined to be the number of generations: 1 for direct superclasses (regardless of which mechanism defined them), then 2 for the direct superclasses of those classes, and so on. To see all the superclasses, with their distance, print the class definition by calling `getClass`. In addition, any class implicitly has class `"ANY"` as a superclass. The distance to `"ANY"` is treated as larger than the distance to any actual class. The special class `"missing"` corresponding to missing arguments has only `"ANY"` as a superclass, while `"ANY"` has no superclasses.

When a method is to be selected by inheritance, a search is made in the table for all methods corresponding to a combination of either the direct class or one of its superclasses, for each argument in the active signature. For an example, suppose there is only one argument in the signature and that the class of the corresponding object was `"dgeMatrix"` (from the recommended package `Matrix`). This class has (currently) three direct superclasses and through these additional superclasses at distances 2 through 4. A method that had been defined for any of these classes or for class `"ANY"` (the default method) would be eligible. Methods for the shortest difference are preferred. If there is only one best method in this sense, method selection is unambiguous.

When there are multiple arguments in the signature, each argument will generate a similar list of inherited classes. The possible matches are now all the combinations of classes from each argument (think of the function `outer` generating an array of all possible combinations). The search now finds all the methods matching any of this combination of classes. For each argument, the distance to the superclass defines which method(s) are preferred for that argument. A method is considered best for selection if it is among the best (i.e., has the least distance) for each argument.

The end result is that zero, one or more methods may be "best". If one, this method is selected and cached in the table of methods. If there is more than one best match, the selection is ambiguous and a message is printed noting which method was selected (the first method lexicographically in the ordering) and what other methods could have been selected. Since the ambiguity is usually nothing the end user could control, this is not a warning. Package authors should examine their package for possible ambiguous inheritance by calling `testInheritedMethods`.

Cached inherited selections are not themselves used in future inheritance searches, since that could result in invalid selections. If you want inheritance computations to be done again (for example, because a newly loaded package has a more direct method than one that has already been used in this session), call `resetGeneric`. Because classes and methods involving them tend to come from the same package, the current implementation does not reset all generics every time a new package is loaded.

Besides being initiated through calls to the generic function, method selection can be done explicitly by calling the function `selectMethod`. Note that some computations may use this function directly, with optional arguments. The prime example is the use of `coerce()` methods by function `as()`. There has been some confusion from comparing coerce methods to a call to `selectMethod` with other options.

Method Evaluation: Details

Once a method has been selected, the evaluator creates a new context in which a call to the method is evaluated. The context is initialized with the arguments from the call to the generic function.

These arguments are not rematched. All the arguments in the signature of the generic will have been evaluated (including any that are currently inactive); arguments that are not in the signature will obey the usual lazy evaluation rules of the language. If an argument was missing in the call, its default expression if any will *not* have been evaluated, since method dispatch always uses class `missing` for such arguments.

A call to a generic function therefore has two contexts: one for the function and a second for the method. The argument objects will be copied to the second context, but not any local objects created in a nonstandard generic function. The other important distinction is that the parent ("enclosing") environment of the second context is the environment of the method as a function, so that all R programming techniques using such environments apply to method definitions as ordinary functions.

For further discussion of method selection and dispatch, see the references in the sections indicated.

Generic Functions

In principle, a generic function could be any function that evaluates a call to `standardGeneric()`, the internal function that selects a method and evaluates a call to the selected method. In practice, generic functions are special objects that in addition to being from a subclass of class `"function"` also extend the class `genericFunction`. Such objects have slots to define information needed to deal with their methods. They also have specialized environments, containing the tables used in method selection.

The slots `"generic"` and `"package"` in the object are the character string names of the generic function itself and of the package from which the function is defined. As with classes, generic functions are uniquely defined in R by the combination of the two names. There can be generic functions of the same name associated with different packages (although inevitably keeping such functions cleanly distinguished is not always easy). On the other hand, R will enforce that only one definition of a generic function can be associated with a particular combination of function and package name, in the current session or other active version of R.

Tables of methods for a particular generic function, in this sense, will often be spread over several other packages. The total set of methods for a given generic function may change during a session, as additional packages are loaded. Each table must be consistent in the signature assumed for the generic function.

R distinguishes *standard* and *nonstandard* generic functions, with the former having a function body that does nothing but dispatch a method. For the most part, the distinction is just one of simplicity: knowing that a generic function only dispatches a method call allows some efficiencies and also removes some uncertainties.

In most cases, the generic function is the visible function corresponding to that name, in the corresponding package. There are two exceptions, *implicit* generic functions and the special computations required to deal with R's *primitive* functions. Packages can contain a table of implicit generic versions of functions in the package, if the package wishes to leave a function non-generic but to constrain what the function would be like if it were generic. Such implicit generic functions are created during the installation of the package, essentially by defining the generic function and possibly methods for it, and then reverting the function to its non-generic form. (See implicitGeneric for how this is done.) The mechanism is mainly used for functions in the older packages in R, which may prefer to ignore S4 methods. Even in this case, the actual mechanism is only needed if something special has to be specified. All functions have a corresponding implicit generic version defined automatically (an implicit, implicit generic function one might say). This function is a standard generic with the same arguments as the non-generic function, with the non-generic version as

the default (and only) method, and with the generic signature being all the formal arguments except ....

The implicit generic mechanism is needed only to override some aspect of the default definition. One reason to do so would be to remove some arguments from the signature. Arguments that may need to be interpreted literally, or for which the lazy evaluation mechanism of the language is needed, must *not* be included in the signature of the generic function, since all arguments in the signature will be evaluated in order to select a method. For example, the argument `expr` to the function `with` is treated literally and must therefore be excluded from the signature.

One would also need to define an implicit generic if the existing non-generic function were not suitable as the default method. Perhaps the function only applies to some classes of objects, and the package designer prefers to have no general default method. In the other direction, the package designer might have some ideas about suitable methods for some classes, if the function were generic. With reasonably modern packages, the simple approach in all these cases is just to define the function as a generic. The implicit generic mechanism is mainly attractive for older packages that do not want to require the methods package to be available.

Generic functions will also be defined but not obviously visible for functions implemented as *primitive* functions in the base package. Primitive functions look like ordinary functions when printed but are in fact not function objects but objects of two types interpreted by the R evaluator to call underlying C code directly. Since their entire justification is efficiency, R refuses to hide primitives behind a generic function object. Methods may be defined for most primitives, and corresponding metadata objects will be created to store them. Calls to the primitive still go directly to the C code, which will sometimes check for applicable methods. The definition of "sometimes" is that methods must have been detected for the function in some package loaded in the session and `isS4(x)` is `TRUE` for the first argument (or for the second argument, in the case of binary operators). You can test whether methods have been detected by calling `isGeneric` for the relevant function and you can examine the generic function by calling `getGeneric`, whether or not methods have been detected. For more on generic functions, see the references and also section 2 of the *R Internals* document supplied with R.

Method Definitions

All method definitions are stored as objects from the `MethodDefinition` class. Like the class of generic functions, this class extends ordinary R functions with some additional slots: `"generic"`, containing the name and package of the generic function, and two signature slots, `"defined"` and `"target"`, the first being the signature supplied when the method was defined by a call to `setMethod`. The `"target"` slot starts off equal to the `"defined"` slot. When an inherited method is cached after being selected, as described above, a copy is made with the appropriate `"target"` signature. Output from `showMethods`, for example, includes both signatures.

Method definitions are required to have the same formal arguments as the generic function, since the method dispatch mechanism does not rematch arguments, for reasons of both efficiency and consistency.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.5 for some details.)

See Also

For more specific information, see `setGeneric`, `setMethod`, and `setClass`.

For the use of ... in methods, see dotsMethods.

`Methods_for_Nongenerics`

Methods for Non-Generic Functions in Other Packages

Description

In writing methods for an R package, it's common for these methods to apply to a function (in another package) that is not generic in that package; that is, there are no formal methods for the function in its own package, although it may have S3 methods. The programming in this case involves one extra step, to call `setGeneric()` to declare that the function *is* generic in your package.

Calls to the function in your package will then use all methods defined there or in any other loaded package that creates the same generic function. Similarly, calls to the function in those packages will use your methods.

The original version, however, remains non-generic. Calls in that package or in other packages that use that version will not dispatch your methods except for special circumstances:

1. If the function is one of the primitive functions that accept methods, the internal C implementation will dispatch methods if one of the arguments is an S4 object, as should be the case.
2. If the other version of the function dispatches S3 methods *and* your methods are also registered as S3 methods, the method will usually be dispatched as that S3 method.
3. Otherwise, you will need to ensure that all calls to the function come from a package in which the function is generic, perhaps by copying code to your package.

Details and the underlying reasons are discussed in the following sections.

Generic and Non-Generic Calls

Creating methods for a function (any function) in a package means that calls to the function in that package will select methods according to the actual arguments. However, if the function was originally a non-generic in another package, calls to the function from that package will *not* dispatch methods. In addition, calls from any third package that imports the non-generic version will also not dispatch methods. This section considers the reason and how one might deal with the consequences.

The reason is simply the R namespace mechanism and its role in evaluating function calls. When a name (such as the name of a function) needs to be evaluated in a call to a function from some package, the evaluator looks first in the frame of the call, then in the namespace of the package and then in the imports to that package.

Defining methods for a function in a package ensures that calls to the function in that package will select the methods, because a generic version of the function is created in the namespace. Similarly, calls from another package that has or imports the generic version will select methods. Because the generic versions are identical, all methods will be available in all these packages.

However, calls from any package that imports the old version or just selects it from the search list will usually *not* select methods.

A an example, consider the function `data.frame()` in the base package. This function takes any number of objects as arguments and attempts to combine them as variables into a data frame object. It does this by calling `as.data.frame()`, also in the base package, for each of the objects.

A reasonable goal would be to extend the classes of objects that can be included in a data frame by defining methods for `as.data.frame()`. But calls to `data.frame()`, will still use the version of that function in the base package, which continues to call the non-generic `as.data.frame()` in that package.

The details of what happens and options for dealing with it depend on the form of the function: a primitive function; a function that dispatches S3 methods; or an ordinary R function.

Primitive functions are not actual R function objects. They go directly to internal C code. Some of them, however, have been implemented to recognize methods. These functions dispatch both S4 and S3 methods from the internal C code. There is no explicit generic function, either S3 or S4. The internal code looks for S4 methods if the first argument, or either of the arguments in the case of a binary operator, is an S4 object. If no S4 method is found, a search is made for an S3 method. So defining methods for these functions works as long as the relevant classes have been defined, which should always be the case.

A function dispatches S3 methods by calling `UseMethod()`, which does *not* look for formal methods regardless of whether the first argument is an S4 object or not. This applies to the `as.data.frame()` example above. To have methods called in this situation, your package must also define the method as an S3 method, if possible. See section 'S3 "Generic" Functions'.

In the third possibility, the function is defined with no expectation of methods. For example, the base package has a number of functions that compute numerical decompositions of matrix arguments. Some, such as `chol()` and `qr()` are implemented to dispatch S3 methods; others, such as `svd()` are implemented directly as a specific computation. A generic version of the latter functions can be written and called directly to define formal methods, but no code in another package that does not import this generic version will dispatch such methods.

In this case, you need to have the generic version used in all the indirect calls to the function supplying arguments that should dispatch methods. This may require supplying new functions that dispatch methods and then call the function they replace. For example, if S3 methods did not work for `as.data.frame()`, one could call a function that applied the generic version to all its arguments and then called `data.frame()` as a replacement for that function. If all else fails, it might be necessary to copy over the relevant functions so that they would find the generic versions.

S3 "Generic" Functions

S3 method dispatch looks at the class of the first argument. S3 methods are ordinary functions with the same arguments as the generic function. The "signature" of an S3 method is identified by the name to which the method is assigned, composed of the name of the generic function, followed by `"."`, followed by the name of the class. For details, see `UseMethod`.

To implement a method for one of these functions corresponding to S4 classes, there are two possibilities: either an S4 method or an S3 method with the S4 class name. The S3 method is only possible if the intended signature has the first argument and nothing else. In this case, the recommended approach is to define the S3 method and also supply the identical function as the definition

of the S4 method. If the S3 generic function was `f3(x, ...)` and the S4 class for the new method was `"myClass"`:

```
f3.myClass <- function(x, ...) { ..... }

setMethod("f3", "myClass", f3.myClass)
```

Defining both methods usually ensures that all calls to the original function will dispatch the intended method. The S4 method alone would not be called from other packages using the original version of the function. On the other hand, an S3 method alone will not be called if there is *any* eligible non-default S4 method.

S4 and S3 method selection are designed to follow compatible rules of inheritance, as far as possible. S3 classes can be used for any S4 method selection, provided that the S3 classes have been registered by a call to `setOldClass`, with that call specifying the correct S3 inheritance pattern. S4 classes can be used for any S3 method selection; when an S4 object is detected, S3 method selection uses the contents of `extends(class(x))` as the equivalent of the S3 inheritance (the inheritance is cached after the first call).

An existing S3 method may not behave as desired for an S4 subclass, in which case utilities such as `asS3` and `S3Part` may be useful. If the S3 method fails on the S4 object, `asS3(x)` may be passed instead; if the object returned by the S3 method needs to be incorporated in the S4 object, the replacement function for `S3Part` may be useful.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Methods_for_S3 for suggested implementation of methods that work for both S3 and S4 dispatch.

Examples

```
## A class that extends a registered S3 class inherits that class' S3
## methods.

setClass("myFrame", contains = "data.frame",
         slots = c(timestamps = "POSIXt"))
df1 <- data.frame(x = 1:10, y = rnorm(10), z = sample(letters,10))
mydf1 <- new("myFrame", df1, timestamps = Sys.time())

## "myFrame" objects inherit "data.frame" S3 methods; e.g., for `[`

mydf1[1:2, ] # a data frame object (with extra attributes)


## a method explicitly for "myFrame" class

setMethod("[",
    signature(x = "myFrame"),
    function (x, i, j, ..., drop = TRUE)
    {
        S3Part(x) <- callNextMethod()
```

```
        x@timestamps <- c(Sys.time(), as.POSIXct(x@timestamps))
        x
    }
)

mydf1[1:2, ]


setClass("myDateTime", contains = "POSIXt")

now <- Sys.time() # class(now) is c("POSIXct", "POSIXt")
nowLt <- as.POSIXlt(now)# class(nowLt) is c("POSIXlt", "POSIXt")

mCt <- new("myDateTime", now)
mLt <- new("myDateTime", nowLt)

## S3 methods for an S4 object will be selected using S4 inheritance
## Objects mCt and mLt have different S3Class() values, but this is
## not used.
f3 <- function(x)UseMethod("f3") # an S3 generic to illustrate inheritance

f3.POSIXct <- function(x) "The POSIXct result"
f3.POSIXlt <- function(x) "The POSIXlt result"
f3.POSIXt <- function(x) "The POSIXt result"

stopifnot(identical(f3(mCt), f3.POSIXt(mCt)))
stopifnot(identical(f3(mLt), f3.POSIXt(mLt)))



## An S4 object selects S3 methods according to its S4 "inheritance"

setClass("classA", contains = "numeric",
         slots = c(realData = "numeric"))

Math.classA <- function(x) { (getFunction(.Generic))(x@realData) }
setMethod("Math", "classA", Math.classA)


x <- new("classA", log(1:10), realData = 1:10)

stopifnot(identical(abs(x), 1:10))

setClass("classB", contains = "classA")

y <- new("classB", x)

stopifnot(identical(abs(y), abs(x))) # (version 2.9.0 or earlier fails here)

## an S3 generic: just for demonstration purposes
f3 <- function(x, ...) UseMethod("f3")

f3.default <- function(x, ...) "Default f3"
```

```
## S3 method (only) for classA
f3.classA <- function(x, ...) "Class classA for f3"

## S3 and S4 method for numeric
f3.numeric <- function(x, ...) "Class numeric for f3"
setMethod("f3", "numeric", f3.numeric)

## The S3 method for classA and the closest inherited S3 method for classB
## are not found.

f3(x); f3(y) # both choose "numeric" method

## to obtain the natural inheritance, set identical S3 and S4 methods
setMethod("f3", "classA", f3.classA)

f3(x); f3(y) # now both choose "classA" method

## Need to define an S3 as well as S4 method to use on an S3 object
## or if called from a package without the S4 generic

MathFun <- function(x) { # a smarter "data.frame" method for Math group
  for (i in seq_len(ncol(x))[sapply(x, is.numeric)])
    x[, i] <- (getFunction(.Generic))(x[, i])
  x
}
setMethod("Math", "data.frame", MathFun)

## S4 method works for an S4 class containing data.frame,
## but not for data.frame objects (not S4 objects)

try(logIris <- log(iris)) #gets an error from the old method

## Define an S3 method with the same computation

Math.data.frame <- MathFun

logIris <- log(iris)
```

Methods_for_S3 *Methods For S3 and S4 Dispatch*

Description

The S3 and S4 software in R are two generations implementing functional object-oriented programming. S3 is the original, simpler for initial programming but less general, less formal and less open to validation. The S4 formal methods and classes provide these features but require more programming.

In modern R, the two versions attempt to work together. This documentation outlines how to write methods for both systems by defining an S4 method for a function that dispatches S3 methods.

The systems can also be combined by using an S3 class with S4 method dispatch or in S4 class definitions. See `setOldClass`.

S3 Method Dispatch

The R evaluator will 'dispatch' a method from a function call either when the body of the function calls the special primitive `UseMethod` or when the call is to one of the builtin primitives such as the `math` functions or the binary operators.

S3 method dispatch looks at the class of the first argument or the class of either argument in a call to one of the primitive binary operators. In pure S3 situations, 'class' in this context means the class attribute or the implied class for a basic data type such as `"numeric"`. The first S3 method that matches a name in the class is called and the value of that call is the value of the original function call. For details, see S3Methods.

In modern R, a function `meth` in a package is registered as an S3 method for function `fun` and class `Class` by including in the package's `NAMESPACE` file the directive

```
S3method(fun, Class, meth)
```

By default (and traditionally), the third argument is taken to be the function `fun.Class`; that is, the name of the generic function, followed by `"."`, followed by the name of the class.

As with S4 methods, a method that has been registered will be added to a table of methods for this function when the corresponding package is loaded into the session. Older versions of R, copying the mechanism in S, looked for the method in the current search list, but packages should now always register S3 methods rather than requiring the package to be attached.

Methods for S4 Classes

Two possible mechanisms for implementing a method corresponding to an S4 class, there are two possibilities are to register it as an S3 method with the S4 class name or to define and set an S4 method, which will have the side effect of creating an S4 generic version of this function.

For most situations either works, but the recommended approach is to do both: register the S3 method and supply the identical function as the definition of the S4 method. This ensures that the proposed method will be dispatched for any applicable call to the function.

As an example, suppose an S4 class `"uncased"` is defined, extending `"character"` and intending to ignore upper- and lower-case. The base function `unique` dispatches S3 methods. To define the class and a method for this function:

```
setClass("uncased", contains = "character")
unique.uncased <- function(x, incomparables = FALSE, ...) nextMethod(tolower(x))
setMethod("unique", "uncased", unique.uncased)
```

In addition, the NAMESPACE for the package should contain:

```
S3method(unique, uncased)
```

```
exportMethods(unique)
```

The result is to define identical S3 and S4 methods and ensure that all calls to unique will dispatch that method when appropriate.

Details

The reasons for defining both S3 and S4 methods are as follows:

1. An S4 method alone will not be seen if the S3 generic function is called directly. This will be the case, for example, if some function calls unique() from a package that does not make that function an S4 generic.

   However, primitive functions and operators are exceptions: The internal C code will look for S4 methods if and only if the object is an S4 object. S4 method dispatch would be used to dispatch any binary operator calls where either of the operands was an S4 object, for example.

2. An S3 method alone will not be called if there is *any* eligible non-default S4 method.

   So if a package defined an S3 method for unique for an S4 class but another package defined an S4 method for a superclass of that class, the superclass method would be chosen, probably not what was intended.

S4 and S3 method selection are designed to follow compatible rules of inheritance, as far as possible. S3 classes can be used for any S4 method selection, provided that the S3 classes have been registered by a call to setOldClass, with that call specifying the correct S3 inheritance pattern. S4 classes can be used for any S3 method selection; when an S4 object is detected, S3 method selection uses the contents of extends(class(x)) as the equivalent of the S3 inheritance (the inheritance is cached after the first call).

For the details of S4 and S3 dispatch see Methods_Details and S3Methods.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

MethodWithNext-class *Class* "MethodWithNext"

Description

Class of method definitions set up for callNextMethod

Objects from the Class

Objects from this class are generated as a side-effect of calls to callNextMethod.

Slots

`.Data`: Object of class `"function"`; the actual function definition.

`nextMethod`: Object of class `"PossibleMethod"` the method to use in response to a `callNextMethod()` call.

`excluded`: Object of class `"list"`; one or more signatures excluded in finding the next method.

`target`: Object of class `"signature"`, from class `"MethodDefinition"`

`defined`: Object of class `"signature"`, from class `"MethodDefinition"`

`generic`: Object of class `"character"`; the function for which the method was created.

Extends

Class `"MethodDefinition"`, directly.
Class `"function"`, from data part.
Class `"PossibleMethod"`, by class `"MethodDefinition"`.
Class `"OptionalMethods"`, by class `"MethodDefinition"`.

Methods

findNextMethod `signature(method = "MethodWithNext")`: used internally by method dispatch.

loadMethod `signature(method = "MethodWithNext")`: used internally by method dispatch.

show `signature(object = "MethodWithNext")`

See Also

`callNextMethod`, and class `MethodDefinition`.

---

new *Generate an Object from a Class*

Description

A call to `new` returns a newly allocated object from the class identified by the first argument. This call in turn calls the method for the generic function `initialize` corresponding to the specified class, passing the `...` arguments to this method. In the default method for `initialize()`, named arguments provide values for the corresponding slots and unnamed arguments must be objects from superclasses of this class.

A call to a generating function for a class (see `setClass`) will pass its ... arguments to a corresponding call to `new()`.

Usage

```
new(Class, ...)

initialize(.Object, ...)
```

Arguments

Class	either the name of a class, a character string, (the usual case) or the object describing the class (e.g., the value returned by getClass). Note that the character string passed from a generating function includes the package name as an attribute, avoiding ambiguity if two packages have identically named classes.
...	arguments to specify properties of the new object, to be passed to initialize().
.Object	An object: see the "Initialize Methods" section.

Initialize Methods

The generic function initialize is not called directly. A call to new begins by copying the prototype object from the class definition, and then calls intialize() with this object as the first argument, followed by the ... arguments.

The interpretation of the ... arguments in a call to a generator function or to new() can be specialized to particular classes, by defining an appropriate method for "initialize".

In the default method, unnamed arguments in the ... are interpreted as objects from a superclass, and named arguments are interpreted as objects to be assigned into the correspondingly named slots. Explicitly specified slots override inherited information for the same slot, regardless of the order in which the arguments appear.

The initialize methods do not have to have ... as their second argument (see the examples). Initialize methods are often written when the natural parameters describing the new object are not the names of the slots. If you do define such a method, you should include ... as a formal argument, and your method should pass such arguments along via callNextMethod. This helps the definition of future subclasses of your class. If these have additional slots and your method does *not* have this argument, it will be difficult for these slots to be included in an initializing call.

See initialize-methods for a discussion of some classes with existing methods.

Methods for initialize can be inherited only by simple inheritance, since it is a requirement that the method return an object from the target class. See the simpleInheritanceOnly argument to setGeneric and the discussion in setIs for the general concept.

Note that the basic vector classes, "numeric", etc. are implicitly defined, so one can use new for these classes. The ... arguments are interpreted as objects of this type and are concatenated into the resulting vector.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Classes_Details for details of class definitions, and setOldClass for the relation to S3 classes.

Examples

```
## using the definition of class "track" from \link{setClass}
```

```
## a new object with two slots specified
t1 <- new("track", x = seq_along(ydata), y = ydata)

# a new object including an object from a superclass, plus a slot
t2 <- new("trackCurve", t1, smooth = ysmooth)

### define a method for initialize, to ensure that new objects have
### equal-length x and y slots.  In this version, the slots must still be
### supplied by name.

setMethod("initialize", "track",
    function(.Object, ...) {
      .Object <- callNextMethod()
      if(length(.Object@x) != length(.Object@y))
      stop("specified x and y of different lengths")
      .Object
    })

### An alternative version that allows x and y to be supplied
### unnamed.  A still more friendly version would make the default x
### a vector of the same length as y, and vice versa.

setMethod("initialize", "track",
          function(.Object, x = numeric(0), y = numeric(0), ...) {
              .Object <- callNextMethod(.Object, ...)
              if(length(x) != length(y))
                  stop("specified x and y of different lengths")
              .Object@x <- x
              .Object@y <- y
              .Object
          })
```

---

nonStructure-class *A non-structure S4 Class for basic types*

Description

S4 classes that are defined to extend one of the basic vector classes should contain the class `structure` if they behave like structures; that is, if they should retain their class behavior under math functions or operators, so long as their length is unchanged. On the other hand, if their class depends on the values in the object, not just its structure, then they should lose that class under any such transformations. In the latter case, they should be defined to contain `nonStructure`.

If neither of these strategies applies, the class likely needs some methods of its own for `Ops`, `Math`, and/or other generic functions. What is not usually a good idea is to allow such computations to drop down to the default, base code. This is inconsistent with most definitions of such classes.

Methods

Methods are defined for operators and math functions (groups Ops, Math and Math2). In all cases the result is an ordinary vector of the appropriate type.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer.

See Also

structure

Examples

```
setClass("NumericNotStructure", contains = c("numeric","nonStructure"))
xx <- new("NumericNotStructure", 1:10)
xx + 1 # vector
log(xx) # vector
sample(xx) # vector
```

ObjectsWithPackage-class

A Vector of Object Names, with associated Package Names

Description

This class of objects is used to represent ordinary character string object names, extended with a package slot naming the package associated with each object.

Objects from the Class

The function getGenerics returns an object of this class.

Slots

.Data: Object of class "character": the object names.

package: Object of class "character" the package names.

Extends

Class "character", from data part.
Class "vector", by class "character".

See Also

Methods for general background.

promptClass *Generate a Shell for Documentation of a Formal Class*

Description

Assembles all relevant slot and method information for a class, with minimal markup for Rd processing; no QC facilities at present.

Usage

```
promptClass(clName, filename = NULL, type = "class",
            keywords = "classes", where = topenv(parent.frame()),
            generatorName = clName)
```

Arguments

clName	a character string naming the class to be documented.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is the topic name for the class documentation, followed by ".Rd". Can also be NA (see below).
type	the documentation type to be declared in the output file.
keywords	the keywords to include in the shell of the documentation. The keyword "classes" should be one of them.
where	where to look for the definition of the class and of methods that use it.
generatorName	the name for a generator function for this class; only required if a generator function was created *and* saved under a name different from the class name.

Details

The class definition is found on the search list. Using that definition, information about classes extended and slots is determined.

In addition, the currently available generics with methods for this class are found (using getGenerics). Note that these methods need not be in the same environment as the class definition; in particular, this part of the output may depend on which packages are currently in the search list.

As with other prompt-style functions, unless filename is NA, the documentation shell is written to a file, and a message about this is given. The file will need editing to give information about the *meaning* of the class. The output of promptClass can only contain information from the metadata about the formal definition and how it is used.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

If a generator function is found assigned under the class name or the optional generatorName, skeleton documentation for that function is added to the file.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Author(s)

VJ Carey <stvjc@channing.harvard.edu> and John Chambers

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`prompt` for documentation of functions, `promptMethods` for documentation of method definitions.

For processing of the edited documentation, either use `R CMD Rdconv`, or include the edited file in the 'man' subdirectory of a package.

Examples

```
## Not run: > promptClass("track")
A shell of class documentation has been written to the
file "track-class.Rd".

## End(Not run)
```

---

promptMethods *Generate a Shell for Documentation of Formal Methods*

Description

Generates a shell of documentation for the methods of a generic function.

Usage

```
promptMethods(f, filename = NULL, methods)
```

Arguments

`f`	a character string naming the generic function whose methods are to be documented.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to the coded topic name for these methods (currently, `f` followed by `"-methods.Rd"`). Can also be `FALSE` or `NA` (see below).

`methods` optional "listOfMethods" object giving the methods to be documented. By default, the first methods object for this generic is used (for example, if the current global environment has some methods for `f`, these would be documented).

If this argument is supplied, it is likely to be `findMethods(f, where)`, with `where` some package containing methods for `f`.

Details

If `filename` is `FALSE`, the text created is returned, presumably to be inserted some other documentation file, such as the documentation of the generic function itself (see `prompt`).

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

Otherwise, the documentation shell is written to the file specified by `filename`.

Value

If `filename` is `FALSE`, the text generated; if `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`prompt` and `promptClass`

ReferenceClasses *Objects With Fields Treated by Reference (OOP-style)*

Description

The software described here allows packages to define *reference classes* that behave in the style of "OOP" languages such as Java and C++. This model for OOP differs from the functional model implemented by S4 (and S3) classes and methods, in which methods are defined for generic functions. Methods for reference classes are "encapsulated" in the class definition.

Computations with objects from reference classes invoke methods on them and extract or set their fields, using the `` `$` `` operator in R. The field and method computations potentially modify the object. All computations referring to the objects see the modifications, in contrast to the usual functional programming model in R.

A call to `setRefClass` in the source code for a package defines the class and returns a generator object. Subsequent calls to the `$methods()` method of the generator will define methods for the

class. As with functional classes, if the class is exported from the package, it will be available when the package is loaded.

Methods are R functions. In their usual implementation, they refer to fields and other methods of the class directly by name. See the section on "Writing Reference Methods".

As with functional classes, reference classes can inherit from other reference classes via a `contains=` argument to `setRefClass`. Fields and methods will be inherited, except where the new class overrides method definitions. See the section on "Inheritance".

Usage

```
setRefClass(Class, fields = , contains = , methods =,
     where =, inheritPackage =, ...)

getRefClass(Class, where =)
```

Arguments

`Class`	character string name for the class. In the call to `getRefClass()` this argument can also be any object from the relevant class.
`fields`	either a character vector of field names or a named list of the fields. The resulting fields will be accessed with reference semantics (see the section on "Reference Objects"). If the argument is a list, each element of the list should usually be the character string name of a class, in which case the object in the field must be from that class or a subclass. An alternative, but not generally recommended, is to supply an *accessor function*; see the section on "Implementation" for accessor functions and the related internal mechanism. Note that fields are distinct from slots. Reference classes should not define class-specific slots. See the note on slots in the "Implementation" section.
`contains`	optional vector of superclasses for this class. If a superclass is also a reference class, the fields and class-based methods will be inherited.
`methods`	a named list of function definitions that can be invoked on objects from this class. These can also be created by invoking the `$methods` method on the generator object returned. See the section on "Writing Reference Methods" for details.
`where`	for `setRefClass`, the environment in which to store the class definition. Should be omitted in calls from a package's source code. For `getRefClass`, the environment from which to search for the definition. If the package is not loaded or you need to be specific, use `asNamespace` with the package name.
`inheritPackage`	Should objects from the new class inherit the package environment of a contained superclass? Default `FALSE`. See the Section "Inter-Package Superclasses and External Methods".
`...`	other arguments to be passed to `setClass`.

Value

`setRefClass()` returns a generator function suitable for creating objects from the class, invisibly. A call to this function takes any number of arguments, which will be passed on to the initialize method. If no `initialize` method is defined for the class or one of its superclasses, the default method expects named arguments with the name of one of the fields and unnamed arguments, if any, that are objects from one of the superclasses of this class (but only superclasses that are themselves reference classes have any effect).

The generator function is similar to the S4 generator function returned by `setClass`. In addition to being a generator function, however, it is also a reference class generator object, with reference class methods for various utilities. See the section on reference class generator objects below.

`getRefClass()` also returns the generator function for the class. Note that the package slot in the value is the correct package from the class definition, regardless of the `where` argument, which is used only to find the class if necessary.

Reference Objects

Normal objects in R are passed as arguments in function calls consistently with functional programming semantics; that is, changes made to an object passed as an argument are local to the function call. The object that supplied the argument is unchanged.

The functional model (sometimes called pass-by-value, although this is inaccurate for R) is suitable for many statistical computations and is implicit, for example, in the basic R software for fitting statistical models. In some other situations, one would like all the code dealing with an object to see the exact same content, so that changes made in any computation would be reflected everywhere. This is often suitable if the object has some "objective" reality, such as a window in a user interface.

In addition, commonly used languages, including Java, C++ and many others, support a version of classes and methods assuming reference semantics. The corresponding programming mechanism is to invoke a method on an object. In the R syntax we use `"$"` for this operation; one invokes a method, `m1` say, on an object `x` by the expression `x$m1(...)`.

Methods in this paradigm are associated with the object, or more precisely with the class of the object, as opposed to methods in a function-based class/method system, which are fundamentally associated with the function (in R, for example, a generic function in an R session has a table of all its currently known methods). In this document "methods for a class" as opposed to "methods for a function" will make the distinction.

Objects in this paradigm usually have named fields on which the methods operate. In the R implementation, the fields are defined when the class is created. The field itself can optionally have a specified class, meaning that only objects from this class or one of its subclasses can be assigned to the field. By default, fields have class `"ANY"`.

Fields are accessed by reference. In particular, invoking a method may modify the content of the fields.

Programming for such classes involves writing new methods for a particular class. In the R implementation, these methods are R functions, with zero or more formal arguments. For standard reference methods, the object itself is not an explicit argument to the method. Instead, fields and methods for the class can be referred to by name in the method definition. The implementation uses R environments to make fields and other methods available by name within the method. Specifically, the parent environment of the method is the object itself. See the section on "Writing Reference Methods". This special use of environments is optional. If a method is defined with an initial

formal argument `.self`, that will be passed in as the whole object, and the method follows the standard rules for any function in a package. See the section on "External Methods"

The goal of the software described here is to provide a uniform programming style in R for software dealing with reference classes, whether implemented directly in R or through an interface to one of the OOP languages.

Writing Reference Methods

Reference methods are functions supplied as elements of a named list, either when invoking `$methods()` on a generator object `g` or as the argument `methods` in a call to `setRefClass`. The two mechanisms have the same effect, but the first makes the code more readable.

Methods are written as ordinary R functions but have some special features and restrictions in their usual form. In contrast to some other languages (e.g., Python), the object itself does not need to be an argument in the method definition. The body of the function can contain calls to any other reference method, including those inherited from other reference classes and may refer to methods and to fields in the object by name.

Alternatively, a method may be an *external* method. This is signalled by `.self` being the first formal argument to the method. The body of the method then works like any ordinary function. The methods are called like other methods (without the `.self` argument, which is supplied internally and always refers to the object itself). Inside the method, fields and other methods are accessed in the form `.self$x`. External methods exist so that reference classes can inherit the package environment of superclasses in other packages; see the section on "External Methods".

Fields may be modified in a method by using the non-local assignment operator, `<<-`, as in the `$edit` and `$undo` methods in the example below. Note that non-local assignment is required: a local assignment with the `<-` operator just creates a local object in the function call, as it would in any R function. When methods are installed, a heuristic check is made for local assignments to field names and a warning issued if any are detected.

Reference methods should be kept simple; if they need to do some specialized R computation, that computation should use a separate R function that is called from the reference method. Specifically, methods can not use special features of the enclosing environment mechanism, since the method's environment is used to access fields and other methods. In particular, methods should not use non-exported entries in the package's namespace, because the methods may be inherited by a reference class in another package.

Two method names are interpreted specially, `initialize` and `finalize`. If an `initialize` method is defined, it will be invoked when an object is generated from the class. See the discussion of method `$new(...)` in the section "Initialization Methods".

If a `finalize` method is defined, a function will be registered to invoke it before the environment in the object is discarded by the garbage collector; finalizers are registered with `atexit=TRUE`, and so are also run at the end of R sessions. See the matrix viewer example for both initialize and finalize methods.

Reference methods can not themselves be generic functions; if you want additional function-based method dispatch, write a separate generic function and call that from the method.

Two special object names are available. The entire object can be referred to in a method by the reserved name `.self`. The object `.refClassDef` contains the definition of the class of the object. These are accessed as fields but are read-only, with one exception. In principal, the `.self` field can

be modified in the `$initialize` method, because the object is still being created at this stage. This is not recommended, as it can invalidate the object with respect to its class.

The methods available include methods inherited from superclasses, as discussed in the section "Inheritance".

Only methods actually used will be included in the environment corresponding to an individual object. To declare that a method requires a particular other method, the first method should include a call to `$usingMethods()` with the name of the other method as an argument. Declaring the methods this way is essential if the other method is used indirectly (e.g., via `sapply()` or `do.call()`). If it is called directly, code analysis will find it. Declaring the method is harmless in any case, however, and may aid readability of the source code.

Documentation for the methods can be obtained by the `$help` method for the generator object. Methods for classes are not documented in the `Rd` format used for R functions. Instead, the `$help` method prints the calling sequence of the method, followed by self-documentation from the method definition, in the style of Python. If the first element of the body of the method is a literal character string (possibly multi-line), that string is interpreted as documentation. See the method definitions in the example.

Initialization Methods

If the class has a method defined for `$initialize()`, this method will be called once the reference object has been created. You should write such a method for a class that needs to do some special initialization. In particular, a reference method is recommended rather than a method for the S4 generic function `initialize()`, because some special initialization is required for reference objects *before* the initialization of fields. As with S4 classes, methods are written for `$initialize()` and not for `$new()`, both for the previous reason and also because `$new()` is invoked on the generator object and would be a method for that class.

The default method for `$initialize()` is equivalent to invoking the method `$initFields(...)`. Named arguments assign initial values to the corresponding fields. Unnamed arguments must be objects from this class or a reference superclass of this class. Fields will be initialized to the contents of the fields in such objects, but named arguments override the corresponding inherited fields. Note that fields are simply assigned. If the field is itself a reference object, that object is not copied. The new and previous object will share the reference. Also, a field assigned from an unnamed argument counts as an assignment for locked fields. To override an inherited value for a locked field, the new value must be one of the named arguments in the initializing call. A later assignment of the field will result in an error.

Initialization methods need some care in design. The generator for a reference class will be called with no arguments, for example when copying the object. To ensure that these calls do not fail, the method must have defaults for all arguments or check for `missing()`. The method should include `...` as an argument and pass this on via `$callSuper()` (or `$initFields()` if you know that your superclasses have no initialization methods). This allows future class definitions that subclass this class, with additional fields.

Inheritance

Reference classes inherit from other reference classes by using the standard R inheritance; that is, by including the superclasses in the `contains=` argument when creating the new class. The names of the reference superclasses are in slot `refSuperClasses` of the class definition. Reference classes

can inherit from ordinary S4 classes also, but this is usually a bad idea if it mixes reference fields and non-reference slots. See the comments in the section on "Implementation".

Class fields are inherited. A class definition can override a field of the same name in a superclass only if the overriding class is a subclass of the class of the inherited field. This ensures that a valid object in the field remains valid for the superclass as well.

Inherited methods are installed in the same way as directly specified methods. The code in a method can refer to inherited methods in the same way as directly specified methods.

A method may override a method of the same name in a superclass. The overriding method can call the superclass method by `callSuper(...)` as described below.

Methods Provided for all Objects

All reference classes inherit from the class `"envRefClass"`. All reference objects can use the following methods.

`$callSuper(...)` Calls the method inherited from a reference superclass. The call is meaningful only from within another method, and will be resolved to call the inherited method of the same name. The arguments to `$callSuper` are passed to the superclass version. See the matrix viewer class in the example.

Note that the intended arguments for the superclass method must be supplied explicitly; there is no convention for supplying the arguments automatically, in contrast to the similar mechanism for functional methods.

`$copy(shallow = FALSE)` Creates a copy of the object. With reference classes, unlike ordinary R objects, merely assigning the object with a different name does not create an independent copy. If `shallow` is `FALSE`, any field that is itself a reference object will also be copied, and similarly recursively for its fields. Otherwise, while reassigning a field to a new reference object will have no side effect, modifying such a field will still be reflected in both copies of the object. The argument has no effect on non-reference objects in fields. When there are reference objects in some fields but it is asserted that they will not be modified, using `shallow = TRUE` will save some memory and time.

`$field(name, value)` With one argument, returns the field of the object with character string `name`. With two arguments, the corresponding field is assigned `value`. Assignment checks that `name` specifies a valid field, but the single-argument version will attempt to get anything of that name from the object's environment.

The `$field()` method replaces the direct use of a field name, when the name of the field must be calculated, or for looping over several fields.

`$export(Class)` Returns the result of coercing the object to `Class` (typically one of the superclasses of the object's class). Calling the method has no side effect on the object itself.

`$getRefClass()`; `$getClass()` These return respectively the generator object and the formal class definition for the reference class of this object, efficiently.

`$import(value, Class = class(value))` Import the object `value` into the current object, replacing the corresponding fields in the current object. Object `value` must come from one of the superclasses of the current object's class. If argument `Class` is supplied, `value` is first coerced to that class.

`$initFields(...)` Initialize the fields of the object from the supplied arguments. This method is usually only called from a class with a `$initialize()` method. It corresponds to the default

initialization for reference classes. If there are slots and non-reference superclasses, these may be supplied in the ... argument as well.

Typically, a specialized `$initialize()` method carries out its own computations, then invokes `$initFields()` to perform standard initialization, as shown in the `matrixViewer` class in the example below.

`$show()` This method is called when the object is printed automatically, analogously to the `show` function. A general method is defined for class `"envRefClass"`. User-defined reference classes will often define their own method: see the Example below.

Note two points in the example. As with any `show()` method, it is a good idea to print the class explicitly to allow for subclasses using the method. Second, to call the *function* `show()` from the method, as opposed to the `$show()` method itself, refer to `methods::show()` explicitly.

`$trace(what, ...)`, `$untrace(what)` Apply the tracing and debugging facilities of the `trace` function to the reference method `what`.

All the arguments of the `trace` function can be supplied, except for `signature`, which is not meaningful.

The reference method can be invoked on either an object or the generator for the class. See the section on Debugging below for details.

`$usingMethods(...)` Reference methods used by this method are named as the arguments either quoted or unquoted. In the code analysis phase of installing the present method, the declared methods will be included. It is essential to declare any methods used in a nonstandard way (e.g., via an apply function). Methods called directly do not need to be declared, but it is harmless to do so. `$usingMethods()` does nothing at run time.

Objects also inherit two reserved fields:

`.self` a reference to the entire object;

`.refClassDef` the class definition.

The defined fields should not override these, and in general it is unwise to define a field whose name begins with `"."`, since the implementation may use such names for special purposes.

External Methods; Inter-Package Superclasses

The environment of a method in a reference class is the object itself, as an environment. This allows the method to refer directly to fields and other methods, without using the whole object and the `"$"` operator. The parent of that environment is the namespace of the package in which the reference class is defined. Computations in the method have access to all the objects in the package's namespace, exported or not.

When defining a class that contains a reference superclass in another package, there is an ambiguity about which package namespace should have that role. The argument `inheritPackage` to `setRefClass()` controls whether the environment of new objects should inherit from an inherited class in another package or continue to inherit from the current package's namespace.

If the superclass is "lean", with few methods, or exists primarily to support a family of subclasses, then it may be better to continue to use the new package's environment. On the other hand, if the superclass was originally written as a standalone, this choice may invalidate existing superclass methods. For the superclass methods to continue to work, they must use only exported functions in their package and the new package must import these.

Either way, some methods may need to be written that do *not* assume the standard model for reference class methods, but behave essentially as ordinary functions would in dealing with reference class objects.

The mechanism is to recognize *external methods*. An external method is written as a function in which the first argument, named `.self`, stands for the reference class object. This function is supplied as the definition for a reference class method. The method will be called, automatically, with the first argument being the current object and the other arguments, if any, passed along from the actual call.

Since an external method is an ordinary function in the source code for its package, it has access to all the objects in the namespace. Fields and methods in the reference class must be referred to in the form `.self$name`.

If for some reason you do not want to use `.self` as the first argument, a function `f()` can be converted explicitly as `externalRefMethod(f)`, which returns an object of class `"externalRefMethod"` that can be supplied as a method for the class. The first argument will still correspond to the whole object.

External methods can be supplied for any reference class, but there is no obvious advantage unless they are needed. They are more work to write, harder to read and (slightly) slower to execute.

NOTE: If you are the author of a package whose reference classes are likely to be subclassed in other packages, you can avoid these questions entirely by writing methods that *only* use exported functions from your package, so that all the methods will work from another package that imports yours.

Reference Class Generators

The call to `setRefClass` defines the specified class and returns a "generator function" object for that class. This object has class `"refObjectGenerator"`; it inherits from `"function"` via `"classGeneratorFunction"` and can be called to generate new objects from the reference class.

The returned object is also a reference class object, although not of the standard construction. It can be used to invoke reference methods and access fields in the usual way, but instead of being implemented directly as an environment it has a subsidiary generator object as a slot, a standard reference object (of class `"refGeneratorSlot"`). Note that if one wanted to extend the reference class generator capability with a subclass, this should be done by subclassing `"refGeneratorSlot"`, not `"refObjectGenerator"`.

The fields are `def`, the class definition, and `className`, the character string name of the class. Methods generate objects from the class, to access help on reference methods, and to define new reference methods for the class. The currently available methods are:

`$new(...)` This method is equivalent to calling the generator function returned by `setRefClass`.

`$help(topic)` Prints brief help on the topic. The topics recognized are reference method names, quoted or not.

The information printed is the calling sequence for the method, plus self-documentation if any. Reference methods can have an initial character string or vector as the first element in the body of the function defining the method. If so, this string is taken as self-documentation for the method (see the section on "Writing Reference Methods" for details).

If no topic is given or if the topic is not a method name, the definition of the class is printed.

`$methods(...)` With no arguments, returns the names of the reference methods for this class. With one character string argument, returns the method of that name.

Named arguments are method definitions, which will be installed in the class, as if they had been supplied in the `methods` argument to `setRefClass()`. Supplying methods in this way, rather than in the call to `setRefClass()`, is recommended for the sake of clearer source code. See the section on "Writing Reference Methods" for details.

All methods for a class should be defined in the source code that defines the class, typically as part of a package. In particular, methods can not be redefined in a class in an attached package with a namespace: The class method checks for a locked binding of the class definition.

The new methods can refer to any currently defined method by name (including other methods supplied in this call to `$methods()`). Note though that previously defined methods are not re-analyzed meaning that they will not call the new method (unless it redefines an existing method of the same name).

To remove a method, supply `NULL` as its new definition.

`$fields()` Returns a list of the fields, each with its corresponding class. Fields for which an accessor function was supplied in the definition have class `"activeBindingFunction"`.

`$lock(...)` The fields named in the arguments are locked; specifically, after the lock method is called, the field may be set once. Any further attempt to set it will generate an error.

If called with no arguments, the method returns the names of the locked fields.

Fields that are defined by an explicit accessor function can not be locked (on the other hand, the accessor function can be defined to generate an error if called with an argument).

All code to lock fields should normally be part of the definition of a class; that is, the read-only nature of the fields is meant to be part of the class definition, not a dynamic property added later. In particular, fields can not be locked in a class in an attached package with a namespace: The class method checks for a locked binding of the class definition. Locked fields can not be subsequently unlocked.

`$trace(what, ..., classMethod = FALSE)` Establish a traced version of method `what` for objects generated from this class. The generator object tracing works like the `$trace()` method for objects from the class, with two differences. Since it changes the method definition in the class object itself, tracing applies to all objects, not just the one on which the trace method is invoked.

Second, the optional argument `classMethod = TRUE` allows tracing on the methods of the generator object itself. By default, `what` is interpreted as the name of a method in the class for which this object is the generator.

`$accessors(...)` A number of systems using the OOP programming paradigm recommend or enforce *getter and setter methods* corresponding to each field, rather than direct access by name. If you like this style and want to extract a field named `abc` by `x$getAbc()` and assign it by `x$setAbc(value)`, the `$accessors` method is a convenience function that creates such getter and setter methods for the specified fields. Otherwise there is no reason to use this mechanism. In particular, it has nothing to do with the general ability to define fields by functions as described in the section on "Reference Objects".

Implementation; Reference Classes as S4 Classes

Reference classes are implemented as S4 classes with a data part of type `"environment"`. Fields correspond to named objects in the environment. A field associated with a function is implemented

as an active binding. In particular, fields with a specified class are implemented as a special form of active binding to enforce valid assignment to the field.

As a related feature, the element in the `fields=` list supplied to `setRefClass` can be an *accessor function*, a function of one argument that returns the field if called with no argument or sets it to the value of the argument otherwise. Accessor functions are used internally and for inter-system interface applications, but not generally recommended as they blur the concept of fields as data within the object.

A field, say `data`, can be accessed generally by an expression of the form `x$data` for any object from the relevant class. In an internal method for this class, the field can be accessed by the name `data`. A field that is not locked can be set by an expression of the form `x$data <- value`. Inside an internal method, a field can be assigned by an expression of the form `x <<- value`. Note the non-local assignment operator. The standard R interpretation of this operator works to assign it in the environment of the object. If the field has an accessor function defined, getting and setting will call that function.

When a method is invoked on an object, the function defining the method is installed in the object's environment, with the same environment as the environment of the function.

Reference classes can have validity methods in the same sense as any S4 class (see `setValidity`). Such methods are often a good idea; they will be called by calling `validObject` and a validity method, if one is defined, will be called when a reference object is created (from version 3.4 of R on). Just remember that these are S4 methods. The function will be called with the `object` as its argument. Fields and methods must be accessed using `$`.

Note: Slots. Because of the implementation, new reference classes can inherit from non-reference S4 classes as well as reference classes, and can include class-specific slots in the definition. This is usually a bad idea, if the slots from the non-reference class are thought of as alternatives to fields. Slots will as always be treated functionally. Therefore, changes to the slots and the fields will behave inconsistently, mixing the functional and reference paradigms for properties of the same object, conceptually unclear and prone to errors. In addition, the initialization method for the class will have to sort out fields from slots, with a good chance of creating anomalous behavior for subclasses of this class.

Inheriting from a class union, however, is a reasonable strategy (with all members of the union likely to be reference classes).

Debugging

The standard R debugging and tracing facilities can be applied to reference methods. Reference methods can be passed to debug and its relatives from an object to debug further method invocations on that object; for example, `debug(xx$edit)`.

Somewhat more flexible use is available for a reference method version of the `trace` function. A corresponding `$trace()` reference method is available for either an object or for the reference class generator (`xx$trace()` or `mEdit$trace()` in the example below). Using `$trace()` on an object sets up a tracing version for future invocations of the specified method for that object. Using `$trace()` on the generator for the class sets up a tracing version for all future objects from that class (and sometimes for existing objects from the class if the method is not declared or previously invoked).

In either case, all the arguments to the standard `trace` function are available, except for `signature=` which is meaningless since reference methods can not be S4 generic functions. This includes the

typical style `trace(what, browser)` for interactive debugging and `trace(what, edit = TRUE)` to edit the reference method interactively.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 11.)

Examples

```
## a simple editor for matrix objects.  Method  $edit() changes some
## range of values; method $undo() undoes the last edit.
mEdit <- setRefClass("mEdit",
      fields = list( data = "matrix",
        edits = "list"))

## The basic edit, undo methods
mEdit$methods(
     edit = function(i, j, value) {
       ## the following string documents the edit method
       'Replaces the range [i, j] of the
        object by value.
        '
         backup <-
             list(i, j, data[i,j])
         data[i,j] <<- value
         edits <<- c(edits, list(backup))
         invisible(value)
     },
     undo = function() {
       'Undoes the last edit() operation
        and update the edits field accordingly.
        '
         prev <- edits
         if(length(prev)) prev <- prev[[length(prev)]]
         else stop("No more edits to undo")
         edit(prev[[1]], prev[[2]], prev[[3]])
         ## trim the edits list
         length(edits) <<- length(edits) - 2
         invisible(prev)
     })

## A method to automatically print objects
mEdit$methods(
     show = function() {
       'Method for automatically printing matrix editors'
       cat("Reference matrix editor object of class",
          classLabel(class(.self)), "\n")
       cat("Data: \n")
       methods::show(data)
       cat("Undo list is of length", length(edits), "\n")
     }
     )
```

```
xMat <- matrix(1:12,4,3)
xx <- mEdit(data = xMat)
xx$edit(2, 2, 0)
xx
xx$undo()
mEdit$help("undo")
stopifnot(all.equal(xx$data, xMat))

utils::str(xx) # show fields and names of methods

## A method to save the object
mEdit$methods(
     save = function(file) {
       'Save the current object on the file
        in R external object format.
       '
         base::save(.self, file = file)
     }
)

tf <- tempfile()
xx$save(tf)


## Not run:
## Inheriting a reference class:  a matrix viewer
mv <- setRefClass("matrixViewer",
    fields = c("viewerDevice", "viewerFile"),
    contains = "mEdit",
    methods = list( view = function() {
        dd <- dev.cur(); dev.set(viewerDevice)
        devAskNewPage(FALSE)
        matplot(data, main = paste("After",length(edits),"edits"))
        dev.set(dd)},
        edit = # invoke previous method, then replot
          function(i, j, value) {
            callSuper(i, j, value)
            view()
          }))

## initialize and finalize methods
mv$methods( initialize =
  function(file = "./matrixView.pdf", ...) {
    viewerFile <<- file
    pdf(viewerFile)
    viewerDevice <<- dev.cur()
    dev.set(dev.prev())
    callSuper(...)
  },
  finalize = function() {
    dev.off(viewerDevice)
  })
```

```
## debugging an object: call browser() in method $edit()
xx$trace(edit, browser)

## debugging all objects from class mEdit in method $undo()
mEdit$trace(undo, browser)

## End(Not run)
```

`removeMethod` *Remove a Method*

Description

Remove the method for a given function and signature. Obsolete for ordinary applications: Method definitions in a package should never need to remove methods and it's very bad practice to remove methods that were defined in other packages.

Usage

```
removeMethod(f, signature, where)
```

Arguments

`f`, `signature`, `where`
: As for `setMethod()`.

Value

`TRUE` if a method was found to be removed.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`representation` *Construct a Representation or a Prototype for a Class Definition*

Description

These are old utility functions to construct, respectively a list designed to represent the slots and superclasses and a list of prototype specifications. The `representation()` function is no longer useful, since the arguments `slots` and `contains` to `setClass` are now recommended.

The `prototype()` function may still be used for the corresponding argument, but a simple list of the same arguments works as well.

Usage

```
representation(...)
prototype(...)
```

Arguments

...	The call to representation takes arguments that are single character strings. Unnamed arguments are classes that a newly defined class extends; named arguments name the explicit slots in the new class, and specify what class each slot should have. In the call to prototype, if an unnamed argument is supplied, it unconditionally forms the basis for the prototype object. Remaining arguments are taken to correspond to slots of this object. It is an error to supply more than one unnamed argument.

Details

The representation function applies tests for the validity of the arguments. Each must specify the name of a class.

The classes named don't have to exist when representation is called, but if they do, then the function will check for any duplicate slot names introduced by each of the inherited classes.

The arguments to prototype are usually named initial values for slots, plus an optional first argument that gives the object itself. The unnamed argument is typically useful if there is a data part to the definition (see the examples below).

Value

The value of representation is just the list of arguments, after these have been checked for validity.

The value of prototype is the object to be used as the prototype. Slots will have been set consistently with the arguments, but the construction does *not* use the class definition to test validity of the contents (it hardly can, since the prototype object is usually supplied to create the definition).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

setClass

Examples

```
## representation for a new class with a directly define slot "smooth"
## which should be a "numeric" object, and extending class "track"
representation("track", smooth ="numeric")
```

```
###  >>> This *is* old syntax -- use 'contains=*, slots=*' instead <<<
###                ==========         ----------  ------   ======


setClass("Character",representation("character"))
setClass("TypedCharacter",representation("Character",type="character"),
          prototype(character(0),type="plain"))
ttt <- new("TypedCharacter", "foo", type = "character")


setClass("num1", representation(comment = "character"),
         contains = "numeric",
         prototype = prototype(pi, comment = "Start with pi"))
```

S3Part *S4 Classes that Contain S3 Classes*

Description

A regular (S4) class may contain an S3 class, if that class has been registered (by calling `setOldClass`). The functions described here provide information about contained S3 classes. See the section 'Functions'.

In modern R, these functions are not usually needed to program with objects from the S4 class. Standard computations work as expected, including method selection for both S4 and S3. To coerce an object to its contained S3 class, use either of the expressions:

`as(object, S3Class)`; `as(object, "S3")`

where `S3Class` evaluates to the name of the contained class. These return slightly different objects, which in rare cases may need to be distinguished. See the section "Contained S3 Objects".

Usage

```
S3Part(object, strictS3 = FALSE, S3Class)

S3Class(object)

isXS3Class(classDef)

slotsFromS3(object)

## the replacement versions of the functions are not recommended
## Create a new object from the class or use the replacement version of as().
```

```
S3Part(object, strictS3 = FALSE, needClass = ) <- value

S3Class(object) <-  value
```

Arguments

object	an object from some class that extends a registered S3 class, or a basic vector, matrix or array object type. For most of the functions, an S3 object can also be supplied, with the interpretation that it is its own S3 part.
strictS3	If TRUE, the value returned by S3Part will be an S3 object, with all the S4 slots removed. Otherwise, an S4 object will always be returned; for example, from the S4 class created by setOldClass as a proxy for an S3 class, rather than the underlying S3 object.
S3Class	the character vector to be stored as the S3 class slot in the object. Usually, and by default, retains the slot from object, but an S3 superclass is allowed.
classDef	a class definition object, as returned by getClass. *The remaining arguments apply only to the replacement versions, which are not recommended.*
needClass	Require that the replacement value be this class or a subclass of it.
value	For S3Part<-, the replacement value for the S3 part of the object. For S3Class<-, the character vector that will be used as a proxy for class(x) in S3 method dispatch.

Functions

S3Part: Returns an object from the S3 class that appeared in the contains= argument to setClass.

If called with strictS3 = TRUE, S3Part() constructs the underlying S3 object by eliminating all the formally defined slots and turning off the S4 bit of the object. With strictS3 = FALSE the object returned is from the corresponding S4 class. For consistency and generality, S3Part() works also for classes that extend the basic vector, matrix and array classes.

A call to is equivalent coercing the object to class "S3" for the strict case, or to whatever the specific S3 class was, for the non-strict case. The as() calls are usually easier for readers to understand.

S3Class: Returns the character vector of S3 class(es) stored in the object, if the class has the corresponding .S3Class slot. Currently, the function defaults to class otherwise.

isXS3Class: Returns TRUE or FALSE according to whether the class defined by ClassDef extends S3 classes (specifically, whether it has the slot for holding the S3 class).

slotsFromS3: returns a list of the relevant slot classes, or an empty list for any other object.

The function slotsFromS3() is a generic function used internally to access the slots associated with the S3 part of the object. Methods for this function are created automatically when setOldClass is called with the S4Class argument. Usually, there is only one S3 slot, containing the S3 class, but the S4Class argument may provide additional slots, in the case that the S3 class has some guaranteed attributes that can be used as formal S4 slots. See the corresponding section in the documentation of setOldClass.

Contained S3 Objects

Registering an S3 class defines an S4 class. Objects from this class are essentially identical in content to an object from the S3 class, except for two differences. The value returned by `class()` will always be a single string for the S4 object, and `isS4()` will return `TRUE` or `FALSE` in the two cases. See the example below. It is barely possible that some S3 code will not work with the S4 object; if so, use `as(x, "S3")`.

Objects from a class that extends an S3 class will have some basic type and possibly some attributes. For an S3 class that has an equivalent S4 definition (e.g., `"data.frame"`), an extending S4 class will have a data part and slots. For other S3 classes (e.g., `"lm"`) an object from the extending S4 class will be some sort of basic type, nearly always a vector type (e.g., `"list"` for `"lm"`), but the data part will not have a formal definition.

Registering an S3 class by a call to `setOldClass` creates a class of the same name with a slot `".S3Class"` to hold the corresponding S3 vector of class strings. New S4 classes that extend such classes also have the same slot, set to the S3 class of the contained S3 *object*, which may be an (S3) subclass of the registered class. For example, an S4 class might contain the S3 class `"lm"`, but an object from the class might contain an object from class `"mlm"`, as in the `"xlm"`example below.

R is somewhat arbitrary about what it treats as an S3 class: `"ts"` is, but `"matrix"` and `"array"` are not. For classes that extend those, assuming they contain an S3 class is incorrect and will cause some confusion—not usually disastrous, but the better strategy is to stick to the explicit "class". Thus `as(x, "matrix")` rather than `as(x, "S3")` or `S3Part(x)`.

S3 and S4 Objects: Conversion Mechanisms

Objects in R have an internal bit that indicates whether or not to treat the object as coming from an S4 class. This bit is tested by `isS4` and can be set on or off by `asS4`. The latter function, however, does no checking or interpretation; you should only use it if you are very certain every detail has been handled correctly.

As a friendlier alternative, methods have been defined for coercing to the virtual classes `"S3"` and `"S4"`. The expressions `as(object, "S3")` and `as(object, "S4")` return S3 and S4 objects, respectively. In addition, they attempt to do conversions in a valid way, and also check validity when coercing to S4.

The expression `as(object, "S3")` can be used in two ways. For objects from one of the registered S3 classes, the expression will ensure that the class attribute is the full multi-string S3 class implied by `class(object)`. If the registered class has known attribute/slots, these will also be provided.

Another use of `as(object, "S3")` is to take an S4 object and turn it into an S3 object with corresponding attributes. This is only meaningful with S4 classes that have a data part. If you want to operate on the object without invoking S4 methods, this conversion is usually the safest way.

The expression `as(object, "S4")` will use the attributes in the object to create an object from the S4 definition of `class(object)`. This is a general mechanism to create partially defined version of S4 objects via S3 computations (not much different from invoking `new` with corresponding arguments, but usable in this form even if the S4 object has an initialize method with different arguments).

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10, particularly Section 10.8)

See Also

setOldClass

Examples

```
## an "mlm" object, regressing two variables on two others

sepal <- as.matrix(datasets::iris[,c("Sepal.Width", "Sepal.Length")])
fit <- lm(sepal ~ Petal.Length + Petal.Width + Species, data = datasets::iris)
class(fit) # S3 class: "mlm", "lm"

## a class that contains "mlm"
myReg <- setClass("myReg", slots = c(title = "character"), contains = "mlm")

fit2 <- myReg(fit, title = "Sepal Regression for iris data")

fit2 # shows the inherited "mlm" object and the title

identical(S3Part(fit2), as(fit2, "mlm"))

class(as(fit2, "mlm")) # the S4 class, "mlm"

class(as(fit2, "S3")) # the S3 class, c("mlm", "lm")

## An object may contain an S3 class from a subclass of that declared:
xlm <- setClass("xlm", slots = c(eps = "numeric"), contains = "lm")

xfit <- xlm(fit, eps = .Machine$double.eps)

xfit@.S3Class # c("mlm", lm")
```

S4groupGeneric *S4 Group Generic Functions*

Description

Methods can be defined for *group generic functions*. Each group generic function has a number of *member* generic functions associated with it.

Methods defined for a group generic function cause the same method to be defined for each member of the group, but a method explicitly defined for a member of the group takes precedence over a method defined, with the same signature, for the group generic.

The functions shown in this documentation page all reside in the **methods** package, but the mechanism is available to any programmer, by calling setGroupGeneric (provided package **methods** is attached).

Usage

```
## S4 group generics:
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)
Logic(e1, e2)
Math(x)
Math2(x, digits)
Summary(x, ..., na.rm = FALSE)
Complex(z)
```

Arguments

x, z, e1, e2	objects.
digits	number of digits to be used in round or signif.
...	further arguments passed to or from methods.
na.rm	logical: should missing values be removed?

Details

Methods can be defined for the group generic functions by calls to setMethod in the usual way. Note that the group generic functions should never be called directly – a suitable error message will result if they are. When metadata for a group generic is loaded, the methods defined become methods for the members of the group, but only if no method has been specified directly for the member function for the same signature. The effect is that group generic definitions are selected before inherited methods but after directly specified methods. For more on method selection, see Methods_Details.

There are also S3 groups Math, Ops, Summary and Complex, see ?S3groupGeneric, with no corresponding R objects, but these are irrelevant for S4 group generic functions.

The members of the group defined by a particular generic can be obtained by calling getGroupMembers. For the group generic functions currently defined in this package the members are as follows:

Arith "+", "-", "*", "^", "%%", "%/%", "/"

Compare "==", ">", "<", "!=", "<=", ">="

Logic "&", "|".

Ops "Arith", "Compare", "Logic"

Math "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "cummax", "cummin", "cumprod", "cumsum", "log", "log10", "log2", "log1p", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "cospi", "sin", "sinh", "sinpi", "tan", "tanh", "tanpi", "gamma", "lgamma", "digamma", "trigamma"

Math2 "round", "signif"

Summary "max", "min", "range", "prod", "sum", "any", "all"

Complex "Arg", "Conj", "Im", "Mod", "Re"

Note that Ops merely consists of three sub groups.

All the functions in these groups (other than the group generics themselves) are basic functions in R. They are not by default S4 generic functions, and many of them are defined as primitives. However, you can still define formal methods for them, both individually and via the group generics. It all works more or less as you might expect, admittedly via a bit of trickery in the background. See Methods_Details for details.

Note that two members of the Math group, log and trunc, have ... as an extra formal argument. Since methods for Math will have only one formal argument, you must set a specific method for these functions in order to call them with the extra argument(s).

For further details about group generic functions see section 10.5 of the second reference.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.5)

See Also

The function callGeneric is nearly always relevant when writing a method for a group generic. See the examples below and in section 10.5 of *Software for Data Analysis*.

See S3groupGeneric for S3 group generics.

Examples

```
setClass("testComplex", slots = c(zz = "complex"))
## method for whole group "Complex"
setMethod("Complex", "testComplex",
          function(z) c("groupMethod", callGeneric(z@zz)))
## exception for Arg() :
setMethod("Arg", "testComplex",
          function(z) c("ArgMethod", Arg(z@zz)))
z1 <- 1+2i
z2 <- new("testComplex", zz = z1)
stopifnot(identical(Mod(z2), c("groupMethod", Mod(z1))))
stopifnot(identical(Arg(z2), c("ArgMethod", Arg(z1))))
```

SClassExtension-class *Class to Represent Inheritance (Extension) Relations*

Description

An object from this class represents a single 'is' relationship; lists of these objects are used to represent all the extensions (superclasses) and subclasses for a given class. The object contains information about how the relation is defined and methods to coerce, test, and replace correspondingly.

Objects from the Class

Objects from this class are generated by setIs, from direct calls and from the contains= information in a call to setClass, and from class unions created by setClassUnion. In the last case, the information is stored in defining the *subclasses* of the union class (allowing unions to contain sealed classes).

Slots

subClass, superClass: The classes being extended: corresponding to the from, and to arguments to setIs.

package: The package to which that class belongs.

coerce: A function to carry out the as() computation implied by the relation. Note that these functions should *not* be used directly. They only deal with the strict=TRUE calls to the as function, with the full method constructed from this mechanically.

test: The function that would test whether the relation holds. Except for explicitly specified test arguments to setIs, this function is trivial.

replace: The method used to implement as(x, Class) <- value.

simple: A "logical" flag, TRUE if this is a simple relation, either because one class is contained in the definition of another, or because a class has been explicitly stated to extend a virtual class. For simple extensions, the three methods are generated automatically.

by: If this relation has been constructed transitively, the first intermediate class from the subclass.

dataPart: A "logical" flag, TRUE if the extended class is in fact the data part of the subclass. In this case the extended class is a basic class (i.e., a type).

distance: The distance between the two classes, 1 for directly contained classes, plus the number of generations between otherwise.

Methods

No methods defined with class "SClassExtension" in the signature.

See Also

is, as, and the classRepresentation class.

---

selectSuperClasses *Super Classes (of Specific Kinds) of a Class*

Description

Return superclasses of ClassDef, possibly only non-virtual or direct or simple ones.

These functions are designed to be fast, and consequently only work with the contains slot of the corresponding class definitions.

Usage

```
selectSuperClasses(Class, dropVirtual = FALSE, namesOnly = TRUE,
                   directOnly = TRUE, simpleOnly = directOnly,
                   where = topenv(parent.frame()))

.selectSuperClasses(ext, dropVirtual = FALSE, namesOnly = TRUE,
                    directOnly = TRUE, simpleOnly = directOnly)
```

Arguments

`Class`	name of the class or (more efficiently) the class definition object (see `getClass`).
`dropVirtual`	logical indicating if only non-virtual superclasses should be returned.
`namesOnly`	logical indicating if only a vector names instead of a named list class-extensions should be returned.
`directOnly`	logical indicating if only a *direct* super classes should be returned.
`simpleOnly`	logical indicating if only simple class extensions should be returned.
`where`	(only used when `Class` is not a class definition) environment where the class definition of `Class` is found.
`ext`	for `.selectSuperClasses()` only, a `list` of class extensions, typically `getClassDef(..)@contains`.

Value

a `character` vector (if `namesOnly` is true, as per default) or a list of class extensions (as the `contains` slot in the result of `getClass`).

Note

The typical user level function is `selectSuperClasses()` which calls `.selectSuperClasses()`; i.e., the latter should only be used for efficiency reasons by experienced useRs.

See Also

`is`, `getClass`; further, the more technical class `classRepresentation` documentation.

Examples

```
setClass("Root")
setClass("Base", contains = "Root", slots = c(length = "integer"))
setClass("A", contains = "Base", slots = c(x = "numeric"))
setClass("B", contains = "Base", slots = c(y = "character"))
setClass("C", contains = c("A", "B"))

extends("C")   #-->  "C"  "A" "B"  "Base" "Root"
selectSuperClasses("C") # "A" "B"
selectSuperClasses("C", directOnly=FALSE) # "A" "B"  "Base"  "Root"
selectSuperClasses("C", dropVirtual=TRUE, directOnly=FALSE)# ditto w/o "Root"
```

setAs *Methods for Coercing an Object to a Class*

Description

A call to setAs defines a method for coercing an object of class from to class to. The methods will then be used by calls to as for objects with class from, including calls that replace part of the object.

Methods for this purpose work indirectly, by defining methods for function coerce. The coerce function is *not* to be called directly, and method selection uses class inheritance only on the first argument.

Usage

```
setAs(from, to, def, replace, where = topenv(parent.frame()))
```

Arguments

from, to	The classes between which the coerce methods def and replace perform coercion.
def	function of one argument. It will get an object from class from and had better return an object of class to. The convention is that the name of the argument is from; if another argument name is used, setAs will attempt to substitute from.
replace	if supplied, the function to use as a replacement method, when as is used on the left of an assignment. Should be a function of two arguments, from, value, although setAs will attempt to substitute if the arguments differ. *The remaining argument will not be used in standard applications.*
where	the position or environment in which to store the resulting methods. Do not use this argument when defining a method in a package. Only the default, the namespace of the package, should be used in normal situations.

Inheritance and Coercion

Objects from one class can turn into objects from another class either automatically or by an explicit call to the as function. Automatic conversion is special, and comes from the designer of one class of objects asserting that this class extends another class. The most common case is that one or more class names are supplied in the contains= argument to setClass, in which case the new class extends each of the earlier classes (in the usual terminology, the earlier classes are *superclasses* of the new class and it is a *subclass* of each of them).

This form of inheritance is called *simple* inheritance in R. See setClass for details. Inheritance can also be defined explicitly by a call to setIs. The two versions have slightly different implications for coerce methods. Simple inheritance implies that inherited slots behave identically in the subclass and the superclass. Whenever two classes are related by simple inheritance, corresponding coerce methods are defined for both direct and replacement use of as. In the case of simple inheritance, these methods do the obvious computation: they extract or replace the slots in the object that correspond to those in the superclass definition.

The implicitly defined coerce methods may be overridden by a call to setAs; note, however, that the implicit methods are defined for each subclass-superclass pair, so that you must override each of these explicitly, not rely on inheritance.

When inheritance is defined by a call to setIs, the coerce methods are provided explicitly, not generated automatically. Inheritance will apply (to the from argument, as described in the section below). You could also supply methods via setAs for non-inherited relationships, and now these also can be inherited.

For further on the distinction between simple and explicit inheritance, see setIs.

How Functions as and setAs Work

The function as turns object into an object of class Class. In doing so, it applies a "coerce method", using S4 classes and methods, but in a somewhat special way. Coerce methods are methods for the function coerce or, in the replacement case the function `coerce<-`. These functions have two arguments in method signatures, from and to, corresponding to the class of the object and the desired coerce class. These functions must not be called directly, but are used to store tables of methods for the use of as, directly and for replacements. In this section we will describe the direct case, but except where noted the replacement case works the same way, using `coerce<-` and the replace argument to setAs, rather than coerce and the def argument.

Assuming the object is not already of the desired class, as first looks for a method in the table of methods for the function coerce for the signature c(from = class(object), to = Class), in the same way method selection would do its initial lookup. To be precise, this means the table of both direct and inherited methods, but inheritance is used specially in this case (see below).

If no method is found, as looks for one. First, if either Class or class(object) is a superclass of the other, the class definition will contain the information needed to construct a coerce method. In the usual case that the subclass contains the superclass (i.e., has all its slots), the method is constructed either by extracting or replacing the inherited slots. Non-simple extensions (the result of a call to setIs) will usually contain explicit methods, though possibly not for replacement.

If no subclass/superclass relationship provides a method, as looks for an inherited method, but applying, inheritance for the argument from only, not for the argument to (if you think about it, you'll probably agree that you wouldn't want the result to be from some class other than the Class specified). Thus, selectMethod("coerce", sig, useInherited= c(from=TRUE, to= FALSE)) replicates the method selection used by as().

In nearly all cases the method found in this way will be cached in the table of coerce methods (the exception being subclass relationships with a test, which are legal but discouraged). So the detailed calculations should be done only on the first occurrence of a coerce from class(object) to Class.

Note that coerce is not a standard generic function. It is not intended to be called directly. To prevent accidentally caching an invalid inherited method, calls are routed to an equivalent call to as, and a warning is issued. Also, calls to selectMethod for this function may not represent the method that as will choose. You can only trust the result if the corresponding call to as has occurred previously in this session.

With this explanation as background, the function setAs does a fairly obvious computation: It constructs and sets a method for the function coerce with signature c(from, to), using the def argument to define the body of the method. The function supplied as def can have one argument (interpreted as an object to be coerced) or two arguments (the from object and the to class). Either way, setAs constructs a function of two arguments, with the second defaulting to the name of the to

class. The method will be called from as with the object as the from argument and no to argument, with the default for this argument being the name of the intended to class, so the method can use this information in messages.

The direct version of the as function also has a strict= argument that defaults to TRUE. Calls during the evaluation of methods for other functions will set this argument to FALSE. The distinction is relevant when the object being coerced is from a simple subclass of the to class; if strict=FALSE in this case, nothing need be done. For most user-written coerce methods, when the two classes have no subclass/superclass, the strict= argument is irrelevant.

The replace argument to setAs provides a method for `coerce<-`. As with all replacement methods, the last argument of the method must have the name value for the object on the right of the assignment. As with the coerce method, the first two arguments are from, to; there is no strict= option for the replace case.

The function coerce exists as a repository for such methods, to be selected as described above by the as function. Actually dispatching the methods using standardGeneric could produce incorrect inherited methods, by using inheritance on the to argument; as mentioned, this is not the logic used for as. To prevent selecting and caching invalid methods, calls to coerce are currently mapped into calls to as, with a warning message.

Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, as(x, "numeric") uses the existing as.numeric function. These built-in methods can be listed by showMethods("coerce").

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

If you think of using try(as(x, cl)), consider canCoerce(x, cl) instead.

Examples

```
## using the definition of class "track" from \link{setClass}



setAs("track", "numeric", function(from) from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")

## The next example shows:
##  1. A virtual class to define setAs for several classes at once.
##  2. as() using inherited information

setClass("ca", slots = c(a = "character", id = "numeric"))
```

```
setClass("cb", slots = c(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")


setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)
CB <- new("cb", b = "B", id = 2)

setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

`setClass` *Create a Class Definition*

Description

Create a class definition and return a generator function to create objects from the class. Typical usage will be of the style:

`myClass <- setClass("myClass", slots= ...., contains =....)`

where the first argument is the name of the new class and, if supplied, the arguments `slots=` and `contains=` specify the slots in the new class and existing classes from which the new class should inherit. Calls to `setClass()` are normally found in the source of a package; when the package is loaded the class will be defined in the package's namespace. Assigning the generator function with the name of the class is convenient for users, but not a requirement.

Usage

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where, version, sealed, package,
         S3methods = FALSE, slots)
```

Arguments

`Class`	character string name for the class.
`slots`	The names and classes for the slots in the new class. This argument must be supplied by name, `slots=`, in the call, for back compatibility with other arguments no longer recommended. The argument must be vector with a names attribute, the names being those of the slots in the new class. Each element of the vector specifies an existing class; the corresponding slot must be from this class or a subclass of it. Usually, this

is a character vector naming the classes. It's also legal for the elements of the vector to be class representation objects, as returned by `getClass`.

As a limiting case, the argument may be an unnamed character vector; the elements are taken as slot names and all slots have the unrestricted class `"ANY"`.

`contains` A vector specifying existing classes from which this class should inherit. The new class will have all the slots of the superclasses, with the same requirements on the classes of these slots. This argument must be supplied by name, `contains=`, in the call, for back compatibility with other arguments no longer recommended.

See the section 'Virtual Classes' for the special superclass `"VIRTUAL"`.

`prototype`, `where`, `validity`, `sealed`, `package`

These arguments are currently allowed, but either they are unlikely to be useful or there are modern alternatives that are preferred.

`prototype`: supplies an object with the default data for the slots in this class. A more flexible approach is to write a method for `initialize()`.

`where`: supplies an environment in which to store the definition. Should not be used: For calls to `setClass()` appearing in the source code for a package the definition will be stored in the namespace of the package.

`validity`: supplied a validity-checking method for objects from this class. For clearer code, use a separate call to `setValidity()`.

`sealed`: if `TRUE`, the class definition will be sealed, so that another call to `setClass` will fail on this class name. But the definition is automatically sealed after the namespace is loaded, so explicit sealing it is not needed.

`package`: supplies an optional package name for the class, but the class attribute should be the package in which the class definition is assigned, as it is by default.

`representation`, `access`, `version`, `S3methods`

All these arguments are deprecated from version 3.0.0 of R *and should be avoided.*

`representation` is an argument inherited from S that included both `slots` and `contains`, but the use of the latter two arguments is clearer and recommended.

`access` and `version` are included for historical compatibility with S-Plus, but ignored.

`S3methods` is a flag indicating that old-style methods will be written involving this class; ignored now.

Value

A generator function suitable for creating objects from the class is returned, invisibly. A call to this function generates a call to `new` for the class. The call takes any number of arguments, which will be passed on to the initialize method. If no `initialize` method is defined for the class or one of its superclasses, the default method expects named arguments with the name of one of the slots and unnamed arguments that are objects from one of the contained classes.

Typically the generator function is assigned the name of the class, for programming clarity. This is not a requirement and objects from the class can also be generated directly from `new`. The advantages of the generator function are a slightly simpler and clearer call, and that the call will contain the package name of the class (eliminating any ambiguity if two classes from different packages have the same name).

If the class is virtual, an attempt to generate an object from either the generator or `new()` will result in an error.

Basic Use: Slots and Inheritance

The two essential arguments other than the class name are `slots` and `contains`, defining the explicit slots and the inheritance (superclasses). Together, these arguments define all the information in an object from this class; that is, the names of all the slots and the classes required for each of them.

The name of the class determines which methods apply directly to objects from this class. The superclass information specifies which methods apply indirectly, through inheritance. See Methods_Details for inheritance in method selection.

The slots in a class definition will be the union of all the slots specified directly by `slots` and all the slots in all the contained classes. There can only be one slot with a given name. A class may override the definition of a slot with a given name, but *only* if the newly specified class is a subclass of the inherited one. For example, if the contained class had a slot `a` with class `"ANY"`, then a subclass could specify `a` with class `"numeric"`, but if the original specification for the slot was class `"character"`, the new call to `setClass` would generate an error.

Slot names `"class"` and `"Class"` are not allowed. There are other slot names with a special meaning; these names start with the `"."` character. To be safe, you should define all of your own slots with names starting with an alphabetic character.

Some inherited classes will be treated specially—object types, S3 classes and a few special cases—whether inherited directly or indirectly. See the next three sections.

Virtual Classes

Classes exist for which no actual objects can be created, the *virtual* classes.

The most common and useful form of virtual class is the *class union*, a virtual class that is defined in a call to `setClassUnion()` rather than a call to `setClass()`. This call lists the *members* of the union—subclasses that extend the new class. Methods that are written with the class union in the signature are eligible for use with objects from any of the member classes. Class unions can include as members classes whose definition is otherwise sealed, including basic R data types.

Calls to `setClass()` will also create a virtual class, either when only the `Class` argument is supplied (no slots or superclasses) or when the `contains=` argument includes the special class name `"VIRTUAL"`.

In the latter case, a virtual class may include slots to provide some common behavior without fully defining the object—see the class `traceable` for an example. Note that `"VIRTUAL"` does not carry over to subclasses; a class that contains a virtual class is not itself automatically virtual.

Inheriting from Object Types

In addition to containing other S4 classes, a class definition can contain either an S3 class (see the next section) or a built-in R pseudo-class—one of the R object types or one of the special R pseudo-classes `"matrix"` and `"array"`. A class can contain at most one of the object types, directly or indirectly. When it does, that contained class determines the "data part" of the class. This appears as a pseudo-slot, `".Data"` and can be treated as a slot but actually determines the type of objects from this slot.

Objects from the new class try to inherit the built in behavior of the contained type. In the case of normal R data types, including vectors, functions and expressions, the implementation is relatively straightforward. For any object `x` from the class, `typeof(x)` will be the contained basic type; and a special pseudo-slot, `.Data`, will be shown with the corresponding class. See the `"numWithId"` example below.

Classes may also inherit from `"vector"`, `"matrix"` or `"array"`. The data part of these objects can be any vector data type.

For an object from any class that does *not* contain one of these types or classes, `typeof(x)` will be `"S4"`.

Some R data types do not behave normally, in the sense that they are non-local references or other objects that are not duplicated. Examples include those corresponding to classes `"environment"`, `"externalptr"`, and `"name"`. These can not be the types for objects with user-defined classes (either S4 or S3) because setting an attribute overwrites the object in all contexts. It is possible to define a class that inherits from such types, through an indirect mechanism that stores the inherited object in a reserved slot, `".xData"`. See the example for class `"stampedEnv"` below. An object from such a class does *not* have a `".Data"` pseudo-slot.

For most computations, these classes behave transparently as if they inherited directly from the anomalous type. S3 method dispatch and the relevant `as.`*type*`()` functions should behave correctly, but code that uses the type of the object directly will not. For example, `as.environment(e1)` would work as expected with the `"stampedEnv"` class, but `typeof(e1)` is `"S4"`.

Inheriting from S3 Classes

Old-style S3 classes have no formal definition. Objects are "from" the class when their class attribute contains the character string considered to be the class name.

Using such classes with formal classes and methods is necessarily a risky business, since there are no guarantees about the content of the objects or about consistency of inherited methods. Given that, it is still possible to define a class that inherits from an S3 class, providing that class has been registered as an old class (see `setOldClass`).

Broadly speaking, both S3 and S4 method dispatch try to behave sensibly with respect to inheritance in either system. Given an S4 object, S3 method dispatch and the `inherits` function should use the S4 inheritance information. Given an S3 object, an S4 generic function will dispatch S4 methods using the S3 inheritance, provided that inheritance has been declared via `setOldClass`. For details, see `setOldClass` and Section 10.8 of the reference.

Classes and Packages

Class definitions normally belong to packages (but can be defined in the global environment as well, by evaluating the expression on the command line or in a file sourced from the command line). The corresponding package name is part of the class definition; that is, part of the `classRepresentation` object holding that definition. Thus, two classes with the same name can exist in different packages, for most purposes.

When a class name is supplied for a slot or a superclass in a call to `setClass`, a corresponding class definition will be found, looking from the namespace of the current package, assuming the call in question appears directly in the source for the package, as it should to avoid ambiguity. The class definition must be already defined in this package, in the imports directives of the package's `DESCRIPTION` and `NAMESPACE` files or in the basic classes defined by the methods package. (The

'methods' package must be included in the imports directives for any package that uses S4 methods and classes, to satisfy the "CMD check" utility.)

If a package imports two classes of the same name from separate packages, the packageSlot of the name argument needs to be set to the package name of the particular class. This should be a rare occurrence.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Classes_Details for a general discussion of classes, Methods_Details for an analogous discussion of methods, makeClassRepresentation

Examples

```
## A simple class with two slots
track <- setClass("track", slots = c(x="numeric", y="numeric"))
## an object from the class
t1 <- track(x = 1:10, y = 1:10 + rnorm(10))

## A class extending the previous, adding one more slot
trackCurve <- setClass("trackCurve",
  slots = c(smooth = "numeric"),
  contains = "track")

## an object containing a superclass object
t1s <- trackCurve(t1, smooth = 1:10)

## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         slots = c(x="numeric", y="matrix", smooth="matrix"),
         prototype = list(x=numeric(), y=matrix(0,0,0),
                          smooth= matrix(0,0,0)))

## A class that extends the built-in data type "numeric"

numWithId <- setClass("numWithId", slots = c(id = "character"),
         contains = "numeric")

numWithId(1:3, id = "An Example")

## inherit from reference object of type "environment"
stampedEnv <- setClass("stampedEnv", contains = "environment",
                       slots = c(update = "POSIXct"))
setMethod("[[<-", c("stampedEnv", "character", "missing"),
   function(x, i, j, ..., value) {
       ev <- as(x, "environment")
       ev[[i]] <- value  #update the object in the environment
       x@update <- Sys.time() # and the update time
```

```
        x})

e1 <- stampedEnv(update = Sys.time())

e1[["noise"]] <- rnorm(10)
```

`setClassUnion` *Classes Defined as the Union of Other Classes*

Description

A class may be defined as the *union* of other classes; that is, as a virtual class defined as a superclass of several other classes. Class unions are useful in method signatures or as slots in other classes, when we want to allow one of several classes to be supplied.

Usage

```
setClassUnion(name, members, where)
isClassUnion(Class)
```

Arguments

`name`	the name for the new union class.
`members`	the names of the classes that should be members of this union.
`where`	where to save the new class definition. In calls from a package's source code, should be omitted to save the definition in the package's namespace.
`Class`	the name or definition of a class.

Details

The classes in `members` must be defined before creating the union. However, members can be added later on to an existing union, as shown in the example below. Class unions can be members of other class unions.

Class unions are the only way to create a new superclass of a class whose definition is sealed. The namespace of all packages is sealed when the package is loaded, protecting the class and other definitions from being overwritten from another class or from the global environment. A call to `setIs` that tried to define a new superclass for class `"numeric"`, for example, would cause an error.

Class unions are the exception; the class union `"maybeNumber"` in the examples defines itself as a new superclass of `"numeric"`. Technically, it does not alter the metadata object in the other package's namespace and, of course, the effect of the class union depends on loading the package it belongs to. But, basically, class unions are sufficiently useful to justify the exemption.

The different behavior for class unions is made possible because the class definition object for class unions has itself a special class, `"ClassUnionRepresentation"`, an extension of class `classRepresentation`.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Examples

```
## a class for either numeric or logical data
setClassUnion("maybeNumber", c("numeric", "logical"))

## use the union as the data part of another class
setClass("withId", contains = "maybeNumber", slots = c(id = "character"))

w1 <- new("withId", 1:10, id = "test 1")
w2 <- new("withId", sqrt(w1)%%1 < .01, id = "Perfect squares")

## add class "complex" to the union "maybeNumber"
setIs("complex", "maybeNumber")

w3 <- new("withId", complex(real = 1:10, imaginary = sqrt(1:10)))

## a class union containing the existing class  union "OptionalFunction"
setClassUnion("maybeCode",
    c("expression", "language", "OptionalFunction"))

is(quote(sqrt(1:10)), "maybeCode")  ## TRUE
```

`setGeneric` *Create a Generic Version of a Function*

Description

Create a generic version of the named function so that methods may be defined for it. A call to `setMethod` will call `setGeneric` automatically if applied to a non-generic function.

An explicit call to `setGeneric` is usually not required, but doesn't hurt and makes explicit that methods are being defined for a non-generic function.

Standard calls will be of the form:

`setGeneric(name)`

where `name` specifies an existing function, possibly in another package. An alternative when creating a new generic function in this package is:

`setGeneric(name, def)`

where the function definition `def` specifies the formal arguments and becomes the default method.

Usage

```
setGeneric(name, def= , group=list(), valueClass=character(),
           where= , package= , signature= , useAsDefault= ,
           genericFunction= , simpleInheritanceOnly = )
```

Arguments

`name`	The character string name of the generic function.
`def`	An optional function object, defining the non-generic version, to become the default method. This is equivalent in effect to assigning `def` as the function and then using the one-argument call to `setGeneric`. *The following arguments are specialized, optionally used when creating a new generic function with non-standard features. They should not be used when the non-generic is in another package.*
`group`	The name of the group generic function to which this function belongs. See Methods_Details for details of group generic functions in method selection and S4groupGeneric for existing groups.
`valueClass`	A character vector specifying one or more class names. The value returned by the generic function must have (or extend) this class, or one of the classes; otherwise, an error is generated.
`signature`	The vector of names from among the formal arguments to the function, that will be allowed in the signature of methods for this function, in calls to `setMethod`. By default and usually, this will be all formal arguments except .... A non-standard signature for the generic function may be used to exclude arguments that take advantage of lazy evaluation; in particular, if the argument may *not* be evaluated then it cannot be part of the signature. While ... cannot be used as part of a general signature, it is possible to have this as the *only* element of the signature. Methods will then be selected if their signature matches all the ... arguments. See the documentation for topic dotsMethods for details. It is not possible to mix ... and other arguments in the signature. It's usually a mistake to omit arguments from the signature in the belief that this improves efficiency. For method selection, the arguments that are used in the signatures for the *methods* are what counts, and then only seriously on the first call to the function with that combination of classes.
`simpleInheritanceOnly`	Supply this argument as `TRUE` to require that methods selected be inherited through simple inheritance only; that is, from superclasses specified in the `contains=` argument to `setClass`, or by simple inheritance to a class union or other virtual class. Generic functions should require simple inheritance if they need to be assured that they get the complete original object, not one that has been transformed. Examples of functions requiring simple inheritance are `initialize`, because by definition it must return an object from the same class as its argument, and `show`, because it claims to give a full description of the object provided as its argument.
`useAsDefault`	Override the usual default method mechanism. Only relevant when defining a nonstandard generic function. See the section 'Specialized Local Generics'. *The remaining arguments are obsolete for normal applications.*
`package`	The name of the package with which this function is associated. Should be determined automatically from the non-generic version.
`where`	Where to store the resulting objects as side effects. The default, to store in the package's namespace, is the only safe choice.

genericFunction
: Obsolete.

Value

The setGeneric function exists for its side effect: saving the generic function to allow methods to be specified later. It returns name.

Basic Use

The setGeneric function is called to initialize a generic function as preparation for defining some methods for that function.

The simplest and most common situation is that name specifies an existing function, usually in another package. You now want to define methods for this function. In this case you should supply only name, for example:

```
setGeneric("colSums")
```

There must be an existing function of this name (in this case in package "base"). The non-generic function can be in the same package as the call, typically the case when you are creating a new function plus methods for it. When the function is in another package, it must be available by name, for example through an importFrom() directive in this package's NAMESPACE file. Not required for functions in "base", which are implicitly imported.

A generic version of the function will be created in the current package. The existing function becomes the default method, and the package slot of the new generic function is set to the location of the original function ("base" in the example).

Two special types of non-generic should be noted. Functions that dispatch S3 methods by calling UseMethod are ordinary functions, not objects from the "genericFunction" class. They are made generic like any other function, but some special considerations apply to ensure that S4 and S3 method dispatch is consistent (see Methods_for_S3).

Primitive functions are handled in C code and don't exist as normal functions. A call to setGeneric is allowed in the simple form, but no actual generic function object is created. Method dispatch will take place in the C code. See the section on Primitive Functions for more details.

It's an important feature that the identical generic function definition is created in every package that uses the same setGeneric() call. When any of these packages is loaded into an R session, this function will be added to a table of generic functions, and will contain a methods table of all the available methods for the function.

Calling setGeneric() is not strictly necessary before calling setMethod(). If the function specified in the call to setMethod is not generic, setMethod will execute the call to setGeneric itself. In the case that the non-generic is in another package, does not dispatch S3 methods and is not a primitive, a message is printed noting the creation of the generic function the first time setMethod is called.

The second common use of setGeneric() is to create a new generic function, unrelated to any existing function. See the asRObject() example below. This case can be handled just like the previous examples, with only the difference that the non-generic function exists in the current package. Again, the non-generic version becomes the default method. For clarity it's best for the assignment to immediately precede the call to setGeneric() in the source code.

Exactly the same result can be obtained by supplying the default as the def argument instead of assigning it. In some applications, there will be no completely general default method. While there is a special mechanism for this (see the 'Specialized Local Generics' section), the recommendation is to provide a default method that signals an error, but with a message that explains as clearly as you can why a non-default method is needed.

Specialized Local Generics

The great majority of calls to setGeneric() should either have one argument to ensure that an existing function can have methods, or arguments name and def to create a new generic function and optionally a default method.

It is possible to create generic functions with nonstandard signatures, or functions that do additional computations besides method dispatch or that belong to a group of generic functions.

None of these mechanisms should be used with a non-generic function from a *different* package, because the result is to create a generic function that may not be consistent from one package to another. When any such options are used, the new generic function will be assigned with a package slot set to the *current* package, not the one in which the non-generic version of the function is found.

There is a mechanism to define a specialized generic version of a non-generic function, the implicitGeneric construction. This defines the generic version, but then reverts the function to it non-generic form, saving the implicit generic in a table to be activated when methods are defined. However, the mechanism can only legitimately be used either for a non-generic in the same package or by the "methods" package itself. And in the first case, there is no compelling reason not to simply make the function generic, with the non-generic as the default method. See implicitGeneric for details.

The body of a generic function usually does nothing except for dispatching methods by a call to standardGeneric. Under some circumstances you might just want to do some additional computation in the generic function itself. As long as your function eventually calls standardGeneric that is permissible. See the example "authorNames" below.

In this case, the def argument will define the nonstandard generic, not the default method. An existing non-generic of the same name and calling sequence should be pre-assigned. It will become the default method, as usual. (An alternative is the useAsDefault argument.)

By default, the generic function can return any object. If valueClass is supplied, it should be a vector of class names; the value returned by a method is then required to satisfy is(object, Class) for one of the specified classes. An empty (i.e., zero length) vector of classes means anything is allowed. Note that more complicated requirements on the result can be specified explicitly, by defining a non-standard generic function.

If the def argument calls standardGeneric() (with or without additional computations) and there is no existing non-generic version of the function, the generic is created without a default method. This is not usually a good idea: better to have a default method that signals an error with a message explaining why the default case is not defined.

A new generic function can be created belonging to an existing group by including the group argument. The argument list of the new generic must agree with that of the group. See setGroupGeneric for defining a new group generic. For the role of group generics in dispatching methods, see GroupGenericFunctions and section 10.5 of the second reference.

Generic Functions and Primitive Functions

A number of the basic R functions are specially implemented as primitive functions, to be evaluated directly in the underlying C code rather than by evaluating an R language definition. Most have implicit generics (see `implicitGeneric`), and become generic as soon as methods (including group methods) are defined on them. Others cannot be made generic.

Calling `setGeneric()` for the primitive functions in the base package differs in that it does not, in fact, generate an explicit generic function. Methods for primitives are selected and dispatched from the internal C code, to satisfy concerns for efficiency. The same is true for a few non-primitive functions that dispatch internally. These include `unlist` and `as.vector`.

Note, that the implementation restrict methods for primitive functions to signatures in which at least one of the classes in the signature is a formal S4 class. Otherwise the internal C code will not look for methods. This is a desirable restriction in principle, since optional packages should not be allowed to change the behavior of basic R computations on existing data types.

To see the generic version of a primitive function, use `getGeneric(name)`. The function `isGeneric` will tell you whether methods are defined for the function in the current session.

Note that S4 methods can only be set on those primitives which are 'internal generic', plus `%*%`.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.5 for some details.)

See Also

`Methods_Details` and the links there for a general discussion, `dotsMethods` for methods that dispatch on `...`, and `setMethod` for method definitions.

Examples

```
## Specify that this package will define methods for plot()
setGeneric("plot")

## create a new generic function, with a default method
setGeneric("props", function(object) attributes(object))

###   A non-standard generic function.  It insists that the methods
###   return a non-empty character vector (a stronger requirement than
###    valueClass = "character" in the call to setGeneric)

setGeneric("authorNames",
    function(text) {
      value <- standardGeneric("authorNames")
      if(!(is(value, "character") && any(nchar(value)>0)))
        stop("authorNames methods must return non-empty strings")
      value
      })

## the asRObject generic function, from package XR
```

```
## Its default method just returns object
## See the reference, Chapter 12 for methods

setGeneric("asRObject", function(object, evaluator) {
        object
})
```

---

setGroupGeneric *Create a Group Generic Version of a Function*

Description

The setGroupGeneric function behaves like setGeneric except that it constructs a group generic function, differing in two ways from an ordinary generic function. First, this function cannot be called directly, and the body of the function created will contain a stop call with this information. Second, the group generic function contains information about the known members of the group, used to keep the members up to date when the group definition changes, through changes in the search list or direct specification of methods, etc.

All members of the group must have the identical argument list.

Usage

```
setGroupGeneric(name, def= , group=list(), valueClass=character(),
                knownMembers=list(), package= , where= )
```

Arguments

name	the character string name of the generic function.
def	A function object. There isn't likely to be an existing nongeneric of this name, so some function needs to be supplied. Any known member or other function with the same argument list will do, because the group generic cannot be called directly.
group, valueClass	arguments to pass to setGeneric.
knownMembers	the names of functions that are known to be members of this group. This information is used to reset cached definitions of the member generics when information about the group generic is changed.
package, where	passed to setGeneric, but obsolete and to be avoided.

Value

The setGroupGeneric function exists for its side effect: saving the generic function to allow methods to be specified later. It returns name.

References

Chambers, John M. (2016) *Extending R* Chapman & Hall

See Also

Methods_Details and the links there for a general discussion, dotsMethods for methods that dispatch on ..., and setMethod for method definitions.

Examples

```
## Not run:
## the definition of the "Logic" group generic in the methods package
setGroupGeneric("Logic", function(e1, e2) NULL,
    knownMembers = c("&", "|"))

## End(Not run)
```

---

setIs *Specify a Superclass Explicitly*

Description

setIs is an explicit alternative to the contains= argument to setClass. It is only needed to create relations with explicit test or coercion. These have not proved to be of much practical value, so this function should not likely be needed in applications.

Where the programming goal is to define methods for transforming one class of objects to another, it is usually better practice to call setAs(), which requires the transformations to be done explicitly.

Usage

```
setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = topenv(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

Arguments

class1, class2	the names of the classes between which is relations are to be examined defined, or (more efficiently) the class definition objects for the classes.
coerce, replace	functions optionally supplied to coerce the object to class2, and to alter the object so that is(object, class2) is identical to value. See the details section below.
test	a *conditional* relationship is defined by supplying this function. Conditional relations are discouraged and are not included in selecting methods. See the details section below. The remaining arguments are for internal use and/or usually omitted.

`extensionObject`
: alternative to the `test, coerce, replace, by` arguments; an object from class `SClassExtension` describing the relation. (Used in internal calls.)

`doComplete`
: when `TRUE`, the class definitions will be augmented with indirect relations as well. (Used in internal calls.)

`by`
: In a call to `setIs`, the name of an intermediary class. Coercion will proceed by first coercing to this class and from there to the target class. (The intermediate coercions have to be valid.)

`where`
: In a call to `setIs`, where to store the metadata defining the relationship. Default is the global environment for calls from the top level of the session or a source file evaluated there. When the call occurs in the top level of a file in the source of a package, the default will be the namespace or environment of the package. Other uses are tricky and not usually a good idea, unless you really know what you are doing.

`classDef`
: Optional class definition for `class` , required internally when `setIs` is called during the initial definition of the class by a call to `setClass`. *Don't* use this argument, unless you really know why you're doing so.

Details

Arranging for a class to inherit from another class is a key tool in programming. In R, there are three basic techniques, the first two providing what is called "simple" inheritance, the preferred form:

1. By the `contains=` argument in a call to `setClass`. This is and should be the most common mechanism. It arranges that the new class contains all the structure of the existing class, and in particular all the slots with the same class specified. The resulting class extension is defined to be `simple`, with important implications for method definition (see the section on this topic below).
2. Making `class1` a subclass of a virtual class either by a call to `setClassUnion` to make the subclass a member of a new class union, or by a call to `setIs` to add a class to an existing class union or as a new subclass of an existing virtual class. In either case, the implication should be that methods defined for the class union or other superclass all work correctly for the subclass. This may depend on some similarity in the structure of the subclasses or simply indicate that the superclass methods are defined in terms of generic functions that apply to all the subclasses. These relationships are also generally simple.
3. Supplying `coerce` and `replace` arguments to `setAs`. R allows arbitrary inheritance relationships, using the same mechanism for defining coerce methods by a call to `setAs`. The difference between the two is simply that `setAs` will require a call to `as` for a conversion to take place, whereas after the call to `setIs`, objects will be automatically converted to the superclass.

   The automatic feature is the dangerous part, mainly because it results in the subclass potentially inheriting methods that do not work. See the section on inheritance below. If the two classes involved do not actually inherit a large collection of methods, as in the first example below, the danger may be relatively slight.

   If the superclass inherits methods where the subclass has only a default or remotely inherited method, problems are more likely. In this case, a general recommendation is to use the `setAs`

mechanism instead, unless there is a strong counter reason. Otherwise, be prepared to override some of the methods inherited.

With this caution given, the rest of this section describes what happens when `coerce=` and `replace=` arguments are supplied to `setIs`.

The `coerce` and `replace` arguments are functions that define how to coerce a `class1` object to `class2`, and how to replace the part of the subclass object that corresponds to `class2`. The first of these is a function of one argument which should be `from`, and the second of two arguments (`from`, `value`). For details, see the section on coerce functions below .

When by is specified, the coerce process first coerces to this class and then to `class2`. It's unlikely you would use the by argument directly, but it is used in defining cached information about classes.

The value returned (invisibly) by `setIs` is the revised class definition of `class1`.

Coerce, replace, and test functions

The `coerce` argument is a function that turns a `class1` object into a `class2` object. The `replace` argument is a function of two arguments that modifies a `class1` object (the first argument) to replace the part of it that corresponds to `class2` (supplied as `value`, the second argument). It then returns the modified object as the value of the call. In other words, it acts as a replacement method to implement the expression `as(object, class2) <- value`.

The easiest way to think of the `coerce` and `replace` functions is by thinking of the case that `class1` contains `class2` in the usual sense, by including the slots of the second class. (To repeat, in this situation you would not call `setIs`, but the analogy shows what happens when you do.)

The `coerce` function in this case would just make a `class2` object by extracting the corresponding slots from the `class1` object. The `replace` function would replace in the `class1` object the slots corresponding to `class2`, and return the modified object as its value.

For additional discussion of these functions, see the documentation of the `setAs` function. (Unfortunately, argument `def` to that function corresponds to argument `coerce` here.)

The inheritance relationship can also be conditional, if a function is supplied as the `test` argument. This should be a function of one argument that returns `TRUE` or `FALSE` according to whether the object supplied satisfies the relation `is(object, class2)`. Conditional relations between classes are discouraged in general because they require a per-object calculation to determine their validity. They cannot be applied as efficiently as ordinary relations and tend to make the code that uses them harder to interpret. *NOTE: conditional inheritance is not used to dispatch methods.* Methods for conditional superclasses will not be inherited. Instead, a method for the subclass should be defined that tests the conditional relationship.

Inherited methods

A method written for a particular signature (classes matched to one or more formal arguments to the function) naturally assumes that the objects corresponding to the arguments can be treated as coming from the corresponding classes. The objects will have all the slots and available methods for the classes.

The code that selects and dispatches the methods ensures that this assumption is correct. If the inheritance was "simple", that is, defined by one or more uses of the `contains=` argument in a call to `setClass`, no extra work is generally needed. Classes are inherited from the superclass, with the same definition.

When inheritance is defined by a general call to `setIs`, extra computations are required. This form of inheritance implies that the subclass does *not* just contain the slots of the superclass, but instead requires the explicit call to the coerce and/or replace method. To ensure correct computation, the inherited method is supplemented by calls to `as` before the body of the method is evaluated.

The calls to `as` generated in this case have the argument `strict = FALSE`, meaning that extra information can be left in the converted object, so long as it has all the appropriate slots. (It's this option that allows simple subclass objects to be used without any change.) When you are writing your coerce method, you may want to take advantage of that option.

Methods inherited through non-simple extensions can result in ambiguities or unexpected selections. If `class2` is a specialized class with just a few applicable methods, creating the inheritance relation may have little effect on the behavior of `class1`. But if `class2` is a class with many methods, you may find that you now inherit some undesirable methods for `class1`, in some cases, fail to inherit expected methods. In the second example below, the non-simple inheritance from class `"factor"` might be assumed to inherit S3 methods via that class. But the S3 class is ambiguous, and in fact is `"character"` rather than `"factor"`.

For some generic functions, methods inherited by non-simple extensions are either known to be invalid or sufficiently likely to be so that the generic function has been defined to exclude such inheritance. For example `initialize` methods must return an object of the target class; this is straightforward if the extension is simple, because no change is made to the argument object, but is essentially impossible. For this reason, the generic function insists on only simple extensions for inheritance. See the `simpleInheritanceOnly` argument to `setGeneric` for the mechanism. You can use this mechanism when defining new generic functions.

If you get into problems with functions that do allow non-simple inheritance, there are two basic choices. Either back off from the `setIs` call and settle for explicit coercing defined by a call to `setAs`; or, define explicit methods involving `class1` to override the bad inherited methods. The first choice is the safer, when there are serious problems.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Examples

```
## Two examples of setIs() with coerce= and replace= arguments
## The first one works fairly well, because neither class has many
## inherited methods do be disturbed by the new inheritance

## The second example does NOT work well, because the new superclass,
## "factor", causes methods to be inherited that should not be.

## First example:
## a class definition (see \link{setClass} for class "track")
setClass("trackCurve", contains = "track",
         slots = c( smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         slots = c(x="numeric", y="matrix", smooth="matrix"),
         prototype = structure(list(), x=numeric(), y=matrix(0,0,0),
```

```
                              smooth= matrix(0,0,0)))
## Automatically convert an object from class "trackCurve" into
## "trackMultiCurve", by making the y, smooth slots into 1-column matrices
setIs("trackCurve",
      "trackMultiCurve",
      coerce = function(obj) {
        new("trackMultiCurve",
            x = obj@x,
            y = as.matrix(obj@y),
            smooth = as.matrix(obj@smooth))
      },
      replace = function(obj, value) {
        obj@y <- as.matrix(value@y)
        obj@x <- value@x
        obj@smooth <- as.matrix(value@smooth)
        obj})



## Second Example:
## A class that adds a slot to "character"
setClass("stringsDated", contains = "character",
         slots = c(stamp="POSIXt"))

## Convert automatically to a factor by explicit coerce
setIs("stringsDated", "factor",
      coerce = function(from) factor(from@.Data),
      replace= function(from, value) {
                  from@.Data <- as.character(value); from })

ll <- sample(letters, 10, replace = TRUE)
ld <- new("stringsDated", ll, stamp = Sys.time())

levels(as(ld, "factor"))
levels(ld) # will be NULL--see comment in section on inheritance above.

## In contrast, a class that simply extends "factor"
## has no such ambiguities
setClass("factorDated", contains = "factor",
         slots = c(stamp="POSIXt"))
fd <- new("factorDated", factor(ll), stamp = Sys.time())
identical(levels(fd), levels(as(fd, "factor")))
```

---

setLoadActions *Set Actions For Package Loading*

Description

These functions provide a mechanism for packages to specify computations to be done during the loading of a package namespace. Such actions are a flexible way to provide information only available at load time (such as locations in a dynamically linked library).

A call to `setLoadAction()` or `setLoadActions()` specifies one or more functions to be called when the corresponding namespace is loaded, with the ... argument names being used as identifying names for the actions.

`getLoadActions` reports the currently defined load actions, given a package's namespace as its argument.

`hasLoadAction` returns `TRUE` if a load action corresponding to the given name has previously been set for the `where` namespace.

`evalOnLoad()` and `evalqOnLoad()` schedule a specific expression for evaluation at load time.

Usage

```
setLoadAction(action, aname=, where=)

setLoadActions(..., .where=)

getLoadActions(where=)

hasLoadAction(aname, where=)

evalOnLoad(expr, where=, aname=)

evalqOnLoad(expr, where=, aname=)
```

Arguments

`action`, `...`	functions of one or more arguments, to be called when this package is loaded. The functions will be called with one argument (the package namespace) so all following arguments must have default values. If the elements of ... are named, these names will be used for the corresponding load metadata.
`where`, `.where`	the namespace of the package for which the list of load actions are defined. This argument is normally omitted if the call comes from the source code for the package itself, but will be needed if a package supplies load actions for another package.
`aname`	the name for the action. If an action is set without supplying a name, the default uses the position in the sequence of actions specified (`".1"`, etc.).
`expr`	an expression to be evaluated in a load action in environment `where`. In the case of `evalqOnLoad()`, the expression is interpreted literally, in that of `evalOnLoad()` it must be precomputed, typically as an object of type `"language"`.

Details

The `evalOnLoad()` and `evalqOnLoad()` functions are for convenience. They construct a function to evaluate the expression and call `setLoadAction()` to schedule a call to that function.

Each of the functions supplied as an argument to `setLoadAction()` or `setLoadActions()` is saved as metadata in the namespace, typically that of the package containing the call to `setLoadActions()`. When this package's namespace is loaded, each of these functions will be called. Action functions are called in the order they are supplied to `setLoadActions()`. The objects assigned have metadata names constructed from the names supplied in the call; unnamed arguments are taken to be named by their position in the list of actions (`".1"`, etc.).

Multiple calls to `setLoadAction()` or `setLoadActions()` can be used in a package's code; the actions will be scheduled after any previously specified, except if the name given to `setLoadAction()` is that of an existing action. In typical applications, `setLoadActions()` is more convenient when calling from the package's own code to set several actions. Calls to `setLoadAction()` are more convenient if the action name is to be constructed, which is more typical when one package constructs load actions for another package.

Actions can be revised by assigning with the same name, actual or constructed, in a subsequent call. The replacement must still be a valid function, but can of course do nothing if the intention was to remove a previously specified action.

The functions must have at least one argument. They will be called with one argument, the namespace of the package. The functions will be called at the end of processing of S4 metadata, after dynamically linking any compiled code, the call to `.onLoad()`, if any, and caching method and class definitions, but before the namespace is sealed. (Load actions are only called if methods dispatch is on.)

Functions may therefore assign or modify objects in the namespace supplied as the argument in the call. The mechanism allows packages to save information not available until load time, such as values obtained from a dynamically linked library.

Load actions should be contrasted with user load hooks supplied by `setHook()`. User hooks are generally provided from outside the package and are run after the namespace has been sealed. Load actions are normally part of the package code, and the list of actions is normally established when the package is installed.

Load actions can be supplied directly in the source code for a package. It is also possible and useful to provide facilities in one package to create load actions in another package. The software needs to be careful to assign the action functions in the correct environment, namely the namespace of the target package.

Value

`setLoadAction()` and `setLoadActions()` are called for their side effect and return no useful value.

`getLoadActions()` returns a named list of the actions in the supplied namespace.

`hasLoadAction()` returns `TRUE` if the specified action name appears in the actions for this package.

See Also

`setHook` for safer (since they are run after the namespace is sealed) and more comprehensive versions in the base package.

Examples

```
## Not run:
## in the code for some package

## ... somewhere else
setLoadActions(function(ns)
   cat("Loaded package", sQuote(getNamespaceName(ns)),
       "at", format(Sys.time()), "\n"),
  setCount = function(ns) assign("myCount", 1, envir = ns),
  function(ns) assign("myPointer", getMyExternalPointer(), envir = ns))
  ... somewhere later
if(countShouldBe0)
  setLoadAction(function(ns) assign("myCount", 0, envir = ns), "setCount")

## End(Not run)
```

`setMethod` *Create and Save a Method*

Description

Create a method for a generic function, corresponding to a signature of classes for the arguments. Standard usage will be of the form:

`setMethod(f, signature, definition)`

where `f` is the name of the function, `signature` specifies the argument classes for which the method applies and `definition` is the function definition for the method.

Usage

```
setMethod(f, signature=character(), definition,
          where = topenv(parent.frame()),
          valueClass = NULL, sealed = FALSE)
```

Arguments

`f`	The character-string name of the generic function. The unquoted name usually works as well (evaluating to the generic function), except for a few functions in the base package.
`signature`	The classes required for some of the arguments. Most applications just require one or two character strings matching the first argument(s) in the signature. More complicated cases follow R's rule for argument matching. See the details below; however, if the signature is not trivial, you should use `method.skeleton` to generate a valid call to `setMethod`.
`definition`	A function definition, which will become the method called when the arguments in a call to `f` match the classes in `signature`, directly or through inheritance. The definition must be a function with the same formal arguments as the generic; however, `setMethod()` will handle methods that add arguments, if `...` is a formal argument to the generic. See the Details section.

`where`, `valueClass`, `sealed`

These arguments are allowed but either obsolete or rarely appropriate.

`where`: where to store the definition; should be the default, the namespace for the package.

`valueClass`: obsolete.

`sealed`: prevents the method being redefined, but should never be needed when the method is defined in the source code of a package.

Value

The function exists for its side-effect. The definition will be stored in a special metadata object and incorporated in the generic function when the corresponding package is loaded into an R session.

Method Selection: Avoiding Ambiguity

When defining methods, it's important to ensure that methods are selected correctly; in particular, packages should be designed to avoid ambiguous method selection.

To describe method selection, consider first the case where only one formal argument is in the active signature; that is, there is only one argument, `x` say, for which methods have been defined. The generic function has a table of methods, indexed by the class for the argument in the calls to `setMethod`. If there is a method in the table for the class of `x` in the call, this method is selected.

If not, the next best methods would correspond to the direct superclasses of `class(x)`—those appearing in the `contains=` argument when that class was defined. If there is no method for any of these, the next best would correspond to the direct superclasses of the first set of superclasses, and so on.

The first possible source of ambiguity arises if the class has several direct superclasses and methods have been defined for more than one of those; R will consider these equally valid and report an ambiguous choice. If your package has the class definition for `class(x)`, then you need to define a method explicitly for this combination of generic function and class.

When more than one formal argument appears in the method signature, R requires the "best" method to be chosen unambiguously for each argument. Ambiguities arise when one method is specific about one argument while another is specific about a different argument. A call that satisfies both requirements is then ambiguous: The two methods look equally valid, which should be chosen? In such cases the package needs to add a third method requiring both arguments to match.

The most common examples arise with binary operators. Methods may be defined for individual operators, for special groups of operators such as `Arith` or for group `Ops`.

Exporting Methods

If a package defines methods for generic functions, those methods should be exported if any of the classes involved are exported; in other words, if someone using the package might expect these methods to be called. Methods are exported by including an `exportMethods()` directive in the `NAMESPACE` file for the package, with the arguments to the directive being the names of the generic functions for which methods have been defined.

Exporting methods is always desirable in the sense of declaring what you want to happen, in that you do expect users to find such methods. It can be essential in the case that the method was defined for a function that is not originally a generic function in its own package (for example, `plot()` in

the graphics package). In this case it may be that the version of the function in the R session is not generic, and your methods will not be called.

Exporting methods for a function also exports the generic version of the function. Keep in mind that this does *not* conflict with the function as it was originally defined in another package; on the contrary, it's designed to ensure that the function in the R session dispatches methods correctly for your classes and continues to behave as expected when no specific methods apply. See Methods_Details for the actual mechanism.

Details

The call to setMethod stores the supplied method definition in the metadata table for this generic function in the environment, typically the global environment or the namespace of a package. In the case of a package, the table object becomes part of the namespace or environment of the package. When the package is loaded into a later session, the methods will be merged into the table of methods in the corresponding generic function object.

Generic functions are referenced by the combination of the function name and the package name; for example, the function "show" from the package "methods". Metadata for methods is identified by the two strings; in particular, the generic function object itself has slots containing its name and its package name. The package name of a generic is set according to the package from which it originally comes; in particular, and frequently, the package where a non-generic version of the function originated. For example, generic functions for all the functions in package **base** will have "base" as the package name, although none of them is an S4 generic on that package. These include most of the base functions that are primitives, rather than true functions; see the section on primitive functions in the documentation for setGeneric for details.

Multiple packages can have methods for the same generic function; that is, for the same combination of generic function name and package name. Even though the methods are stored in separate tables in separate environments, loading the corresponding packages adds the methods to the table in the generic function itself, for the duration of the session.

The class names in the signature can be any formal class, including basic classes such as "numeric", "character", and "matrix". Two additional special class names can appear: "ANY", meaning that this argument can have any class at all; and "missing", meaning that this argument *must not* appear in the call in order to match this signature. Don't confuse these two: if an argument isn't mentioned in a signature, it corresponds implicitly to class "ANY", not to "missing". See the example below. Old-style ('S3') classes can also be used, if you need compatibility with these, but you should definitely declare these classes by calling setOldClass if you want S3-style inheritance to work.

Method definitions can have default expressions for arguments, but only if the generic function must have *some* default expression for the same argument. (This restriction is imposed by the way R manages formal arguments.) If so, and if the corresponding argument is missing in the call to the generic function, the default expression in the method is used. If the method definition has no default for the argument, then the expression supplied in the definition of the generic function itself is used, but note that this expression will be evaluated using the enclosing environment of the method, not of the generic function. Method selection does not evaluate default expressions. All actual (non-missing) arguments in the signature of the generic function will be evaluated when a method is selected—when the call to standardGeneric(f) occurs. Note that specifying class "missing" in the signature does not require any default expressions.

It is possible to have some differences between the formal arguments to a method supplied to setMethod and those of the generic. Roughly, if the generic has ... as one of its arguments, then

the method may have extra formal arguments, which will be matched from the arguments matching ... in the call to f. (What actually happens is that a local function is created inside the method, with the modified formal arguments, and the method is re-defined to call that local function.)

Method dispatch tries to match the class of the actual arguments in a call to the available methods collected for f. If there is a method defined for the exact same classes as in this call, that method is used. Otherwise, all possible signatures are considered corresponding to the actual classes or to superclasses of the actual classes (including "ANY"). The method having the least distance from the actual classes is chosen; if more than one method has minimal distance, one is chosen (the lexicographically first in terms of superclasses) but a warning is issued. All inherited methods chosen are stored in another table, so that the inheritance calculations only need to be done once per session per sequence of actual classes. See Methods_Details and Section 10.7 of the reference for more details.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Methods_for_Nongenerics discusses method definition for functions that are not generic functions in their original package; Methods_for_S3 discusses the integration of formal methods with the older S3 methods.

method.skeleton, which is the recommended way to generate a skeleton of the call to setMethod, with the correct formal arguments and other details.

Methods_Details and the links there for a general discussion, dotsMethods for methods that dispatch on "...", and setGeneric for generic functions.

Examples

```
## examples for a simple class with two numeric slots.
## (Run example(setMethod) to see the class and function definitions)


## methods for plotting track objects
##
## First, with only one object as argument, plot the two slots
##  y must be included in the signature, it would default to "ANY"
setMethod("plot", signature(x="track", y="missing"),
  function(x,  y, ...) plot(x@x, x@y, ...)
)

## plot numeric data on either axis against a track object
## (reducing the track object to the cumulative distance along the track)
## Using a short form for the signature, which matches like formal arguments
setMethod("plot", c("track", "numeric"),
 function(x, y, ...) plot(cumdist(x@x, x@y), y,  xlab = "Distance",...)
)

## and similarly for the other axis
setMethod("plot", c("numeric", "track"),
```

```
 function(x, y, ...) plot(x, cumdist(y@x, y@y),  ylab = "Distance",...)
)

t1 <- new("track", x=1:20, y=(1:20)^2)
plot(t1)
plot(qnorm(ppoints(20)), t1)

## Now a class that inherits from "track", with a vector for data at
## the points
  setClass("trackData", contains = c("numeric", "track"))


tc1 <- new("trackData", t1, rnorm(20))


## a method for plotting the object
## This method has an extra argument, allowed because ... is an
## argument to the generic function.
setMethod("plot", c("trackData", "missing"),
function(x, y, maxRadius = max(par("cin")), ...) {
  plot(x@x, x@y, type = "n", ...)
  symbols(x@x, x@y, circles = abs(x), inches = maxRadius)
  }
)
plot(tc1)

## Without other methods for "trackData", methods for "track"
## will be selected by inheritance

plot(qnorm(ppoints(20)), tc1)

## defining methods for primitive function.
## Although "[" and "length" are not ordinary functions
## methods can be defined for them.
setMethod("[", "track",
  function(x, i, j, ..., drop) {
    x@x <- x@x[i]; x@y <- x@y[i]
    x
  })
plot(t1[1:15])

setMethod("length", "track", function(x)length(x@y))
length(t1)

## Methods for binary operators
## A method for the group generic "Ops" will apply to all operators
## unless a method for a more specific operator has been defined.

## For one trackData argument, go on with just the data part
setMethod("Ops", signature(e1 = "trackData"),
    function(e1, e2) callGeneric(e1@.Data, e2))

setMethod("Ops", signature(e2 = "trackData"),
```

```
    function(e1, e2) callGeneric(e1, e2@.Data))

## At this point, the choice of a method for a call with BOTH
## arguments from "trackData" is ambiguous.  We must define a method.

setMethod("Ops", signature(e1 = "trackData", e2 = "trackData"),
    function(e1, e2) callGeneric(e1@.Data, e2@.Data))
## (well, really we should only do this if the "track" part
## of the two arguments matched)

tc1 +1

1/tc1

all(tc1 == tc1)
```

`setOldClass` *Register Old-Style (S3) Classes and Inheritance*

Description

Register an old-style (a.k.a. 'S3') class as a formally defined class. Simple usage will be of the form:

`setOldClass(Classes)`

where `Classes` is the character vector that would be the `class` attribute of the S3 object. Calls to `setOldClass()` in the code for a package allow the class to be used as a slot in formal (S4) classes and in signatures for methods (see Methods_for_S3). Formal classes can also contain a registered S3 class (see S3Part for details).

If the S3 class has a known set of attributes, an equivalent S4 class can be specified by `S4Class=` in the call to `setOldClass()`; see the section "Known Attributes".

Usage

```
setOldClass(Classes, prototype, where, test = FALSE, S4Class)
```

Arguments

`Classes`	A character vector, giving the names for S3 classes, as they would appear on the right side of an assignment of the `class` attribute in S3 computations. In addition to S3 classes, an object type or other valid data part can be specified, if the S3 class is known to require its data to be of that form.
`S4Class`	optionally, the class definition or the class name of an S4 class. The new class will have all the slots and other properties of this class, plus any S3 inheritance implied by multiple names in the `Classes` argument. See the section on "S3 classes with known attributes" below.

`prototype`, `where`, `test`

These arguments are currently allowed, but not recommended in typical applications.

`prototype`: An optional object to use as the prototype. If the S3 class is not to be `VIRTUAL` (the default), the use of `S4Class=` is preferred.

`where`: Where to store the class definitions. Should be the default (the package namespace) for normal use in an application package.

`test`: flag, if `TRUE`, arrange to test inheritance explicitly for each object, needed if the S3 class can have a different set of class strings, with the same first string. Such classes are inherently malformed, are rare, and should be avoided.

Details

The name (or each of the names) in `Classes` will be defined as an S4 class, extending class `oldClass`, which is the 'root' of all old-style classes. S3 classes with multiple names in their class attribute will have a corresponding inheritance as formal classes. See the `"mlm"` example.

S3 classes have no formal definition, and therefore no formally defined slots. If no S4 class is supplied as a model, the class created will be a virtual class. If a virtual class (any virtual class) is used for a slot in another class, then the initializing method for the class needs to put something legal in that slot; otherwise it will be set to `NULL`.

See Methods_for_S3 for the details of method dispatch and inheritance with mixed S3 and S4 methods.

Some S3 classes cannot be represented as an ordinary combination of S4 classes and superclasses, because objects with the same initial string in the class attribute can have different strings following. Such classes are fortunately rare. They violate the basic idea of object-oriented programming and should be avoided. If you must deal with them, it is still possible to register such classes as S4 classes, but now the inheritance has to be verified for each object, and you must call `setOldClass` with argument `test=TRUE`.

Pre-Defined Old Classes

Many of the widely used S3 classes in the standard R distribution come pre-defined for use with S4. These don't need to be explicitly declared in your package (although it does no harm to do so).

The list `.OldClassesList` contains the old-style classes that are defined by the methods package. Each element of the list is a character vector, with multiple strings if inheritance is included. Each element of the list was passed to `setOldClass` when creating the **methods** package; therefore, these classes can be used in `setMethod` calls, with the inheritance as implied by the list.

S3 Classes with known attributes

A further specification of an S3 class can be made *if* the class is guaranteed to have some attributes of known class (where as with slots, "known" means that the attribute is an object of a specified class, or a subclass of that class).

In this case, the call to `setOldClass()` can supply an S4 class definition representing the known structure. Since S4 slots are implemented as attributes (largely for just this reason), the known attributes can be specified in the representation of the S4 class. The usual technique will be to create an S4 class with the desired structure, and then supply the class name or definition as the argument `S4Class=` to `setOldClass()`.

See the definition of class "ts" in the examples below and the data.frame example in Section 10.2 of the reference. The call to setClass to create the S4 class can use the same class name, as here, so long as the call to setOldClass follows in the same package. For clarity it should be the next expression in the same file.

In the example, we define "ts" as a vector structure with a numeric slot for "tsp". The validity of this definition relies on an assertion that all the S3 code for this class is consistent with that definition; specifically, that all "ts" objects will behave as vector structures and will have a numeric "tsp" attribute. We believe this to be true of all the base code in R, but as always with S3 classes, no guarantee is possible.

The S4 class definition can have virtual superclasses (as in the "ts" case) if the S3 class is asserted to behave consistently with these (in the example, time-series objects are asserted to be consistent with the structure class).

Failures of the S3 class to live up to its asserted behavior will usually go uncorrected, since S3 classes inherently have no definition, and the resulting invalid S4 objects can cause all sorts of grief. Many S3 classes are not candidates for known slots, either because the presence or class of the attributes are not guaranteed (e.g., dimnames in arrays, although these are not even S3 classes), or because the class uses named components of a list rather than attributes (e.g., "lm"). An attribute that is sometimes missing cannot be represented as a slot, not even by pretending that it is present with class "NULL", because attributes, unlike slots, can not have value NULL.

One irregularity that is usually tolerated, however, is to optionally add other attributes to those guaranteed to exist (for example, "terms" in "data.frame" objects returned by model.frame). Validity checks by validObject ignore extra attributes; even if this check is tightened in the future, classes extending S3 classes would likely be exempted because extra attributes are so common.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10, particularly Section 10.8)

See Also

setClass, setMethod

Examples

```
require(stats)

## "lm" and "mlm" are predefined; if they were not this would do it:
## Not run:
setOldClass(c("mlm", "lm"))
## End(Not run)

## Define a new generic function to compute the residual degrees of freedom
setGeneric("dfResidual",
  function(model) stop(gettextf(
    "This function only works for fitted model objects, not class %s",
                                class(model))))
```

```
setMethod("dfResidual", "lm", function(model)model$df.residual)

## dfResidual will work on mlm objects as well as lm objects
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData)



## two examples extending S3 class "lm": class "xlm" directly
## and "ylm" indirectly
setClass("xlm", slots = c(eps = "numeric"), contains = "lm")
setClass("ylm", slots = c(header = "character"), contains = "xlm")
ym1 = new("ylm", myLm, header = "Example", eps = 0.)
## for more examples, see ?\link{S3Class}.




## Not run:
## The code in R that defines "ts" as an S4 class
setClass("ts", contains = "structure", slots = c(tsp = "numeric"),
         prototype(NA, tsp = rep(1,3)))
       # prototype to be a legal S3 time-series
## and now registers it as an S3 class
setOldClass("ts", S4Class = "ts", where = envir)

## End(Not run)
```

---

show *Show an Object*

Description

Display the object, by printing, plotting or whatever suits its class. This function exists to be specialized by methods. The default method calls `showDefault`.

Formal methods for `show` will usually be invoked for automatic printing (see the details).

Usage

```
show(object)
```

Arguments

`object`	Any R object

Details

Objects from an S4 class (a class defined by a call to setClass) will be displayed automatically is if by a call to show. S4 objects that occur as attributes of S3 objects will also be displayed in this form; conversely, S3 objects encountered as slots in S4 objects will be printed using the S3 convention, as if by a call to print.

Methods defined for show will only be inherited by simple inheritance, since otherwise the method would not receive the complete, original object, with misleading results. See the simpleInheritanceOnly argument to setGeneric and the discussion in setIs for the general concept.

Value

show returns an invisible NULL.

See Also

showMethods prints all the methods for one or more functions.

Examples

```
## following the example shown in the setMethod documentation ...
setClass("track", slots = c(x="numeric", y="numeric"))
setClass("trackCurve", contains = "track", slots = c(smooth = "numeric"))

t1 <- new("track", x=1:20, y=(1:20)^2)

tc1 <- new("trackCurve", t1)

setMethod("show", "track",
  function(object)print(rbind(x = object@x, y=object@y))
)
## The method will now be used for automatic printing of t1

t1

## Not run:   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400

## End(Not run)
## and also for tc1, an object of a class that extends "track"
tc1

## Not run:   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
```

```
y   169   196   225   256   289   324   361   400

## End(Not run)
```

---

showMethods *Show all the methods for the specified function(s) or class*

Description

Show a summary of the methods for one or more generic functions, possibly restricted to those involving specified classes.

Usage

```
showMethods(f = character(), where = topenv(parent.frame()),
            classes = NULL, includeDefs = FALSE,
            inherited = !includeDefs,
            showEmpty, printTo = stdout(), fdef)
.S4methods(generic.function, class)
```

Arguments

f	one or more function names. If omitted, all functions will be shown that match the other arguments. The argument can also be an expression that evaluates to a single generic function, in which case argument fdef is ignored. Providing an expression for the function allows examination of hidden or anonymous functions; see the example for isDiagonal().
where	Where to find the generic function, if not supplied as an argument. When f is missing, or length 0, this also determines which generic functions to examine. If where is supplied, only the generic functions returned by getGenerics(where) are eligible for printing. If where is also missing, all the cached generic functions are considered.
classes	If argument classes is supplied, it is a vector of class names that restricts the displayed results to those methods whose signatures include one or more of those classes.
includeDefs	If includeDefs is TRUE, include the definitions of the individual methods in the printout.
inherited	logical indicating if methods that have been found by inheritance, so far in the session, will be included and marked as inherited. Note that an inherited method will not usually appear until it has been used in this session. See selectMethod if you want to know what method would be dispatched for particular classes of arguments.
showEmpty	logical indicating whether methods with no defined methods matching the other criteria should be shown at all. By default, TRUE if and only if argument f is not missing.

`printTo` The connection on which the information will be shown; by default, on standard output.

`fdef` Optionally, the generic function definition to use; if missing, one is found, looking in `where` if that is specified. See also comment in 'Details'.

`generic.function`, `class`
See `methods`.

Details

See `methods` for a description of `.S4methods`.

The name and package of the generic are followed by the list of signatures for which methods are currently defined, according to the criteria determined by the various arguments. Note that the package refers to the source of the generic function. Individual methods for that generic can come from other packages as well.

When more than one generic function is involved, either as specified or because `f` was missing, the functions are found and `showMethods` is recalled for each, including the generic as the argument `fdef`. In complicated situations, this can avoid some anomalous results.

Value

If `printTo` is `FALSE`, the character vector that would have been printed is returned; otherwise the value is the connection or filename, via `invisible`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setMethod`, and `GenericFunctions` for other tools involving methods; `selectMethod` will show you the method dispatched for a particular function and signature of classes for the arguments.

`methods` provides method discovery tools for light-weight interactive use.

Examples

```
require(graphics)

## Assuming the methods for plot
## are set up as in the example of help(setMethod),
## print (without definitions) the methods that involve class "track":
showMethods("plot", classes = "track")
## Not run:
# Function "plot":
# x = ANY, y = track
# x = track, y = missing
# x = track, y = ANY
```

```
require("Matrix")
showMethods("%*%")# many!
    methods(class = "Matrix")# nothing
showMethods(class = "Matrix")# everything
showMethods(Matrix:::isDiagonal) # a non-exported generic

## End(Not run)

if(no4 <- is.na(match("stats4", loadedNamespaces())))
   loadNamespace("stats4")
showMethods(classes = "mle") # -> a method for show()
if(no4) unloadNamespace("stats4")
```

`signature-class` *Class* `"signature"` *For Method Definitions*

Description

This class represents the mapping of some of the formal arguments of a function onto the corresponding classes. It is used for two slots in the `MethodDefinition` class.

Objects from the Class

Objects can be created by calls of the form `new("signature", functionDef, ...)`. The `functionDef` argument, if it is supplied as a function object, defines the formal names. The other arguments define the classes. More typically, the objects are created as side effects of defining methods. Either way, note that the classes are expected to be well defined, usually because the corresponding class definitions exist. See the comment on the `package` slot.

Slots

`.Data`**:** Object of class `"character"` the class names.

`names`**:** Object of class `"character"` the corresponding argument names.

`package`**:** Object of class `"character"` the names of the packages corresponding to the class names. The combination of class name and package uniquely defines the class. In principle, the same class name could appear in more than one package, in which case the `package` information is required for the signature to be well defined.

Extends

Class `"character"`, from data part. Class `"vector"`, by class "character".

Methods

initialize `signature(object = "signature")`: see the discussion of objects from the class, above.

See Also

class `MethodDefinition` for the use of this class.

slot *The Slots in an Object from a Formal Class*

Description

These functions return or set information about the individual slots in an object.

Usage

```
object@name
object@name <- value

slot(object, name)
slot(object, name, check = TRUE) <- value
.hasSlot(object, name)

slotNames(x)
.slotNames(x)
getSlots(x)
```

Arguments

object	An object from a formally defined class.
name	The name of the slot. The operator takes a fixed name, which can be unquoted if it is syntactically a name in the language. A slot name can be any non-empty string, but if the name is not made up of letters, numbers, and ., it needs to be quoted (by backticks or single or double quotes). In the case of the slot function, name can be any expression that evaluates to a valid slot in the class definition. Generally, the only reason to use the functional form rather than the simpler operator is *because* the slot name has to be computed.
value	A new value for the named slot. The value must be valid for this slot in this object's class.
check	In the replacement version of slot, a flag. If TRUE, check the assigned value for validity as the value of this slot. User's code should not set this to FALSE in normal use, since the resulting object can be invalid.
x	either the name of a class (as character string), or a class definition. If given an argument that is neither a character string nor a class definition, slotNames (only) uses class(x) instead.

Details

The definition of the class specifies all slots directly and indirectly defined for that class. Each slot has a name and an associated class. Extracting a slot returns an object from that class. Setting a slot first coerces the value to the specified slot and then stores it.

Unlike general attributes, slots are not partially matched, and asking for (or trying to set) a slot with an invalid name for that class generates an error.

The `@` extraction operator and `slot` function themselves do no checking against the class definition, simply matching the name in the object itself. The replacement forms do check (except for `slot` in the case `check=FALSE`). So long as slots are set without cheating, the extracted slots will be valid.

Be aware that there are two ways to cheat, both to be avoided but with no guarantees. The obvious way is to assign a slot with `check=FALSE`. Also, slots in R are implemented as attributes, for the sake of some back compatibility. The current implementation does not prevent attributes being assigned, via `attr<-`, and such assignments are not checked for legitimate slot names.

Note that the `"@"` operators for extraction and replacement are primitive and actually reside in the **base** package.

The replacement versions of `"@"` and `slot()` differ in the computations done to coerce the right side of the assignment to the declared class of the slot. Both verify that the value provided is from a subclass of the declared slot class. The `slot()` version will go on to call the coerce method if there is one, in effect doing the computation `as(value, slotClass, strict = FALSE)`. The `"@"` version just verifies the relation, leaving any coerce to be done later (e.g., when a relevant method is dispatched).

In most uses the result is equivalent, and the `"@"` version saves an extra function call, but if empirical evidence shows that a conversion is needed, either call `as()` before the replacement or use the replacement version of `slot()`.

Value

The `"@"` operator and the `slot` function extract or replace the formally defined slots for the object.

Functions `slotNames` and `getSlots` return respectively the names of the slots and the classes associated with the slots in the specified class definition. Except for its extended interpretation of `x` (above), `slotNames(x)` is just `names(getSlots(x))`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`@`, `Classes_Details`, `Methods_Details`, `getClass`, `names`.

Examples

```
setClass("track", slots = c(x="numeric", y="numeric"))
myTrack <- new("track", x = -4:4, y = exp(-4:4))
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "y"))
utils::str(myTrack)

getSlots("track") # or
```

```
getSlots(getClass("track"))
slotNames(class(myTrack)) # is the same as
slotNames(myTrack)

## Transform such an S4 object to a list, e.g. to "export" it:
S4toList <- function(obj) {
   sn <- slotNames(obj)
   structure(lapply(sn, slot, object = obj), names = sn)
}
S4toList(myTrack)
```

`StructureClasses` *Classes Corresponding to Basic Structures*

Description

The virtual class `structure` and classes that extend it are formal classes analogous to S language structures such as arrays and time-series.

Usage

```
## The following class names can appear in method signatures,
## as the class in as() and is() expressions, and, except for
## the classes commented as VIRTUAL, in calls to new()

"matrix"
"array"
"ts"

"structure" ## VIRTUAL
```

Objects from the Classes

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted name of the specific class (e.g., `"matrix"`), and the other arguments, if any, are interpreted as arguments to the corresponding function, e.g., to function `matrix()`. There is no particular advantage over calling those functions directly, unless you are writing software designed to work for multiple classes, perhaps with the class name and the arguments passed in.

Objects created from the classes `"matrix"` and `"array"` are unusual, to put it mildly, and have been for some time. Although they may appear to be objects from these classes, they do not have the internal structure of either an S3 or S4 class object. In particular, they have no `"class"` attribute and are not recognized as objects with classes (that is, both `is.object` and `isS4` will return `FALSE` for such objects). However, methods (both S4 and S3) can be defined for these pseudo-classes and new classes (both S4 and S3) can inherit from them.

That the objects still behave as if they came from the corresponding class (most of the time, anyway) results from special code recognizing such objects being built into the base code of R. For most

purposes, treating the classes in the usual way will work, fortunately. One consequence of the special treatment is that these two classes*may* be used as the data part of an S4 class; for example, you can get away with `contains = "matrix"` in a call to `setGeneric` to create an S4 class that is a subclass of `"matrix"`. There is no guarantee that everything will work perfectly, but a number of classes have been written in this form successfully.

Note that a class containing `"matrix"` or `"array"` will have a `.Data` slot with that class. This is the only use of `.Data` other than as a pseudo-class indicating the type of the object. In this case the type of the object will be the type of the contained matrix or array. See `Classes_Details` for a general discussion.

The class `"ts"` is basically an S3 class that has been registered with S4, using the `setOldClass` mechanism. Versions of R through 2.7.0 treated this class as a pure S4 class, which was in principal a good idea, but in practice did not allow subclasses to be defined and had other intrinsic problems. (For example, setting the `"tsp"` parameters as a slot often fails because the built-in implementation does not allow the slot to be temporarily inconsistent with the length of the data. Also, the S4 class prevented the correct specification of the S3 inheritance for class `"mts"`.)

Time-series objects, in contrast to matrices and arrays, have a valid S3 class, `"ts"`, registered using an S4-style definition (see the documentation for `setOldClass` in the examples section for an abbreviated listing of how this is done). The S3 inheritance of `"mts"` in package **stats** is also registered. These classes, as well as `"matrix"` and `"array"` should be valid in most examples as superclasses for new S4 class definitions.

All of these classes have special S4 methods for `initialize` that accept the same arguments as the basic generator functions, `matrix`, `array`, and `ts`, in so far as possible. The limitation is that a class that has more than one non-virtual superclass must accept objects from that superclass in the call to `new`; therefore, a such a class (what is called a "mixin" in some languages) uses the default method for `initialize`, with no special arguments.

Extends

The specific classes all extend class `"structure"`, directly, and class `"vector"`, by class `"structure"`.

Methods

coerce Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "matrix")` calls `as.matrix(x)`. If `strict = TRUE` in the call to `as()`, the method goes on to delete all other slots and attributes other than the `dim` and `dimnames`.

Ops Group methods (see, e.g., `S4groupGeneric`) are defined for combinations of structures and vectors (including special cases for array and matrix), implementing the concept of vector structures as in the reference. Essentially, structures combined with vectors retain the structure as long as the resulting object has the same length. Structures combined with other structures remove the structure, since there is no automatic way to determine what should happen to the slots defining the structure.

Note that these methods will be activated when a package is loaded containing a class that inherits from any of the structure classes or class `"vector"`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (for the original vector structures).

See Also

Class nonStructure, which enforces the alternative model, in which all slots are dropped if any math transformation or operation is applied to an object from a class extending one of the basic classes.

Examples

```
showClass("structure")

## explore a bit :
showClass("ts")
(ts0 <- new("ts"))
str(ts0)

showMethods("Ops") # six methods from these classes, but maybe many more
```

---

testInheritedMethods *Test for and Report about Selection of Inherited Methods*

Description

A set of distinct inherited signatures is generated to test inheritance for all the methods of a specified generic function. If method selection is ambiguous for some of these, a summary of the ambiguities is attached to the returned object. This test should be performed by package authors *before* releasing a package.

Usage

```
testInheritedMethods(f, signatures, test = TRUE, virtual = FALSE,
                     groupMethods = TRUE, where = .GlobalEnv)
```

Arguments

f	a generic function or the character string name of one. By default, all currently defined subclasses of all the method signatures for this generic will be examined. The other arguments are mainly options to modify which inheritance patterns will be examined.
signatures	An optional set of subclass signatures to use instead of the relevant subclasses computed by testInheritedMethods. See the Details for how this is done. This argument might be supplied after a call with test = FALSE, to test selection in batches.

test	optional flag to control whether method selection is actually tested. If FALSE, returns just the list of relevant signatures for subclasses, without calling selectMethod for each signature. If there are a very large number of signatures, you may want to collect the full list and then test them in batches.
virtual	should virtual classes be included in the relevant subclasses. Normally not, since only the classes of actual arguments will trigger the inheritance calculation in a call to the generic function. Including virtual classes may be useful if the class has no current non-virtual subclasses but you anticipate your users may define such classes in the future.
groupMethods	should methods for the group generic function be included?
where	the environment in which to look for class definitions. Nearly always, use the default global environment after attaching all the packages with relevant methods and/or class definitions.

Details

The following description applies when the optional arguments are omitted, the usual case. First, the defining signatures for all methods are computed by calls to findMethodSignatures. From these all the known non-virtual subclasses are found for each class that appears in the signature of some method. These subclasses are split into groups according to which class they inherit from, and only one subclass from each group is retained (for each argument in the generic signature). So if a method was defined with class "vector" for some argument, one actual vector class is chosen arbitrarily. The case of "ANY" is dealt with specially, since all classes extend it. A dummy, nonvirtual class, ".Other", is used to correspond to all classes that have no superclasses among those being tested.

All combinations of retained subclasses for the arguments in the generic signature are then computed. Each row of the resulting matrix is a signature to be tested by a call to selectMethod. To collect information on ambiguous selections, testInheritedMethods establishes a calling handler for the special signal "ambiguousMethodSelection", by setting the corresponding option.

Value

An object of class "methodSelectionReport". The details of this class are currently subject to change. It has slots "target", "selected", "candidates", and "note", all referring to the ambiguous cases (and so of length 0 if there were none). These slots are intended to be examined by the programmer to detect and preferably fix ambiguous method selections. The object contains in addition slots "generic", the name of the generic function, and "allSelections", giving the vector of labels for all the signatures tested.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.6 for basics of method selection.)

Chambers, John M. (2009) *Class Inheritance in R* https://johnmchambers.su.domains/classInheritance.pdf.

Examples

```
## if no other attached packages have methods for `+` or its group
## generic functions, this returns a 16 by 2 matrix of selection
## patterns (in R 2.9.0)
testInheritedMethods("+")
```

TraceClasses *Classes Used Internally to Control Tracing*

Description

The classes described here are used by the R function `trace` to create versions of functions and methods including browser calls, etc., and also to `untrace` the same objects.

Usage

```
### Objects from the following classes are generated
### by calling trace() on an object from the corresponding
### class without the "WithTrace" in the name.

"functionWithTrace"
"MethodDefinitionWithTrace"
"MethodWithNextWithTrace"
"genericFunctionWithTrace"
"groupGenericFunctionWithTrace"

### the following is a virtual class extended by each of the
### classes above

"traceable"
```

Objects from the Class

Objects will be created from these classes by calls to `trace`. (There is an `initialize` method for class `"traceable"`, but you are unlikely to need it directly.)

Slots

`.Data`**:** The data part, which will be `"function"` for class `"functionWithTrace"`, and similarly for the other classes.

`original`**:** Object of the original class; e.g., `"function"` for class `"functionWithTrace"`.

Extends

Each of the classes extends the corresponding untraced class, from the data part; e.g., `"functionWithTrace"` extends `"function"`. Each of the specific classes extends `"traceable"`, directly, and class `"VIRTUAL"`, by class `"traceable"`.

Methods

The point of the specific classes is that objects generated from them, by function `trace()`, remain callable or dispatchable, in addition to their new trace information.

See Also

function `trace`

`validObject` *Test the Validity of an Object*

Description

`validObject()` tests the validity of `object` related to its class definition; specifically, it checks that all slots specified in the class definition are present and that the object in the slot is from the required class or a subclass of that class.

If the object is valid, `TRUE` is returned; otherwise, an error is generated, reporting all the validity failures encountered. If argument `test` is `TRUE`, the errors are returned as a character vector rather than generating an error.

When an object from a class is initialized, the default method for `initialize()` calls `validObject`.

A class definition may have a validity method, set by a call to the function `setValidity`, in the package or environment that defines the class (or via the `validity` argument to `setClass`). The method should be a function of one object that returns `TRUE` or a character-string description of the non-validity. If such a method exists, it will be called from `validObject` and any strings from failure will be included in the result or the error message. Any validity methods defined for superclasses (from the `contains=` argument to `setClass`), will also be called.

Usage

```
validObject(object, test = FALSE, complete = FALSE)

setValidity(Class, method, where = topenv(parent.frame()) )

getValidity(ClassDef)
```

Arguments

`object`	any object, but not much will happen unless the object's class has a formal definition.
`test`	logical; if `TRUE` and validity fails, the function returns a vector of strings describing the problems. If `test` is `FALSE` (the default) validity failure generates an error.
`complete`	logical; if `TRUE`, `validObject` is called recursively for each of the slots. The default is `FALSE`.
`Class`	the name or class definition of the class whose validity method is to be set.

ClassDef	a class definition object, as from getClassDef.
method	a validity method; that is, either NULL or a function of one argument (object). Like validObject, the function should return TRUE if the object is valid, and one or more descriptive strings if any problems are found. Unlike validObject, it should never generate an error.
where	an environment to store the modified class definition. Should be omitted, specifically for calls from a package that defines the class. The definition will be stored in the namespace of the package.

Details

Validity testing takes place 'bottom up', checking the slots, then the superclasses, then the object's own validity method, if there is one.

For each slot and superclass, the existence of the specified class is checked. For each slot, the object in the slot is tested for inheritance from the corresponding class. If complete is TRUE, validObject is called recursively for the object in the slot.

Then, for each of the classes that this class extends (the 'superclasses'), the explicit validity method of that class is called, if one exists. Finally, the validity method of object's class is called, if there is one.

Value

validObject returns TRUE if the object is valid. Otherwise a vector of strings describing problems found, except that if test is FALSE, validity failure generates an error, with the corresponding strings in the error message.

Validity methods

A validity method must be a function of one argument; formally, that argument should be named object. If the argument has a different name, setValidity makes the substitution but in obscure cases that might fail, so it's wiser to name the argument object.

A good method checks all the possible errors and returns a character vector citing all the exceptions found, rather than returning after the first one. validObject will accumulate these errors in its error message or its return value.

Note that validity methods do not have to check validity of superclasses: validObject calls such methods explicitly.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

setClass; class classRepresentation.

Examples

```
setClass("track",
         slots = c(x="numeric", y = "numeric"))
t1 <- new("track", x=1:10, y=sort(stats::rnorm(10)))
## A valid "track" object has the same number of x, y values
validTrackObject <- function(object) {
    if(length(object@x) == length(object@y)) TRUE
    else paste("Unequal x,y lengths: ", length(object@x), ", ",
               length(object@y), sep="")
}
## assign the function as the validity method for the class
setValidity("track", validTrackObject)
## t1 should be a valid "track" object
validObject(t1)
## Now we do something bad
t2 <- t1
t2@x <- 1:20
## This should generate an error
## Not run: try(validObject(t2))


setClass("trackCurve", contains = "track",
         slots = c(smooth = "numeric"))

## all superclass validity methods are used when validObject
## is called from initialize() with arguments, so this fails
## Not run: trynew("trackCurve", t2)


setClass("twoTrack", slots = c(tr1 = "track", tr2 ="track"))

## validity tests are not applied recursively by default,
## so this object is created (invalidly)
tT  <- new("twoTrack", tr2 = t2)

## A stricter test detects the problem
## Not run: try(validObject(tT, complete = TRUE))
```

Chapter 8

The `parallel` package

`parallel-package` *Support for Parallel Computation*

Description

Support for parallel computation, including random-number generation.

Details

This package was first included with R 2.14.0 in 2011.

There is support for multiple RNG streams with the '"L'Ecuyer-CMRG"' RNG: see `nextRNGStream`.

It contains functionality derived from and pretty much equivalent to that contained in packages **multicore** (formerly on CRAN, with some low-level functions renamed and not exported) and **snow** (for socket clusters only, but MPI clusters generated by **snow** are also supported). There have been many enhancements and bug fixes since 2011.

This package also provides `makeForkCluster` to create socket clusters by forking (not Windows).

For a complete list of exported functions, use `library(help = "parallel")`.

Author(s)

Brian Ripley, Luke Tierney and Simon Urbanek

Maintainer: R Core Team <R-core@r-project.org>

See Also

Parallel computation involves launching worker processes: functions `psnice` and `pskill` in package **tools** provide means to manage such processes.

clusterApply *Apply Operations using Clusters*

Description

These functions provide several ways to parallelize computations using a cluster.

Usage

```
clusterCall(cl = NULL, fun, ...)
clusterApply(cl = NULL, x, fun, ...)
clusterApplyLB(cl = NULL, x, fun, ...)
clusterEvalQ(cl = NULL, expr)
clusterExport(cl = NULL, varlist, envir = .GlobalEnv)
clusterMap(cl = NULL, fun, ..., MoreArgs = NULL, RECYCLE = TRUE,
           SIMPLIFY = FALSE, USE.NAMES = TRUE,
           .scheduling = c("static", "dynamic"))
clusterSplit(cl = NULL, seq)

parLapply(cl = NULL, X, fun, ..., chunk.size = NULL)
parSapply(cl = NULL, X, FUN, ..., simplify = TRUE,
          USE.NAMES = TRUE, chunk.size = NULL)
parApply(cl = NULL, X, MARGIN, FUN, ..., chunk.size = NULL)
parRapply(cl = NULL, x, FUN, ..., chunk.size = NULL)
parCapply(cl = NULL, x, FUN, ..., chunk.size = NULL)

parLapplyLB(cl = NULL, X, fun, ..., chunk.size = NULL)
parSapplyLB(cl = NULL, X, FUN, ..., simplify = TRUE,
            USE.NAMES = TRUE, chunk.size = NULL)
```

Arguments

cl	a cluster object, created by this package or by package **snow**. If NULL, use the registered default cluster.
fun, FUN	function or character string naming a function.
expr	expression to evaluate.
seq	vector to split.
varlist	character vector of names of objects to export.
envir	environment from which to export variables
x	a vector for clusterApply and clusterApplyLB, a matrix for parRapply and parCapply.
...	additional arguments to pass to fun or FUN: beware of partial matching to earlier arguments.
MoreArgs	additional arguments for fun.

RECYCLE	logical; if true shorter arguments are recycled.
X	A vector (atomic or list) for parLapply and parSapply, an array for parApply.
chunk.size	scalar number; number of invocations of fun or FUN in one chunk; a chunk is a unit for scheduling.
MARGIN	vector specifying the dimensions to use.
simplify, USE.NAMES	logical; see sapply.
SIMPLIFY	logical; see mapply.
.scheduling	should tasks be statically allocated to nodes or dynamic load-balancing used?

Details

clusterCall calls a function fun with identical arguments ... on each node.

clusterEvalQ evaluates a literal expression on each cluster node. It is a parallel version of evalq, and is a convenience function invoking clusterCall.

clusterApply calls fun on the first node with arguments x[[1]] and ..., on the second node with x[[2]] and ..., and so on, recycling nodes as needed.

clusterApplyLB is a load balancing version of clusterApply. If the length n of x is not greater than the number of nodes p, then a job is sent to n nodes. Otherwise the first p jobs are placed in order on the p nodes. When the first job completes, the next job is placed on the node that has become free; this continues until all jobs are complete. Using clusterApplyLB can result in better cluster utilization than using clusterApply, but increased communication can reduce performance. Furthermore, the node that executes a particular job is non-deterministic. This means that simulations that assign RNG streams to nodes will not be reproducible.

clusterMap is a multi-argument version of clusterApply, analogous to mapply and Map. If RECYCLE is true shorter arguments are recycled (and either none or all must be of length zero); otherwise, the result length is the length of the shortest argument. Nodes are recycled if the length of the result is greater than the number of nodes. (mapply always uses RECYCLE = TRUE, and has argument SIMPLIFY = TRUE. Map always uses RECYCLE = TRUE.)

clusterExport assigns the values on the master R process of the variables named in varlist to variables of the same names in the global environment (aka 'workspace') of each node. The environment on the master from which variables are exported defaults to the global environment.

clusterSplit splits seq into a consecutive piece for each cluster and returns the result as a list with length equal to the number of nodes. Currently the pieces are chosen to be close to equal in length: the computation is done on the master.

parLapply, parSapply, and parApply are parallel versions of lapply, sapply and apply. Chunks of computation are statically allocated to nodes using clusterApply. By default, the number of chunks is the same as the number of nodes. parLapplyLB, parSapplyLB are load-balancing versions, intended for use when applying FUN to different elements of X takes quite variable amounts of time, and either the function is deterministic or reproducible results are not required. Chunks of computation are allocated dynamically to nodes using clusterApplyLB. From R 3.5.0, the default number of chunks is twice the number of nodes. Before R 3.5.0, the (fixed) number of chunks was the same as the number of nodes. As for clusterApplyLB, with load balancing the node that executes a particular job is non-deterministic and simulations that assign RNG streams to nodes will not be reproducible.

parRapply and parCapply are parallel row and column apply functions for a matrix x; they may be slightly more efficient than parApply but do less post-processing of the result.

A chunk size of 0 with static scheduling uses the default (one chunk per node). With dynamic scheduling, chunk size of 0 has the same effect as 1 (one invocation of FUN/fun per chunk).

Value

For clusterCall, clusterEvalQ and clusterSplit, a list with one element per node.

For clusterApply and clusterApplyLB, a list the same length as x.

clusterMap follows mapply.

clusterExport returns nothing.

parLapply returns a list the length of X.

parSapply and parApply follow sapply and apply respectively.

parRapply and parCapply always return a vector. If FUN always returns a scalar result this will be of length the number of rows or columns: otherwise it will be the concatenation of the returned values.

An error is signalled on the master if any of the workers produces an error.

Note

These functions are almost identical to those in package **snow**.

Two exceptions: parLapply has argument X not x for consistency with lapply, and parSapply has been updated to match sapply.

Author(s)

Luke Tierney and R Core.

Derived from the **snow** package.

Examples

```
## Use option cl.cores to choose an appropriate cluster size.
cl <- makeCluster(getOption("cl.cores", 2))

clusterApply(cl, 1:2, get("+"), 3)
xx <- 1
clusterExport(cl, "xx")
clusterCall(cl, function(y) xx + y, 2)

## Use clusterMap like an mapply example
clusterMap(cl, function(x, y) seq_len(x) + y,
          c(a =  1, b = 2, c = 3), c(A = 10, B = 0, C = -10))


parSapply(cl, 1:20, get("+"), 3)

## A bootstrapping example, which can be done in many ways:
clusterEvalQ(cl, {
```

```
  ## set up each worker.  Could also use clusterExport()
  library(boot)
  cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
  cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
  NULL
})
res <- clusterEvalQ(cl, boot(cd4, corr, R = 100,
                    sim = "parametric", ran.gen = cd4.rg, mle = cd4.mle))
library(boot)
cd4.boot <- do.call(c, res)
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
stopCluster(cl)

## or
library(boot)
run1 <- function(...) {
   library(boot)
   cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
   cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
   boot(cd4, corr, R = 500, sim = "parametric",
        ran.gen = cd4.rg, mle = cd4.mle)
}
cl <- makeCluster(mc <- getOption("cl.cores", 2))
## to make this reproducible
clusterSetRNGStream(cl, 123)
cd4.boot <- do.call(c, parLapply(cl, seq_len(mc), run1))
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
stopCluster(cl)
```

`detectCores` *Detect the Number of CPU Cores*

Description

Attempt to detect the number of CPU cores on the current host.

Usage

```
detectCores(all.tests = FALSE, logical = TRUE)
```

Arguments

`all.tests`	Logical: if true apply all known tests.
`logical`	Logical: if possible, use the number of physical CPUs/cores (if `FALSE`) or logical CPUs (if `TRUE`). Currently this is honoured only on macOS, Solaris and Windows.

Details

This attempts to detect the number of available CPU cores.

It has methods to do so for Linux, macOS, FreeBSD, OpenBSD, Solaris and Windows. `detectCores(TRUE)` could be tried on other Unix-alike systems.

Value

An integer, `NA` if the answer is unknown.

Exactly what this represents is OS-dependent: where possible by default it counts logical (e.g., hyperthreaded) CPUs and not physical cores or packages.

Under macOS there is a further distinction between 'available in the current power management mode' and 'could be available this boot', and this function returns the first.

On Sparc Solaris `logical = FALSE` returns the number of physical cores and `logical = TRUE` returns the number of available hardware threads. (Some Sparc CPUs have multiple cores per CPU, others have multiple threads per core and some have both.) For example, the UltraSparc T2 CPU in the former CRAN check server was a single physical CPU with 8 cores, and each core supports 8 hardware threads. So `detectCores(logical = FALSE)` returns 8, and `detectCores(logical = TRUE)` returns 64.

Where virtual machines are in use, one would hope that the result for `logical = TRUE` represents the number of CPUs available (or potentially available) to that particular VM.

Note

This is not suitable for use directly for the `mc.cores` argument of `mclapply` nor specifying the number of cores in `makeCluster`. First because it may return `NA`, second because it does not give the number of *allowed* cores, and third because on Sparc Solaris and some Windows boxes it is not reasonable to try to use all the logical CPUs at once.

Author(s)

Simon Urbanek and Brian Ripley

Examples

```
detectCores()
detectCores(logical = FALSE)
```

`makeCluster` *Create a Parallel Socket Cluster*

Description

Creates a set of copies of R running in parallel and communicating over sockets.

Usage

```
makeCluster(spec, type, ...)
makePSOCKcluster(names, ...)
makeForkCluster(nnodes = getOption("mc.cores", 2L), ...)

stopCluster(cl = NULL)

setDefaultCluster(cl = NULL)
getDefaultCluster()
```

Arguments

spec	A specification appropriate to the type of cluster.
names	Either a character vector of host names on which to run the worker copies of R, or a positive integer (in which case that number of copies is run on 'localhost').
nnodes	The number of nodes to be forked.
type	One of the supported types: see 'Details'.
...	Options to be passed to the function spawning the workers. See 'Details'.
cl	an object of class "cluster".

Details

makeCluster creates a cluster of one of the supported types. The default type, "PSOCK", calls makePSOCKcluster. Type "FORK" calls makeForkCluster. Other types are passed to package **snow**.

makePSOCKcluster is an enhanced version of makeSOCKcluster in package **snow**. It runs Rscript on the specified host(s) to set up a worker process which listens on a socket for expressions to evaluate, and returns the results (as serialized objects).

makeForkCluster is merely a stub on Windows. On Unix-alike platforms it creates the worker process by forking.

The workers are most often running on the same host as the master, when no options need be set.

Several options are supported (mainly for makePSOCKcluster):

master The host name of the master, as known to the workers. This may not be the same as it is known to the master, and on private subnets it may be necessary to specify this as a numeric IP address. For example, macOS is likely to detect a machine as 'somename.local', a name known only to itself.

port The port number for the socket connection, default taken from the environment variable R_PARALLEL_PORT, then a randomly chosen port in the range 11000:11999.

timeout The timeout in seconds for that port. This is the maximum time of zero communication between master and worker before failing. Default is 30 days (and the POSIX standard only requires values up to 31 days to be supported).

setup_timeout The maximum number of seconds a worker attempts to connect to master before failing. Default is 2 minutes. The waiting time before the next attempt starts at 0.1 seconds and is incremented 50% after each retry.

`outfile` Where to direct the `stdout` and `stderr` connection output from the workers. `""` indicates no redirection (which may only be useful for workers on the local machine). Defaults to '`/dev/null`' ('`nul:`' on Windows). The other possibility is a file path on the worker's host. Files will be opened in append mode, as all workers log to the same file.

`homogeneous` Logical, default true. See 'Note'.

`rscript` See 'Note'.

`rscript_args` Character vector of additional arguments for `Rscript` such as '`--no-environ`'.

`renice` A numerical 'niceness' to set for the worker processes, e.g. `15` for a low priority. OS-dependent: see `psnice` for details.

`rshcmd` The command to be run on the master to launch a process on another host. Defaults to `ssh`.

`user` The user name to be used when communicating with another host.

`manual` Logical. If true the workers will need to be run manually.

`methods` Logical. If true (default) the workers will load the **methods** package: not loading it saves ca 30% of the startup CPU time of the cluster.

`useXDR` Logical. If true (default) serialization will use XDR: where large amounts of data are to be transferred and all the nodes are little-endian, communication may be substantially faster if this is set to false.

`setup_strategy` Character. If `"parallel"` (default) workers will be started in parallel during cluster setup when this is possible, which is now for homogeneous `"PSOCK"` clusters with all workers started automatically (`manual = FALSE`) on the local machine. Workers will be started sequentially on other clusters, on all clusters with `setup_strategy = "sequential"` and on R 3.6.0 and older. This option is for expert use only (e.g. debugging) and may be removed in future versions of R.

Function `makeForkCluster` creates a socket cluster by forking (and hence is not available on Windows). It supports options `port`, `timeout` and `outfile`, and always uses `useXDR = FALSE`. It is *strongly discouraged* to use the `"FORK"` cluster with GUI front-ends or multi-threaded libraries. See `mcfork` for details.

It is good practice to shut down the workers by calling `stopCluster`: however the workers will terminate themselves once the socket on which they are listening for commands becomes unavailable, which it should if the master R session is completed (or its process dies).

Function `setDefaultCluster` registers a cluster as the default one for the current session. Using `setDefaultCluster(NULL)` removes the registered cluster, as does stopping that cluster.

Value

For the cluster creators, an object of class `c("SOCKcluster", "cluster")`.

For the default cluster setter and getter, the registered default cluster or `NULL` if there is no such cluster.

Note

Option `homogeneous = TRUE` was for years documented as 'Are all the hosts running identical setups?', but this was apparently more restrictive than its author intended and not required by the code.

The current interpretation of `homogeneous = TRUE` is that `Rscript` can be launched using the same path on each worker. That path is given by the option `rscript` and defaults to the full path to `Rscript` on the master. (The workers are not required to be running the same version of R as the master, nor even as each other.)

For `homogeneous = FALSE`, `Rscript` on the workers is found on their default shell's path.

For the very common usage of running both master and worker on a single multi-core host, the default settings are the appropriate ones.

A socket connection is used to communicate from the master to each worker so the maximum number of connections (default 128 but some will be in use) may need to be increased when the master process is started.

Author(s)

Luke Tierney and R Core.

Derived from the **snow** package.

`mcaffinity` *Get or Set CPU Affinity Mask of the Current Process*

Description

`mcaffinity` retrieves or sets the CPU affinity mask of the current process, i.e., the set of CPUs the process is allowed to be run on. (CPU here means logical CPU which can be CPU, core or hyperthread unit.)

Usage

```
mcaffinity(affinity = NULL)
```

Arguments

`affinity`	specification of the CPUs to lock this process to (numeric vector) or `NULL` if no change is requested

Details

`mcaffinity` can be used to obtain (`affinity = NULL`) or set the CPU affinity mask of the current process. The affinity mask is a list of integer CPU identifiers (starting from 1) that this process is allowed to run on. Not all systems provide user access to the process CPU affinity, in cases where no support is present at all `mcaffinity()` will return `NULL`. Some systems may take into account only the number of CPUs present in the mask.

Typically, it is legal to specify larger set than the number of logical CPUs (but at most as many as the OS can handle) and the system will return back the actually present set.

Value

NULL if CPU affinity is not supported by the system or an integer vector with the set of CPUs in the active affinity mask for this process (this may be different than affinity).

Author(s)

Simon Urbanek.

See Also

mcparallel

---

mcchildren *Low-level Functions for Management of Forked Processes*

Description

These are low-level support functions for the forking approach.

They are not available on Windows, and not exported from the namespace.

Usage

```
children(select)
readChild(child)
readChildren(timeout = 0)
selectChildren(children = NULL, timeout = 0)
sendChildStdin(child, what)
sendMaster(what, raw.asis = TRUE)

mckill(process, signal = 2L)
```

Arguments

select	if omitted, all active children are returned, otherwise select should be a list of processes and only those from the list that are active will be returned.
child	child process (object of the class "childProcess") or a process ID (pid). See also 'Details'.
timeout	timeout (in seconds, fractions supported) to wait for a response before giving up.
children	list of child processes or a single child process object or a vector of process IDs or NULL. If NULL behaves as if all currently known children were supplied.
what	For sendChildStdin: Character or raw vector. In the former case elements are collapsed using the newline character. (But no trailing newline is added at the end!)

	For sendMaster: Data to send to the master process. If what is not a raw vector, it will be serialized into a raw vector. Do NOT send an empty raw vector – that is reserved for internal use.
raw.asis	logical, if TRUE and what is a raw vector then it is sent directly as-is to the master (default, suitable for arbitrary payload passing), otherwise raw vectors are serialized before sending just as any other objects (suitable for passing evaluation results).
process	process (object of the class process) or a process ID (pid)
signal	integer: signal to send. Values of 2 (SIGINT), 9 (SIGKILL) and 15 (SIGTERM) are pretty much portable, but for maximal portability use tools::SIGTERM and so on.

Details

children returns currently active children.

readChild reads data (sent by sendMaster) from a given child process.

selectChildren checks children for available data.

readChildren checks all children for available data and reads from the first child that has available data.

sendChildStdin sends a string (or data) to one or more child's standard input. Note that if the master session was interactive, it will also be echoed on the standard output of the master process (unless disabled). The function is vector-compatible, so you can specify child as a list or a vector of process IDs.

sendMaster sends data from the child to the master process.

mckill sends a signal to a child process: it is equivalent to pskill in package **tools**.

Value

children returns a (possibly empty) list of objects of class "process", the process ID.

readChild and readChildren return a raw vector with a "pid" attribute if data were available, an integer vector of length one with the process ID if a child terminated or NULL if the child no longer exists (no children at all for readChildren).

selectChildren returns TRUE is the timeout was reached, FALSE if an error occurred (e.g., if the master process was interrupted) or an integer vector of process IDs with children that have data available, or NULL if there are no children.

sendChildStdin returns a vector of TRUE values (one for each member of child) or throws an error.

sendMaster returns TRUE or throws an error.

mckill returns TRUE.

Warning

This is a very low-level interface for expert use only: it not regarded as part of the R API and subject to change without notice.

`sendMaster`, `readChild` and `sendChildStdin` did not support long vectors prior to R 3.4.0 and so were limited to $2^{31} - 1$ bytes (and still are on 32-bit platforms).

Author(s)

Simon Urbanek and R Core.

Derived from the **multicore** package formerly on CRAN.

See Also

`mcfork`, `sendMaster`, `mcparallel`

Examples

```
## Not run:
p  <- mcparallel(scan(n = 1, quiet = TRUE))
sendChildStdin(p, "17.4\n")
mccollect(p)[[1]]

## End(Not run)
```

---

mcfork *Fork a Copy of the Current R Process*

Description

These are low-level functions, not available on Windows, and not exported from the namespace.

`mcfork` creates a new child process as a copy of the current R process.

`mcexit` closes the current child process, informing the master process as necessary.

Usage

```
mcfork(estranged = FALSE)

mcexit(exit.code = 0L, send = NULL)
```

Arguments

`estranged`	logical, if `TRUE` then the new process has no ties to the parent process, will not show in the list of children and will not be killed on exit.
`exit.code`	process exit code. By convention `0L` signifies a clean exit, `1L` an error.
`send`	if not `NULL` send this data before exiting (equivalent to using `sendMaster`).

Details

The mcfork function provides an interface to the fork system call. In addition it sets up a pipe between the master and child process that can be used to send data from the child process to the master (see sendMaster) and child's 'stdin' is re-mapped to another pipe held by the master process (see sendChildStdin).

If you are not familiar with the fork system call, do not use this function directly as it leads to very complex inter-process interactions amongst the R processes involved.

In a nutshell fork spawns a copy (child) of the current process, that can work in parallel to the master (parent) process. At the point of forking both processes share exactly the same state including the workspace, global options, loaded packages etc. Forking is relatively cheap in modern operating systems and no real copy of the used memory is created, instead both processes share the same memory and only modified parts are copied. This makes mcfork an ideal tool for parallel processing since there is no need to setup the parallel working environment, data and code is shared automatically from the start.

mcexit is to be run in the child process. It sends send to the master (unless NULL) and then shuts down the child process. The child can also be shut down by sending it the signal SIGUSR1, as is done by the unexported function parallel:::rmChild.

Value

mcfork returns an object of the class "childProcess" to the master and of class "masterProcess" to the child: both the classes inherit from class "process". If estranged is set to TRUE then the child process will be of the class "estrangedProcess" and cannot communicate with the master process nor will it show up on the list of children. These are lists with components pid (the process id of the *other* process) and a vector fd of the two file descriptor numbers for ends in the current process of the inter-process pipes.

mcexit never returns.

GUI/embedded environments

It is *strongly discouraged* to use mcfork and the higher-level functions which rely on it (e.g., mcparallel, mclapply and pvec) in GUI or embedded environments, because it leads to several processes sharing the same GUI which will likely cause chaos (and possibly crashes). Child processes should never use on-screen graphics devices. Some precautions have been taken to make this usable in R.app on macOS, but users of third-party front-ends should consult their documentation.

This can also apply to other connections (e.g., to an X server) created before forking, and to files opened by e.g. graphics devices.

Note that **tcltk** counts as a GUI for these purposes since Tcl runs an event loop. That event loop is inhibited in a child process but there could still be problems with Tk graphical connections.

It is *strongly discouraged* to use mcfork and the higher-level functions in any multi-threaded R process (with additional threads created by a third-party library or package). Such use can lead to deadlocks or crashes, because the child process created by mcfork may not be able to access resources locked in the parent or may see an inconsistent version of global data (mcfork runs system call fork without exec).

If in doubt, it is safer to use a non-FORK cluster (see makeCluster, clusterApply).

Warning

This is a very low-level API for expert use only.

Author(s)

Simon Urbanek and R Core.

Derived from the **multicore** package formerly on CRAN.

See Also

mcparallel, sendMaster

Examples

```
## This will work when run as an example, but not when pasted in.
p <- parallel:::mcfork()
if (inherits(p, "masterProcess")) {
    cat("I'm a child! ", Sys.getpid(), "\n")
    parallel:::mcexit(,"I was a child")
}
cat("I'm the master\n")
unserialize(parallel:::readChildren(1.5))
```

---

mclapply *Parallel Versions of* lapply *and* mapply *using Forking*

Description

mclapply is a parallelized version of lapply, it returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

It relies on forking and hence is not available on Windows unless mc.cores = 1.

mcmapply is a parallelized version of mapply, and mcMap corresponds to Map.

Usage

```
mclapply(X, FUN, ...,
         mc.preschedule = TRUE, mc.set.seed = TRUE,
         mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
         mc.cleanup = TRUE, mc.allow.recursive = TRUE, affinity.list = NULL)

mcmapply(FUN, ...,
         MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE,
         mc.preschedule = TRUE, mc.set.seed = TRUE,
         mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
         mc.cleanup = TRUE, affinity.list = NULL)

mcMap(f, ...)
```

Arguments

X	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by as.list.
FUN	the function to be applied to (mclapply) each element of X or (mcmapply) in parallel to ....
f	the function to be applied in parallel to ....
...	For mclapply, optional arguments to FUN. For mcmapply and mcMap, vector or list inputs: see mapply.
MoreArgs, SIMPLIFY, USE.NAMES	see mapply.
mc.preschedule	if set to TRUE then the computation is first divided to (at most) as many jobs are there are cores and then the jobs are started, each job possibly covering more than one value. If set to FALSE then one job is forked for each value of X. The former is better for short computations or large number of values in X, the latter is better for jobs that have high variance of completion time and not too many values of X compared to mc.cores.
mc.set.seed	See mcparallel.
mc.silent	if set to TRUE then all output on 'stdout' will be suppressed for all parallel processes forked ('stderr' is not affected).
mc.cores	The number of cores to use, i.e. at most how many child processes will be run simultaneously. The option is initialized from environment variable MC_CORES if set. Must be at least one, and parallelization requires at least two cores.
mc.cleanup	if set to TRUE then all children that have been forked by this function will be killed (by sending SIGTERM) before this function returns. Under normal circumstances mclapply waits for the children to deliver results, so this option usually has only effect when mclapply is interrupted. If set to FALSE then child processes are collected, but not forcefully terminated. As a special case this argument can be set to the number of the signal that should be used to kill the children instead of SIGTERM.
mc.allow.recursive	Unless true, calling mclapply in a child process will use the child and not fork again.
affinity.list	a vector (atomic or list) containing the CPU affinity mask for each element of X. The CPU affinity mask describes on which CPU (core or hyperthread unit) a given item is allowed to run, see mcaffinity. To use this parameter prescheduling has to be deactivated (mc.preschedule = FALSE).

Details

mclapply is a parallelized version of lapply, provided mc.cores > 1: for mc.cores == 1 (and the affinity.list is NULL) it simply calls lapply.

By default (mc.preschedule = TRUE) the input X is split into as many parts as there are cores (currently the values are spread across the cores sequentially, i.e. first value to core 1, second to core 2, ...(core + 1)-th value to core 1 etc.) and then one process is forked to each core and the results are collected.

Without prescheduling, a separate job is forked for each value of X. To ensure that no more than mc.cores jobs are running at once, once that number has been forked the master process waits for a child to complete before the next fork.

Due to the parallel nature of the execution random numbers are not sequential (in the random number sequence) as they would be when using lapply. They are sequential for each forked process, but not all jobs as a whole. See mcparallel or the package's vignette for ways to make the results reproducible with mc.preschedule = TRUE.

Note: the number of file descriptors (and processes) is usually limited by the operating system, so you may have trouble using more than 100 cores or so (see ulimit -n or similar in your OS documentation) unless you raise the limit of permissible open file descriptors (fork will fail with error "unable to create a pipe").

Prior to R 3.4.0 and on a 32-bit platform, the serialized result from each forked process is limited to $2^{31} - 1$ bytes. (Returning very large results via serialization is inefficient and should be avoided.)

affinity.list can be used to run elements of X on specific CPUs. This can be helpful, if elements of X have a high variance of completion time or if the hardware architecture is heterogeneous. It also enables the development of scheduling strategies for optimizing the overall runtime of parallel jobs. If affinity.list is set, the mc.core parameter is replaced with the number of CPU ids used in the affinity masks.

Value

For mclapply, a list of the same length as X and named by X.

For mcmapply, a list, vector or array: see mapply.

For mcMap, a list.

Each forked process runs its job inside try(..., silent = TRUE) so if errors occur they will be stored as class "try-error" objects in the return value and a warning will be given. Note that the job will typically involve more than one value of X and hence a "try-error" object will be returned for all the values involved in the failure, even if not all of them failed. If any forked process is killed or fails to deliver a result for any reason, values involved in the failure will be NULL. To allow detection of such errors, FUN should not return NULL. As of R 4.0, the return value of mcmapply is always a list when it needs to contain "try-error" objects (SIMPLIFY is overridden to FALSE).

Warning

It is *strongly discouraged* to use these functions in GUI or embedded environments, because it leads to several processes sharing the same GUI which will likely cause chaos (and possibly crashes). Child processes should never use on-screen graphics devices.

Some precautions have been taken to make this usable in R.app on macOS, but users of third-party front-ends should consult their documentation.

Note that **tcltk** counts as a GUI for these purposes since Tcl runs an event loop. That event loop is inhibited in a child process but there could still be problems with Tk graphical connections.

It is *strongly discouraged* to use these functions with multi-threaded libraries or packages (see mcfork for more details). If in doubt, it is safer to use a non-FORK cluster (see makeCluster, clusterApply).

Author(s)

Simon Urbanek and R Core. The `affinity.list` feature by Helena Kotthaus and Andreas Lang, TU Dortmund. Derived from the **multicore** package formerly on CRAN.

See Also

`mcparallel`, `pvec`, `parLapply`, `clusterMap`.

`simplify2array` for results like `sapply`.

Examples

```
simplify2array(mclapply(rep(4, 5), rnorm))
# use the same random numbers for all values
set.seed(1)
simplify2array(mclapply(rep(4, 5), rnorm, mc.preschedule = FALSE,
                        mc.set.seed = FALSE))

## Contrast this with the examples for clusterCall
library(boot)
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
mc <- getOption("mc.cores", 2)
run1 <- function(...) boot(cd4, corr, R = 500, sim = "parametric",
                           ran.gen = cd4.rg, mle = cd4.mle)
## To make this reproducible:
set.seed(123, "L'Ecuyer")
res <- mclapply(seq_len(mc), run1)
cd4.boot <- do.call(c, res)
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)

## Usage of the affinity.list parameter
A <- runif(2500000,0,100)
B <- runif(2500000,0,100)
C <- runif(5000000,0,100)
first <- function(i) head(sort(i), n = 1)

# Restict all elements of X to run on CPU 1 and 2
affL <- list(c(1,2), c(1,2), c(1,2))
mclapply(list(A, A, A), first, mc.preschedule = FALSE, affinity.list = affL)


# Completion times are assumed to have a high variance
# To optimize the overall execution time elements of X are scheduled to suitable CPUs
# Assuming that the runtime for C is as long as the runtime of A plus B
# mapping: A to 1 , B to 1, C to 2
X <- list(A, B, C)
affL <- c(1, 1, 2)
mclapply(X, first, mc.preschedule = FALSE, affinity.list = affL)
```

mcparallel *Evaluate an R Expression Asynchronously in a Separate Process*

Description

These functions are based on forking and so are not available on Windows.

`mcparallel` starts a parallel R process which evaluates the given expression.

`mccollect` collects results from one or more parallel processes.

Usage

```
mcparallel(expr, name, mc.set.seed = TRUE, silent = FALSE,
           mc.affinity = NULL, mc.interactive = FALSE,
    detached = FALSE)

mccollect(jobs, wait = TRUE, timeout = 0, intermediate = FALSE)
```

Arguments

`expr`	expression to evaluate (do *not* use any on-screen devices or GUI elements in this code, see `mcfork` for the inadvisability of using `mcparallel` with GUI front-ends and multi-threaded libraries). Raw vectors are reserved for internal use and cannot be returned, but the expression may evaluate e.g. to a list holding a raw vector. `NULL` should not be returned because it is used by `mccollect` to signal an error.
`name`	an optional name (character vector of length one) that can be associated with the job.
`mc.set.seed`	logical: see section 'Random numbers'.
`silent`	if set to `TRUE` then all output on stdout will be suppressed (stderr is not affected).
`mc.affinity`	either a numeric vector specifying CPUs to restrict the child process to (1-based) or `NULL` to not modify the CPU affinity
`mc.interactive`	logical, if `TRUE` or `FALSE` then the child process will be set as interactive or non-interactive respectively. If `NA` then the child process will inherit the interactive flag from the parent.
`detached`	logical, if `TRUE` then the job is detached from the current session and cannot deliver any results back - it is used for the code side-effect only.
`jobs`	list of jobs (or a single job) to collect results for. Alternatively `jobs` can also be an integer vector of process IDs. If omitted `collect` will wait for all currently existing children.
`wait`	if set to `FALSE` it checks for any results that are available within `timeout` seconds from now, otherwise it waits for all specified jobs to finish.
`timeout`	timeout (in seconds) to check for job results – applies only if `wait` is `FALSE`.
`intermediate`	`FALSE` or a function which will be called while `collect` waits for results. The function will be called with one parameter which is the list of results received so far.

Details

mcparallel evaluates the expr expression in parallel to the current R process. Everything is shared read-only (or in fact copy-on-write) between the parallel process and the current process, i.e. no side-effects of the expression affect the main process. The result of the parallel execution can be collected using mccollect function.

mccollect function collects any available results from parallel jobs (or in fact any child process). If wait is TRUE then collect waits for all specified jobs to finish before returning a list containing the last reported result for each job. If wait is FALSE then mccollect merely checks for any results available at the moment and will not wait for jobs to finish. If jobs is specified, jobs not listed there will not be affected or acted upon.

Note: If expr uses low-level multicore functions such as sendMaster a single job can deliver results multiple times and it is the responsibility of the user to interpret them correctly. mccollect will return NULL for a terminating job that has sent its results already after which the job is no longer available.

Jobs are identified by process IDs (even when referred to as job objects), which are reused by the operating system. Detached jobs created by mcparallel can thus never be safely referred to by their process IDs nor job objects. Non-detached jobs are guaranteed to exist until collected by mccollect, even if crashed or terminated by a signal. Once collected by mccollect, a job is regarded as detached, and thus must no longer be referred to by its process ID nor its job object. With wait = TRUE, all jobs passed to mccollect are collected. With wait = FALSE, the collected jobs are given as names of the result vector, and thus in subsequent calls to mccollect these jobs must be excluded. Job objects should be used in preference of process IDs whenever accepted by the API.

The mc.affinity parameter can be used to try to restrict the child process to specific CPUs. The availability and the extent of this feature is system-dependent (e.g., some systems will only consider the CPU count, others will ignore it completely).

Value

mcparallel returns an object of the class "parallelJob" which inherits from "childProcess" (see the 'Value' section of the help for mcfork). If argument name was supplied this will have an additional component name.

mccollect returns any results that are available in a list. The results will have the same order as the specified jobs. If there are multiple jobs and a job has a name it will be used to name the result, otherwise its process ID will be used. If none of the specified children are still running, it returns NULL.

Random numbers

If mc.set.seed = FALSE, the child process has the same initial random number generator (RNG) state as the current R session. If the RNG has been used (or .Random.seed was restored from a saved workspace), the child will start drawing random numbers at the same point as the current session. If the RNG has not yet been used, the child will set a seed based on the time and process ID when it first uses the RNG: this is pretty much guaranteed to give a different random-number stream from the current session and any other child process.

The behaviour with mc.set.seed = TRUE is different only if RNGkind("L'Ecuyer-CMRG") has been selected. Then each time a child is forked it is given the next stream (see nextRNGStream). So if you

select that generator, set a seed and call `mc.reset.stream` just before the first use of `mcparallel` the results of simulations will be reproducible provided the same tasks are given to the first, second, ... forked process.

Note

Prior to R 3.4.0 and on a 32-bit platform, the serialized result from each forked process is limited to $2^{31} - 1$ bytes. (Returning very large results via serialization is inefficient and should be avoided.)

Author(s)

Simon Urbanek and R Core.

Derived from the **multicore** package formerly on CRAN. (but with different handling of the RNG stream).

See Also

`pvec`, `mclapply`

Examples

```
p <- mcparallel(1:10)
q <- mcparallel(1:20)
# wait for both jobs to finish and collect all results
res <- mccollect(list(p, q))


p <- mcparallel(1:10)
mccollect(p, wait = FALSE, 10) # will retrieve the result (since it's fast)
mccollect(p, wait = FALSE)     # will signal the job as terminating
mccollect(p, wait = FALSE)     # there is no longer such a job


# a naive parallel lapply can be created using mcparallel alone:
jobs <- lapply(1:10, function(x) mcparallel(rnorm(x), name = x))
mccollect(jobs)
```

---

pvec *Parallelize a Vector Map Function using Forking*

Description

`pvec` parallelizes the execution of a function on vector elements by splitting the vector and submitting each part to one core. The function must be a vectorized map, i.e. it takes a vector input and creates a vector output of exactly the same length as the input which doesn't depend on the partition of the vector.

It relies on forking and hence is not available on Windows unless `mc.cores = 1`.

Usage

```
pvec(v, FUN, ..., mc.set.seed = TRUE, mc.silent = FALSE,
     mc.cores = getOption("mc.cores", 2L), mc.cleanup = TRUE)
```

Arguments

v	vector to operate on
FUN	function to call on each part of the vector
...	any further arguments passed to FUN after the vector
mc.set.seed	See mcparallel.
mc.silent	if set to TRUE then all output on 'stdout' will be suppressed for all parallel processes forked ('stderr' is not affected).
mc.cores	The number of cores to use, i.e. at most how many child processes will be run simultaneously. Must be at least one, and at least two for parallel operation. The option is initialized from environment variable MC_CORES if set.
mc.cleanup	See the description of this argument in mclapply.

Details

pvec parallelizes FUN(x, ...) where FUN is a function that returns a vector of the same length as x. FUN must also be pure (i.e., without side-effects) since side-effects are not collected from the parallel processes. The vector is split into nearly identically sized subvectors on which FUN is run. Although it is in principle possible to use functions that are not necessarily maps, the interpretation would be case-specific as the splitting is in theory arbitrary (a warning is given in such cases).

The major difference between pvec and mclapply is that mclapply will run FUN on each element separately whereas pvec assumes that c(FUN(x[1]), FUN(x[2])) is equivalent to FUN(x[1:2]) and thus will split into as many calls to FUN as there are cores (or elements, if fewer), each handling a subset vector. This makes it more efficient than mclapply but requires the above assumption on FUN.

If mc.cores == 1 this evaluates FUN(v, ...) in the current process.

Value

The result of the computation – in a successful case it should be of the same length as v. If an error occurred or the function was not a map the result may be shorter or longer, and a warning is given.

Note

Due to the nature of the parallelization, error handling does not follow the usual rules since errors will be returned as strings and results from killed child processes will show up simply as non-existent data. Therefore it is the responsibility of the user to check the length of the result to make sure it is of the correct size. pvec raises a warning if that is the case since it does not know whether such an outcome is intentional or not.

See mcfork for the inadvisability of using this with GUI front-ends and multi-threaded libraries.

Author(s)

Simon Urbanek and R Core.

Derived from the **multicore** package formerly on CRAN.

See Also

mcparallel, mclapply, parLapply, clusterMap.

Examples

```
x <- pvec(1:1000, sqrt)
stopifnot(all(x == sqrt(1:1000)))


# One use is to convert date strings to unix time in large datasets
# as that is a relatively slow operation.
# So let's get some random dates first
# (A small test only with 2 cores: set options("mc.cores")
# and increase N for a larger-scale test.)
N <- 1e5
dates <- sprintf('%04d-%02d-%02d', as.integer(2000+rnorm(N)),
                 as.integer(runif(N, 1, 12)), as.integer(runif(N, 1, 28)))

system.time(a <- as.POSIXct(dates))

# But specifying the format is faster
system.time(a <- as.POSIXct(dates, format = "%Y-%m-%d"))

# pvec ought to be faster, but system overhead can be high
system.time(b <- pvec(dates, as.POSIXct, format = "%Y-%m-%d"))
stopifnot(all(a == b))

# using mclapply for this would much slower because each value
# will require a separate call to as.POSIXct()
# as lapply(dates, as.POSIXct) does
system.time(c <- unlist(mclapply(dates, as.POSIXct,  format = "%Y-%m-%d")))
stopifnot(all(a == c))
```

RNGstreams *Implementation of Pierre L'Ecuyer's RngStreams*

Description

This is an R re-implementation of Pierre L'Ecuyer's 'RngStreams' multiple streams of pseudo-random numbers.

Usage

```
nextRNGStream(seed)
nextRNGSubStream(seed)

clusterSetRNGStream(cl = NULL, iseed)
mc.reset.stream()
```

Arguments

seed	An integer vector of length 7 as given by .Random.seed when the '"L'Ecuyer-CMRG"' RNG is in use. See RNG for the valid values.
cl	A cluster from this package or package **snow**, or (if NULL) the registered cluster.
iseed	An integer to be supplied to set.seed, or NULL not to set reproducible seeds.

Details

The 'RngStreams' interface works with (potentially) multiple streams of pseudo-random numbers: this is particularly suitable for working with parallel computations since each task can be assigned a separate RNG stream.

This uses as its underlying generator RNGkind("L'Ecuyer-CMRG"), of L'Ecuyer (1999), which has a seed vector of 6 (signed) integers and a period of around 2^{191}. Each 'stream' is a subsequence of the period of length 2^{127} which is in turn divided into 'substreams' of length 2^{76}.

The idea of L'Ecuyer *et al* (2002) is to use a separate stream for each of the parallel computations (which ensures that the random numbers generated never get into to sync) and the parallel computations can themselves use substreams if required. The original interface stores the original seed of the first stream, the original seed of the current stream and the current seed: this could be implemented in R, but it is as easy to work by saving the relevant values of .Random.seed: see the examples.

clusterSetRNGStream selects the "L'Ecuyer-CMRG" RNG and then distributes streams to the members of a cluster, optionally setting the seed of the streams by set.seed(iseed) (otherwise they are set from the current seed of the master process: after selecting the L'Ecuyer generator).

When not on Windows, Calling mc.reset.stream() after setting the L'Ecuyer random number generator and seed makes runs from mcparallel(mc.set.seed = TRUE) reproducible. This is done internally in mclapply and pvec. (Note that it does not set the seed in the master process, so does not affect the fallback-to-serial versions of these functions.)

Value

For nextRNGStream and nextRNGSubStream, a value which can be assigned to .Random.seed.

Note

Interfaces to L'Ecuyer's C code are available in CRAN packages **rlecuyer** and **rstream**.

Author(s)

Brian Ripley

References

L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, **47**, 159–164. doi:10.1287/opre.47.1.159.

L'Ecuyer, P., Simard, R., Chen, E. J. and Kelton, W. D. (2002). An object-oriented random-number package with many long streams and substreams. *Operations Research*, **50**, 1073–1075. doi:10.1287/opre.50.6.1073.358.

See Also

RNG for fuller details of R's built-in random number generators.

The vignette for package **parallel**.

Examples

```
RNGkind("L'Ecuyer-CMRG")
set.seed(123)
(s <- .Random.seed)
## do some work involving random numbers.
nextRNGStream(s)
nextRNGSubStream(s)
```

---

splitIndices *Divide Tasks for Distribution in a Cluster*

Description

This divides up `1:nx` into `ncl` lists of approximately equal size, as a way to allocate tasks to nodes in a cluster.

It is mainly for internal use, but some package authors have found it useful.

Usage

```
splitIndices(nx, ncl)
```

Arguments

`nx`	Number of tasks.
`ncl`	Number of cluster nodes.

Value

A list of length `ncl`, each element being an integer vector.

Examples

```
splitIndices(20, 3)
```

Chapter 9

The splines package

---

splines-package *Regression Spline Functions and Classes*

Description

Regression spline functions and classes.

Details

This package provides functions for working with regression splines using the B-spline basis, bs, and the natural cubic spline basis, ns.

For a complete list of functions, use `library(help = "splines")`.

Author(s)

Douglas M. Bates <bates@stat.wisc.edu> and William N. Venables <Bill.Venables@csiro.au>

Maintainer: R Core Team <R-core@r-project.org>

---

asVector *Coerce an Object to a Vector*

Description

This is a generic function. Methods for this function coerce objects of given classes to vectors.

Usage

```
asVector(object)
```

Arguments

object	An object.

Details

Methods for vector coercion in new classes must be created for the asVector generic instead of as.vector. The as.vector function is internal and not easily extended. Currently the only class with an asVector method is the xyVector class.

Value

a vector

Author(s)

Douglas Bates and Bill Venables

See Also

xyVector

Examples

```
require(stats)
ispl <- interpSpline( weight ~ height,  women )
pred <- predict(ispl)
class(pred)
utils::str(pred)
asVector(pred)
```

---

backSpline *Monotone Inverse Spline*

Description

Create a monotone inverse of a monotone natural spline.

Usage

```
backSpline(object)
```

Arguments

object	an object that inherits from class nbSpline or npolySpline. That is, the object must represent a natural interpolation spline but it can be either in the B-spline representation or the piecewise polynomial one. The spline is checked to see if it represents a monotone function.

Value

An object of class `polySpline` that contains the piecewise polynomial representation of a function that has the appropriate values and derivatives at the knot positions to be an inverse of the spline represented by `object`. Technically this object is not a spline because the second derivative is not constrained to be continuous at the knot positions. However, it is often a much better approximation to the inverse than fitting an interpolation spline to the y/x pairs.

Author(s)

Douglas Bates and Bill Venables

See Also

`interpSpline`

Examples

```
require(graphics)
ispl <- interpSpline( women$height, women$weight )
bspl <- backSpline( ispl )
plot( bspl )                  # plots over the range of the knots
points( women$weight, women$height )
```

`bs` *B-Spline Basis for Polynomial Splines*

Description

Generate the B-spline basis matrix for a polynomial spline.

Usage

```
bs(x, df = NULL, knots = NULL, degree = 3, intercept = FALSE,
   Boundary.knots = range(x), warn.outside = TRUE)
```

Arguments

`x`	the predictor variable. Missing values are allowed.
`df`	degrees of freedom; one can specify `df` rather than `knots`; `bs()` then chooses `df-degree` (minus one if there is an intercept) knots at suitable quantiles of `x` (which will ignore missing values). The default, `NULL`, takes the number of inner knots as `length(knots)`. If that is zero as per default, that corresponds to `df = degree - intercept`.
`knots`	the *internal* breakpoints that define the spline. The default is `NULL`, which results in a basis for ordinary polynomial regression. Typical values are the mean or median for one knot, quantiles for more knots. See also `Boundary.knots`.
`degree`	degree of the piecewise polynomial—default is `3` for cubic splines.

intercept	if TRUE, an intercept is included in the basis; default is FALSE.
Boundary.knots	boundary points at which to anchor the B-spline basis (default the range of the non-NA data). If both knots and Boundary.knots are supplied, the basis parameters do not depend on x. Data can extend beyond Boundary.knots.
warn.outside	logical indicating if a warning should be signalled in case some x values are outside the boundary knots.

Details

bs is based on the function splineDesign. It generates a basis matrix for representing the family of piecewise polynomials with the specified interior knots and degree, evaluated at the values of x. A primary use is in modeling formulas to directly specify a piecewise polynomial term in a model.

When Boundary.knots are set *inside* range(x), bs() now uses a 'pivot' inside the respective boundary knot which is important for derivative evaluation. In R versions $\leq$ 3.2.2, the boundary knot itself had been used as pivot, which lead to somewhat wrong extrapolations.

Value

A matrix of dimension c(length(x), df), where either df was supplied or if knots were supplied, df = length(knots) + degree plus one if there is an intercept. Attributes are returned that correspond to the arguments to bs, and explicitly give the knots, Boundary.knots etc for use by predict.bs().

Author(s)

Douglas Bates and Bill Venables. Tweaks by R Core, and a patch fixing extrapolation "outside" Boundary.knots by Trevor Hastie.

References

Hastie, T. J. (1992) Generalized additive models. Chapter 7 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

ns, poly, smooth.spline, predict.bs, SafePrediction

Examples

```
require(stats); require(graphics)
bs(women$height, df = 5)
summary(fm1 <- lm(weight ~ bs(height, df = 5), data = women))

## example of safe prediction
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, length.out = 200)
lines(ht, predict(fm1, data.frame(height = ht)))
```

`interpSpline` *Create an Interpolation Spline*

Description

Create an interpolation spline, either from `x` and `y` vectors (`default` method), or from a `formula` / `data.frame` combination (`formula` method).

Usage

```
interpSpline(obj1, obj2, bSpline = FALSE, period = NULL,
             ord = 4L,
             na.action = na.fail, sparse = FALSE)
```

Arguments

`obj1`	either a numeric vector of `x` values or a formula.
`obj2`	if `obj1` is numeric this should be a numeric vector of the same length. If `obj1` is a formula this can be an optional data frame in which to evaluate the names in the formula.
`bSpline`	if `TRUE` the b-spline representation is returned, otherwise the piecewise polynomial representation is returned. Defaults to `FALSE`.
`period`	an optional positive numeric value giving a period for a periodic interpolation spline.
`ord`	an integer specifying the spline *order*, the number of coefficients per interval. $ord = d + 1$ where d is the *degree* polynomial degree. Currently, only cubic splines (`ord = 4`) are implemented.
`na.action`	a optional function which indicates what should happen when the data contain `NA`s. The default action (`na.omit`) is to omit any incomplete observations. The alternative action `na.fail` causes `interpSpline` to print an error message and terminate if there are any incomplete observations.
`sparse`	logical passed to the underlying `splineDesign`. If true, saves memory and is faster when there are more than a few hundred points.

Value

An object that inherits from (S3) class `spline`. The object can be in the B-spline representation, in which case it will be of class `nbSpline` for natural B-spline, or in the piecewise polynomial representation, in which case it will be of class `npolySpline`.

Author(s)

Douglas Bates and Bill Venables

See Also

splineKnots, splineOrder, periodicSpline.

Examples

```
require(graphics); require(stats)
ispl <- interpSpline( women$height, women$weight )
ispl2 <- interpSpline( weight ~ height,  women )
# ispl and ispl2 should be the same
plot( predict( ispl, seq( 55, 75, length.out = 51 ) ), type = "l" )
points( women$height, women$weight )
plot( ispl )    # plots over the range of the knots
points( women$height, women$weight )
splineKnots( ispl )
```

---

ns *Generate a Basis Matrix for Natural Cubic Splines*

Description

Generate the B-spline basis matrix for a natural cubic spline.

Usage

```
ns(x, df = NULL, knots = NULL, intercept = FALSE,
   Boundary.knots = range(x))
```

Arguments

x	the predictor variable. Missing values are allowed.
df	degrees of freedom. One can supply df rather than knots; ns() then chooses df - 1 - intercept knots at suitably chosen quantiles of x (which will ignore missing values). The default, df = NULL, sets the number of inner knots as length(knots).
knots	breakpoints that define the spline. The default is no knots; together with the natural boundary conditions this results in a basis for linear regression on x. Typical values are the mean or median for one knot, quantiles for more knots. See also Boundary.knots.
intercept	if TRUE, an intercept is included in the basis; default is FALSE.
Boundary.knots	boundary points at which to impose the natural boundary conditions and anchor the B-spline basis (default the range of the data). If both knots and Boundary.knots are supplied, the basis parameters do not depend on x. Data can extend beyond Boundary.knots

Details

ns is based on the function splineDesign. It generates a basis matrix for representing the family of piecewise-cubic splines with the specified sequence of interior knots, and the natural boundary conditions. These enforce the constraint that the function is linear beyond the boundary knots, which can either be supplied or default to the extremes of the data.

A primary use is in modeling formula to directly specify a natural spline term in a model: see the examples.

Value

A matrix of dimension length(x) * df where either df was supplied or if knots were supplied, df = length(knots) + 1 + intercept. Attributes are returned that correspond to the arguments to ns, and explicitly give the knots, Boundary.knots etc for use by predict.ns().

References

Hastie, T. J. (1992) Generalized additive models. Chapter 7 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

bs, predict.ns, SafePrediction

Examples

```
require(stats); require(graphics)
ns(women$height, df = 5)
summary(fm1 <- lm(weight ~ ns(height, df = 5), data = women))

## To see what knots were selected
attr(terms(fm1), "predvars")

## example of safe prediction
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, length.out = 200) ; nD <- data.frame(height = ht)
lines(ht, p1 <- predict(fm1, nD))
stopifnot(all.equal(p1, predict(update(fm1, . ~
                            splines::ns(height, df=5)), nD)))
          # not true in R < 3.5.0
```

---

periodicSpline *Create a Periodic Interpolation Spline*

Description

Create a periodic interpolation spline, either from x and y vectors, or from a formula/data.frame combination.

Usage

```
periodicSpline(obj1, obj2, knots, period = 2*pi, ord = 4L)
```

Arguments

obj1	either a numeric vector of x values or a formula.
obj2	if obj1 is numeric this should be a numeric vector of the same length. If obj1 is a formula this can be an optional data frame in which to evaluate the names in the formula.
knots	optional numeric vector of knot positions.
period	positive numeric value giving the period for the periodic spline. Defaults to 2 * pi.
ord	integer giving the order of the spline, at least 2. Defaults to 4. See splineOrder for a definition of the order of a spline.

Value

An object that inherits from class spline. The object can be in the B-spline representation, in which case it will be a pbSpline object, or in the piecewise polynomial representation (a ppolySpline object).

Author(s)

Douglas Bates and Bill Venables

See Also

splineKnots, interpSpline

Examples

```
require(graphics); require(stats)
xx <- seq( -pi, pi, length.out = 16 )[-1]
yy <- sin( xx )
frm <- data.frame( xx, yy )
pispl <- periodicSpline( xx, yy, period = 2 * pi)
pispl
pispl2 <- periodicSpline( yy ~ xx, frm, period = 2 * pi )
stopifnot(all.equal(pispl, pispl2))  # pispl and pispl2 are the same

plot( pispl )          # displays over one period
points( yy ~ xx, col = "brown")
plot( predict( pispl, seq(-3*pi, 3*pi, length.out = 101) ), type = "l" )
```

polySpline *Piecewise Polynomial Spline Representation*

Description

Create the piecewise polynomial representation of a spline object.

Usage

```
polySpline(object, ...)
as.polySpline(object, ...)
```

Arguments

object	An object that inherits from class spline.
...	Optional additional arguments. At present no additional arguments are used.

Value

An object that inherits from class polySpline. This is the piecewise polynomial representation of a univariate spline function. It is defined by a set of distinct numeric values called knots. The spline function is a polynomial function between each successive pair of knots. At each interior knot the polynomial segments on each side are constrained to have the same value of the function and some of its derivatives.

Author(s)

Douglas Bates and Bill Venables

See Also

interpSpline, periodicSpline, splineKnots, splineOrder

Examples

```
require(graphics)
ispl <- polySpline(interpSpline( weight ~ height,  women, bSpline = TRUE))

print( ispl )   # print the piecewise polynomial representation

plot( ispl )    # plots over the range of the knots
points( women$height, women$weight )
```

predict.bs *Evaluate a Spline Basis*

Description

Evaluate a predefined spline basis at given values.

Usage

```
## S3 method for class 'bs'
predict(object, newx, ...)

## S3 method for class 'ns'
predict(object, newx, ...)
```

Arguments

object	the result of a call to bs or ns having attributes describing knots, degree, etc.
newx	the x values at which evaluations are required.
...	Optional additional arguments. At present no additional arguments are used.

Value

An object just like object, except evaluated at the new values of x.

These are methods for the generic function predict for objects inheriting from classes "bs" or "ns". See predict for the general behavior of this function.

See Also

bs, ns, poly.

Examples

```
require(stats)
basis <- ns(women$height, df = 5)
newX <- seq(58, 72, length.out = 51)
# evaluate the basis at the new data
predict(basis, newX)
```

predict.bSpline *Evaluate a Spline at New Values of x*

Description

The predict methods for the classes that inherit from the virtual classes bSpline and polySpline are used to evaluate the spline or its derivatives. The plot method for a spline object first evaluates predict with the x argument missing, then plots the resulting xyVector with type = "l".

Usage

```
## S3 method for class 'bSpline'
predict(object, x, nseg = 50, deriv = 0, ...)
## S3 method for class 'nbSpline'
predict(object, x, nseg = 50, deriv = 0, ...)
## S3 method for class 'pbSpline'
predict(object, x, nseg = 50, deriv = 0, ...)
## S3 method for class 'npolySpline'
predict(object, x, nseg = 50, deriv = 0, ...)
## S3 method for class 'ppolySpline'
predict(object, x, nseg = 50, deriv = 0, ...)
```

Arguments

object	An object that inherits from the bSpline or the polySpline class.
x	A numeric vector of x values at which to evaluate the spline. If this argument is missing a suitable set of x values is generated as a sequence of nseq segments spanning the range of the knots.
nseg	A positive integer giving the number of segments in a set of equally-spaced x values spanning the range of the knots in object. This value is only used if x is missing.
deriv	An integer between 0 and splineOrder(object) - 1 specifying the derivative to evaluate.
...	further arguments passed to or from other methods.

Value

an xyVector with components

x	the supplied or inferred numeric vector of x values
y	the value of the spline (or its deriv'th derivative) at the x vector

Author(s)

Douglas Bates and Bill Venables

See Also

xyVector, interpSpline, periodicSpline

Examples

```
require(graphics); require(stats)
ispl <- interpSpline( weight ~ height,  women )
opar <- par(mfrow = c(2, 2), las = 1)
plot(predict(ispl, nseg = 201),     # plots over the range of the knots
     main = "Original data with interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
points(women$height, women$weight, col = 4)
plot(predict(ispl, nseg = 201, deriv = 1),
     main = "First derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
plot(predict(ispl, nseg = 201, deriv = 2),
     main = "Second derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
plot(predict(ispl, nseg = 401, deriv = 3),
     main = "Third derivative of interpolating spline", type = "l",
     xlab = "height", ylab = "weight")
par(opar)
```

---

splineDesign *Design Matrix for B-splines*

Description

Evaluate the design matrix for the B-splines defined by knots at the values in x.

Usage

```
splineDesign(knots, x, ord = 4, derivs, outer.ok = FALSE,
             sparse = FALSE)
spline.des  (knots, x, ord = 4, derivs, outer.ok = FALSE,
             sparse = FALSE)
```

Arguments

knots	a numeric vector of knot positions (which will be sorted increasingly if needed).
x	a numeric vector of values at which to evaluate the B-spline functions or derivatives. Unless outer.ok is true, the values in x must be between the "inner" knots knots[ord] and knots[length(knots) - (ord-1)].
ord	a positive integer giving the order of the spline function. This is the number of coefficients in each piecewise polynomial segment, thus a cubic spline has order 4. Defaults to 4.

derivs	an integer vector with values between 0 and ord - 1, conceptually recycled to the length of x. The derivative of the given order is evaluated at the x positions. Defaults to zero (or a vector of zeroes of the same length as x).
outer.ok	logical indicating if x should be allowed outside the *inner* knots, see the x argument.
sparse	logical indicating if the result should inherit from class "sparseMatrix" (from package **Matrix**).

Value

A matrix with length(x) rows and length(knots) - ord columns. The i-th row of the matrix contains the coefficients of the B-splines (or the indicated derivative of the B-splines) defined by the knot vector and evaluated at the i-th value of x. Each B-spline is defined by a set of ord successive knots so the total number of B-splines is length(knots) - ord.

Note

The older spline.des function takes the same arguments but returns a list with several components including knots, ord, derivs, and design. The design component is the same as the value of the splineDesign function.

Author(s)

Douglas Bates and Bill Venables

Examples

```
require(graphics)
splineDesign(knots = 1:10, x = 4:7)
splineDesign(knots = 1:10, x = 4:7, derivs = 1)
## visualize band structure
Matrix::drop0(zapsmall(6*splineDesign(knots = 1:40, x = 4:37, sparse = TRUE)))

knots <- c(1,1.8,3:5,6.5,7,8.1,9.2,10)  # 10 => 10-4 = 6 Basis splines
x <- seq(min(knots)-1, max(knots)+1, length.out = 501)
bb <- splineDesign(knots, x = x, outer.ok = TRUE)

plot(range(x), c(0,1), type = "n", xlab = "x", ylab = "",
     main =  "B-splines - sum to 1 inside inner knots")
mtext(expression(B[j](x) *"  and "* sum(B[j](x), j == 1, 6)), adj = 0)
abline(v = knots, lty = 3, col = "light gray")
abline(v = knots[c(4,length(knots)-3)], lty = 3, col = "gray10")
lines(x, rowSums(bb), col = "gray", lwd = 2)
matlines(x, bb, ylim = c(0,1), lty = 1)
```

splineKnots *Knot Vector from a Spline*

Description

Return the knot vector corresponding to a spline object.

Usage

```
splineKnots(object)
```

Arguments

object	an object that inherits from class "spline".

Value

A non-decreasing numeric vector of knot positions.

Author(s)

Douglas Bates and Bill Venables

Examples

```
ispl <- interpSpline( weight ~ height, women )
splineKnots( ispl )
```

splineOrder *Determine the Order of a Spline*

Description

Return the order of a spline object.

Usage

```
splineOrder(object)
```

Arguments

object	An object that inherits from class "spline".

Details

The order of a spline is the number of coefficients in each piece of the piecewise polynomial representation. Thus a cubic spline has order 4.

Value

A positive integer.

Author(s)

Douglas Bates and Bill Venables

See Also

splineKnots, interpSpline, periodicSpline

Examples

```
splineOrder( interpSpline( weight ~ height, women ) )
```

---

xyVector *Construct an* xyVector *Object*

Description

Create an object to represent a set of x-y pairs. The resulting object can be treated as a matrix or as a data frame or as a vector. When treated as a vector it reduces to the y component only.

The result of functions such as `predict.spline` is returned as an `xyVector` object so the x-values used to generate the y-positions are retained, say for purposes of generating plots.

Usage

```
xyVector(x, y)
```

Arguments

x	a numeric vector
y	a numeric vector of the same length as x

Value

An object of class `xyVector` with components

x	a numeric vector
y	a numeric vector of the same length as x

Author(s)

Douglas Bates and Bill Venables

Examples

```
require(stats); require(graphics)
ispl <- interpSpline( weight ~ height, women )
weights <- predict( ispl, seq( 55, 75, length.out = 51 ))
class( weights )
plot( weights, type = "l", xlab = "height", ylab = "weight" )
points( women$height, women$weight )
weights
```

Chapter 10

The `stats` package

`stats-package` *The R Stats Package*

Description

R statistical functions

Details

This package contains functions for statistical calculations and random number generation.

For a complete list of functions, use `library(help = "stats")`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

`.checkMFClasses` *Functions to Check the Type of Variables passed to Model Frames*

Description

`.checkMFClasses` checks if the variables used in a predict method agree in type with those used for fitting.

`.MFclass` categorizes variables for this purpose.

`.getXlevels()` extracts factor levels from `factor` or `character` variables.

Usage

```
.checkMFClasses(cl, m, ordNotOK = FALSE)
.MFclass(x)
.getXlevels(Terms, m)
```

Arguments

cl	a character vector of class descriptions to match.
m	a model frame (model.frame() result).
x	any R object.
ordNotOK	logical: are ordered factors different?
Terms	a terms object (terms.object).

Details

For applications involving model.matrix() such as linear models we do not need to differentiate between *ordered* factors and factors as although these affect the coding, the coding used in the fit is already recorded and imposed during prediction. However, other applications may treat ordered factors differently: rpart does, for example.

Value

.checkMFClasses() checks and either signals an error calling stop() or returns NULL invisibly.

.MFclass() returns a character string, one of "logical", "ordered", "factor", "numeric", "nmatrix.*" (a numeric matrix with a number of columns appended) or "other".

.getXlevels returns a named list of character vectors, possibly empty, or NULL.

Examples

```
sapply(warpbreaks, .MFclass) # "numeric" plus 2 x "factor"
sapply(iris,       .MFclass) # 4 x "numeric" plus "factor"

mf <- model.frame(Sepal.Width ~ Species,      iris)
mc <- model.frame(Sepal.Width ~ Sepal.Length, iris)

.checkMFClasses("numeric", mc) # nothing else
.checkMFClasses(c("numeric", "factor"), mf)

## simple .getXlevels() cases :
(xl <- .getXlevels(terms(mf), mf)) # a list with one entry " $ Species" with 3 levels:
stopifnot(exprs = {
  identical(xl$Species, levels(iris$Species))
  identical(.getXlevels(terms(mc), mc), xl[0]) # a empty named list, as no factors
  is.null(.getXlevels(terms(x~x), list(x=1)))
})
```

acf *Auto- and Cross- Covariance and -Correlation Function Estimation*

Description

The function acf computes (and by default plots) estimates of the autocovariance or autocorrelation function. Function pacf is the function used for the partial autocorrelations. Function ccf computes the cross-correlation or cross-covariance of two univariate series.

Usage

```
acf(x, lag.max = NULL,
    type = c("correlation", "covariance", "partial"),
    plot = TRUE, na.action = na.fail, demean = TRUE, ...)

pacf(x, lag.max, plot, na.action, ...)

## Default S3 method:
pacf(x, lag.max = NULL, plot = TRUE, na.action = na.fail,
    ...)

ccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
    plot = TRUE, na.action = na.fail, ...)

## S3 method for class 'acf'
x[i, j]
```

Arguments

x, y	a univariate or multivariate (not ccf) numeric time series object or a numeric vector or matrix, or an "acf" object.
lag.max	maximum lag at which to calculate the acf. Default is $10\log_{10}(N/m)$ where N is the number of observations and m the number of series. Will be automatically limited to one less than the number of observations in the series.
type	character string giving the type of acf to be computed. Allowed values are "correlation" (the default), "covariance" or "partial". Will be partially matched.
plot	logical. If TRUE (the default) the acf is plotted.
na.action	function to be called to handle missing values. na.pass can be used.
demean	logical. Should the covariances be about the sample means?
...	further arguments to be passed to plot.acf.
i	a set of lags (time differences) to retain.
j	a set of series (names or numbers) to retain.

Details

For type = "correlation" and "covariance", the estimates are based on the sample covariance. (The lag 0 autocorrelation is fixed at 1 by convention.)

By default, no missing values are allowed. If the na.action function passes through missing values (as na.pass does), the covariances are computed from the complete cases. This means that the estimate computed may well not be a valid autocorrelation sequence, and may contain missing values. Missing values are not allowed when computing the PACF of a multivariate time series.

The partial correlation coefficient is estimated by fitting autoregressive models of successively higher orders up to lag.max.

The generic function plot has a method for objects of class "acf".

The lag is returned and plotted in units of time, and not numbers of observations.

There are print and subsetting methods for objects of class "acf".

Value

An object of class "acf", which is a list with the following elements:

lag	A three dimensional array containing the lags at which the acf is estimated.
acf	An array with the same dimensions as lag containing the estimated acf.
type	The type of correlation (same as the type argument).
n.used	The number of observations in the time series.
series	The name of the series x.
snames	The series names for a multivariate time series.

The lag k value returned by ccf(x, y) estimates the correlation between x[t+k] and y[t].

The result is returned invisibly if plot is TRUE.

Author(s)

Original: Paul Gilbert, Martyn Plummer. Extensive modifications and univariate case of pacf by B. D. Ripley.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth Edition. Springer-Verlag.

(This contains the exact definitions used.)

See Also

plot.acf, ARMAacf for the exact autocorrelations of a given ARMA process.

Examples

```
require(graphics)

## Examples from Venables & Ripley
acf(lh)
acf(lh, type = "covariance")
pacf(lh)

acf(ldeaths)
acf(ldeaths, ci.type = "ma")
acf(ts.union(mdeaths, fdeaths))
ccf(mdeaths, fdeaths, ylab = "cross-correlation")
# (just the cross-correlations)

presidents # contains missing values
acf(presidents, na.action = na.pass)
pacf(presidents, na.action = na.pass)
```

`acf2AR` *Compute an AR Process Exactly Fitting an ACF*

Description

Compute an AR process exactly fitting an autocorrelation function.

Usage

```
acf2AR(acf)
```

Arguments

`acf`	An autocorrelation or autocovariance sequence.

Value

A matrix, with one row for the computed AR(p) coefficients for `1 <= p <= length(acf)`.

See Also

`ARMAacf`, `ar.yw` which does this from an empirical ACF.

Examples

```
(Acf <- ARMAacf(c(0.6, 0.3, -0.2)))
acf2AR(Acf)
```

add1 *Add or Drop All Possible Single Terms to a Model*

Description

Compute all the single terms in the scope argument that can be added to or dropped from the model, fit those models and compute a table of the changes in fit.

Usage

```
add1(object, scope, ...)

## Default S3 method:
add1(object, scope, scale = 0, test = c("none", "Chisq"),
     k = 2, trace = FALSE, ...)

## S3 method for class 'lm'
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
     x = NULL, k = 2, ...)

## S3 method for class 'glm'
add1(object, scope, scale = 0,
     test = c("none", "Rao", "LRT", "Chisq", "F"),
     x = NULL, k = 2, ...)

drop1(object, scope, ...)

## Default S3 method:
drop1(object, scope, scale = 0, test = c("none", "Chisq"),
      k = 2, trace = FALSE, ...)

## S3 method for class 'lm'
drop1(object, scope, scale = 0, all.cols = TRUE,
      test = c("none", "Chisq", "F"), k = 2, ...)

## S3 method for class 'glm'
drop1(object, scope, scale = 0,
      test = c("none", "Rao", "LRT", "Chisq", "F"),
      k = 2, ...)
```

Arguments

object	a fitted model object.
scope	a formula giving the terms to be considered for adding or dropping.
scale	an estimate of the residual mean square to be used in computing C_p. Ignored if 0 or NULL.

test	should the results include a test statistic relative to the original model? The F test is only appropriate for lm and aov models or perhaps for glm fits with estimated dispersion. The χ^2 test can be an exact test (lm models with known scale) or a likelihood-ratio test or a test of the reduction in scaled deviance depending on the method. For glm fits, you can also choose "LRT" and "Rao" for likelihood ratio tests and Rao's efficient score test. The former is synonymous with "Chisq" (although both have an asymptotic chi-square distribution). Values can be abbreviated.
k	the penalty constant in AIC / C_p.
trace	if TRUE, print out progress reports.
x	a model matrix containing columns for the fitted model and all terms in the upper scope. Useful if add1 is to be called repeatedly. **Warning:** no checks are done on its validity.
all.cols	(Provided for compatibility with S.) Logical to specify whether all columns of the design matrix should be used. If FALSE then non-estimable columns are dropped, but the result is not usually statistically meaningful.
...	further arguments passed to or from other methods.

Details

For drop1 methods, a missing scope is taken to be all terms in the model. The hierarchy is respected when considering terms to be added or dropped: all main effects contained in a second-order interaction must remain, and so on.

In a scope formula . means 'what is already there'.

The methods for lm and glm are more efficient in that they do not recompute the model matrix and call the fit methods directly.

The default output table gives AIC, defined as minus twice log likelihood plus $2p$ where p is the rank of the model (the number of effective parameters). This is only defined up to an additive constant (like log-likelihoods). For linear Gaussian models with fixed scale, the constant is chosen to give Mallows' C_p, $RSS/scale + 2p - n$. Where C_p is used, the column is labelled as Cp rather than AIC.

The F tests for the "glm" methods are based on analysis of deviance tests, so if the dispersion is estimated it is based on the residual deviance, unlike the F tests of anova.glm.

Value

An object of class "anova" summarizing the differences in fit between the models.

Warning

The model fitting must apply the models to the same dataset. Most methods will attempt to use a subset of the data with no missing values for any of the variables if na.action = na.omit, but this may give biased results. Only use these functions with data containing missing values with great care.

The default methods make calls to the function nobs to check that the number of observations involved in the fitting process remained unchanged.

Note

These are not fully equivalent to the functions in S. There is no keep argument, and the methods used are not quite so computationally efficient.

Their authors' definitions of Mallows' C_p and Akaike's AIC are used, not those of the authors of the models chapter of S.

Author(s)

The design was inspired by the S functions of the same names described in Chambers (1992).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

step, aov, lm, extractAIC, anova

Examples

```
require(graphics); require(utils)
## following example(swiss)
lm1 <- lm(Fertility ~ ., data = swiss)
add1(lm1, ~ I(Education^2) + .^2)
drop1(lm1, test = "F")  # So called 'type II' anova

## following example(glm)

drop1(glm.D93, test = "Chisq")
drop1(glm.D93, test = "F")
add1(glm.D93, scope = ~outcome*treatment, test = "Rao") ## Pearson Chi-square
```

---

addmargins *Puts Arbitrary Margins on Multidimensional Tables or Arrays*

Description

For a given table one can specify which of the classifying factors to expand by one or more levels to hold margins to be calculated. One may for example form sums and means over the first dimension and medians over the second. The resulting table will then have two extra levels for the first dimension and one extra level for the second. The default is to sum over all margins in the table. Other possibilities may give results that depend on the order in which the margins are computed. This is flagged in the printed output from the function.

Usage

```
addmargins(A, margin = seq_along(dim(A)), FUN = sum, quiet = FALSE)
```

Arguments

A	table or array. The function uses the presence of the "dim" and "dimnames" attributes of A.
margin	vector of dimensions over which to form margins. Margins are formed in the order in which dimensions are specified in margin.
FUN	list of the same length as margin, each element of the list being either a function or a list of functions. In the length-1 case, can be a function instead of a list of one. Names of the list elements will appear as levels in dimnames of the result. Unnamed list elements will have names constructed: the name of a function or a constructed name based on the position in the table.
quiet	logical which suppresses the message telling the order in which the margins were computed.

Details

If the functions used to form margins are not commutative, the result depends on the order in which margins are computed. Annotation of margins is done via naming the FUN list.

Value

A table or array with the same number of dimensions as A, but with extra levels of the dimensions mentioned in margin. The number of levels added to each dimension is the length of the entries in FUN. A message with the order of computation of margins is printed.

Author(s)

Bendix Carstensen, Steno Diabetes Center & Department of Biostatistics, University of Copenhagen, https://BendixCarstensen.com, autumn 2003. Margin naming enhanced by Duncan Murdoch.

See Also

table, ftable, margin.table.

Examples

```
Aye <- sample(c("Yes", "Si", "Oui"), 177, replace = TRUE)
Bee <- sample(c("Hum", "Buzz"), 177, replace = TRUE)
Sea <- sample(c("White", "Black", "Red", "Dead"), 177, replace = TRUE)
(A <- table(Aye, Bee, Sea))
(aA <- addmargins(A))

ftable(A)
ftable(aA)

# Non-commutative functions - note differences between resulting tables:
ftable( addmargins(A, c(3, 1),
                   FUN = list(list(Min = min, Max = max),
                              Sum = sum)))
```

```
ftable( addmargins(A, c(1, 3),
                   FUN = list(Sum = sum,
                              list(Min = min, Max = max))))

# Weird function needed to return the N when computing percentages
sqsm <- function(x) sum(x)^2/100
B <- table(Sea, Bee)
round(sweep(addmargins(B, 1, list(list(All = sum, N = sqsm))), 2,
            apply(B, 2, sum)/100, `/`), 1)
round(sweep(addmargins(B, 2, list(list(All = sum, N = sqsm))), 1,
            apply(B, 1, sum)/100, `/`), 1)

# A total over Bee requires formation of the Bee-margin first:
mB <-  addmargins(B, 2, FUN = list(list(Total = sum)))
round(ftable(sweep(addmargins(mB, 1, list(list(All = sum, N = sqsm))), 2,
                   apply(mB, 2, sum)/100, `/`)), 1)

## Zero.Printing table+margins:
set.seed(1)
x <- sample( 1:7, 20, replace = TRUE)
y <- sample( 1:7, 20, replace = TRUE)
tx <- addmargins( table(x, y) )
print(tx, zero.print = ".")
```

---

aggregate *Compute Summary Statistics of Data Subsets*

Description

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

Usage

```
aggregate(x, ...)

## Default S3 method:
aggregate(x, ...)

## S3 method for class 'data.frame'
aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)

## S3 method for class 'formula'
aggregate(x, data, FUN, ...,
          subset, na.action = na.omit)

## S3 method for class 'ts'
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

Arguments

x	an R object. For the formula method a formula, such as y ~ x or cbind(y1, y2) ~ x1 + x2, where the y variables are numeric data to be split into groups according to the grouping x variables (usually factors).
by	a list of grouping elements, each as long as the variables in the data frame x, or a formula. The elements are coerced to factors before use.
FUN	a function to compute the summary statistics which can be applied to all data subsets.
simplify	a logical indicating whether results should be simplified to a vector or matrix if possible.
drop	a logical indicating whether to drop unused combinations of grouping values. The non-default case drop=FALSE has been amended for R 3.5.0 to drop unused combinations.
data	a data frame (or list) from which the variables in the formula should be taken.
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NA values. The default is to only consider *complete cases* with respect to the given variables.
nfrequency	new number of observations per unit of time; must be a divisor of the frequency of x.
ndeltat	new fraction of the sampling period between successive observations; must be a divisor of the sampling interval of x.
ts.eps	tolerance used to decide if nfrequency is a sub-multiple of the original frequency.
...	further arguments passed to or used by methods.

Details

aggregate is a generic function with methods for data frames and time series.

The default method, aggregate.default, uses the time series method if x is a time series, and otherwise coerces x to a data frame and calls the data frame method.

aggregate.data.frame is the data frame method. If x is not a data frame, it is coerced to one, which must have a non-zero number of rows. Then, each of the variables (columns) in x is split into subsets of cases (rows) of identical combinations of the components of by, and FUN is applied to each such subset with further arguments in ... passed to it. The result is reformatted into a data frame containing the variables in by and x. The ones arising from by contain the unique combinations of grouping values used for determining the subsets, and the ones arising from x the corresponding summaries for the subset of the respective variables in x. If simplify is true, summaries are simplified to vectors or matrices if they have a common length of one or greater than one, respectively; otherwise, lists of summary results according to subsets are obtained. Rows with missing values in any of the by variables will be omitted from the result. (Note that versions of R prior to 2.11.0 required FUN to be a scalar function.)

The formula method provides a standard formula interface to aggregate.data.frame. The latter invokes the formula method if by is a formula, in which case aggregate(x, by, FUN) is the same as aggregate(by, x, FUN) for a data frame x.

`aggregate.ts` is the time series method, and requires `FUN` to be a scalar function. If `x` is not a time series, it is coerced to one. Then, the variables in `x` are split into appropriate blocks of length `frequency(x) / nfrequency`, and `FUN` is applied to each such block, with further (named) arguments in `...` passed to it. The result returned is a time series with frequency `nfrequency` holding the aggregated values. Note that this make most sense for a quarterly or yearly result when the original series covers a whole number of quarters or years: in particular aggregating a monthly series to quarters starting in February does not give a conventional quarterly series.

`FUN` is passed to `match.fun`, and hence it can be a function or a symbol or character string naming a function.

Value

For the time series method, a time series of class `"ts"` or class `c("mts", "ts")`.

For the data frame method, a data frame with columns corresponding to the grouping variables in `by` followed by aggregated columns from `x`. If the `by` has names, the non-empty times are used to label the columns in the results, with unnamed grouping variables being named `Group.`*i* for `by[[`*i*`]]`.

Warning

The first argument of the `"formula"` method was named `formula` rather than `x` prior to R 4.2.0. Portable uses should not name that argument.

Author(s)

Kurt Hornik, with contributions by Arni Magnusson.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`apply`, `lapply`, `tapply`.

Examples

```
## Compute the averages for the variables in 'state.x77', grouped
## according to the region (Northeast, South, North Central, West) that
## each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)

## Compute the averages according to region and the occurrence of more
## than 130 days of frost.
aggregate(state.x77,
          list(Region = state.region,
               Cold = state.x77[,"Frost"] > 130),
          mean)
## (Note that no state in 'South' is THAT cold.)
```

```
## example with character variables and NAs
testDF <- data.frame(v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
                     v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99) )
by1 <- c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12)
by2 <- c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA)
aggregate(x = testDF, by = list(by1, by2), FUN = "mean")

# and if you want to treat NAs as a group
fby1 <- factor(by1, exclude = "")
fby2 <- factor(by2, exclude = "")
aggregate(x = testDF, by = list(fby1, fby2), FUN = "mean")


## Formulas, one ~ one, one ~ many, many ~ one, and many ~ many:
aggregate(weight ~ feed, data = chickwts, mean)
aggregate(breaks ~ wool + tension, data = warpbreaks, mean)
aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, mean)
aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, data = esoph, sum)

## "complete cases" vs. "available cases"
colSums(is.na(airquality))  # NAs in Ozone but not Temp
## the default is to summarize *complete cases*:
aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, FUN = mean)
## to handle missing values *per variable*:
aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, FUN = mean,
          na.action = na.pass, na.rm = TRUE)

## Dot notation:
aggregate(. ~ Species, data = iris, mean)
aggregate(len ~ ., data = ToothGrowth, mean)

## Often followed by xtabs():
ag <- aggregate(len ~ ., data = ToothGrowth, mean)
xtabs(len ~ ., data = ag)

## Formula interface via 'by' (for pipe operations)
ToothGrowth |> aggregate(len ~ ., FUN = mean)

## Compute the average annual approval ratings for American presidents.
aggregate(presidents, nfrequency = 1, FUN = mean)
## Give the summer less weight.
aggregate(presidents, nfrequency = 1,
          FUN = weighted.mean, w = c(1, 1, 0.5, 1))
```

AIC *Akaike's An Information Criterion*

Description

Generic function calculating Akaike's 'An Information Criterion' for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula $-2\text{log-likelihood} + kn_{par}$, where n_{par} represents the number of parameters in the fitted model, and $k = 2$ for the usual AIC, or $k = \log(n)$ (n being the number of observations) for the so-called BIC or SBC (Schwarz's Bayesian criterion).

Usage

```
AIC(object, ..., k = 2)

BIC(object, ...)
```

Arguments

`object`	a fitted model object for which there exists a `logLik` method to extract the corresponding log-likelihood, or an object inheriting from class `logLik`.
`...`	optionally more fitted model objects.
`k`	numeric, the *penalty* per parameter to be used; the default `k = 2` is the classical AIC.

Details

When comparing models fitted by maximum likelihood to the same data, the smaller the AIC or BIC, the better the fit.

The theory of AIC requires that the log-likelihood has been maximized: whereas AIC can be computed for models not fitted by maximum likelihood, their AIC values should not be compared.

Examples of models not 'fitted to the same data' are where the response is transformed (accelerated-life models are fitted to log-times) and where contingency tables have been used to summarize data.

These are generic functions (with S4 generics defined in package **stats4**): however methods should be defined for the log-likelihood function `logLik` rather than these functions: the action of their default methods is to call `logLik` on all the supplied objects and assemble the results. Note that in several common cases `logLik` does not return the value at the MLE: see its help page.

The log-likelihood and hence the AIC/BIC is only defined up to an additive constant. Different constants have conventionally been used for different purposes and so `extractAIC` and `AIC` may give different values (and do for models of class `"lm"`: see the help for `extractAIC`). Particular care is needed when comparing fits of different classes (with, for example, a comparison of a Poisson and gamma GLM being meaningless since one has a discrete response, the other continuous).

`BIC` is defined as `AIC(object, ..., k = log(nobs(object)))`. This needs the number of observations to be known: the default method looks first for a `"nobs"` attribute on the return value from the `logLik` method, then tries the `nobs` generic, and if neither succeed returns BIC as `NA`.

Value

If just one object is provided, a numeric value with the corresponding AIC (or BIC, or ..., depending on `k`).

If multiple objects are provided, a `data.frame` with rows corresponding to the objects and columns representing the number of parameters in the model (`df`) and the AIC or BIC.

Author(s)

Originally by José Pinheiro and Douglas Bates, more recent revisions by R-core.

References

Sakamoto, Y., Ishiguro, M., and Kitagawa G. (1986). *Akaike Information Criterion Statistics*. D. Reidel Publishing Company.

See Also

`extractAIC`, `logLik`, `nobs`.

Examples

```
lm1 <- lm(Fertility ~ . , data = swiss)
AIC(lm1)
stopifnot(all.equal(AIC(lm1),
                    AIC(logLik(lm1))))
BIC(lm1)

lm2 <- update(lm1, . ~ . -Examination)
AIC(lm1, lm2)
BIC(lm1, lm2)
```

---

alias *Find Aliases (Dependencies) in a Model*

Description

Find aliases (linearly dependent terms) in a linear model specified by a formula.

Usage

```
alias(object, ...)

## S3 method for class 'formula'
alias(object, data, ...)

## S3 method for class 'lm'
alias(object, complete = TRUE, partial = FALSE,
      partial.pattern = FALSE, ...)
```

Arguments

object	A fitted model object, for example from lm or aov, or a formula for alias.formula.
data	Optionally, a data frame to search for the objects in the formula.
complete	Should information on complete aliasing be included?
partial	Should information on partial aliasing be included?
partial.pattern	Should partial aliasing be presented in a schematic way? If this is done, the results are presented in a more compact way, usually giving the deciles of the coefficients.
...	further arguments passed to or from other methods.

Details

Although the main method is for class "lm", alias is most useful for experimental designs and so is used with fits from aov. Complete aliasing refers to effects in linear models that cannot be estimated independently of the terms which occur earlier in the model and so have their coefficients omitted from the fit. Partial aliasing refers to effects that can be estimated less precisely because of correlations induced by the design.

Some parts of the "lm" method require recommended package **MASS** to be installed.

Value

A list (of class "listof") containing components

Model	Description of the model; usually the formula.
Complete	A matrix with columns corresponding to effects that are linearly dependent on the rows.
Partial	The correlations of the estimable effects, with a zero diagonal. An object of class "mtable" which has its own print method.

Note

The aliasing pattern may depend on the contrasts in use: Helmert contrasts are probably most useful.

The defaults are different from those in S.

Author(s)

The design was inspired by the S function of the same name described in Chambers et al. (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
op <- options(contrasts = c("contr.helmert", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
alias(npk.aov)
options(op)  # reset
```

---

anova *ANOVA Tables*

Description

Compute analysis of variance (or deviance) tables for one or more fitted model objects.

Usage

```
anova(object, ...)
```

Arguments

`object`	an object containing the results returned by a model fitting function (e.g., `lm` or `glm`).
`...`	additional objects of the same type.

Value

This (generic) function returns an object of class `anova`. These objects represent analysis-of-variance and analysis-of-deviance tables. When given a single argument it produces a table which tests whether the model terms are significant.

When given a sequence of objects, `anova` tests the models against one another in the order specified.

The print method for `anova` objects prints tables in a 'pretty' form.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

See Also

`coefficients`, `effects`, `fitted.values`, `residuals`, `summary`, `drop1`, `add1`.

anova.glm *Analysis of Deviance for Generalized Linear Model Fits*

Description

Compute an analysis of deviance table for one or more generalized linear model fits.

Usage

```
## S3 method for class 'glm'
anova(object, ..., dispersion = NULL, test = NULL)
```

Arguments

object, ...	objects of class glm, typically the result of a call to glm, or a list of objects for the "glmlist" method.
dispersion	the dispersion parameter for the fitting family. By default it is obtained from the object(s).
test	a character string, (partially) matching one of "Chisq", "LRT", "Rao", "F" or "Cp". See stat.anova. Or logical FALSE, which suppresses any test.

Details

Specifying a single object gives a sequential analysis of deviance table for that fit. That is, the reductions in the residual deviance as each term of the formula is added in turn are given in as the rows of a table, plus the residual deviances themselves.

If more than one object is specified, the table has a row for the residual degrees of freedom and deviance for each model. For all but the first model, the change in degrees of freedom and deviance is also given. (This only makes statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

The table will optionally contain test statistics (and P values) comparing the reduction in deviance for the row to the residuals. For models with known dispersion (e.g., binomial and Poisson fits) the chi-squared test is most appropriate, and for those with dispersion estimated by moments (e.g., gaussian, quasibinomial and quasipoisson fits) the F test is most appropriate. If anova.glm can determine which of these cases applies then by default it will use one of the above tests. If the dispersion argument is supplied, the dispersion is considered known and the chi-squared test will be used. Argument test=FALSE suppresses the test statistics and P values. Mallows' C_p statistic is the residual deviance plus twice the estimate of σ^2 times the residual degrees of freedom, which is closely related to AIC (and a multiple of it if the dispersion is known). You can also choose "LRT" and "Rao" for likelihood ratio tests and Rao's efficient score test. The former is synonymous with "Chisq" (although both have an asymptotic chi-square distribution).

The dispersion estimate will be taken from the largest model, using the value returned by summary.glm. As this will in most cases use a Chi-squared-based estimate, the F tests are not based on the residual deviance in the analysis of deviance table shown.

Value

An object of class "anova" inheriting from class "data.frame".

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of na.action = na.omit is used, and anova will detect this with an error.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

glm, anova.

drop1 for so-called 'type II' ANOVA where each term is dropped one at a time respecting their hierarchy.

Examples

```
## --- Continuing the Example from  '?glm':

anova(glm.D93, test = FALSE)
anova(glm.D93, test = "Cp")
anova(glm.D93, test = "Chisq")
glm.D93a <-
   update(glm.D93, ~treatment*outcome) # equivalent to Pearson Chi-square
anova(glm.D93, glm.D93a, test = "Rao")
```

---

anova.lm *ANOVA for Linear Model Fits*

Description

Compute an analysis of variance table for one or more linear model fits.

Usage

```
## S3 method for class 'lm'
anova(object, ...)

## S3 method for class 'lmlist'
anova(object, ..., scale = 0, test = "F")
```

Arguments

object, ...	objects of class lm, usually, a result of a call to lm.
test	a character string specifying the test statistic to be used. Can be one of "F", "Chisq" or "Cp", with partial matching allowed, or NULL for no test.
scale	numeric. An estimate of the noise variance σ^2. If zero this will be estimated from the largest model considered.

Details

Specifying a single object gives a sequential analysis of variance table for that fit. That is, the reductions in the residual sum of squares as each term of the formula is added in turn are given in as the rows of a table, plus the residual sum of squares.

The table will contain F statistics (and P values) comparing the mean square for the row to the residual mean square.

If more than one object is specified, the table has a row for the residual degrees of freedom and sum of squares for each model. For all but the first model, the change in degrees of freedom and sum of squares is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

Optionally the table can include test statistics. Normally the F statistic is most appropriate, which compares the mean square for a row to the residual sum of squares for the largest model considered. If scale is specified chi-squared tests can be used. Mallows' C_p statistic is the residual sum of squares plus twice the estimate of σ^2 times the residual degrees of freedom.

Value

An object of class "anova" inheriting from class "data.frame".

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of na.action = na.omit is used, and anova.lmlist will detect this with an error.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function lm, anova.

drop1 for so-called 'type II' ANOVA where each term is dropped one at a time respecting their hierarchy.

Examples

```
## sequential table
fit <- lm(sr ~ ., data = LifeCycleSavings)
anova(fit)

## same effect via separate models
fit0 <- lm(sr ~ 1, data = LifeCycleSavings)
fit1 <- update(fit0, . ~ . + pop15)
fit2 <- update(fit1, . ~ . + pop75)
fit3 <- update(fit2, . ~ . + dpi)
fit4 <- update(fit3, . ~ . + ddpi)
anova(fit0, fit1, fit2, fit3, fit4, test = "F")

anova(fit4, fit2, fit0, test = "F") # unconventional order
```

---

anova.mlm *Comparisons between Multivariate Linear Models*

Description

Compute a (generalized) analysis of variance table for one or more multivariate linear models.

Usage

```
## S3 method for class 'mlm'
anova(object, ...,
      test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy",
               "Spherical"),
      Sigma = diag(nrow = p), T = Thin.row(Proj(M) - Proj(X)),
      M = diag(nrow = p), X = ~0,
      idata = data.frame(index = seq_len(p)), tol = 1e-7)
```

Arguments

object	an object of class "mlm".
...	further objects of class "mlm".
test	choice of test statistic (see below). Can be abbreviated.
Sigma	(only relevant if test == "Spherical"). Covariance matrix assumed proportional to Sigma.
T	transformation matrix. By default computed from M and X.
M	formula or matrix describing the outer projection (see below).
X	formula or matrix describing the inner projection (see below).
idata	data frame describing intra-block design.
tol	tolerance to be used in deciding if the residuals are rank-deficient: see qr.

Details

The `anova.mlm` method uses either a multivariate test statistic for the summary table, or a test based on sphericity assumptions (i.e. that the covariance is proportional to a given matrix).

For the multivariate test, Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987). See `summary.manova` for further details.

For the `"Spherical"` test, proportionality is usually with the identity matrix but a different matrix can be specified using `Sigma`. Corrections for asphericity known as the Greenhouse–Geisser, respectively Huynh–Feldt, epsilons are given and adjusted F tests are performed.

It is common to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A transformation matrix `T` can be given directly or specified as the difference between two projections onto the spaces spanned by `M` and `X`, which in turn can be given as matrices or as model formulas with respect to `idata` (the tests will be invariant to parametrization of the quotient space `M/X`).

As with `anova.lm`, all test statistics use the SSD matrix from the largest model considered as the (generalized) denominator.

Contrary to other `anova` methods, the intercept is not excluded from the display in the single-model case. When contrast transformations are involved, it often makes good sense to test for a zero intercept.

Value

An object of class `"anova"` inheriting from class `"data.frame"`

Note

The Huynh–Feldt epsilon differs from that calculated by SAS (as of v. 8.2) except when the DF is equal to the number of observations minus one. This is believed to be a bug in SAS, not in R.

References

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

`summary.manova`

Examples

```
require(graphics)
utils::example(SSD) # Brings in the mlmfit and reacttime objects

mlmfit0 <- update(mlmfit, ~0)

### Traditional tests of intrasubj. contrasts
## Using MANOVA techniques on contrasts:
```

```
anova(mlmfit, mlmfit0, X = ~1)

## Assuming sphericity
anova(mlmfit, mlmfit0, X = ~1, test = "Spherical")


### tests using intra-subject 3x2 design
idata <- data.frame(deg = gl(3, 1, 6, labels = c(0, 4, 8)),
                    noise = gl(2, 3, 6, labels = c("A", "P")))

anova(mlmfit, mlmfit0, X = ~ deg + noise,
      idata = idata, test = "Spherical")
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ noise,
      idata = idata, test = "Spherical" )
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ deg,
      idata = idata, test = "Spherical" )

f <- factor(rep(1:2, 5)) # bogus, just for illustration
mlmfit2 <- update(mlmfit, ~f)
anova(mlmfit2, mlmfit, mlmfit0, X = ~1, test = "Spherical")
anova(mlmfit2, X = ~1, test = "Spherical")
# one-model form, eqiv. to previous

### There seems to be a strong interaction in these data
plot(colMeans(reacttime))
```

`ansari.test` *Ansari-Bradley Test*

Description

Performs the Ansari-Bradley two-sample test for a difference in scale parameters.

Usage

```
ansari.test(x, ...)

## Default S3 method:
ansari.test(x, y,
            alternative = c("two.sided", "less", "greater"),
            exact = NULL, conf.int = FALSE, conf.level = 0.95,
            ...)

## S3 method for class 'formula'
ansari.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	numeric vector of data values.

y	numeric vector of data values.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter.
exact	a logical indicating whether an exact p-value should be computed.
conf.int	a logical,indicating whether a confidence interval should be computed.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

Suppose that x and y are independent samples from distributions with densities $f((t-m)/s)/s$ and $f(t-m)$, respectively, where m is an unknown nuisance parameter and s, the ratio of scales, is the parameter of interest. The Ansari-Bradley test is used for testing the null that s equals 1, the two-sided alternative being that $s \neq 1$ (the distributions differ only in variance), and the one-sided alternatives being $s > 1$ (the distribution underlying x has a larger variance, "greater") or $s < 1$ ("less").

By default (if exact is not specified), an exact p-value is computed if both samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally, a nonparametric confidence interval and an estimator for s are computed. If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

Note that mid-ranks are used in the case of ties rather than average scores as employed in Hollander & Wolfe (1973). See, e.g., Hajek, Sidak and Sen (1999), pages 131ff, for more information.

Value

A list with class "htest" containing the following components:

statistic	the value of the Ansari-Bradley test statistic.
p.value	the p-value of the test.
null.value	the ratio of scales s under the null, 1.
alternative	a character string describing the alternative hypothesis.
method	the string "Ansari-Bradley test".
data.name	a character string giving the names of the data.
conf.int	a confidence interval for the scale parameter. (Only present if argument conf.int = TRUE.)
estimate	an estimate of the ratio of scales. (Only present if argument conf.int = TRUE.)

Note

To compare results of the Ansari-Bradley test to those of the F test to compare two variances (under the assumption of normality), observe that s is the ratio of scales and hence s^2 is the ratio of variances (provided they exist), whereas for the F test the ratio of variances itself is the parameter of interest. In particular, confidence intervals are for s in the Ansari-Bradley test but for s^2 in the F test.

References

David F. Bauer (1972). Constructing confidence sets using rank statistics. *Journal of the American Statistical Association*, **67**, 687–690. doi:10.1080/01621459.1972.10481279.

Jaroslav Hajek, Zbynek Sidak and Pranab K. Sen (1999). *Theory of Rank Tests*. San Diego, London: Academic Press.

Myles Hollander and Douglas A. Wolfe (1973). *Nonparametric Statistical Methods*. New York: John Wiley & Sons. Pages 83–92.

See Also

`fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `mood.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

`ansari_test` in package **coin** for exact and approximate *conditional* p-values for the Ansari-Bradley test, as well as different methods for handling ties.

Examples

```
## Hollander & Wolfe (1973, p. 86f):
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
ansari.test(ramsay, jung.parekh)

ansari.test(rnorm(10), rnorm(10, 0, 2), conf.int = TRUE)

## try more points - failed in 2.4.1
ansari.test(rnorm(100), rnorm(100, 0, 2), conf.int = TRUE)
```

---

aov *Fit an Analysis of Variance Model*

Description

Fit an analysis of variance model by a call to `lm` (for each stratum if an `Error(.)` is used).

Usage

```
aov(formula, data = NULL, projections = FALSE, qr = TRUE,
    contrasts = NULL, ...)
```

Arguments

formula	A formula specifying the model.
data	A data frame in which the variables specified in the formula will be found. If missing, the variables are searched for in the standard way.
projections	Logical flag: should the projections be returned?
qr	Logical flag: should the QR decomposition be returned?
contrasts	A list of contrasts to be used for some of the factors in the formula. These are not used for any Error term, and supplying contrasts for factors only in the Error term will give a warning.
...	Arguments to be passed to lm, such as subset or na.action. See 'Details' about weights.

Details

This provides a wrapper to lm for fitting linear models to balanced or unbalanced experimental designs.

The main difference from lm is in the way print, summary and so on handle the fit: this is expressed in the traditional language of the analysis of variance rather than that of linear models.

If the formula contains a single Error term, this is used to specify error strata, and appropriate models are fitted within each error stratum.

The formula can specify multiple responses.

Weights can be specified by a weights argument, but should not be used with an Error term, and are incompletely supported (e.g., not by model.tables).

Value

An object of class c("aov", "lm") or for multiple responses of class c("maov", "aov", "mlm", "lm") or for multiple error strata of class c("aovlist", "listof"). There are print and summary methods available for these.

Note

aov is designed for balanced designs, and the results can be hard to interpret without balance: beware that missing values in the response(s) will likely lose the balance. If there are two or more error strata, the methods used are statistically inefficient without balance, and it may be better to use lme in package **nlme**.

Balance can be checked with the replications function.

The default 'contrasts' in R are not orthogonal contrasts, and aov and its helper functions will work better with such contrasts: see the examples for how to select these.

Author(s)

The design was inspired by the S function of the same name described in Chambers et al. (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

lm, summary.aov, replications, alias, proj, model.tables, TukeyHSD

Examples

```
## From Venables and Ripley (2002) p.165.

## Set orthogonal contrasts.
op <- options(contrasts = c("contr.helmert", "contr.poly"))
( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

## to show the effects of re-ordering terms contrast the two fits
aov(yield ~ block + N * P + K, npk)
aov(terms(yield ~ block + N * P + K, keep.order = TRUE), npk)


## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
npk.aovE
summary(npk.aovE)
options(op)  # reset to previous
```

---

approxfun *Interpolation Functions*

Description

Return a list of points which linearly interpolate given data points, or a function performing the linear (or constant) interpolation.

Usage

```
approx   (x, y = NULL, xout, method = "linear", n = 50,
          yleft, yright, rule = 1, f = 0, ties = mean, na.rm = TRUE)

approxfun(x, y = NULL,       method = "linear",
          yleft, yright, rule = 1, f = 0, ties = mean, na.rm = TRUE)
```

Arguments

x, y	numeric vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see xy.coords.
xout	an optional set of numeric values specifying where interpolation is to take place.
method	specifies the interpolation method to be used. Choices are "linear" or "constant".
n	If xout is not specified, interpolation takes place at n equally spaced points spanning the interval [min(x), max(x)].
yleft	the value to be returned when input x values are less than min(x). The default is defined by the value of rule given below.
yright	the value to be returned when input x values are greater than max(x). The default is defined by the value of rule given below.
rule	an integer (of length 1 or 2) describing how interpolation is to take place outside the interval [min(x), max(x)]. If rule is 1 then NAs are returned for such points and if it is 2, the value at the closest data extreme is used. Use, e.g., rule = 2:1, if the left and right side extrapolation should differ.
f	for method = "constant" a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If y0 and y1 are the values to the left and right of the point then the value is y0 if f == 0, y1 if f == 1, and y0*(1-f)+y1*f for intermediate values. In this way the result is right-continuous for f == 0 and left-continuous for f == 1, even for non-finite y values.
ties	handling of tied x values. The string "ordered" or a function (or the name of a function) taking a single vector argument and returning a single number or a list of both, e.g., list("ordered", mean), see 'Details'.
na.rm	logical specifying how missing values (NA's) should be handled. Setting na.rm=FALSE will propagate NA's in y to the interpolated values, also depending on the rule set. Note that in this case, NA's in x are invalid, see also the examples.

Details

The inputs can contain missing values which are deleted (if na.rm is true, i.e., by default), so at least two complete (x, y) pairs are required (for method = "linear", one otherwise). If there are duplicated (tied) x values and ties contains a function it is applied to the y values for each distinct x value to produce (x,y) pairs with unique x. Useful functions in this context include mean, min, and max.

If ties = "ordered" the x values are assumed to be already ordered (and unique) and ties are *not* checked but kept if present. This is the fastest option for large length(x).

If ties is a list of length two, ties[[2]] must be a function to be applied to ties, see above, but if ties[[1]] is identical to "ordered", the x values are assumed to be sorted and are only checked for ties. Consequently, ties = list("ordered", mean) will be slightly more efficient than the default ties = mean in such a case.

The first y value will be used for interpolation to the left and the last one for interpolation to the right.

Value

approx returns a list with components x and y, containing n coordinates which interpolate the given data points according to the method (and rule) desired.

The function approxfun returns a function performing (linear or constant) interpolation of the given data points. For a given set of x values, this function will return the corresponding interpolated values. It uses data stored in its environment when it was created, the details of which are subject to change.

Warning

The value returned by approxfun contains references to the code in the current version of R: it is not intended to be saved and loaded into a different R session. This is safer for R >= 3.0.0.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

spline and splinefun for spline interpolation.

Examples

```
require(graphics)

x <- 1:10
y <- rnorm(10)
par(mfrow = c(2,1))
plot(x, y, main = "approx(.) and approxfun(.)")
points(approx(x, y), col = 2, pch = "*")
points(approx(x, y, method = "constant"), col = 4, pch = "*")

f <- approxfun(x, y)
curve(f(x), 0, 11, col = "green2")
points(x, y)
is.function(fc <- approxfun(x, y, method = "const")) # TRUE
curve(fc(x), 0, 10, col = "darkblue", add = TRUE)
## different extrapolation on left and right side :
plot(approxfun(x, y, rule = 2:1), 0, 11,
     col = "tomato", add = TRUE, lty = 3, lwd = 2)

### Treatment of 'NA's -- are kept if  na.rm=FALSE :

xn <- 1:4
yn <- c(1,NA,3:4)
xout <- (1:9)/2
## Default behavior (na.rm = TRUE): NA's omitted; extrapolation gives NA
data.frame(approx(xn,yn, xout))
data.frame(approx(xn,yn, xout, rule = 2))# -> *constant* extrapolation
## New (2019-2020)  na.rm = FALSE: NA's are "kept"
```

```
data.frame(approx(xn,yn, xout, na.rm=FALSE, rule = 2))
data.frame(approx(xn,yn, xout, na.rm=FALSE, rule = 2, method="constant"))

## NA's in x[] are not allowed:
stopifnot(inherits( try( approx(yn,yn, na.rm=FALSE) ), "try-error"))

## Give a nice overview of all possibilities  rule * method * na.rm :
##             ----------------------------  ====   ======   =====
## extrapolations "N":= NA;   "C":= Constant :
rules <- list(N=1, C=2, NC=1:2, CN=2:1)
methods <- c("constant","linear")
ry <- sapply(rules, function(R) {
       sapply(methods, function(M)
        sapply(setNames(,c(TRUE,FALSE)), function(na.)
                 approx(xn, yn, xout=xout, method=M, rule=R, na.rm=na.)$y),
        simplify="array")
   }, simplify="array")
names(dimnames(ry)) <- c("x = ", "na.rm", "method", "rule")
dimnames(ry)[[1]] <- format(xout)
ftable(aperm(ry, 4:1)) # --> (4 * 2 * 2) x length(xout)  =  16 x 9 matrix


## Show treatment of 'ties' :

x <- c(2,2:4,4,4,5,5,7,7,7)
y <- c(1:6, 5:4, 3:1)
(amy <- approx(x, y, xout = x)$y) # warning, can be avoided by specifying 'ties=':
op <- options(warn=2) # warnings would be error
stopifnot(identical(amy, approx(x, y, xout = x, ties=mean)$y))
(ay <- approx(x, y, xout = x, ties = "ordered")$y)
stopifnot(amy == c(1.5,1.5, 3, 5,5,5, 4.5,4.5, 2,2,2),
          ay  == c(2, 2,    3, 6,6,6, 4, 4,    1,1,1))
approx(x, y, xout = x, ties = min)$y
approx(x, y, xout = x, ties = max)$y
options(op) # revert 'warn'ing level
```

---

ar *Fit Autoregressive Models to Time Series*

Description

Fit an autoregressive time series model to the data, by default selecting the complexity by AIC.

Usage

```
ar(x, aic = TRUE, order.max = NULL,
   method = c("yule-walker", "burg", "ols", "mle", "yw"),
   na.action, series, ...)

ar.burg(x, ...)
```

```
## Default S3 method:
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)
## S3 method for class 'mts'
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)

ar.yw(x, ...)
## Default S3 method:
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, ...)
## S3 method for class 'mts'
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series,
      var.method = 1, ...)

ar.mle(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, series, ...)

## S3 method for class 'ar'
predict(object, newdata, n.ahead = 1, se.fit = TRUE, ...)
```

Arguments

x	a univariate or multivariate time series.
aic	logical. If TRUE then the Akaike Information Criterion is used to choose the order of the autoregressive model. If FALSE, the model of order order.max is fitted.
order.max	maximum order (or order) of model to fit. Defaults to the smaller of $N - 1$ and $10 \log_{10}(N)$ where N is the number of non-missing observations except for method = "mle" where it is the minimum of this quantity and 12.
method	character string specifying the method to fit the model. Must be one of the strings in the default argument (the first few characters are sufficient). Defaults to "yule-walker".
na.action	function to be called to handle missing values. Currently, via na.action = na.pass, only Yule-Walker method can handle missing values which must be consistent within a time point: either all variables must be missing or none.
demean	should a mean be estimated during fitting?
series	names for the series. Defaults to deparse1(substitute(x)).
var.method	the method to estimate the innovations variance (see 'Details').
...	additional arguments for specific methods.
object	a fit from ar().
newdata	data to which to apply the prediction.
n.ahead	number of steps ahead at which to predict.
se.fit	logical: return estimated standard errors of the prediction error?

Details

For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

`ar` is just a wrapper for the functions `ar.yw`, `ar.burg`, `ar.ols` and `ar.mle`.

Order selection is done by AIC if `aic` is true. This is problematic, as of the methods here only `ar.mle` performs true maximum likelihood estimation. The AIC is computed as if the variance estimate were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values. In `ar.yw` the variance matrix of the innovations is computed from the fitted coefficients and the autocovariance of `x`.

`ar.burg` allows two methods to estimate the innovations variance and hence AIC. Method 1 is to use the update given by the Levinson-Durbin recursion (Brockwell and Davis, 1991, (8.2.6) on page 242), and follows S-PLUS. Method 2 is the mean of the sum of squares of the forward and backward prediction errors (as in Brockwell and Davis, 1996, page 145). Percival and Walden (1998) discuss both. In the multivariate case the estimated coefficients will depend (slightly) on the variance estimation method.

Remember that `ar` includes by default a constant in the model, by removing the overall mean of `x` before fitting the AR model, or (`ar.mle`) estimating a constant to subtract.

Value

For `ar` and its methods a list of class `"ar"` with the following elements:

`order`	The order of the fitted model. This is chosen by minimizing the AIC if `aic = TRUE`, otherwise it is `order.max`.
`ar`	Estimated autoregression coefficients for the fitted model.
`var.pred`	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
`x.mean`	The estimated mean of the series used in fitting and for use in prediction.
`x.intercept`	(`ar.ols` only.) The intercept in the model for `x - x.mean`.
`aic`	The differences in AIC between each model and the best-fitting model. Note that the latter can have an AIC of `-Inf`.
`n.used`	The number of observations in the time series, including missing.
`n.obs`	The number of non-missing observations in the time series.
`order.max`	The value of the `order.max` argument.
`partialacf`	The estimate of the partial autocorrelation function up to lag `order.max`.
`resid`	residuals from the fitted model, conditioning on the first `order` observations. The first `order` residuals are set to `NA`. If `x` is a time series, so is `resid`.
`method`	The value of the `method` argument.
`series`	The name(s) of the time series.
`frequency`	The frequency of the time series.
`call`	The matched call.

`asy.var.coef` (univariate case, `order > 0`.) The asymptotic-theory variance matrix of the coefficient estimates.

For `predict.ar`, a time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

Note

Only the univariate case of `ar.mle` is implemented.

Fitting by `method="mle"` to long series can be very slow.

If `x` contains missing values, see `NA`, also consider using `arima()`, possibly with `method = "ML"`.

Author(s)

Martyn Plummer. Univariate case of `ar.yw`, `ar.mle` and C code for univariate case of `ar.burg` by B. D. Ripley.

References

Brockwell, P. J. and Davis, R. A. (1991). *Time Series and Forecasting Methods*, second edition. Springer, New York. Section 11.4.

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.

Percival, D. P. and Walden, A. T. (1998). *Spectral Analysis for Physical Applications*. Cambridge University Press.

Whittle, P. (1963). On the fitting of multivariate autoregressions and the approximate canonical factorization of a spectral density matrix. *Biometrika*, **40**, 129–134. doi:10.2307/2333753.

See Also

`ar.ols`, `arima` for ARMA models; `acf2AR`, for AR construction from the ACF.

`arima.sim` for simulation of AR processes.

Examples

```
ar(lh)
ar(lh, method = "burg")
ar(lh, method = "ols")
ar(lh, FALSE, 4) # fit ar(4)

(sunspot.ar <- ar(sunspot.year))
predict(sunspot.ar, n.ahead = 25)
## try the other methods too

ar(ts.union(BJsales, BJsales.lead))
## Burg is quite different here, as is OLS (see ar.ols)
ar(ts.union(BJsales, BJsales.lead), method = "burg")
```

`ar.ols` *Fit Autoregressive Models to Time Series by OLS*

Description

Fit an autoregressive time series model to the data by ordinary least squares, by default selecting the complexity by AIC.

Usage

```
ar.ols(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, intercept = demean, series, ...)
```

Arguments

`x`	A univariate or multivariate time series.
`aic`	Logical flag. If `TRUE` then the Akaike Information Criterion is used to choose the order of the autoregressive model. If `FALSE`, the model of order `order.max` is fitted.
`order.max`	Maximum order (or order) of model to fit. Defaults to $10\log_{10}(N)$ where N is the number of observations.
`na.action`	function to be called to handle missing values.
`demean`	should the AR model be for `x` minus its mean?
`intercept`	should a separate intercept term be fitted?
`series`	names for the series. Defaults to `deparse1(substitute(x))`.
`...`	further arguments to be passed to or from methods.

Details

`ar.ols` fits the general AR model to a possibly non-stationary and/or multivariate system of series `x`. The resulting unconstrained least squares estimates are consistent, even if some of the series are non-stationary and/or co-integrated. For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_0 + a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

where a_0 is zero unless `intercept` is true, and μ is the sample mean if `demean` is true, zero otherwise.

Order selection is done by AIC if `aic` is true. This is problematic, as `ar.ols` does not perform true maximum likelihood estimation. The AIC is computed as if the variance estimate (computed from the variance matrix of the residuals) were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values.

Some care is needed if `intercept` is true and `demean` is false. Only use this is the series are roughly centred on zero. Otherwise the computations may be inaccurate or fail entirely.

Value

A list of class "ar" with the following elements:

order	The order of the fitted model. This is chosen by minimizing the AIC if aic = TRUE, otherwise it is order.max.
ar	Estimated autoregression coefficients for the fitted model.
var.pred	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
x.mean	The estimated mean (or zero if demean is false) of the series used in fitting and for use in prediction.
x.intercept	The intercept in the model for x - x.mean, or zero if intercept is false.
aic	The differences in AIC between each model and the best-fitting model. Note that the latter can have an AIC of -Inf.
n.used	The number of observations in the time series.
order.max	The value of the order.max argument.
partialacf	NULL. For compatibility with ar.
resid	residuals from the fitted model, conditioning on the first order observations. The first order residuals are set to NA. If x is a time series, so is resid.
method	The character string "Unconstrained LS".
series	The name(s) of the time series.
frequency	The frequency of the time series.
call	The matched call.
asy.se.coef	The asymptotic-theory standard errors of the coefficient estimates.

Author(s)

Adrian Trapletti, Brian Ripley.

References

Luetkepohl, H. (1991): *Introduction to Multiple Time Series Analysis.* Springer Verlag, NY, pp. 368–370.

See Also

ar

Examples

```
ar(lh, method = "burg")
ar.ols(lh)
ar.ols(lh, FALSE, 4) # fit ar(4)

ar.ols(ts.union(BJsales, BJsales.lead))

x <- diff(log(EuStockMarkets))
ar.ols(x, order.max = 6, demean = FALSE, intercept = TRUE)
```

arima *ARIMA Modelling of Time Series*

Description

Fit an ARIMA model to a univariate time series.

Usage

```
arima(x, order = c(0L, 0L, 0L),
      seasonal = list(order = c(0L, 0L, 0L), period = NA),
      xreg = NULL, include.mean = TRUE,
      transform.pars = TRUE,
      fixed = NULL, init = NULL,
      method = c("CSS-ML", "ML", "CSS"), n.cond,
      SSinit = c("Gardner1980", "Rossignol2011"),
      optim.method = "BFGS",
      optim.control = list(), kappa = 1e6)
```

Arguments

x	a univariate time series
order	A specification of the non-seasonal part of the ARIMA model: the three integer components (p, d, q) are the AR order, the degree of differencing, and the MA order.
seasonal	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to frequency(x)). This may be a list with components order and period, or just a numeric vector of length 3 which specifies the seasonal order. In the latter case the default period is used.
xreg	Optionally, a vector or matrix of external regressors, which must have the same number of rows as x.
include.mean	Should the ARMA model include a mean/intercept term? The default is TRUE for undifferenced series, and it is ignored for ARIMA models with differencing.
transform.pars	logical; if true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for method = "CSS". For method = "ML", it has been advantageous to set transform.pars = FALSE in some cases, see also fixed.
fixed	optional numeric vector of the same length as the total number of coefficients to be estimated. It should be of the form $$(\phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q, \Phi_1, \ldots, \Phi_P, \Theta_1, \ldots, \Theta_Q, \mu),$$ where ϕ_i are the AR coefficients, θ_i are the MA coefficients, Φ_i are the seasonal AR coefficients, Θ_i are the seasonal MA coefficients and μ is the intercept term. Note that the μ entry is required if and only if include.mean is TRUE. In particular it should not be present if the model is an ARIMA model with differencing.

The entries of the `fixed` vector should consist of the values at which the user wishes to "fix" the corresponding coefficient, or `NA` if that coefficient should *not* be fixed, but estimated.

The argument `transform.pars` will be set to `FALSE` if any AR parameters are fixed. A warning will be given if `transform.pars` is set to (or left at its default) `TRUE`. It may be wise to set `transform.pars = FALSE` even when fixing MA parameters, especially at values that cause the model to be nearly non-invertible.

`init` optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in `fixed` will be ignored.

`method` fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood. Can be abbreviated.

`n.cond` only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.

`SSinit` a string specifying the algorithm to compute the state-space initialization of the likelihood; see `KalmanLike` for details. Can be abbreviated.

`optim.method` The value passed as the `method` argument to `optim`.

`optim.control` List of control parameters for `optim`.

`kappa` the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model. Do not reduce this.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition used here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \cdots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those used in documentation written for S-PLUS. Further, if `include.mean` is true (the default for an ARMA model), this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model. If an `xreg` term is included, a linear regression (with a constant term if `include.mean` is true and there is no differencing) is fitted with an ARMA model for the error term.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide.

Optimization is done by `optim`. It will work best if the columns in `xreg` are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Value

A list of class `"Arima"` with components:

`coef` a vector of AR, MA and regression coefficients, which can be extracted by the `coef` method.

sigma2	the MLE of the innovations variance.
var.coef	the estimated variance matrix of the coefficients coef, which can be extracted by the vcov method.
loglik	the maximized log-likelihood (of the differenced data), or the approximation to it used.
arma	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
aic	the AIC value corresponding to the log-likelihood. Only valid for method = "ML" fits.
residuals	the fitted innovations.
call	the matched call.
series	the name of the series x.
code	the convergence value returned by optim.
n.cond	the number of initial observations not used in the fitting.
nobs	the number of "used" observations for the fitting, can also be extracted via nobs() and is used by BIC.
model	A list representing the Kalman filter used in the fitting. See KalmanLike.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARIMA process, and the innovations and their variance found by a Kalman filter. The initialization of the differenced ARMA process uses stationarity and is based on Gardner et al. (1980). For a differenced process the non-stationary components are given a diffuse prior (controlled by kappa). Observations which are still controlled by the diffuse prior (determined by having a Kalman gain of at least 1e4) are excluded from the likelihood calculations. (This gives comparable results to arima0 in the absence of missing values, when the observations excluded are precisely those dropped by the differencing.)

Missing values are allowed, and are handled exactly in method "ML".

If transform.pars is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are not constrained to be invertible during optimization, but they will be converted to invertible form after optimization if transform.pars is true.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation n.cond on, (where n.cond is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument n.cond can be used to allow comparability between different fits. The 'part log-likelihood' is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

arima is very similar to arima0 for ARMA models or for differenced models without missing values, but handles differenced models with missing values exactly. It is somewhat slower than arima0, particularly for seasonally differenced models.

References

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 3.3 and 8.3.

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980). Algorithm AS 154: An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics*, **29**, 311–322. doi:10.2307/2346910.

Harvey, A. C. (1993). *Time Series Models*. 2nd Edition. Harvester Wheatsheaf. Sections 3.3 and 4.4.

Jones, R. H. (1980). Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics*, **22**, 389–395. doi:10.2307/1268324.

Ripley, B. D. (2002). "Time series in R 1.5.0". *R News*, **2**(2), 2–7. https://www.r-project.org/doc/Rnews/Rnews_2002-2.pdf

See Also

predict.Arima, arima.sim for simulating from an ARIMA model, tsdiag, arima0, ar

Examples

```
arima(lh, order = c(1,0,0))
arima(lh, order = c(3,0,0))
arima(lh, order = c(1,0,1))

arima(lh, order = c(3,0,0), method = "CSS")

arima(USAccDeaths, order = c(0,1,1), seasonal = list(order = c(0,1,1)))
arima(USAccDeaths, order = c(0,1,1), seasonal = list(order = c(0,1,1)),
      method = "CSS") # drops first 13 observations.
# for a model with as few years as this, we want full ML

arima(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron) - 1920)

## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
require(graphics)
(fit1 <- arima(presidents, c(1, 0, 0)))
nobs(fit1)
tsdiag(fit1)
(fit3 <- arima(presidents, c(3, 0, 0)))  # smaller AIC
tsdiag(fit3)
BIC(fit1, fit3)
```

```
## compare a whole set of models; BIC() would choose the smallest
AIC(fit1, arima(presidents, c(2,0,0)),
          arima(presidents, c(2,0,1)), # <- chosen (barely) by AIC
    fit3, arima(presidents, c(3,0,1)))

## An example of using the  'fixed'  argument:
## Note that the period of the seasonal component is taken to be
## frequency(presidents), i.e. 4.
(fitSfx <- arima(presidents, order=c(2,0,1), seasonal=c(1,0,0),
                 fixed=c(NA, NA, 0.5, -0.1, 50), transform.pars=FALSE))
## The partly-fixed & smaller model seems better (as we "knew too much"):
AIC(fitSfx, arima(presidents, order=c(2,0,1), seasonal=c(1,0,0)))

## An example of ARIMA forecasting:
predict(fit3, 3)
```

`arima.sim` *Simulate from an ARIMA Model*

Description

Simulate from an ARIMA model.

Usage

```
arima.sim(model, n, rand.gen = rnorm, innov = rand.gen(n, ...),
          n.start = NA, start.innov = rand.gen(n.start, ...),
          ...)
```

Arguments

`model`	A list with component `ar` and/or `ma` giving the AR and MA coefficients respectively. Optionally a component `order` can be used. An empty list gives an ARIMA(0, 0, 0) model, that is white noise.
`n`	length of output series, before un-differencing. A strictly positive integer.
`rand.gen`	optional: a function to generate the innovations.
`innov`	an optional times series of innovations. If not provided, `rand.gen` is used.
`n.start`	length of 'burn-in' period. If `NA`, the default, a reasonable value is computed.
`start.innov`	an optional times series of innovations to be used for the burn-in period. If supplied there must be at least `n.start` values (and `n.start` is by default computed inside the function).
`...`	additional arguments for `rand.gen`. Most usefully, the standard deviation of the innovations generated by `rnorm` can be specified by `sd`.

Details

See arima for the precise definition of an ARIMA model.

The ARMA model is checked for stationarity.

ARIMA models are specified via the order component of model, in the same way as for arima. Other aspects of the order component are ignored, but inconsistent specifications of the MA and AR orders are detected. The un-differencing assumes previous values of zero, and to remind the user of this, those values are returned.

Random inputs for the 'burn-in' period are generated by calling rand.gen.

Value

A time-series object of class "ts".

See Also

arima

Examples

```
require(graphics)

arima.sim(n = 63, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
          sd = sqrt(0.1796))
# mildly long-tailed
arima.sim(n = 63, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
          rand.gen = function(n, ...) sqrt(0.1796) * rt(n, df = 5))

# An ARIMA simulation
ts.sim <- arima.sim(list(order = c(1,1,0), ar = 0.7), n = 200)
ts.plot(ts.sim)
```

---

arima0 *ARIMA Modelling of Time Series – Preliminary Version*

Description

Fit an ARIMA model to a univariate time series, and forecast from the fitted model.

Usage

```
arima0(x, order = c(0, 0, 0),
       seasonal = list(order = c(0, 0, 0), period = NA),
       xreg = NULL, include.mean = TRUE, delta = 0.01,
       transform.pars = TRUE, fixed = NULL, init = NULL,
       method = c("ML", "CSS"), n.cond, optim.control = list())

## S3 method for class 'arima0'
predict(object, n.ahead = 1, newxreg, se.fit = TRUE, ...)
```

Arguments

x	a univariate time series
order	A specification of the non-seasonal part of the ARIMA model: the three components (p, d, q) are the AR order, the degree of differencing, and the MA order.
seasonal	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to frequency(x)). This should be a list with components order and period, but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the order.
xreg	Optionally, a vector or matrix of external regressors, which must have the same number of rows as x.
include.mean	Should the ARIMA model include a mean term? The default is TRUE for undifferenced series, FALSE for differenced ones (where a mean would not affect the fit nor predictions).
delta	A value to indicate at which point 'fast recursions' should be used. See the 'Details' section.
transform.pars	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for method = "CSS".
fixed	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. transform.pars = TRUE will be overridden (with a warning) if any ARMA parameters are fixed.
init	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in fixed will be ignored.
method	Fitting method: maximum likelihood or minimize conditional sum-of-squares. Can be abbreviated.
n.cond	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.
optim.control	List of control parameters for optim.
object	The result of an arima0 fit.
newxreg	New values of xreg to be used for prediction. Must have at least n.ahead rows.
n.ahead	The number of steps ahead for which prediction is required.
se.fit	Logical: should standard errors of prediction be returned?
...	arguments passed to or from other methods.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \cdots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those given by S-PLUS. Further, if `include.mean` is true, this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide, especially for fits close to the boundary of invertibility.

Optimization is done by `optim`. It will work best if the columns in `xreg` are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Finite-history prediction is used. This is only statistically efficient if the MA part of the fit is invertible, so `predict.arima0` will give a warning for non-invertible MA models.

Value

For `arima0`, a list of class `"arima0"` with components:

`coef`	a vector of AR, MA and regression coefficients,
`sigma2`	the MLE of the innovations variance.
`var.coef`	the estimated variance matrix of the coefficients `coef`.
`loglik`	the maximized log-likelihood (of the differenced data), or the approximation to it used.
`arma`	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
`aic`	the AIC value corresponding to the log-likelihood. Only valid for `method = "ML"` fits.
`residuals`	the fitted innovations.
`call`	the matched call.
`series`	the name of the series `x`.
`convergence`	the value returned by `optim`.
`n.cond`	the number of initial observations not used in the fitting.

For `predict.arima0`, a time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARMA process, and the innovations and their variance found by a Kalman filter based on Gardner et al. (1980). This has the option to switch to 'fast recursions' (assume an effectively infinite past) if the innovations variance is close enough to its asymptotic bound. The argument `delta` sets the tolerance: at its default value the approximation is normally negligible and the speed-up considerable. Exact computations can be ensured by setting `delta` to a negative value.

If `transform.pars` is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are also constrained to

be invertible during optimization by the same transformation if `transform.pars` is true. Note that the MLE for MA terms does sometimes occur for MA polynomials with unit roots: such models can be fitted by using `transform.pars = FALSE` and specifying a good set of initial values (often obtainable from a fit with `transform.pars = TRUE`).

Missing values are allowed, but any missing values will force `delta` to be ignored and full recursions used. Note that missing values will be propagated by differencing, so the procedure used in this function is not fully efficient in that case.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation `n.cond` on, (where `n.cond` is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument `n.cond` can be used to allow comparability between different fits. The 'part log-likelihood' is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

This is a preliminary version, and will be replaced by `arima`.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients.

The results are likely to be different from S-PLUS's `arima.mle`, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by `arima.mle` reverses the signs of the MA coefficients.

References

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 3.3 and 8.3.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980). Algorithm AS 154: An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics*, **29**, 311–322. doi:10.2307/2346910.

Harvey, A. C. (1993). *Time Series Models*. 2nd Edition. Harvester Wheatsheaf. Sections 3.3 and 4.4.

Harvey, A. C. and McKenzie, C. R. (1982). Algorithm AS 182: An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics*, **31**, 180–187. doi:10.2307/2347987.

Jones, R. H. (1980). Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics*, **22**, 389–395. doi:10.2307/1268324.

See Also

`arima`, `ar`, `tsdiag`

Examples

```
## Not run: arima0(lh, order = c(1,0,0))
arima0(lh, order = c(3,0,0))
arima0(lh, order = c(1,0,1))
predict(arima0(lh, order = c(3,0,0)), n.ahead = 12)

arima0(lh, order = c(3,0,0), method = "CSS")

# for a model with as few years as this, we want full ML
(fit <- arima0(USAccDeaths, order = c(0,1,1),
               seasonal = list(order=c(0,1,1)), delta = -1))
predict(fit, n.ahead = 6)

arima0(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)
## Not run:
## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima0(presidents, c(1, 0, 0), delta = -1))  # avoid warning
tsdiag(fit1)
(fit3 <- arima0(presidents, c(3, 0, 0), delta = -1))  # smaller AIC
tsdiag(fit3)
## End(Not run)
```

ARMAacf *Compute Theoretical ACF for an ARMA Process*

Description

Compute the theoretical autocorrelation function or partial autocorrelation function for an ARMA process.

Usage

```
ARMAacf(ar = numeric(), ma = numeric(), lag.max = r, pacf = FALSE)
```

Arguments

ar	numeric vector of AR coefficients
ma	numeric vector of MA coefficients
lag.max	integer. Maximum lag required. Defaults to max(p, q+1), where p, q are the numbers of AR and MA terms respectively.
pacf	logical. Should the partial autocorrelations be returned?

Details

The methods used follow Brockwell & Davis (1991, section 3.3). Their equations (3.3.8) are solved for the autocovariances at lags $0, \ldots, \max(p, q+1)$, and the remaining autocorrelations are given by a recursive filter.

Value

A vector of (partial) autocorrelations, named by the lags.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

`arima`, `ARMAtoMA`, `acf2AR` for inverting part of `ARMAacf`; further `filter`.

Examples

```
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10)

## Example from Brockwell & Davis (1991, pp.92-4)
## answer: 2^(-n) * (32/3 + 8 * n) /(32/3)
n <- 1:10
a.n <- 2^(-n) * (32/3 + 8 * n) /(32/3)
(A.n <- ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10))
stopifnot(all.equal(unname(A.n), c(1, a.n)))

ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10, pacf = TRUE)
zapsmall(ARMAacf(c(1.0, -0.25), lag.max = 10, pacf = TRUE))

## Cov-Matrix of length-7 sub-sample of AR(1) example:
toeplitz(ARMAacf(0.8, lag.max = 7))
```

ARMAtoMA *Convert ARMA Process to Infinite MA Process*

Description

Convert ARMA process to infinite MA process.

Usage

```
ARMAtoMA(ar = numeric(), ma = numeric(), lag.max)
```

Arguments

`ar`	numeric vector of AR coefficients
`ma`	numeric vector of MA coefficients
`lag.max`	Largest MA(Inf) coefficient required.

Value

A vector of coefficients.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

arima, ARMAacf.

Examples

```
ARMAtoMA(c(1.0, -0.25), 1.0, 10)
## Example from Brockwell & Davis (1991, p.92)
## answer (1 + 3*n)*2^(-n)
n <- 1:10; (1 + 3*n)*2^(-n)
```

---

as.hclust *Convert Objects to Class* "hclust"

Description

Converts objects from other hierarchical clustering functions to class "hclust".

Usage

```
as.hclust(x, ...)
```

Arguments

x	Hierarchical clustering object
...	further arguments passed to or from other methods.

Details

Currently there is only support for converting objects of class "twins" as produced by the functions diana and agnes from the package **cluster**. The default method throws an error unless passed an "hclust" object.

Value

An object of class "hclust".

See Also

hclust, and from package **cluster**, diana and agnes

Examples

```
x <- matrix(rnorm(30), ncol = 3)
hc <- hclust(dist(x), method = "complete")

if(require("cluster", quietly = TRUE)) {# is a recommended package
  ag <- agnes(x, method = "complete")
  hcag <- as.hclust(ag)
  ## The dendrograms order slightly differently:
  op <- par(mfrow = c(1,2))
  plot(hc) ;  mtext("hclust", side = 1)
  plot(hcag); mtext("agnes",  side = 1)
  detach("package:cluster")
}
```

---

asOneSidedFormula *Convert to One-Sided Formula*

Description

Names, calls, expressions (first element), numeric values, and character strings are converted to one-sided formulae associated with the global environment. If the input is a formula, it must be one-sided, in which case it is returned unaltered.

Usage

```
asOneSidedFormula(object)
```

Arguments

`object`	a one-sided formula, name, call, expression, numeric value, or character string.

Value

a one-sided formula representing `object`

Author(s)

José Pinheiro and Douglas Bates

See Also

formula

Examples

```
(form <- asOneSidedFormula("age"))
stopifnot(exprs = {
    identical(form, asOneSidedFormula(form))
    identical(form, asOneSidedFormula(as.name("age")))
    identical(form, asOneSidedFormula(expression(age)))
})
asOneSidedFormula(quote(log(age)))
asOneSidedFormula(1)
```

---

ave *Group Averages Over Level Combinations of Factors*

Description

Subsets of `x[]` are averaged, where each subset consist of those observations with the same factor levels.

Usage

```
ave(x, ..., FUN = mean)
```

Arguments

`x`	A numeric.
`...`	Grouping variables, typically factors, all of the same `length` as `x`.
`FUN`	Function to apply for each factor level combination.

Value

A numeric vector, say `y` of length `length(x)`. If `...` is `g1`, `g2`, e.g., `y[i]` is equal to `FUN(x[j]`, for all `j` with `g1[j] == g1[i]` and `g2[j] == g2[i])`.

See Also

`mean`, `median`.

Examples

```
require(graphics)

ave(1:3)  # no grouping -> grand mean

attach(warpbreaks)
ave(breaks, wool)
ave(breaks, tension)
ave(breaks, tension, FUN = function(x) mean(x, trim = 0.1))
plot(breaks, main =
```

```
     "ave( Warpbreaks )  for   wool  x  tension  combinations")
lines(ave(breaks, wool, tension             ), type = "s", col = "blue")
lines(ave(breaks, wool, tension, FUN = median), type = "s", col = "green")
legend(40, 70, c("mean", "median"), lty = 1,
      col = c("blue","green"), bg = "gray90")
detach()
```

---

bandwidth *Bandwidth Selectors for Kernel Density Estimation*

Description

Bandwidth selectors for Gaussian kernels in density.

Usage

```
bw.nrd0(x)

bw.nrd(x)

bw.ucv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax,
       tol = 0.1 * lower)

bw.bcv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax,
       tol = 0.1 * lower)

bw.SJ(x, nb = 1000, lower = 0.1 * hmax, upper = hmax,
      method = c("ste", "dpi"), tol = 0.1 * lower)
```

Arguments

x	numeric vector.
nb	number of bins to use.
lower, upper	range over which to minimize. The default is almost always satisfactory. hmax is calculated internally from a normal reference bandwidth.
method	either "ste" ("solve-the-equation") or "dpi" ("direct plug-in"). Can be abbreviated.
tol	for method "ste", the convergence tolerance for uniroot. The default leads to bandwidth estimates with only slightly more than one digit accuracy, which is sufficient for practical density estimation, but possibly not for theoretical simulation studies.

Details

`bw.nrd0` implements a rule-of-thumb for choosing the bandwidth of a Gaussian kernel density estimator. It defaults to 0.9 times the minimum of the standard deviation and the interquartile range divided by 1.34 times the sample size to the negative one-fifth power (= Silverman's 'rule of thumb', Silverman (1986, page 48, eqn (3.31))) *unless* the quartiles coincide when a positive result will be guaranteed.

`bw.nrd` is the more common variation given by Scott (1992), using factor 1.06.

`bw.ucv` and `bw.bcv` implement unbiased and biased cross-validation respectively.

`bw.SJ` implements the methods of Sheather & Jones (1991) to select the bandwidth using pilot estimation of derivatives.
The algorithm for method `"ste"` solves an equation (via `uniroot`) and because of that, enlarges the interval `c(lower, upper)` when the boundaries were not user-specified and do not bracket the root.

The last three methods use all pairwise binned distances: they are of complexity $O(n^2)$ up to `n = nb/2` and $O(n)$ thereafter. Because of the binning, the results differ slightly when `x` is translated or sign-flipped.

Value

A bandwidth on a scale suitable for the `bw` argument of `density`.

Note

Long vectors `x` are not supported, but neither are they by `density` and kernel density estimation and for more than a few thousand points a histogram would be preferred.

Author(s)

B. D. Ripley, taken from early versions of package **MASS**.

References

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* New York: Wiley.

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society Series B*, **53**, 683–690. doi:10.1111/j.25176161.1991.tb01857.x.

Silverman, B. W. (1986). *Density Estimation*. London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer.

See Also

`density`.

`bandwidth.nrd`, `ucv`, `bcv` and `width.SJ` in package **MASS**, which are all scaled to the `width` argument of `density` and so give answers four times as large.

Examples

```
require(graphics)

plot(density(precip, n = 1000))
rug(precip)
lines(density(precip, bw = "nrd"), col = 2)
lines(density(precip, bw = "ucv"), col = 3)
lines(density(precip, bw = "bcv"), col = 4)
lines(density(precip, bw = "SJ-ste"), col = 5)
lines(density(precip, bw = "SJ-dpi"), col = 6)
legend(55, 0.035,
       legend = c("nrd0", "nrd", "ucv", "bcv", "SJ-ste", "SJ-dpi"),
       col = 1:6, lty = 1)
```

---

bartlett.test *Bartlett Test of Homogeneity of Variances*

Description

Performs Bartlett's test of the null that the variances in each of the groups (samples) are the same.

Usage

```
bartlett.test(x, ...)

## Default S3 method:
bartlett.test(x, g, ...)

## S3 method for class 'formula'
bartlett.test(formula, data, subset, na.action, ...)
```

Arguments

x	a numeric vector of data values, or a list of numeric data vectors representing the respective samples, or fitted linear model objects (inheriting from class "lm").
g	a vector or factor object giving the group for the corresponding elements of x. Ignored if x is a list.
formula	a formula of the form lhs ~ rhs where lhs gives the data values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

If x is a list, its elements are taken as the samples or fitted linear models to be compared for homogeneity of variances. In this case, the elements must either all be numeric data vectors or fitted linear model objects, g is ignored, and one can simply use `bartlett.test(x)` to perform the test. If the samples are not yet contained in a list, use `bartlett.test(list(x, ...))`.

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

Value

A list of class `"htest"` containing the following components:

`statistic`	Bartlett's K-squared test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	the character string `"Bartlett test of homogeneity of variances"`.
`data.name`	a character string giving the names of the data.

References

Bartlett, M. S. (1937). Properties of sufficiency and statistical tests. *Proceedings of the Royal Society of London Series A* **160**, 268–282. doi:10.1098/rspa.1937.0109.

See Also

`var.test` for the special case of comparing variances in two samples from normal distributions; `fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `ansari.test` and `mood.test` for two rank based two-sample tests for difference in scale.

Examples

```
require(graphics)

plot(count ~ spray, data = InsectSprays)
bartlett.test(InsectSprays$count, InsectSprays$spray)
bartlett.test(count ~ spray, data = InsectSprays)
```

Beta *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters shape1 and shape2 (and optional non-centrality parameter ncp).

Usage

```
dbeta(x, shape1, shape2, ncp = 0, log = FALSE)
pbeta(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qbeta(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2, ncp = 0)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
shape1, shape2	non-negative parameters of the Beta distribution.
ncp	non-centrality parameter.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The Beta distribution with parameters shape1 $= a$ and shape2 $= b$ has density

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

for $a > 0$, $b > 0$ and $0 \le x \le 1$ where the boundary values at $x = 0$ or $x = 1$ are defined as by continuity (as limits).
The mean is $a/(a+b)$ and the variance is $ab/((a+b)^2(a+b+1))$. If $a, b > 1$, (or one of them $= 1$), the mode is $(a-1)/(a+b-2)$. These and all other distributional properties can be defined as limits (leading to point masses at 0, 1/2, or 1) when a or b are zero or infinite, and the corresponding [dpqr]beta() functions are defined correspondingly.

pbeta is closely related to the incomplete beta function. As defined by Abramowitz and Stegun 6.6.1

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt,$$

and 6.6.2 $I_x(a,b) = B_x(a,b)/B(a,b)$ where $B(a,b) = B_1(a,b)$ is the Beta function (beta).

$I_x(a,b)$ is pbeta(x, a, b).

The noncentral Beta distribution (with ncp $= \lambda$) is defined (Johnson et al., 1995, pp. 502) as the distribution of $X/(X+Y)$ where $X \sim \chi^2_{2a}(\lambda)$ and $Y \sim \chi^2_{2b}$. There, $\chi^2_n(\lambda)$ is the noncentral chi-squared distribution with n degrees of freedom and non-centrality parameter λ, see Chisquare.

Value

dbeta gives the density, pbeta the distribution function, qbeta the quantile function, and rbeta generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by `n` for `rbeta`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

Supplying `ncp = 0` uses the algorithm for the non-central distribution, which is not the same algorithm as when `ncp` is omitted. This is to give consistent behaviour in extreme cases with values of `ncp` very near zero.

Source

- The central `dbeta` is based on a binomial probability, using code contributed by Catherine Loader (see `dbinom`) if either shape parameter is larger than one, otherwise directly from the definition. The non-central case is based on the derivation as a Poisson mixture of betas (Johnson et al., 1995, pp. 502–3).

- The central `pbeta` for the default (`log_p = FALSE`) uses a C translation based on

  Didonato, A. and Morris, A., Jr, (1992) Algorithm 708: Significant digit computation of the incomplete beta function ratios, *ACM Transactions on Mathematical Software*, **18**, 360–373, doi:10.1145/131766.131776. (See also
  Brown, B. and Lawrence Levy, L. (1994) Certification of algorithm 708: Significant digit computation of the incomplete beta, *ACM Transactions on Mathematical Software*, **20**, 393–397, doi:10.1145/192115.192155.)
  We have slightly tweaked the original "TOMS 708" algorithm, and enhanced for `log.p = TRUE`. For that (log-scale) case, underflow to `-Inf` (i.e., $P = 0$) or `0`, (i.e., $P = 1$) still happens because the original algorithm was designed without log-scale considerations. Underflow to `-Inf` now typically signals a `warning`.

- The non-central `pbeta` uses a C translation of

  Lenth, R. V. (1987) Algorithm AS 226: Computing noncentral beta probabilities. *Applied Statistics*, **36**, 241–244, doi:10.2307/2347558, incorporating
  Frick, H. (1990)'s AS R84, *Applied Statistics*, **39**, 311–2, doi:10.2307/2347780 and
  Lam, M.L. (1995)'s AS R95, *Applied Statistics*, **44**, 551–2, doi:10.2307/2986147.

  This computes the lower tail only, so the upper tail suffers from cancellation and a warning will be given when this is likely to be significant.

- The central case of `qbeta` is based on a C translation of

  Cran, G. W., K. J. Martin and G. E. Thomas (1977). Remark AS R19 and Algorithm AS 109, *Applied Statistics*, **26**, 111–114, doi:10.2307/2346887, and subsequent remarks (AS83 and correction).

  Enhancements, notably for starting values and switching to a log-scale Newton search, by R Core.

- The central case of `rbeta` is based on a C translation of

  R. C. H. Cheng (1978). Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, **21**, 317–322.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, especially chapter 25. Wiley, New York.

See Also

Distributions for other standard distributions.

`beta` for the Beta function.

Examples

```
x <- seq(0, 1, length.out = 21)
dbeta(x, 1, 1)
pbeta(x, 1, 1)

## Visualization, including limit cases:
pl.beta <- function(a,b, asp = if(isLim) 1, ylim = if(isLim) c(0,1.1)) {
  if(isLim <- a == 0 || b == 0 || a == Inf || b == Inf) {
    eps <- 1e-10
    x <- c(0, eps, (1:7)/16, 1/2+c(-eps,0,eps), (9:15)/16, 1-eps, 1)
  } else {
    x <- seq(0, 1, length.out = 1025)
  }
  fx <- cbind(dbeta(x, a,b), pbeta(x, a,b), qbeta(x, a,b))
  f <- fx; f[fx == Inf] <- 1e100
  matplot(x, f, ylab="", type="l", ylim=ylim, asp=asp,
          main = sprintf("[dpq]beta(x, a=%g, b=%g)", a,b))
  abline(0,1,     col="gray", lty=3)
  abline(h = 0:1, col="gray", lty=3)
  legend("top", paste0(c("d","p","q"), "beta(x, a,b)"),
         col=1:3, lty=1:3, bty = "n")
  invisible(cbind(x, fx))
}
pl.beta(3,1)

pl.beta(2, 4)
pl.beta(3, 7)
pl.beta(3, 7, asp=1)

pl.beta(0, 0)   ## point masses at  {0, 1}

pl.beta(0, 2)   ## point mass at 0 ; the same as
pl.beta(1, Inf)

pl.beta(Inf, 2) ## point mass at 1 ; the same as
pl.beta(3, 0)
```

```
pl.beta(Inf, Inf)# point mass at 1/2
```

`binom.test` *Exact Binomial Test*

Description

Performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment.

Usage

```
binom.test(x, n, p = 0.5,
           alternative = c("two.sided", "less", "greater"),
           conf.level = 0.95)
```

Arguments

`x`	number of successes, or a vector of length 2 giving the numbers of successes and failures, respectively.
`n`	number of trials; ignored if `x` has length 2.
`p`	hypothesized probability of success.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter.
`conf.level`	confidence level for the returned confidence interval.

Details

Confidence intervals are obtained by a procedure first given in Clopper and Pearson (1934). This guarantees that the confidence level is at least `conf.level`, but in general does not give the shortest-length confidence intervals.

Value

A list with class `"htest"` containing the following components:

`statistic`	the number of successes.
`parameter`	the number of trials.
`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the probability of success.
`estimate`	the estimated probability of success.
`null.value`	the probability of success under the null, `p`.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"Exact binomial test"`.
`data.name`	a character string giving the names of the data.

References

Clopper, C. J. & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, **26**, 404–413. doi:10.2307/2331986.

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 97–104.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 15–22.

See Also

`prop.test` for a general (approximate) test for equal or given proportions.

Examples

```
## Conover (1971), p. 97f.
## Under (the assumption of) simple Mendelian inheritance, a cross
##  between plants of two particular genotypes produces progeny 1/4 of
##  which are "dwarf" and 3/4 of which are "giant", respectively.
##  In an experiment to determine if this assumption is reasonable, a
##  cross results in progeny having 243 dwarf and 682 giant plants.
##  If "giant" is taken as success, the null hypothesis is that p =
##  3/4 and the alternative that p != 3/4.
binom.test(c(682, 243), p = 3/4)
binom.test(682, 682 + 243, p = 3/4)   # The same.
## => Data are in agreement with the null hypothesis.
```

Binomial *The Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the binomial distribution with parameters `size` and `prob`.

This is conventionally interpreted as the number of 'successes' in `size` trials.

Usage

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
size	number of trials (zero or more).
prob	probability of success on each trial.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The binomial distribution with size $= n$ and prob $= p$ has density

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, \ldots, n$. Note that binomial *coefficients* can be computed by choose in R.

If an element of x is not integer, the result of dbinom is zero, with a warning.

$p(x)$ is computed using Loader's algorithm, see the reference below.

The quantile is defined as the smallest value x such that $F(x) \ge p$, where F is the distribution function.

Value

dbinom gives the density, pbinom gives the distribution function, qbinom gives the quantile function and rbinom generates random deviates.

If size is not an integer, NaN is returned.

The length of the result is determined by n for rbinom, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

Source

For dbinom a saddle-point expansion is used: see

Catherine Loader (2000). *Fast and Accurate Computation of Binomial Probabilities*; available as https://www.r-project.org/doc/reports/CLoader-dbinom-2002.pdf

pbinom uses pbeta.

qbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rbinom (for size < .Machine$integer.max) is based on

Kachitvichyanukul, V. and Schmeiser, B. W. (1988) Binomial random variate generation. *Communications of the ACM*, **31**, 216–222.

For larger values it uses inversion.

See Also

Distributions for other standard distributions, including dnbinom for the negative binomial, and dpois for the Poisson distribution.

Examples

```
require(graphics)
# Compute P(45 < X < 55) for X Binomial(100,0.5)
sum(dbinom(46:54, 100, 0.5))

## Using "log = TRUE" for an extended range :
n <- 2000
k <- seq(0, n, by = 20)
plot (k, dbinom(k, n, pi/10, log = TRUE), type = "l", ylab = "log density",
      main = "dbinom(*, log=TRUE) is better than  log(dbinom(*))")
lines(k, log(dbinom(k, n, pi/10)), col = "red", lwd = 2)
## extreme points are omitted since dbinom gives 0.
mtext("dbinom(k, log=TRUE)", adj = 0)
mtext("extended range", adj = 0, line = -1, font = 4)
mtext("log(dbinom(k))", col = "red", adj = 1)
```

---

biplot *Biplot of Multivariate Data*

Description

Plot a biplot on the current graphics device.

Usage

```
biplot(x, ...)

## Default S3 method:
biplot(x, y, var.axes = TRUE, col, cex = rep(par("cex"), 2),
       xlabs = NULL, ylabs = NULL, expand = 1,
       xlim  = NULL, ylim  = NULL, arrow.len = 0.1,
       main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

x	The biplot, a fitted object. For biplot.default, the first set of points (a two-column matrix), usually associated with observations.
y	The second set of points (a two-column matrix), usually associated with variables.
var.axes	If TRUE the second set of points have arrows representing them as (unscaled) axes.

`col` A vector of length 2 giving the colours for the first and second set of points respectively (and the corresponding axes). If a single colour is specified it will be used for both sets. If missing the default colour is looked for in the `palette`: if there it and the next colour as used, otherwise the first two colours of the palette are used.

`cex` The character expansion factor used for labelling the points. The labels can be of different sizes for the two sets by supplying a vector of length two.

`xlabs` A vector of character strings to label the first set of points: the default is to use the row dimname of `x`, or `1:n` if the dimname is `NULL`.

`ylabs` A vector of character strings to label the second set of points: the default is to use the row dimname of `y`, or `1:n` if the dimname is `NULL`.

`expand` An expansion factor to apply when plotting the second set of points relative to the first. This can be used to tweak the scaling of the two sets to a physically comparable scale.

`arrow.len` The length of the arrow heads on the axes plotted in `var.axes` is true. The arrow head can be suppressed by `arrow.len = 0`.

`xlim`, `ylim` Limits for the x and y axes in the units of the first set of variables.

`main`, `sub`, `xlab`, `ylab`, `...`
graphical parameters.

Details

A biplot is plot which aims to represent both the observations and variables of a matrix of multivariate data on the same plot. There are many variations on biplots (see the references) and perhaps the most widely used one is implemented by `biplot.princomp`. The function `biplot.default` merely provides the underlying code to plot two sets of variables on the same figure.

Graphical parameters can also be given to `biplot`: the size of `xlabs` and `ylabs` is controlled by `cex`.

Side Effects

a plot is produced on the current graphics device.

References

K. R. Gabriel (1971). The biplot graphical display of matrices with application to principal component analysis. *Biometrika*, **58**, 453–467. doi:10.2307/2334381.

J.C. Gower and D. J. Hand (1996). *Biplots*. Chapman & Hall.

See Also

`biplot.princomp`, also for examples.

`biplot.princomp` *Biplot for Principal Components*

Description

Produces a biplot (in the strict sense) from the output of `princomp` or `prcomp`

Usage

```
## S3 method for class 'prcomp'
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)

## S3 method for class 'princomp'
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)
```

Arguments

`x`	an object of class `"princomp"`.
`choices`	length 2 vector specifying the components to plot. Only the default is a biplot in the strict sense.
`scale`	The variables are scaled by `lambda ^ scale` and the observations are scaled by `lambda ^ (1-scale)` where `lambda` are the singular values as computed by `princomp`. Normally `0 <= scale <= 1`, and a warning will be issued if the specified `scale` is outside this range.
`pc.biplot`	If true, use what Gabriel (1971) refers to as a "principal component biplot", with `lambda = 1` and observations scaled up by sqrt(n) and variables scaled down by sqrt(n). Then inner products between variables approximate covariances and distances between observations approximate Mahalanobis distance.
`...`	optional arguments to be passed to `biplot.default`.

Details

This is a method for the generic function `biplot`. There is considerable confusion over the precise definitions: those of the original paper, Gabriel (1971), are followed here. Gabriel and Odoroff (1990) use the same definitions, but their plots actually correspond to `pc.biplot = TRUE`.

Side Effects

a plot is produced on the current graphics device.

References

Gabriel, K. R. (1971). The biplot graphical display of matrices with applications to principal component analysis. *Biometrika*, **58**, 453–467. doi:10.2307/2334381.

Gabriel, K. R. and Odoroff, C. L. (1990). Biplots in biomedical research. *Statistics in Medicine*, **9**, 469–485. doi:10.1002/sim.4780090502.

See Also

biplot, princomp.

Examples

```
require(graphics)
biplot(princomp(USArrests))
```

`birthday` *Probability of coincidences*

Description

Computes answers to a generalised *birthday paradox* problem. `pbirthday` computes the probability of a coincidence and `qbirthday` computes the smallest number of observations needed to have at least a specified probability of coincidence.

Usage

```
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
```

Arguments

`classes`	How many distinct categories the people could fall into
`prob`	The desired probability of coincidence
`n`	The number of people
`coincident`	The number of people to fall in the same category

Details

The birthday paradox is that a very small number of people, 23, suffices to have a 50–50 chance that two or more of them have the same birthday. This function generalises the calculation to probabilities other than 0.5, numbers of coincident events other than 2, and numbers of classes other than 365.

The formula used is approximate for `coincident` > 2. The approximation is very good for moderate values of `prob` but less good for very small probabilities.

Value

`qbirthday`	Minimum number of people needed for a probability of at least `prob` that `k` or more of them have the same one out of `classes` equiprobable labels.
`pbirthday`	Probability of the specified coincidence.

References

Diaconis, P. and Mosteller F. (1989). Methods for studying coincidences. *Journal of the American Statistical Association*, **84**, 853–861. doi:10.1080/01621459.1989.10478847.

Examples

```
require(graphics)

## the standard version
qbirthday() # 23
## probability of > 2 people with the same birthday
pbirthday(23, coincident = 3)

## examples from Diaconis & Mosteller p. 858.
## 'coincidence' is that husband, wife, daughter all born on the 16th
qbirthday(classes = 30, coincident = 3) # approximately 18
qbirthday(coincident = 4)  # exact value 187
qbirthday(coincident = 10) # exact value 1181

## same 4-digit PIN number
qbirthday(classes = 10^4)

## 0.9 probability of three or more coincident birthdays
qbirthday(coincident = 3, prob = 0.9)

## Chance of 4 or more coincident birthdays in 150 people
pbirthday(150, coincident = 4)

## 100 or more coincident birthdays in 1000 people: very rare
pbirthday(1000, coincident = 100)
```

Box.test *Box-Pierce and Ljung-Box Tests*

Description

Compute the Box–Pierce or Ljung–Box test statistic for examining the null hypothesis of independence in a given time series. These are sometimes known as 'portmanteau' tests.

Usage

```
Box.test(x, lag = 1, type = c("Box-Pierce", "Ljung-Box"), fitdf = 0)
```

Arguments

x	a numeric vector or univariate time series.
lag	the statistic will be based on lag autocorrelation coefficients.
type	test to be performed: partial matching is used.
fitdf	number of degrees of freedom to be subtracted if x is a series of residuals.

Details

These tests are sometimes applied to the residuals from an `ARMA(p, q)` fit, in which case the references suggest a better approximation to the null-hypothesis distribution is obtained by setting `fitdf = p+q`, provided of course that `lag > fitdf`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic (taking `fitdf` into account).
`p.value`	the p-value of the test.
`method`	a character string indicating which type of test was performed.
`data.name`	a character string giving the name of the data.

Note

Missing values are not handled.

Author(s)

A. Trapletti

References

Box, G. E. P. and Pierce, D. A. (1970), Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, **65**, 1509–1526. doi:10.2307/2284333.

Ljung, G. M. and Box, G. E. P. (1978), On a measure of lack of fit in time series models. *Biometrika*, **65**, 297–303. doi:10.2307/2335207.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf, NY, pp. 44, 45.

Examples

```
x <- rnorm (100)
Box.test (x, lag = 1)
Box.test (x, lag = 1, type = "Ljung")
```

C *Sets Contrasts for a Factor*

Description

Sets the "contrasts" attribute for the factor.

Usage

```
C(object, contr, how.many, ...)
```

Arguments

object	a factor or ordered factor
contr	which contrasts to use. Can be a matrix with one row for each level of the factor or a suitable function like contr.poly or a character string giving the name of the function
how.many	the number of contrasts to set, by default one less than nlevels(object).
...	additional arguments for the function contr.

Details

For compatibility with S, contr can be treatment, helmert, sum or poly (without quotes) as shorthand for contr.treatment and so on.

Value

The factor object with the "contrasts" attribute set.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

contrasts, contr.sum, etc.

Examples

```
## reset contrasts to defaults
options(contrasts = c("contr.treatment", "contr.poly"))
tens <- with(warpbreaks, C(tension, poly, 1))
attributes(tens)
## tension SHOULD be an ordered factor, but as it is not we can use
aov(breaks ~ wool + tens + tension, data = warpbreaks)

## show the use of ...  The default contrast is contr.treatment here
```

```
summary(lm(breaks ~ wool + C(tension, base = 2), data = warpbreaks))


# following on from help(esoph)
model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
     C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

---

cancor *Canonical Correlations*

Description

Compute the canonical correlations between two data matrices.

Usage

```
cancor(x, y, xcenter = TRUE, ycenter = TRUE)
```

Arguments

x	numeric matrix ($n \times p_1$), containing the x coordinates.
y	numeric matrix ($n \times p_2$), containing the y coordinates.
xcenter	logical or numeric vector of length p_1, describing any centering to be done on the x values before the analysis. If TRUE (default), subtract the column means. If FALSE, do not adjust the columns. Otherwise, a vector of values to be subtracted from the columns.
ycenter	analogous to xcenter, but for the y values.

Details

The canonical correlation analysis seeks linear combinations of the y variables which are well explained by linear combinations of the x variables. The relationship is symmetric as 'well explained' is measured by correlations.

Value

A list containing the following components:

cor	correlations.
xcoef	estimated coefficients for the x variables.
ycoef	estimated coefficients for the y variables.
xcenter	the values used to adjust the x variables.
ycenter	the values used to adjust the x variables.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Hotelling H. (1936). Relations between two sets of variables. *Biometrika*, **28**, 321–327. doi:10.1093/biomet/28.34.321.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley. Page 506f.

See Also

qr, svd.

Examples

```
## signs of results are random
pop <- LifeCycleSavings[, 2:3]
oec <- LifeCycleSavings[, -(2:3)]
cancor(pop, oec)

x <- matrix(rnorm(150), 50, 3)
y <- matrix(rnorm(250), 50, 5)
(cxy <- cancor(x, y))
all(abs(cor(x %*% cxy$xcoef,
            y %*% cxy$ycoef)[,1:3] - diag(cxy $ cor)) < 1e-15)
all(abs(cor(x %*% cxy$xcoef) - diag(3)) < 1e-15)
all(abs(cor(y %*% cxy$ycoef) - diag(5)) < 1e-15)
```

---

case+variable.names *Case and Variable Names of Fitted Models*

Description

Simple utilities returning (non-missing) case names, and (non-eliminated) variable names.

Usage

```
case.names(object, ...)
## S3 method for class 'lm'
case.names(object, full = FALSE, ...)

variable.names(object, ...)
## S3 method for class 'lm'
variable.names(object, full = FALSE, ...)
```

Arguments

object	an R object, typically a fitted model.
full	logical; if TRUE, all names (including zero weights, ...) are returned.
...	further arguments passed to or from other methods.

Value

A character vector.

See Also

`lm`; further, `all.names`, `all.vars` for functions with a similar name but only slightly related purpose.

Examples

```
x <- 1:20
y <-  setNames(x + (x/4 - 2)^3 + rnorm(20, sd = 3),
               paste("O", x, sep = "."))
ww <- rep(1, 20); ww[13] <- 0
summary(lmxy <- lm(y ~ x + I(x^2)+I(x^3) + I((x-10)^2), weights = ww),
        correlation = TRUE)
variable.names(lmxy)
variable.names(lmxy, full = TRUE)  # includes the last
case.names(lmxy)
case.names(lmxy, full = TRUE)      # includes the 0-weight case
```

Cauchy *The Cauchy Distribution*

Description

Density, distribution function, quantile function and random generation for the Cauchy distribution with location parameter `location` and scale parameter `scale`.

Usage

```
dcauchy(x, location = 0, scale = 1, log = FALSE)
pcauchy(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qcauchy(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rcauchy(n, location = 0, scale = 1)
```

Arguments

`x`, `q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location`, `scale`	location and scale parameters.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are not specified, they assume the default values of `0` and `1` respectively.

The Cauchy distribution with location l and scale s has density

$$f(x) = \frac{1}{\pi s}\left(1 + \left(\frac{x-l}{s}\right)^2\right)^{-1}$$

for all x.

Value

`dcauchy`, `pcauchy`, and `qcauchy` are respectively the density, distribution function and quantile function of the Cauchy distribution. `rcauchy` generates random deviates from the Cauchy.

The length of the result is determined by `n` for `rcauchy`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Source

`dcauchy`, `pcauchy` and `qcauchy` are all calculated from numerically stable versions of the definitions.

`rcauchy` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 16. Wiley, New York.

See Also

Distributions for other standard distributions, including `dt` for the t distribution which generalizes `dcauchy(*, l = 0, s = 1)`.

Examples

```
dcauchy(-1:4)
```

`chisq.test` *Pearson's Chi-squared Test for Count Data*

Description

`chisq.test` performs chi-squared contingency table tests and goodness-of-fit tests.

Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)), rescale.p = FALSE,
           simulate.p.value = FALSE, B = 2000)
```

Arguments

`x`	a numeric vector or matrix. `x` and `y` can also both be factors.
`y`	a numeric vector; ignored if `x` is a matrix. If `x` is a factor, `y` should be a factor of the same length.
`correct`	a logical indicating whether to apply continuity correction when computing the test statistic for 2 by 2 tables: one half is subtracted from all $\|O - E\|$ differences; however, the correction will not be bigger than the differences themselves. No correction is done if `simulate.p.value` = `TRUE`.
`p`	a vector of probabilities of the same length as `x`. An error is given if any entry of `p` is negative.
`rescale.p`	a logical scalar; if TRUE then `p` is rescaled (if necessary) to sum to 1. If `rescale.p` is FALSE, and `p` does not sum to 1, an error is given.
`simulate.p.value`	a logical indicating whether to compute p-values by Monte Carlo simulation.
`B`	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If `x` is a matrix with one row or column, or if `x` is a vector and `y` is not given, then a *goodness-of-fit test* is performed (`x` is treated as a one-dimensional contingency table). The entries of `x` must be non-negative integers. In this case, the hypothesis tested is whether the population probabilities equal those in `p`, or are all equal if `p` is not given.

If `x` is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table: the entries of `x` must be non-negative integers. Otherwise, `x` and `y` must be vectors or factors of the same length; cases with missing values are removed, the objects are coerced to factors, and the contingency table is computed from these. Then Pearson's chi-squared test is performed of the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals.

If `simulate.p.value` is `FALSE`, the p-value is computed from the asymptotic chi-squared distribution of the test statistic; continuity correction is only used in the 2-by-2 case (if `correct` is `TRUE`, the

default). Otherwise the p-value is computed for a Monte Carlo test (Hope, 1968) with B replicates. The default B = 2000 implies a minimum p-value of about 0.0005 ($1/(B+1)$).

In the contingency table case, simulation is done by random sampling from the set of all contingency tables with given marginals, and works only if the marginals are strictly positive. Continuity correction is never used, and the statistic is quoted without it. Note that this is not the usual sampling situation assumed for the chi-squared test but rather that for Fisher's exact test.

In the goodness-of-fit case simulation is done by random sampling from the discrete distribution specified by p, each sample being of size n = sum(x). This simulation is done in R and may be slow.

Value

A list with class "htest" containing the following components:

statistic	the value the chi-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic, NA if the p-value is computed by Monte Carlo simulation.
p.value	the p-value for the test.
method	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.
data.name	a character string giving the name(s) of the data.
observed	the observed counts.
expected	the expected counts under the null hypothesis.
residuals	the Pearson residuals, (observed - expected) / sqrt(expected).
stdres	standardized residuals, (observed - expected) / sqrt(V), where V is the residual cell variance (Agresti, 2007, section 2.4.5 for the case where x is a matrix, n * p * (1 - p) otherwise).

Source

The code for Monte Carlo simulation is a C translation of the Fortran algorithm of Patefield (1981).

References

Hope, A. C. A. (1968). A simplified Monte Carlo significance test procedure. *Journal of the Royal Statistical Society Series B*, **30**, 582–598. doi:10.1111/j.25176161.1968.tb00759.x.

Patefield, W. M. (1981). Algorithm AS 159: An efficient method of generating r x c tables with given row and column totals. *Applied Statistics*, **30**, 91–97. doi:10.2307/2346669.

Agresti, A. (2007). *An Introduction to Categorical Data Analysis*, 2nd ed. New York: John Wiley & Sons. Page 38.

See Also

For goodness-of-fit testing, notably of continuous distributions, ks.test.

Examples

```
## From Agresti(2007) p.39
M <- as.table(rbind(c(762, 327, 468), c(484, 239, 477)))
dimnames(M) <- list(gender = c("F", "M"),
                    party = c("Democrat","Independent", "Republican"))
(Xsq <- chisq.test(M))  # Prints test summary
Xsq$observed   # observed counts (same as M)
Xsq$expected   # expected counts under the null
Xsq$residuals  # Pearson residuals
Xsq$stdres     # standardized residuals


## Effect of simulating p-values
x <- matrix(c(12, 5, 7, 7), ncol = 2)
chisq.test(x)$p.value           # 0.4233
chisq.test(x, simulate.p.value = TRUE, B = 10000)$p.value
                                # around 0.29!

## Testing for population probabilities
## Case A. Tabulated data
x <- c(A = 20, B = 15, C = 25)
chisq.test(x)
chisq.test(as.table(x))             # the same
x <- c(89,37,30,28,2)
p <- c(40,20,20,15,5)
try(
chisq.test(x, p = p)                # gives an error
)
chisq.test(x, p = p, rescale.p = TRUE)
                                # works
p <- c(0.40,0.20,0.20,0.19,0.01)
                                # Expected count in category 5
                                # is 1.86 < 5 ==> chi square approx.
chisq.test(x, p = p)            #               maybe doubtful, but is ok!
chisq.test(x, p = p, simulate.p.value = TRUE)

## Case B. Raw data
x <- trunc(5 * runif(100))
chisq.test(table(x))            # NOT 'chisq.test(x)'!
```

`Chisquare` *The (non-central) Chi-Squared Distribution*

Description

Density, distribution function, quantile function and random generation for the chi-squared (χ^2) distribution with `df` degrees of freedom and optional non-centrality parameter `ncp`.

Usage

```
dchisq(x, df, ncp = 0, log = FALSE)
pchisq(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp = 0)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
df	degrees of freedom (non-negative, but can be non-integer).
ncp	non-centrality parameter (non-negative).
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The chi-squared distribution with df$= n \ge 0$ degrees of freedom has density

$$f_n(x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$$

for $x > 0$, where $f_0(x) := \lim_{n\to 0} f_n(x) = \delta_0(x)$, a point mass at zero, is not a density function proper, but a "δ distribution".
The mean and variance are n and $2n$.

The non-central chi-squared distribution with df$= n$ degrees of freedom and non-centrality parameter ncp $= \lambda$ has density

$$f(x) = f_{n,\lambda}(x) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} f_{n+2r}(x)$$

for $x \ge 0$. For integer n, this is the distribution of the sum of squares of n normals each with variance one, λ being the sum of squares of the normal means; further,
$E(X) = n + \lambda$, $Var(X) = 2(n + 2 * \lambda)$, and $E((X - E(X))^3) = 8(n + 3 * \lambda)$.

Note that the degrees of freedom df$= n$, can be non-integer, and also $n = 0$ which is relevant for non-centrality $\lambda > 0$, see Johnson et al. (1995, chapter 29). In that (noncentral, zero df) case, the distribution is a mixture of a point mass at $x = 0$ (of size pchisq(0, df=0, ncp=ncp)) and a continuous part, and dchisq() is *not* a density with respect to that mixture measure but rather the limit of the density for $df \to 0$.

Note that ncp values larger than about 1e5 (and even smaller) may give inaccurate results with many warnings for pchisq and qchisq.

Value

`dchisq` gives the density, `pchisq` gives the distribution function, `qchisq` gives the quantile function, and `rchisq` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rchisq`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

Supplying `ncp = 0` uses the algorithm for the non-central distribution, which is not the same algorithm used if `ncp` is omitted. This is to give consistent behaviour in extreme cases with values of `ncp` very near zero.

The code for non-zero `ncp` is principally intended to be used for moderate values of `ncp`: it will not be highly accurate, especially in the tails, for large values.

Source

The central cases are computed via the gamma distribution.

The non-central `dchisq` and `rchisq` are computed as a Poisson mixture of central chi-squares (Johnson et al., 1995, p.436).

The non-central `pchisq` is for `ncp < 80` computed from the Poisson mixture of central chi-squares and for larger `ncp` *via* a C translation of

Ding, C. G. (1992) Algorithm AS275: Computing the non-central chi-squared distribution function. *Applied Statistics*, **41** 478–482.

which computes the lower tail only (so the upper tail suffers from cancellation and a warning will be given when this is likely to be significant).

The non-central `qchisq` is based on inversion of `pchisq`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, chapters 18 (volume 1) and 29 (volume 2). Wiley, New York.

See Also

Distributions for other standard distributions.

A central chi-squared distribution with n degrees of freedom is the same as a Gamma distribution with `shape` $\alpha = n/2$ and `scale` $\sigma = 2$. Hence, see `dgamma` for the Gamma distribution.

The central chi-squared distribution with 2 d.f. is identical to the exponential distribution with rate 1/2: $\chi^2_2 = Exp(1/2)$, see `dexp`.

Examples

```
require(graphics)

dchisq(1, df = 1:3)
pchisq(1, df =  3)
pchisq(1, df =  3, ncp = 0:4)  # includes the above

x <- 1:10
## Chi-squared(df = 2) is a special exponential distribution
all.equal(dchisq(x, df = 2), dexp(x, 1/2))
all.equal(pchisq(x, df = 2), pexp(x, 1/2))

## non-central RNG -- df = 0 with ncp > 0:  Z0 has point mass at 0!
Z0 <- rchisq(100, df = 0, ncp = 2.)
graphics::stem(Z0)

## visual testing
## do P-P plots for 1000 points at various degrees of freedom
L <- 1.2; n <- 1000; pp <- ppoints(n)
op <- par(mfrow = c(3,3), mar = c(3,3,1,1)+.1, mgp = c(1.5,.6,0),
          oma = c(0,0,3,0))
for(df in 2^(4*rnorm(9))) {
  plot(pp, sort(pchisq(rr <- rchisq(n, df = df, ncp = L), df = df, ncp = L)),
       ylab = "pchisq(rchisq(.),.)", pch = ".")
  mtext(paste("df = ", formatC(df, digits = 4)), line =  -2, adj = 0.05)
  abline(0, 1, col = 2)
}
mtext(expression("P-P plots : Noncentral  "*
                 chi^2 *"(n=1000, df=X, ncp= 1.2)"),
      cex = 1.5, font = 2, outer = TRUE)
par(op)

## "analytical" test
lam <- seq(0, 100, by = .25)
p00 <- pchisq(0,      df = 0, ncp = lam)
p.0 <- pchisq(1e-300, df = 0, ncp = lam)
stopifnot(all.equal(p00, exp(-lam/2)),
          all.equal(p.0, exp(-lam/2)))
```

`cmdscale` *Classical (Metric) Multidimensional Scaling*

Description

Classical multidimensional scaling (MDS) of a data matrix. Also known as *principal coordinates analysis* (Gower, 1966).

Usage

```
cmdscale(d, k = 2, eig = FALSE, add = FALSE, x.ret = FALSE,
         list. = eig || add || x.ret)
```

Arguments

d	a distance structure such as that returned by dist or a full symmetric matrix containing the dissimilarities.
k	the maximum dimension of the space which the data are to be represented in; must be in $\{1, 2, \ldots, n-1\}$.
eig	indicates whether eigenvalues should be returned.
add	logical indicating if an additive constant $c*$ should be computed, and added to the non-diagonal dissimilarities such that the modified dissimilarities are Euclidean.
x.ret	indicates whether the doubly centred symmetric distance matrix should be returned.
list.	logical indicating if a list should be returned or just the $n \times k$ matrix, see 'Value:'.

Details

Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities. (It is a major part of what ecologists call 'ordination'.)

A set of Euclidean distances on n points can be represented exactly in at most $n-1$ dimensions. cmdscale follows the analysis of Mardia (1978), and returns the best-fitting k-dimensional representation, where k may be less than the argument k.

The representation is only determined up to location (cmdscale takes the column means of the configuration to be at the origin), rotations and reflections. The configuration returned is given in principal-component axes, so the reflection chosen may differ between R platforms (see prcomp).

When add = TRUE, a minimal additive constant $c*$ is computed such that the dissimilarities $d_{ij} + c*$ are Euclidean and hence can be represented in n - 1 dimensions. Whereas S (Becker et al., 1988) computes this constant using an approximation suggested by Torgerson, R uses the analytical solution of Cailliez (1983), see also Cox and Cox (2001). Note that because of numerical errors the computed eigenvalues need not all be non-negative, and even theoretically the representation could be in fewer than n - 1 dimensions.

Value

If .list is false (as per default), a matrix with k columns whose rows give the coordinates of the points chosen to represent the dissimilarities.

Otherwise, a list containing the following components.

points	a matrix with up to k columns whose rows give the coordinates of the points chosen to represent the dissimilarities.
eig	the n eigenvalues computed during the scaling process if eig is true. **NB**: versions of R before 2.12.1 returned only k but were documented to return $n-1$.
x	the doubly centered distance matrix if x.ret is true.
ac	the additive constant $c*$, 0 if add = FALSE.

GOF — a numeric vector of length 2, equal to say (g_1, g_2), where $g_i = (\sum_{j=1}^{k} \lambda_j)/(\sum_{j=1}^{n} T_i(\lambda_j))$, where λ_j are the eigenvalues (sorted in decreasing order), $T_1(v) = |v|$, and $T_2(v) = max(v, 0)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Cailliez, F. (1983). The analytical solution of the additive constant problem. *Psychometrika*, **48**, 343–349. doi:10.1007/BF02294026.

Cox, T. F. and Cox, M. A. A. (2001). *Multidimensional Scaling*. Second edition. Chapman and Hall.

Gower, J. C. (1966). Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika*, **53**, 325–328. doi:10.2307/2333639.

Krzanowski, W. J. and Marriott, F. H. C. (1994). *Multivariate Analysis. Part I. Distributions, Ordination and Inference.* London: Edward Arnold. (Especially pp. 108–111.)

Mardia, K.V. (1978). Some properties of classical multidimensional scaling. *Communications on Statistics – Theory and Methods*, **A7**, 1233–41. doi:10.1080/03610927808827707

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979). Chapter 14 of *Multivariate Analysis*, London: Academic Press.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley.

Torgerson, W. S. (1958). *Theory and Methods of Scaling*. New York: Wiley.

See Also

`dist`.

`isoMDS` and `sammon` in package **MASS** provide alternative methods of multidimensional scaling.

Examples

```
require(graphics)

loc <- cmdscale(eurodist)
x <- loc[, 1]
y <- -loc[, 2] # reflect so North is at the top
## note asp = 1, to ensure Euclidean distances are represented correctly
plot(x, y, type = "n", xlab = "", ylab = "", asp = 1, axes = FALSE,
     main = "cmdscale(eurodist)")
text(x, y, rownames(loc), cex = 0.6)
```

coef *Extract Model Coefficients*

Description

coef is a generic function which extracts model coefficients from objects returned by modeling functions. coefficients is an *alias* for it.

Usage

```
coef(object, ...)
coefficients(object, ...)
## Default S3 method:
coef(object, complete = TRUE, ...)
## S3 method for class 'aov'
coef(object, complete = FALSE, ...)
```

Arguments

object	an object for which the extraction of model coefficients is meaningful.
complete	for the default (used for lm, etc) and aov methods: logical indicating if the full coefficient vector should be returned also in case of an over-determined system where some coefficients will be set to NA, see also alias. Note that the default *differs* for lm() and aov() results.
...	other arguments.

Details

All object classes which are returned by model fitting functions should provide a coef method or use the default one. (Note that the method is for coef and not coefficients.)

The "aov" method does not report aliased coefficients (see alias) by default where complete = FALSE.

The complete argument also exists for compatibility with vcov methods, and coef and aov methods for other classes should typically also keep the complete = * behavior in sync. By that, with p <- length(coef(obj, complete = TF)), dim(vcov(obj, complete = TF)) == c(p,p) will be fulfilled for both complete settings and the default.

Value

Coefficients extracted from the model object object.

For standard model fitting classes this will be a named numeric vector. For "maov" objects (produced by aov) it will be a matrix.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

fitted.values and residuals for related methods; glm, lm for model fitting.

Examples

```
x <- 1:5; coef(lm(c(1:3, 7, 6) ~ x))
```

---

complete.cases *Find Complete Cases*

Description

Return a logical vector indicating which cases are complete, i.e., have no missing values.

Usage

```
complete.cases(...)
```

Arguments

...	a sequence of vectors, matrices and data frames.

Value

A logical vector specifying which observations/rows have no missing values across the entire sequence.

Note

A current limitation of this function is that it uses low level functions to determine lengths and missingness, ignoring the class. This will lead to spurious errors when some columns have classes with length or is.na methods, for example "POSIXlt", as described in PR#16648.

See Also

is.na, na.omit, na.fail.

Examples

```
x <- airquality[, -1] # x is a regression design matrix
y <- airquality[,  1] # y is the corresponding response

stopifnot(complete.cases(y) != is.na(y))
ok <- complete.cases(x, y)
sum(!ok) # how many are not "ok" ?
x <- x[ok,]
y <- y[ok]
```

confint *Confidence Intervals for Model Parameters*

Description

Computes confidence intervals for one or more parameters in a fitted model. There is a default and a method for objects inheriting from class "lm".

Usage

```
confint(object, parm, level = 0.95, ...)
## Default S3 method:
confint(object, parm, level = 0.95, ...)
## S3 method for class 'lm'
confint(object, parm, level = 0.95, ...)
## S3 method for class 'glm'
confint(object, parm, level = 0.95, trace = FALSE, test=c("LRT", "Rao"), ...)
## S3 method for class 'nls'
confint(object, parm, level = 0.95, ...)
```

Arguments

object	a fitted model object.
parm	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level	the confidence level required.
trace	logical. Should profiling be traced?
test	use Likelihood Ratio or Rao Score test in profiling.
...	additional argument(s) for methods.

Details

confint is a generic function. The default method assumes normality, and needs suitable coef and vcov methods to be available. The default method can be called directly for comparison with other methods.

For objects of class "lm" the direct formulae based on t values are used.

Methods for classes "glm" and "nls" call the appropriate profile method, then find the confidence intervals by interpolation in the profile traces. If the profile object is already available it can be used as the main argument rather than the fitted model object itself.

Value

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as (1-level)/2 and 1 - (1-level)/2 in % (by default 2.5% and 97.5%).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

See Also

Original versions: `confint.glm` and `confint.nls` in package **MASS**.

Examples

```
fit <- lm(100/mpg ~ disp + hp + wt + am, data = mtcars)
confint(fit)
confint(fit, "wt")

## from example(glm)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3, 1, 9); treatment <- gl(3, 3)
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
confint(glm.D93)
confint.default(glm.D93)  # based on asymptotic normality
```

---

constrOptim *Linearly Constrained Optimization*

Description

Minimise a function subject to linear inequality constraints using an adaptive barrier algorithm.

Usage

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
            method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
            outer.iterations = 100, outer.eps = 1e-05, ...,
            hessian = FALSE)
```

Arguments

`theta`	numeric (vector) starting value (of length p): must be in the feasible region.
`f`	function to minimise (see below).
`grad`	gradient of `f` (a `function` as well), or `NULL` (see below).
`ui`	constraint matrix ($k \times p$), see below.
`ci`	constraint vector of length k (see below).
`mu`	(Small) tuning parameter.
`control`, `method`, `hessian`	passed to `optim`.
`outer.iterations`	iterations of the barrier algorithm.

`outer.eps` non-negative number; the relative convergence tolerance of the barrier algorithm.

`...` Other named arguments to be passed to `f` and `grad`: needs to be passed through `optim` so should not match its argument names.

Details

The feasible region is defined by `ui %*% theta - ci >= 0`. The starting value must be in the interior of the feasible region, but the minimum may be on the boundary.

A logarithmic barrier is added to enforce the constraints and then `optim` is called. The barrier function is chosen so that the objective function should decrease at each outer iteration. Minima in the interior of the feasible region are typically found quite quickly, but a substantial number of outer iterations may be needed for a minimum on the boundary.

The tuning parameter `mu` multiplies the barrier term. Its precise value is often relatively unimportant. As `mu` increases the augmented objective function becomes closer to the original objective function but also less smooth near the boundary of the feasible region.

Any `optim` method that permits infinite values for the objective function may be used (currently all but "L-BFGS-B").

The objective function `f` takes as first argument the vector of parameters over which minimisation is to take place. It should return a scalar result. Optional arguments `...` will be passed to `optim` and then (if not used by `optim`) to `f`. As with `optim`, the default is to minimise, but maximisation can be performed by setting `control$fnscale` to a negative value.

The gradient function `grad` must be supplied except with `method = "Nelder-Mead"`. It should take arguments matching those of `f` and return a vector containing the gradient.

Value

As for `optim`, but with two extra components: `barrier.value` giving the value of the barrier function at the optimum and `outer.iterations` gives the number of outer iterations (calls to `optim`). The `counts` component contains the *sum* of all `optim()$counts`.

References

K. Lange *Numerical Analysis for Statisticians.* Springer 2001, p185ff

See Also

`optim`, especially `method = "L-BFGS-B"` which does box-constrained optimisation.

Examples

```
## from optim
fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
```

```
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}

optim(c(-1.2,1), fr, grr)
#Box-constraint, optimum on the boundary
constrOptim(c(-1.2,0.9), fr, grr, ui = rbind(c(-1,0), c(0,-1)), ci = c(-1,-1))
#  x <= 0.9,  y - x > 0.1
constrOptim(c(.5,0), fr, grr, ui = rbind(c(-1,0), c(1,-1)), ci = c(-0.9,0.1))


## Solves linear and quadratic programming problems
## but needs a feasible starting value
#
# from example(solve.QP) in 'quadprog'
# no derivative
fQP <- function(b) {-sum(c(0,5,0)*b)+0.5*sum(b*b)}
Amat       <- matrix(c(-4,-3,0,2,1,0,0,-2,1), 3, 3)
bvec       <- c(-8, 2, 0)
constrOptim(c(2,-1,-1), fQP, NULL, ui = t(Amat), ci = bvec)
# derivative
gQP <- function(b) {-c(0, 5, 0) + b}
constrOptim(c(2,-1,-1), fQP, gQP, ui = t(Amat), ci = bvec)

## Now with maximisation instead of minimisation
hQP <- function(b) {sum(c(0,5,0)*b)-0.5*sum(b*b)}
constrOptim(c(2,-1,-1), hQP, NULL, ui = t(Amat), ci = bvec,
            control = list(fnscale = -1))
```

`contrast` *(Possibly Sparse) Contrast Matrices*

Description

Return a matrix of contrasts.

Usage

```
contr.helmert(n, contrasts = TRUE, sparse = FALSE)
contr.poly(n, scores = 1:n, contrasts = TRUE, sparse = FALSE)
contr.sum(n, contrasts = TRUE, sparse = FALSE)
contr.treatment(n, base = 1, contrasts = TRUE, sparse = FALSE)
contr.SAS(n, contrasts = TRUE, sparse = FALSE)
```

Arguments

`n`	a vector of levels for a factor, or the number of levels.

contrasts	a logical indicating whether contrasts should be computed.
sparse	logical indicating if the result should be sparse (of class dgCMatrix), using package **Matrix**.
scores	the set of values over which orthogonal polynomials are to be computed.
base	an integer specifying which group is considered the baseline group. Ignored if contrasts is FALSE.

Details

These functions are used for creating contrast matrices for use in fitting analysis of variance and regression models. The columns of the resulting matrices contain contrasts which can be used for coding a factor with n levels. The returned value contains the computed contrasts. If the argument contrasts is FALSE a square indicator matrix (the dummy coding) is returned **except** for contr.poly (which includes the 0-degree, i.e. constant, polynomial when contrasts = FALSE).

contr.helmert returns Helmert contrasts, which contrast the second level with the first, the third with the average of the first two, and so on. contr.poly returns contrasts based on orthogonal polynomials. contr.sum uses 'sum to zero contrasts'.

contr.treatment contrasts each level with the baseline level (specified by base): the baseline level is omitted. Note that this does not produce 'contrasts' as defined in the standard theory for linear models as they are not orthogonal to the intercept.

contr.SAS is a wrapper for contr.treatment that sets the base level to be the last level of the factor. The coefficients produced when using these contrasts should be equivalent to those produced by many (but not all) SAS procedures.

For consistency, sparse is an argument to all these contrast functions, however sparse = TRUE for contr.poly is typically pointless and is rarely useful for contr.helmert.

Value

A matrix with n rows and k columns, with k=n-1 if contrasts is TRUE and k=n if contrasts is FALSE.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

contrasts, C, and aov, glm, lm.

Examples

```
(cH <- contr.helmert(4))
apply(cH, 2, sum) # column sums are 0
crossprod(cH) # diagonal -- columns are orthogonal
contr.helmert(4, contrasts = FALSE) # just the 4 x 4 identity matrix

(cT <- contr.treatment(5))
```

```
all(crossprod(cT) == diag(4)) # TRUE: even orthonormal

(cT. <- contr.SAS(5))
all(crossprod(cT.) == diag(4)) # TRUE

zapsmall(cP <- contr.poly(3)) # Linear and Quadratic
zapsmall(crossprod(cP), digits = 15) # orthonormal up to fuzz
```

`contrasts` *Get and Set Contrast Matrices*

Description

Set and view the contrasts associated with a factor.

Usage

```
contrasts(x, contrasts = TRUE, sparse = FALSE)
contrasts(x, how.many = NULL) <- value
```

Arguments

`x`	a factor or a logical variable.
`contrasts`	logical. See 'Details'.
`sparse`	logical indicating if the result should be sparse (of class `dgCMatrix`), using package **Matrix**.
`how.many`	integer number indicating how many contrasts should be made. Defaults to one less than the number of levels of `x`. This need not be the same as the number of columns of `value`.
`value`	either a numeric matrix (or a sparse or dense matrix of a class extending `dMatrix` from package **Matrix**) whose columns give coefficients for contrasts in the levels of `x`, or (the quoted name of) a function which computes such matrices.

Details

If contrasts are not set for a factor the default functions from `options("contrasts")` are used.

A logical vector `x` is converted into a two-level factor with levels `c(FALSE, TRUE)` (regardless of which levels occur in the variable).

The argument `contrasts` is ignored if `x` has a matrix `contrasts` attribute set. Otherwise if `contrasts = TRUE` it is passed to a contrasts function such as `contr.treatment` and if `contrasts = FALSE` an identity matrix is returned. Suitable functions have a first argument which is the character vector of levels, a named argument `contrasts` (always called with `contrasts = TRUE`) and optionally a logical argument `sparse`.

If `value` supplies more than `how.many` contrasts, the first `how.many` are used. If too few are supplied, a suitable contrast matrix is created by extending `value` after ensuring its columns are contrasts (orthogonal to the constant term) and not collinear.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

C, contr.helmert, contr.poly, contr.sum, contr.treatment; glm, aov, lm.

Examples

```
utils::example(factor)
fff <- ff[, drop = TRUE]  # reduce to 5 levels.
contrasts(fff) # treatment contrasts by default
contrasts(C(fff, sum))
contrasts(fff, contrasts = FALSE) # the 5x5 identity matrix

contrasts(fff) <- contr.sum(5); contrasts(fff)  # set sum contrasts
contrasts(fff, 2) <- contr.sum(5); contrasts(fff)  # set 2 contrasts
# supply 2 contrasts, compute 2 more to make full set of 4.
contrasts(fff) <- contr.sum(5)[, 1:2]; contrasts(fff)

## using sparse contrasts: % useful, once model.matrix() works with these :
ffs <- fff
contrasts(ffs) <- contr.sum(5, sparse = TRUE)[, 1:2]; contrasts(ffs)
stopifnot(all.equal(ffs, fff))
contrasts(ffs) <- contr.sum(5, sparse = TRUE); contrasts(ffs)
```

---

convolve *Convolution of Sequences via FFT*

Description

Use the Fast Fourier Transform to compute the several kinds of convolutions of two sequences.

Usage

```
convolve(x, y, conj = TRUE, type = c("circular", "open", "filter"))
```

Arguments

x, y	numeric sequences *of the same length* to be convolved.
conj	logical; if TRUE, take the complex *conjugate* before back-transforming (default, and used for usual convolution).
type	character; partially matched to "circular", "open", "filter". For "circular", the two sequences are treated as *circular*, i.e., periodic. For "open" and "filter", the sequences are padded with 0s (from left and right) first; "filter" returns the middle sub-vector of "open", namely, the result of running a weighted mean of x with weights y.

Details

The Fast Fourier Transform, `fft`, is used for efficiency.

The input sequences `x` and `y` must have the same length if `circular` is true.

Note that the usual definition of convolution of two sequences `x` and `y` is given by `convolve(x, rev(y), type = "o")`.

Value

If `r <- convolve(x, y, type = "open")` and `n <- length(x)`, `m <- length(y)`, then

$$r_k = \sum_i x_{k-m+i} y_i$$

where the sum is over all valid indices i, for $k = 1, \ldots, n + m - 1$.

If `type == "circular"`, $n = m$ is required, and the above is true for $i, k = 1, \ldots, n$ when $x_j := x_{n+j}$ for $j < 1$.

References

Brillinger, D. R. (1981) *Time Series: Data Analysis and Theory*, Second Edition. San Francisco: Holden-Day.

See Also

`fft`, `nextn`, and particularly `filter` (from the **stats** package) which may be more appropriate.

Examples

```
require(graphics)

x <- c(0,0,0,100,0,0,0)
y <- c(0,0,1, 2 ,1,0,0)/4
zapsmall(convolve(x, y))         #  *NOT* what you first thought.
zapsmall(convolve(x, y[3:5], type = "f")) # rather
x <- rnorm(50)
y <- rnorm(50)
# Circular convolution *has* this symmetry:
all.equal(convolve(x, y, conj = FALSE), rev(convolve(rev(y),x)))

n <- length(x <- -20:24)
y <- (x-10)^2/1000 + rnorm(x)/8

Han <- function(y) # Hanning
       convolve(y, c(1,2,1)/4, type = "filter")

plot(x, y, main = "Using  convolve(.) for Hanning filters")
lines(x[-c(1  , n)      ], Han(y), col = "red")
lines(x[-c(1:2, (n-1):n)], Han(Han(y)), lwd = 2, col = "dark blue")
```

cophenetic *Cophenetic Distances for a Hierarchical Clustering*

Description

Computes the cophenetic distances for a hierarchical clustering.

Usage

```
cophenetic(x)
## Default S3 method:
cophenetic(x)
## S3 method for class 'dendrogram'
cophenetic(x)
```

Arguments

x	an R object representing a hierarchical clustering. For the default method, an object of class "hclust" or with a method for as.hclust() such as "agnes" in package **cluster**.

Details

The cophenetic distance between two observations that have been clustered is defined to be the intergroup dissimilarity at which the two observations are first combined into a single cluster. Note that this distance has many ties and restrictions.

It can be argued that a dendrogram is an appropriate summary of some data if the correlation between the original distances and the cophenetic distances is high. Otherwise, it should simply be viewed as the description of the output of the clustering algorithm.

cophenetic is a generic function. Support for classes which represent hierarchical clusterings (total indexed hierarchies) can be added by providing an as.hclust() or, more directly, a cophenetic() method for such a class.

The method for objects of class "dendrogram" requires that all leaves of the dendrogram object have non-null labels.

Value

An object of class "dist".

Author(s)

Robert Gentleman

References

Sneath, P.H.A. and Sokal, R.R. (1973) *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, p. 278 ff; Freeman, San Francisco.

See Also

dist, hclust

Examples

```
require(graphics)

d1 <- dist(USArrests)
hc <- hclust(d1, "ave")
d2 <- cophenetic(hc)
cor(d1, d2) # 0.7659

## Example from Sneath & Sokal, Fig. 5-29, p.279
d0 <- c(1,3.8,4.4,5.1, 4,4.2,5, 2.6,5.3, 5.4)
attributes(d0) <- list(Size = 5, diag = TRUE)
class(d0) <- "dist"
names(d0) <- letters[1:5]
d0
utils::str(upgma <- hclust(d0, method = "average"))
plot(upgma, hang = -1)
#
(d.coph <- cophenetic(upgma))
cor(d0, d.coph) # 0.9911
```

---

cor *Correlation, Variance and Covariance (Matrices)*

Description

var, cov and cor compute the variance of x and the covariance or correlation of x and y if these are vectors. If x and y are matrices then the covariances (or correlations) between the columns of x and the columns of y are computed.

cov2cor scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

x	a numeric vector, matrix or data frame.
y	NULL (default) or a vector, matrix or data frame with compatible dimensions to x. The default is equivalent to y = x (but more efficient).
na.rm	logical. Should missing values be removed?
use	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise.complete.obs".
method	a character string indicating which correlation coefficient (or covariance) is to be computed. One of "pearson" (default), "kendall", or "spearman": can be abbreviated.
V	symmetric numeric matrix, usually positive definite such as a covariance matrix.

Details

For cov and cor one must *either* give a matrix or data frame for x *or* give both x and y.

The inputs must be numeric (as determined by is.numeric: logical values are also allowed for historical compatibility): the "kendall" and "spearman" methods make sense for ordered inputs but xtfrm can be used to find a suitable prior transformation to numbers.

var is just another interface to cov, where na.rm is used to determine the default for use when that is unspecified. If na.rm is TRUE then the complete observations (rows) are used (use = "na.or.complete") to compute the variance. Otherwise, by default use = "everything".

If use is "everything", NAs will propagate conceptually, i.e., a resulting value will be NA whenever one of its contributing observations is NA.
If use is "all.obs", then the presence of missing observations will produce an error. If use is "complete.obs" then missing values are handled by casewise deletion (and if there are no complete cases, that gives an error).
"na.or.complete" is the same unless there are no complete cases, that gives NA. Finally, if use has the value "pairwise.complete.obs" then the correlation or covariance between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or correlation matrices which are not positive semi-definite, as well as NA entries if there are no complete pairs for that pair of variables. For cov and var, "pairwise.complete.obs" only works with the "pearson" method. Note that (the equivalent of) var(double(0), use = *) gives NA for use = "everything" and "na.or.complete", and gives an error in the other cases.

The denominator $n - 1$ is used which gives an unbiased estimator of the (co)variance for i.i.d. observations. These functions return NA when there is only one observation.

For cor(), if method is "kendall" or "spearman", Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These are more robust and have been recommended if the data do not necessarily come from a bivariate normal distribution.
For cov(), a non-Pearson method is unusual but available for the sake of completeness. Note that "spearman" basically computes cor(R(x), R(y)) (or cov(., .)) where R(u) := rank(u, na.last = "keep"). In the case of missing values, the ranks are calculated depending on the value of use, either based on complete observations, or based on pairwise completeness with reranking for each pair.

When there are ties, Kendall's τ_b is computed, as proposed by Kendall (1945).

Scaling a covariance matrix into a correlation one can be achieved in many ways, mathematically most appealing by multiplication with a diagonal matrix from left and right, or more efficiently by using `sweep(.., FUN = "/")` twice. The `cov2cor` function is even a bit more efficient, and provided mostly for didactical reasons.

Value

For `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(abs(r) <= 1)`.

Note

Some people have noted that the code for Kendall's tau is slow for very large datasets (many more than 1000 cases). It rarely makes sense to do such a computation, but see function `cor.fk` in package **pcaPP**.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Kendall, M. G. (1938). A new measure of rank correlation, *Biometrika*, **30**, 81–93. doi:10.1093/biomet/30.12.81.

Kendall, M. G. (1945). The treatment of ties in rank problems. *Biometrika*, **33** 239–251. doi:10.1093/biomet/33.3.239

See Also

`cor.test` for confidence intervals (and tests).

`cov.wt` for *weighted* covariance computation.

`sd` for standard deviation (vectors).

Examples

```
var(1:10)  # 9.166667

var(1:5, 1:5) # 2.5

## Two simple vectors
cor(1:10, 2:11) # == 1

## Correlation Matrix of Multivariate sample:
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated

## Spearman's rho  and  Kendall's tau
symnum(clS <- cor(longley, method = "spearman"))
symnum(clK <- cor(longley, method = "kendall"))
## How much do they differ?
```

```
i <- lower.tri(Cl)
cor(cbind(P = Cl[i], S = clS[i], K = clK[i]))


## cov2cor() scales a covariance matrix by its diagonal
##           to become the correlation matrix.
cov2cor # see the function definition {and learn ..}
stopifnot(all.equal(Cl, cov2cor(cov(longley))),
          all.equal(cor(longley, method = "kendall"),
            cov2cor(cov(longley, method = "kendall"))))

##--- Missing value treatment:
C1 <- cov(swiss)
range(eigen(C1, only.values = TRUE)$values) # 6.19        1921

## swM := "swiss" with  3 "missing"s :
swM <- swiss
colnames(swM) <- abbreviate(colnames(swiss), minlength=6)
swM[1,2] <- swM[7,3] <- swM[25,5] <- NA # create 3 "missing"

## Consider all 5 "use" cases :
(C. <- cov(swM)) # use="everything"  quite a few NA's in cov.matrix
try(cov(swM, use = "all")) # Error: missing obs...
C2 <- cov(swM, use = "complete")
stopifnot(identical(C2, cov(swM, use = "na.or.complete")))
range(eigen(C2, only.values = TRUE)$values) # 6.46   1930
C3 <- cov(swM, use = "pairwise")
range(eigen(C3, only.values = TRUE)$values) # 6.19   1938

## Kendall's tau doesn't change much:
symnum(Rc <- cor(swM, method = "kendall", use = "complete"))
symnum(Rp <- cor(swM, method = "kendall", use = "pairwise"))
symnum(R. <- cor(swiss, method = "kendall"))

## "pairwise" is closer componentwise,
summary(abs(c(1 - Rp/R.)))
summary(abs(c(1 - Rc/R.)))

## but "complete" is closer in Eigen space:
EV <- function(m) eigen(m, only.values=TRUE)$values
summary(abs(1 - EV(Rp)/EV(R.)) / abs(1 - EV(Rc)/EV(R.)))
```

`cor.test` *Test for Association/Correlation Between Paired Samples*

Description

Test for association between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's τ or Spearman's ρ.

Usage

```
cor.test(x, ...)

## Default S3 method:
cor.test(x, y,
         alternative = c("two.sided", "less", "greater"),
         method = c("pearson", "kendall", "spearman"),
         exact = NULL, conf.level = 0.95, continuity = FALSE, ...)

## S3 method for class 'formula'
cor.test(formula, data, subset, na.action, ...)
```

Arguments

x, y	numeric vectors of data values. x and y must have the same length.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter. "greater" corresponds to positive association, "less" to negative association.
method	a character string indicating which correlation coefficient is to be used for the test. One of "pearson", "kendall", or "spearman", can be abbreviated.
exact	a logical indicating whether an exact p-value should be computed. Used for Kendall's τ and Spearman's ρ. See 'Details' for the meaning of NULL (the default).
conf.level	confidence level for the returned confidence interval. Currently only used for the Pearson product moment correlation coefficient if there are at least 4 complete pairs of observations.
continuity	logical: if true, a continuity correction is used for Kendall's τ and Spearman's ρ when not computed exactly.
formula	a formula of the form ~ u + v, where each of u and v are numeric variables giving the data values for one sample. The samples must be of the same length.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The three methods each estimate the association between paired samples and compute a test of the value being zero. They use different measures of association, all in the range $[-1, 1]$ with 0 indicating no association. These are sometimes referred to as tests of no *correlation*, but that term is often confined to the default method.

If `method` is `"pearson"`, the test statistic is based on Pearson's product moment correlation coefficient `cor(x, y)` and follows a t distribution with `length(x)-2` degrees of freedom if the samples follow independent normal distributions. If there are at least 4 complete pairs of observation, an asymptotic confidence interval is given based on Fisher's Z transform.

If `method` is `"kendall"` or `"spearman"`, Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.

For Kendall's test, by default (if `exact` is NULL), an exact p-value is computed if there are less than 50 paired samples containing finite values and there are no ties. Otherwise, the test statistic is the estimate scaled to zero mean and unit variance, and is approximately normally distributed.

For Spearman's test, p-values are computed using algorithm AS 89 for $n < 1290$ and `exact = TRUE`, otherwise via the asymptotic t approximation. Note that these are 'exact' for $n < 10$, and use an Edgeworth series approximation for larger sample sizes (the cutoff has been changed from the original paper).

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the degrees of freedom of the test statistic in the case that it follows a t distribution.
`p.value`	the p-value of the test.
`estimate`	the estimated measure of association, with name `"cor"`, `"tau"`, or `"rho"` corresponding to the method employed.
`null.value`	the value of the association measure under the null hypothesis, always `0`.
`alternative`	a character string describing the alternative hypothesis.
`method`	a character string indicating how the association was measured.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the measure of association. Currently only given for Pearson's product moment correlation coefficient in case of at least 4 complete pairs of observations.

References

D. J. Best & D. E. Roberts (1975). Algorithm AS 89: The Upper Tail Probabilities of Spearman's ρ. *Applied Statistics*, **24**, 377–379. doi:10.2307/2347111.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 185–194 (Kendall and Spearman tests).

See Also

`Kendall` in package **Kendall**.

`pKendall` and `pSpearman` in package **SuppDists**, `spearman.test` in package **pspearman**, which supply different (and often more accurate) approximations.

Examples

```
## Hollander & Wolfe (1973), p. 187f.
## Assessment of tuna quality.  We compare the Hunter L measure of
##  lightness to the averages of consumer panel scores (recoded as
##  integer values from 1 to 6 and averaged over 80 such values) in
##  9 lots of canned tuna.

x <- c(44.4, 45.9, 41.9, 53.3, 44.7, 44.1, 50.7, 45.2, 60.1)
y <- c( 2.6,  3.1,  2.5,  5.0,  3.6,  4.0,  5.2,  2.8,  3.8)

##  The alternative hypothesis of interest is that the
##  Hunter L value is positively associated with the panel score.

cor.test(x, y, method = "kendall", alternative = "greater")
## => p=0.05972

cor.test(x, y, method = "kendall", alternative = "greater",
         exact = FALSE) # using large sample approximation
## => p=0.04765

## Compare this to
cor.test(x, y, method = "spearm", alternative = "g")
cor.test(x, y,                    alternative = "g")

## Formula interface.
require(graphics)
pairs(USJudgeRatings)
cor.test(~ CONT + INTG, data = USJudgeRatings)
```

`cov.wt` *Weighted Covariance Matrices*

Description

Returns a list containing estimates of the weighted covariance matrix and the mean of the data, and optionally of the (weighted) correlation matrix.

Usage

```
cov.wt(x, wt = rep(1/nrow(x), nrow(x)), cor = FALSE, center = TRUE,
       method = c("unbiased", "ML"))
```

Arguments

`x`	a matrix or data frame. As usual, rows are observations and columns are variables.
`wt`	a non-negative and non-zero vector of weights for each observation. Its length must equal the number of rows of `x`.

cor	a logical indicating whether the estimated correlation weighted matrix will be returned as well.
center	either a logical or a numeric vector specifying the centers to be used when computing covariances. If TRUE, the (weighted) mean of each variable is used, if FALSE, zero is used. If center is numeric, its length must equal the number of columns of x.
method	string specifying how the result is scaled, see 'Details' below. Can be abbreviated.

Details

By default, method = "unbiased", The covariance matrix is divided by one minus the sum of squares of the weights, so if the weights are the default $(1/n)$ the conventional unbiased estimate of the covariance matrix with divisor $(n - 1)$ is obtained.

Value

A list containing the following named components:

cov	the estimated (weighted) covariance matrix
center	an estimate for the center (mean) of the data.
n.obs	the number of observations (rows) in x.
wt	the weights used in the estimation. Only returned if given as an argument.
cor	the estimated correlation matrix. Only returned if cor is TRUE.

See Also

cov and var.

Examples

```
(xy <- cbind(x = 1:10, y = c(1:3, 8:5, 8:10)))
w1 <- c(0,0,0,1,1,1,1,1,0,0)
cov.wt(xy, wt = w1) # i.e. method = "unbiased"
cov.wt(xy, wt = w1, method = "ML", cor = TRUE)
```

---

cpgram *Plot Cumulative Periodogram*

Description

Plots a cumulative periodogram.

Usage

```
cpgram(ts, taper = 0.1,
       main = paste("Series: ", deparse1(substitute(ts))),
       ci.col = "blue")
```

Arguments

ts	a univariate time series
taper	proportion tapered in forming the periodogram
main	main title
ci.col	colour for confidence band.

Value

None.

Side Effects

Plots the cumulative periodogram in a square plot.

Note

From package **MASS**.

Author(s)

B.D. Ripley

Examples

```
require(graphics)

par(pty = "s", mfrow = c(1,2))
cpgram(lh)
lh.ar <- ar(lh, order.max = 9)
cpgram(lh.ar$resid, main = "AR(3) fit to lh")

cpgram(ldeaths)
```

---

cutree *Cut a Tree into Groups of Data*

Description

Cuts a tree, e.g., as resulting from hclust, into several groups either by specifying the desired number(s) of groups or the cut height(s).

Usage

```
cutree(tree, k = NULL, h = NULL)
```

Arguments

tree	a tree as produced by hclust. cutree() only expects a list with components merge, height, and labels, of appropriate content each.
k	an integer scalar or vector with the desired number of groups
h	numeric scalar or vector with heights where the tree should be cut.

At least one of k or h must be specified, k overrides h if both are given.

Details

Cutting trees at a given height is only possible for ultrametric trees (with monotone clustering heights).

Value

cutree returns a vector with group memberships if k or h are scalar, otherwise a matrix with group memberships is returned where each column corresponds to the elements of k or h, respectively (which are also used as column names).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

hclust, dendrogram for cutting trees themselves.

Examples

```
hc <- hclust(dist(USArrests))

cutree(hc, k = 1:5) #k = 1 is trivial
cutree(hc, h = 250)

## Compare the 2 and 4 grouping:
g24 <- cutree(hc, k = c(2,4))
table(grp2 = g24[,"2"], grp4 = g24[,"4"])
```

---

decompose *Classical Seasonal Decomposition by Moving Averages*

Description

Decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.

Usage

```
decompose(x, type = c("additive", "multiplicative"), filter = NULL)
```

Arguments

x	A time series.
type	The type of seasonal component. Can be abbreviated.
filter	A vector of filter coefficients in reverse time order (as for AR or MA coefficients), used for filtering out the seasonal component. If NULL, a moving average with symmetric window is performed.

Details

The additive model used is:

$$Y_t = T_t + S_t + e_t$$

The multiplicative model used is:

$$Y_t = T_t \, S_t \, e_t$$

The function first determines the trend component using a moving average (if filter is NULL, a symmetric window with equal weights is used), and removes it from the time series. Then, the seasonal figure is computed by averaging, for each time unit, over all periods. The seasonal figure is then centered. Finally, the error component is determined by removing trend and seasonal figure (recycled as needed) from the original time series.

This only works well if x covers an integer number of complete periods.

Value

An object of class "decomposed.ts" with following components:

x	The original series.
seasonal	The seasonal component (i.e., the repeated seasonal figure).
figure	The estimated seasonal figure only.
trend	The trend component.
random	The remainder part.
type	The value of type.

Note

The function stl provides a much more sophisticated decomposition.

Author(s)

David Meyer <David.Meyer@wu.ac.at>

References

M. Kendall and A. Stuart (1983) *The Advanced Theory of Statistics*, Vol.3, Griffin. pp. 410–414.

See Also

stl

Examples

```
require(graphics)

m <- decompose(co2)
m$figure
plot(m)

## example taken from Kendall/Stuart
x <- c(-50, 175, 149, 214, 247, 237, 225, 329, 729, 809,
       530, 489, 540, 457, 195, 176, 337, 239, 128, 102, 232, 429, 3,
       98, 43, -141, -77, -13, 125, 361, -45, 184)
x <- ts(x, start = c(1951, 1), end = c(1958, 4), frequency = 4)
m <- decompose(x)
## seasonal figure: 6.25, 8.62, -8.84, -6.03
round(decompose(x)$figure / 10, 2)
```

---

delete.response *Modify Terms Objects*

Description

delete.response returns a terms object for the same model but with no response variable.

drop.terms removes variables from the right-hand side of the model. There is also a "[.terms" method to perform the same function (with keep.response = TRUE).

reformulate creates a formula from a character vector. If length(termlabels) > 1, its elements are concatenated with +. Non-syntactic names (e.g. containing spaces or special characters; see make.names) must be protected with backticks (see examples). A non-parseable response still works for now, back compatibly, with a deprecation warning.

Usage

```
delete.response(termobj)

reformulate(termlabels, response = NULL, intercept = TRUE, env = parent.frame())

drop.terms(termobj, dropx = NULL, keep.response = FALSE)
```

Arguments

termobj	A terms object
termlabels	character vector giving the right-hand side of a model formula. Cannot be zero-length.

response	character string, symbol or call giving the left-hand side of a model formula, or NULL.
intercept	logical: should the formula have an intercept?
env	the environment of the formula returned.
dropx	vector of positions of variables to drop from the right-hand side of the model.
keep.response	Keep the response in the resulting object?

Value

delete.response and drop.terms return a terms object.

reformulate returns a formula.

See Also

terms

Examples

```
ff <- y ~ z + x + w
tt <- terms(ff)
tt
delete.response(tt)
drop.terms(tt, 2:3, keep.response = TRUE)
tt[-1]
tt[2:3]
reformulate(attr(tt, "term.labels"))

## keep LHS :
reformulate("x*w", ff[[2]])
fS <- surv(ft, case) ~ a + b
reformulate(c("a", "b*f"), fS[[2]])

## using non-syntactic names:
reformulate(c("`P/E`", "`% Growth`"), response = as.name("+-"))

x <- c("a name", "another name")
tryCatch( reformulate(x), error = function(e) "Syntax error." )
## rather backquote the strings in x :
reformulate(sprintf("`%s`", x))

stopifnot(identical(      ~ var, reformulate("var")),
          identical(~ a + b + c, reformulate(letters[1:3])),
          identical(  y ~ a + b, reformulate(letters[1:2], "y"))
         )
```

dendrapply *Apply a Function to All Nodes of a Dendrogram*

Description

Apply function FUN to each node of a dendrogram recursively. When y <- dendrapply(x, fn), then y is a dendrogram of the same graph structure as x and for each node, y.node[j] <- FUN(x.node[j], ...) (where y.node[j] is an (invalid!) notation for the j-th node of y).

Usage

```
dendrapply(X, FUN, ...)
```

Arguments

X	an object of class "dendrogram".
FUN	an R function to be applied to each dendrogram node, typically working on its attributes alone, returning an altered version of the same node.
...	potential further arguments passed to FUN.

Value

Usually a dendrogram of the same (graph) structure as X. For that, the function must be conceptually of the form FUN <- function(X) { attributes(X) <- .....; X }, i.e., returning the node with some attributes added or changed.

Note

The implementation is somewhat experimental and suggestions for enhancements (or nice examples of usage) are very welcome. The current implementation is *recursive* and inefficient for dendrograms with many non-leaves. See the 'Warning' in dendrogram.

Author(s)

Martin Maechler

See Also

as.dendrogram, lapply for applying a function to each component of a list, rapply for doing so to each non-list component of a nested list.

Examples

```
require(graphics)

## a smallish simple dendrogram
dhc <- as.dendrogram(hc <- hclust(dist(USArrests), "ave"))
(dhc21 <- dhc[[2]][[1]])

## too simple:
dendrapply(dhc21, function(n) utils::str(attributes(n)))

## toy example to set colored leaf labels :
local({
  colLab <<- function(n) {
      if(is.leaf(n)) {
        a <- attributes(n)
        i <<- i+1
        attr(n, "nodePar") <-
            c(a$nodePar, list(lab.col = mycols[i], lab.font = i%%3))
      }
      n
  }
  mycols <- grDevices::rainbow(attr(dhc21,"members"))
  i <- 0
 })
dL <- dendrapply(dhc21, colLab)
op <- par(mfrow = 2:1)
 plot(dhc21)
 plot(dL) ## --> colored labels!
par(op)
```

---

dendrogram *General Tree Structures*

Description

Class "dendrogram" provides general functions for handling tree-like structures. It is intended as a replacement for similar functions in hierarchical clustering and classification/regression trees, such that all of these can use the same engine for plotting or cutting trees.

Usage

```
as.dendrogram(object, ...)
## S3 method for class 'hclust'
as.dendrogram(object, hang = -1, check = TRUE, ...)

## S3 method for class 'dendrogram'
as.hclust(x, ...)

## S3 method for class 'dendrogram'
```

```
plot(x, type = c("rectangle", "triangle"),
      center = FALSE,
      edge.root = is.leaf(x) || !is.null(attr(x,"edgetext")),
      nodePar = NULL, edgePar = list(),
      leaflab = c("perpendicular", "textlike", "none"),
      dLeaf = NULL, xlab = "", ylab = "", xaxt = "n", yaxt = "s",
      horiz = FALSE, frame.plot = FALSE, xlim, ylim, ...)

## S3 method for class 'dendrogram'
cut(x, h, ...)

## S3 method for class 'dendrogram'
merge(x, y, ..., height,
      adjust = c("auto", "add.max", "none"))

## S3 method for class 'dendrogram'
nobs(object, ...)

## S3 method for class 'dendrogram'
print(x, digits, ...)

## S3 method for class 'dendrogram'
rev(x)

## S3 method for class 'dendrogram'
str(object, max.level = NA, digits.d = 3,
    give.attr = FALSE, wid = getOption("width"),
    nest.lev = 0, indent.str = "",
    last.str = getOption("str.dendrogram.last"), stem = "--",
    ...)

is.leaf(object)
```

Arguments

object	any R object that can be made into one of class "dendrogram".
x, y	object(s) of class "dendrogram".
hang	numeric scalar indicating how the *height* of leaves should be computed from the heights of their parents; see plot.hclust.
check	logical indicating if object should be checked for validity. This check is not necessary when x is known to be valid such as when it is the direct result of hclust(). The default is check=TRUE, e.g. for protecting against memory explosion with invalid inputs.
type	type of plot.
center	logical; if TRUE, nodes are plotted centered with respect to the leaves in the branch. Otherwise (default), plot them in the middle of all direct child nodes.
edge.root	logical; if true, draw an edge to the root node.

nodePar — a list of plotting parameters to use for the nodes (see points) or NULL by default which does not draw symbols at the nodes. The list may contain components named pch, cex, col, xpd, and/or bg each of which can have length two for specifying separate attributes for *inner* nodes and *leaves*. Note that the default of pch is 1:2, so you may want to use pch = NA if you specify nodePar.

edgePar — a list of plotting parameters to use for the edge segments and labels (if there's an edgetext). The list may contain components named col, lty and lwd (for the segments), p.col, p.lwd, and p.lty (for the polygon around the text) and t.col for the text color. As with nodePar, each can have length two for differentiating leaves and inner nodes.

leaflab — a string specifying how leaves are labeled. The default "perpendicular" write text vertically (by default).
"textlike" writes text horizontally (in a rectangle), and
"none" suppresses leaf labels.

dLeaf — a number specifying the **d**istance in user coordinates between the tip of a leaf and its label. If NULL as per default, 3/4 of a letter width or height is used.

horiz — logical indicating if the dendrogram should be drawn *horizontally* or not.

frame.plot — logical indicating if a box around the plot should be drawn, see plot.default.

h — height at which the tree is cut.

height — height at which the two dendrograms should be merged. If not specified (or NULL), the default is ten percent larger than the (larger of the) two component heights.

adjust — a string determining if the leaf values should be adjusted. The default, "auto", checks if the (first) two dendrograms both start at 1; if they do, "add.max" is chosen, which adds the maximum of the previous dendrogram leaf values to each leaf of the "next" dendrogram. Specifying adjust to another value skips the check and hence is a tad more efficient.

xlim, ylim — optional x- and y-limits of the plot, passed to plot.default. The defaults for these show the full dendrogram.

..., xlab, ylab, xaxt, yaxt — graphical parameters, or arguments for other methods.

digits — integer specifying the precision for printing, see print.default.

max.level, digits.d, give.attr, wid, nest.lev, indent.str — arguments to str, see str.default(). Note that give.attr = FALSE still shows height and members attributes for each node.

last.str, stem — strings used for str() specifying how the last branch (at each level) should start and the *stem* to use for each dendrogram branch. In some environments, using last.str = "'" will provide much nicer looking output, than the historical default last.str = "`".

Details

The dendrogram is directly represented as a nested list where each component corresponds to a branch of the tree. Hence, the first branch of tree z is z[[1]], the second branch of the corresponding subtree is z[[1]][[2]], or shorter z[[c(1,2)]], etc.. Each node of the tree carries some

information needed for efficient plotting or cutting as attributes, of which only `members`, `height` and `leaf` for leaves are compulsory:

`members` total number of leaves in the branch

`height` numeric non-negative height at which the node is plotted.

`midpoint` numeric horizontal distance of the node from the left border (the leftmost leaf) of the branch (unit 1 between all leaves). This is used for `plot(*, center = FALSE)`.

`label` character; the label of the node

`x.member` for `cut()$upper`, the number of *former* members; more generally a substitute for the `members` component used for 'horizontal' (when `horiz = FALSE`, else 'vertical') alignment.

`edgetext` character; the label for the edge leading to the node

`nodePar` a named list (of length-1 components) specifying node-specific attributes for `points` plotting, see the `nodePar` argument above.

`edgePar` a named list (of length-1 components) specifying attributes for `segments` plotting of the edge leading to the node, and drawing of the `edgetext` if available, see the `edgePar` argument above.

`leaf` logical, if `TRUE`, the node is a leaf of the tree.

`cut.dendrogram()` returns a list with components `$upper` and `$lower`, the first is a truncated version of the original tree, also of class `dendrogram`, the latter a list with the branches obtained from cutting the tree, each a `dendrogram`.

There are `[[`, `print`, and `str` methods for `"dendrogram"` objects where the first one (extraction) ensures that selecting sub-branches keeps the class, i.e., returns a dendrogram even if only a leaf. On the other hand, `[` (*single* bracket) extraction returns the underlying list structure.

Objects of class `"hclust"` can be converted to class `"dendrogram"` using method `as.dendrogram()`, and since R 2.13.0, there is also a `as.hclust()` method as an inverse.

`rev.dendrogram` simply returns the dendrogram `x` with reversed nodes, see also `reorder.dendrogram`.

The `merge(x, y, ...)` method merges two or more dendrograms into a new one which has `x` and `y` (and optional further arguments) as branches. Note that before R 3.1.2, `adjust = "none"` was used implicitly, which is invalid when, e.g., the dendrograms are from `as.dendrogram(hclust(..))`.

`nobs(object)` returns the total number of leaves (the `members` attribute, see above).

`is.leaf(object)` returns logical indicating if `object` is a leaf (the most simple dendrogram).

`plotNode()` and `plotNodeLimit()` are helper functions.

Warning

Some operations on dendrograms such as `merge()` make use of recursion. For deep trees it may be necessary to increase `options("expressions")`: if you do, you are likely to need to set the C stack size (`Cstack_info()[["size"]]`) larger than the default where possible.

Note

`plot()`: When using `type = "triangle"`, `center = TRUE` often looks better.

`str(d)`: If you really want to see the *internal* structure, use `str(unclass(d))` instead.

See Also

dendrapply for applying a function to *each* node. order.dendrogram and reorder.dendrogram; further, the labels method.

Examples

```
require(graphics); require(utils)

hc <- hclust(dist(USArrests), "ave")
(dend1 <- as.dendrogram(hc)) # "print()" method
str(dend1)          # "str()" method
str(dend1, max.level = 2, last.str =  "'") # only the first two sub-levels
oo <- options(str.dendrogram.last = "\\") # yet another possibility
str(dend1, max.level = 2) # only the first two sub-levels
options(oo)  # .. resetting them

op <- par(mfrow =  c(2,2), mar = c(5,2,1,4))
plot(dend1)
## "triangle" type and show inner nodes:
plot(dend1, nodePar = list(pch = c(1,NA), cex = 0.8, lab.cex = 0.8),
      type = "t", center = TRUE)
plot(dend1, edgePar = list(col = 1:2, lty = 2:3),
     dLeaf = 1, edge.root = TRUE)
plot(dend1, nodePar = list(pch = 2:1, cex = .4*2:1, col = 2:3),
     horiz = TRUE)

## simple test for as.hclust() as the inverse of as.dendrogram():
stopifnot(identical(as.hclust(dend1)[1:4], hc[1:4]))

dend2 <- cut(dend1, h = 70)
## leaves are wrong horizontally in R 4.0 and earlier:
plot(dend2$upper)
plot(dend2$upper, nodePar = list(pch = c(1,7), col = 2:1))
##  dend2$lower is *NOT* a dendrogram, but a list of .. :
plot(dend2$lower[[3]], nodePar = list(col = 4), horiz = TRUE, type = "tr")
## "inner" and "leaf" edges in different type & color :
plot(dend2$lower[[2]], nodePar = list(col = 1),   # non empty list
     edgePar = list(lty = 1:2, col = 2:1), edge.root = TRUE)
par(op)
d3 <- dend2$lower[[2]][[2]][[1]]
stopifnot(identical(d3, dend2$lower[[2]][[c(2,1)]]))
str(d3, last.str = "'")

## to peek at the inner structure "if you must", use '[..]' indexing :
str(d3[2][[1]]) ## or the full
str(d3[])

## merge() to join dendrograms:
(d13 <- merge(dend2$lower[[1]], dend2$lower[[3]]))
## merge() all parts back (using default 'height' instead of original one):
den.1 <- Reduce(merge, dend2$lower)
## or merge() all four parts at same height --> 4 branches (!)
```

```
d. <- merge(dend2$lower[[1]], dend2$lower[[2]], dend2$lower[[3]],
            dend2$lower[[4]])
## (with a warning) or the same using  do.call :
stopifnot(identical(d., do.call(merge, dend2$lower)))
plot(d., main = "merge(d1, d2, d3, d4)  |->  dendrogram with a 4-split")

## "Zoom" in to the first dendrogram :
plot(dend1, xlim = c(1,20), ylim = c(1,50))

nP <- list(col = 3:2, cex = c(2.0, 0.75), pch =  21:22,
           bg =  c("light blue", "pink"),
           lab.cex = 0.75, lab.col = "tomato")
plot(d3, nodePar= nP, edgePar = list(col = "gray", lwd = 2), horiz = TRUE)
addE <- function(n) {
      if(!is.leaf(n)) {
        attr(n, "edgePar") <- list(p.col = "plum")
        attr(n, "edgetext") <- paste(attr(n,"members"),"members")
      }
      n
}
d3e <- dendrapply(d3, addE)
plot(d3e, nodePar =  nP)
plot(d3e, nodePar =  nP, leaflab = "textlike")
```

`density` *Kernel Density Estimation*

Description

The (S3) generic function `density` computes kernel density estimates. Its default method does so with the given kernel and bandwidth for univariate observations.

Usage

```
density(x, ...)
## Default S3 method:
density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov", "rectangular",
                   "triangular", "biweight",
                   "cosine", "optcosine"),
        weights = NULL, window = kernel, width,
        give.Rkern = FALSE, subdensity = FALSE,
        warnWbw = var(weights) > 0,
        n = 512, from, to, cut = 3, ext = 4,
        old.coords = FALSE,
        na.rm = FALSE, ...)
```

Arguments

x	the data from which the estimate is to be computed. For the default method a numeric vector: long vectors are not supported.
bw	the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below.) bw can also be a character string giving a rule to choose the bandwidth. See bw.nrd. The default, "nrd0", has remained the default for historical and compatibility reasons, rather than as a general recommendation, where e.g., "SJ" would rather fit, see also Venables and Ripley (2002). The specified (or computed) value of bw is multiplied by adjust.
adjust	the bandwidth used is actually adjust*bw. This makes it easy to specify values like 'half the default' bandwidth.
kernel, window	a character string giving the smoothing kernel to be used. This must partially match one of "gaussian", "rectangular", "triangular", "epanechnikov", "biweight", "cosine" or "optcosine", with default "gaussian", and may be abbreviated to a unique prefix (single letter). "cosine" is smoother than "optcosine", which is the usual 'cosine' kernel in the literature and almost MSE-efficient. However, "cosine" is the version used by S.
weights	numeric vector of non-negative observation weights, hence of same length as x. The default NULL is equivalent to weights = rep(1/nx, nx) where nx is the length of (the finite entries of) x[]. If na.rm = TRUE and there are NA's in x, they *and* the corresponding weights are removed before computations. In that case, when the original weights have summed to one, they are re-scaled to keep doing so. Note that weights are *not* taken into account for automatic bandwidth rules, i.e., when bw is a string. When the weights are proportional to true counts cn, density(x = rep(x, cn)) may be used instead of weights.
width	this exists for compatibility with S; if given, and bw is not, will set bw to width if this is a character string, or to a kernel-dependent multiple of width if this is numeric.
give.Rkern	logical; if true, *no* density is estimated, and the 'canonical bandwidth' of the chosen kernel is returned instead.
subdensity	used only when weights are specified which do not sum to one. When true, it indicates that a "sub-density" is desired and no warning should be signalled. By default, when false, a warning is signalled when the weights do not sum to one.
warnWbw	logical, used only when weights are specified *and* bw is character, i.e., automatic bandwidth selection is chosen (as by default). When true (as by default), a warning is signalled to alert the user that automatic bandwidth selection will not take the weights into account and hence may be suboptimal.
n	the number of equally spaced points at which the density is to be estimated. When n > 512, it is rounded up to a power of 2 during the calculations (as fft is used) and the final result is interpolated by approx. So it almost always makes sense to specify n as a power of two.

from, to	the left and right-most points of the grid at which the density is to be estimated; the defaults are cut * bw outside of range(x).
cut	by default, the values of from and to are cut bandwidths beyond the extremes of the data. This allows the estimated density to drop to approximately zero at the extremes.
ext	a positive extension factor, 4 by default. The values from and to are further extended on both sides to lo <- from - ext * bw and up <- to + ext * bw which are then used to build the grid used for the FFT and interpolation, see n above. Do not change unless you know what you are doing!
old.coords	logical to require pre-R 4.4.0 behaviour which gives too large values by a factor of about $(1 + 1/(2n - 2))$.
na.rm	logical; if TRUE, missing values are removed from x. If FALSE any missing values cause an error.
...	further arguments for (non-default) methods.

Details

The algorithm used in density.default disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.

The statistical properties of a kernel are determined by $\sigma_K^2 = \int t^2 K(t)dt$ which is always $= 1$ for our kernels (and hence the bandwidth bw is the standard deviation of the kernel) and $R(K) = \int K^2(t)dt$.
MSE-equivalent bandwidths (for different kernels) are proportional to $\sigma_K R(K)$ which is scale invariant and for our kernels equal to $R(K)$. This value is returned when give.Rkern = TRUE. See the examples for using exact equivalent bandwidths.

Infinite values in x are assumed to correspond to a point mass at +/-Inf and the density estimate is of the sub-density on (-Inf, +Inf).

Value

If give.Rkern is true, the number $R(K)$, otherwise an object with class "density" whose underlying structure is a list containing the following components.

x	the n coordinates of the points where the density is estimated.
y	the estimated density values. These will be non-negative, but can be zero.
bw	the bandwidth used.
n	the sample size after elimination of missing values.
call	the call which produced the result.
data.name	the deparsed name of the x argument.
has.na	logical, for compatibility (always FALSE).

The print method reports summary values on the x and y components.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole (for S version).

Scott, D. W. (1992). *Multivariate Density Estimation. Theory, Practice and Visualization*. New York: Wiley.

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society Series B*, **53**, 683–690. doi:10.1111/j.25176161.1991.tb01857.x.

Silverman, B. W. (1986). *Density Estimation*. London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. New York: Springer.

See Also

`bw.nrd`, `plot.density`, `hist`; `fft` and `convolve` for the computational short cut used.

Examples

```
require(graphics)

plot(density(c(-20, rep(0,98), 20)), xlim = c(-4, 4))  # IQR = 0

# The Old Faithful geyser data
d <- density(faithful$eruptions, bw = "sj")
d
plot(d)

plot(d, type = "n")
polygon(d, col = "wheat")

## Missing values:
x <- xx <- faithful$eruptions
x[i.out <- sample(length(x), 10)] <- NA
doR <- density(x, bw = 0.15, na.rm = TRUE)
lines(doR, col = "blue")
points(xx[i.out], rep(0.01, 10))

## Weighted observations:
fe <- sort(faithful$eruptions) # has quite a few non-unique values
## use 'counts / n' as weights:
dw <- density(unique(fe), weights = table(fe)/length(fe), bw = d$bw)
utils::str(dw) ## smaller n: only 126, but identical estimate:
stopifnot(all.equal(d[1:3], dw[1:3]))

## simulation from a density() fit:
# a kernel density fit is an equally-weighted mixture.
fit <- density(xx)
N <- 1e6
x.new <- rnorm(N, sample(xx, size = N, replace = TRUE), fit$bw)
plot(fit)
lines(density(x.new), col = "blue")
```

```
## The available kernels:
(kernels <- eval(formals(density.default)$kernel))

## show the kernels in the R parametrization
plot (density(0, bw = 1), xlab = "",
      main = "R's density() kernels with bw = 1")
for(i in 2:length(kernels))
   lines(density(0, bw = 1, kernel =  kernels[i]), col = i)
legend(1.5,.4, legend = kernels, col = seq(kernels),
       lty = 1, cex = .8, y.intersp = 1)

## show the kernels in the S parametrization
plot(density(0, from = -1.2, to = 1.2, width = 2, kernel = "gaussian"),
     type = "l", ylim = c(0, 1), xlab = "",
     main = "R's density() kernels with width = 1")
for(i in 2:length(kernels))
   lines(density(0, width = 2, kernel =  kernels[i]), col = i)
legend(0.6, 1.0, legend = kernels, col = seq(kernels), lty = 1)

##-------- Semi-advanced theoretic from here on -------------

## Explore the old.coords TRUE --> FALSE change:
set.seed(7); x <- runif(2^12) # N = 4096
den  <- density(x) # -> grid of n = 512 points
den0 <- density(x, old.coords = TRUE)
summary(den0$y / den$y) # 1.001 ... 1.011
summary(    den0$y / den$y - 1) # ~= 1/(2n-2)
summary(1/ (den0$y / den$y - 1))# ~=    2n-2 = 1022
corr0 <- 1 - 1/(2*512-2) # 1 - 1/(2n-2)
all.equal(den$y, den0$y * corr0)# ~ 0.0001
plot(den$x, (den0$y - den$y)/den$y, type='o', cex=1/4)
title("relative error of density(runif(2^12), old.coords=TRUE)")
abline(h = 1/1022, v = range(x), lty=2); axis(2, at=1/1022, "1/(2n-2)", las=1)


## The R[K] for our kernels:
(RKs <- cbind(sapply(kernels,
                     function(k) density(kernel = k, give.Rkern = TRUE))))
100*round(RKs["epanechnikov",]/RKs, 4) ## Efficiencies

bw <- bw.SJ(precip) ## sensible automatic choice
plot(density(precip, bw = bw),
     main = "same sd bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, kernel = kernels[i]), col = i)

## Bandwidth Adjustment for "Exactly Equivalent Kernels"
h.f <- sapply(kernels, function(k)density(kernel = k, give.Rkern = TRUE))
(h.f <- (h.f["gaussian"] / h.f)^ .2)
## -> 1, 1.01, .995, 1.007,... close to 1 => adjustment barely visible..
```

```
plot(density(precip, bw = bw),
     main = "equivalent bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, adjust = h.f[i], kernel = kernels[i]),
         col = i)
legend(55, 0.035, legend = kernels, col = seq(kernels), lty = 1)
```

---

deriv *Symbolic and Algorithmic Derivatives of Simple Expressions*

Description

Compute derivatives of simple expressions, symbolically and algorithmically.

Usage

```
    D (expr, name)
 deriv(expr, ...)
deriv3(expr, ...)

 ## Default S3 method:
deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)
 ## S3 method for class 'formula'
deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)

## Default S3 method:
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
## S3 method for class 'formula'
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
```

Arguments

expr	a expression or call or (except D) a formula with no lhs.
name, namevec	character vector, giving the variable names (only one for D()) with respect to which derivatives will be computed.
function.arg	if specified and non-NULL, a character vector of arguments for a function return, or a function (with empty body) or TRUE, the latter indicating that a function with argument names namevec should be used.
tag	character; the prefix to be used for the locally created variables in result. Must be no longer than 60 bytes when translated to the native encoding.
hessian	a logical value indicating whether the second derivatives should be calculated and incorporated in the return value.
...	arguments to be passed to or from methods.

Details

D is modelled after its S namesake for taking simple symbolic derivatives.

deriv is a *generic* function with a default and a formula method. It returns a call for computing the expr and its (partial) derivatives, simultaneously. It uses so-called *algorithmic derivatives*. If function.arg is a function, its arguments can have default values, see the fx example below.

Currently, deriv.formula just calls deriv.default after extracting the expression to the right of ~.

deriv3 and its methods are equivalent to deriv and its methods except that hessian defaults to TRUE for deriv3.

The internal code knows about the arithmetic operators +, -, *, / and ^, and the single-variable functions exp, log, sin, cos, tan, sinh, cosh, sqrt, pnorm, dnorm, asin, acos, atan, gamma, lgamma, digamma and trigamma, as well as psigamma for one or two arguments (but derivative only with respect to the first). (Note that only the standard normal distribution is considered.)
Since R 3.4.0, the single-variable functions log1p, expm1, log2, log10, cospi, sinpi, tanpi, factorial, and lfactorial are supported as well.

Value

D returns a call and therefore can easily be iterated for higher derivatives.

deriv and deriv3 normally return an expression object whose evaluation returns the function values with a "gradient" attribute containing the gradient matrix. If hessian is TRUE the evaluation also returns a "hessian" attribute containing the Hessian array.

If function.arg is not NULL, deriv and deriv3 return a function with those arguments rather than an expression.

References

Griewank, A. and Corliss, G. F. (1991) *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM proceedings, Philadelphia.

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

nlm and optim for numeric minimization which could make use of derivatives,

Examples

```
## formula argument :
dx2x <- deriv(~ x^2, "x") ; dx2x
## Not run: expression({
         .value <- x^2
         .grad <- array(0, c(length(.value), 1), list(NULL, c("x")))
         .grad[, "x"] <- 2 * x
         attr(.value, "gradient") <- .grad
         .value
})
```

```
## End(Not run)
mode(dx2x)
x <- -1:2
eval(dx2x)

## Something 'tougher':
trig.exp <- expression(sin(cos(x + y^2)))
( D.sc <- D(trig.exp, "x") )
all.equal(D(trig.exp[[1]], "x"), D.sc)

( dxy <- deriv(trig.exp, c("x", "y")) )
y <- 1
eval(dxy)
eval(D.sc)

## function returned:
deriv((y ~ sin(cos(x) * y)), c("x","y"), function.arg = TRUE)

## function with defaulted arguments:
(fx <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
             function(b0, b1, th, x = 1:7){} ) )
fx(2, 3, 4)

## First derivative

D(expression(x^2), "x")
stopifnot(D(as.name("x"), "x") == 1)

## Higher derivatives
deriv3(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
     c("b0", "b1", "th", "x") )

## Higher derivatives:
DD <- function(expr, name, order = 1) {
   if(order < 1) stop("'order' must be >= 1")
   if(order == 1) D(expr, name)
   else DD(D(expr, name), name, order - 1)
}
DD(expression(sin(x^2)), "x", 3)
## showing the limits of the internal "simplify()" :
## Not run:
-sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
    2) * (2 * x) + sin(x^2) * (2 * x) * 2)

## End(Not run)

## New (R 3.4.0, 2017):
D(quote(log1p(x^2)), "x") ## log1p(x) = log(1 + x)
stopifnot(identical(
       D(quote(log1p(x^2)), "x"),
       D(quote(log(1+x^2)), "x")))
D(quote(expm1(x^2)), "x") ## expm1(x) = exp(x) - 1
stopifnot(identical(
```

```
        D(quote(expm1(x^2)), "x") -> Dex1,
        D(quote(exp(x^2)-1), "x")),
        identical(Dex1, quote(exp(x^2) * (2 * x))))

D(quote(sinpi(x^2)), "x") ## sinpi(x) = sin(pi*x)
D(quote(cospi(x^2)), "x") ## cospi(x) = cos(pi*x)
D(quote(tanpi(x^2)), "x") ## tanpi(x) = tan(pi*x)

stopifnot(identical(D(quote(log2 (x^2)), "x"),
                    quote(2 * x/(x^2 * log(2)))),
          identical(D(quote(log10(x^2)), "x"),
                    quote(2 * x/(x^2 * log(10)))))
```

`deviance` *Model Deviance*

Description

Returns the deviance of a fitted model object.

Usage

```
deviance(object, ...)
```

Arguments

`object`	an object for which the deviance is desired.
`...`	additional optional argument.

Details

This is a generic function which can be used to extract deviances for fitted models. Consult the individual modeling functions for details on how to use this function.

Value

The value of the deviance extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`df.residual`, `extractAIC`, `glm`, `lm`.

df.residual *Residual Degrees-of-Freedom*

Description

Returns the residual degrees-of-freedom extracted from a fitted model object.

Usage

```
df.residual(object, ...)
```

Arguments

object	an object for which the degrees-of-freedom are desired.
...	additional optional arguments.

Details

This is a generic function which can be used to extract residual degrees-of-freedom for fitted models. Consult the individual modeling functions for details on how to use this function.

The default method just extracts the `df.residual` component.

Value

The value of the residual degrees-of-freedom extracted from the object `x`.

See Also

deviance, glm, lm.

diffinv *Discrete Integration: Inverse of Differencing*

Description

Computes the inverse function of the lagged differences function diff.

Usage

```
diffinv(x, ...)

## Default S3 method:
diffinv(x, lag = 1, differences = 1, xi, ...)
## S3 method for class 'ts'
diffinv(x, lag = 1, differences = 1, xi, ...)
```

Arguments

x	a numeric vector, matrix, or time series.
lag	a scalar lag parameter.
differences	an integer representing the order of the difference.
xi	a numeric vector, matrix, or time series containing the initial values for the integrals. If missing, zeros are used.
...	arguments passed to or from other methods.

Details

diffinv is a generic function with methods for class "ts" and default for vectors and matrices.

Missing values are not handled.

Value

A numeric vector, matrix, or time series (the latter for the "ts" method) representing the discrete integral of x.

Author(s)

A. Trapletti

See Also

diff

Examples

```
s <- 1:10
d <- diff(s)
diffinv(d, xi = 1)
```

---

dist *Distance Matrix Computation*

Description

This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.

Usage

```
dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)

as.dist(m, diag = FALSE, upper = FALSE)
## Default S3 method:
as.dist(m, diag = FALSE, upper = FALSE)

## S3 method for class 'dist'
print(x, diag = NULL, upper = NULL,
      digits = getOption("digits"), justify = "none",
      right = TRUE, ...)

## S3 method for class 'dist'
as.matrix(x, ...)
```

Arguments

x	a numeric matrix, data frame or "dist" object.
method	the distance measure to be used. This must be one of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski". Any unambiguous substring can be given.
diag	logical value indicating whether the diagonal of the distance matrix should be printed by print.dist.
upper	logical value indicating whether the upper triangle of the distance matrix should be printed by print.dist.
p	The power of the Minkowski distance.
m	An object with distance information to be converted to a "dist" object. For the default method, a "dist" object, or a matrix (of distances) or an object which can be coerced to such a matrix using as.matrix(). (Only the lower triangle of the matrix is used, the rest is ignored).
digits, justify	passed to format inside of print().
right, ...	further arguments, passed to other methods.

Details

Available distance measures are (written for two vectors x and y):

euclidean: Usual distance between the two vectors (2 norm aka L_2), $\sqrt{\sum_i (x_i - y_i)^2}$.

maximum: Maximum distance between two components of x and y (supremum norm)

manhattan: Absolute distance between the two vectors (1 norm aka L_1).

canberra: $\sum_i |x_i - y_i| / (|x_i| + |y_i|)$. Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

This is intended for non-negative values (e.g., counts), in which case the denominator can be written in various equivalent ways; Originally, R used $x_i + y_i$, then from 1998 to 2017, $|x_i + y_i|$, and then the correct $|x_i| + |y_i|$.

`binary`**:** (aka *asymmetric binary*): The vectors are regarded as binary bits, so non-zero elements are 'on' and zero elements are 'off'. The distance is the *proportion* of bits in which only one is on amongst those in which at least one is on. This also called "Jaccard" distance in some contexts. Here, two all-zero observations have distance `0`, whereas in traditional Jaccard definitions, the distance would be undefined for that case and give `NaN` numerically.

`minkowski`**:** The p norm, the p-th root of the sum of the p-th powers of the differences of the components.

Missing values are allowed, and are excluded from all computations involving the rows within which they occur. Further, when `Inf` values are involved, all pairs of values are excluded when their contribution to the distance gave `NaN` or `NA`. If some columns are excluded in calculating a Euclidean, Manhattan, Canberra or Minkowski distance, the sum is scaled up proportionally to the number of columns used. If all pairs are excluded when calculating a particular distance, the value is `NA`.

The `"dist"` method of `as.matrix()` and `as.dist()` can be used for conversion between objects of class `"dist"` and conventional distance matrices.

`as.dist()` is a generic function. Its default method handles objects inheriting from class `"dist"`, or coercible to matrices using `as.matrix()`. Support for classes representing distances (also known as dissimilarities) can be added by providing an `as.matrix()` or, more directly, an `as.dist` method for such a class.

Value

`dist` returns an object of class `"dist"`.

The lower triangle of the distance matrix stored by columns in a vector, say `do`. If `n` is the number of observations, i.e., `n <- attr(do, "Size")`, then for $i < j \le n$, the dissimilarity between (row) `i` and `j` is `do[n*(i-1) - i*(i-1)/2 + j-i]`. The length of the vector is $n * (n-1)/2$, i.e., of order n^2.

The object has the following attributes (besides `"class"` equal to `"dist"`):

`Size`	integer, the number of observations in the dataset.
`Labels`	optionally, contains the labels, if any, of the observations of the dataset.
`Diag`, `Upper`	logicals corresponding to the arguments `diag` and `upper` above, specifying how the object should be printed.
`call`	optionally, the `call` used to create the object.
`method`	optionally, the distance method used; resulting from `dist()`, the (`match.arg()`ed) `method` argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis.* Academic Press.

Borg, I. and Groenen, P. (1997) *Modern Multidimensional Scaling. Theory and Applications.* Springer.

See Also

daisy in the **cluster** package with more possibilities in the case of *mixed* (continuous / categorical) variables. hclust.

Examples

```
require(graphics)

x <- matrix(rnorm(100), nrow = 5)
dist(x)
dist(x, diag = TRUE)
dist(x, upper = TRUE)
m <- as.matrix(dist(x))
d <- as.dist(m)
stopifnot(d == dist(x))

## Use correlations between variables "as distance"
dd <- as.dist((1 - cor(USJudgeRatings))/2)
round(1000 * dd) # (prints more nicely)
plot(hclust(dd)) # to see a dendrogram of clustered variables

## example of binary and canberra distances.
x <- c(0, 0, 1, 1, 1, 1)
y <- c(1, 0, 1, 1, 0, 1)
dist(rbind(x, y), method = "binary")
## answer 0.4 = 2/5
dist(rbind(x, y), method = "canberra")
## answer 2 * (6/5)

## To find the names
labels(eurodist)

## Examples involving "Inf" :
## 1)
x[6] <- Inf
(m2 <- rbind(x, y))
dist(m2, method = "binary")   # warning, answer 0.5 = 2/4
## These all give "Inf":
stopifnot(Inf == dist(m2, method =  "euclidean"),
          Inf == dist(m2, method =  "maximum"),
          Inf == dist(m2, method =  "manhattan"))
##  "Inf" is same as very large number:
x1 <- x; x1[6] <- 1e100
stopifnot(dist(cbind(x, y), method = "canberra") ==
    print(dist(cbind(x1, y), method = "canberra")))

## 2)
y[6] <- Inf #-> 6-th pair is excluded
dist(rbind(x, y), method = "binary"  )   # warning; 0.5
dist(rbind(x, y), method = "canberra"  ) # 3
dist(rbind(x, y), method = "maximum")    # 1
dist(rbind(x, y), method = "manhattan")  # 2.4
```

`Distributions` *Distributions in the stats package*

Description

Density, cumulative distribution function, quantile function and random variate generation for many standard probability distributions are available in the **stats** package.

Details

The functions for the density/mass function, cumulative distribution function, quantile function and random variate generation are named in the form `dxxx`, `pxxx`, `qxxx` and `rxxx` respectively.

For the beta distribution see `dbeta`.

For the binomial (including Bernoulli) distribution see `dbinom`.

For the Cauchy distribution see `dcauchy`.

For the chi-squared distribution see `dchisq`.

For the exponential distribution see `dexp`.

For the F distribution see `df`.

For the gamma distribution see `dgamma`.

For the geometric distribution see `dgeom`. (This is also a special case of the negative binomial.)

For the hypergeometric distribution see `dhyper`.

For the log-normal distribution see `dlnorm`.

For the multinomial distribution see `dmultinom`.

For the negative binomial distribution see `dnbinom`.

For the normal distribution see `dnorm`.

For the Poisson distribution see `dpois`.

For the Student's t distribution see `dt`.

For the uniform distribution see `dunif`.

For the Weibull distribution see `dweibull`.

For less common distributions of test statistics see `pbirthday`, `dsignrank`, `ptukey` and `dwilcox` (and see the 'See Also' section of `cor.test`).

See Also

`RNG` about random number generation in R.

The CRAN task view on distributions, `https://CRAN.R-project.org/view=Distributions`, mentioning several CRAN packages for additional distributions.

dummy.coef *Extract Coefficients in Original Coding*

Description

This extracts coefficients in terms of the original levels of the coefficients rather than the coded variables.

Usage

```
dummy.coef(object, ...)

## S3 method for class 'lm'
dummy.coef(object, use.na = FALSE, ...)

## S3 method for class 'aovlist'
dummy.coef(object, use.na = FALSE, ...)
```

Arguments

object	a linear model fit.
use.na	logical flag for coefficients in a singular model. If use.na is true, undetermined coefficients will be missing; if false they will get one possible value.
...	arguments passed to or from other methods.

Details

A fitted linear model has coefficients for the contrasts of the factor terms, usually one less in number than the number of levels. This function re-expresses the coefficients in the original coding; as the coefficients will have been fitted in the reduced basis, any implied constraints (e.g., zero sum for contr.helmert or contr.sum) will be respected. There will be little point in using dummy.coef for contr.treatment contrasts, as the missing coefficients are by definition zero.

The method used has some limitations, and will give incomplete results for terms such as poly(x, 2). However, it is adequate for its main purpose, aov models.

Value

A list giving for each term the values of the coefficients. For a multistratum aov model, such a list for each stratum.

Warning

This function is intended for human inspection of the output: it should not be used for calculations. Use coded variables for all calculations.

The results differ from S for singular values, where S can be incorrect.

See Also

aov, model.tables

Examples

```
options(contrasts = c("contr.helmert", "contr.poly"))
## From Venables and Ripley (2002) p.165.
npk.aov <- aov(yield ~ block + N*P*K, npk)
dummy.coef(npk.aov)

npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
dummy.coef(npk.aovE)
```

---

ecdf *Empirical Cumulative Distribution Function*

Description

Compute an empirical cumulative distribution function, with several methods for plotting, printing and computing with such an "ecdf" object.

Usage

```
ecdf(x)

## S3 method for class 'ecdf'
plot(x, ..., ylab="Fn(x)", verticals = FALSE,
     col.01line = "gray70", pch = 19)

## S3 method for class 'ecdf'
print(x, digits= getOption("digits") - 2, ...)

## S3 method for class 'ecdf'
summary(object, ...)
## S3 method for class 'ecdf'
quantile(x, ...)
```

Arguments

x, object	numeric vector of the observations for ecdf; for the methods, an object inheriting from class "ecdf".
...	arguments to be passed to subsequent methods, e.g., plot.stepfun for the plot method.
ylab	label for the y-axis.
verticals	see plot.stepfun.

col.01line	numeric or character specifying the color of the horizontal lines at y = 0 and 1, see colors.
pch	plotting character.
digits	number of significant digits to use, see print.

Details

The e.c.d.f. (empirical cumulative distribution function) F_n is a step function with jumps i/n at observation values, where i is the number of tied observations at that value. Missing values are ignored.

For observations x= $(x_1, x_2, \ldots x_n)$, F_n is the fraction of observations less or equal to t, i.e.,

$$F_n(t) = \#\{x_i \le t\} \,/n = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{[x_i \le t]}.$$

The function plot.ecdf which implements the plot method for ecdf objects, is implemented via a call to plot.stepfun; see its documentation.

Value

For ecdf, a function of class "ecdf", inheriting from the "stepfun" class, and hence inheriting a knots() method.

For the summary method, a summary of the knots of object with a "header" attribute.

The quantile(obj, ...) method computes the same quantiles as quantile(x, ...) would where x is the original sample.

Note

The objects of class "ecdf" are not intended to be used for permanent storage and may change structure between versions of R (and did at R 3.0.0). They can usually be re-created by

```
eval(attr(old_obj, "call"), environment(old_obj))
```

since the data used is stored as part of the object's environment.

Author(s)

Martin Maechler; fixes and new features by other R-core members.

See Also

stepfun, the more general class of step functions, approxfun and splinefun.

Examples

```
##-- Simple didactical  ecdf  example :
x <- rnorm(12)
Fn <- ecdf(x)
Fn     # a *function*
Fn(x)  # returns the percentiles for x
tt <- seq(-2, 2, by = 0.1)
12 * Fn(tt) # Fn is a 'simple' function {with values k/12}
summary(Fn)
##--> see below for graphics
knots(Fn)  # the unique data values {12 of them if there were no ties}

y <- round(rnorm(12), 1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
knots(Fn12) # unique values (always less than 12!)
summary(Fn12)
summary.stepfun(Fn12)

## Advanced: What's inside the function closure?
ls(environment(Fn12))
## "f"     "method" "na.rm"  "nobs"   "x"     "y"    "yleft"  "yright"
utils::ls.str(environment(Fn12))
stopifnot(all.equal(quantile(Fn12), quantile(y)))

###----------------- Plotting --------------------------
require(graphics)

op <- par(mfrow = c(3, 1), mgp = c(1.5, 0.8, 0), mar =  .1+c(3,3,2,1))

F10 <- ecdf(rnorm(10))
summary(F10)

plot(F10)
plot(F10, verticals = TRUE, do.points = FALSE)

plot(Fn12 , lwd = 2) ; mtext("lwd = 2", adj = 1)
xx <- unique(sort(c(seq(-3, 2, length.out = 201), knots(Fn12))))
lines(xx, Fn12(xx), col = "blue")
abline(v = knots(Fn12), lty = 2, col = "gray70")

plot(xx, Fn12(xx), type = "o", cex = .1)  #- plot.default {ugly}
plot(Fn12, col.hor = "red", add =  TRUE)  #- plot method
abline(v = knots(Fn12), lty = 2, col = "gray70")
## luxury plot
plot(Fn12, verticals = TRUE, col.points = "blue",
     col.hor = "red", col.vert = "bisque")

##-- this works too (automatic call to  ecdf(.)):
plot.ecdf(rnorm(24))
title("via  simple  plot.ecdf(x)", adj = 1)
```

```
par(op)
```

`eff.aovlist` *Compute Efficiencies of Multistratum Analysis of Variance*

Description

Computes the efficiencies of fixed-effect terms in an analysis of variance model with multiple strata.

Usage

```
eff.aovlist(aovlist)
```

Arguments

`aovlist`	The result of a call to `aov` with an `Error` term.

Details

Fixed-effect terms in an analysis of variance model with multiple strata may be estimable in more than one stratum, in which case there is less than complete information in each. The efficiency for a term is the fraction of the maximum possible precision (inverse variance) obtainable by estimating in just that stratum. Under the assumption of balance, this is the same for all contrasts involving that term.

This function is used to pick strata in which to estimate terms in `model.tables.aovlist` and `se.contrast.aovlist`.

In many cases terms will only occur in one stratum, when all the efficiencies will be one: this is detected and no further calculations are done.

The calculation used requires orthogonal contrasts for each term, and will throw an error if non-orthogonal contrasts (e.g., treatment contrasts or an unbalanced design) are detected.

Value

A matrix giving for each non-pure-error stratum (row) the efficiencies for each fixed-effect term in the model.

References

Heiberger, R. M. (1989) *Computation for the Analysis of Designed Experiments*. Wiley.

See Also

`aov`, `model.tables.aovlist`, `se.contrast.aovlist`

Examples

```
## An example from Yates (1932),
## a 2^3 design in 2 blocks replicated 4 times

Block <- gl(8, 4)
A <- factor(c(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
              0,1,0,1,0,1,0,1,0,1,0,1))
B <- factor(c(0,0,1,1,0,0,1,1,0,1,0,1,1,0,1,0,0,0,1,1,
              0,0,1,1,0,0,1,1,0,0,1,1))
C <- factor(c(0,1,1,0,1,0,0,1,0,0,1,1,0,0,1,1,0,1,0,1,
              1,0,1,0,0,0,1,1,1,1,0,0))
Yield <- c(101, 373, 398, 291, 312, 106, 265, 450, 106, 306, 324, 449,
           272, 89, 407, 338, 87, 324, 279, 471, 323, 128, 423, 334,
           131, 103, 445, 437, 324, 361, 302, 272)
aovdat <- data.frame(Block, A, B, C, Yield)

old <- getOption("contrasts")
options(contrasts = c("contr.helmert", "contr.poly"))

(fit <- aov(Yield ~ A*B*C + Error(Block), data = aovdat))

eff.aovlist(fit)
options(contrasts = old)
```

`effects` *Effects from Fitted Model*

Description

Returns (orthogonal) effects from a fitted model, usually a linear model. This is a generic function, but currently only has a methods for objects inheriting from classes `"lm"` and `"glm"`.

Usage

```
effects(object, ...)

## S3 method for class 'lm'
effects(object, set.sign = FALSE, ...)
```

Arguments

`object`	an R object; typically, the result of a model fitting function such as `lm`.
`set.sign`	logical. If `TRUE`, the sign of the effects corresponding to coefficients in the model will be set to agree with the signs of the corresponding coefficients, otherwise the sign is arbitrary.
`...`	arguments passed to or from other methods.

Details

For a linear model fitted by `lm` or `aov`, the effects are the uncorrelated single-degree-of-freedom values obtained by projecting the data onto the successive orthogonal subspaces generated by the QR decomposition during the fitting process. The first r (the rank of the model) are associated with coefficients and the remainder span the space of residuals (but are not associated with particular residuals).

Empty models do not have effects.

Value

A (named) numeric vector of the same length as `residuals`, or a matrix if there were multiple responses in the fitted model, in either case of class `"coef"`.

The first r rows are labelled by the corresponding coefficients, and the remaining rows are unlabelled. Note that in rank-deficient models the corresponding coefficients will be in a different order if pivoting occurred.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`coef`

Examples

```
y <- c(1:3, 7, 5)
x <- c(1:3, 6:7)
( ee <- effects(lm(y ~ x)) )
c( round(ee - effects(lm(y+10 ~ I(x-3.8))), 3) )
# just the first is different
```

`embed` *Embedding a Time Series*

Description

Embeds the time series `x` into a low-dimensional Euclidean space.

Usage

```
embed (x, dimension = 1)
```

Arguments

`x`	a numeric vector, matrix, or time series.
`dimension`	a scalar representing the embedding dimension.

Details

Each row of the resulting matrix consists of sequences `x[t]`, `x[t-1]`, ..., `x[t-dimension+1]`, where `t` is the original index of `x`. If `x` is a matrix, i.e., `x` contains more than one variable, then `x[t]` consists of the `t`-th observation on each variable.

Value

A matrix containing the embedded time series `x`.

Author(s)

A. Trapletti, B.D. Ripley

Examples

```
x <- 1:10
embed (x, 3)
```

---

expand.model.frame *Add new variables to a model frame*

Description

Evaluates new variables as if they had been part of the formula of the specified model. This ensures that the same `na.action` and `subset` arguments are applied and allows, for example, `x` to be recovered for a model using `sin(x)` as a predictor.

Usage

```
expand.model.frame(model, extras,
                   envir = environment(formula(model)),
                   na.expand = FALSE)
```

Arguments

`model`	a fitted model
`extras`	one-sided formula or vector of character strings describing new variables to be added
`envir`	an environment to evaluate things in
`na.expand`	logical; see below

Details

If `na.expand = FALSE` then `NA` values in the extra variables will be passed to the `na.action` function used in `model`. This may result in a shorter data frame (with `na.omit`) or an error (with `na.fail`). If `na.expand = TRUE` the returned data frame will have precisely the same rows as `model.frame(model)`, but the columns corresponding to the extra variables may contain `NA`.

Value

A data frame.

See Also

model.frame, predict

Examples

```
model <- lm(log(Volume) ~ log(Girth) + log(Height), data = trees)
expand.model.frame(model, ~ Girth) # prints data.frame like

dd <- data.frame(x = 1:5, y = rnorm(5), z = c(1,2,NA,4,5))
model <- glm(y ~ x, data = dd, subset = 1:4, na.action = na.omit)
expand.model.frame(model, "z", na.expand = FALSE) # = default
expand.model.frame(model, "z", na.expand = TRUE)
```

Exponential *The Exponential Distribution*

Description

Density, distribution function, quantile function and random generation for the exponential distribution with rate `rate` (i.e., mean `1/rate`).

Usage

```
dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If `length(n) > 1`, the length is taken to be the number required.
rate	vector of rates.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `rate` is not specified, it assumes the default value of `1`.

The exponential distribution with rate λ has density

$$f(x) = \lambda e^{-\lambda x}$$

for $x \geq 0$.

Value

`dexp` gives the density, `pexp` gives the distribution function, `qexp` gives the quantile function, and `rexp` generates random deviates.

The length of the result is determined by `n` for `rexp`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pexp(t, r, lower = FALSE, log = TRUE)`.

Source

`dexp`, `pexp` and `qexp` are all calculated from numerically stable versions of the definitions.

`rexp` uses

Ahrens, J. H. and Dieter, U. (1972). Computer methods for sampling from the exponential and normal distributions. *Communications of the ACM*, **15**, 873–882.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 19. Wiley, New York.

See Also

`exp` for the exponential function.

Distributions for other standard distributions, including `dgamma` for the gamma distribution and `dweibull` for the Weibull distribution, both of which generalize the exponential.

Examples

```
dexp(1) - exp(-1) #-> 0

## a fast way to generate *sorted*  U[0,1]  random numbers:
rsunif <- function(n) { n1 <- n+1
   cE <- cumsum(rexp(n1)); cE[seq_len(n)]/cE[n1] }
```

```
plot(rsunif(1000), ylim=0:1, pch=".")
abline(0,1/(1000+1), col=adjustcolor(1, 0.5))
```

`extractAIC` *Extract AIC from a Fitted Model*

Description

Computes the (generalized) Akaike **A**n **I**nformation **C**riterion for a fitted parametric model.

Usage

```
extractAIC(fit, scale, k = 2, ...)
```

Arguments

`fit`	fitted model, usually the result of a fitter like `lm`.
`scale`	optional numeric specifying the scale parameter of the model, see `scale` in `step`. Currently only used in the `"lm"` method, where `scale` specifies the estimate of the error variance, and `scale = 0` indicates that it is to be estimated by maximum likelihood.
`k`	numeric specifying the 'weight' of the *equivalent degrees of freedom* (≡ `edf`) part in the AIC formula.
`...`	further arguments (currently unused in base R).

Details

This is a generic function, with methods in base R for classes `"aov"`, `"glm"` and `"lm"` as well as for `"negbin"` (package **MASS**) and `"coxph"` and `"survreg"` (package **survival**).

The criterion used is

$$AIC = -2\log L + k \times \text{edf},$$

where L is the likelihood and `edf` the equivalent degrees of freedom (i.e., the number of free parameters for usual parametric models) of `fit`.

For linear models with unknown scale (i.e., for `lm` and `aov`), $-2\log L$ is computed from the *deviance* and uses a different additive constant to `logLik` and hence `AIC`. If RSS denotes the (weighted) residual sum of squares then `extractAIC` uses for $-2\log L$ the formulae $RSS/s - n$ (corresponding to Mallows' C_p) in the case of known scale s and $n\log(RSS/n)$ for unknown scale. `AIC` only handles unknown scale and uses the formula $n\log(RSS/n) + n + n\log 2\pi - \sum \log w$ where w are the weights. Further `AIC` counts the scale estimation as a parameter in the `edf` and `extractAIC` does not.

For `glm` fits the family's `aic()` function is used to compute the AIC: see the note under `logLik` about the assumptions this makes.

`k = 2` corresponds to the traditional AIC, using `k = log(n)` provides the BIC (Bayesian IC) instead.

Note that the methods for this function may differ in their assumptions from those of methods for `AIC` (usually *via* a method for `logLik`). We have already mentioned the case of `"lm"` models

with estimated scale, and there are similar issues in the "glm" and "negbin" methods where the dispersion parameter may or may not be taken as 'free'. This is immaterial as extractAIC is only used to compare models of the same class (where only differences in AIC values are considered).

Value

A numeric vector of length 2, with first and second elements giving

edf	the '**e**quivalent **d**egrees of **f**reedom' for the fitted model fit.
AIC	the (generalized) Akaike Information Criterion for fit.

Note

This function is used in add1, drop1 and step and the similar functions in package **MASS** from which it was adopted.

Author(s)

B. D. Ripley

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

See Also

AIC, deviance, add1, step

Examples

```
utils::example(glm)
extractAIC(glm.D93)  #>>  5  15.129
```

---

factanal *Factor Analysis*

Description

Perform maximum-likelihood factor analysis on a covariance matrix or data matrix.

Usage

```
factanal(x, factors, data = NULL, covmat = NULL, n.obs = NA,
         subset, na.action, start = NULL,
         scores = c("none", "regression", "Bartlett"),
         rotation = "varimax", control = NULL, ...)
```

Arguments

x	A formula or a numeric matrix or an object that can be coerced to a numeric matrix.
factors	The number of factors to be fitted.
data	An optional data frame (or similar: see model.frame), used only if x is a formula. By default the variables are taken from environment(formula).
covmat	A covariance matrix, or a covariance list as returned by cov.wt. Of course, correlation matrices are covariance matrices.
n.obs	The number of observations, used if covmat is a covariance matrix.
subset	A specification of the cases to be used, if x is used as a matrix or formula.
na.action	The na.action to be used if x is used as a formula.
start	NULL or a matrix of starting values, each column giving an initial set of uniquenesses.
scores	Type of scores to produce, if any. The default is none, "regression" gives Thompson's scores, "Bartlett" given Bartlett's weighted least-squares scores. Partial matching allows these names to be abbreviated.
rotation	character. "none" or the name of a function to be used to rotate the factors: it will be called with first argument the loadings matrix, and should return a list with component loadings giving the rotated loadings, or just the rotated loadings.
control	A list of control values, **nstart** The number of starting values to be tried if start = NULL. Default 1. **trace** logical. Output tracing information? Default FALSE. **lower** The lower bound for uniquenesses during optimization. Should be > 0. Default 0.005. **opt** A list of control values to be passed to optim's control argument. **rotate** a list of additional arguments for the rotation function.
...	Components of control can also be supplied as named arguments to factanal.

Details

The factor analysis model is

$$x = \Lambda f + e$$

for a p–element vector x, a $p \times k$ matrix Λ of *loadings*, a k–element vector f of *scores* and a p–element vector e of errors. None of the components other than x is observed, but the major restriction is that the scores be uncorrelated and of unit variance, and that the errors be independent with variances Ψ, the *uniquenesses*. It is also common to scale the observed variables to unit variance, and done in this function.

Thus factor analysis is in essence a model for the correlation matrix of x,

$$\Sigma = \Lambda\Lambda' + \Psi$$

There is still some indeterminacy in the model for it is unchanged if Λ is replaced by $G\Lambda$ for any orthogonal matrix G. Such matrices G are known as *rotations* (although the term is applied also to non-orthogonal invertible matrices).

If covmat is supplied it is used. Otherwise x is used if it is a matrix, or a formula x is used with data to construct a model matrix, and that is used to construct a covariance matrix. (It makes no sense for the formula to have a response, and all the variables must be numeric.) Once a covariance matrix is found or calculated from x, it is converted to a correlation matrix for analysis. The correlation matrix is returned as component correlation of the result.

The fit is done by optimizing the log likelihood assuming multivariate normality over the uniquenesses. (The maximizing loadings for given uniquenesses can be found analytically: Lawley & Maxwell (1971, p. 27).) All the starting values supplied in start are tried in turn and the best fit obtained is used. If start = NULL then the first fit is started at the value suggested by Jöreskog (1963) and given by Lawley & Maxwell (1971, p. 31), and then control$nstart - 1 other values are tried, randomly selected as equal values of the uniquenesses.

The uniquenesses are technically constrained to lie in $[0, 1]$, but near-zero values are problematical, and the optimization is done with a lower bound of control$lower, default 0.005 (Lawley & Maxwell, 1971, p. 32).

Scores can only be produced if a data matrix is supplied and used. The first method is the regression method of Thomson (1951), the second the weighted least squares method of Bartlett (1937, 8). Both are estimates of the unobserved scores f. Thomson's method regresses (in the population) the unknown f on x to yield

$$\hat{f} = \Lambda' \Sigma^{-1} x$$

and then substitutes the sample estimates of the quantities on the right-hand side. Bartlett's method minimizes the sum of squares of standardized errors over the choice of f, given (the fitted) Λ.

If x is a formula then the standard NA-handling is applied to the scores (if requested): see napredict.

The print method (documented under loadings) follows the factor analysis convention of drawing attention to the patterns of the results, so the default precision is three decimal places, and small loadings are suppressed.

Value

An object of class "factanal" with components

loadings	A matrix of loadings, one column for each factor. The factors are ordered in decreasing order of sums of squares of loadings, and given the sign that will make the sum of the loadings positive. This is of class "loadings": see loadings for its print method.
uniquenesses	The uniquenesses computed.
correlation	The correlation matrix used.
criteria	The results of the optimization: the value of the criterion (a linear function of the negative log-likelihood) and information on the iterations used.
factors	The argument factors.
dof	The number of degrees of freedom of the factor analysis model.
method	The method: always "mle".
rotmat	The rotation matrix if relevant.
scores	If requested, a matrix of scores. napredict is applied to handle the treatment of values omitted by the na.action.

n.obs	The number of observations if available, or NA.
call	The matched call.
na.action	If relevant.
STATISTIC, PVAL	The significance-test statistic and P value, if it can be computed.

Note

There are so many variations on factor analysis that it is hard to compare output from different programs. Further, the optimization in maximum likelihood factor analysis is hard, and many other examples we compared had less good fits than produced by this function. In particular, solutions which are 'Heywood cases' (with one or more uniquenesses essentially zero) are much more common than most texts and some other programs would lead one to believe.

References

Bartlett, M. S. (1937). The statistical conception of mental factors. *British Journal of Psychology*, **28**, 97–104. doi:10.1111/j.20448295.1937.tb00863.x.

Bartlett, M. S. (1938). Methods of estimating mental factors. *Nature*, **141**, 609–610. doi:10.1038/141246a0.

Jöreskog, K. G. (1963). *Statistical Estimation in Factor Analysis*. Almqvist and Wicksell.

Lawley, D. N. and Maxwell, A. E. (1971). *Factor Analysis as a Statistical Method*. Second edition. Butterworths.

Thomson, G. H. (1951). *The Factorial Analysis of Human Ability*. London University Press.

See Also

loadings (which explains some details of the print method), varimax, princomp, ability.cov, Harman23.cor, Harman74.cor.

Other rotation methods are available in various contributed packages, including **GPArotation** and **psych**.

Examples

```
# A little demonstration, v2 is just v1 with noise,
# and same for v4 vs. v3 and v6 vs. v5
# Last four cases are there to add noise
# and introduce a positive manifold (g factor)
v1 <- c(1,1,1,1,1,1,1,1,1,1,3,3,3,3,3,4,5,6)
v2 <- c(1,2,1,1,1,1,2,1,2,1,3,4,3,3,3,4,6,5)
v3 <- c(3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,5,4,6)
v4 <- c(3,3,4,3,3,1,1,2,1,1,1,1,2,1,1,5,6,4)
v5 <- c(1,1,1,1,1,3,3,3,3,3,1,1,1,1,1,6,4,5)
v6 <- c(1,1,1,2,1,3,3,3,4,3,1,1,1,2,1,6,5,4)
m1 <- cbind(v1,v2,v3,v4,v5,v6)
cor(m1)
factanal(m1, factors = 3) # varimax is the default
factanal(m1, factors = 3, rotation = "promax")
# The following shows the g factor as PC1
```

```
prcomp(m1) # signs may depend on platform

## formula interface
factanal(~v1+v2+v3+v4+v5+v6, factors = 3,
         scores = "Bartlett")$scores

## a realistic example from Bartholomew (1987, pp. 61-65)
utils::example(ability.cov)
```

`factor.scope` *Compute Allowed Changes in Adding to or Dropping from a Formula*

Description

`add.scope` and `drop.scope` compute those terms that can be individually added to or dropped from a model while respecting the hierarchy of terms.

Usage

```
add.scope(terms1, terms2)

drop.scope(terms1, terms2)

factor.scope(factor, scope)
```

Arguments

`terms1`	the terms or formula for the base model.
`terms2`	the terms or formula for the upper (`add.scope`) or lower (`drop.scope`) scope. If missing for `drop.scope` it is taken to be the null formula, so all terms (except any intercept) are candidates to be dropped.
`factor`	the `"factor"` attribute of the terms of the base object.
`scope`	a list with one or both components `drop` and `add` giving the `"factor"` attribute of the lower and upper scopes respectively.

Details

`factor.scope` is not intended to be called directly by users.

Value

For `add.scope` and `drop.scope` a character vector of terms labels. For `factor.scope`, a list with components `drop` and `add`, character vectors of terms labels.

See Also

`add1`, `drop1`, `aov`, `lm`

Examples

```
add.scope( ~ a + b + c + a:b,  ~ (a + b + c)^3)
# [1] "a:c" "b:c"
drop.scope( ~ a + b + c + a:b)
# [1] "c"   "a:b"
```

`family` *Family Objects for Models*

Description

Family objects provide a convenient way to specify the details of the models used by functions such as `glm`. See the documentation for `glm` for the details on how such model fitting takes place.

Usage

```
family(object, ...)

binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

Arguments

`link`	a specification for the model link function. This can be a name/expression, a literal character string, a length-one character vector, or an object of class `"link-glm"` (such as generated by `make.link`) provided it is not specified *via* one of the standard names given next. The `gaussian` family accepts the links (as names) `identity`, `log` and `inverse`; the `binomial` family the links `logit`, `probit`, `cauchit`, (corresponding to logistic, normal and Cauchy CDFs respectively) `log` and `cloglog` (complementary log-log); the `Gamma` family the links `inverse`, `identity` and `log`; the `poisson` family the links `log`, `identity`, and `sqrt`; and the `inverse.gaussian` family the links `1/mu^2`, `inverse`, `identity` and `log`. The `quasi` family accepts the links `logit`, `probit`, `cloglog`, `identity`, `inverse`, `log`, `1/mu^2` and `sqrt`, and the function `power` can be used to create a power link function.
`variance`	for all families other than `quasi`, the variance function is determined by the family. The `quasi` family will accept the literal character string (or unquoted as a name/expression) specifications `"constant"`, `"mu(1-mu)"`, `"mu"`, `"mu^2"` and `"mu^3"`, a length-one character vector taking one of those values, or a list containing components `varfun`, `validmu`, `dev.resids`, `initialize` and `name`.

object	the function family accesses the family objects which are stored within objects created by modelling functions (e.g., glm).
...	further arguments passed to methods.

Details

family is a generic function with methods for classes "glm" and "lm" (the latter returning gaussian()).

For the binomial and quasibinomial families the response can be specified in one of three ways:

1. As a factor: 'success' is interpreted as the factor not having the first level (and hence usually of having the second level).
2. As a numerical vector with values between 0 and 1, interpreted as the proportion of successful cases (with the total number of cases given by the weights).
3. As a two-column integer matrix: the first column gives the number of successes and the second the number of failures.

The quasibinomial and quasipoisson families differ from the binomial and poisson families only in that the dispersion parameter is not fixed at one, so they can model over-dispersion. For the binomial case see McCullagh and Nelder (1989, pp. 124–8). Although they show that there is (under some restrictions) a model with variance proportional to mean as in the quasi-binomial model, note that glm does not compute maximum-likelihood estimates in that model. The behaviour of S is closer to the quasi- variants.

Value

An object of class "family" (which has a concise print method). This is a list with elements

family	character: the family name.
link	character: the link name.
linkfun	function: the link.
linkinv	function: the inverse of the link function.
variance	function: the variance as a function of the mean.
dev.resids	function giving the deviance for each observation as a function of (y, mu, wt), used by the residuals method when computing deviance residuals.
aic	function giving the AIC value if appropriate (but NA for the quasi- families). More precisely, this function returns $-2\ell + 2s$, where ℓ is the log-likelihood and s is the number of estimated scale parameters. Note that the penalty term for the location parameters (typically the "regression coefficients") is added elsewhere, e.g., in glm.fit(), or AIC(), see the AIC example in glm. See logLik for the assumptions made about the dispersion parameter.
mu.eta	function: derivative of the inverse-link function with respect to the linear predictor. If the inverse-link function is $\mu = g^{-1}(\eta)$ where η is the value of the linear predictor, then this function returns $d(g^{-1})/d\eta = d\mu/d\eta$.
initialize	expression. This needs to set up whatever data objects are needed for the family as well as n (needed for AIC in the binomial family) and mustart (see glm).

`validmu` logical function. Returns `TRUE` if a mean vector `mu` is within the domain of `variance`.

`valideta` logical function. Returns `TRUE` if a linear predictor `eta` is within the domain of `linkinv`.

`simulate` (optional) function `simulate(object, nsim)` to be called by the `"lm"` method of `simulate`. It will normally return a matrix with `nsim` columns and one row for each fitted value, but it can also return a list of length `nsim`. Clearly this will be missing for 'quasi-' families.

`dispersion` (optional since R version 4.3.0) numeric: value of the dispersion parameter, if fixed, or `NA_real_` if free.

Note

The `link` and `variance` arguments have rather awkward semantics for back-compatibility. The recommended way is to supply them as quoted character strings, but they can also be supplied unquoted (as names or expressions). Additionally, they can be supplied as a length-one character vector giving the name of one of the options, or as a list (for `link`, of class `"link-glm"`). The restrictions apply only to links given as names: when given as a character string all the links known to `make.link` are accepted.

This is potentially ambiguous: supplying `link = logit` could mean the unquoted name of a link or the value of object `logit`. It is interpreted if possible as the name of an allowed link, then as an object. (You can force the interpretation to always be the value of an object via `logit[1]`.)

Author(s)

The design was inspired by S functions of the same names described in Hastie & Pregibon (1992) (except `quasibinomial` and `quasipoisson`).

References

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Dobson, A. J. (1983) *An Introduction to Statistical Modelling.* London: Chapman and Hall.

Cox, D. R. and Snell, E. J. (1981). *Applied Statistics; Principles and Examples.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`glm`, `power`, `make.link`.

For binomial *coefficients*, `choose`; the binomial and negative binomial *distributions*, `Binomial`, and `NegBinomial`.

Examples

```
require(utils) # for str

nf <- gaussian()  # Normal family
nf
str(nf)

gf <- Gamma()
gf
str(gf)
gf$linkinv
gf$variance(-3:4) #- == (.)^2

## Binomial with default 'logit' link:  Check some properties visually:
bi <- binomial()
et <- seq(-10,10, by=1/8)
plot(et, bi$mu.eta(et), type="l")
## show that mu.eta() is derivative of linkinv() :
lines((et[-1]+et[-length(et)])/2, col=adjustcolor("red", 1/4),
      diff(bi$linkinv(et))/diff(et), type="l", lwd=4)
## which here is the logistic density:
lines(et, dlogis(et), lwd=3, col=adjustcolor("blue", 1/4))
stopifnot(exprs = {
  all.equal(bi$ mu.eta(et), dlogis(et))
  all.equal(bi$linkinv(et), plogis(et) -> m)
  all.equal(bi$linkfun(m ), qlogis(m))    #  logit(.) == qlogis(.) !
})

## Data from example(glm) :
d.AD <- data.frame(treatment = gl(3,3),
                   outcome   = gl(3,1,9),
                   counts    = c(18,17,15, 20,10,20, 25,13,12))
glm.D93 <- glm(counts ~ outcome + treatment, d.AD, family = poisson())
## Quasipoisson: compare with above / example(glm) :
glm.qD93 <- glm(counts ~ outcome + treatment, d.AD, family = quasipoisson())

glm.qD93
anova  (glm.qD93, test = "F")
summary(glm.qD93)
## for Poisson results (same as from 'glm.D93' !) use
anova  (glm.qD93, dispersion = 1, test = "Chisq")
summary(glm.qD93, dispersion = 1)



## Example of user-specified link, a logit model for p^days
## See Shaffer, T.  2004. Auk 121(2): 526-540.
logexp <- function(days = 1)
{
    linkfun <- function(mu) qlogis(mu^(1/days))
    linkinv <- function(eta) plogis(eta)^days
    mu.eta  <- function(eta) days * plogis(eta)^(days-1) *
```

```
                binomial()$mu.eta(eta)
    valideta <- function(eta) TRUE
    link <- paste0("logexp(", days, ")")
    structure(list(linkfun = linkfun, linkinv = linkinv,
                   mu.eta = mu.eta, valideta = valideta, name = link),
              class = "link-glm")
}
(bil3 <- binomial(logexp(3)))

## in practice this would be used with a vector of 'days', in
## which case use an offset of 0 in the corresponding formula
## to get the null deviance right.

## Binomial with identity link: often not a good idea, as both
## computationally and conceptually difficult:
binomial(link = "identity")  ## is exactly the same as
binomial(link = make.link("identity"))



## tests of quasi
x <- rnorm(100)
y <- rpois(100, exp(1+x))
glm(y ~ x, family = quasi(variance = "mu", link = "log"))
# which is the same as
glm(y ~ x, family = poisson)
glm(y ~ x, family = quasi(variance = "mu^2", link = "log"))
## Not run: glm(y ~ x, family = quasi(variance = "mu^3", link = "log")) # fails
y <- rbinom(100, 1, plogis(x))
# need to set a starting value for the next fit
glm(y ~ x, family = quasi(variance = "mu(1-mu)", link = "logit"), start = c(0,1))
```

FDist *The F Distribution*

Description

Density, distribution function, quantile function and random generation for the F distribution with df1 and df2 degrees of freedom (and optional non-centrality parameter ncp).

Usage

```
df(x, df1, df2, ncp, log = FALSE)
pf(q, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2, ncp)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
df1, df2	degrees of freedom. Inf is allowed.
ncp	non-centrality parameter. If omitted the central F is assumed.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The F distribution with df1 = ν_1 and df2 = ν_2 degrees of freedom has density

$$f(x) = \frac{\Gamma(\nu_1/2 + \nu_2/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \left(\frac{\nu_1}{\nu_2}\right)^{\nu_1/2} x^{\nu_1/2-1} \left(1 + \frac{\nu_1 x}{\nu_2}\right)^{-(\nu_1+\nu_2)/2}$$

for $x > 0$.

The F distribution's cumulative distribution function (cdf), F_{ν_1,ν_2} fulfills (Abramowitz & Stegun 26.6.2, p.946) $F_{\nu_1,\nu_2}(qF) = 1 - I_x(\nu_2/2, \nu_1/2) = I_{1-x}(\nu_1/2, \nu_2/2)$, where $x := \frac{\nu_2}{\nu_2+\nu_1*qF}$, and $I_x(a, b)$ is the incomplete beta function; in R, = pbeta(x, a,b).

It is the distribution of the ratio of the mean squares of ν_1 and ν_2 independent standard normals, and hence of the ratio of two independent chi-squared variates each divided by its degrees of freedom. Since the ratio of a normal and the root mean-square of m independent normals has a Student's t_m distribution, the square of a t_m variate has a F distribution on 1 and m degrees of freedom.

The non-central F distribution is again the ratio of mean squares of independent normals of unit variance, but those in the numerator are allowed to have non-zero means and ncp is the sum of squares of the means. See Chisquare for further details on non-central distributions.

Value

df gives the density, pf gives the distribution function qf gives the quantile function, and rf generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rf, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

Supplying ncp = 0 uses the algorithm for the non-central distribution, which is not the same algorithm used if ncp is omitted. This is to give consistent behaviour in extreme cases with values of ncp very near zero.

The code for non-zero ncp is principally intended to be used for moderate values of ncp: it will not be highly accurate, especially in the tails, for large values.

Source

For the central case of df, computed *via* a binomial probability, code contributed by Catherine Loader (see dbinom); for the non-central case computed *via* dbeta, code contributed by Peter Ruckdeschel.

For pf, *via* pbeta (or for large df2, *via* pchisq).

For qf, *via* qchisq for large df2, else *via* qbeta.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 27 and 30. Wiley, New York.

See Also

Distributions for other standard distributions, including dchisq for chi-squared and dt for Student's t distributions.

Examples

```
## Equivalence of pt(.,nu) with pf(.^2, 1,nu):
x <- seq(0.001, 5, length.out = 100)
nu <- 4
stopifnot(all.equal(2*pt(x,nu) - 1, pf(x^2, 1,nu)),
          ## upper tails:
    all.equal(2*pt(x,     nu, lower.tail=FALSE),
        pf(x^2, 1,nu, lower.tail=FALSE)))

## the density of the square of a t_m is 2*dt(x, m)/(2*x)
# check this is the same as the density of F_{1,m}
all.equal(df(x^2, 1, 5), dt(x, 5)/x)

## Identity (F <-> t):  qf(2*p - 1, 1, df) == qt(p, df)^2  for  p >= 1/2
p <- seq(1/2, .99, length.out = 50); df <- 10
rel.err <- function(x, y) ifelse(x == y, 0, abs(x-y)/mean(abs(c(x,y))))
stopifnot(all.equal(qf(2*p - 1, df1 = 1, df2 = df),
                    qt(p, df)^2))

## Identity (F <-> Beta <-> incompl.beta):
n1 <- 7 ; n2 <- 12; qF <- c((0:4)/4, 1.5, 2:16)
x <- n2/(n2 + n1*qF)
stopifnot(all.equal(pf(qF, n1, n2, lower.tail=FALSE),
                    pbeta(x, n2/2, n1/2)))
```

`fft` *Fast Discrete Fourier Transform (FFT)*

Description

Computes the Discrete Fourier Transform (DFT) of an array with a fast algorithm, the "Fast Fourier Transform" (FFT).

Usage

```
fft(z, inverse = FALSE)
mvfft(z, inverse = FALSE)
```

Arguments

`z`	a real or complex array containing the values to be transformed. Long vectors are not supported.
`inverse`	if `TRUE`, the unnormalized inverse transform is computed (the inverse has a `+` in the exponent of e, but here, we do *not* divide by `1/length(x)`).

Value

When `z` is a vector, the value computed and returned by `fft` is the unnormalized univariate discrete Fourier transform of the sequence of values in `z`. Specifically, `y <- fft(z)` returns

$$y[h] = \sum_{k=1}^{n} z[k] \exp(-2\pi i(k-1)(h-1)/n)$$

for $h = 1, \ldots, n$ where `n = length(y)`. If `inverse` is `TRUE`, $\exp(-2\pi \ldots)$ is replaced with $\exp(2\pi \ldots)$.

When `z` contains an array, `fft` computes and returns the multivariate (spatial) transform. If `inverse` is `TRUE`, the (unnormalized) inverse Fourier transform is returned, i.e., if `y <- fft(z)`, then `z` is `fft(y, inverse = TRUE) / length(y)`.

By contrast, `mvfft` takes a real or complex matrix as argument, and returns a similar shaped matrix, but with each column replaced by its discrete Fourier transform. This is useful for analyzing vector-valued series.

The FFT is fastest when the length of the series being transformed is highly composite (i.e., has many factors). If this is not the case, the transform may take a long time to compute and will use a large amount of memory.

Source

Uses C translation of Fortran code in Singleton (1979).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Singleton, R. C. (1979). Mixed Radix Fast Fourier Transforms, in *Programs for Digital Signal Processing*, IEEE Digital Signal Processing Committee eds. IEEE Press.

Cooley, James W., and Tukey, John W. (1965). An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, **19**(90), 297–301. doi:10.2307/2003354.

See Also

convolve, nextn.

Examples

```
x <- 1:4
fft(x)
fft(fft(x), inverse = TRUE)/length(x)

## Slow Discrete Fourier Transform (DFT) - e.g., for checking the formula
fft0 <- function(z, inverse=FALSE) {
  n <- length(z)
  if(n == 0) return(z)
  k <- 0:(n-1)
  ff <- (if(inverse) 1 else -1) * 2*pi * 1i * k/n
  vapply(1:n, function(h) sum(z * exp(ff*(h-1))), complex(1))
}

relD <- function(x,y) 2* abs(x - y) / abs(x + y)
n <- 2^8
z <- complex(n, rnorm(n), rnorm(n))
## relative differences in the order of 4*10^{-14} :
summary(relD(fft(z), fft0(z)))
summary(relD(fft(z, inverse=TRUE), fft0(z, inverse=TRUE)))
```

---

filter *Linear Filtering on a Time Series*

Description

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series.

Usage

```
filter(x, filter, method = c("convolution", "recursive"),
       sides = 2, circular = FALSE, init)
```

Arguments

x	a univariate or multivariate time series.
filter	a vector of filter coefficients in reverse time order (as for AR or MA coefficients).
method	Either "convolution" or "recursive" (and can be abbreviated). If "convolution" a moving average is used: if "recursive" an autoregression is used.
sides	for convolution filters only. If sides = 1 the filter coefficients are for past values only; if sides = 2 they are centred around lag 0. In this case the length of the filter should be odd, but if it is even, more of the filter is forward in time than backward.
circular	for convolution filters only. If TRUE, wrap the filter around the ends of the series, otherwise assume external values are missing (NA).
init	for recursive filters only. Specifies the initial values of the time series just prior to the start value, in reverse time order. The default is a set of zeros.

Details

Missing values are allowed in x but not in filter (where they would lead to missing values everywhere in the output).

Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y_i = x_i + f_1 y_{i-1} + \cdots + f_p y_{i-p}$$

No check is made to see if recursive filter is invertible: the output may diverge if it is not.

The convolution filter is

$$y_i = f_1 x_{i+o} + \cdots + f_p x_{i+o-(p-1)}$$

where o is the offset: see sides for how it is determined.

Value

A time series object.

Note

convolve(, type = "filter") uses the FFT for computations and so *may* be faster for long filters on univariate series, but it does not return a time series (and so the time alignment is unclear), nor does it handle missing values. filter is faster for a filter of length 100 on a series of length 1000, for example.

See Also

convolve, arima.sim

Examples

```
x <- 1:100
filter(x, rep(1, 3))
filter(x, rep(1, 3), sides = 1)
filter(x, rep(1, 3), sides = 1, circular = TRUE)

filter(presidents, rep(1, 3))
```

`fisher.test` *Fisher's Exact Test for Count Data*

Description

Performs Fisher's exact test for testing the null of independence of rows and columns in a contingency table with fixed marginals.

Usage

```
fisher.test(x, y = NULL, workspace = 200000, hybrid = FALSE,
            hybridPars = c(expect = 5, percent = 80, Emin = 1),
            control = list(), or = 1, alternative = "two.sided",
            conf.int = TRUE, conf.level = 0.95,
            simulate.p.value = FALSE, B = 2000)
```

Arguments

`x`	either a two-dimensional contingency table in matrix form, or a factor object.
`y`	a factor object; ignored if `x` is a matrix.
`workspace`	an integer specifying the size of the workspace used in the network algorithm. In units of 4 bytes. Only used for non-simulated p-values larger than 2×2 tables. Since R version 3.5.0, this also increases the internal stack size which allows larger problems to be solved, however sometimes needing hours. In such cases, `simulate.p.values=TRUE` may be more reasonable.
`hybrid`	a logical. Only used for larger than 2×2 tables, in which cases it indicates whether the exact probabilities (default) or a hybrid approximation thereof should be computed.
`hybridPars`	a numeric vector of length 3, by default describing "Cochran's conditions" for the validity of the chi-squared approximation, see 'Details'.
`control`	a list with named components for low level algorithm control. At present the only one used is `"mult"`, a positive integer ≥ 2 with default 30 used only for larger than 2×2 tables. This says how many times as much space should be allocated to paths as to keys: see file '`fexact.c`' in the sources of this package.
`or`	the hypothesized odds ratio. Only used in the 2×2 case.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used in the 2×2 case.

`conf.int`	logical indicating if a confidence interval for the odds ratio in a 2×2 table should be computed (and returned).
`conf.level`	confidence level for the returned confidence interval. Only used in the 2×2 case and if `conf.int = TRUE`.
`simulate.p.value`	a logical indicating whether to compute p-values by Monte Carlo simulation, in larger than 2×2 tables.
`B`	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If `x` is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both `x` and `y` must be vectors or factors of the same length. Incomplete cases are removed, vectors are coerced into factor objects, and the contingency table is computed from these.

For 2×2 cases, p-values are obtained directly using the (central or non-central) hypergeometric distribution. Otherwise, computations are based on a C version of the FORTRAN subroutine `FEXACT` which implements the network developed by Mehta and Patel (1983, 1986) and improved by Clarkson, Fan and Joe (1993). The FORTRAN code can be obtained from https://netlib.org/toms/643. Note this fails (with an error message) when the entries of the table are too large. (It transposes the table if necessary so it has no more rows than columns. One constraint is that the product of the row marginals be less than $2^{31} - 1$.)

For 2×2 tables, the null of conditional independence is equivalent to the hypothesis that the odds ratio equals one. 'Exact' inference can be based on observing that in general, given all marginal totals fixed, the first element of the contingency table has a non-central hypergeometric distribution with non-centrality parameter given by the odds ratio (Fisher, 1935). The alternative for a one-sided test is based on the odds ratio, so `alternative = "greater"` is a test of the odds ratio being bigger than `or`.

Two-sided tests are based on the probabilities of the tables, and take as 'more extreme' all tables with probabilities less than or equal to that of the observed table, the p-value being the sum of such probabilities.

For larger than 2×2 tables and `hybrid = TRUE`, asymptotic chi-squared probabilities are only used if the 'Cochran conditions' (or modified version thereof) specified by `hybridPars = c(expect = 5, percent = 80, Emin = 1)` are satisfied, that is if no cell has expected counts less than 1 (= `Emin`) and more than 80% (= `percent`) of the cells have expected counts at least 5 (= `expect`), otherwise the exact calculation is used. A corresponding `if()` decision is made for all sub-tables considered. Accidentally, R has used `180` instead of `80` as `percent`, i.e., `hybridPars[2]` in R versions between 3.0.0 and 3.4.1 (inclusive), i.e., the 2nd of the `hybridPars` (all of which used to be hard-coded previous to R 3.5.0). Consequently, in these versions of R, `hybrid=TRUE` never made a difference.

In the $r \times c$ case with $r > 2$ or $c > 2$, internal tables can get too large for the exact test in which case an error is signalled. Apart from increasing `workspace` sufficiently, which then may lead to very long running times, using `simulate.p.value = TRUE` may then often be sufficient and hence advisable.

Simulation is done conditional on the row and column marginals, and works only if the marginals are strictly positive. (A C translation of the algorithm of Patefield (1981) is used.) Note that the default number of replicates (`B = 2000`) implies a minimum p-value of about 0.0005 ($1/(B + 1)$).

Value

A list with class "htest" containing the following components:

p.value	the p-value of the test.
conf.int	a confidence interval for the odds ratio. Only present in the 2×2 case and if argument conf.int = TRUE.
estimate	an estimate of the odds ratio. Note that the *conditional* Maximum Likelihood Estimate (MLE) rather than the unconditional MLE (the sample odds ratio) is used. Only present in the 2×2 case.
null.value	the odds ratio under the null, or. Only present in the 2×2 case.
alternative	a character string describing the alternative hypothesis.
method	the character string "Fisher's Exact Test for Count Data".
data.name	a character string giving the name(s) of the data.

References

Agresti, A. (1990). *Categorical data analysis*. New York: Wiley. Pages 59–66.

Agresti, A. (2002). *Categorical data analysis*. Second edition. New York: Wiley. Pages 91–101.

Fisher, R. A. (1935). The logic of inductive inference. *Journal of the Royal Statistical Society Series A*, **98**, 39–54. doi:10.2307/2342435.

Fisher, R. A. (1962). Confidence limits for a cross-product ratio. *Australian Journal of Statistics*, **4**, 41. doi:10.1111/j.1467842X.1962.tb00285.x.

Fisher, R. A. (1970). *Statistical Methods for Research Workers*. Oliver & Boyd.

Mehta, Cyrus R. and Patel, Nitin R. (1983). A network algorithm for performing Fisher's exact test in $r \times c$ contingency tables. *Journal of the American Statistical Association*, **78**, 427–434. doi:10.1080/01621459.1983.10477989.

Mehta, C. R. and Patel, N. R. (1986). Algorithm 643: FEXACT, a FORTRAN subroutine for Fisher's exact test on unordered $r \times c$ contingency tables. *ACM Transactions on Mathematical Software*, **12**, 154–161. doi:10.1145/6497.214326.

Clarkson, D. B., Fan, Y. and Joe, H. (1993) A Remark on Algorithm 643: FEXACT: An Algorithm for Performing Fisher's Exact Test in $r \times c$ Contingency Tables. *ACM Transactions on Mathematical Software*, **19**, 484–488. doi:10.1145/168173.168412.

Patefield, W. M. (1981). Algorithm AS 159: An efficient method of generating r x c tables with given row and column totals. *Applied Statistics*, **30**, 91–97. doi:10.2307/2346669.

See Also

chisq.test

fisher.exact in package **exact2x2** for alternative interpretations of two-sided tests and confidence intervals for 2×2 tables.

Examples

```
## Agresti (1990, p. 61f; 2002, p. 91) Fisher's Tea Drinker
## A British woman claimed to be able to distinguish whether milk or
##  tea was added to the cup first.  To test, she was given 8 cups of
##  tea, in four of which milk was added first.  The null hypothesis
##  is that there is no association between the true order of pouring
##  and the woman's guess, the alternative that there is a positive
##  association (that the odds ratio is greater than 1).
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nrow = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
fisher.test(TeaTasting, alternative = "greater")
## => p = 0.2429, association could not be established

## Fisher (1962, 1970), Criminal convictions of like-sex twins
Convictions <- matrix(c(2, 10, 15, 3), nrow = 2,
               dimnames =
        list(c("Dizygotic", "Monozygotic"),
      c("Convicted", "Not convicted")))
Convictions
fisher.test(Convictions, alternative = "less")
fisher.test(Convictions, conf.int = FALSE)
fisher.test(Convictions, conf.level = 0.95)$conf.int
fisher.test(Convictions, conf.level = 0.99)$conf.int

## A r x c table  Agresti (2002, p. 57) Job Satisfaction
Job <- matrix(c(1,2,1,0, 3,3,6,1, 10,10,14,9, 6,7,12,11), 4, 4,
           dimnames = list(income = c("< 15k", "15-25k", "25-40k", "> 40k"),
                      satisfaction = c("VeryD", "LittleD", "ModerateS", "VeryS")))
fisher.test(Job) # 0.7827
fisher.test(Job, simulate.p.value = TRUE, B = 1e5) # also close to 0.78

## 6th example in Mehta & Patel's JASA paper
MP6 <- rbind(
        c(1,2,2,1,1,0,1),
        c(2,0,0,2,3,0,0),
        c(0,1,1,1,2,7,3),
        c(1,1,2,0,0,0,1),
        c(0,1,1,1,1,0,0))
fisher.test(MP6)
# Exactly the same p-value, as Cochran's conditions are never met:
fisher.test(MP6, hybrid=TRUE)
```

`fitted` *Extract Model Fitted Values*

Description

`fitted` is a generic function which extracts fitted values from objects returned by modeling functions. `fitted.values` is an alias for it.

All object classes which are returned by model fitting functions should provide a `fitted` method. (Note that the generic is `fitted` and not `fitted.values`.)

Methods can make use of `napredict` methods to compensate for the omission of missing values. The default and `nls` methods do.

Usage

```
fitted(object, ...)
fitted.values(object, ...)
```

Arguments

`object`	an object for which the extraction of model fitted values is meaningful.
`...`	other arguments.

Value

Fitted values extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`coefficients`, `glm`, `lm`, `residuals`.

`fivenum` *Tukey Five-Number Summaries*

Description

Returns Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum) for the input data.

Usage

```
fivenum(x, na.rm = TRUE)
```

Arguments

`x`	numeric, maybe including `NA`s and $\pm$`Inf`s.
`na.rm`	logical; if `TRUE`, all `NA` and `NaN`s are dropped, before the statistics are computed.

Value

A numeric vector of length 5 containing the summary information. See boxplot.stats for more details.

See Also

IQR, boxplot.stats, median, quantile, range.

Examples

```
fivenum(c(rnorm(100), -1:1/0))
```

---

fligner.test *Fligner-Killeen Test of Homogeneity of Variances*

Description

Performs a Fligner-Killeen (median) test of the null that the variances in each of the groups (samples) are the same.

Usage

```
fligner.test(x, ...)

## Default S3 method:
fligner.test(x, g, ...)

## S3 method for class 'formula'
fligner.test(formula, data, subset, na.action, ...)
```

Arguments

x	a numeric vector of data values, or a list of numeric data vectors.
g	a vector or factor object giving the group for the corresponding elements of x. Ignored if x is a list.
formula	a formula of the form lhs ~ rhs where lhs gives the data values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

If x is a list, its elements are taken as the samples to be compared for homogeneity of variances, and hence have to be numeric data vectors. In this case, g is ignored, and one can simply use fligner.test(x) to perform the test. If the samples are not yet contained in a list, use fligner.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

The Fligner-Killeen (median) test has been determined in a simulation study as one of the many tests for homogeneity of variances which is most robust against departures from normality, see Conover, Johnson & Johnson (1981). It is a k-sample simple linear rank which uses the ranks of the absolute values of the centered samples and weights $a(i) = \mathrm{qnorm}((1+i/(n+1))/2)$. The version implemented here uses median centering in each of the samples (F-K:med X^2 in the reference).

Value

A list of class "htest" containing the following components:

statistic	the Fligner-Killeen:med X^2 test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Fligner-Killeen test of homogeneity of variances".
data.name	a character string giving the names of the data.

References

William J. Conover, Mark E. Johnson and Myrle M. Johnson (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data. *Technometrics*, **23**, 351–361. doi:10.2307/1268225.

See Also

ansari.test and mood.test for rank-based two-sample test for a difference in scale parameters; var.test and bartlett.test for parametric tests for the homogeneity of variances.

Examples

```
require(graphics)

plot(count ~ spray, data = InsectSprays)
fligner.test(InsectSprays$count, InsectSprays$spray)
fligner.test(count ~ spray, data = InsectSprays)
## Compare this to bartlett.test()
```

formula *Model Formulae*

Description

The generic function formula and its specific methods provide a way of extracting formulae which have been included in other objects.

as.formula is almost identical, additionally preserving attributes when object already inherits from "formula".

Usage

```
formula(x, ...)
DF2formula(x, env = parent.frame())
as.formula(object, env = parent.frame())

## S3 method for class 'formula'
print(x, showEnv = !identical(e, .GlobalEnv), ...)
```

Arguments

x, object	R object, for DF2formula() a data.frame.
...	further arguments passed to or from other methods.
env	the environment to associate with the result, if not already a formula.
showEnv	logical indicating if the environment should be printed as well.

Details

The models fitted by, e.g., the lm and glm functions are specified in a compact symbolic form. The ~ operator is basic in the formation of such models. An expression of the form y ~ model is interpreted as a specification that the response y is modelled by a linear predictor specified symbolically by model. Such a model consists of a series of terms separated by + operators. The terms themselves consist of variable and factor names separated by : operators. Such a term is interpreted as the interaction of all the variables and factors appearing in the term.

In addition to + and :, a number of other operators are useful in model formulae.

- The * operator denotes factor crossing: a*b is interpreted as a + b + a:b.
- The ^ operator indicates crossing to the specified degree. For example (a+b+c)^2 is identical to (a+b+c)*(a+b+c) which in turn expands to a formula containing the main effects for a, b and c together with their second-order interactions.
- The %in% operator indicates that the terms on its left are nested within those on the right. For example a + b %in% a expands to the formula a + a:b.
- The / operator provides a shorthand, so that a / b is equivalent to a + b %in% a.

- The - operator removes the specified terms, hence `(a+b+c)^2 - a:b` is identical to `a + b + c + b:c + a:c`. It can also used to remove the intercept term: when fitting a linear model `y ~ x - 1` specifies a line through the origin. A model with no intercept can be also specified as `y ~ x + 0` or `y ~ 0 + x`.

While formulae usually involve just variable and factor names, they can also involve arithmetic expressions. The formula `log(y) ~ a + log(x)` is quite legal. When such arithmetic expressions involve operators which are also used symbolically in model formulae, there can be confusion between arithmetic and symbolic operator use.

To avoid this confusion, the function `I()` can be used to bracket those portions of a model formula where the operators are used in their arithmetic sense. For example, in the formula `y ~ a + I(b+c)`, the term `b+c` is to be interpreted as the sum of `b` and `c`.

Variable names can be quoted by backticks `` `like this` `` in formulae, although there is no guarantee that all code using formulae will accept such non-syntactic names.

Most model-fitting functions accept formulae with right-hand-side including the function `offset` to indicate terms with a fixed coefficient of one. Some functions accept other 'specials' such as `strata` or `cluster` (see the `specials` argument of `terms.formula`).

There are two special interpretations of `.` in a formula. The usual one is in the context of a `data` argument of model fitting functions and means 'all columns not otherwise in the formula': see `terms.formula`. In the context of `update.formula`, **only**, it means 'what was previously in this part of the formula'.

When `formula` is called on a fitted model object, either a specific method is used (such as that for class `"nls"`) or the default method. The default first looks for a `"formula"` component of the object (and evaluates it), then a `"terms"` component, then a `formula` parameter of the call (and evaluates its value) and finally a `"formula"` attribute.

There is a `formula` method for data frames. When there's `"terms"` attribute with a formula, e.g., for a `model.frame()`, that formula is returned. If you'd like the previous (R $\leq$ 3.5.x) behavior, use the auxiliary `DF2formula()` which does not consider a `"terms"` attribute. Otherwise, if there is only one column this forms the RHS with an empty LHS. For more columns, the first column is the LHS of the formula and the remaining columns separated by + form the RHS.

Value

All the functions above produce an object of class `"formula"` which contains a symbolic model formula.

Environments

A formula object has an associated environment, and this environment (rather than the parent environment) is used by `model.frame` to evaluate variables that are not found in the supplied `data` argument.

Formulas created with the `~` operator use the environment in which they were created. Formulas created with `as.formula` will use the `env` argument for their environment.

Note

In R versions up to 3.6.0, `character x` of length more than one were parsed as separate lines of R code and the first complete expression was evaluated into a formula when possible. This silently

truncates such vectors of characters inefficiently and to some extent inconsistently as this behaviour had been undocumented. For this reason, such use has been deprecated. If you must work via character x, do use a string, i.e., a character vector of length one.

E.g., `eval(call("~", quote(foo + bar)))` has been an order of magnitude more efficient than `formula(c("~", "foo + bar"))`.

Further, character "expressions" needing an `eval()` to return a formula are now deprecated.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`~`, `I`, `offset`.

For formula manipulation: `update.formula`, `terms.formula`, and `all.vars`. For typical use: `lm`, `glm`, and `coplot`. For formula construction: `reformulate`.

Examples

```
class(fo <- y ~ x1*x2) # "formula"
fo
typeof(fo)  # R internal : "language"
terms(fo)

environment(fo)
environment(as.formula("y ~ x"))
environment(as.formula("y ~ x", env = new.env()))


## Create a formula for a model with a large number of variables:
xnam <- paste0("x", 1:25)
(fmla <- as.formula(paste("y ~ ", paste(xnam, collapse= "+"))))
## Equivalent with reformulate():
fmla2 <- reformulate(xnam, response = "y")
stopifnot(identical(fmla, fmla2))
```

---

formula.nls *Extract Model Formula from* nls *Object*

Description

Returns the model used to fit `object`.

Usage

```
## S3 method for class 'nls'
formula(x, ...)
```

Arguments

x	an object inheriting from class "nls", representing a nonlinear least squares fit.
...	further arguments passed to or from other methods.

Value

a formula representing the model used to obtain object.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, formula

Examples

```
fm1 <- nls(circumference ~ A/(1+exp((B-age)/C)), Orange,
           start = list(A = 160, B = 700, C = 350))
formula(fm1)
```

---

friedman.test *Friedman Rank Sum Test*

Description

Performs a Friedman rank sum test with unreplicated blocked data.

Usage

```
friedman.test(y, ...)

## Default S3 method:
friedman.test(y, groups, blocks, ...)

## S3 method for class 'formula'
friedman.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.

formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

friedman.test can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in y for each combination of levels of groups and blocks) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of blocks, the location parameter of y is the same in each of the groups.

If y is a matrix, groups and blocks are obtained from the column and row indices, respectively. NA's are not allowed in groups or blocks; if y contains NA's, corresponding blocks are removed.

Value

A list with class "htest" containing the following components:

statistic	the value of Friedman's chi-squared statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Friedman rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander and Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 139–146.

See Also

quade.test.

Examples

```
## Hollander & Wolfe (1973), p. 140ff.
## Comparison of three methods ("round out", "narrow angle", and
##  "wide angle") for rounding first base.  For each of 18 players
##  and the three method, the average time of two runs from a point on
##  the first base line 35ft from home plate to a point 15ft short of
```

```
##  second base is recorded.
RoundingTimes <-
matrix(c(5.40, 5.50, 5.55,
         5.85, 5.70, 5.75,
         5.20, 5.60, 5.50,
         5.55, 5.50, 5.40,
         5.90, 5.85, 5.70,
         5.45, 5.55, 5.60,
         5.40, 5.40, 5.35,
         5.45, 5.50, 5.35,
         5.25, 5.15, 5.00,
         5.85, 5.80, 5.70,
         5.25, 5.20, 5.10,
         5.65, 5.55, 5.45,
         5.60, 5.35, 5.45,
         5.05, 5.00, 4.95,
         5.50, 5.50, 5.40,
         5.45, 5.55, 5.50,
         5.55, 5.55, 5.35,
         5.45, 5.50, 5.55,
         5.50, 5.45, 5.25,
         5.65, 5.60, 5.40,
         5.70, 5.65, 5.55,
         6.30, 6.30, 6.25),
       nrow = 22,
       byrow = TRUE,
       dimnames = list(1 : 22,
                       c("Round Out", "Narrow Angle", "Wide Angle")))
friedman.test(RoundingTimes)
## => strong evidence against the null that the methods are equivalent
##    with respect to speed

wb <- aggregate(warpbreaks$breaks,
                by = list(w = warpbreaks$wool,
                          t = warpbreaks$tension),
                FUN = mean)
wb
friedman.test(wb$x, wb$w, wb$t)
friedman.test(x ~ w | t, data = wb)
```

`ftable` *Flat Contingency Tables*

Description

Create 'flat' contingency tables.

Usage

```
ftable(x, ...)
```

```
## Default S3 method:
ftable(..., exclude = c(NA, NaN), row.vars = NULL,
       col.vars = NULL)
```

Arguments

x, ...	R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, or a contingency table object of class "table" or "ftable".
exclude	values to use in the exclude argument of factor when interpreting non-factor objects.
row.vars	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the rows of the flat contingency table.
col.vars	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the columns of the flat contingency table.

Details

ftable creates 'flat' contingency tables. Similar to the usual contingency tables, these contain the counts of each combination of the levels of the variables (factors) involved. This information is then re-arranged as a matrix whose rows and columns correspond to unique combinations of the levels of the row and column variables (as specified by row.vars and col.vars, respectively). The combinations are created by looping over the variables in reverse order (so that the levels of the left-most variable vary the slowest). Displaying a contingency table in this flat matrix form (via print.ftable, the print method for objects of class "ftable") is often preferable to showing it as a higher-dimensional array.

ftable is a generic function. Its default method, ftable.default, first creates a contingency table in array form from all arguments except row.vars and col.vars. If the first argument is of class "table", it represents a contingency table and is used as is; if it is a flat table of class "ftable", the information it contains is converted to the usual array representation using as.table. Otherwise, the arguments should be R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, which are cross-tabulated using table. Then, the arguments row.vars and col.vars are used to collapse the contingency table into flat form. If neither of these two is given, the last variable is used for the columns. If both are given and their union is a proper subset of all variables involved, the other variables are summed out.

When the arguments are R expressions interpreted as factors, additional arguments will be passed to table to control how the variable names are displayed; see the last example below.

Function ftable.formula provides a formula method for creating flat contingency tables.

There are methods for as.table, as.matrix and as.data.frame.

Value

ftable returns an object of class "ftable", which is a matrix with counts of each combination of the levels of variables with information on the names and levels of the (row and columns) variables stored as attributes "row.vars" and "col.vars".

See Also

ftable.formula for the formula interface (which allows a data = . argument); read.ftable for information on reading, writing and coercing flat contingency tables; table for ordinary cross-tabulation; xtabs for formula-based cross-tabulation.

Examples

```
## Start with a contingency table.
ftable(Titanic, row.vars = 1:3)
ftable(Titanic, row.vars = 1:2, col.vars = "Survived")
ftable(Titanic, row.vars = 2:1, col.vars = "Survived")

## Start with a data frame.
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))

## Start with expressions, use table()'s "dnn" to change labels
ftable(mtcars$cyl, mtcars$vs, mtcars$am, mtcars$gear, row.vars = c(2, 4),
       dnn = c("Cylinders", "V/S", "Transmission", "Gears"))
```

---

ftable.formula *Formula Notation for Flat Contingency Tables*

Description

Produce or manipulate a flat contingency table using formula notation.

Usage

```
## S3 method for class 'formula'
ftable(formula, data = NULL, subset, na.action, ...)
```

Arguments

formula	a formula object with both left and right hand sides specifying the column and row variables of the flat table.
data	a data frame, list or environment (or similar: see model.frame) containing the variables to be cross-tabulated, or a contingency table (see below).
subset	an optional vector specifying a subset of observations to be used. Ignored if data is a contingency table.

na.action	a function which indicates what should happen when the data contain NAs. Ignored if data is a contingency table.
...	further arguments to the default ftable method may also be passed as arguments, see ftable.default.

Details

This is a method of the generic function ftable.

The left and right hand side of formula specify the column and row variables, respectively, of the flat contingency table to be created. Only the + operator is allowed for combining the variables. A . may be used once in the formula to indicate inclusion of all the remaining variables.

If data is an object of class "table" or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. Otherwise, if it is not a flat contingency table (i.e., an object of class "ftable"), it should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, na.action is applied to the data to handle missing values, and, after possibly selecting a subset of the data as specified by the subset argument, a contingency table is computed from the variables.

The contingency table is then collapsed to a flat table, according to the row and column variables specified by formula.

Value

A flat contingency table which contains the counts of each combination of the levels of the variables, collapsed into a matrix for suitably displaying the counts.

See Also

ftable, ftable.default; table.

Examples

```
Titanic
x <- ftable(Survived ~ ., data = Titanic)
x
ftable(Sex ~ Class + Age, data = x)
```

GammaDist *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

```
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
pgamma(q, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
qgamma(p, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
rgamma(n, shape, rate = 1, scale = 1/rate)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
rate	an alternative way to specify the scale.
shape, scale	shape and scale parameters. Must be positive, scale strictly.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If scale is omitted, it assumes the default value of 1.

The Gamma distribution with parameters shape $= \alpha$ and scale $= \sigma$ has density

$$f(x) = \frac{1}{\sigma^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\sigma}$$

for $x \geq 0$, $\alpha > 0$ and $\sigma > 0$. (Here $\Gamma(\alpha)$ is the function implemented by R's gamma() and defined in its help. Note that $a = 0$ corresponds to the trivial distribution with all mass at point 0.)

The mean and variance are $E(X) = \alpha\sigma$ and $Var(X) = \alpha\sigma^2$.

The cumulative hazard $H(t) = -\log(1 - F(t))$ is

```
-pgamma(t, ..., lower = FALSE, log = TRUE)
```

Note that for smallish values of shape (and moderate scale) a large parts of the mass of the Gamma distribution is on values of x so near zero that they will be represented as zero in computer arithmetic. So rgamma may well return values which will be represented as zero. (This will also happen for very large values of scale since the actual generation is done for scale = 1.)

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The S (Becker et al., 1988) parametrization was via `shape` and `rate`: S had no `scale` parameter. It is an error to supply both `scale` and `rate`.

`pgamma` is closely related to the incomplete gamma function. As defined by Abramowitz and Stegun 6.5.1 (and by 'Numerical Recipes') this is

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

$P(a,x)$ is `pgamma(x, a)`. Other authors (for example Karl Pearson in his 1922 tables) omit the normalizing factor, defining the incomplete gamma function $\gamma(a,x)$ as $\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt$, i.e., `pgamma(x, a) * gamma(a)`. Yet other use the 'upper' incomplete gamma function,

$$\Gamma(a,x) = \int_x^\infty t^{a-1} e^{-t} dt,$$

which can be computed by `pgamma(x, a, lower = FALSE) * gamma(a)`.

Note however that `pgamma(x, a, ..)` currently requires $a > 0$, whereas the incomplete gamma function is also defined for negative a. In that case, you can use `gamma_inc(a,x)` (for $\Gamma(a,x)$) from package **gsl**.

See also https://en.wikipedia.org/wiki/Incomplete_gamma_function, or https://dlmf.nist.gov/8.2#i.

Source

`dgamma` is computed via the Poisson density, using code contributed by Catherine Loader (see `dbinom`).

`pgamma` uses an unpublished (and not otherwise documented) algorithm 'mainly by Morten Welinder'.

`qgamma` is based on a C translation of

Best, D. J. and D. E. Roberts (1975). Algorithm AS91. Percentage points of the chi-squared distribution. *Applied Statistics*, **24**, 385–388.

plus a final Newton step to improve the approximation.

`rgamma` for `shape >= 1` uses

Ahrens, J. H. and Dieter, U. (1982). Generating gamma variates by a modified rejection technique. *Communications of the ACM*, **25**, 47–54,

and for `0 < shape < 1` uses

Ahrens, J. H. and Dieter, U. (1974). Computer methods for sampling from gamma, beta, Poisson and binomial distributions. *Computing*, **12**, 223–246.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Shea, B. L. (1988). Algorithm AS 239: Chi-squared and incomplete Gamma integral, *Applied Statistics (JRSS C)*, **37**, 466–473. doi:10.2307/2347328.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/, section 8.2.

See Also

gamma for the gamma function.

Distributions for other standard distributions, including dbeta for the Beta distribution and dchisq for the chi-squared distribution which is a special case of the Gamma distribution.

Examples

```
-log(dgamma(1:4, shape = 1))
p <- (1:9)/10
pgamma(qgamma(p, shape = 2), shape = 2)
1 - 1/exp(qgamma(p, shape = 1))

# even for shape = 0.001 about half the mass is on numbers
# that cannot be represented accurately (and most of those as zero)
pgamma(.Machine$double.xmin, 0.001)
pgamma(5e-324, 0.001)  # on most machines 5e-324 is the smallest
                       # representable non-zero number
table(rgamma(1e4, 0.001) == 0)/1e4
```

Geometric *The Geometric Distribution*

Description

Density, distribution function, quantile function and random generation for the geometric distribution with parameter prob.

Usage

```
dgeom(x, prob, log = FALSE)
pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)
qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)
rgeom(n, prob)
```

Arguments

x, q	vector of quantiles representing the number of failures in a sequence of Bernoulli trials before success occurs.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
prob	probability of success in each trial. 0 < prob <= 1.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The geometric distribution with `prob` $= p$ has density

$$p(x) = p(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $0 < p \leq 1$.

If an element of `x` is not integer, the result of `dgeom` is zero, with a warning.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

`dgeom` gives the density, `pgeom` gives the distribution function, `qgeom` gives the quantile function, and `rgeom` generates random deviates.

Invalid `prob` will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rgeom`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

`rgeom` returns a vector of type integer unless generated values exceed the maximum representable integer when `double` values are returned.

Source

`dgeom` computes via `dbinom`, using code contributed by Catherine Loader (see `dbinom`).

`pgeom` and `qgeom` are based on the closed-form formulae.

`rgeom` uses the derivation as an exponential mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for other standard distributions, including `dnbinom` for the negative binomial which generalizes the geometric distribution.

Examples

```
qgeom((1:9)/10, prob = .2)
Ni <- rgeom(20, prob = 1/4); table(factor(Ni, 0:max(Ni)))
```

getInitial *Get Initial Parameter Estimates*

Description

This function evaluates initial parameter estimates for a nonlinear regression model. If `data` is a parameterized data frame or `pframe` object, its `parameters` attribute is returned. Otherwise the object is examined to see if it contains a call to a `selfStart` object whose `initial` attribute can be evaluated.

Usage

```
getInitial(object, data, ...)
```

Arguments

`object`	a formula or a `selfStart` model that defines a nonlinear regression model
`data`	a data frame in which the expressions in the formula or arguments to the `selfStart` model can be evaluated
`...`	optional additional arguments

Value

A named numeric vector or list of starting estimates for the parameters. The construction of many `selfStart` models is such that these "starting" estimates are, in fact, the converged parameter estimates.

Author(s)

José Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`, `selfStart.default`, `selfStart.formula`. Further, `nlsList` from **nlme**.

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
print(getInitial( rate ~ SSmicmen( conc, Vm, K ), PurTrt ), digits = 3)
```

glm *Fitting Generalized Linear Models*

Description

glm is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

Usage

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart, offset,
    control = list(...), model = TRUE, method = "glm.fit",
    x = FALSE, y = TRUE, singular.ok = TRUE, contrasts = NULL, ...)

glm.fit(x, y, weights = rep.int(1, nobs),
        start = NULL, etastart = NULL, mustart = NULL,
        offset = rep.int(0, nobs), family = gaussian(),
        control = list(), intercept = TRUE, singular.ok = TRUE)

## S3 method for class 'glm'
weights(object, type = c("prior", "working"), ...)
```

Arguments

formula	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
family	a description of the error distribution and link function to be used in the model. For glm this can be a character string naming a family function, a family function or the result of a call to a family function. For glm.fit only the third option is supported. (See family for details of family functions.)
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which glm is called.
weights	an optional vector of 'prior weights' to be used in the fitting process. Should be NULL or a numeric vector.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
start	starting values for the parameters in the linear predictor.

etastart	starting values for the linear predictor.
mustart	starting values for the vector of means.
offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if more than one is specified their sum is used. See model.offset.
control	a list of parameters for controlling the fitting process. For glm.fit this is passed to glm.control.
model	a logical value indicating whether *model frame* should be included as a component of the returned value.
method	the method to be used in fitting the model. The default method "glm.fit" uses iteratively reweighted least squares (IWLS): the alternative "model.frame" returns the model frame and does no fitting. User-supplied fitting functions can be supplied either as a function or a character string naming a function, with a function which takes the same arguments as glm.fit. If specified as a character string it is looked up from within the **stats** namespace.
x, y	For glm: logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value. For glm.fit: x is a design matrix of dimension n * p, and y is a vector of observations of length n.
singular.ok	logical; if FALSE a singular fit is an error.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
intercept	logical. Should an intercept be included in the *null* model?
object	an object inheriting from class "glm".
type	character, partial matching allowed. Type of weights to extract from the fitted model object. Can be abbreviated.
...	For glm: arguments to be used to form the default control argument if it is not supplied directly. For weights: further arguments passed to or from other methods.

Details

A typical predictor has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. For binomial and quasibinomial families the response can also be specified as a factor (when the first level denotes failure and all others success) or as a two-column matrix with the columns giving the numbers of successes and failures. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with any duplicates removed.

A specification of the form first:second indicates the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula.

Non-`NULL` `weights` can be used to indicate that different observations have different dispersions (with the values in `weights` being inversely proportional to the dispersions); or equivalently, when the elements of `weights` are positive integers w_i, that each response y_i is the mean of w_i unit-weight observations. For a binomial GLM prior weights are used to give the number of trials when the response is the proportion of successes: they would rarely be used for a Poisson GLM.

`glm.fit` is the workhorse function: it is not normally called directly but can be more efficient where the response vector, design matrix and family have already been calculated.

If more than one of `etastart`, `start` and `mustart` is specified, the first in the list will be used. It is often advisable to supply starting values for a `quasi` family, and also for families with unusual links such as `gaussian("log")`.

All of `weights`, `subset`, `offset`, `etastart` and `mustart` are evaluated in the same way as variables in `formula`, that is first in `data` and then in the environment of `formula`.

For the background to warning messages about 'fitted probabilities numerically 0 or 1 occurred' for binomial GLMs, see Venables & Ripley (2002, pp. 197–8).

Value

`glm` returns an object of class inheriting from `"glm"` which inherits from the class `"lm"`. See later in this section. If a non-standard `method` is used, the object will also inherit from the class (if any) returned by that function.

The function `summary` (i.e., `summary.glm`) can be used to obtain or print a summary of the results and the function `anova` (i.e., `anova.glm`) to produce an analysis of variance table.

The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` can be used to extract various useful features of the value returned by `glm`.

`weights` extracts a vector of weights, one for each case in the fit (after subsetting and `na.action`).

An object of class `"glm"` is a list containing at least the following components:

`coefficients` a named vector of coefficients

`residuals` the *working* residuals, that is the residuals in the final iteration of the IWLS fit. Since cases with zero weights are omitted, their working residuals are `NA`.

`fitted.values` the fitted mean values, obtained by transforming the linear predictors by the inverse of the link function.

`rank` the numeric rank of the fitted linear model.

`family` the `family` object used.

`linear.predictors` the linear fit on link scale.

`deviance` up to a constant, minus twice the maximized log-likelihood. Where sensible, the constant is chosen so that a saturated model has deviance zero.

`aic` A version of Akaike's *An Information Criterion*, minus twice the maximized log-likelihood plus twice the number of parameters, computed via the `aic` component of the family. For binomial and Poison families the dispersion is fixed

at one and the number of parameters is the number of coefficients. For gaussian, Gamma and inverse gaussian families the dispersion is estimated from the residual deviance, and the number of parameters is the number of coefficients plus one. For a gaussian family the MLE of the dispersion is used so this is a valid value of AIC, but for Gamma and inverse gaussian families it is not. For families fitted by quasi-likelihood the value is NA.

null.deviance	The deviance for the null model, comparable with deviance. The null model will include the offset, and an intercept if there is one in the model. Note that this will be incorrect if the link function depends on the data other than through the fitted mean: specify a zero offset to force a correct calculation.
iter	the number of iterations of IWLS used.
weights	the *working* weights, that is the weights in the final iteration of the IWLS fit.
prior.weights	the weights initially supplied, a vector of 1s if none were.
df.residual	the residual degrees of freedom.
df.null	the residual degrees of freedom for the null model.
y	if requested (the default) the y vector used. (It is a vector even for a binomial model.)
x	if requested, the model matrix.
model	if requested (the default), the model frame.
converged	logical. Was the IWLS algorithm judged to have converged?
boundary	logical. Is the fitted value on the boundary of the attainable values?
call	the matched call.
formula	the formula supplied.
terms	the terms object used.
data	the data argument.
offset	the offset vector used.
control	the value of the control argument used.
method	the name of the fitter function used (when provided as a character string to glm()) or the fitter function (when provided as that).
contrasts	(where relevant) the contrasts used.
xlevels	(where relevant) a record of the levels of the factors used in fitting.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-empty fits will have components qr, R and effects relating to the final weighted linear fit.

Objects of class "glm" are normally of class c("glm", "lm"), that is inherit from class "lm", and well-designed methods for class "lm" will be applied to the weighted linear model at the final iteration of IWLS. However, care is needed, as extractor functions for class "glm" such as residuals and weights do **not** just pick out the component of the fit with the same name.

If a binomial glm model was specified by giving a two-column response, the weights returned by prior.weights are the total numbers of cases (factored by the supplied case weights) and the component y of the result is the proportion of successes.

Fitting functions

The argument `method` serves two purposes. One is to allow the model frame to be recreated with no fitting. The other is to allow the default fitting function `glm.fit` to be replaced by a function which takes the same arguments and uses a different fitting algorithm. If `glm.fit` is supplied as a character string it is used to search for a function of that name, starting in the **stats** namespace.

The class of the object return by the fitter (if any) will be prepended to the class returned by `glm`.

Author(s)

The original R implementation of `glm` was written by Simon Davies working for Ross Ihaka at the University of Auckland, but has since been extensively re-written by members of the R Core team.

The design was inspired by the S function of the same name described in Hastie & Pregibon (1992).

References

Dobson, A. J. (1990) *An Introduction to Generalized Linear Models.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer.

See Also

`anova.glm`, `summary.glm`, etc. for `glm` methods, and the generic functions `anova`, `summary`, `effects`, `fitted.values`, and `residuals`.

`lm` for non-generalized *linear* models (which SAS calls GLMs, for 'general' linear models).

`loglin` and `loglm` (package **MASS**) for fitting log-linear models (which binomial and Poisson GLMs are) to contingency tables.

`bigglm` in package **biglm** for an alternative way to fit GLMs to large datasets (especially those with many cases).

`esoph`, `infert` and `predict.glm` have examples of fitting binomial GLMs.

Examples

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
data.frame(treatment, outcome, counts) # showing data
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
anova(glm.D93)
summary(glm.D93)
## Computing AIC [in many ways]:
(A0 <- AIC(glm.D93))
(ll <- logLik(glm.D93))
A1 <- -2*c(ll) + 2*attr(ll, "df")
A2 <- glm.D93$family$aic(counts, mu=fitted(glm.D93), wt=1) +
```

```
        2 * length(coef(glm.D93))
stopifnot(exprs = {
  all.equal(A0, A1)
  all.equal(A1, A2)
  all.equal(A1, glm.D93$aic)
})


## an example with offsets from Venables & Ripley (2002, p.189)
utils::data(anorexia, package = "MASS")

anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                family = gaussian, data = anorexia)
summary(anorex.1)


# A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
summary(glm(lot1 ~ log(u), data = clotting, family = Gamma))
summary(glm(lot2 ~ log(u), data = clotting, family = Gamma))
## Aliased ("S"ingular) -> 1 NA coefficient
(fS <- glm(lot2 ~ log(u) + log(u^2), data = clotting, family = Gamma))
tools::assertError(update(fS, singular.ok=FALSE), verbose=interactive())
## -> .. "singular fit encountered"

## Not run:
## for an example of the use of a terms object as a formula
demo(glm.vr)

## End(Not run)
```

`glm.control` *Auxiliary for Controlling GLM Fitting*

Description

Auxiliary function for `glm` fitting. Typically only used internally by `glm.fit`, but may be used to construct a `control` argument to either function.

Usage

```
glm.control(epsilon = 1e-8, maxit = 25, trace = FALSE)
```

Arguments

`epsilon`	positive convergence tolerance ϵ; the iterations converge when $\|dev - dev_{old}\|/(\|dev\| + 0.1) < \epsilon$.

maxit	integer giving the maximal number of IWLS iterations.
trace	logical indicating if output should be produced for each iteration.

Details

The control argument of glm is by default passed to the control argument of glm.fit, which uses its elements as arguments to glm.control: the latter provides defaults and sanity checking.

If epsilon is small (less than 10^{-10}) it is also used as the tolerance for the detection of collinearity in the least squares solution.

When trace is true, calls to cat produce the output for each IWLS iteration. Hence, options(digits = *) can be used to increase the precision, see the example.

Value

A list with components named as the arguments.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

glm.fit, the fitting procedure used by glm.

Examples

```
### A variation on  example(glm) :

## Annette Dobson's example ...
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
oo <- options(digits = 12) # to see more when tracing :
glm.D93X <- glm(counts ~ outcome + treatment, family = poisson(),
                trace = TRUE, epsilon = 1e-14)
options(oo)
coef(glm.D93X) # the last two are closer to 0 than in ?glm's  glm.D93
```

---

glm.summaries *Accessing Generalized Linear Model Fits*

Description

These functions are all methods for class glm or summary.glm objects.

Usage

```
## S3 method for class 'glm'
family(object, ...)

## S3 method for class 'glm'
residuals(object, type = c("deviance", "pearson", "working",
                           "response", "partial"), ...)
```

Arguments

object	an object of class glm, typically the result of a call to glm.
type	the type of residuals which should be returned. The alternatives are: "deviance" (default), "pearson", "working", "response", and "partial". Can be abbreviated.
...	further arguments passed to or from other methods.

Details

The references define the types of residuals: Davison & Snell is a good reference for the usages of each.

The partial residuals are a matrix of working residuals, with each column formed by omitting a term from the model.

How residuals treats cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear, with residual value NA. See also naresid.

For fits done with y = FALSE the response values are computed from other components.

References

Davison, A. C. and Snell, E. J. (1991) *Residuals and diagnostics.* In: Statistical Theory and Modelling. In Honour of Sir David Cox, FRS, eds. Hinkley, D. V., Reid, N. and Snell, E. J., Chapman & Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

See Also

glm for computing glm.obj, anova.glm; the corresponding *generic* functions, summary.glm, coef, deviance, df.residual, effects, fitted, residuals.

influence.measures for deletion diagnostics, including standardized (rstandard) and studentized (rstudent) residuals.

`hclust` *Hierarchical Clustering*

Description

Hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it.

Usage

```
hclust(d, method = "complete", members = NULL)

## S3 method for class 'hclust'
plot(x, labels = NULL, hang = 0.1, check = TRUE,
     axes = TRUE, frame.plot = FALSE, ann = TRUE,
     main = "Cluster Dendrogram",
     sub = NULL, xlab = NULL, ylab = "Height", ...)
```

Arguments

`d`	a dissimilarity structure as produced by `dist`.
`method`	the agglomeration method to be used. This should be (an unambiguous abbreviation of) one of `"ward.D"`, `"ward.D2"`, `"single"`, `"complete"`, `"average"` (= UPGMA), `"mcquitty"` (= WPGMA), `"median"` (= WPGMC) or `"centroid"` (= UPGMC).
`members`	`NULL` or a vector with length size of `d`. See the 'Details' section.
`x`	an object of the type produced by `hclust`.
`hang`	The fraction of the plot height by which labels should hang below the rest of the plot. A negative value will cause the labels to hang down from 0.
`check`	logical indicating if the `x` object should be checked for validity. This check is not necessary when `x` is known to be valid such as when it is the direct result of `hclust()`. The default is `check=TRUE`, as invalid inputs may crash R due to memory violation in the internal C plotting code.
`labels`	A character vector of labels for the leaves of the tree. By default the row names or row numbers of the original data are used. If `labels = FALSE` no labels at all are plotted.
`axes, frame.plot, ann`	logical flags as in `plot.default`.
`main, sub, xlab, ylab`	character strings for `title`. `sub` and `xlab` have a non-NULL default when there's a `tree$call`.
`...`	Further graphical arguments. E.g., `cex` controls the size of the labels (if plotted) in the same way as `text`.

Details

This function performs a hierarchical cluster analysis using a set of dissimilarities for the n objects being clustered. Initially, each object is assigned to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. At each stage distances between clusters are recomputed by the Lance–Williams dissimilarity update formula according to the particular clustering method being used.

A number of different clustering methods are provided. *Ward's* minimum variance method aims at finding compact, spherical clusters. The *complete linkage* method finds similar clusters. The *single linkage* method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy. The other methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods. Note however, that methods `"median"` and `"centroid"` are *not* leading to a *monotone distance* measure, or equivalently the resulting dendrograms can have so called *inversions* or *reversals* which are hard to interpret, but note the trichotomies in Legendre and Legendre (2012).

Two different algorithms are found in the literature for Ward clustering. The one used by option `"ward.D"` (equivalent to the only Ward option `"ward"` in R versions $\leq$ 3.0.3) *does not* implement Ward's (1963) clustering criterion, whereas option `"ward.D2"` implements that criterion (Murtagh and Legendre 2014). With the latter, the dissimilarities are *squared* before cluster updating. Note that `agnes(*, method="ward")` corresponds to `hclust(*, "ward.D2")`.

If `members != NULL`, then `d` is taken to be a dissimilarity matrix between clusters instead of dissimilarities between singletons and `members` gives the number of observations per cluster. This way the hierarchical cluster algorithm can be 'started in the middle of the dendrogram', e.g., in order to reconstruct the part of the tree above a cut (see examples). Dissimilarities between clusters can be efficiently computed (i.e., without `hclust` itself) only for a limited number of distance/linkage combinations, the simplest one being *squared* Euclidean distance and centroid linkage. In this case the dissimilarities between the clusters are the squared Euclidean distances between cluster means.

In hierarchical cluster displays, a decision is needed at each merge to specify which subtree should go on the left and which on the right. Since, for n observations there are $n-1$ merges, there are $2^{(n-1)}$ possible orderings for the leaves in a cluster tree, or dendrogram. The algorithm used in `hclust` is to order the subtree so that the tighter cluster is on the left (the last, i.e., most recent, merge of the left subtree is at a lower value than the last merge of the right subtree). Single observations are the tightest clusters possible, and merges involving two observations place them in order by their observation sequence number.

Value

An object of class `"hclust"` which describes the tree produced by the clustering process. The object is a list with components:

`merge` an $n-1$ by 2 matrix. Row i of `merge` describes the merging of clusters at step i of the clustering. If an element j in the row is negative, then observation $-j$ was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in `merge` indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons.

`height` a set of $n-1$ real values (non-decreasing for ultrametric trees). The clustering *height*: that is, the value of the criterion associated with the clustering `method`

	for the particular agglomeration.
`order`	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix `merge` will not have crossings of the branches.
`labels`	labels for each of the objects being clustered.
`call`	the call which produced the result.
`method`	the cluster method that has been used.
`dist.method`	the distance that has been used to create `d` (only returned if the distance object has a `"method"` attribute).

There are `print`, `plot` and `identify` (see `identify.hclust`) methods and the `rect.hclust()` function for `hclust` objects.

Note

Method `"centroid"` is typically meant to be used with *squared* Euclidean distances.

Author(s)

The `hclust` function is based on Fortran code contributed to STATLIB by F. Murtagh.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole. (S version.)

Everitt, B. (1974). *Cluster Analysis*. London: Heinemann Educ. Books.

Hartigan, J.A. (1975). *Clustering Algorithms*. New York: Wiley.

Sneath, P. H. A. and R. R. Sokal (1973). *Numerical Taxonomy*. San Francisco: Freeman.

Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Academic Press: New York.

Gordon, A. D. (1999). *Classification*. Second Edition. London: Chapman and Hall / CRC

Murtagh, F. (1985). "Multidimensional Clustering Algorithms", in *COMPSTAT Lectures 4*. Wuerzburg: Physica-Verlag (for algorithmic details of algorithms used).

McQuitty, L.L. (1966). Similarity Analysis by Reciprocal Pairs for Discrete and Continuous Data. *Educational and Psychological Measurement*, **26**, 825–831. doi:10.1177/001316446602600402.

Legendre, P. and L. Legendre (2012). *Numerical Ecology*, 3rd English ed. Amsterdam: Elsevier Science BV.

Murtagh, Fionn and Legendre, Pierre (2014). Ward's hierarchical agglomerative clustering method: which algorithms implement Ward's criterion? *Journal of Classification*, **31**, 274–295. doi:10.1007/s003570149161z.

See Also

`identify.hclust`, `rect.hclust`, `cutree`, `dendrogram`, `kmeans`.

For the Lance–Williams formula and methods that apply it generally, see `agnes` from package **cluster**.

Examples

```
require(graphics)

### Example 1: Violent crime rates by US state

hc <- hclust(dist(USArrests), "ave")
plot(hc)
plot(hc, hang = -1)

## Do the same with centroid clustering and *squared* Euclidean distance,
## cut the tree into ten clusters and reconstruct the upper part of the
## tree from the cluster centers.
hc <- hclust(dist(USArrests)^2, "cen")
memb <- cutree(hc, k = 10)
cent <- NULL
for(k in 1:10){
  cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
}
hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
opar <- par(mfrow = c(1, 2))
plot(hc,  labels = FALSE, hang = -1, main = "Original Tree")
plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
par(opar)

### Example 2: Straight-line distances among 10 US cities
##  Compare the results of algorithms "ward.D" and "ward.D2"

mds2 <- -cmdscale(UScitiesD)
plot(mds2, type="n", axes=FALSE, ann=FALSE)
text(mds2, labels=rownames(mds2), xpd = NA)

hcity.D  <- hclust(UScitiesD, "ward.D") # "wrong"
hcity.D2 <- hclust(UScitiesD, "ward.D2")
opar <- par(mfrow = c(1, 2))
plot(hcity.D,  hang=-1)
plot(hcity.D2, hang=-1)
par(opar)
```

---

heatmap *Draw a Heat Map*

Description

A heat map is a false color image (basically `image(t(x))`) with a dendrogram added to the left side and to the top. Typically, reordering of the rows and columns according to some set of values (row or column means) within the restrictions imposed by the dendrogram is carried out.

Usage

```
heatmap(x, Rowv = NULL, Colv = if(symm)"Rowv" else NULL,
        distfun = dist, hclustfun = hclust,
        reorderfun = function(d, w) reorder(d, w),
        add.expr, symm = FALSE, revC = identical(Colv, "Rowv"),
        scale = c("row", "column", "none"), na.rm = TRUE,
        margins = c(5, 5), ColSideColors, RowSideColors,
        cexRow = 0.2 + 1/log10(nr), cexCol = 0.2 + 1/log10(nc),
        labRow = NULL, labCol = NULL, main = NULL,
        xlab = NULL, ylab = NULL,
        keep.dendro = FALSE, verbose = getOption("verbose"), ...)
```

Arguments

x	numeric matrix of the values to be plotted.
Rowv	determines if and how the *row* dendrogram should be computed and reordered. Either a dendrogram or a vector of values used to reorder the row dendrogram or NA to suppress any row dendrogram (and reordering) or by default, NULL, see 'Details' below.
Colv	determines if and how the *column* dendrogram should be reordered. Has the same options as the Rowv argument above and *additionally* when x is a square matrix, Colv = "Rowv" means that columns should be treated identically to the rows (and so if there is to be no row dendrogram there will not be a column one either).
distfun	function used to compute the distance (dissimilarity) between both rows and columns. Defaults to dist.
hclustfun	function used to compute the hierarchical clustering when Rowv or Colv are not dendrograms. Defaults to hclust. Should take as argument a result of distfun and return an object to which as.dendrogram can be applied.
reorderfun	function(d, w) of dendrogram and weights for reordering the row and column dendrograms. The default uses reorder.dendrogram.
add.expr	expression that will be evaluated after the call to image. Can be used to add components to the plot.
symm	logical indicating if x should be treated **symm**etrically; can only be true when x is a square matrix.
revC	logical indicating if the column order should be reversed for plotting, such that e.g., for the symmetric case, the symmetry axis is as usual.
scale	character indicating if the values should be centered and scaled in either the row direction or the column direction, or none. The default is "row" if symm false, and "none" otherwise.
na.rm	logical indicating whether NA's should be removed.
margins	numeric vector of length 2 containing the margins (see par(mar = *)) for column and row names, respectively.
ColSideColors	(optional) character vector of length ncol(x) containing the color names for a horizontal side bar that may be used to annotate the columns of x.

RowSideColors	(optional) character vector of length nrow(x) containing the color names for a vertical side bar that may be used to annotate the rows of x.
cexRow, cexCol	positive numbers, used as cex.axis in for the row or column axis labeling. The defaults currently only use number of rows or columns, respectively.
labRow, labCol	character vectors with row and column labels to use; these default to rownames(x) or colnames(x), respectively.
main, xlab, ylab	main, x- and y-axis titles; defaults to none.
keep.dendro	logical indicating if the dendrogram(s) should be kept as part of the result (when Rowv and/or Colv are not NA).
verbose	logical indicating if information should be printed.
...	additional arguments passed on to image, e.g., col specifying the colors.

Details

If either Rowv or Colv are dendrograms they are honored (and not reordered). Otherwise, dendrograms are computed as dd <- as.dendrogram(hclustfun(distfun(X))) where X is either x or t(x).

If either is a vector (of 'weights') then the appropriate dendrogram is reordered according to the supplied values subject to the constraints imposed by the dendrogram, by reorder(dd, Rowv), in the row case. If either is missing, as by default, then the ordering of the corresponding dendrogram is by the mean value of the rows/columns, i.e., in the case of rows, Rowv <- rowMeans(x, na.rm = na.rm). If either is NA, *no reordering* will be done for the corresponding side.

By default (scale = "row") the rows are scaled to have mean zero and standard deviation one. There is some empirical evidence from genomic plotting that this is useful.

Value

Invisibly, a list with components

rowInd	**r**ow index permutation vector as returned by order.dendrogram.
colInd	**c**olumn index permutation vector.
Rowv	the row dendrogram; only if input Rowv was not NA and keep.dendro is true.
Colv	the column dendrogram; only if input Colv was not NA and keep.dendro is true.

Note

Unless Rowv = NA (or Colw = NA), the original rows and columns are reordered *in any case* to match the dendrogram, e.g., the rows by order.dendrogram(Rowv) where Rowv is the (possibly reorder()ed) row dendrogram.

heatmap() uses layout and draws the image in the lower right corner of a 2x2 layout. Consequentially, it can **not** be used in a multi column/row layout, i.e., when par(mfrow = *) or (mfcol = *) has been called.

Author(s)

Andy Liaw, original; R. Gentleman, M. Maechler, W. Huber, revisions.

See Also

image, hclust

Examples

```
require(graphics); require(grDevices)
x  <- as.matrix(mtcars)
rc <- rainbow(nrow(x), start = 0, end = .3)
cc <- rainbow(ncol(x), start = 0, end = .3)
hv <- heatmap(x, col = cm.colors(256), scale = "column",
              RowSideColors = rc, ColSideColors = cc, margins = c(5,10),
              xlab = "specification variables", ylab =  "Car Models",
              main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
utils::str(hv) # the two re-ordering index vectors

## no column dendrogram (nor reordering) at all:
heatmap(x, Colv = NA, col = cm.colors(256), scale = "column",
        RowSideColors = rc, margins = c(5,10),
        xlab = "specification variables", ylab =  "Car Models",
        main = "heatmap(<Mtcars data>, ..., scale = \"column\")")

## "no nothing"
heatmap(x, Rowv = NA, Colv = NA, scale = "column",
        main = "heatmap(*, NA, NA) ~= image(t(x))")

round(Ca <- cor(attitude), 2)
symnum(Ca) # simple graphic
heatmap(Ca,               symm = TRUE, margins = c(6,6)) # with reorder()
heatmap(Ca, Rowv = FALSE, symm = TRUE, margins = c(6,6)) # _NO_ reorder()
## slightly artificial with color bar, without and with ordering:
cc <- rainbow(nrow(Ca))
heatmap(Ca, Rowv = FALSE, symm = TRUE, RowSideColors = cc, ColSideColors = cc,
 margins = c(6,6))
heatmap(Ca,  symm = TRUE, RowSideColors = cc, ColSideColors = cc,
 margins = c(6,6))

## For variable clustering, rather use distance based on cor():
symnum( cU <- cor(USJudgeRatings) )

hU <- heatmap(cU, Rowv = FALSE, symm = TRUE, col = topo.colors(16),
              distfun = function(c) as.dist(1 - c), keep.dendro = TRUE)
## The Correlation matrix with same reordering:
round(100 * cU[hU[[1]], hU[[2]]])
## The column dendrogram:
utils::str(hU$Colv)
```

HoltWinters *Holt-Winters Filtering*

Description

Computes Holt-Winters Filtering of a given time series. Unknown parameters are determined by minimizing the squared prediction error.

Usage

```
HoltWinters(x, alpha = NULL, beta = NULL, gamma = NULL,
            seasonal = c("additive", "multiplicative"),
            start.periods = 2, l.start = NULL, b.start = NULL,
            s.start = NULL,
            optim.start = c(alpha = 0.3, beta = 0.1, gamma = 0.1),
            optim.control = list())
```

Arguments

x	An object of class ts
alpha	$alpha$ parameter of Holt-Winters Filter.
beta	$beta$ parameter of Holt-Winters Filter. If set to FALSE, the function will do exponential smoothing.
gamma	$gamma$ parameter used for the seasonal component. If set to FALSE, an non-seasonal model is fitted.
seasonal	Character string to select an "additive" (the default) or "multiplicative" seasonal model. The first few characters are sufficient. (Only takes effect if gamma is non-zero).
start.periods	Start periods used in the autodetection of start values. Must be at least 2.
l.start	Start value for level (a[0]).
b.start	Start value for trend (b[0]).
s.start	Vector of start values for the seasonal component ($s_1[0] \ldots s_p[0]$)
optim.start	Vector with named components alpha, beta, and gamma containing the starting values for the optimizer. Only the values needed must be specified. Ignored in the one-parameter case.
optim.control	Optional list with additional control parameters passed to optim if this is used. Ignored in the one-parameter case.

Details

The additive Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = a[t] + hb[t] + s[t-p+1+(h-1) \bmod p],$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t] - s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t] - a[t]) + (1-\gamma)s[t-p]$$

The multiplicative Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = (a[t] + hb[t]) \times s[t-p+1+(h-1) \bmod p].$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t]/s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t]/a[t]) + (1-\gamma)s[t-p]$$

The data in `x` are required to be non-zero for a multiplicative model, but it makes most sense if they are all positive.

The function tries to find the optimal values of α and/or β and/or γ by minimizing the squared one-step prediction error if they are `NULL` (the default). `optimize` will be used for the single-parameter case, and `optim` otherwise.

For seasonal models, start values for `a`, `b` and `s` are inferred by performing a simple decomposition in trend and seasonal component using moving averages (see function `decompose`) on the `start.periods` first periods (a simple linear regression on the trend component is used for starting level and trend). For level/trend-models (no seasonal component), start values for `a` and `b` are `x[2]` and `x[2] - x[1]`, respectively. For level-only models (ordinary exponential smoothing), the start value for `a` is `x[1]`.

Value

An object of class `"HoltWinters"`, a list with components:

`fitted`	A multiple time series with one column for the filtered series as well as for the level, trend and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
`x`	The original series
`alpha`	alpha used for filtering
`beta`	beta used for filtering
`gamma`	gamma used for filtering
`coefficients`	A vector with named components `a, b, s1, ..., sp` containing the estimated values for the level, trend and seasonal components
`seasonal`	The specified `seasonal` parameter
`SSE`	The final sum of squared errors achieved in optimizing
`call`	The call used

Author(s)

David Meyer <David.Meyer@wu.ac.at>

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, *ONR Research Memorandum, Carnegie Institute of Technology* **52**. (reprint at doi:10.1016/j.ijforecast.2003.09.015).

P. R. Winters (1960). Forecasting sales by exponentially weighted moving averages. *Management Science*, **6**, 324–342. doi:10.1287/mnsc.6.3.324.

See Also

`predict.HoltWinters`, `optim`.

Examples

```
require(graphics)

## Seasonal Holt-Winters
(m <- HoltWinters(co2))
plot(m)
plot(fitted(m))

(m <- HoltWinters(AirPassengers, seasonal = "mult"))
plot(m)

## Non-Seasonal Holt-Winters
x <- uspop + rnorm(uspop, sd = 5)
m <- HoltWinters(x, gamma = FALSE)
plot(m)

## Exponential Smoothing
m2 <- HoltWinters(x, gamma = FALSE, beta = FALSE)
lines(fitted(m2)[,1], col = 3)
```

Hypergeometric *The Hypergeometric Distribution*

Description

Density, distribution function, quantile function and random generation for the hypergeometric distribution.

Usage

```
dhyper(x, m, n, k, log = FALSE)
phyper(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
qhyper(p, m, n, k, lower.tail = TRUE, log.p = FALSE)
rhyper(nn, m, n, k)
```

Arguments

`x`, `q`	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
`p`	probability, it must be between 0 and 1.
`nn`	number of observations. If `length(nn) > 1`, the length is taken to be the number required.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The hypergeometric distribution is used for sampling *without* replacement. The density of this distribution with parameters `m`, `n` and `k` (named Np, $N - Np$, and n, respectively in the reference below, where $N := m + n$ is also used in other references) is given by

$$p(x) = \binom{m}{x}\binom{n}{k-x} \Big/ \binom{m+n}{k}$$

for $x = 0, \ldots, k$.

Note that $p(x)$ is non-zero only for $\max(0, k - n) \leq x \leq \min(k, m)$.

With $p := m/(m + n)$ (hence $Np = N \times p$ in the reference's notation), the first two moments are mean

$$E[X] = \mu = kp$$

and variance

$$\mathrm{Var}(X) = kp(1 - p)\frac{m + n - k}{m + n - 1},$$

which shows the closeness to the Binomial(k, p) (where the hypergeometric has smaller variance unless $k = 1$).

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

In `rhyper()`, if one of m, n, k exceeds `.Machine$integer.max`, currently the equivalent of `qhyper(runif(nn), m,n,k)` is used which is comparably slow while instead a binomial approximation may be considerably more efficient.

Value

`dhyper` gives the density, `phyper` gives the distribution function, `qhyper` gives the quantile function, and `rhyper` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rhyper`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Source

dhyper computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

phyper is based on calculating dhyper and phyper(...)/dhyper(...) (as a summation), based on ideas of Ian Smith and Morten Welinder.

qhyper is based on inversion (of an earlier phyper() algorithm).

rhyper is based on a corrected version of

Kachitvichyanukul, V. and Schmeiser, B. (1985). Computer generation of hypergeometric random variates. *Journal of Statistical Computation and Simulation*, **22**, 127–145.

References

Johnson, N. L., Kotz, S., and Kemp, A. W. (1992) *Univariate Discrete Distributions*, Second Edition. New York: Wiley.

See Also

Distributions for other standard distributions.

Examples

```
m <- 10; n <- 7; k <- 8
x <- 0:(k+1)
rbind(phyper(x, m, n, k), dhyper(x, m, n, k))
all(phyper(x, m, n, k) == cumsum(dhyper(x, m, n, k)))  # FALSE
## but errors are very small:
signif(phyper(x, m, n, k) - cumsum(dhyper(x, m, n, k)), digits = 3)

stopifnot(abs(phyper(x, m, n, k) - cumsum(dhyper(x, m, n, k))) < 5e-16)
```

---

identify.hclust *Identify Clusters in a Dendrogram*

Description

identify.hclust reads the position of the graphics pointer when the (first) mouse button is pressed. It then cuts the tree at the vertical position of the pointer and highlights the cluster containing the horizontal position of the pointer. Optionally a function is applied to the index of data points contained in the cluster.

Usage

```
## S3 method for class 'hclust'
identify(x, FUN = NULL, N = 20, MAXCLUSTER = 20, DEV.FUN = NULL,
         ...)
```

Arguments

x	an object of the type produced by hclust.
FUN	(optional) function to be applied to the index numbers of the data points in a cluster (see 'Details' below).
N	the maximum number of clusters to be identified.
MAXCLUSTER	the maximum number of clusters that can be produced by a cut (limits the effective vertical range of the pointer).
DEV.FUN	(optional) integer scalar. If specified, the corresponding graphics device is made active before FUN is applied.
...	further arguments to FUN.

Details

By default clusters can be identified using the mouse and an invisible list of indices of the respective data points is returned.

If FUN is not NULL, then the index vector of data points is passed to this function as first argument, see the examples below. The active graphics device for FUN can be specified using DEV.FUN.

The identification process is terminated by pressing any mouse button other than the first, see also identify.

Value

Either a list of data point index vectors or a list of return values of FUN.

See Also

hclust, rect.hclust

Examples

```
## Not run:
require(graphics)

hca <- hclust(dist(USArrests))
plot(hca)
(x <- identify(hca)) ##  Terminate with 2nd mouse button !!

hci <- hclust(dist(iris[,1:4]))
plot(hci)
identify(hci, function(k) print(table(iris[k,5])))

# open a new device (one for dendrogram, one for bars):
dev.new() # << make that narrow (& small)
          # and *beside* 1st one
nD <- dev.cur()            # to be for the barplot
dev.set(dev.prev())  # old one for dendrogram
plot(hci)
## select subtrees in dendrogram and "see" the species distribution:
```

```
identify(hci, function(k) barplot(table(iris[k,5]), col = 2:4), DEV.FUN = nD)

## End(Not run)
```

`influence.measures` *Regression Deletion Diagnostics*

Description

This suite of functions can be used to compute some of the regression (leave-one-out deletion) diagnostics for linear and generalized linear models discussed in Belsley, Kuh and Welsch (1980), Cook and Weisberg (1982), etc.

Usage

```
influence.measures(model, infl = influence(model))

rstandard(model, ...)
## S3 method for class 'lm'
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          sd = sqrt(deviance(model)/df.residual(model)),
          type = c("sd.1", "predictive"), ...)
## S3 method for class 'glm'
rstandard(model, infl = influence(model, do.coef = FALSE),
          type = c("deviance", "pearson"), ...)

rstudent(model, ...)
## S3 method for class 'lm'
rstudent(model, infl = lm.influence(model, do.coef = FALSE),
         res = infl$wt.res, ...)
## S3 method for class 'glm'
rstudent(model, infl = influence(model, do.coef = FALSE), ...)

dffits(model, infl = , res = )

dfbeta(model, ...)
## S3 method for class 'lm'
dfbeta(model, infl = lm.influence(model, do.coef = TRUE), ...)

dfbetas(model, ...)
## S3 method for class 'lm'
dfbetas(model, infl = lm.influence(model, do.coef = TRUE), ...)

covratio(model, infl = lm.influence(model, do.coef = FALSE),
         res = weighted.residuals(model))

cooks.distance(model, ...)
```

```
## S3 method for class 'lm'
cooks.distance(model, infl = lm.influence(model, do.coef = FALSE),
               res = weighted.residuals(model),
               sd = sqrt(deviance(model)/df.residual(model)),
               hat = infl$hat, ...)
## S3 method for class 'glm'
cooks.distance(model, infl = influence(model, do.coef = FALSE),
               res = infl$pear.res,
               dispersion = summary(model)$dispersion,
               hat = infl$hat, ...)

hatvalues(model, ...)
## S3 method for class 'lm'
hatvalues(model, infl = lm.influence(model, do.coef = FALSE), ...)

hat(x, intercept = TRUE)
```

Arguments

model	an R object, typically returned by lm or glm.
infl	influence structure as returned by lm.influence or influence (the latter only for the glm method of rstudent and cooks.distance).
res	(possibly weighted) residuals, with proper default.
sd	standard deviation to use, see default.
dispersion	dispersion (for glm objects) to use, see default.
hat	hat values H_{ii}, see default.
type	type of residuals for rstandard, with different options and meanings for lm and glm. Can be abbreviated.
x	the X or design matrix.
intercept	should an intercept column be prepended to x?
...	further arguments passed to or from other methods.

Details

The primary high-level function is influence.measures which produces a class "infl" object tabular display showing the DFBETAs for each model variable, DFFITs, covariance ratios, Cook's distances and the diagonal elements of the hat matrix. Cases which are influential with respect to any of these measures are marked with an asterisk.

The functions dfbetas, dffits, covratio and cooks.distance provide direct access to the corresponding diagnostic quantities. Functions rstandard and rstudent give the standardized and Studentized residuals respectively. (These re-normalize the residuals to have unit variance, using an overall and leave-one-out measure of the error variance respectively.)

Note that for *multivariate* lm() models (of class "mlm"), these functions return 3d arrays instead of matrices, or matrices instead of vectors.

Values for generalized linear models are approximations, as described in Williams (1987) (except that Cook's distances are scaled as F rather than as chi-square values). The approximations can be poor when some cases have large influence.

The optional `infl`, `res` and `sd` arguments are there to encourage the use of these direct access functions, in situations where, e.g., the underlying basic influence measures (from `lm.influence` or the generic `influence`) are already available.

Note that cases with `weights == 0` are *dropped* from all these functions, but that if a linear model has been fitted with `na.action = na.exclude`, suitable values are filled in for the cases excluded during fitting.

For linear models, `rstandard(*, type = "predictive")` provides leave-one-out cross validation residuals, and the "PRESS" statistic (**PRE**dictive **S**um of **S**quares, the same as the CV score) of model `model` is

```
PRESS <- sum(rstandard(model, type="pred")^2)
```

The function `hat()` exists mainly for S (version 2) compatibility; we recommend using `hatvalues()` instead.

Note

For `hatvalues`, `dfbeta`, and `dfbetas`, the method for linear models also works for generalized linear models.

Author(s)

Several R core team members and John Fox, originally in his '`car`' package.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980). *Regression Diagnostics*. New York: Wiley.

Cook, R. D. and Weisberg, S. (1982). *Residuals and Influence in Regression*. London: Chapman and Hall.

Williams, D. A. (1987). Generalized linear model diagnostics using the deviance and single case deletions. *Applied Statistics*, **36**, 181–191. doi:10.2307/2347550.

Fox, J. (1997). *Applied Regression, Linear Models, and Related Methods*. Sage.

Fox, J. (2002) *An R and S-Plus Companion to Applied Regression*. Sage Publ.

Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*, second edition. Sage Publ; https://socialsciences.mcmaster.ca/jfox/Books/Companion/.

See Also

`influence` (containing `lm.influence`).

'plotmath' for the use of `hat` in plot annotation.

Examples

```
require(graphics)

## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

inflm.SR <- influence.measures(lm.SR)
which(apply(inflm.SR$is.inf, 1, any))
# which observations 'are' influential
summary(inflm.SR) # only these
inflm.SR          # all
plot(rstudent(lm.SR) ~ hatvalues(lm.SR)) # recommended by some
plot(lm.SR, which = 5) # an enhanced version of that via plot(<lm>)

## The 'infl' argument is not needed, but avoids recomputation:
rs <- rstandard(lm.SR)
iflSR <- influence(lm.SR)
all.equal(rs, rstandard(lm.SR, infl = iflSR), tolerance = 1e-10)
## to "see" the larger values:
1000 * round(dfbetas(lm.SR, infl = iflSR), 3)
cat("PRESS :"); (PRESS <- sum( rstandard(lm.SR, type = "predictive")^2 ))
stopifnot(all.equal(PRESS, sum( (residuals(lm.SR) / (1 - iflSR$hat))^2)))

## Show that "PRE-residuals"  ==  L.O.O. Crossvalidation (CV) errors:
X <- model.matrix(lm.SR)
y <- model.response(model.frame(lm.SR))
## Leave-one-out CV least-squares prediction errors (relatively fast)
rCV <- vapply(seq_len(nrow(X)), function(i)
              y[i] - X[i,] %*% .lm.fit(X[-i,], y[-i])$coefficients,
              numeric(1))
## are the same as the *faster* rstandard(*, "pred") :
stopifnot(all.equal(rCV, unname(rstandard(lm.SR, type = "predictive"))))


## Huber's data [Atkinson 1985]
xh <- c(-4:0, 10)
yh <- c(2.48, .73, -.04, -1.44, -1.32, 0)
lmH <- lm(yh ~ xh)
summary(lmH)
im <- influence.measures(lmH)
 im
is.inf <- apply(im$is.inf, 1, any)
plot(xh,yh, main = "Huber's data: L.S. line and influential obs.")
abline(lmH); points(xh[is.inf], yh[is.inf], pch = 20, col = 2)

## Irwin's data [Williams 1987]
xi <- 1:5
yi <- c(0,2,14,19,30)    # number of mice responding to dose xi
mi <- rep(40, 5)         # number of mice exposed
glmI <- glm(cbind(yi, mi -yi) ~ xi, family = binomial)
summary(glmI)
```

```
signif(cooks.distance(glmI), 3)   # ~= Ci in Table 3, p.184
imI <- influence.measures(glmI)
 imI
stopifnot(all.equal(imI$infmat[,"cook.d"],
          cooks.distance(glmI)))
```

`integrate` *Integration of One-Dimensional Functions*

Description

Adaptive quadrature of functions of one variable over a finite or infinite interval.

Usage

```
integrate(f, lower, upper, ..., subdivisions = 100L,
          rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
          stop.on.error = TRUE, keep.xy = FALSE, aux = NULL)
```

Arguments

`f`	an R function taking a numeric first argument and returning a numeric vector of the same length. Returning a non-finite element will generate an error.
`lower`, `upper`	the limits of integration. Can be infinite.
`...`	additional arguments to be passed to `f`.
`subdivisions`	the maximum number of subintervals.
`rel.tol`	relative accuracy requested.
`abs.tol`	absolute accuracy requested.
`stop.on.error`	logical. If true (the default) an error stops the function. If false some errors will give a result with a warning in the `message` component.
`keep.xy`	unused. For compatibility with S.
`aux`	unused. For compatibility with S.

Details

Note that arguments after `...` must be matched exactly.

If one or both limits are infinite, the infinite range is mapped onto a finite interval.

For a finite interval, globally adaptive interval subdivision is used in connection with extrapolation by Wynn's Epsilon algorithm, with the basic step being Gauss–Kronrod quadrature.

`rel.tol` cannot be less than `max(50*.Machine$double.eps, 0.5e-28)` if `abs.tol <= 0`.

Note that the comments in the C source code in '*R*`/src/appl/integrate.c`' give more details, particularly about reasons for failure (internal error code `ier >= 1`).

In R versions $\leq$ 3.2.x, the first entries of `lower` and `upper` were used whereas an error is signalled now if they are not of length one.

Value

A list of class "integrate" with components

value	the final estimate of the integral.
abs.error	estimate of the modulus of the absolute error.
subdivisions	the number of subintervals produced in the subdivision process.
message	"OK" or a character string giving the error message.
call	the matched call.

Note

Like all numerical integration routines, these evaluate the function on a finite set of points. If the function is approximately constant (in particular, zero) over nearly all its range it is possible that the result and error estimate may be seriously wrong.

When integrating over infinite intervals do so explicitly, rather than just using a large number as the endpoint. This increases the chance of a correct answer – any function whose integral over an infinite interval is finite must be near zero for most of that interval.

For values at a finite set of points to be a fair reflection of the behaviour of the function elsewhere, the function needs to be well-behaved, for example differentiable except perhaps for a small number of jumps or integrable singularities.

f must accept a vector of inputs and produce a vector of function evaluations at those points. The Vectorize function may be helpful to convert f to this form.

Source

Based on QUADPACK routines dqags and dqagi by R. Piessens and E. deDoncker–Kapenga, available from Netlib.

References

R. Piessens, E. deDoncker–Kapenga, C. Uberhuber, D. Kahaner (1983) *Quadpack: a Subroutine Package for Automatic Integration*; Springer Verlag.

Examples

```
integrate(dnorm, -1.96, 1.96)
integrate(dnorm, -Inf, Inf)

## a slowly-convergent integral
integrand <- function(x) {1/((x+1)*sqrt(x))}
integrate(integrand, lower = 0, upper = Inf)

## don't do this if you really want the integral from 0 to Inf
integrate(integrand, lower = 0, upper = 10)
integrate(integrand, lower = 0, upper = 100000)
integrate(integrand, lower = 0, upper = 1000000, stop.on.error = FALSE)

## some functions do not handle vector input properly
```

```
f <- function(x) 2.0
try(integrate(f, 0, 1))
integrate(Vectorize(f), 0, 1)  ## correct
integrate(function(x) rep(2.0, length(x)), 0, 1)  ## correct

## integrate can fail if misused
integrate(dnorm, 0, 2)
integrate(dnorm, 0, 20)
integrate(dnorm, 0, 200)
integrate(dnorm, 0, 2000)
integrate(dnorm, 0, 20000) ## fails on many systems
integrate(dnorm, 0, Inf)   ## works

integrate(dnorm, 0:1, 20) #-> error!
## "silently" gave  integrate(dnorm, 0, 20)  in earlier versions of R
```

`interaction.plot` *Two-way Interaction Plot*

Description

Plots the mean (or other summary) of the response for two-way combinations of factors, thereby illustrating possible interactions.

Usage

```
interaction.plot(x.factor, trace.factor, response, fun = mean,
                 type = c("l", "p", "b", "o", "c"), legend = TRUE,
                 trace.label = deparse1(substitute(trace.factor)),
                 fixed = FALSE,
                 xlab = deparse1(substitute(x.factor)),
                 ylab = ylabel,
                 ylim = range(cells, na.rm = TRUE),
                 lty = nc:1, col = 1, pch = c(1:9, 0, letters),
                 xpd = NULL, leg.bg = par("bg"), leg.bty = "n",
                 xtick = FALSE, xaxt = par("xaxt"), axes = TRUE,
                 ...)
```

Arguments

`x.factor`	a factor whose levels will form the x axis.
`trace.factor`	another factor whose levels will form the traces.
`response`	a numeric variable giving the response.
`fun`	the function to compute the summary. Should return a single real value.
`type`	the type of plot (see `plot.default`): lines or points or both.
`legend`	logical. Should a legend be included?

trace.label	overall label for the legend.
fixed	logical. Should the legend be in the order of the levels of trace.factor (TRUE) or in the order of the traces at their right-hand ends (FALSE, the default)?
xlab, ylab	the x and y label of the plot each with a sensible default.
ylim	numeric of length 2 giving the y limits for the plot.
lty	line type for the lines drawn, with sensible default.
col	the color to be used for plotting.
pch	a vector of plotting symbols or characters, with sensible default.
xpd	determines clipping behaviour for the legend used, see par(xpd). Per default, the legend is *not* clipped at the figure border.
leg.bg, leg.bty	arguments passed to legend().
xtick	logical. Should tick marks be used on the x axis?
xaxt, axes, ...	graphics parameters to be passed to the plotting routines.

Details

By default the levels of x.factor are plotted on the x axis in their given order, with extra space on the right for the legend (if specified). If x.factor is an ordered factor and the levels are numeric, these numeric values are used for the x axis.

The response and hence its summary can contain missing values. If so, the missing values and the line segments joining them are omitted from the plot (and this can be somewhat disconcerting).

The graphics parameters xlab, ylab, ylim, lty, col and pch are given suitable defaults (and xlim and xaxs are set and cannot be overridden). The defaults are to cycle through the line types, use the foreground colour, and to use the symbols 1:9, 0, and the small letters to plot the traces.

Note

Some of the argument names and the precise behaviour are chosen for S-compatibility.

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
require(graphics)

with(ToothGrowth, {
interaction.plot(dose, supp, len, fixed = TRUE)
dose <- ordered(dose)
interaction.plot(dose, supp, len, fixed = TRUE,
                 col = 2:3, leg.bty = "o", xtick = TRUE)
interaction.plot(dose, supp, len, fixed = TRUE, col = 2:3, type = "p")
})
```

```
with(OrchardSprays, {
  interaction.plot(treatment, rowpos, decrease)
  interaction.plot(rowpos, treatment, decrease, cex.axis = 0.8)
  ## order the rows by their mean effect
  rowpos <- factor(rowpos,
                   levels = sort.list(tapply(decrease, rowpos, mean)))
  interaction.plot(rowpos, treatment, decrease, col = 2:9, lty = 1)
})
```

IQR *The Interquartile Range*

Description

computes interquartile range of the x values.

Usage

```
IQR(x, na.rm = FALSE, type = 7)
```

Arguments

x	a numeric vector.
na.rm	logical. Should missing values be removed?
type	an integer selecting one of the many quantile algorithms, see quantile.

Details

Note that this function computes the quartiles using the quantile function rather than following Tukey's recommendations, i.e., IQR(x) = quantile(x, 3/4) - quantile(x, 1/4).

For normally $N(m, 1)$ distributed X, the expected value of IQR(X) is 2*qnorm(3/4) = 1.3490, i.e., for a normal-consistent estimate of the standard deviation, use IQR(x) / 1.349.

References

Tukey, J. W. (1977). *Exploratory Data Analysis.* Reading: Addison-Wesley.

See Also

fivenum, mad which is more robust, range, quantile.

Examples

```
IQR(rivers)
```

is.empty.model *Test if a Model's Formula is Empty*

Description

R's formula notation allows models with no intercept and no predictors. These require special handling internally. `is.empty.model()` checks whether an object describes an empty model.

Usage

```
is.empty.model(x)
```

Arguments

x	A `terms` object or an object with a `terms` method.

Value

`TRUE` if the model is empty

See Also

`lm`, `glm`

Examples

```
y <- rnorm(20)
is.empty.model(y ~ 0)
is.empty.model(y ~ -1)
is.empty.model(lm(y ~ 0))
```

isoreg *Isotonic / Monotone Regression*

Description

Compute the isotonic (monotonically increasing nonparametric) least squares regression which is piecewise constant.

Usage

```
isoreg(x, y = NULL)
```

Arguments

x, y	coordinate vectors of the regression points. Alternatively a single plotting structure can be specified: see `xy.coords`. The y values, and even `sum(y)` must be finite, currently.

Details

The algorithm determines the convex minorant $m(x)$ of the *cumulative* data (i.e., `cumsum(y)`) which is piecewise linear and the result is $m'(x)$, a step function with level changes at locations where the convex $m(x)$ touches the cumulative data polygon and changes slope.
`as.stepfun()` returns a `stepfun` object which can be more parsimonious.

Value

`isoreg()` returns an object of class `isoreg` which is basically a list with components

`x`	original (constructed) abscissa values `x`.
`y`	corresponding y values.
`yf`	fitted values corresponding to *ordered* x values.
`yc`	cumulative y values corresponding to *ordered* x values.
`iKnots`	integer vector giving indices where the fitted curve jumps, i.e., where the convex minorant has kinks.
`isOrd`	logical indicating if original x values were ordered increasingly already.
`ord`	`if(!isOrd)`: integer permutation `order(x)` of *original* `x`.
`call`	the `call` to `isoreg()` used.

Note

The inputs can be long vectors, but `iKnots` will wrap around at 2^{31}.

The code should be improved to accept *weights* additionally and solve the corresponding weighted least squares problem.
'Patches are welcome!'

References

Barlow, R. E., Bartholomew, D. J., Bremner, J. M., and Brunk, H. D. (1972) *Statistical Inference under Order Restrictions*; Wiley, London.

Robertson, T., Wright, F. T. and Dykstra, R. L. (1988) *Order Restricted Statistical Inference*; Wiley, New York.

See Also

the plotting method `plot.isoreg` with more examples; `isoMDS()` from the **MASS** package internally uses isotonic regression.

Examples

```
require(graphics)

(ir <- isoreg(c(1,0,4,3,3,5,4,2,0)))
plot(ir, plot.type = "row")

(ir3 <- isoreg(y3 <- c(1,0,4,3,3,5,4,2, 3))) # last "3", not "0"
```

```
(fi3 <- as.stepfun(ir3))
(ir4 <- isoreg(1:10, y4 <- c(5, 9, 1:2, 5:8, 3, 8)))
cat(sprintf("R^2 = %.2f\n",
            1 - sum(residuals(ir4)^2) / ((10-1)*var(y4))))

## If you are interested in the knots alone :
with(ir4, cbind(iKnots, yf[iKnots]))

## Example of unordered x[] with ties:
x <- sample((0:30)/8)
y <- exp(x)
x. <- round(x) # ties!
plot(m <- isoreg(x., y))
stopifnot(all.equal(with(m, yf[iKnots]),
                    as.vector(tapply(y, x., mean))))
```

KalmanLike *Kalman Filtering*

Description

Use Kalman Filtering to find the (Gaussian) log-likelihood, or for forecasting or smoothing.

Usage

```
KalmanLike(y, mod, nit = 0L, update = FALSE)
KalmanRun(y, mod, nit = 0L, update = FALSE)
KalmanSmooth(y, mod, nit = 0L)
KalmanForecast(n.ahead = 10L, mod, update = FALSE)

makeARIMA(phi, theta, Delta, kappa = 1e6,
          SSinit = c("Gardner1980", "Rossignol2011"),
          tol = .Machine$double.eps)
```

Arguments

y	a univariate time series.
mod	a list describing the state-space model: see 'Details'.
nit	the time at which the initialization is computed. nit = 0L implies that the initialization is for a one-step prediction, so Pn should not be computed at the first step.
update	if TRUE the update mod object will be returned as attribute "mod" of the result.
n.ahead	the number of steps ahead for which prediction is required.
phi, theta	numeric vectors of length ≥ 0 giving AR and MA parameters.
Delta	vector of differencing coefficients, so an ARMA model is fitted to y[t] - Delta[1]*y[t-1] - ....

kappa	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model.
SSinit	a string specifying the algorithm to compute the Pn part of the state-space initialization; see 'Details'.
tol	tolerance eventually passed to solve.default when SSinit = "Rossignol2011".

Details

These functions work with a general univariate state-space model with state vector 'a', transitions 'a <- T a + R e', $e \sim \mathcal{N}(0, \kappa Q)$ and observation equation 'y = Z'a + eta', $(eta \equiv \eta), \eta \sim \mathcal{N}(0, \kappa h)$. The likelihood is a profile likelihood after estimation of κ.

The model is specified as a list with at least components

T the transition matrix

Z the observation coefficients

h the observation variance

V 'RQR''

a the current state estimate

P the current estimate of the state uncertainty matrix Q

Pn the estimate at time $t-1$ of the state uncertainty matrix Q (not updated by KalmanForecast).

KalmanSmooth is the workhorse function for tsSmooth.

makeARIMA constructs the state-space model for an ARIMA model, see also arima.

The state-space initialization has used Gardner et al.'s method (SSinit = "Gardner1980"), as only method for years. However, that suffers sometimes from deficiencies when close to non-stationarity. For this reason, it may be replaced as default in the future and only kept for reproducibility reasons. Explicit specification of SSinit is therefore recommended, notably also in arima(). The "Rossignol2011" method has been proposed and partly documented by Raphael Rossignol, Univ. Grenoble, on 2011-09-20 (see PR#14682, below), and later been ported to C by Matwey V. Kornilov. It computes the covariance matrix of $(X_{t-1}, ..., X_{t-p}, Z_t, ..., Z_{t-q})$ by the method of difference equations (page 93 of Brockwell and Davis (1991)), apparently suggested by a referee of Gardner et al. (see p.314 of their paper).

Value

For KalmanLike, a list with components Lik (the log-likelihood less some constants) and s2, the estimate of κ.

For KalmanRun, a list with components values, a vector of length 2 giving the output of KalmanLike, resid (the residuals) and states, the contemporaneous state estimates, a matrix with one row for each observation time.

For KalmanSmooth, a list with two components. Component smooth is a n by p matrix of state estimates based on all the observations, with one row for each time. Component var is a n by p by p array of variance matrices.

For KalmanForecast, a list with components pred, the predictions, and var, the unscaled variances of the prediction errors (to be multiplied by s2).

For makeARIMA, a model list including components for its arguments.

Warning

These functions are designed to be called from other functions which check the validity of the arguments passed, so very little checking is done.

References

Brockwell, P. J. and Davis, R. A. (1991). *Time Series: Theory and Methods*, second edition. Springer.

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980). Algorithm AS 154: An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics*, **29**, 311–322. doi:10.2307/2346910.

R bug report PR#14682 (2011-2013) https://bugs.r-project.org/show_bug.cgi?id=14682.

See Also

arima, StructTS. tsSmooth.

Examples

```
## an ARIMA fit
fit3 <- arima(presidents, c(3, 0, 0))
predict(fit3, 12)
## reconstruct this
pr <- KalmanForecast(12, fit3$model)
pr$pred + fit3$coef[4]
sqrt(pr$var * fit3$sigma2)
## and now do it year by year
mod <- fit3$model
for(y in 1:3) {
  pr <- KalmanForecast(4, mod, TRUE)
  print(list(pred = pr$pred + fit3$coef["intercept"],
             se = sqrt(pr$var * fit3$sigma2)))
  mod <- attr(pr, "mod")
}
```

---

kernapply *Apply Smoothing Kernel*

Description

kernapply computes the convolution between an input sequence and a specific kernel.

Usage

```
kernapply(x, ...)

## Default S3 method:
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'ts'
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'vector'
kernapply(x, k, circular = FALSE, ...)

## S3 method for class 'tskernel'
kernapply(x, k, ...)
```

Arguments

x	an input vector, matrix, time series or kernel to be smoothed.
k	smoothing "tskernel" object.
circular	a logical indicating whether the input sequence to be smoothed is treated as circular, i.e., periodic.
...	arguments passed to or from other methods.

Value

A smoothed version of the input sequence.

Note

This uses fft to perform the convolution, so is fastest when NROW(x) is a power of 2 or some other highly composite integer.

Author(s)

A. Trapletti

See Also

kernel, convolve, filter, spectrum

Examples

```
## see 'kernel' for examples
```

kernel *Smoothing Kernel Objects*

Description

The "tskernel" class is designed to represent discrete symmetric normalized smoothing kernels. These kernels can be used to smooth vectors, matrices, or time series objects.

There are print, plot and [methods for these kernel objects.

Usage

```
kernel(coef, m = 2, r, name)

df.kernel(k)
bandwidth.kernel(k)
is.tskernel(k)

## S3 method for class 'tskernel'
plot(x, type = "h", xlab = "k", ylab = "W[k]",
     main = attr(x,"name"), ...)
```

Arguments

coef	the upper half of the smoothing kernel coefficients (including coefficient zero) *or* the name of a kernel (currently "daniell", "dirichlet", "fejer" or "modified.daniell").
m	the kernel dimension(s) if coef is a name. When m has length larger than one, it means the convolution of kernels of dimension m[j], for j in 1:length(m). Currently this is supported only for the named "*daniell" kernels.
name	the name the kernel will be called.
r	the kernel order for a Fejer kernel.
k, x	a "tskernel" object.
type, xlab, ylab, main, ...	arguments passed to plot.default.

Details

kernel is used to construct a general kernel or named specific kernels. The modified Daniell kernel halves the end coefficients.

The [method allows natural indexing of kernel objects with indices in (-m) : m. The normalization is such that for k <- kernel(*), sum(k[-k$m : k$m]) is one.

df.kernel returns the 'equivalent degrees of freedom' of a smoothing kernel as defined in Brockwell and Davis (1991), page 362, and bandwidth.kernel returns the equivalent bandwidth as defined in Bloomfield (1976), p. 201, with a continuity correction.

Value

`kernel()` returns an object of class `"tskernel"` which is basically a list with the two components `coef` and the kernel dimension `m`. An additional attribute is `"name"`.

Author(s)

A. Trapletti; modifications by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer, pp. 350–365.

See Also

`kernapply`

Examples

```
require(graphics)

## Demonstrate a simple trading strategy for the
## financial time series German stock index DAX.
x <- EuStockMarkets[,1]
k1 <- kernel("daniell", 50)  # a long moving average
k2 <- kernel("daniell", 10)  # and a short one
plot(k1)
plot(k2)
x1 <- kernapply(x, k1)
x2 <- kernapply(x, k2)
plot(x)
lines(x1, col = "red")    # go long if the short crosses the long upwards
lines(x2, col = "green")  # and go short otherwise

## More interesting kernels
kd <- kernel("daniell", c(3, 3))
kd # note the unusual indexing
kd[-2:2]
plot(kernel("fejer", 100, r = 6))
plot(kernel("modified.daniell", c(7,5,3)))

# Reproduce example 10.4.3 from Brockwell and Davis (1991)
spectrum(sunspot.year, kernel = kernel("daniell", c(11,7,3)), log = "no")
```

kmeans *K-Means Clustering*

Description

Perform k-means clustering on a data matrix.

Usage

```
kmeans(x, centers, iter.max = 10, nstart = 1,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy",
                     "MacQueen"), trace = FALSE)
## S3 method for class 'kmeans'
fitted(object, method = c("centers", "classes"), ...)
```

Arguments

x	numeric matrix of data, or an object that can be coerced to such a matrix (such as a numeric vector or a data frame with all numeric columns).
centers	either the number of clusters, say k, or a set of initial (distinct) cluster centres. If a number, a random set of (distinct) rows in x is chosen as the initial centres.
iter.max	the maximum number of iterations allowed.
nstart	if centers is a number, how many random sets should be chosen?
algorithm	character: may be abbreviated. Note that "Lloyd" and "Forgy" are alternative names for one algorithm.
object	an R object of class "kmeans", typically the result ob of ob <- kmeans(..).
method	character: may be abbreviated. "centers" causes fitted to return cluster centers (one for each input point) and "classes" causes fitted to return a vector of class assignments.
trace	logical or integer number, currently only used in the default method ("Hartigan-Wong"): if positive (or true), tracing information on the progress of the algorithm is produced. Higher values may produce more tracing information.
...	not used.

Details

The data given by x are clustered by the k-means method, which aims to partition the points into k groups such that the sum of squares from points to the assigned cluster centres is minimized. At the minimum, all cluster centres are at the mean of their Voronoi sets (the set of data points which are nearest to the cluster centre).

The algorithm of Hartigan and Wong (1979) is used by default. Note that some authors use k-means to refer to a specific algorithm rather than the general method: most commonly the algorithm given by MacQueen (1967) but sometimes that given by Lloyd (1957) and Forgy (1965). The

Hartigan–Wong algorithm generally does a better job than either of those, but trying several random starts (`nstart`> 1) is often recommended. In rare cases, when some of the points (rows of `x`) are extremely close, the algorithm may not converge in the "Quick-Transfer" stage, signalling a warning (and returning `ifault = 4`). Slight rounding of the data may be advisable in that case.

For ease of programmatic exploration, $k = 1$ is allowed, notably returning the center and `withinss`.

Except for the Lloyd–Forgy method, k clusters will always be returned if a number is specified. If an initial matrix of centres is supplied, it is possible that no point will be closest to one or more centres, which is currently an error for the Hartigan–Wong method.

Value

`kmeans` returns an object of class `"kmeans"` which has a `print` and a `fitted` method. It is a list with at least the following components:

`cluster`	A vector of integers (from `1:k`) indicating the cluster to which each point is allocated.
`centers`	A matrix of cluster centres.
`totss`	The total sum of squares.
`withinss`	Vector of within-cluster sum of squares, one component per cluster.
`tot.withinss`	Total within-cluster sum of squares, i.e. `sum(withinss)`.
`betweenss`	The between-cluster sum of squares, i.e. `totss-tot.withinss`.
`size`	The number of points in each cluster.
`iter`	The number of (outer) iterations.
`ifault`	integer: indicator of a possible algorithm problem – for experts.

Note

The clusters are numbered in the returned object, but they are a *set* and no ordering is implied. (Their apparent ordering may differ by platform.)

References

Forgy, E. W. (1965). Cluster analysis of multivariate data: efficiency vs interpretability of classifications. *Biometrics*, **21**, 768–769.

Hartigan, J. A. and Wong, M. A. (1979). Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics*, **28**, 100–108. doi:10.2307/2346830.

Lloyd, S. P. (1957, 1982). Least squares quantization in PCM. Technical Note, Bell Laboratories. Published in 1982 in *IEEE Transactions on Information Theory*, **28**, 128–137.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, eds L. M. Le Cam & J. Neyman, **1**, pp. 281–297. Berkeley, CA: University of California Press.

Examples

```
require(graphics)

# a 2-dimensional example
x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
colnames(x) <- c("x", "y")
(cl <- kmeans(x, 2))
plot(x, col = cl$cluster)
points(cl$centers, col = 1:2, pch = 8, cex = 2)

# sum of squares
ss <- function(x) sum(scale(x, scale = FALSE)^2)

## cluster centers "fitted" to each obs.:
fitted.x <- fitted(cl);  head(fitted.x)
resid.x <- x - fitted(cl)

## Equalities : ----------------------------------
cbind(cl[c("betweenss", "tot.withinss", "totss")], # the same two columns
         c(ss(fitted.x), ss(resid.x),    ss(x)))
stopifnot(all.equal(cl$ totss,        ss(x)),
   all.equal(cl$ tot.withinss, ss(resid.x)),
   ## these three are the same:
   all.equal(cl$ betweenss,    ss(fitted.x)),
   all.equal(cl$ betweenss, cl$totss - cl$tot.withinss),
   ## and hence also
   all.equal(ss(x), ss(fitted.x) + ss(resid.x))
   )

kmeans(x,1)$withinss # trivial one-cluster, (its W.SS == ss(x))

## random starts do help here with too many clusters
## (and are often recommended anyway!):
## The ordering of the clusters may be platform-dependent.

(cl <- kmeans(x, 5, nstart = 25))

plot(x, col = cl$cluster)
points(cl$centers, col = 1:5, pch = 8)
```

`kruskal.test` *Kruskal-Wallis Rank Sum Test*

Description

Performs a Kruskal-Wallis rank sum test.

Usage

```
kruskal.test(x, ...)

## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula'
kruskal.test(formula, data, subset, na.action, ...)
```

Arguments

x	a numeric vector of data values, or a list of numeric data vectors. Non-numeric elements of a list will be coerced, with a warning.
g	a vector or factor object giving the group for the corresponding elements of x. Ignored with a warning if x is a list.
formula	a formula of the form response ~ group where response gives the data values and group a vector or factor of the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

kruskal.test performs a Kruskal-Wallis rank sum test of the null that the location parameters of the distribution of x are the same in each group (sample). The alternative is that they differ in at least one.

If x is a list, its elements are taken as the samples to be compared, and hence have to be numeric data vectors. In this case, g is ignored, and one can simply use kruskal.test(x) to perform the test. If the samples are not yet contained in a list, use kruskal.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

Value

A list with class "htest" containing the following components:

statistic	the Kruskal-Wallis rank sum statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Kruskal-Wallis rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander and Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 115–120.

See Also

The Wilcoxon rank sum test (`wilcox.test`) as the special case for two samples; `lm` together with `anova` for performing one-way location analysis under normality assumptions; with Student's t test (`t.test`) as the special case for two samples.

`wilcox_test` in package **coin** for exact, asymptotic and Monte Carlo *conditional* p-values, including in the presence of ties.

Examples

```
## Hollander & Wolfe (1973), 116.
## Mucociliary efficiency from the rate of removal of dust in normal
##  subjects, subjects with obstructive airway disease, and subjects
##  with asbestosis.
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)      # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis
kruskal.test(list(x, y, z))
## Equivalently,
x <- c(x, y, z)
g <- factor(rep(1:3, c(5, 4, 5)),
            labels = c("Normal subjects",
                       "Subjects with obstructive airway disease",
                       "Subjects with asbestosis"))
kruskal.test(x, g)

## Formula interface.
require(graphics)
boxplot(Ozone ~ Month, data = airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

---

ks.test *Kolmogorov-Smirnov Tests*

Description

Perform a one- or two-sample Kolmogorov-Smirnov test.

Usage

```
ks.test(x, ...)
## Default S3 method:
ks.test(x, y, ...,
        alternative = c("two.sided", "less", "greater"),
```

```
        exact = NULL, simulate.p.value = FALSE, B = 2000)
## S3 method for class 'formula'
ks.test(formula, data, subset, na.action, ...)
```

Arguments

x	a numeric vector of data values.
y	either a numeric vector of data values, or a character string naming a cumulative distribution function or an actual cumulative distribution function such as pnorm. Only continuous CDFs are valid.
...	for the default method, parameters of the distribution specified (as a character string) by y. Otherwise, further arguments to be passed to or from methods.
alternative	indicates the alternative hypothesis and must be one of "two.sided" (default), "less", or "greater". You can specify just the initial letter of the value, but the argument name must be given in full. See 'Details' for the meanings of the possible values.
exact	NULL or a logical indicating whether an exact p-value should be computed. See 'Details' for the meaning of NULL.
simulate.p.value	a logical indicating whether to compute p-values by Monte Carlo simulation. (Ignored for the one-sample test.)
B	an integer specifying the number of replicates used in the Monte Carlo test.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs either 1 for a one-sample test or a factor with two levels giving the corresponding groups for a two-sample test.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").

Details

If y is numeric, a two-sample (Smirnov) test of the null hypothesis that x and y were drawn from the same distribution is performed.

Alternatively, y can be a character string naming a continuous (cumulative) distribution function, or such a function. In this case, a one-sample (Kolmogorov) test is carried out of the null that the distribution function which generated x is distribution y with parameters specified by .... The presence of ties always generates a warning in the one-sample case, as continuous distributions do not generate them. If the ties arose from rounding the tests may be approximately valid, but even modest amounts of rounding can have a significant effect on the calculated statistic.

Missing values are silently omitted from x and (in the two-sample case) y.

The possible values "two.sided", "less" and "greater" of alternative specify the null hypothesis that the true cumulative distribution function (CDF) of x is equal to, not less than or not

greater than the hypothesized CDF (one-sample case) or the CDF of y (two-sample case), respectively. The test compares the CDFs taking their maximal difference as test statistic, with the statistic in the "greater" alternative being $D^+ = \max_u[F_x(u) - F_y(u)]$. Thus in the two-sample case alternative = "greater" includes distributions for which x is stochastically *smaller* than y (the CDF of x lies above and hence to the left of that for y), in contrast to t.test or wilcox.test.

Exact p-values are not available for the one-sample case in the presence of ties. If exact = NULL (the default), an exact p-value is computed if the sample size is less than 100 in the one-sample case *and there are no ties*, and if the product of the sample sizes is less than 10000 in the two-sample case, with or without ties (using the algorithm described in Schröer and Trenkler (1995)). Otherwise, the p-value is computed via Monte Carlo simulation in the two-sample case if simulate.p.value is TRUE, or else asymptotic distributions are used whose approximations may be inaccurate in small samples. In the one-sample two-sided case, exact p-values are obtained as described in Marsaglia, Tsang & Wang (2003) (but not using the optional approximation in the right tail, so this can be slow for small p-values). The formula of Birnbaum & Tingey (1951) is used for the one-sample one-sided case.

If a one-sample test is used, the parameters specified in ... must be pre-specified and not estimated from the data. There is some more refined distribution theory for the KS test with estimated parameters (see Durbin, 1973), but that is not implemented in ks.test.

Value

A list inheriting from classes "ks.test" and "htest" containing the following components:

statistic	the value of the test statistic.
p.value	the p-value of the test.
alternative	a character string describing the alternative hypothesis.
method	a character string indicating what type of test was performed.
data.name	a character string giving the name(s) of the data.

Source

The two-sided one-sample distribution comes *via* Marsaglia, Tsang and Wang (2003).

Exact distributions for the two-sample (Smirnov) test are computed by the algorithm proposed by Schröer (1991) and Schröer & Trenkler (1995) using numerical improvements along the lines of Viehmann (2021).

References

Z. W. Birnbaum and Fred H. Tingey (1951). One-sided confidence contours for probability distribution functions. *The Annals of Mathematical Statistics*, **22**/4, 592–596. doi:10.1214/aoms/1177729550.

William J. Conover (1971). *Practical Nonparametric Statistics*. New York: John Wiley & Sons. Pages 295–301 (one-sample Kolmogorov test), 309–314 (two-sample Smirnov test).

Durbin, J. (1973). *Distribution theory for tests based on the sample distribution function*. SIAM.

W. Feller (1948). On the Kolmogorov-Smirnov limit theorems for empirical distributions. *The Annals of Mathematical Statistics*, **19**(2), 177–189. doi:10.1214/aoms/1177730243.

George Marsaglia, Wai Wan Tsang and Jingbo Wang (2003). Evaluating Kolmogorov's distribution. *Journal of Statistical Software*, **8**/18. doi:10.18637/jss.v008.i18.

Gunar Schröer (1991). Computergestützte statistische Inferenz am Beispiel der Kolmogorov-Smirnov Tests. Diplomarbeit Universität Osnabrück.

Gunar Schröer and Dietrich Trenkler (1995). Exact and Randomization Distributions of Kolmogorov-Smirnov Tests for Two or Three Samples. *Computational Statistics & Data Analysis*, **20**(2), 185–202. doi:10.1016/01679473(94)00040P.

Thomas Viehmann (2021). Numerically more stable computation of the p-values for the two-sample Kolmogorov-Smirnov test. https://arxiv.org/abs/2102.08037.

See Also

psmirnov.

shapiro.test which performs the Shapiro-Wilk test for normality.

Examples

```
require("graphics")

x <- rnorm(50)
y <- runif(30)
# Do x and y come from the same distribution?
ks.test(x, y)
# Does x come from a shifted gamma distribution with shape 3 and rate 2?
ks.test(x+2, "pgamma", 3, 2) # two-sided, exact
ks.test(x+2, "pgamma", 3, 2, exact = FALSE)
ks.test(x+2, "pgamma", 3, 2, alternative = "gr")

# test if x is stochastically larger than x2
x2 <- rnorm(50, -1)
plot(ecdf(x), xlim = range(c(x, x2)))
plot(ecdf(x2), add = TRUE, lty = "dashed")
t.test(x, x2, alternative = "g")
wilcox.test(x, x2, alternative = "g")
ks.test(x, x2, alternative = "l")

# with ties, example from Schröer and Trenkler (1995)
# D = 3/7, p = 8/33 = 0.242424..
ks.test(c(1, 2, 2, 3, 3),
        c(1, 2, 3, 3, 4, 5, 6))# -> exact

# formula interface, see ?wilcox.test
ks.test(Ozone ~ Month, data = airquality,
        subset = Month %in% c(5, 8))
```

`ksmooth` *Kernel Regression Smoother*

Description

The Nadaraya–Watson kernel regression estimate.

Usage

```
ksmooth(x, y, kernel = c("box", "normal"), bandwidth = 0.5,
        range.x = range(x),
        n.points = max(100L, length(x)), x.points)
```

Arguments

`x`	input x values. Long vectors are supported.
`y`	input y values. Long vectors are supported.
`kernel`	the kernel to be used. Can be abbreviated.
`bandwidth`	the bandwidth. The kernels are scaled so that their quartiles (viewed as probability densities) are at $\pm$ `0.25*bandwidth`.
`range.x`	the range of points to be covered in the output.
`n.points`	the number of points at which to evaluate the fit.
`x.points`	points at which to evaluate the smoothed fit. If missing, `n.points` are chosen uniformly to cover `range.x`. Long vectors are supported.

Value

A list with components

`x`	values at which the smoothed fit is evaluated. Guaranteed to be in increasing order.
`y`	fitted values corresponding to `x`.

Note

This function was implemented for compatibility with S, although it is nowhere near as slow as the S function. Better kernel smoothers are available in other packages such as **KernSmooth**.

Examples

```
require(graphics)

with(cars, {
    plot(speed, dist)
    lines(ksmooth(speed, dist, "normal", bandwidth = 2), col = 2)
    lines(ksmooth(speed, dist, "normal", bandwidth = 5), col = 3)
})
```

`lag` *Lag a Time Series*

Description

Compute a lagged version of a time series, shifting the time base back by a given number of observations.

`lag` is a generic function; this page documents its default method.

Usage

```
lag(x, ...)

## Default S3 method:
lag(x, k = 1, ...)
```

Arguments

`x`	A vector or matrix or univariate or multivariate time series
`k`	The number of lags (in units of observations).
`...`	further arguments to be passed to or from methods.

Details

Vector or matrix arguments `x` are given a `tsp` attribute *via* `hasTsp`.

Value

A time series object with the same class as `x`.

Note

Note the sign of `k`: a series lagged by a positive `k` starts *earlier*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`diff`, `deltat`

Examples

```
lag(ldeaths, 12) # starts one year earlier
```

lag.plot *Time Series Lag Plots*

Description

Plot time series against lagged versions of themselves. Helps visualizing 'auto-dependence' even when auto-correlations vanish.

Usage

```
lag.plot(x, lags = 1, layout = NULL, set.lags = 1:lags,
         main = NULL, asp = 1,
         diag = TRUE, diag.col = "gray", type = "p", oma = NULL,
         ask = NULL, do.lines = (n <= 150), labels = do.lines,
         ...)
```

Arguments

x	time-series (univariate or multivariate)
lags	number of lag plots desired, see argument set.lags.
layout	the layout of multiple plots, basically the mfrow par() argument. The default uses about a square layout (see n2mfrow) such that all plots are on one page.
set.lags	vector of positive integers allowing specification of the set of lags used; defaults to 1:lags.
main	character with a main header title to be done on the top of each page.
asp	Aspect ratio to be fixed, see plot.default.
diag	logical indicating if the x=y diagonal should be drawn.
diag.col	color to be used for the diagonal if(diag).
type	plot type to be used, but see plot.ts about its restricted meaning.
oma	outer margins, see par.
ask	logical or NULL; if true, the user is asked to confirm before a new page is started.
do.lines	logical indicating if lines should be drawn.
labels	logical indicating if labels should be used.
...	Further arguments to plot.ts. Several graphical parameters are set in this function and so cannot be changed: these include xlab, ylab, mgp, col.lab and font.lab: this also applies to the arguments xy.labels and xy.lines.

Details

If just one plot is produced, this is a conventional plot. If more than one plot is to be produced, par(mfrow) and several other graphics parameters will be set, so it is not (easily) possible to mix such lag plots with other plots on the same page.

If ask = NULL, par(ask = TRUE) will be called if more than one page of plots is to be produced and the device is interactive.

Note

It is more flexible and has different default behaviour than the S version. We use `main =` instead of `head =` for internal consistency.

Author(s)

Martin Maechler

See Also

`plot.ts` which is the basic work horse.

Examples

```
require(graphics)

lag.plot(nhtemp, 8, diag.col = "forest green")
lag.plot(nhtemp, 5, main = "Average Temperatures in New Haven")
## ask defaults to TRUE when we have more than one page:
lag.plot(nhtemp, 6, layout = c(2,1), asp = NA,
         main = "New Haven Temperatures", col.main = "blue")

## Multivariate (but non-stationary! ...)
lag.plot(freeny.x, lags = 3)

## no lines for long series :
lag.plot(sqrt(sunspots), set.lags = c(1:4, 9:12), pch = ".", col = "gold")
```

---

line *Robust Line Fitting*

Description

Fit a line robustly as recommended in *Exploratory Data Analysis*.

Currently by default (`iter = 1`) the initial median-median line is *not* iterated (as opposed to Tukey's "resistant line" in the references).

Usage

```
line(x, y, iter = 1)
```

Arguments

`x, y`	the arguments can be any way of specifying x-y pairs. See `xy.coords`.
`iter`	positive integer specifying the number of "polishing" iterations. Note that this was hard coded to 1 in R versions before 3.5.0, and more importantly that such simple iterations may not converge, see Siegel's 9-point example.

Details

Cases with missing values are omitted.

Contrary to the references where the data is split in three (almost) equally sized groups with symmetric sizes depending on n and `n %% 3` and computes medians inside each group, the `line()` code splits into three groups using all observations with `x[.] <= q1` and `x[.] >= q2`, where `q1, q2` are (a kind of) quantiles for probabilities $p = 1/3$ and $p = 2/3$ of the form `(x[j1]+x[j2])/2` where `j1 = floor(p*(n-1))` and `j2 = ceiling(p(n-1))`, `n = length(x)`.

Long vectors are not supported yet.

Value

An object of class `"tukeyline"`.

Methods are available for the generic functions `coef`, `residuals`, `fitted`, and `print`.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

Velleman, P. F. and Hoaglin, D. C. (1981). *Applications, Basics and Computing of Exploratory Data Analysis*, Duxbury Press. Chapter 5.

Emerson, J. D. and Hoaglin, D. C. (1983). Resistant Lines for y versus x. Chapter 5 of *Understanding Robust and Exploratory Data Analysis*, eds. David C. Hoaglin, Frederick Mosteller and John W. Tukey. Wiley.

Iain M. Johnstone and Paul F. Velleman (1985). The Resistant Line and Related Regression Methods. *Journal of the American Statistical Association*, **80**, 1041–1054. doi:10.2307/2288572.

See Also

`lm`.

There are alternatives for robust linear regression more robust and more (statistically) efficient, see `rlm()` from **MASS**, or `lmrob()` from **robustbase**.

Examples

```
require(graphics)

plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))

## Andrew Siegel's pathological 9-point data, y-values multiplied by 3:
d.AS <- data.frame(x = c(-4:3, 12), y = 3*c(rep(0,6), -5, 5, 1))
cAS <- with(d.AS, t(sapply(1:10,
                   function(it) line(x,y, iter=it)$coefficients)))
dimnames(cAS) <- list(paste("it =", format(1:10)), c("intercept", "slope"))
cAS
## iterations started to oscillate, repeating iteration 7,8 indefinitely
```

listof *A Class for Lists of (Parts of) Model Fits*

Description

Class "listof" is used by aov and the "lm" method of alias for lists of model fits or parts thereof. It is simply a list with an assigned class to control the way methods, especially printing, act on it.

It has a coef method in this package (which returns an object of this class), and [and print methods in package **base**.

lm *Fitting Linear Models*

Description

lm is used to fit linear models, including multivariate ones. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)

## S3 method for class 'lm'
print(x, digits = max(3L, getOption("digits") - 3L), ...)
```

Arguments

formula	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
subset	an optional vector specifying a subset of observations to be used in the fitting process. (See additional details about how this argument interacts with data-dependent bases in the 'Details' section of the model.frame documentation.)
weights	an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights weights (that is, minimizing sum(w*e^2)); otherwise ordinary least squares is used. See also 'Details',

na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
method	the method to be used; for fitting, currently only method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).
model, x, y, qr	logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.
singular.ok	logical. If FALSE (the default in S but not in R) a singular fit is an error.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector or matrix of extents matching those of the response. One or more offset terms can be included in the formula instead or as well, and if more than one are specified their sum is used. See model.offset.
...	For lm(): additional arguments to be passed to the low level regression fitting functions (see below).
digits	the number of *significant* digits to be passed to format(coef(x), .) when print()ing.

Details

Models for lm are specified symbolically. A typical model has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. A specification of the form first:second indicates the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

If the formula includes an offset, this is evaluated and subtracted from the response.

If response is a matrix a linear model is fitted separately by least-squares to each column of the matrix and the result inherits from "mlm" ("multivariate linear model").

See model.matrix for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula (see aov and demo(glm.vr) for an example).

A formula has an implied intercept term. To remove this use either y ~ x - 1 or y ~ 0 + x. See formula for more details of allowed formulae.

Non-NULL weights can be used to indicate that different observations have different variances (with the values in weights being inversely proportional to the variances); or equivalently, when the elements of weights are positive integers w_i, that each response y_i is the mean of w_i unit-weight observations (including the case that there are w_i observations equal to y_i and the data have been summarized). However, in the latter case, notice that within-group variation is not used. Therefore, the sigma estimate and residual degrees of freedom may be suboptimal; in the case of replication

weights, even wrong. Hence, standard errors and analysis of variance tables should be treated with care.

lm calls the lower level functions lm.fit, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

Value

lm returns an object of class "lm" or for multivariate ('multiple') responses of class c("mlm", "lm").

The functions summary and anova are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions coefficients, effects, fitted.values and residuals extract various useful features of the value returned by lm.

An object of class "lm" is a list containing at least the following components:

coefficients	a named vector of coefficients
residuals	the residuals, that is response minus fitted values.
fitted.values	the fitted mean values.
rank	the numeric rank of the fitted linear model.
weights	(only for weighted fits) the specified weights.
df.residual	the residual degrees of freedom.
call	the matched call.
terms	the terms object used.
contrasts	(only where relevant) the contrasts used.
xlevels	(only where relevant) a record of the levels of the factors used in fitting.
offset	the offset used (missing if none were used).
y	if requested, the response used.
x	if requested, the model matrix used.
model	if requested (the default), the model frame used.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-null fits will have components assign, effects and (unless not requested) qr relating to the linear fit, for use by extractor functions such as summary and effects.

Using time series

Considerable care is needed when using lm with time series.

Unless na.action = NULL, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by ts.intersect(..., dframe = TRUE), then apply a suitable na.action to that data frame and call lm with na.action = NULL so that residuals and fitted values are time series.

Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973). Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, 392–399. doi:10.2307/2346786.

See Also

`summary.lm` for more detailed summaries and `anova.lm` for the ANOVA table; `aov` for a different interface.

The generic functions `coef`, `effects`, `residuals`, `fitted`, `vcov`.

`predict.lm` (via `predict`) for prediction, including confidence and prediction intervals; `confint` for confidence intervals of *parameters*.

`lm.influence` for regression diagnostics, and `glm` for **generalized** linear models.

The underlying low level functions, `lm.fit` for plain, and `lm.wfit` for weighted regression fitting.

More `lm()` examples are available e.g., in `anscombe`, `attitude`, `freeny`, `LifeCycleSavings`, `longley`, `stackloss`, `swiss`.

`biglm` in package **biglm** for an alternative way to fit linear models to large datasets (especially those with many cases).

Examples

```
require(graphics)

## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D90 <- lm(weight ~ group - 1) # omitting intercept

anova(lm.D9)
summary(lm.D90)

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

### less simple examples in "See Also" above
```

lm.fit *Fitter Functions for Linear Models*

Description

These are the basic computing engines called by lm used to fit linear models. These should usually *not* be used directly unless by experienced users. .lm.fit() is a bare-bones wrapper to the innermost QR-based C code, on which glm.fit and lsfit are also based, for even more experienced users.

Usage

```
lm.fit (x, y,    offset = NULL, method = "qr", tol = 1e-7,
       singular.ok = TRUE, ...)

lm.wfit(x, y, w, offset = NULL, method = "qr", tol = 1e-7,
        singular.ok = TRUE, ...)

.lm.fit(x, y, tol = 1e-7)
```

Arguments

x	design matrix of dimension n * p.
y	vector of observations of length n, or a matrix with n rows.
w	vector of weights (length n) to be used in the fitting process for the wfit functions. Weighted least squares is used with weights w, i.e., sum(w * e^2) is minimized.
offset	(numeric of length n). This can be used to specify an *a priori* known component to be included in the linear predictor during fitting.
method	currently, only method = "qr" is supported.
tol	tolerance for the qr decomposition. Default is 1e-7.
singular.ok	logical. If FALSE, a singular model is an error.
...	currently disregarded.

Details

If y is a matrix, offset can be a numeric matrix of the same dimensions, in which case each column is applied to the corresponding column of y.

Value

a list with components (for lm.fit and lm.wfit)

coefficients	p vector
residuals	n vector or matrix

`fitted.values`	n vector or matrix
`effects`	n vector of orthogonal single-df effects. The first `rank` of them correspond to non-aliased coefficients, and are named accordingly.
`weights`	n vector — *only* for the `*wfit*` functions.
`rank`	integer, giving the rank
`df.residual`	degrees of freedom of residuals
`qr`	the QR decomposition, see `qr`.

Fits without any columns or non-zero weights do not have the `effects` and `qr` components.

`.lm.fit()` returns a subset of the above, the `qr` part unwrapped, plus a logical component `pivoted` indicating if the underlying QR algorithm did pivot.

See Also

`lm` which you should use for linear least squares regression, unless you know better.

Examples

```
require(utils)
set.seed(129)

n <- 7 ; p <- 2
X <- matrix(rnorm(n * p), n, p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2

str(lmw <- lm.wfit(x = X, y = y, w = w))

str(lm. <- lm.fit (x = X, y = y))

## fits w/o intercept:
all.equal(unname(coef(lm(y ~ X-1))),
          unname(coef( lm.fit(X,y))))
all.equal(unname(coef( lm.fit(X,y))),
                 coef(.lm.fit(X,y)))

if(require("microbenchmark")) {
  mb <- microbenchmark(lm(y~X-1), lm.fit(X,y), .lm.fit(X,y))
  print(mb)
  boxplot(mb, notch=TRUE)
}
```

`lm.influence` *Regression Diagnostics*

Description

This function provides the basic quantities which are used in forming a wide variety of diagnostics for checking the quality of regression fits.

Usage

```
influence(model, ...)
## S3 method for class 'lm'
influence(model, do.coef = TRUE, ...)
## S3 method for class 'glm'
influence(model, do.coef = TRUE, ...)

lm.influence(model, do.coef = TRUE)
```

Arguments

`model`	an object as returned by `lm` or `glm`.
`do.coef`	logical indicating if the changed `coefficients` (see below) are desired. These need $O(n^2p)$ computing time.
`...`	further arguments passed to or from other methods.

Details

The `influence.measures()` and other functions listed in **See Also** provide a more user oriented way of computing a variety of regression diagnostics. These all build on `lm.influence`. Note that for GLMs (other than the Gaussian family with identity link) these are based on one-step approximations which may be inadequate if a case has high influence.

An attempt is made to ensure that computed hat values that are probably one are treated as one, and the corresponding rows in `sigma` and `coefficients` are `NaN`. (Dropping such a case would normally result in a variable being dropped, so it is not possible to give simple drop-one diagnostics.)

`naresid` is applied to the results and so will fill in with `NA`s it the fit had `na.action = na.exclude`.

Value

A list containing the following components of the same length or number of rows n, which is the number of non-zero weights. Cases omitted in the fit are omitted unless a `na.action` method was used (such as `na.exclude`) which restores them.

`hat`	a vector containing the diagonal of the 'hat' matrix.
`coefficients`	(unless `do.coef` is false) a matrix whose i-th row contains the change in the estimated coefficients which results when the i-th case is dropped from the regression. Note that aliased coefficients are not included in the matrix.

sigma	a vector whose i-th element contains the estimate of the residual standard deviation obtained when the i-th case is dropped from the regression. (The approximations needed for GLMs can result in this being NaN.)
wt.res	a vector of *weighted* (or for class glm rather *deviance*) residuals.

Note

The coefficients returned by the R version of lm.influence differ from those computed by S. Rather than returning the coefficients which result from dropping each case, we return the changes in the coefficients. This is more directly useful in many diagnostic measures.
Since these need $O(np^2)$ computing time, they can be omitted by do.coef = FALSE.

Note that cases with weights == 0 are *dropped* (contrary to the situation in S).

If a model has been fitted with na.action = na.exclude (see na.exclude), cases excluded in the fit *are* considered here.

References

See the list in the documentation for influence.measures.

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

summary.lm for summary and related methods;
influence.measures,
hat for the hat matrix diagonals,
dfbetas, dffits, covratio, cooks.distance, lm.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
summary(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi,
                    data = LifeCycleSavings),
        correlation = TRUE)
utils::str(lmI <- lm.influence(lm.SR))

## For more "user level" examples, use example(influence.measures)
```

---

lm.summaries *Accessing Linear Model Fits*

Description

All these functions are methods for class "lm" objects.

Usage

```
## S3 method for class 'lm'
family(object, ...)

## S3 method for class 'lm'
formula(x, ...)

## S3 method for class 'lm'
residuals(object,
          type = c("working", "response", "deviance", "pearson",
                   "partial"),
          ...)

## S3 method for class 'lm'
labels(object, ...)
```

Arguments

`object`, `x`	an object inheriting from class `lm`, usually the result of a call to `lm` or `aov`.
`...`	further arguments passed to or from other methods.
`type`	the type of residuals which should be returned. Can be abbreviated.

Details

The generic accessor functions `coef`, `effects`, `fitted` and `residuals` can be used to extract various useful features of the value returned by `lm`.

The working and response residuals are 'observed - fitted'. The deviance and Pearson residuals are weighted residuals, scaled by the square root of the weights used in fitting. The partial residuals are a matrix with each column formed by omitting a term from the model. In all these, zero weight cases are never omitted (as opposed to the standardized `rstudent` residuals, and the `weighted.residuals`).

How `residuals` treats cases with missing values in the original fit is determined by the `na.action` argument of that fit. If `na.action = na.omit` omitted cases will not appear in the residuals, whereas if `na.action = na.exclude` they will appear, with residual value `NA`. See also `naresid`.

The `"lm"` method for generic `labels` returns the term labels for estimable terms, that is the names of the terms with an least one estimable coefficient.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function `lm`, `anova.lm`.

coef, deviance, df.residual, effects, fitted, glm for **generalized** linear models, influence (etc on that page) for regression diagnostics, weighted.residuals, residuals, residuals.glm, summary.lm, weights.

influence.measures for deletion diagnostics, including standardized (rstandard) and studentized (rstudent) residuals.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90) # the bare coefficients

## The 2 basic regression diagnostic plots [plot.lm(.) is preferred]
plot(resid(lm.D90), fitted(lm.D90)) # Tukey-Anscombe's
abline(h = 0, lty = 2, col = "gray")

qqnorm(residuals(lm.D90))
```

---

loadings *Print Loadings in Factor Analysis*

Description

Extract or print loadings in factor analysis (or principal components analysis).

Usage

```
loadings(x, ...)

## S3 method for class 'loadings'
print(x, digits = 3, cutoff = 0.1, sort = FALSE, ...)

## S3 method for class 'factanal'
print(x, digits = 3, ...)
```

Arguments

x	an object of class "factanal" or "princomp" or the loadings component of such an object.
digits	number of decimal places to use in printing uniquenesses and loadings.
cutoff	loadings smaller than this (in absolute value) are suppressed.
sort	logical. If true, the variables are sorted by their importance on each factor. Each variable with any loading larger than 0.5 (in modulus) is assigned to the factor with the largest loading, and the variables are printed in the order of the factor they are assigned to, then those unassigned.
...	further arguments for other methods, ignored for loadings.

Details

'Loadings' is a term from *factor analysis*, but because factor analysis and principal component analysis (PCA) are often conflated in the social science literature, it was used for PCA by SPSS and hence by princomp in S-PLUS to help SPSS users.

Small loadings are conventionally not printed (replaced by spaces), to draw the eye to the pattern of the larger loadings.

The print method for class "factanal" calls the "loadings" method to print the loadings, and so passes down arguments such as cutoff and sort.

The signs of the loadings vectors are arbitrary for both factor analysis and PCA.

Note

There are other functions called loadings in contributed packages which are S3 or S4 generic: the ... argument is to make it easier for this one to become a default method.

See Also

factanal, princomp

---

loess *Local Polynomial Regression Fitting*

Description

Fit a locally polynomial surface determined by one or more numerical predictors, using local fitting.

Usage

```
loess(formula, data, weights, subset, na.action, model = FALSE,
      span = 0.75, enp.target, degree = 2,
      parametric = FALSE, drop.square = FALSE, normalize = TRUE,
      family = c("gaussian", "symmetric"),
      method = c("loess", "model.frame"),
      control = loess.control(...), ...)
```

Arguments

formula	a formula specifying the numeric response and one to four numeric predictors (best specified via an interaction, but can also be specified additively). Will be coerced to a formula if necessary.
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which loess is called.
weights	optional weights for each case.

`subset`	an optional specification of a subset of the data to be used.
`na.action`	the action to be taken with missing values in the response or predictors. The default is given by `getOption("na.action")`.
`model`	should the model frame be returned?
`span`	the parameter α which controls the degree of smoothing.
`enp.target`	an alternative way to specify `span`, as the approximate equivalent number of parameters to be used.
`degree`	the degree of the polynomials to be used, normally 1 or 2. (Degree 0 is also allowed, but see the 'Note'.)
`parametric`	should any terms be fitted globally rather than locally? Terms can be specified by name, number or as a logical vector of the same length as the number of predictors.
`drop.square`	for fits with more than one predictor and `degree = 2`, should the quadratic term be dropped for particular predictors? Terms are specified in the same way as for `parametric`.
`normalize`	should the predictors be normalized to a common scale if there is more than one? The normalization used is to set the 10% trimmed standard deviation to one. Set to false for spatial coordinate predictors and others known to be on a common scale.
`family`	if `"gaussian"` fitting is by least-squares, and if `"symmetric"` a re-descending M estimator is used with Tukey's biweight function. Can be abbreviated.
`method`	fit the model or just extract the model frame. Can be abbreviated.
`control`	control parameters: see `loess.control`.
`...`	control parameters can also be supplied directly (*if* `control` is not specified).

Details

Fitting is done locally. That is, for the fit at point x, the fit is made using points in a neighbourhood of x, weighted by their distance from x (with differences in 'parametric' variables being ignored when computing the distance). The size of the neighbourhood is controlled by α (set by `span` or `enp.target`). For $\alpha < 1$, the neighbourhood includes proportion α of the points, and these have tricubic weighting (proportional to $(1 - (\mathrm{dist}/\mathrm{maxdist})^3)^3$). For $\alpha > 1$, all points are used, with the 'maximum distance' assumed to be $\alpha^{1/p}$ times the actual maximum distance for p explanatory variables.

For the default family, fitting is by (weighted) least squares. For `family="symmetric"` a few iterations of an M-estimation procedure with Tukey's biweight are used. Be aware that as the initial value is the least-squares fit, this need not be a very resistant fit.

It can be important to tune the control list to achieve acceptable speed. See `loess.control` for details.

Value

An object of class `"loess"`, with `print()`, `summary()`, `predict` and `anova` methods.

Note

As this is based on `cloess`, it is similar to but not identical to the `loess` function of S. In particular, conditioning is not implemented.

The memory usage of this implementation of `loess` is roughly quadratic in the number of points, with 1000 points taking about 10Mb.

`degree = 0`, local constant fitting, is allowed in this implementation but not documented in the reference. It seems very little tested, so use with caution.

Author(s)

B. D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu.

Source

The 1998 version of `cloess` package of Cleveland, Grosse and Shyu. A later version is available as `dloess` at https://netlib.org/a/.

References

W. S. Cleveland, E. Grosse and W. M. Shyu (1992) Local regression models. Chapter 8 of *Statistical Models in S* eds J.M. Chambers and T.J. Hastie, Wadsworth & Brooks/Cole.

See Also

`loess.control`, `predict.loess`.

`lowess`, the ancestor of `loess` (with different defaults!).

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed = seq(5, 30, 1)), se = TRUE)
# to allow extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
                  control = loess.control(surface = "direct"))
predict(cars.lo2, data.frame(speed = seq(5, 30, 1)), se = TRUE)
```

`loess.control` *Set Parameters for* `loess`

Description

Set control parameters for `loess` fits.

Usage

```
loess.control(surface = c("interpolate", "direct"),
              statistics = c("approximate", "exact", "none"),
              trace.hat = c("exact", "approximate"),
              cell = 0.2, iterations = 4, iterTrace = FALSE, ...)
```

Arguments

surface	should the fitted surface be computed exactly (`"direct"`) or via interpolation from a k-d tree? Can be abbreviated.
statistics	should the statistics be computed exactly, approximately or not at all? Exact computation can be very slow. Can be abbreviated.
trace.hat	Only for the (default) case (`surface = "interpolate"`, `statistics = "approximate"`): should the trace of the smoother matrix be computed exactly or approximately? It is recommended to use the approximation for more than about 1000 data points. Can be abbreviated.
cell	if interpolation is used this controls the accuracy of the approximation via the maximum number of points in a cell in the k-d tree. Cells with more than `floor(n*span*cell)` points are subdivided.
iterations	the number of iterations used in robust fitting, i.e. only if `family` is `"symmetric"`.
iterTrace	logical (or integer) determining if tracing information during the robust iterations (`iterations` ≥ 2) is produced.
...	further arguments which are ignored.

Value

A list with components

`surface`

`statistics`

`trace.hat`

`cell`

`iterations`

`iterTrace`

with meanings as explained under 'Arguments'.

See Also

loess

Logistic *The Logistic Distribution*

Description

Density, distribution function, quantile function and random generation for the logistic distribution with parameters `location` and `scale`.

Usage

```
dlogis(x, location = 0, scale = 1, log = FALSE)
plogis(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qlogis(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rlogis(n, location = 0, scale = 1)
```

Arguments

`x`, `q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location`, `scale`	location and scale parameters.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are omitted, they assume the default values of `0` and `1` respectively.

The Logistic distribution with `location` $= \mu$ and `scale` $= \sigma$ has distribution function

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

and density

$$f(x) = \frac{1}{\sigma} \frac{e^{(x-\mu)/\sigma}}{(1 + e^{(x-\mu)/\sigma})^2}$$

It is a long-tailed distribution with mean μ and variance $\pi^2/3\sigma^2$.

Value

`dlogis` gives the density, `plogis` gives the distribution function, `qlogis` gives the quantile function, and `rlogis` generates random deviates.

The length of the result is determined by `n` for `rlogis`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

`qlogis(p)` is the same as the well known '*logit*' function, $logit(p) = \log p/(1-p)$, and `plogis(x)` has consequently been called the 'inverse logit'.

The distribution function is a rescaled hyperbolic tangent, `plogis(x) == (1+ tanh(x/2))/2`, and it is called a *sigmoid function* in contexts such as neural networks.

Source

`[dpq]logis` are calculated directly from the definitions.

`rlogis` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapter 23. Wiley, New York.

See Also

Distributions for other standard distributions.

Examples

```
var(rlogis(4000, 0, scale = 5))  # approximately (+/- 3)
pi^2/3 * 5^2
```

---

logLik *Extract Log-Likelihood*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which have methods for this function include: `"glm"`, `"lm"`, `"nls"` and `"Arima"`. Packages contain methods for other classes, such as `"fitdistr"`, `"negbin"` and `"polr"` in package **MASS**, `"multinom"` in package **nnet** and `"gls"`, `"gnls"` `"lme"` and others in package **nlme**.

Usage

```
logLik(object, ...)

## S3 method for class 'lm'
logLik(object, REML = FALSE, ...)
```

Arguments

object	any object from which a log-likelihood value, or a contribution to a log-likelihood value, can be extracted.
...	some methods for this generic function require additional arguments.
REML	an optional logical value. If TRUE the restricted log-likelihood is returned, else, if FALSE, the log-likelihood is returned. Defaults to FALSE.

Details

logLik is most commonly used for a model fitted by maximum likelihood, and some uses, e.g. by AIC, assume this. So care is needed where other fit criteria have been used, for example REML (the default for "lme").

For a "glm" fit the family does not have to specify how to calculate the log-likelihood, so this is based on using the family's aic() function to compute the AIC. For the gaussian, Gamma and inverse.gaussian families it assumed that the dispersion of the GLM is estimated and has been counted as a parameter in the AIC value, and for all other families it is assumed that the dispersion is known. Note that this procedure does not give the maximized likelihood for "glm" fits from the Gamma and inverse gaussian families, as the estimate of dispersion used is not the MLE.

For "lm" fits it is assumed that the scale has been estimated (by maximum likelihood or REML), and all the constants in the log-likelihood are included. That method is only applicable to single-response fits.

Value

Returns an object of class logLik. This is a number with at least one attribute, "df" (**d**egrees of **f**reedom), giving the number of (estimated) parameters in the model.

There is a simple print method for "logLik" objects.

There may be other attributes depending on the method used: see the appropriate documentation. One that is used by several methods is "nobs", the number of observations used in estimation (after the restrictions if REML = TRUE).

Author(s)

José Pinheiro and Douglas Bates

References

For logLik.lm:

Harville, D.A. (1974). Bayesian inference for variance components using only error contrasts. *Biometrika*, **61**, 383–385. doi:10.2307/2334370.

See Also

logLik.gls, logLik.lme, in package **nlme**, etc.

AIC

Examples

```
x <- 1:5
lmx <- lm(x ~ 1)
logLik(lmx) # using print.logLik() method
utils::str(logLik(lmx))

## lm method
(fm1 <- lm(rating ~ ., data = attitude))
logLik(fm1)
logLik(fm1, REML = TRUE)

utils::data(Orthodont, package = "nlme")
fm1 <- lm(distance ~ Sex * age, Orthodont)
logLik(fm1)
logLik(fm1, REML = TRUE)
```

`loglin` *Fitting Log-Linear Models*

Description

`loglin` is used to fit log-linear models to multidimensional contingency tables by Iterative Proportional Fitting.

Usage

```
loglin(table, margin, start = rep(1, length(table)), fit = FALSE,
       eps = 0.1, iter = 20, param = FALSE, print = TRUE)
```

Arguments

`table`	a contingency table to be fit, typically the output from `table`.
`margin`	a list of vectors with the marginal totals to be fit. (Hierarchical) log-linear models can be specified in terms of these marginal totals which give the 'maximal' factor subsets contained in the model. For example, in a three-factor model, `list(c(1, 2), c(1, 3))` specifies a model which contains parameters for the grand mean, each factor, and the 1-2 and 1-3 interactions, respectively (but no 2-3 or 1-2-3 interaction), i.e., a model where factors 2 and 3 are independent conditional on factor 1 (sometimes represented as '[12][13]'). The names of factors (i.e., `names(dimnames(table))`) may be used rather than numeric indices.
`start`	a starting estimate for the fitted table. This optional argument is important for incomplete tables with structural zeros in `table` which should be preserved in the fit. In this case, the corresponding entries in `start` should be zero and the others can be taken as one.
`fit`	a logical indicating whether the fitted values should be returned.

eps	maximum deviation allowed between observed and fitted margins.
iter	maximum number of iterations.
param	a logical indicating whether the parameter values should be returned.
print	a logical. If TRUE, the number of iterations and the final deviation are printed.

Details

The Iterative Proportional Fitting algorithm as presented in Haberman (1972) is used for fitting the model. At most iter iterations are performed, convergence is taken to occur when the maximum deviation between observed and fitted margins is less than eps. All internal computations are done in double precision; there is no limit on the number of factors (the dimension of the table) in the model.

Assuming that there are no structural zeros, both the Likelihood Ratio Test and Pearson test statistics have an asymptotic chi-squared distribution with df degrees of freedom.

Note that the IPF steps are applied to the factors in the order given in margin. Hence if the model is decomposable and the order given in margin is a running intersection property ordering then IPF will converge in one iteration.

Package **MASS** contains loglm, a front-end to loglin which allows the log-linear model to be specified and fitted in a formula-based manner similar to that of other fitting functions such as lm or glm.

Value

A list with the following components.

lrt	the Likelihood Ratio Test statistic.
pearson	the Pearson test statistic (X-squared).
df	the degrees of freedom for the fitted model. There is no adjustment for structural zeros.
margin	list of the margins that were fit. Basically the same as the input margin, but with numbers replaced by names where possible.
fit	An array like table containing the fitted values. Only returned if fit is TRUE.
param	A list containing the estimated parameters of the model. The 'standard' constraints of zero marginal sums (e.g., zero row and column sums for a two factor parameter) are employed. Only returned if param is TRUE.

Author(s)

Kurt Hornik

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Haberman, S. J. (1972). Algorithm AS 51: Log-linear fit for contingency tables. *Applied Statistics*, **21**, 218–225. doi:10.2307/2346506.

Agresti, A. (1990). *Categorical data analysis*. New York: Wiley.

See Also

`table`.

`loglm` in package **MASS** for a user-friendly wrapper.

`glm` for another way to fit log-linear models.

Examples

```
## Model of joint independence of sex from hair and eye color.
fm <- loglin(HairEyeColor, list(c(1, 2), c(1, 3), c(2, 3)))
fm
1 - pchisq(fm$lrt, fm$df)
## Model with no three-factor interactions fits well.
```

Lognormal *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

Usage

```
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)
plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
rlnorm(n, meanlog = 0, sdlog = 1)
```

Arguments

`x`, `q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`meanlog`, `sdlog`	mean and standard deviation of the distribution on the log scale with default values of `0` and `1` respectively.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

The length of the result is determined by `n` for `rlnorm`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

Consequently, they model a single point mass at `exp(meanlog)` for the boundary case `sdlog = 0`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

Distributions for other standard distributions, including `dnorm` for the normal distribution.

Examples

```
dlnorm(1) == dnorm(0)
```

`lowess` *Scatter Plot Smoothing*

Description

This function performs the computations for the *LOWESS* smoother which uses locally-weighted polynomial regression (see the references).

Usage

```
lowess(x, y = NULL, f = 2/3, iter = 3, delta = 0.01 * diff(range(x)))
```

Arguments

`x`, `y`	vectors giving the coordinates of the points in the scatter plot. Alternatively a single plotting structure can be specified – see `xy.coords`.
`f`	the smoother span. This gives the proportion of points in the plot which influence the smooth at each value. Larger values give more smoothness.
`iter`	the number of 'robustifying' iterations which should be performed. Using smaller values of `iter` will make `lowess` run faster.
`delta`	See 'Details'. Defaults to 1/100th of the range of `x`.

Details

`lowess` is defined by a complex algorithm, the Ratfor original of which (by W. S. Cleveland) can be found in the R sources as file '`src/library/stats/src/lowess.doc`'. Normally a local linear polynomial fit is used, but under some circumstances (see the file) a local constant fit can be used. 'Local' is defined by the distance to the `floor(f*n)`-th nearest neighbour, and tricubic weighting is used for `x` which fall within the neighbourhood.

The initial fit is done using weighted least squares. If `iter > 0`, further weighted fits are done using the product of the weights from the proximity of the `x` values and case weights derived from the residuals at the previous iteration. Specifically, the case weight is Tukey's biweight, with cutoff 6 times the MAD of the residuals. (The current R implementation differs from the original in stopping iteration if the MAD is effectively zero since the algorithm is highly unstable in that case.)

`delta` is used to speed up computation: instead of computing the local polynomial fit at each data point it is not computed for points within `delta` of the last computed point, and linear interpolation is used to fill in the fitted values for the skipped points.

Value

`lowess` returns a list containing components `x` and `y` which give the coordinates of the smooth. The smooth can be added to a plot of the original points with the function `lines`: see the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, **74**, 829–836. doi:10.1080/01621459.1979.10481038.

Cleveland, W. S. (1981) LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, **35**, 54. doi:10.2307/2683591.

See Also

`loess`, a newer formula based version of `lowess` (with different defaults!).

Examples

```
require(graphics)

plot(cars, main = "lowess(cars)")
lines(lowess(cars), col = 2)
lines(lowess(cars, f = .2), col = 3)
legend(5, 120, c(paste("f = ", c("2/3", ".2"))), lty = 1, col = 2:3)
```

`ls.diag` *Compute Diagnostics for* `lsfit` *Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients.

Usage

```
ls.diag(ls.out)
```

Arguments

`ls.out`	Typically the result of `lsfit()`

Value

A `list` with the following numeric components.

`std.dev`	The standard deviation of the errors, an estimate of σ.
`hat`	diagonal entries h_{ii} of the hat matrix H
`std.res`	standardized residuals
`stud.res`	studentized residuals
`cooks`	Cook's distances
`dfits`	DFITS statistics

correlation	correlation matrix
std.err	standard errors of the regression coefficients
cov.scaled	Scaled covariance matrix of the coefficients
cov.unscaled	Unscaled covariance matrix of the coefficients

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

See Also

hat for the hat matrix diagonals, ls.print, lm.influence, summary.lm, anova.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = as.numeric(gl(2, 10, 20)), y = weight)
dlsD9 <- ls.diag(lsD9)
utils::str(dlsD9, give.attr = FALSE)
abs(1 - sum(dlsD9$hat) / 2) < 10*.Machine$double.eps # sum(h.ii) = p
plot(dlsD9$hat, dlsD9$stud.res, xlim = c(0, 0.11))
abline(h = 0, lty = 2, col = "lightgray")
```

---

ls.print *Print* lsfit *Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients and prints them if print.it is TRUE.

Usage

```
ls.print(ls.out, digits = 4, print.it = TRUE)
```

Arguments

ls.out	Typically the result of lsfit()
digits	The number of significant digits used for printing
print.it	a logical indicating whether the result should also be printed

Value

A list with the components

summary	The ANOVA table of the regression
coef.table	matrix with regression coefficients, standard errors, t- and p-values

Note

Usually you would use `summary(lm(...))` and `anova(lm(...))` to obtain similar output.

See Also

`ls.diag`, `lsfit`, also for examples; `lm`, `lm.influence` which usually are preferable.

`lsfit` *Find the Least Squares Fit*

Description

The least squares estimate of β in the model

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

is found.

Usage

```
lsfit(x, y, wt = NULL, intercept = TRUE, tolerance = 1e-07,
      yname = NULL)
```

Arguments

`x`	a matrix whose rows correspond to cases and whose columns correspond to variables.
`y`	the responses, possibly a matrix if you want to fit multiple left hand sides.
`wt`	an optional vector of weights for performing weighted least squares.
`intercept`	whether or not an intercept term should be used.
`tolerance`	the tolerance to be used in the matrix decomposition.
`yname`	names to be used for the response variables.

Details

If weights are specified then a weighted least squares is performed with the weight given to the *j*-th case specified by the *j*-th entry in `wt`.

If any observation has a missing value in any field, that observation is removed before the analysis is carried out. This can be quite inefficient if there is a lot of missing data.

The implementation is via a modification of the LINPACK subroutines which allow for multiple left-hand sides.

Value

A list with the following named components:

coef	the least squares estimates of the coefficients in the model (β as stated above).
residuals	residuals from the fit.
intercept	indicates whether an intercept was fitted.
qr	the QR decomposition of the design matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

lm which usually is preferable; ls.print, ls.diag.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = unclass(gl(2, 10)), y = weight)
ls.print(lsD9)
```

---

mad *Median Absolute Deviation*

Description

Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.

Usage

```
mad(x, center = median(x), constant = 1.4826, na.rm = FALSE,
    low = FALSE, high = FALSE)
```

Arguments

x	a numeric vector.
center	Optionally, the centre: defaults to the median.
constant	scale factor.
na.rm	if TRUE then NA values are stripped from x before computation takes place.
low	if TRUE, compute the 'lo-median', i.e., for even sample size, do not average the two middle values, but take the smaller one.
high	if TRUE, compute the 'hi-median', i.e., take the larger of the two middle values for even sample size.

Details

The actual value calculated is `constant * cMedian(abs(x - center))` with the default value of `center` being `median(x)`, and `cMedian` being the usual, the 'low' or 'high' median, see the arguments description for `low` and `high` above.

In the case of $n = 1$ non-missing values and default `center`, the result is `0`, consistent with "no deviation from the center".

The default `constant = 1.4826` (approximately $1/\Phi^{-1}(\frac{3}{4})$ = `1/qnorm(3/4)`) ensures consistency, i.e.,

$$E[mad(X_1, \ldots, X_n)] = \sigma$$

for X_i distributed as $N(\mu, \sigma^2)$ and large n.

If `na.rm` is `TRUE` then `NA` values are stripped from `x` before computation takes place. If this is not done then an `NA` value in `x` will cause `mad` to return `NA`.

See Also

`IQR` which is simpler but less robust, `median`, `var`.

Examples

```
mad(c(1:9))
print(mad(c(1:9),     constant = 1)) ==
      mad(c(1:8, 100), constant = 1)       # = 2 ; TRUE
x <- c(1,2,3,5,7,8)
sort(abs(x - median(x)))
c(mad(x, constant = 1),
  mad(x, constant = 1, low = TRUE),
  mad(x, constant = 1, high = TRUE))
```

`mahalanobis` *Mahalanobis Distance*

Description

Returns the squared Mahalanobis distance of all rows in `x` and the vector μ = `center` with respect to Σ = `cov`. This is (for vector `x`) defined as

$$D^2 = (x - \mu)' \Sigma^{-1} (x - \mu)$$

Usage

```
mahalanobis(x, center, cov, inverted = FALSE, ...)
```

Arguments

x	vector or matrix of data with, say, p columns.
center	mean vector of the distribution or second data vector of length p or recyclable to that length. If set to FALSE, the centering step is skipped.
cov	covariance matrix ($p \times p$) of the distribution.
inverted	logical. If TRUE, cov is supposed to contain the *inverse* of the covariance matrix.
...	passed to solve for computing the inverse of the covariance matrix (if inverted is false).

See Also

cov, var

Examples

```
require(graphics)

ma <- cbind(1:6, 1:3)
(S <-  var(ma))
mahalanobis(c(0, 0), 1:2, S)

x <- matrix(rnorm(100*3), ncol = 3)
stopifnot(mahalanobis(x, 0, diag(ncol(x))) == rowSums(x*x))
        ##- Here, D^2 = usual squared Euclidean distances

Sx <- cov(x)
D2 <- mahalanobis(x, colMeans(x), Sx)
plot(density(D2, bw = 0.5),
     main="Squared Mahalanobis distances, n=100, p=3") ; rug(D2)
qqplot(qchisq(ppoints(100), df = 3), D2,
       main = expression("Q-Q plot of Mahalanobis" * ~D^2 *
                         " vs. quantiles of" * ~ chi[3]^2))
abline(0, 1, col = 'gray')
```

---

make.link *Create a Link for GLM Families*

Description

This function is used with the family functions in glm(). Given the name of a link, it returns a link function, an inverse link function, the derivative $d\mu/d\eta$ and a function for domain checking.

Usage

```
make.link(link)
```

Arguments

link	character; one of "logit", "probit", "cauchit", "cloglog", "identity", "log", "sqrt", "1/mu^2", "inverse".

Value

A object of class "link-glm", a list with components

linkfun	Link function function(mu)
linkinv	Inverse link function function(eta)
mu.eta	Derivative function(eta) $d\mu/d\eta$
valideta	function(eta){ TRUE if eta is in the domain of linkinv }.
name	a name to be used for the link

.

See Also

power, glm, family.

Examples

```
utils::str(make.link("logit"))
```

---

makepredictcall *Utility Function for Safe Prediction*

Description

A utility to help model.frame.default create the right matrices when predicting from models with terms like (univariate) poly or ns.

Usage

```
makepredictcall(var, call)
```

Arguments

var	A variable.
call	The term in the formula, as a call.

Details

This is a generic function with methods for poly, bs and ns: the default method handles scale. If model.frame.default encounters such a term when creating a model frame, it modifies the predvars attribute of the terms supplied by replacing the term with one which will work for predicting new data. For example makepredictcall.ns adds arguments for the knots and intercept.

To make use of this, have your model-fitting function return the terms attribute of the model frame, or copy the predvars attribute of the terms attribute of the model frame to your terms object.

To extend this, make sure the term creates variables with a class, and write a suitable method for that class.

Value

A replacement for call for the predvars attribute of the terms.

See Also

model.frame, poly, scale; bs and ns in package **splines**.

cars for an example of prediction from a polynomial fit.

Examples

```
require(graphics)

## using poly: this did not work in R < 1.5.0
fm <- lm(weight ~ poly(height, 2), data = women)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, length.out = 200)
nD <- data.frame(height = ht)
pfm <- predict(fm, nD)
lines(ht, pfm)
pf2 <- predict(update(fm, ~ stats::poly(height, 2)), nD)
stopifnot(all.equal(pfm, pf2)) ## was off (rel.diff. 0.0766) in R <= 3.5.0

## see also example(cars)

## see bs and ns for spline examples.
```

---

manova *Multivariate Analysis of Variance*

Description

A class for the multivariate analysis of variance.

Usage

```
manova(...)
```

Arguments

...	Arguments to be passed to aov.

Details

Class "manova" differs from class "aov" in selecting a different summary method. Function manova calls aov and then add class "manova" to the result object for each stratum.

Value

See aov and the comments in 'Details' here.

References

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

aov, summary.manova, the latter containing more examples.

Examples

```
## Set orthogonal contrasts.
op <- options(contrasts = c("contr.helmert", "contr.poly"))

## Fake a 2nd response variable
npk2 <- within(npk, foo <- rnorm(24))
( npk2.aov <- manova(cbind(yield, foo) ~ block + N*P*K, npk2) )
summary(npk2.aov)

( npk2.aovE <- manova(cbind(yield, foo) ~  N*P*K + Error(block), npk2) )
summary(npk2.aovE)
```

---

mantelhaen.test *Cochran-Mantel-Haenszel Chi-Squared Test for Count Data*

Description

Performs a Cochran-Mantel-Haenszel chi-squared test of the null that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction.

Usage

```
mantelhaen.test(x, y = NULL, z = NULL,
                alternative = c("two.sided", "less", "greater"),
                correct = TRUE, exact = FALSE, conf.level = 0.95)
```

Arguments

x	either a 3-dimensional contingency table in array form where each dimension is at least 2 and the last dimension corresponds to the strata, or a factor object with at least 2 levels.
y	a factor object with at least 2 levels; ignored if x is an array.
z	a factor object with at least 2 levels identifying to which stratum the corresponding elements in x and y belong; ignored if x is an array.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter. Only used in the 2 by 2 by K case.
correct	a logical indicating whether to apply continuity correction when computing the test statistic. Only used in the 2 by 2 by K case.
exact	a logical indicating whether the Mantel-Haenszel test or the exact conditional test (given the strata margins) should be computed. Only used in the 2 by 2 by K case.
conf.level	confidence level for the returned confidence interval. Only used in the 2 by 2 by K case.

Details

If x is an array, each dimension must be at least 2, and the entries should be nonnegative integers. NA's are not allowed. Otherwise, x, y and z must have the same length. Triples containing NA's are removed. All variables must take at least two different values.

Value

A list with class "htest" containing the following components:

statistic	Only present if no exact test is performed. In the classical case of a 2 by 2 by K table (i.e., of dichotomous underlying variables), the Mantel-Haenszel chi-squared statistic; otherwise, the generalized Cochran-Mantel-Haenszel statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic (1 in the classical case). Only present if no exact test is performed.
p.value	the p-value of the test.
conf.int	a confidence interval for the common odds ratio. Only present in the 2 by 2 by K case.
estimate	an estimate of the common odds ratio. If an exact test is performed, the conditional Maximum Likelihood Estimate is given; otherwise, the Mantel-Haenszel estimate. Only present in the 2 by 2 by K case.
null.value	the common odds ratio under the null of independence, 1. Only present in the 2 by 2 by K case.
alternative	a character string describing the alternative hypothesis. Only present in the 2 by 2 by K case.
method	a character string indicating the method employed, and whether or not continuity correction was used.
data.name	a character string giving the names of the data.

Note

The asymptotic distribution is only valid if there is no three-way interaction. In the classical 2 by 2 by K case, this is equivalent to the conditional odds ratios in each stratum being identical. Currently, no inference on homogeneity of the odds ratios is performed.

See also the example below.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 230–235.

Alan Agresti (2002). *Categorical data analysis* (second edition). New York: Wiley.

Examples

```
## Agresti (1990), pages 231--237, Penicillin and Rabbits
## Investigation of the effectiveness of immediately injected or 1.5
##  hours delayed penicillin in protecting rabbits against a lethal
##  injection with beta-hemolytic streptococci.
Rabbits <-
array(c(0, 0, 6, 5,
        3, 0, 3, 6,
        6, 2, 0, 4,
        5, 6, 1, 0,
        2, 5, 0, 0),
      dim = c(2, 2, 5),
      dimnames = list(
          Delay = c("None", "1.5h"),
          Response = c("Cured", "Died"),
          Penicillin.Level = c("1/8", "1/4", "1/2", "1", "4")))
Rabbits
## Classical Mantel-Haenszel test
mantelhaen.test(Rabbits)
## => p = 0.047, some evidence for higher cure rate of immediate
##               injection
## Exact conditional test
mantelhaen.test(Rabbits, exact = TRUE)
## => p - 0.040
## Exact conditional test for one-sided alternative of a higher
## cure rate for immediate injection
mantelhaen.test(Rabbits, exact = TRUE, alternative = "greater")
## => p = 0.020

## UC Berkeley Student Admissions
mantelhaen.test(UCBAdmissions)
## No evidence for association between admission and gender
## when adjusted for department.  However,
apply(UCBAdmissions, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
## This suggests that the assumption of homogeneous (conditional)
## odds ratios may be violated.  The traditional approach would be
## using the Woolf test for interaction:
woolf <- function(x) {
  x <- x + 1 / 2
```

```
  k <- dim(x)[3]
  or <- apply(x, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
  w <-  apply(x, 3, function(x) 1 / sum(1 / x))
  1 - pchisq(sum(w * (log(or) - weighted.mean(log(or), w)) ^ 2), k - 1)
}
woolf(UCBAdmissions)
## => p = 0.003, indicating that there is significant heterogeneity.
## (And hence the Mantel-Haenszel test cannot be used.)

## Agresti (2002), p. 287f and p. 297.
## Job Satisfaction example.
Satisfaction <-
    as.table(array(c(1, 2, 0, 0, 3, 3, 1, 2,
                     11, 17, 8, 4, 2, 3, 5, 2,
                     1, 0, 0, 0, 1, 3, 0, 1,
                     2, 5, 7, 9, 1, 1, 3, 6),
                   dim = c(4, 4, 2),
                   dimnames =
                   list(Income =
                        c("<5000", "5000-15000",
                          "15000-25000", ">25000"),
                        "Job Satisfaction" =
                        c("V_D", "L_S", "M_S", "V_S"),
                        Gender = c("Female", "Male"))))
## (Satisfaction categories abbreviated for convenience.)
ftable(. ~ Gender + Income, Satisfaction)
## Table 7.8 in Agresti (2002), p. 288.
mantelhaen.test(Satisfaction)
## See Table 7.12 in Agresti (2002), p. 297.
```

`mauchly.test` *Mauchly's Test of Sphericity*

Description

Tests whether a Wishart-distributed covariance matrix (or transformation thereof) is proportional to a given matrix.

Usage

```
mauchly.test(object, ...)
## S3 method for class 'mlm'
mauchly.test(object, ...)
## S3 method for class 'SSD'
mauchly.test(object, Sigma = diag(nrow = p),
   T = Thin.row(Proj(M) - Proj(X)), M = diag(nrow = p), X = ~0,
   idata = data.frame(index = seq_len(p)), ...)
```

Arguments

object	object of class SSD or mlm.
Sigma	matrix to be proportional to.
T	transformation matrix. By default computed from M and X.
M	formula or matrix describing the outer projection (see below).
X	formula or matrix describing the inner projection (see below).
idata	data frame describing intra-block design.
...	arguments to be passed to or from other methods.

Details

This is a generic function with methods for classes "mlm" and "SSD".

The basic method is for objects of class SSD the method for mlm objects just extracts the SSD matrix and invokes the corresponding method with the same options and arguments.

The T argument is used to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A matrix T can be given directly or specified as the difference between two projections onto the spaces spanned by M and X, which in turn can be given as matrices or as model formulas with respect to idata (the tests will be invariant to parametrization of the quotient space M/X).

The common use of this test is in repeated measurements designs, with X = ~1. This is almost, but not quite the same as testing for compound symmetry in the untransformed covariance matrix.

Notice that the defaults involve p, which is calculated internally as the dimension of the SSD matrix, and a couple of hidden functions in the **stats** namespace, namely proj which calculates projection matrices from design matrices or model formulas and Thin.row which removes linearly dependent rows from a matrix until it has full row rank.

Value

An object of class "htest"

Note

The p-value differs slightly from that of SAS because a second order term is included in the asymptotic approximation in R.

References

T. W. Anderson (1958). *An Introduction to Multivariate Statistical Analysis.* Wiley.

See Also

SSD, anova.mlm, rWishart

Examples

```
utils::example(SSD) # Brings in the mlmfit and reacttime objects

### traditional test of intrasubj. contrasts
mauchly.test(mlmfit, X = ~1)

### tests using intra-subject 3x2 design
idata <- data.frame(deg = gl(3, 1, 6, labels = c(0,4,8)),
                    noise = gl(2, 3, 6, labels = c("A","P")))
mauchly.test(mlmfit, X = ~ deg + noise, idata = idata)
mauchly.test(mlmfit, M = ~ deg + noise, X = ~ noise, idata = idata)
```

---

mcnemar.test *McNemar's Chi-squared Test for Count Data*

Description

Performs McNemar's chi-squared test for symmetry of rows and columns in a two-dimensional contingency table.

Usage

```
mcnemar.test(x, y = NULL, correct = TRUE)
```

Arguments

x	either a two-dimensional contingency table in matrix form, or a factor object.
y	a factor object; ignored if x is a matrix.
correct	a logical indicating whether to apply continuity correction when computing the test statistic.

Details

The null is that the probabilities of being classified into cells `[i,j]` and `[j,i]` are the same.

If x is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both x and y must be vectors or factors of the same length. Incomplete cases are removed, vectors are coerced into factors, and the contingency table is computed from these.

Continuity correction is only used in the 2-by-2 case if `correct` is `TRUE`.

Value

A list with class `"htest"` containing the following components:

statistic	the value of McNemar's statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.

p.value	the p-value of the test.
method	a character string indicating the type of test performed, and whether continuity correction was used.
data.name	a character string giving the name(s) of the data.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 350–354.

Examples

```
## Agresti (1990), p. 350.
## Presidential Approval Ratings.
##  Approval of the President's performance in office in two surveys,
##  one month apart, for a random sample of 1600 voting-age Americans.
Performance <-
matrix(c(794, 86, 150, 570),
       nrow = 2,
       dimnames = list("1st Survey" = c("Approve", "Disapprove"),
                       "2nd Survey" = c("Approve", "Disapprove")))
Performance
mcnemar.test(Performance)
## => significant change (in fact, drop) in approval ratings
```

---

median *Median Value*

Description

Compute the sample median.

Usage

```
median(x, na.rm = FALSE, ...)
## Default S3 method:
median(x, na.rm = FALSE, ...)
```

Arguments

x	an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	potentially further arguments for methods; not used in the default method.

Details

This is a generic function for which methods can be written. However, the default method makes use of `is.na`, `sort` and `mean` from package **base** all of which are generic, and so the default method will work for most classes (e.g., "Date") for which a median is a reasonable concept.

Value

The default method returns a length-one object of the same type as `x`, except when `x` is logical or integer of even length, when the result will be double.

If there are no values or if `na.rm = FALSE` and there are `NA` values the result is `NA` of the same type as `x` (or more generally the result of `x[NA_integer_]`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`quantile` for general quantiles.

Examples

```
median(1:4)               # = 2.5 [even number]
median(c(1:3, 100, 1000))  # = 3 [odd, robust]
```

---

medpolish *Median Polish (Robust Two-way Decomposition) of a Matrix*

Description

Fits an additive model (two-way decomposition) using Tukey's *median polish* procedure.

Usage

```
medpolish(x, eps = 0.01, maxiter = 10, trace.iter = TRUE,
          na.rm = FALSE)
```

Arguments

`x`	a numeric matrix.
`eps`	real number greater than 0. A tolerance for convergence: see 'Details'.
`maxiter`	the maximum number of iterations
`trace.iter`	logical. Should progress in convergence be reported?
`na.rm`	logical. Should missing values be removed?

Details

The model fitted is additive (constant + rows + columns). The algorithm works by alternately removing the row and column medians, and continues until the proportional reduction in the sum of absolute residuals is less than `eps` or until there have been `maxiter` iterations. The sum of absolute residuals is printed at each iteration of the fitting process, if `trace.iter` is `TRUE`. If `na.rm` is `FALSE` the presence of any `NA` value in `x` will cause an error, otherwise `NA` values are ignored.

`medpolish` returns an object of class `medpolish` (see below). There are printing and plotting methods for this class, which are invoked via by the generics `print` and `plot`.

Value

An object of class `medpolish` with the following named components:

`overall`	the fitted constant term.
`row`	the fitted row effects.
`col`	the fitted column effects.
`residuals`	the residuals.
`name`	the name of the dataset.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

`median`; `aov` for a *mean* instead of *median* decomposition.

Examples

```
require(graphics)

## Deaths from sport parachuting;  from ABC of EDA, p.224:
deaths <-
    rbind(c(14,15,14),
          c( 7, 4, 7),
          c( 8, 2,10),
          c(15, 9,10),
          c( 0, 2, 0))
dimnames(deaths) <- list(c("1-24", "25-74", "75-199", "200++", "NA"),
                         paste(1973:1975))
deaths
(med.d <- medpolish(deaths))
plot(med.d)
## Check decomposition:
all(deaths ==
    med.d$overall + outer(med.d$row,med.d$col, `+`) + med.d$residuals)
```

model.extract *Extract Components from a Model Frame*

Description

Returns the response, offset, subset, weights or other special components of a model frame passed as optional arguments to model.frame.

Usage

```
model.extract(frame, component)
model.offset(x)
model.response(data, type = "any")
model.weights(x)
```

Arguments

frame, x, data	a model frame, see model.frame.
component	literal character string or name. The name of a component to extract, such as "weights" or "subset".
type	One of "any", "numeric" or "double". Using either of latter two coerces the result to have storage mode "double".

Details

model.extract is provided for compatibility with S, which does not have the more specific functions. It is also useful to extract e.g. the etastart and mustart components of a glm fit.

model.extract(m, "offset") and model.extract(m, "response") are equivalent to model.offset(m) and model.response(m) respectively. model.offset sums any terms specified by offset terms in the formula or by offset arguments in the call producing the model frame: it does check that the offset is numeric.

model.weights is slightly different from model.extract(, "weights") in not naming the vector it returns.

Value

The specified component of the model frame, usually a vector. model.response() now *drops* a possible "Asis" class (stemming from I(.)).

model.offset returns NULL if no offset was specified.

See Also

model.frame, offset

Examples

```
a <- model.frame(cbind(ncases,ncontrols) ~ agegp + tobgp + alcgp, data = esoph)
model.extract(a, "response")
stopifnot(model.extract(a, "response") == model.response(a))

a <- model.frame(ncases/(ncases+ncontrols) ~ agegp + tobgp + alcgp,
                 data = esoph, weights = ncases+ncontrols)
model.response(a)
(mw <- model.extract(a, "weights"))
stopifnot(identical(unname(mw), model.weights(a)))

a <- model.frame(cbind(ncases,ncontrols) ~ agegp,
                 something = tobgp, data = esoph)
names(a)
stopifnot(model.extract(a, "something") == esoph$tobgp)
```

---

model.frame *Extracting the Model Frame from a Formula or Fit*

Description

`model.frame` (a generic function) and its methods return a `data.frame` with the variables needed to use `formula` and any `...` arguments.

Usage

```
model.frame(formula, ...)

## Default S3 method:
model.frame(formula, data = NULL,
            subset = NULL, na.action,
            drop.unused.levels = FALSE, xlev = NULL, ...)

## S3 method for class 'aovlist'
model.frame(formula, data = NULL, ...)

## S3 method for class 'glm'
model.frame(formula, ...)

## S3 method for class 'lm'
model.frame(formula, ...)

get_all_vars(formula, data, ...)
```

Arguments

`formula`	a model `formula` or `terms` object or an R object.

`data` a data frame, list or environment (or object coercible by `as.data.frame` to a data frame), containing the variables in `formula`. Neither a matrix nor an array will be accepted.

`subset` a specification of the rows/observations to be used: defaults to all. This can be any valid indexing vector (see `[.data.frame`) for the rows of `data`, or a (logical) expression using variables in `data` or if that is not supplied, in `formula`. (See additional details about how this argument interacts with data-dependent bases under 'Details' below.)

`na.action` an optional (name of a) function for treating missing values (`NA`s). The default is first, any `na.action` attribute of `data`, second a `na.action` setting of `options`, and third `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`.

`drop.unused.levels`
should factors have unused levels dropped? Defaults to `FALSE`.

`xlev` a named list of character vectors giving the full set of levels to be assumed for each factor.

`...` for `model.frame` methods, a mix of further arguments such as `data`, `na.action`, `subset` to pass to the default method. Any additional arguments (such as `offset` and `weights` or other named arguments) which reach the default method are used to create further columns in the model frame, with parenthesised names such as `"(offset)"`.

For `get_all_vars`, further named columns to include in the model frame.

Details

Exactly what happens depends on the class and attributes of the object `formula`. If this is an object of fitted-model class such as `"lm"`, the method will either return the saved model frame used when fitting the model (if any, often selected by argument `model = TRUE`) or pass the call used when fitting on to the default method. The default method itself can cope with rather standard model objects such as those of class `"lqs"` from package **MASS** if no other arguments are supplied.

The rest of this section applies only to the default method.

If either `formula` or `data` is already a model frame (a data frame with a `"terms"` attribute) and the other is missing, the model frame is returned. Unless `formula` is a terms object, `as.formula` and then `terms` is called on it. (If you wish to use the `keep.order` argument of `terms.formula`, pass a terms object rather than a formula.)

Row names for the model frame are taken from the `data` argument if present, then from the names of the response in the formula (or rownames if it is a matrix), if there is one.

All the variables in `formula`, `subset` and in `...` are looked for first in `data` and then in the environment of `formula` (see the help for `formula()` for further details) and collected into a data frame. Then the `subset` expression is evaluated, and it is used as a row index to the data frame. Then the `na.action` function is applied to the data frame (and may well add attributes). The levels of any factors in the data frame are adjusted according to the `drop.unused.levels` and `xlev` arguments: if `xlev` specifies a factor and a character variable is found, it is converted to a factor (as from R 2.10.0).

Because variables in the formula are evaluated before rows are dropped based on subset, the characteristics of data-dependent bases such as orthogonal polynomials (i.e. from terms using poly) or splines will be computed based on the full data set rather than the subsetted one.

Unless na.action = NULL, time-series attributes will be removed from the variables found (since they will be wrong if NAs are removed).

Note that *all* the variables in the formula are included in the data frame, even those preceded by -.

Only variables whose type is raw, logical, integer, real, complex or character can be included in a model frame: this includes classed variables such as factors (whose underlying type is integer), but excludes lists.

get_all_vars returns a data.frame containing the variables used in formula plus those specified in ... which are recycled to the number of data frame rows. Unlike model.frame.default, it returns the input variables and not those resulting from function calls in formula.

Value

A data.frame containing the variables used in formula plus those specified in .... It will have additional attributes, including "terms" for an object of class "terms" derived from formula, and possibly "na.action" giving information on the handling of NAs (which will not be present if no special handling was done, e.g. by na.pass).

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

model.matrix for the 'design matrix', formula for formulas, model.extract to extract components, and expand.model.frame for model.frame manipulation.

Examples

```
data.class(model.frame(dist ~ speed, data = cars))

## using a subset and an extra variable
model.frame(dist ~ speed, data = cars, subset = speed < 10, z = log(dist))

## get_all_vars(): new var.s are recycled (iff length matches: 50 = 2*25)
ncars <- get_all_vars(sqrt(dist) ~ I(speed/2), data = cars, newVar = 2:3)
stopifnot(is.data.frame(ncars),
          identical(cars, ncars[,names(cars)]),
          ncol(ncars) == ncol(cars) + 1)
```

model.matrix *Construct Design Matrices*

Description

model.matrix creates a design (or model) matrix, e.g., by expanding factors to a set of dummy variables (depending on the contrasts) and expanding interactions similarly.

Usage

```
model.matrix(object, ...)

## Default S3 method:
model.matrix(object, data = environment(object),
             contrasts.arg = NULL, xlev = NULL, ...)
## S3 method for class 'lm'
model.matrix(object, ...)
```

Arguments

object	an object of an appropriate class. For the default method, a model formula or a terms object.
data	a data frame created with model.frame. If another sort of object, model.frame is called first.
contrasts.arg	a list, whose entries are values (numeric matrices, functions or character strings naming functions) to be used as replacement values for the contrasts replacement function and whose names are the names of columns of data containing factors.
xlev	to be used as argument of model.frame if data is such that model.frame is called.
...	further arguments passed to or from other methods.

Details

model.matrix creates a design matrix from the description given in terms(object), using the data in data which must supply variables with the same names as would be created by a call to model.frame(object) or, more precisely, by evaluating attr(terms(object), "variables"). If data is a data frame, there may be other columns and the order of columns is not important. Any character variables are coerced to factors. After coercion, all the variables used on the right-hand side of the formula must be logical, integer, numeric or factor.

If contrasts.arg is specified for a factor it overrides the default factor coding for that variable and any "contrasts" attribute set by C or contrasts. Whereas invalid contrasts.args have been ignored always, they are warned about since R version 3.6.0.

In an interaction term, the variable whose levels vary fastest is the first one to appear in the formula (and not in the term), so in ~ a + b + b:a the interaction will have a varying fastest.

By convention, if the response variable also appears on the right-hand side of the formula it is dropped (with a warning), although interactions involving the term are retained.

Value

The design matrix for a regression-like model with the specified formula and data.

There is an attribute `"assign"`, an integer vector with an entry for each column in the matrix giving the term in the formula which gave rise to the column. Value `0` corresponds to the intercept (if any), and positive values to terms in the order given by the `term.labels` attribute of the `terms` structure corresponding to `object`.

If there are any factors in terms in the model, there is an attribute `"contrasts"`, a named list with an entry for each factor. This specifies the contrasts that would be used in terms in which the factor is coded by contrasts (in some terms dummy coding may be used), either as a character vector naming a function or as a numeric matrix.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`model.frame`, `model.extract`, `terms`

`sparse.model.matrix` from package **Matrix** for creating *sparse* model matrices, which may be more efficient in large dimensions.

Examples

```
ff <- log(Volume) ~ log(Height) + log(Girth)
utils::str(m <- model.frame(ff, trees))
mat <- model.matrix(ff, m)

dd <- data.frame(a = gl(3,4), b = gl(4,1,12)) # balanced 2-way
options("contrasts") # typically 'treatment' (for unordered factors)
model.matrix(~ a + b, dd)
model.matrix(~ a + b, dd, contrasts.arg = list(a = "contr.sum"))
model.matrix(~ a + b, dd, contrasts.arg = list(a = "contr.sum", b = contr.poly))
m.orth <- model.matrix(~a+b, dd, contrasts.arg = list(a = "contr.helmert"))
crossprod(m.orth) # m.orth is  ALMOST  orthogonal
# invalid contrasts.. ignored with a warning:
stopifnot(identical(
   model.matrix(~ a + b, dd),
   model.matrix(~ a + b, dd, contrasts.arg = "contr.FOO")))
```

model.tables *Compute Tables of Results from an* aov *Model Fit*

Description

Computes summary tables for model fits, especially complex aov fits.

Usage

```
model.tables(x, ...)

## S3 method for class 'aov'
model.tables(x, type = "effects", se = FALSE, cterms, ...)

## S3 method for class 'aovlist'
model.tables(x, type = "effects", se = FALSE, ...)
```

Arguments

x	a model object, usually produced by aov
type	type of table: currently only "effects" and "means" are implemented. Can be abbreviated.
se	should standard errors be computed?
cterms	A character vector giving the names of the terms for which tables should be computed. The default is all tables.
...	further arguments passed to or from other methods.

Details

For type = "effects" give tables of the coefficients for each term, optionally with standard errors.

For type = "means" give tables of the mean response for each combinations of levels of the factors in a term.

The "aov" method cannot be applied to components of a "aovlist" fit.

Value

An object of class "tables.aov", as list which may contain components

tables	A list of tables for each requested term.
n	The replication information for each term.
se	Standard error information.

Warning

The implementation is incomplete, and only the simpler cases have been tested thoroughly.

Weighted aov fits are not supported.

See Also

aov, proj, replications, TukeyHSD, se.contrast

Examples

```
options(contrasts = c("contr.helmert", "contr.treatment"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
model.tables(npk.aov, "means", se = TRUE)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
model.tables(npk.aovE, se = TRUE)
model.tables(npk.aovE, "means")
```

---

monthplot *Plot a Seasonal or other Subseries from a Time Series*

Description

These functions plot seasonal (or other) subseries of a time series. For each season (or other category), a time series is plotted.

Usage

```
monthplot(x, ...)

## S3 method for class 'stl'
monthplot(x, labels = NULL, ylab = choice, choice = "seasonal",
          ...)

## S3 method for class 'StructTS'
monthplot(x, labels = NULL, ylab = choice, choice = "sea", ...)

## S3 method for class 'ts'
monthplot(x, labels = NULL, times = time(x), phase = cycle(x),
             ylab = deparse1(substitute(x)), ...)

## Default S3 method:
monthplot(x, labels = 1L:12L,
          ylab = deparse1(substitute(x)),
          times = seq_along(x),
          phase = (times - 1L)%%length(labels) + 1L, base = mean,
          axes = TRUE, type = c("l", "h"), box = TRUE,
          add = FALSE,
          col = par("col"), lty = par("lty"), lwd = par("lwd"),
          col.base = col, lty.base = lty, lwd.base = lwd, ...)
```

Arguments

x	Time series or related object.
labels	Labels to use for each 'season'.
ylab	y label.
times	Time of each observation.
phase	Indicator for each 'season'.
base	Function to use for reference line for subseries.
choice	Which series of an stl or StructTS object?
...	Arguments to be passed to the default method or graphical parameters.
axes	Should axes be drawn (ignored if add = TRUE)?
type	Type of plot. The default is to join the points with lines, and "h" is for histogram-like vertical lines.
box	Should a box be drawn (ignored if add = TRUE)?
add	Should thus just add on an existing plot.
col, lty, lwd	Graphics parameters for the series.
col.base, lty.base, lwd.base	Graphics parameters for the segments used for the reference lines.

Details

These functions extract subseries from a time series and plot them all in one frame. The ts, stl, and StructTS methods use the internally recorded frequency and start and finish times to set the scale and the seasons. The default method assumes observations come in groups of 12 (though this can be changed).

If the labels are not given but the phase is given, then the labels default to the unique values of the phase. If both are given, then the phase values are assumed to be indices into the labels array, i.e., they should be in the range from 1 to length(labels).

Value

These functions are executed for their side effect of drawing a seasonal subseries plot on the current graphical window.

Author(s)

Duncan Murdoch

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ts, stl, StructTS

Examples

```
require(graphics)

## The CO2 data
fit <- stl(log(co2), s.window = 20, t.window = 20)
plot(fit)
op <- par(mfrow = c(2,2))
monthplot(co2, ylab = "data", cex.axis = 0.8)
monthplot(fit, choice = "seasonal", cex.axis = 0.8)
monthplot(fit, choice = "trend", cex.axis = 0.8)
monthplot(fit, choice = "remainder", type = "h", cex.axis = 0.8)
par(op)

## The CO2 data, grouped quarterly
quarter <- (cycle(co2) - 1) %/% 3
monthplot(co2, phase = quarter)

## see also JohnsonJohnson
```

---

mood.test *Mood Two-Sample Test of Scale*

Description

Performs Mood's two-sample test for a difference in scale parameters.

Usage

```
mood.test(x, ...)

## Default S3 method:
mood.test(x, y,
          alternative = c("two.sided", "less", "greater"), ...)

## S3 method for class 'formula'
mood.test(formula, data, subset, na.action, ...)
```

Arguments

x, y	numeric vectors of data values.
alternative	indicates the alternative hypothesis and must be one of "two.sided" (default), "greater" or "less" all of which can be abbreviated.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).

subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The underlying model is that the two samples are drawn from $f(x - l)$ and $f((x - l)/s)/s$, respectively, where l is a common location parameter and s is a scale parameter.

The null hypothesis is $s = 1$.

There are more useful tests for this problem.

In the case of ties, the formulation of Mielke (1967) is employed.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
p.value	the p-value of the test.
alternative	a character string describing the alternative hypothesis. You can specify just the initial letter.
method	the character string "Mood two-sample test of scale".
data.name	a character string giving the names of the data.

References

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 234f.

Paul W. Mielke, Jr. (1967). Note on some squared rank tests with existing ties. *Technometrics*, **9**/2, 312–314. doi:10.2307/1266427.

See Also

fligner.test for a rank-based (nonparametric) k-sample test for homogeneity of variances; ansari.test for another rank-based two-sample test for a difference in scale parameters; var.test and bartlett.test for parametric tests for the homogeneity in variance.

Examples

```
## Same data as for the Ansari-Bradley test:
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
mood.test(ramsay, jung.parekh)
## Compare this to ansari.test(ramsay, jung.parekh)
```

`Multinom` *The Multinomial Distribution*

Description

Generate multinomially distributed random number vectors and compute multinomial probabilities.

Usage

```
rmultinom(n, size, prob)
dmultinom(x, size = NULL, prob, log = FALSE)
```

Arguments

`x`	vector of length K of integers in `0:size`.
`n`	number of random vectors to draw.
`size`	integer, say N, specifying the total number of objects that are put into K boxes in the typical multinomial experiment. For `dmultinom`, it defaults to `sum(x)`.
`prob`	numeric non-negative vector of length K, specifying the probability for the K classes; is internally normalized to sum 1. Infinite and missing values are not allowed.
`log`	logical; if TRUE, log probabilities are computed.

Details

If `x` is a K-component vector, `dmultinom(x, prob)` is the probability

$$P(X_1 = x_1, \ldots, X_K = x_k) = C \times \prod_{j=1}^{K} \pi_j^{x_j}$$

where C is the 'multinomial coefficient' $C = N!/(x_1! \cdots x_K!)$ and $N = \sum_{j=1}^{K} x_j$.
By definition, each component X_j is binomially distributed as `Bin(size, prob[j])` for $j = 1, \ldots, K$.

The `rmultinom()` algorithm draws binomials X_j from $Bin(n_j, P_j)$ sequentially, where $n_1 = N$ (N := `size`), $P_1 = \pi_1$ (π is `prob` scaled to sum 1), and for $j \geq 2$, recursively, $n_j = N - \sum_{k=1}^{j-1} X_k$ and $P_j = \pi_j/(1 - \sum_{k=1}^{j-1} \pi_k)$.

Value

For `rmultinom()`, an integer $K \times n$ matrix where each column is a random vector generated according to the desired multinomial law, and hence summing to `size`. Whereas the *transposed* result would seem more natural at first, the returned matrix is more efficient because of columnwise storage.

Note

dmultinom is currently *not vectorized* at all and has no C interface (API); this may be amended in the future.

See Also

Distributions for standard distributions, including dbinom which is a special case conceptually.

Examples

```
rmultinom(10, size = 12, prob = c(0.1,0.2,0.8))

pr <- c(1,3,6,10) # normalization not necessary for generation
rmultinom(10, 20, prob = pr)

## all possible outcomes of Multinom(N = 3, K = 3)
X <- t(as.matrix(expand.grid(0:3, 0:3))); X <- X[, colSums(X) <= 3]
X <- rbind(X, 3:3 - colSums(X)); dimnames(X) <- list(letters[1:3], NULL)
X
round(apply(X, 2, function(x) dmultinom(x, prob = c(1,2,5))), 3)
```

---

na.action *NA Action*

Description

Extract information on the NA action used to create an object.

Usage

```
na.action(object, ...)
```

Arguments

object	any object whose NA action is given.
...	further arguments special methods could require.

Details

na.action is a generic function, and na.action.default its default method. The latter extracts the "na.action" component of a list if present, otherwise the "na.action" attribute.

When model.frame is called, it records any information on NA handling in a "na.action" attribute. Most model-fitting functions return this as a component of their result.

Value

Information from the action which was applied to object if NAs were handled specially, or NULL.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`options("na.action")`, `na.omit`, `na.fail`, also for `na.exclude`, `na.pass`.

Examples

```
na.action(na.omit(c(1, NA)))
```

`na.contiguous` *Find Longest Contiguous Stretch of non-NAs*

Description

Find the longest consecutive stretch of non-missing values in a time series object. (In the event of a tie, the first such stretch.)

Usage

```
na.contiguous(object, ...)
```

Arguments

`object`	a univariate or multivariate time series.
`...`	further arguments passed to or from other methods.

Value

A time series without missing values. The class of `object` will be preserved.

See Also

`na.omit` and `na.omit.ts`; `na.fail`

Examples

```
na.contiguous(presidents)
```

na.fail *Handle Missing Values in Objects*

Description

These generic functions are useful for dealing with NAs in e.g., data frames. na.fail returns the object if it does not contain any missing values, and signals an error otherwise. na.omit returns the object with incomplete cases removed. na.pass returns the object unchanged.

Usage

```
na.fail(object, ...)
na.omit(object, ...)
na.exclude(object, ...)
na.pass(object, ...)
```

Arguments

object	an R object, typically a data frame
...	further arguments special methods could require.

Details

At present these will handle vectors, matrices and data frames comprising vectors and matrices (only).

If na.omit removes cases, the row numbers of the cases form the "na.action" attribute of the result, of class "omit".

na.exclude differs from na.omit only in the class of the "na.action" attribute of the result, which is "exclude". This gives different behaviour in functions making use of naresid and napredict: when na.exclude is used the residuals and predictions are padded to the correct length by inserting NAs for cases omitted by na.exclude.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

na.action; options with argument na.action for setting NA actions; and lm and glm for functions using these. na.contiguous as alternative for time series.

Examples

```
DF <- data.frame(x = c(1, 2, 3), y = c(0, 10, NA))
na.omit(DF)
m <- as.matrix(DF)
na.omit(m)
stopifnot(all(na.omit(1:3) == 1:3))  # does not affect objects with no NA's
try(na.fail(DF))   #> Error: missing values in ...

options("na.action")
```

---

naprint *Adjust for Missing Values*

Description

Use missing value information to report the effects of an `na.action`.

Usage

```
naprint(x, ...)
```

Arguments

`x`	An object produced by an `na.action` function.
`...`	further arguments passed to or from other methods.

Details

This is a generic function, and the exact information differs by method. `naprint.omit` reports the number of rows omitted: `naprint.default` reports an empty string.

Value

A character string providing information on missing values, for example the number.

---

naresid *Adjust for Missing Values*

Description

Use missing value information to adjust residuals and predictions.

Usage

```
naresid(omit, x, ...)
napredict(omit, x, ...)
```

Arguments

omit	an object produced by an na.action function, typically the "na.action" attribute of the result of na.omit or na.exclude.
x	a vector, data frame, or matrix to be adjusted based upon the missing value information.
...	further arguments passed to or from other methods.

Details

These are utility functions used to allow predict, fitted and residuals methods for modelling functions to compensate for the removal of NAs in the fitting process. They are used by the default, "lm", "glm" and "nls" methods, and by further methods in packages **MASS**, **rpart** and **survival**. Also used for the scores returned by factanal, prcomp and princomp.

The default methods do nothing. The default method for the na.exclude action is to pad the object with NAs in the correct positions to have the same number of rows as the original data frame.

Currently naresid and napredict are identical, but future methods need not be. naresid is used for residuals, and napredict for fitted values, predictions and weights.

Value

These return a similar object to x.

Note

In the early 2000s, packages **rpart** and **survival5** contained versions of these functions that had an na.omit action equivalent to that now used for na.exclude.

NegBinomial *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters size and prob.

Usage

```
dnbinom(x, size, prob, mu, log = FALSE)
pnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
qnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
rnbinom(n, size, prob, mu)
```

Arguments

x	vector of (non-negative integer) quantiles.
q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
size	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution). Must be strictly positive, need not be integer.
prob	probability of success in each trial. 0 < prob <= 1.
mu	alternative parametrization via mean: see 'Details'.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The negative binomial distribution with size $= n$ and prob $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. The mean is $\mu = n(1-p)/p$ and variance $n(1-p)/p^2$.

A negative binomial distribution can also arise as a mixture of Poisson distributions with mean distributed as a gamma distribution (see pgamma) with scale parameter (1 - prob)/prob and shape parameter size. (This definition allows non-integer values of size.)

An alternative parametrization (often used in ecology) is by the *mean* mu (see above), and size, the *dispersion parameter*, where prob = size/(size+mu). The variance is mu + mu^2/size in this parametrization.

If an element of x is not integer, the result of dnbinom is zero, with a warning.

The case size == 0 is the distribution concentrated at zero. This is the limiting distribution for size approaching zero, even if mu rather than prob is held constant. Notice though, that the mean of the limit distribution is 0, whatever the value of mu.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

dnbinom gives the density, pnbinom gives the distribution function, qnbinom gives the quantile function, and rnbinom generates random deviates.

Invalid size or prob will result in return value NaN, with a warning.

The length of the result is determined by n for rnbinom, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

rnbinom returns a vector of type integer unless generated values exceed the maximum representable integer when double values are returned.

Source

dnbinom computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

pnbinom uses pbeta.

qnbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rnbinom uses the derivation as a gamma mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for standard distributions, including dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
require(graphics)
x <- 0:11
dnbinom(x, size = 1, prob = 1/2) * 2^(1 + x) # == 1
126 /  dnbinom(0:8, size  = 2, prob  = 1/2) #- theoretically integer

## Cumulative ('p') = Sum of discrete prob.s ('d');  Relative error :
summary(1 - cumsum(dnbinom(x, size = 2, prob = 1/2)) /
                  pnbinom(x, size  = 2, prob = 1/2))

x <- 0:15
size <- (1:20)/4
persp(x, size, dnb <- outer(x, size, function(x,s) dnbinom(x, s, prob = 0.4)),
      xlab = "x", ylab = "s", zlab = "density", theta = 150)
title(tit <- "negative binomial density(x,s, pr = 0.4)  vs.  x & s")

image  (x, size, log10(dnb), main = paste("log [", tit, "]"))
contour(x, size, log10(dnb), add = TRUE)

## Alternative parametrization
x1 <- rnbinom(500, mu = 4, size = 1)
x2 <- rnbinom(500, mu = 4, size = 10)
x3 <- rnbinom(500, mu = 4, size = 100)
h1 <- hist(x1, breaks = 20, plot = FALSE)
h2 <- hist(x2, breaks = h1$breaks, plot = FALSE)
h3 <- hist(x3, breaks = h1$breaks, plot = FALSE)
barplot(rbind(h1$counts, h2$counts, h3$counts),
        beside = TRUE, col = c("red","blue","cyan"),
```

```
names.arg = round(h1$breaks[-length(h1$breaks)]))
```

---

nextn *Find Highly Composite Numbers*

Description

nextn returns the smallest integer, greater than or equal to n, which can be obtained as a product of powers of the values contained in factors.

nextn() is intended to be used to find a suitable length to zero-pad the argument of fft so that the transform is computed quickly. The default value for factors ensures this.

Usage

```
nextn(n, factors = c(2,3,5))
```

Arguments

n	a vector of integer numbers (of type "integer" *or* "double").
factors	a vector of positive integer factors (at least 2 and preferably relative prime, see the note).

Value

a vector of the same length as n, of type "integer" when the values are small enough (determined before computing them) and "double" otherwise.

Note

If the factors in factors are *not* relative prime, i.e., have themselves a common factor larger than one, the result may be wrong in the sense that it may not be the *smallest* integer. E.g., nextn(91, c(2,6)) returns 128 instead of 96 as nextn(91, c(2,3)) returns.

When the resulting N <- nextn(..) is larger than 2^53, a warning with the true 64-bit integer value is signalled, as integers above that range may not be representable in double precision.

If you really need to deal with such large integers, it may be advisable to use package **gmp**.

See Also

convolve, fft.

Examples

```
nextn(1001) # 1024
table(nextn(599:630))
n <- 1:100 ; plot(n, nextn(n) - n, type = "o", lwd=2, cex=1/2)
```

nlm *Non-Linear Minimization*

Description

This function carries out a minimization of the function f using a Newton-type algorithm. See the references for details.

Usage

```
nlm(f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
    fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-6,
    stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),
    steptol = 1e-6, iterlim = 100, check.analyticals = TRUE)
```

Arguments

f	the function to be minimized, returning a single numeric value. This should be a function with first argument a vector of the length of p followed by any other arguments specified by the ... argument. If the function value has an attribute called gradient or both gradient and hessian attributes, these will be used in the calculation of updated parameter values. Otherwise, numerical derivatives are used. deriv returns a function with suitable gradient attribute and optionally a hessian attribute.
p	starting parameter values for the minimization.
...	additional arguments to be passed to f.
hessian	if TRUE, the hessian of f at the minimum is returned.
typsize	an estimate of the size of each parameter at the minimum.
fscale	an estimate of the size of f at the minimum.
print.level	this argument determines the level of printing which is done during the minimization process. The default value of 0 means that no printing occurs, a value of 1 means that initial and final details are printed and a value of 2 means that full tracing information is printed.
ndigit	the number of significant digits in the function f.
gradtol	a positive scalar giving the tolerance at which the scaled gradient is considered close enough to zero to terminate the algorithm. The scaled gradient is a measure of the relative change in f in each direction p[i] divided by the relative change in p[i].
stepmax	a positive scalar which gives the maximum allowable scaled step length. stepmax is used to prevent steps which would cause the optimization function to overflow, to prevent the algorithm from leaving the area of interest in parameter space, or to detect divergence in the algorithm. stepmax would be chosen small enough to prevent the first two of these occurrences, but should be larger than any anticipated reasonable step.

`steptol` A positive scalar providing the minimum allowable relative step length.

`iterlim` a positive integer specifying the maximum number of iterations to be performed before the program is terminated.

`check.analyticals`
a logical scalar specifying whether the analytic gradients and Hessians, if they are supplied, should be checked against numerical derivatives at the initial parameter values. This can help detect incorrectly formulated gradients or Hessians.

Details

Note that arguments after `...` must be matched exactly.

If a gradient or hessian is supplied but evaluates to the wrong mode or length, it will be ignored if `check.analyticals = TRUE` (the default) with a warning. The hessian is not even checked unless the gradient is present and passes the sanity checks.

The C code for the "perturbed" Cholesky, `choldc()` has had a bug in all R versions before 3.4.1.

From the three methods available in the original source, we always use method "1" which is line search.

The functions supplied should always return finite (including not `NA` and not `NaN`) values: for the function value itself non-finite values are replaced by the maximum positive value with a warning.

Value

A list containing the following components:

`minimum` the value of the estimated minimum of `f`.

`estimate` the point at which the minimum value of `f` is obtained.

`gradient` the gradient at the estimated minimum of `f`.

`hessian` the hessian at the estimated minimum of `f` (if requested).

`code` an integer indicating why the optimization process terminated.

- **1:** relative gradient is close to zero, current iterate is probably solution.
- **2:** successive iterates within tolerance, current iterate is probably solution.
- **3:** last global step failed to locate a point lower than `estimate`. Either `estimate` is an approximate local minimum of the function or `steptol` is too small.
- **4:** iteration limit exceeded.
- **5:** maximum step size `stepmax` exceeded five consecutive times. Either the function is unbounded below, becomes asymptotic to a finite value from above in some direction or `stepmax` is too small.

`iterations` the number of iterations performed.

Source

The current code is by Saikat DebRoy and the R Core team, using a C translation of Fortran code by Richard H. Jones.

References

Dennis, J. E. and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ.

Schnabel, R. B., Koontz, J. E. and Weiss, B. E. (1985). A modular system of algorithms for unconstrained minimization. *ACM Transactions on Mathematical Software*, **11**, 419–440. doi:10.1145/6187.6192.

See Also

`optim` and `nlminb`.

`constrOptim` for constrained optimization, `optimize` for one-dimensional minimization and `uniroot` for root finding. `deriv` to calculate analytical derivatives.

For nonlinear regression, `nls` may be better.

Examples

```
f <- function(x) sum((x-1:length(x))^2)
nlm(f, c(10,10))
nlm(f, c(10,10), print.level = 2)
utils::str(nlm(f, c(5), hessian = TRUE))

f <- function(x, a) sum((x-a)^2)
nlm(f, c(10,10), a = c(3,5))
f <- function(x, a)
{
    res <- sum((x-a)^2)
    attr(res, "gradient") <- 2*(x-a)
    res
}
nlm(f, c(10,10), a = c(3,5))

## more examples, including the use of derivatives.
## Not run: demo(nlm)
```

---

nlminb *Optimization using PORT routines*

Description

Unconstrained and box-constrained optimization using PORT routines.

For historical compatibility.

Usage

```
nlminb(start, objective, gradient = NULL, hessian = NULL, ...,
       scale = 1, control = list(), lower = -Inf, upper = Inf)
```

Arguments

start	numeric vector, initial values for the parameters to be optimized.
objective	Function to be minimized. Must return a scalar value. The first argument to objective is the vector of parameters to be optimized, whose initial values are supplied through start. Further arguments (fixed during the course of the optimization) to objective may be specified as well (see ...).
gradient	Optional function that takes the same arguments as objective and evaluates the gradient of objective at its first argument. Must return a vector as long as start.
hessian	Optional function that takes the same arguments as objective and evaluates the hessian of objective at its first argument. Must return a square matrix of order length(start). Only the lower triangle is used.
...	Further arguments to be supplied to objective.
scale	See PORT documentation (or leave alone).
control	A list of control parameters. See below for details.
lower, upper	vectors of lower and upper bounds, replicated to be as long as start. If unspecified, all parameters are assumed to be unconstrained.

Details

Any names of start are passed on to objective and where applicable, gradient and hessian. The parameter vector will be coerced to double.

If any of the functions returns NA or NaN this is an error for the gradient and Hessian, and such values for function evaluation are replaced by +Inf with a warning.

Value

A list with components:

par	The best set of parameters found.
objective	The value of objective corresponding to par.
convergence	An integer code. 0 indicates successful convergence.
message	A character string giving any additional information returned by the optimizer, or NULL. For details, see PORT documentation.
iterations	Number of iterations performed.
evaluations	Number of objective function and gradient function evaluations

Control parameters

Possible names in the control list and their default values are:

eval.max Maximum number of evaluations of the objective function allowed. Defaults to 200.

iter.max Maximum number of iterations allowed. Defaults to 150.

trace The value of the objective function and the parameters is printed every trace'th iteration. Defaults to 0 which indicates no trace information is to be printed.

`abs.tol` Absolute tolerance. Defaults to 0 so the absolute convergence test is not used. If the objective function is known to be non-negative, the previous default of `1e-20` would be more appropriate.

`rel.tol` Relative tolerance. Defaults to `1e-10`.

`x.tol` X tolerance. Defaults to `1.5e-8`.

`xf.tol` false convergence tolerance. Defaults to `2.2e-14`.

`step.min, step.max` Minimum and maximum step size. Both default to `1.`.

sing.tol singular convergence tolerance; defaults to `rel.tol`.

scale.init ...

diff.g an estimated bound on the relative error in the objective function value.

Author(s)

R port: Douglas Bates and Deepayan Sarkar.

Underlying Fortran code by David M. Gay

Source

https://netlib.org/port/

References

David M. Gay (1990), Usage summary for selected optimization routines. Computing Science Technical Report 153, AT&T Bell Laboratories, Murray Hill.

See Also

`optim` (which is preferred) and `nlm`.

`optimize` for one-dimensional minimization and `constrOptim` for constrained optimization.

Examples

```
x <- rnbinom(100, mu = 10, size = 10)
hdev <- function(par)
    -sum(dnbinom(x, mu = par[1], size = par[2], log = TRUE))
nlminb(c(9, 12), hdev)
nlminb(c(20, 20), hdev, lower = 0, upper = Inf)
nlminb(c(20, 20), hdev, lower = 0.001, upper = Inf)

## slightly modified from the S-PLUS help page for nlminb
# this example minimizes a sum of squares with known solution y
sumsq <- function( x, y) {sum((x-y)^2)}
y <- rep(1,5)
x0 <- rnorm(length(y))
nlminb(start = x0, sumsq, y = y)
# now use bounds with a y that has some components outside the bounds
y <- c( 0, 2, 0, -2, 0)
nlminb(start = x0, sumsq, lower = -1, upper = 1, y = y)
```

```
# try using the gradient
sumsq.g <- function(x, y) 2*(x-y)
nlminb(start = x0, sumsq, sumsq.g,
       lower = -1, upper = 1, y = y)
# now use the hessian, too
sumsq.h <- function(x, y) diag(2, nrow = length(x))
nlminb(start = x0, sumsq, sumsq.g, sumsq.h,
       lower = -1, upper = 1, y = y)

## Rest lifted from optim help page

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
nlminb(c(-1.2,1), fr)
nlminb(c(-1.2,1), fr, grr)


flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
## par[24] is *not* at boundary
nlminb(rep(3, 25), flb, lower = rep(2, 25), upper = rep(4, 25))
## trying to use a too small tolerance:
r <- nlminb(rep(3, 25), flb, control = list(rel.tol = 1e-16))
stopifnot(grepl("rel.tol", r$message))
```

`nls` *Nonlinear Least Squares*

Description

Determine the nonlinear (weighted) least-squares estimates of the parameters of a nonlinear model.

Usage

```
nls(formula, data, start, control, algorithm,
    trace, subset, weights, na.action, model,
    lower, upper, ...)
```

Arguments

formula	a nonlinear model formula including variables and parameters. Will be coerced to a formula if necessary.
data	an optional data frame in which to evaluate the variables in formula and weights. Can also be a list or an environment, but not a matrix.
start	a named list or named numeric vector of starting estimates. When start is missing (and formula is not a self-starting model, see selfStart), a very cheap guess for start is tried (if algorithm != "plinear").
control	an optional list of control settings. See nls.control for the names of the settable control values and their effect.
algorithm	character string specifying the algorithm to use. The default algorithm is a Gauss-Newton algorithm. Other possible values are "plinear" for the Golub-Pereyra algorithm for partially linear least-squares models and "port" for the 'nl2sol' algorithm from the Port library – see the references. Can be abbreviated.
trace	logical value indicating if a trace of the iteration progress should be printed. Default is FALSE. If TRUE the residual (weighted) sum-of-squares, the convergence criterion and the parameter values are printed at the conclusion of each iteration. Note that format() is used, so these mostly depend on getOption("digits"). When the "plinear" algorithm is used, the conditional estimates of the linear parameters are printed after the nonlinear parameters. When the "port" algorithm is used the objective function value printed is half the residual (weighted) sum-of-squares.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional numeric vector of (fixed) weights. When present, the objective function is weighted least squares.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Value na.exclude can be useful.
model	logical. If true, the model frame is returned as part of the object. Default is FALSE.
lower, upper	vectors of lower and upper bounds, replicated to be as long as start. If unspecified, all parameters are assumed to be unconstrained. Bounds can only be used with the "port" algorithm. They are ignored, with a warning, if given for other algorithms.
...	Additional optional arguments. None are used at present.

Details

An nls object is a type of fitted model object. It has methods for the generic functions anova, coef, confint, deviance, df.residual, fitted, formula, logLik, predict, print, profile, residuals, summary, vcov and weights.

Variables in formula (and weights if not missing) are looked for first in data, then the environment of formula and finally along the search path. Functions in formula are searched for first in the environment of formula and then along the search path.

Arguments subset and na.action are supported only when all the variables in the formula taken from data are of the same length: other cases give a warning.

Note that the anova method does not check that the models are nested: this cannot easily be done automatically, so use with care.

Value

A list of

m	an nlsModel object incorporating the model.
data	the expression that was passed to nls as the data argument. The actual data values are present in the environment of the m components, e.g., environment(m$conv).
call	the matched call with several components, notably algorithm.
na.action	the "na.action" attribute (if any) of the model frame.
dataClasses	the "dataClasses" attribute (if any) of the "terms" attribute of the model frame.
model	if model = TRUE, the model frame.
weights	if weights is supplied, the weights.
convInfo	a list with convergence information.
control	the control list used, see the control argument.
convergence, message	for an algorithm = "port" fit only, a convergence code (0 for convergence) and message. To use these is *deprecated*, as they are available from convInfo now.

Warning

The default settings of nls generally fail on artificial "zero-residual" data problems.

The nls function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. This performs well on data of the form

$$y = f(x, \theta) + \varepsilon$$

(with $var(\varepsilon) > 0$). It fails to indicate convergence on data of the form

$$y = f(x, \theta)$$

because the criterion amounts to comparing two components of the round-off error. To avoid a zero-divide in computing the convergence testing value, a positive constant scaleOffset should be added to the denominator sum-of-squares; it is set in control, as in the example below; this does not yet apply to algorithm = "port".

The algorithm = "port" code appears unfinished, and does not even check that the starting value is within the bounds. Use with caution, especially where bounds are supplied.

Note

Setting `warnOnly = TRUE` in the `control` argument (see `nls.control`) returns a non-converged object (since R version 2.5.0) which might be useful for further convergence analysis, *but* ***not*** *for inference*.

Author(s)

Douglas M. Bates and Saikat DebRoy: David M. Gay for the Fortran code used by `algorithm = "port"`.

References

Bates, D. M. and Watts, D. G. (1988) *Nonlinear Regression Analysis and Its Applications*, Wiley

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

https://netlib.org/port/ for the Port library documentation.

See Also

`summary.nls`, `predict.nls`, `profile.nls`.

Self starting models (with 'automatic initial values'): `selfStart`.

Examples

```
require(graphics)

DNase1 <- subset(DNase, Run == 1)

## using a selfStart model
fm1DNase1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
summary(fm1DNase1)
## the coefficients only:
coef(fm1DNase1)
## including their SE, etc:
coef(summary(fm1DNase1))

## using conditional linearity
fm2DNase1 <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(xmid = 0, scal = 1),
                 algorithm = "plinear")
summary(fm2DNase1)

## without conditional linearity
fm3DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1))
summary(fm3DNase1)

## using Port's nl2sol algorithm
```

```
fm4DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 algorithm = "port")
summary(fm4DNase1)

## weighted nonlinear regression
Treated <- Puromycin[Puromycin$state == "treated", ]
weighted.MM <- function(resp, conc, Vm, K)
{
    ## Purpose: exactly as white book p. 451 -- RHS for nls()
    ##  Weighted version of Michaelis-Menten model
    ## ---------------------------------------------------------
    ## Arguments: 'y', 'x' and the two parameters (see book)
    ## ---------------------------------------------------------
    ## Author: Martin Maechler, Date: 23 Mar 2001

    pred <- (Vm * conc)/(K + conc)
    (resp - pred) / sqrt(pred)
}

Pur.wt <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
              start = list(Vm = 200, K = 0.1))
summary(Pur.wt)

## Passing arguments using a list that can not be coerced to a data.frame
lisTreat <- with(Treated,
                 list(conc1 = conc[1], conc.1 = conc[-1], rate = rate))

weighted.MM1 <- function(resp, conc1, conc.1, Vm, K)
{
     conc <- c(conc1, conc.1)
     pred <- (Vm * conc)/(K + conc)
    (resp - pred) / sqrt(pred)
}
Pur.wt1 <- nls( ~ weighted.MM1(rate, conc1, conc.1, Vm, K),
               data = lisTreat, start = list(Vm = 200, K = 0.1))
stopifnot(all.equal(coef(Pur.wt), coef(Pur.wt1)))

## Chambers and Hastie (1992) Statistical Models in S  (p. 537):
## If the value of the right side [of formula] has an attribute called
## 'gradient' this should be a matrix with the number of rows equal
## to the length of the response and one column for each parameter.

weighted.MM.grad <- function(resp, conc1, conc.1, Vm, K)
{
  conc <- c(conc1, conc.1)

  K.conc <- K+conc
  dy.dV <- conc/K.conc
  dy.dK <- -Vm*dy.dV/K.conc
  pred <- Vm*dy.dV
  pred.5 <- sqrt(pred)
```

```
  dev <- (resp - pred) / pred.5
  Ddev <- -0.5*(resp+pred)/(pred.5*pred)
  attr(dev, "gradient") <- Ddev * cbind(Vm = dy.dV, K = dy.dK)
  dev
}

Pur.wt.grad <- nls( ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K),
                   data = lisTreat, start = list(Vm = 200, K = 0.1))

rbind(coef(Pur.wt), coef(Pur.wt1), coef(Pur.wt.grad))

## In this example, there seems no advantage to providing the gradient.
## In other cases, there might be.


## The two examples below show that you can fit a model to
## artificial data with noise but not to artificial data
## without noise.
x <- 1:10
y <- 2*x + 3                            # perfect fit
## terminates in an error, because convergence cannot be confirmed:
try(nls(y ~ a + b*x, start = list(a = 0.12345, b = 0.54321)))
## adjusting the convergence test by adding 'scaleOffset' to its denominator RSS:
nls(y ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
    control = list(scaleOffset = 1, printEval=TRUE))
## Alternatively jittering the "too exact" values, slightly:
set.seed(27)
yeps <- y + rnorm(length(y), sd = 0.01) # added noise
nls(yeps ~ a + b*x, start = list(a = 0.12345, b = 0.54321))


## the nls() internal cheap guess for starting values can be sufficient:
x <- -(1:100)/10
y <- 100 + 10 * exp(x / 2) + rnorm(x)/10
nlmod <- nls(y ~  Const + A * exp(B * x))

plot(x,y, main = "nls(*), data, true function and fit, n=100")
curve(100 + 10 * exp(x / 2), col = 4, add = TRUE)
lines(x, predict(nlmod), col = 2)


## Here, requiring close convergence, must use more accurate numerical differentiation,
## as this typically gives Error: "step factor .. reduced below 'minFactor' .."

try(nlm1 <- update(nlmod, control = list(tol = 1e-7)))
o2 <- options(digits = 10) # more accuracy for 'trace'
## central differencing works here typically (PR#18165: not converging on *some*):
ctr2 <- nls.control(nDcentral=TRUE, tol = 8e-8, # <- even smaller than above
   warnOnly =
        TRUE || # << work around; e.g. needed on some ATLAS-Lapack setups
        (grepl("^aarch64.*linux", R.version$platform) && grepl("^NixOS", osVersion)
             ))
(nlm2 <- update(nlmod, control = ctr2, trace = TRUE)); options(o2)
```

```
## --> convergence tolerance  4.997e-8 (in 11 iter.)


## The muscle dataset in MASS is from an experiment on muscle
## contraction on 21 animals.  The observed variables are Strip
## (identifier of muscle), Conc (Cacl concentration) and Length
## (resulting length of muscle section).

if(requireNamespace("MASS", quietly = TRUE)) withAutoprint({
## The non linear model considered is
##       Length = alpha + beta*exp(-Conc/theta) + error
## where theta is constant but alpha and beta may vary with Strip.

with(MASS::muscle, table(Strip)) # 2, 3 or 4 obs per strip

## We first use the plinear algorithm to fit an overall model,
## ignoring that alpha and beta might vary with Strip.
musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), MASS::muscle,
              start = list(th = 1), algorithm = "plinear")
summary(musc.1)

## Then we use nls' indexing feature for parameters in non-linear
## models to use the conventional algorithm to fit a model in which
## alpha and beta vary with Strip.  The starting values are provided
## by the previously fitted model.
## Note that with indexed parameters, the starting values must be
## given in a list (with names):
b <- coef(musc.1)
musc.2 <- nls(Length ~ a[Strip] + b[Strip]*exp(-Conc/th), MASS::muscle,
              start = list(a = rep(b[2], 21), b = rep(b[3], 21), th = b[1]))
summary(musc.2)
})
```

`nls.control` *Control the Iterations in* `nls`

Description

Allow the user to set some characteristics of the `nls` nonlinear least squares algorithm.

Usage

```
nls.control(maxiter = 50, tol = 1e-05, minFactor = 1/1024,
            printEval = FALSE, warnOnly = FALSE, scaleOffset = 0,
            nDcentral = FALSE)
```

Arguments

maxiter	A positive integer specifying the maximum number of iterations allowed.
tol	A positive numeric value specifying the tolerance level for the relative offset convergence criterion.
minFactor	A positive numeric value specifying the minimum step-size factor allowed on any step in the iteration. The increment is calculated with a Gauss-Newton algorithm and successively halved until the residual sum of squares has been decreased or until the step-size factor has been reduced below this limit.
printEval	a logical specifying whether the number of evaluations (steps in the gradient direction taken each iteration) is printed.
warnOnly	a logical specifying whether nls() should return instead of signalling an error in the case of termination before convergence. Termination before convergence happens upon completion of maxiter iterations, in the case of a singular gradient, and in the case that the step-size factor is reduced below minFactor.
scaleOffset	a constant to be added to the denominator of the relative offset convergence criterion calculation to avoid a zero divide in the case where the fit of a model to data is very close. The default value of 0 keeps the legacy behaviour of nls(). A value such as 1 seems to work for problems of reasonable scale with very small residuals.
nDcentral	only when *numerical* derivatives are used: logical indicating if *central* differences should be employed, i.e., numericDeriv(*, central=TRUE) be used.

Value

A list with components

maxiter

tol

minFactor

printEval

warnOnly

scaleOffset

nDcentreal

with meanings as explained under 'Arguments'.

Author(s)

Douglas Bates and Saikat DebRoy; John C. Nash for part of the scaleOffset option.

References

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley.

See Also

nls

Examples

```
nls.control(minFactor = 1/2048)
```

`NLSstAsymptotic` *Fit the Asymptotic Regression Model*

Description

Fits the asymptotic regression model, in the form `b0 + b1*(1-exp(-exp(lrc) * x))` to the `xy` data. This can be used as a building block in determining starting estimates for more complicated models.

Usage

```
NLSstAsymptotic(xy)
```

Arguments

`xy`	a `sortedXyData` object

Value

A numeric value of length 3 with components labelled `b0`, `b1`, and `lrc`. `b0` is the estimated intercept on the `y`-axis, `b1` is the estimated difference between the asymptote and the `y`-intercept, and `lrc` is the estimated logarithm of the rate constant.

Author(s)

José Pinheiro and Douglas Bates

See Also

`SSasymp`

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
print(NLSstAsymptotic(sortedXyData(expression(age),
                                   expression(height),
                                   Lob.329)), digits = 3)
```

NLSstClosestX *Inverse Interpolation*

Description

Use inverse linear interpolation to approximate the x value at which the function represented by xy is equal to yval.

Usage

```
NLSstClosestX(xy, yval)
```

Arguments

xy	a sortedXyData object
yval	a numeric value on the y scale

Value

A single numeric value on the x scale.

Author(s)

José Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstLfAsymptote, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData(expression(log(conc)), expression(density), DNase.2)
NLSstClosestX(DN.srt, 1.0)
```

NLSstLfAsymptote *Horizontal Asymptote on the Left Side*

Description

Provide an initial guess at the horizontal asymptote on the left side (i.e., small values of x) of the graph of y versus x from the xy object. Primarily used within initial functions for self-starting nonlinear regression models.

Usage

```
NLSstLfAsymptote(xy)
```

Arguments

xy	a sortedXyData object

Value

A single numeric value estimating the horizontal asymptote for small x.

Author(s)

José Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstClosestX, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstLfAsymptote( DN.srt )
```

NLSstRtAsymptote *Horizontal Asymptote on the Right Side*

Description

Provide an initial guess at the horizontal asymptote on the right side (i.e., large values of x) of the graph of y versus x from the xy object. Primarily used within initial functions for self-starting nonlinear regression models.

Usage

```
NLSstRtAsymptote(xy)
```

Arguments

xy	a sortedXyData object

Value

A single numeric value estimating the horizontal asymptote for large x.

Author(s)

José Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstClosestX, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstRtAsymptote( DN.srt )
```

---

nobs *Extract the Number of Observations from a Fit*

Description

Extract the number of 'observations' from a model fit. This is principally intended to be used in computing BIC (see AIC).

Usage

```
nobs(object, ...)

## Default S3 method:
nobs(object, use.fallback = FALSE, ...)
```

Arguments

object	a fitted model object.
use.fallback	logical: should fallback methods be used to try to guess the value?
...	further arguments to be passed to methods.

Details

This is a generic function, with an S4 generic in package **stats4**. There are methods in this package for objects of classes "lm", "glm", "nls" and "logLik", as well as a default method (which throws an error, unless use.fallback = TRUE when it looks for weights and residuals components – use with care!).

The main usage is in determining the appropriate penalty for BIC, but nobs is also used by the stepwise fitting methods step, add1 and drop1 as a quick check that different fits have been fitted to the same set of data (and not, say, that further rows have been dropped because of NAs in the new predictors).

For lm, glm and nls fits, observations with zero weight are not included.

Value

A single number, normally an integer. Could be NA.

See Also

AIC.

Normal *The Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
mean	vector of means.
sd	vector of standard deviations.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$ otherwise, $P[X > x]$.

Details

If mean or sd are not specified they assume the default values of 0 and 1, respectively.

The normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

where μ is the mean of the distribution and σ the standard deviation.

Value

dnorm gives the density, pnorm gives the distribution function, qnorm gives the quantile function, and rnorm generates random deviates.

The length of the result is determined by n for rnorm, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

For sd = 0 this gives the limit as sd decreases to 0, a point mass at mu. sd < 0 is an error and returns NaN.

Source

For pnorm, based on

Cody, W. D. (1993) Algorithm 715: SPECFUN – A portable FORTRAN package of special function routines and test drivers. *ACM Transactions on Mathematical Software* **19**, 22–32.

For qnorm, the code is based on a C translation of

Wichura, M. J. (1988) Algorithm AS 241: The percentage points of the normal distribution. *Applied Statistics*, **37**, 477–484; doi:10.2307/2347330.

which provides precise results up to about 16 digits for log.p=FALSE. For log scale probabilities in the extreme tails, since R version 4.1.0, extensively since 4.3.0, asymptotic expansions are used which have been derived and explored in

Maechler, M. (2022) Asymptotic tail formulas for gaussian quantiles; **DPQ** vignette https://CRAN.R-project.org/package=DPQ/vignettes/qnorm-asymp.pdf.

For rnorm, see RNG for how to select the algorithm and for references to the supplied methods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 13. Wiley, New York.

See Also

Distributions for other standard distributions, including dlnorm for the *Log*normal distribution.

Examples

```
require(graphics)

dnorm(0) == 1/sqrt(2*pi)
dnorm(1) == exp(-1/2)/sqrt(2*pi)
dnorm(1) == 1/sqrt(2*pi*exp(1))

## Using "log = TRUE" for an extended range :
par(mfrow = c(2,1))
plot(function(x) dnorm(x, log = TRUE), -60, 50,
     main = "log { Normal density }")
curve(log(dnorm(x)), add = TRUE, col = "red", lwd = 2)
mtext("dnorm(x, log=TRUE)", adj = 0)
mtext("log(dnorm(x))", col = "red", adj = 1)

plot(function(x) pnorm(x, log.p = TRUE), -50, 10,
     main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add = TRUE, col = "red", lwd = 2)
mtext("pnorm(x, log=TRUE)", adj = 0)
mtext("log(pnorm(x))", col = "red", adj = 1)

## if you want the so-called 'error function'
```

```
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
## (see Abramowitz and Stegun 29.2.29)
## and the so-called 'complementary error function'
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
## and the inverses
erfinv <- function (x) qnorm((1 + x)/2)/sqrt(2)
erfcinv <- function (x) qnorm(x/2, lower = FALSE)/sqrt(2)
```

`numericDeriv` *Evaluate Derivatives Numerically*

Description

`numericDeriv` numerically evaluates the gradient of an expression.

Usage

```
numericDeriv(expr, theta, rho = parent.frame(), dir = 1,
             eps = .Machine$double.eps ^ (1/if(central) 3 else 2), central = FALSE)
```

Arguments

`expr`	`expression` or `call` to be differentiated. Should evaluate to a `numeric` vector.
`theta`	`character` vector of names of numeric variables used in `expr`.
`rho`	`environment` containing all the variables needed to evaluate `expr`.
`dir`	numeric vector of directions, typically with values in `-1, 1` to use for the finite differences; will be recycled to the length of `theta`.
`eps`	a positive number, to be used as unit step size h for the approximate numerical derivative $(f(x+h) - f(x))/h$ or the central version, see `central`.
`central`	logical indicating if *central* divided differences should be computed, i.e., $(f(x+h) - f(x-h))/2h$. These are typically more accurate but need more evaluations of $f()$.

Details

This is a front end to the C function `numeric_deriv`, which is described in *Writing R Extensions*.

The numeric variables must be of type `double` and not `integer`.

Value

The value of `eval(expr, envir = rho)` plus a matrix attribute `"gradient"`. The columns of this matrix are the derivatives of the value with respect to the variables listed in `theta`.

Author(s)

Saikat DebRoy <saikat@stat.wisc.edu>; tweaks and `eps`, `central` options by R Core Team.

Examples

```
myenv <- new.env()
myenv$mean <- 0.
myenv$sd   <- 1.
myenv$x    <- seq(-3., 3., length.out = 31)
nD <- numericDeriv(quote(pnorm(x, mean, sd)), c("mean", "sd"), myenv)
str(nD)

## Visualize :
require(graphics)
matplot(myenv$x, cbind(c(nD), attr(nD, "gradient")), type="l")
abline(h=0, lty=3)
## "gradient" is close to the true derivatives, you don't see any diff.:
curve( - dnorm(x), col=2, lty=3, lwd=2, add=TRUE)
curve(-x*dnorm(x), col=3, lty=3, lwd=2, add=TRUE)
##
# shows 1.609e-8 on most platforms
all.equal(attr(nD,"gradient"),
          with(myenv, cbind(-dnorm(x), -x*dnorm(x))))
```

`offset` *Include an Offset in a Model Formula*

Description

An offset is a term to be added to a linear predictor, such as in a generalised linear model, with known coefficient 1 rather than an estimated coefficient.

Usage

```
offset(object)
```

Arguments

`object`	An offset to be included in a model frame

Details

There can be more than one offset in a model formula, but `-` is not supported for `offset` terms (and is equivalent to `+`).

Value

The input value.

See Also

`model.offset`, `model.frame`.

For examples see `glm` and `Insurance` in package **MASS**.

`oneway.test` *Test for Equal Means in a One-Way Layout*

Description

Test whether two or more samples from normal distributions have the same means. The variances are not necessarily assumed to be equal.

Usage

```
oneway.test(formula, data, subset, na.action, var.equal = FALSE)
```

Arguments

`formula`	a formula of the form `lhs ~ rhs` where `lhs` gives the sample values and `rhs` the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`var.equal`	a logical variable indicating whether to treat the variances in the samples as equal. If `TRUE`, then a simple F test for the equality of means in a one-way analysis of variance is performed. If `FALSE`, an approximate method of Welch (1951) is used, which generalizes the commonly known 2-sample Welch test to the case of arbitrarily many samples.

Details

If the right-hand side of the formula contains more than one term, their interaction is taken to form the grouping.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the degrees of freedom of the exact or approximate F distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	a character string indicating the test performed.
`data.name`	a character string giving the names of the data.

References

B. L. Welch (1951). On the comparison of several mean values: an alternative approach. *Biometrika*, **38**, 330–336. doi:10.2307/2332579.

See Also

The standard t test (`t.test`) as the special case for two samples; the Kruskal-Wallis test `kruskal.test` for a nonparametric test for equal location parameters in a one-way layout.

Examples

```
## Not assuming equal variances
oneway.test(extra ~ group, data = sleep)
## Assuming equal variances
oneway.test(extra ~ group, data = sleep, var.equal = TRUE)
## which gives the same result as
anova(lm(extra ~ group, data = sleep))
```

`optim` *General-purpose Optimization*

Description

General-purpose optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithms. It includes an option for box-constrained optimization and simulated annealing.

Usage

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)

optimHess(par, fn, gr = NULL, ..., control = list())
```

Arguments

`par`	Initial values for the parameters to be optimized over.
`fn`	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
`gr`	A function to return the gradient for the `"BFGS"`, `"CG"` and `"L-BFGS-B"` methods. If it is `NULL`, a finite-difference approximation will be used. For the `"SANN"` method it specifies a function to generate a new candidate point. If it is `NULL` a default Gaussian Markov kernel is used.

...	Further arguments to be passed to fn and gr.
method	The method to be used. See 'Details'. Can be abbreviated.
lower, upper	Bounds on the variables for the "L-BFGS-B" method, or bounds in which to *search* for method "Brent".
control	a list of control parameters. See 'Details'.
hessian	Logical. Should a numerically differentiated Hessian matrix be returned?

Details

Note that arguments after ... must be matched exactly.

By default optim performs minimization, but it will maximize if control$fnscale is negative. optimHess is an auxiliary function to compute the Hessian at a later stage if hessian = TRUE was forgotten.

The default method is an implementation of that of Nelder and Mead (1965), that uses only function values and is robust but relatively slow. It will work reasonably well for non-differentiable functions.

Method "BFGS" is a quasi-Newton method (also known as a variable metric algorithm), specifically that published simultaneously in 1970 by Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

Method "CG" is a conjugate gradients method based on that by Fletcher and Reeves (1964) (but with the option of Polak–Ribiere or Beale–Sorenson updates). Conjugate gradient methods will generally be more fragile than the BFGS method, but as they do not store a matrix they may be successful in much larger optimization problems.

Method "L-BFGS-B" is that of Byrd et al. (1995) which allows *box constraints*, that is each variable can be given a lower and/or upper bound. The initial value must satisfy the constraints. This uses a limited-memory modification of the BFGS quasi-Newton method. If non-trivial bounds are supplied, this method will be selected, with a warning.

Nocedal and Wright (1999) is a comprehensive reference for the previous three methods.

Method "SANN" is by default a variant of simulated annealing given in Belisle (1992). Simulated-annealing belongs to the class of stochastic global optimization methods. It uses only function values but is relatively slow. It will also work for non-differentiable functions. This implementation uses the Metropolis function for the acceptance probability. By default the next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. If a function to generate a new candidate point is given, method "SANN" can also be used to solve combinatorial optimization problems. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992, p. 890); specifically, the temperature is set to temp / log(((t-1) %/% tmax)*tmax + exp(1)), where t is the current iteration step and temp and tmax are specifiable via control, see below. Note that the "SANN" method depends critically on the settings of the control parameters. It is not a general-purpose method but can be very useful in getting to a good value on a very rough surface.

Method "Brent" is for one-dimensional problems only, using optimize(). It can be useful in cases where optim() is used inside other functions where only method can be specified, such as in mle from package **stats4**.

Function fn can return NA or Inf if the function cannot be evaluated at the supplied value, but the initial value must have a computable finite value of fn. (Except for method "L-BFGS-B" where the values should always be finite.)

optim can be used recursively, and for a single parameter as well as many. It also accepts a zero-length par, and just evaluates the function with that argument.

The control argument is a list that can supply any of the following components:

trace Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method "L-BFGS-B" there are six levels of tracing. (To understand exactly what these do see the source code: higher levels give more detail.)

fnscale An overall scaling to be applied to the value of fn and gr during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on fn(par)/fnscale.

parscale A vector of scaling values for the parameters. Optimization is performed on par/parscale and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value. Not used (nor needed) for method = "Brent".

ndeps A vector of step sizes for the finite-difference approximation to the gradient, on par/parscale scale. Defaults to 1e-3.

maxit The maximum number of iterations. Defaults to 100 for the derivative-based methods, and 500 for "Nelder-Mead".

For "SANN" maxit gives the total number of function evaluations: there is no other stopping criterion. Defaults to 10000.

abstol The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.

reltol Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of reltol * (abs(val) + reltol) at a step. Defaults to sqrt(.Machine$double.eps), typically about 1e-8.

alpha, beta, gamma Scaling parameters for the "Nelder-Mead" method. alpha is the reflection factor (default 1.0), beta the contraction factor (0.5) and gamma the expansion factor (2.0).

REPORT The frequency of reports for the "BFGS", "L-BFGS-B" and "SANN" methods if control$trace is positive. Defaults to every 10 iterations for "BFGS" and "L-BFGS-B", or every 100 temperatures for "SANN".

warn.1d.NelderMead a logical indicating if the (default) "Nelder-Mead" method should signal a warning when used for one-dimensional minimization. As the warning is sometimes inappropriate, you can suppress it by setting this option to false.

type for the conjugate-gradients method. Takes value 1 for the Fletcher–Reeves update, 2 for Polak–Ribiere and 3 for Beale–Sorenson.

lmm is an integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 5.

factr controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is 1e7, that is a tolerance of about 1e-8.

`pgtol` helps control the convergence of the `"L-BFGS-B"` method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed.

`temp` controls the `"SANN"` method. It is the starting temperature for the cooling schedule. Defaults to `10`.

`tmax` is the number of function evaluations at each temperature for the `"SANN"` method. Defaults to `10`.

Any names given to `par` will be copied to the vectors passed to `fn` and `gr`. Note that no other attributes of `par` are copied over.

The parameter vector passed to `fn` has special semantics and may be shared between calls: the function should not change or copy it.

Value

For `optim`, a list with components:

`par` The best set of parameters found.

`value` The value of `fn` corresponding to `par`.

`counts` A two-element integer vector giving the number of calls to `fn` and `gr` respectively. This excludes those calls needed to compute the Hessian, if requested, and any calls to `fn` to compute a finite-difference approximation to the gradient.

`convergence` An integer code. `0` indicates successful completion (which is always the case for `"SANN"` and `"Brent"`). Possible error codes are

- `1` indicates that the iteration limit `maxit` had been reached.
- `10` indicates degeneracy of the Nelder–Mead simplex.
- `51` indicates a warning from the `"L-BFGS-B"` method; see component `message` for further details.
- `52` indicates an error from the `"L-BFGS-B"` method; see component `message` for further details.

`message` A character string giving any additional information returned by the optimizer, or `NULL`.

`hessian` Only if argument `hessian` is true. A symmetric matrix giving an estimate of the Hessian at the solution found. Note that this is the Hessian of the unconstrained problem even if the box constraints are active.

For `optimHess`, the description of the `hessian` component applies.

Note

`optim` will work with one-dimensional `par`s, but the default method does not work well (and will warn). Method `"Brent"` uses `optimize` and needs bounds to be available; `"BFGS"` often works well enough if not.

Source

The code for methods "Nelder-Mead", "BFGS" and "CG" was based originally on Pascal code in Nash (1990) that was translated by p2c and then hand-optimized. Dr Nash has agreed that the code can be made freely available.

The code for method "L-BFGS-B" is based on Fortran code by Zhu, Byrd, Lu-Chen and Nocedal obtained from Netlib (file 'opt/lbfgs_bcm.shar': another version is in 'toms/778').

The code for method "SANN" was contributed by A. Trapletti.

References

Belisle, C. J. P. (1992). Convergence theorems for a class of simulated annealing algorithms on R^d. *Journal of Applied Probability*, **29**, 885–895. doi:10.2307/3214721.

Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, **16**, 1190–1208. doi:10.1137/0916069.

Fletcher, R. and Reeves, C. M. (1964). Function minimization by conjugate gradients. *Computer Journal* **7**, 148–154. doi:10.1093/comjnl/7.2.149.

Nash, J. C. (1990). *Compact Numerical Methods for Computers. Linear Algebra and Function Minimisation*. Adam Hilger.

Nelder, J. A. and Mead, R. (1965). A simplex algorithm for function minimization. *Computer Journal*, **7**, 308–313. doi:10.1093/comjnl/7.4.308.

Nocedal, J. and Wright, S. J. (1999). *Numerical Optimization*. Springer.

See Also

nlm, nlminb.

optimize for one-dimensional minimization and constrOptim for constrained optimization.

Examples

```
require(graphics)

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
optim(c(-1.2,1), fr)
(res <- optim(c(-1.2,1), fr, grr, method = "BFGS"))
optimHess(res$par, fr, grr)
optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
```

```
## These do not converge in the default number of steps
optim(c(-1.2,1), fr, grr, method = "CG")
optim(c(-1.2,1), fr, grr, method = "CG", control = list(type = 2))
optim(c(-1.2,1), fr, grr, method = "L-BFGS-B")

flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
optim(rep(3, 25), flb, NULL, method = "L-BFGS-B",
      lower = rep(2, 25), upper = rep(4, 25)) # par[24] is *not* at boundary


## "wild" function , global minimum at about -15.81515
fw <- function (x)
    10*sin(0.3*x)*sin(1.3*x^2) + 0.00001*x^4 + 0.2*x+80
plot(fw, -50, 50, n = 1000, main = "optim() minimising 'wild function'")

res <- optim(50, fw, method = "SANN",
             control = list(maxit = 20000, temp = 20, parscale = 20))
res
## Now improve locally {typically only by a small bit}:
(r2 <- optim(res$par, fw, method = "BFGS"))
points(r2$par,  r2$value,  pch = 8, col = "red", cex = 2)

## Combinatorial optimization: Traveling salesman problem
library(stats) # normally loaded

eurodistmat <- as.matrix(eurodist)

distance <- function(sq) {  # Target function
    sq2 <- embed(sq, 2)
    sum(eurodistmat[cbind(sq2[,2], sq2[,1])])
}

genseq <- function(sq) {  # Generate new candidate sequence
    idx <- seq(2, NROW(eurodistmat)-1)
    changepoints <- sample(idx, size = 2, replace = FALSE)
    tmp <- sq[changepoints[1]]
    sq[changepoints[1]] <- sq[changepoints[2]]
    sq[changepoints[2]] <- tmp
    sq
}

sq <- c(1:nrow(eurodistmat), 1)  # Initial sequence: alphabetic
distance(sq)
# rotate for conventional orientation
loc <- -cmdscale(eurodist, add = TRUE)$points
x <- loc[,1]; y <- loc[,2]
s <- seq_len(nrow(eurodistmat))
tspinit <- loc[sq,]

plot(x, y, type = "n", asp = 1, xlab = "", ylab = "",
     main = "initial solution of traveling salesman problem", axes = FALSE)
```

```
arrows(tspinit[s,1], tspinit[s,2], tspinit[s+1,1], tspinit[s+1,2],
       angle = 10, col = "green")
text(x, y, labels(eurodist), cex = 0.8)

set.seed(123) # chosen to get a good soln relatively quickly
res <- optim(sq, distance, genseq, method = "SANN",
             control = list(maxit = 30000, temp = 2000, trace = TRUE,
                            REPORT = 500))
res  # Near optimum distance around 12842

tspres <- loc[res$par,]
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "",
     main = "optim() 'solving' traveling salesman problem", axes = FALSE)
arrows(tspres[s,1], tspres[s,2], tspres[s+1,1], tspres[s+1,2],
       angle = 10, col = "red")
text(x, y, labels(eurodist), cex = 0.8)

## 1-D minimization: "Brent" or optimize() being preferred.. but NM may be ok and "unavoidable",
## ----------------   so we can suppress the check+warning :
system.time(r0 <- optimize(function(x) (x-pi)^2, c(0, 10)))
system.time(ro <- optim(1, function(x) (x-pi)^2, control=list(warn.1d.NelderMead = FALSE)))
r0$minimum - pi # 0 (perfect), on one platform
ro$par - pi     # ~= 1.9e-4    on one platform
utils::str(ro)
```

---

optimize *One Dimensional Optimization*

Description

The function optimize searches the interval from lower to upper for a minimum or maximum of the function f with respect to its first argument.

optimise is an alias for optimize.

Usage

```
optimize(f, interval, ..., lower = min(interval), upper = max(interval),
         maximum = FALSE,
         tol = .Machine$double.eps^0.25)
optimise(f, interval, ..., lower = min(interval), upper = max(interval),
         maximum = FALSE,
         tol = .Machine$double.eps^0.25)
```

Arguments

f	the function to be optimized. The function is either minimized or maximized over its first argument depending on the value of maximum.
interval	a vector containing the end-points of the interval to be searched for the minimum.

`...`	additional named or unnamed arguments to be passed to `f`.
`lower`	the lower end point of the interval to be searched.
`upper`	the upper end point of the interval to be searched.
`maximum`	logical. Should we maximize or minimize (the default)?
`tol`	the desired accuracy.

Details

Note that arguments after `...` must be matched exactly.

The method used is a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions. Convergence is never much slower than that for a Fibonacci search. If `f` has a continuous second derivative which is positive at the minimum (which is not at `lower` or `upper`), then convergence is superlinear, and usually of the order of about 1.324.

The function `f` is never evaluated at two points closer together than $\epsilon|x_0| + (tol/3)$, where ϵ is approximately `sqrt(.Machine$double.eps)` and x_0 is the final abscissa `optimize()$minimum`. If `f` is a unimodal function and the computed values of `f` are always unimodal when separated by at least $\epsilon\,|x| + (tol/3)$, then x_0 approximates the abscissa of the global minimum of `f` on the interval `lower,upper` with an error less than $\epsilon|x_0| + tol$.
If `f` is not unimodal, then `optimize()` may approximate a local, but perhaps non-global, minimum to the same accuracy.

The first evaluation of `f` is always at $x_1 = a + (1 - \phi)(b - a)$ where `(a,b) = (lower, upper)` and $\phi = (\sqrt{5} - 1)/2 = 0.61803..$ is the golden section ratio. Almost always, the second evaluation is at $x_2 = a + \phi(b - a)$. Note that a local minimum inside $[x_1, x_2]$ will be found as solution, even when `f` is constant in there, see the last example.

`f` will be called as `f(x, ...)` for a numeric value of x.

The argument passed to `f` has special semantics and used to be shared between calls. The function should not copy it.

Value

A list with components `minimum` (or `maximum`) and `objective` which give the location of the minimum (or maximum) and the value of the function at that point.

Source

A C translation of Fortran code https://netlib.org/fmm/fmin.f (author(s) unstated) based on the Algol 60 procedure `localmin` given in the reference.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs N.J.: Prentice-Hall.

See Also

`nlm`, `uniroot`.

Examples

```
require(graphics)

f <- function (x, a) (x - a)^2
xmin <- optimize(f, c(0, 1), tol = 0.0001, a = 1/3)
xmin

## See where the function is evaluated:
optimize(function(x) x^2*(print(x)-1), lower = 0, upper = 10)

## "wrong" solution with unlucky interval and piecewise constant f():
f  <- function(x) ifelse(x > -1, ifelse(x < 4, exp(-1/abs(x - 1)), 10), 10)
fp <- function(x) { print(x); f(x) }

plot(f, -2,5, ylim = 0:1, col = 2)
optimize(fp, c(-4, 20))   # doesn't see the minimum
optimize(fp, c(-7, 20))   # ok
```

---

order.dendrogram *Ordering or Labels of the Leaves in a Dendrogram*

Description

Theses functions return the order (index) or the "label" attribute for the leaves in a dendrogram. These indices can then be used to access the appropriate components of any additional data.

Usage

```
order.dendrogram(x)

## S3 method for class 'dendrogram'
labels(object, ...)
```

Arguments

x, object	a dendrogram (see as.dendrogram).
...	additional arguments

Details

The indices or labels for the leaves in left to right order are retrieved.

Value

A vector with length equal to the number of leaves in the dendrogram is returned. From r <- order.dendrogram(), each element is the index into the original data (from which the dendrogram was computed).

Author(s)

R. Gentleman (`order.dendrogram`) and Martin Maechler (`labels.dendrogram`).

See Also

`reorder`, `dendrogram`.

Examples

```
set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
hc$order
dd <- as.dendrogram(hc)
order.dendrogram(dd) ## the same :
stopifnot(hc$order == order.dendrogram(dd))

d2 <- as.dendrogram(hclust(dist(USArrests)))
labels(d2) ## in this case the same as
stopifnot(identical(labels(d2),
   rownames(USArrests)[order.dendrogram(d2)]))
```

---

p.adjust *Adjust P-values for Multiple Comparisons*

Description

Given a set of p-values, returns p-values adjusted using one of several methods.

Usage

```
p.adjust(p, method = p.adjust.methods, n = length(p))

p.adjust.methods
# c("holm", "hochberg", "hommel", "bonferroni", "BH", "BY",
#   "fdr", "none")
```

Arguments

`p`	numeric vector of p-values (possibly with `NA`s). Any other R object is coerced by `as.numeric`.
`method`	correction method, a `character` string. Can be abbreviated.
`n`	number of comparisons, must be at least `length(p)`; only set this (to non-default) when you know what you are doing!

Details

The adjustment methods include the Bonferroni correction ("bonferroni") in which the p-values are multiplied by the number of comparisons. Less conservative corrections are also included by Holm (1979) ("holm"), Hochberg (1988) ("hochberg"), Hommel (1988) ("hommel"), Benjamini & Hochberg (1995) ("BH" or its alias "fdr"), and Benjamini & Yekutieli (2001) ("BY"), respectively. A pass-through option ("none") is also included. The set of methods are contained in the p.adjust.methods vector for the benefit of methods that need to have the method as an option and pass it on to p.adjust.

The first four methods are designed to give strong control of the family-wise error rate. There seems no reason to use the unmodified Bonferroni correction because it is dominated by Holm's method, which is also valid under arbitrary assumptions.

Hochberg's and Hommel's methods are valid when the hypothesis tests are independent or when they are non-negatively associated (Sarkar, 1998; Sarkar and Chang, 1997). Hommel's method is more powerful than Hochberg's, but the difference is usually small and the Hochberg p-values are faster to compute.

The "BH" (aka "fdr") and "BY" methods of Benjamini, Hochberg, and Yekutieli control the false discovery rate, the expected proportion of false discoveries amongst the rejected hypotheses. The false discovery rate is a less stringent condition than the family-wise error rate, so these methods are more powerful than the others.

Note that you can set n larger than length(p) which means the unobserved p-values are assumed to be greater than all the observed p for "bonferroni" and "holm" methods and equal to 1 for the other methods.

Value

A numeric vector of corrected p-values (of the same length as p, with names copied from p).

References

Benjamini, Y., and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series B*, **57**, 289–300. doi:10.1111/j.25176161.1995.tb02031.x.

Benjamini, Y., and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, **29**, 1165–1188. doi:10.1214/aos/1013699998.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, **6**, 65–70. https://www.jstor.org/stable/4615733.

Hommel, G. (1988). A stagewise rejective multiple test procedure based on a modified Bonferroni test. *Biometrika*, **75**, 383–386. doi:10.2307/2336190.

Hochberg, Y. (1988). A sharper Bonferroni procedure for multiple tests of significance. *Biometrika*, **75**, 800–803. doi:10.2307/2336325.

Shaffer, J. P. (1995). Multiple hypothesis testing. *Annual Review of Psychology*, **46**, 561–584. doi:10.1146/annurev.ps.46.020195.003021. (An excellent review of the area.)

Sarkar, S. (1998). Some probability inequalities for ordered MTP2 random variables: a proof of Simes conjecture. *Annals of Statistics*, **26**, 494–504. doi:10.1214/aos/1028144846.

Sarkar, S., and Chang, C. K. (1997). The Simes method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, **92**, 1601–1608. doi:10.2307/2965431.

Wright, S. P. (1992). Adjusted P-values for simultaneous inference. *Biometrics*, **48**, 1005–1013. doi:10.2307/2532694. (Explains the adjusted P-value approach.)

See Also

`pairwise.*` functions such as `pairwise.t.test`.

Examples

```
require(graphics)

set.seed(123)
x <- rnorm(50, mean = c(rep(0, 25), rep(3, 25)))
p <- 2*pnorm(sort(-abs(x)))

round(p, 3)
round(p.adjust(p), 3)
round(p.adjust(p, "BH"), 3)

## or all of them at once (dropping the "fdr" alias):
p.adjust.M <- p.adjust.methods[p.adjust.methods != "fdr"]
p.adj    <- sapply(p.adjust.M, function(meth) p.adjust(p, meth))
p.adj.60 <- sapply(p.adjust.M, function(meth) p.adjust(p, meth, n = 60))
stopifnot(identical(p.adj[,"none"], p), p.adj <= p.adj.60)
round(p.adj, 3)
## or a bit nicer:
noquote(apply(p.adj, 2, format.pval, digits = 3))


## and a graphic:
matplot(p, p.adj, ylab="p.adjust(p, meth)", type = "l", asp = 1, lty = 1:6,
        main = "P-value adjustments")
legend(0.7, 0.6, p.adjust.M, col = 1:6, lty = 1:6)

## Can work with NA's:
pN <- p; iN <- c(46, 47); pN[iN] <- NA
pN.a <- sapply(p.adjust.M, function(meth) p.adjust(pN, meth))
## The smallest 20 P-values all affected by the NA's :
round((pN.a / p.adj)[1:20, ] , 4)
```

`Pair` *Construct a Paired-Data Object*

Description

Combines two vectors into an object of class `"Pair"`.

Usage

```
Pair(x, y)
```

Arguments

x	a vector, the 1st element of the pair.
y	a vector, the 2nd element of the pair. Should have the same length as x.

Value

A 2-column matrix of class "Pair".

Note

Mostly designed as part of the formula interface to paired tests.

See Also

t.test and wilcox.test

---

pairwise.prop.test *Pairwise comparisons for proportions*

Description

Calculate pairwise comparisons between pairs of proportions with correction for multiple testing

Usage

```
pairwise.prop.test(x, n, p.adjust.method = p.adjust.methods, ...)
```

Arguments

x	Vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
n	Vector of counts of trials; ignored if x is a matrix.
p.adjust.method	Method for adjusting p values (see p.adjust). Can be abbreviated.
...	Additional arguments to pass to prop.test

Value

Object of class "pairwise.htest"

See Also

prop.test, p.adjust

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
pairwise.prop.test(smokers, patients)
```

`pairwise.t.test` *Pairwise t tests*

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing

Usage

```
pairwise.t.test(x, g, p.adjust.method = p.adjust.methods,
                pool.sd = !paired, paired = FALSE,
                alternative = c("two.sided", "less", "greater"),
                ...)
```

Arguments

`x`	response vector.
`g`	grouping vector or factor.
`p.adjust.method`	Method for adjusting p values (see `p.adjust`).
`pool.sd`	switch to allow/disallow the use of a pooled SD
`paired`	a logical indicating whether you want paired t-tests.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. Can be abbreviated.
`...`	additional arguments to pass to `t.test`.

Details

The `pool.sd` switch calculates a common SD for all groups and uses that for all comparisons (this can be useful if some groups are small). This method does not actually call `t.test`, so extra arguments are ignored. Pooling does not generalize to paired tests so `pool.sd` and `paired` cannot both be `TRUE`.

Only the lower triangle of the matrix of possible comparisons is being calculated, so setting `alternative` to anything other than `"two.sided"` requires that the levels of `g` are ordered sensibly.

Value

Object of class `"pairwise.htest"`

See Also

t.test, p.adjust

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
pairwise.t.test(Ozone, Month)
pairwise.t.test(Ozone, Month, p.adjust.method = "bonf")
pairwise.t.test(Ozone, Month, pool.sd = FALSE)
detach()
```

---

pairwise.table *Tabulate p values for pairwise comparisons*

Description

Creates table of p values for pairwise comparisons with corrections for multiple testing.

Usage

```
pairwise.table(compare.levels, level.names, p.adjust.method)
```

Arguments

compare.levels	a function to compute (raw) p value given indices i and j.
level.names	names of the group levels
p.adjust.method	a character string specifying the method for multiple testing adjustment; almost always one of p.adjust.methods. Can be abbreviated.

Details

Functions that do multiple group comparisons create separate compare.levels functions (assumed to be symmetrical in i and j) and passes them to this function.

Value

Table of p values in lower triangular form.

See Also

pairwise.t.test

`pairwise.wilcox.test` *Pairwise Wilcoxon Rank Sum Tests*

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing.

Usage

```
pairwise.wilcox.test(x, g, p.adjust.method = p.adjust.methods,
                     paired = FALSE, ...)
```

Arguments

`x`	response vector.
`g`	grouping vector or factor.
`p.adjust.method`	method for adjusting p values (see `p.adjust`). Can be abbreviated.
`paired`	a logical indicating whether you want a paired test.
`...`	additional arguments to pass to `wilcox.test`.

Details

Extra arguments that are passed on to `wilcox.test` may or may not be sensible in this context. In particular, only the lower triangle of the matrix of possible comparisons is being calculated, so setting `alternative` to anything other than `"two.sided"` requires that the levels of `g` are ordered sensibly.

Value

Object of class `"pairwise.htest"`

See Also

`wilcox.test`, `p.adjust`

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
## These give warnings because of ties :
pairwise.wilcox.test(Ozone, Month)
pairwise.wilcox.test(Ozone, Month, p.adjust.method = "bonf")
detach()
```

plot.acf *Plot Autocovariance and Autocorrelation Functions*

Description

Plot method for objects of class "acf".

Usage

```
## S3 method for class 'acf'
plot(x, ci = 0.95, type = "h", xlab = "Lag", ylab = NULL,
     ylim = NULL, main = NULL,
     ci.col = "blue", ci.type = c("white", "ma"),
     max.mfrow = 6, ask = Npgs > 1 && dev.interactive(),
     mar = if(nser > 2) c(3,2,2,0.8) else par("mar"),
     oma = if(nser > 2) c(1,1.2,1,1) else par("oma"),
     mgp = if(nser > 2) c(1.5,0.6,0) else par("mgp"),
     xpd = par("xpd"),
     cex.main = if(nser > 2) 1 else par("cex.main"),
     verbose = getOption("verbose"),
     ...)
```

Arguments

x	an object of class "acf".
ci	coverage probability for confidence interval. Plotting of the confidence interval is suppressed if ci is zero or negative.
type	the type of plot to be drawn, default to histogram like vertical lines.
xlab	the x label of the plot.
ylab	the y label of the plot.
ylim	numeric of length 2 giving the y limits for the plot.
main	overall title for the plot.
ci.col	colour to plot the confidence interval lines.
ci.type	should the confidence limits assume a white noise input or for lag k an MA($k-1$) input? Can be abbreviated.
max.mfrow	positive integer; for multivariate x indicating how many rows and columns of plots should be put on one page, using par(mfrow = c(m,m)).
ask	logical; if TRUE, the user is asked before a new page is started.
mar, oma, mgp, xpd, cex.main	graphics parameters as in par(*), by default adjusted to use smaller than default margins for multivariate x only.
verbose	logical. Should R report extra information on progress?
...	graphics parameters to be passed to the plotting routines.

Note

The confidence interval plotted in `plot.acf` is based on an *uncorrelated* series and should be treated with appropriate caution. Using `ci.type = "ma"` may be less potentially misleading.

See Also

`acf` which calls `plot.acf` by default.

Examples

```
require(graphics)

z4  <- ts(matrix(rnorm(400), 100, 4), start = c(1961, 1), frequency = 12)
z7  <- ts(matrix(rnorm(700), 100, 7), start = c(1961, 1), frequency = 12)
acf(z4)
acf(z7, max.mfrow = 7)   # squeeze onto 1 page
acf(z7) # multi-page
```

---

plot.density *Plot Method for Kernel Density Estimation*

Description

The `plot` method for density objects.

Usage

```
## S3 method for class 'density'
plot(x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
     zero.line = TRUE, ...)
```

Arguments

`x`	a `"density"` object.
`main`, `xlab`, `ylab`, `type`	plotting parameters with useful defaults.
`...`	further plotting parameters.
`zero.line`	logical; if `TRUE`, add a base line at $y = 0$

Value

None.

See Also

`density`.

plot.HoltWinters *Plot function for* "HoltWinters" *objects*

Description

Produces a chart of the original time series along with the fitted values. Optionally, predicted values (and their confidence bounds) can also be plotted.

Usage

```
## S3 method for class 'HoltWinters'
plot(x, predicted.values = NA, intervals = TRUE,
        separator = TRUE, col = 1, col.predicted = 2,
        col.intervals = 4, col.separator = 1, lty = 1,
        lty.predicted = 1, lty.intervals = 1, lty.separator = 3,
        ylab = "Observed / Fitted",
        main = "Holt-Winters filtering",
        ylim = NULL, ...)
```

Arguments

x	Object of class "HoltWinters"
predicted.values	Predicted values as returned by predict.HoltWinters
intervals	If TRUE, the prediction intervals are plotted (default).
separator	If TRUE, a separating line between fitted and predicted values is plotted (default).
col, lty	Color/line type of original data (default: black solid).
col.predicted, lty.predicted	Color/line type of fitted and predicted values (default: red solid).
col.intervals, lty.intervals	Color/line type of prediction intervals (default: blue solid).
col.separator, lty.separator	Color/line type of observed/predicted values separator (default: black dashed).
ylab	Label of the y-axis.
main	Main title.
ylim	Limits of the y-axis. If NULL, the range is chosen such that the plot contains the original series, the fitted values, and the predicted values if any.
...	Other graphics parameters.

Author(s)

David Meyer <David.Meyer@wu.ac.at>

References

C. C. Holt (1957) Forecasting trends and seasonals by exponentially weighted moving averages, *ONR Research Memorandum, Carnegie Institute of Technology* **52**.

P. R. Winters (1960). Forecasting sales by exponentially weighted moving averages. *Management Science*, **6**, 324–342. doi:10.1287/mnsc.6.3.324.

See Also

HoltWinters, predict.HoltWinters

`plot.isoreg` *Plot Method for* `isoreg` *Objects*

Description

The plot and lines method for R objects of class isoreg.

Usage

```
## S3 method for class 'isoreg'
plot(x, plot.type = c("single", "row.wise", "col.wise"),
      main = paste("Isotonic regression", deparse(x$call)),
      main2 = "Cumulative Data and Convex Minorant",
      xlab = "x0", ylab = "x$y",
      par.fit = list(col = "red", cex = 1.5, pch = 13, lwd = 1.5),
      mar = if (both) 0.1 + c(3.5, 2.5, 1, 1) else par("mar"),
      mgp = if (both) c(1.6, 0.7, 0) else par("mgp"),
      grid = length(x$x) < 12, ...)

## S3 method for class 'isoreg'
lines(x, col = "red", lwd = 1.5,
       do.points = FALSE, cex = 1.5, pch = 13, ...)
```

Arguments

x	an isoreg object.
plot.type	character indicating which type of plot is desired. The first (default) only draws the data and the fit, where the others add a plot of the cumulative data and fit. Can be abbreviated.
main	main title of plot, see title.
main2	title for second (cumulative) plot.
xlab, ylab	x- and y- axis annotation.
par.fit	a list of arguments (for points and lines) for drawing the fit.
mar, mgp	graphical parameters, see par, mainly for the case of two plots.

grid	logical indicating if grid lines should be drawn. If true, grid() is used for the first plot, where as vertical lines are drawn at 'touching' points for the cumulative plot.
do.points	for lines(): logical indicating if the step points should be drawn as well (and as they are drawn in plot()).
col, lwd, cex, pch	graphical arguments for lines(), where cex and pch are only used when do.points is TRUE.
...	further arguments passed to and from methods.

See Also

isoreg for computation of isoreg objects.

Examples

```
require(graphics)

utils::example(isoreg) # for the examples there

plot(y3, main = "simple plot(.)  +  lines(<isoreg>)")
lines(ir3)

## 'same' plot as above, "proving" that only ranks of 'x' are important
plot(isoreg(2^(1:9), c(1,0,4,3,3,5,4,2,0)), plot.type = "row", log = "x")

plot(ir3, plot.type = "row", ylab = "y3")
plot(isoreg(y3 - 4), plot.type = "r", ylab = "y3 - 4")
plot(ir4, plot.type = "ro",  ylab = "y4", xlab = "x = 1:n")

## experiment a bit with these (C-c C-j):
plot(isoreg(sample(9),  y3), plot.type = "row")
plot(isoreg(sample(9),  y3), plot.type = "col.wise")

plot(ir <- isoreg(sample(10), sample(10, replace = TRUE)),
                  plot.type = "r")
```

---

plot.lm *Plot Diagnostics for an* lm *Object*

Description

Six plots (selectable by which) are currently available: a plot of residuals against fitted values, a Scale-Location plot of $\sqrt{|residuals|}$ against fitted values, a Q-Q plot of residuals, a plot of Cook's distances versus row labels, a plot of residuals against leverages, and a plot of Cook's distances against leverage/(1-leverage). By default, the first three and 5 are provided.

Usage

```
## S3 method for class 'lm'
plot(x, which = c(1,2,3,5),
     caption = list("Residuals vs Fitted", "Q-Q Residuals",
       "Scale-Location", "Cook's distance",
       "Residuals vs Leverage",
       expression("Cook's dist vs Leverage* " * h[ii] / (1 - h[ii]))),
     panel = if(add.smooth) function(x, y, ...)
              panel.smooth(x, y, iter=iter.smooth, ...) else points,
     sub.caption = NULL, main = "",
     ask = prod(par("mfcol")) < length(which) && dev.interactive(),
     ...,
     id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
     qqline = TRUE, cook.levels = c(0.5, 1.0),
     cook.col = 8, cook.lty = 2, cook.legendChanges = list(),
     add.smooth = getOption("add.smooth"),
     iter.smooth = if(isGlm) 0 else 3,
     label.pos = c(4,2),
     cex.caption = 1, cex.oma.main = 1.25
   , extend.ylim.f = 0.08
     )
```

Arguments

x	lm object, typically result of lm or glm.
which	a subset of the numbers 1:6, by default 1:3, 5, referring to 1. "Residuals vs Fitted", aka 'Tukey-Anscombe' plot 2. "Residual Q-Q" plot 3. "Scale-Location" 4. "Cook's distance" 5. "Residuals vs Leverage" 6. "Cook's dist vs Lev./(1-Lev.)" See also 'Details' below.
caption	captions to appear above the plots; character vector or list of valid graphics annotations, see as.graphicsAnnot, of length 6, the j-th entry corresponding to which[j], see also the default vector in 'Usage'. Can be set to "" or NA to suppress all captions.
panel	panel function. The useful alternative to points, panel.smooth can be chosen by add.smooth = TRUE.
sub.caption	common title—above the figures if there are more than one; used as sub (s.title) otherwise. If NULL, as by default, a possible abbreviated version of deparse(x$call) is used.
main	title to each plot—in addition to caption.
ask	logical; if TRUE, the user is *ask*ed before each plot, see par(ask=.).
...	other parameters to be passed through to plotting functions.

id.n	number of points to be labelled in each plot, starting with the most extreme.
labels.id	vector of labels, from which the labels for extreme points will be chosen. NULL uses observation numbers.
cex.id	magnification of point labels.
qqline	logical indicating if a qqline() should be added to the normal Q-Q plot.
cook.levels	levels of Cook's distance at which to draw contours.
cook.col, cook.lty	color and line type to use for these contour lines.
cook.legendChanges	a list (or NULL to suppress the call) of arguments to legend which should be *modified* from (or added to) the plot.lm() default list(x = "bottomleft", legend = "Cook's distance", lty = cook.lty, col = cook.col, text.col = cook.col, bty = "n", x.intersp = 1/4, y.intersp = 1/8) .
add.smooth	logical indicating if a smoother should be added to most plots; see also panel above.
iter.smooth	the number of robustness iterations, the argument iter in panel.smooth(); the default uses no such iterations for glm fits which is particularly desirable for the (predominant) case of binary observations, but also for other models where the response distribution can be highly skewed.
label.pos	positioning of labels, for the left half and right half of the graph respectively, for plots 1-3, 5, 6.
cex.caption	controls the size of caption.
cex.oma.main	controls the size of the sub.caption only if that is *above* the figures when there is more than one.
extend.ylim.f	a numeric vector of length 1 or 2, to be used in ylim <- extendrange(r=ylim, f = *) for plots 1 and 5 when id.n is non-empty.

Details

sub.caption—by default the function call—is shown as a subtitle (under the x-axis title) on each plot when plots are on separate pages, or as a subtitle in the outer margin (if any) when there are multiple plots per page.

The 'Scale-Location' plot (which=3), also called 'Spread-Location' or 'S-L' plot, takes the square root of the absolute residuals in order to diminish skewness ($\sqrt{|E|}$ is much less skewed than $|E|$ for Gaussian zero-mean E).

The 'S-L', the Q-Q, and the Residual-Leverage (which=5) plot use *standardized* residuals which have identical variance (under the hypothesis). They are given as $R_i/(s \times \sqrt{1 - h_{ii}})$ where the *'leverages'* h_{ii} are the diagonal entries of the hat matrix, influence()$hat (see also hat), and where the Residual-Leverage plot uses the standardized Pearson residuals (residuals.glm(type = "pearson")) for $R[i]$.

The Residual-Leverage plot (which=5) shows contours of equal Cook's distance, for values of cook.levels (by default 0.5 and 1) and omits cases with leverage one with a warning. If the leverages are constant (as is typically the case in a balanced aov situation) the plot uses factor level combinations instead of the leverages for the x-axis. (The factor levels are ordered by mean fitted value.)

In the Cook's distance vs leverage/(1-leverage) (= "leverage*") plot (`which=6`), contours of standardized residuals (`rstandard(.)`) that are equal in magnitude are lines through the origin. These lines are labelled with the magnitudes. The x-axis is labeled with the (non equidistant) leverages h_{ii}.

For the `glm` case, the Q-Q plot is based on the absolute value of the standardized deviance residuals. When the saddlepoint approximation applies, these have an approximate half-normal distribution. The saddlepoint approximation is exact for the normal and inverse Gaussian family, and holds approximately for the Gamma family with small dispersion (large shape) and for the Poisson and binomial families with large counts (Dunn and Smyth 2018).

Author(s)

John Maindonald and Martin Maechler.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980). *Regression Diagnostics*. New York: Wiley.

Cook, R. D. and Weisberg, S. (1982). *Residuals and Influence in Regression*. London: Chapman and Hall.

Firth, D. (1991) Generalized Linear Models. In Hinkley, D. V. and Reid, N. and Snell, E. J., eds: Pp. 55-82 in Statistical Theory and Modelling. In Honour of Sir David Cox, FRS. London: Chapman and Hall.

Hinkley, D. V. (1975). On power transformations to symmetry. *Biometrika*, **62**, 101–111. doi:10.2307/2334491.

McCullagh, P. and Nelder, J. A. (1989). *Generalized Linear Models*. London: Chapman and Hall.

Dunn, P.K. and Smyth G.K. (2018) *Generalized Linear Models with Examples in R*. New York: Springer-Verlag.

See Also

`termplot`, `lm.influence`, `cooks.distance`, `hatvalues`.

Examples

```
require(graphics)

## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
plot(lm.SR)

## 4 plots on 1 page;
## allow room for printing model formula in outer margin:
par(mfrow = c(2, 2), oma = c(0, 0, 2, 0)) -> opar
plot(lm.SR)
plot(lm.SR, id.n = NULL)                # no id's
plot(lm.SR, id.n = 5, labels.id = NULL)  # 5 id numbers

## Was default in R <= 2.1.x:
```

```
## Cook's distances instead of Residual-Leverage plot
plot(lm.SR, which = 1:4)

## All the above fit a smooth curve where applicable
## by default unless "add.smooth" is changed.
## Give a smoother curve by increasing the lowess span :
plot(lm.SR, panel = function(x, y) panel.smooth(x, y, span = 1))

par(mfrow = c(2,1)) # same oma as above
plot(lm.SR, which = 1:2, sub.caption = "Saving Rates, n=50, p=5")

## Cook's distance tweaking
par(mfrow = c(2,3)) # same oma ...
plot(lm.SR, which = 1:6, cook.col = "royalblue")

## A case where over plotting of the "legend" is to be avoided:
if(dev.interactive(TRUE)) getOption("device")(height = 6, width = 4)
par(mfrow = c(3,1), mar = c(5,5,4,2)/2 +.1, mgp = c(1.4, .5, 0))
plot(lm.SR, which = 5, extend.ylim.f = c(0.2, 0.08))
plot(lm.SR, which = 5, cook.lty = "dotdash",
     cook.legendChanges = list(x = "bottomright", legend = "Cook"))
plot(lm.SR, which = 5, cook.legendChanges = NULL)  # no "legend"


par(opar) # reset par()s
```

---

plot.ppr *Plot Ridge Functions for Projection Pursuit Regression Fit*

Description

Plot the ridge functions for a projection pursuit regression (ppr) fit.

Usage

```
## S3 method for class 'ppr'
plot(x, ask, type = "o", cex = 1/2,
     main = quote(bquote(
         "term"[.(i)]*":" ~~ hat(beta[.(i)]) == .(bet.i))),
     xlab = quote(bquote(bold(alpha)[.(i)]^T * bold(x))),
     ylab = "", ...)
```

Arguments

x	an R object of class "ppr" as produced by a call to ppr.
ask	the graphics parameter ask: see par for details. If set to TRUE will ask between the plot of each cross-section.
type	the type of line (see plot.default) to draw.

cex	plot symbol expansion factor (*relative* to par("cex")).
main, xlab, ylab	axis annotations, see also title. Can be an expression (depending on i and bet.i), as by default which will be eval()uated.
...	further graphical parameters, passed to plot().

Value

None

Side Effects

A series of plots are drawn on the current graphical device, one for each term in the fit.

See Also

ppr, par

Examples

```
require(graphics)

rock1 <- within(rock, { area1 <- area/10000; peri1 <- peri/10000 })
par(mfrow = c(3,2)) # maybe: , pty = "s"
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock1, nterms = 2, max.terms = 5)
plot(rock.ppr, main = "ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass = 5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method = "gcv", gcvpen = 2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
```

---

plot.profile *Plotting Functions for 'profile' Objects*

Description

plot and pairs methods for objects of class "profile".

Usage

```
## S3 method for class 'profile'
plot(x, ...)
## S3 method for class 'profile'
pairs(x, colours = 2:3, which = names(x), ...)
```

Arguments

x	an object inheriting from class "profile".
colours	colours to be used for the mean curves conditional on x and y respectively.
which	names or number of parameters in pairs plot
...	arguments passed to or from other methods.

Details

This is the main plot method for objects created by profile.glm. It can also be called on objects created by profile.nls, but they have a specific method, plot.profile.nls.

The pairs method shows, for each pair of parameters x and y, two curves intersecting at the maximum likelihood estimate, which give the loci of the points at which the tangents to the contours of the bivariate profile likelihood become vertical and horizontal, respectively. In the case of an exactly bivariate normal profile likelihood, these two curves would be straight lines giving the conditional means of y|x and x|y, and the contours would be exactly elliptical. The which argument allows you to select a subset of parameters; the default corresponds to the set of parameters that have been profiled.

Author(s)

Originally, D. M. Bates and W. N. Venables for S (in 1996). Taken from **MASS** where these functions were re-written by B. D. Ripley for R (by 1998).

See Also

profile.glm, profile.nls.

Examples

```
## see ?profile.glm for another example using glm fits.

## a version of example(profile.nls) from R >= 2.8.0
fm1 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
pr1 <- profile(fm1, alphamax = 0.1)
stats:::plot.profile(pr1) ## override dispatch to plot.profile.nls
pairs(pr1) # a little odd since the parameters are highly correlated

## an example from ?nls
x <- -(1:100)/10
y <- 100 + 10 * exp(x / 2) + rnorm(x)/10
nlmod <- nls(y ~  Const + A * exp(B * x), start=list(Const=100, A=10, B=1))
pairs(profile(nlmod))

## example from Dobson (1990) (see ?glm)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
## this example is only formally a Poisson model. It is really a
## comparison of 3 multinomials. Only the interaction parameters are of
```

```
## interest.
glm.D93i <- glm(counts ~ outcome * treatment, family = poisson())
pr1 <- profile(glm.D93i)
pr2 <- profile(glm.D93i, which=6:9)
plot(pr1)
plot(pr2)
pairs(pr1)
pairs(pr2)
```

`plot.profile.nls` *Plot a* `profile.nls` *Object*

Description

Displays a series of plots of the profile t function and interpolated confidence intervals for the parameters in a nonlinear regression model that has been fit with `nls` and profiled with `profile.nls`.

Usage

```
## S3 method for class 'profile.nls'
plot(x, levels, conf = c(99, 95, 90, 80, 50)/100,
     absVal = TRUE, ylab = NULL, lty = 2, ...)
```

Arguments

`x`	an object of class "`profile.nls`"
`levels`	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually `conf` is used instead of giving `levels` explicitly.
`conf`	a numeric vector of confidence levels for profile-based confidence intervals on the parameters. Defaults to `c(0.99, 0.95, 0.90, 0.80, 0.50)`.
`absVal`	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to `TRUE`.
`lty`	the line type to be used for axis and dropped lines.
`ylab, ...`	other arguments to the `plot.default` function can be passed here (but not `xlab`, `xlim`, `ylim` nor `type`).

Details

The plots are produced in a set of hard-coded colours, but as these are coded by number their effect can be changed by setting the `palette`. Colour 1 is used for the axes and 4 for the profile itself. Colours 3 and 6 are used for the axis line at zero and the horizontal/vertical lines dropping to the axes.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6)

See Also

nls, profile, profile.nls

Examples

```
require(graphics)

# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
# get the profile for the fitted model
pr1 <- profile(fm1, alphamax = 0.05)
opar <- par(mfrow = c(2,2), oma = c(1.1, 0, 1.1, 0), las = 1)
plot(pr1, conf = c(95, 90, 80, 50)/100)
plot(pr1, conf = c(95, 90, 80, 50)/100, absVal = FALSE)
mtext("Confidence intervals based on the profile sum of squares",
      side = 3, outer = TRUE)
mtext("BOD data - confidence levels of 50%, 80%, 90% and 95%",
      side = 1, outer = TRUE)
par(opar)
```

---

plot.spec *Plotting Spectral Densities*

Description

Plotting method for objects of class "spec". For multivariate time series it plots the marginal spectra of the series or pairs plots of the coherency and phase of the cross-spectra.

Usage

```
## S3 method for class 'spec'
plot(x, add = FALSE, ci = 0.95, log = c("yes", "dB", "no"),
     xlab = "frequency", ylab = NULL, type = "l",
     ci.col = "blue", ci.lty = 3,
     main = NULL, sub = NULL,
     plot.type = c("marginal", "coherency", "phase"),
     ...)

plot.spec.phase(x, ci = 0.95,
                xlab = "frequency", ylab = "phase",
                ylim = c(-pi, pi), type = "l",
                main = NULL, ci.col = "blue", ci.lty = 3, ...)
```

```
plot.spec.coherency(x, ci = 0.95,
                    xlab = "frequency",
                    ylab = "squared coherency",
                    ylim = c(0, 1), type = "l",
                    main = NULL, ci.col = "blue", ci.lty = 3, ...)
```

Arguments

x	an object of class "spec".
add	logical. If TRUE, add to already existing plot. Only valid for plot.type = "marginal".
ci	coverage probability for confidence interval. Plotting of the confidence bar/limits is omitted unless ci is strictly positive.
log	If "dB", plot on log10 (decibel) scale, otherwise use conventional log scale or linear scale. Logical values are also accepted. The default is "yes" unless options(ts.S.compat = TRUE) has been set, when it is "dB". Only valid for plot.type = "marginal".
xlab	the x label of the plot.
ylab	the y label of the plot. If missing a suitable label will be constructed.
type	the type of plot to be drawn, defaults to lines.
ci.col	colour for plotting confidence bar or confidence intervals for coherency and phase.
ci.lty	line type for confidence intervals for coherency and phase.
main	overall title for the plot. If missing, a suitable title is constructed.
sub	a subtitle for the plot. Only used for plot.type = "marginal". If missing, a description of the smoothing is used.
plot.type	For multivariate time series, the type of plot required. Only the first character is needed.
ylim, ...	Graphical parameters.

See Also

spectrum

---

plot.stepfun *Plot Step Functions*

Description

Method of the generic plot for stepfun objects and utility for plotting piecewise constant functions.

Usage

```
## S3 method for class 'stepfun'
plot(x, xval, xlim, ylim = range(c(y, Fn.kn)),
     xlab = "x", ylab = "f(x)", main = NULL,
     add = FALSE, verticals = TRUE, do.points = (n < 1000),
     pch = par("pch"), col = par("col"),
     col.points = col, cex.points = par("cex"),
     col.hor = col, col.vert = col,
     lty = par("lty"), lwd = par("lwd"), ...)

## S3 method for class 'stepfun'
lines(x, ...)
```

Arguments

x	an R object inheriting from "stepfun".
xval	numeric vector of abscissa values at which to evaluate x. Defaults to knots(x) restricted to xlim.
xlim, ylim	limits for the plot region: see plot.window. Both have sensible defaults if omitted.
xlab, ylab	labels for x and y axis.
main	main title.
add	logical; if TRUE only *add* to an existing plot.
verticals	logical; if TRUE, draw vertical lines at steps.
do.points	logical; if TRUE, also draw points at the (xlim restricted) knot locations. Default is true, for sample size < 1000.
pch	character; point character if do.points.
col	default color of all points and lines.
col.points	character or integer code; color of points if do.points.
cex.points	numeric; character expansion factor if do.points.
col.hor	color of horizontal lines.
col.vert	color of vertical lines.
lty, lwd	line type and thickness for all lines.
...	further arguments of plot(.), or if(add) segments(.).

Value

A list with two components

t	abscissa (x) values, including the two outermost ones.
y	y values 'in between' the t[].

Author(s)

Martin Maechler <maechler@stat.math.ethz.ch>, 1990, 1993; ported to R, 1997.

See Also

ecdf for empirical distribution functions as special step functions, approxfun and splinefun.

Examples

```
require(graphics)

y0 <- c(1,2,4,3)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1  <- stepfun(1:3, y0, right = TRUE)

tt <- seq(0, 3, by = 0.1)
op <- par(mfrow = c(2,2))
plot(sfun0); plot(sfun0, xval = tt, add = TRUE, col.hor = "bisque")
plot(sfun.2);plot(sfun.2, xval = tt, add = TRUE, col = "orange") # all colors
plot(sfun1);lines(sfun1, xval = tt, col.hor = "coral")
##-- This is  revealing :
plot(sfun0, verticals = FALSE,
     main = "stepfun(x, y0, f=f)  for f = 0, .2, 1")
for(i in 1:3)
  lines(list(sfun0, sfun.2, stepfun(1:3, y0, f = 1))[[i]], col = i)
legend(2.5, 1.9, paste("f =", c(0, 0.2, 1)), col = 1:3, lty = 1, y.intersp = 1)
par(op)

# Extend and/or restrict 'viewport':
plot(sfun0, xlim = c(0,5), ylim = c(0, 3.5),
     main = "plot(stepfun(*), xlim= . , ylim = .)")

##-- this works too (automatic call to  ecdf(.)):
plot.stepfun(rt(50, df = 3), col.vert = "gray20")
```

---

plot.ts *Plotting Time-Series Objects*

Description

Plotting method for objects inheriting from class "ts".

Usage

```
## S3 method for class 'ts'
plot(x, y = NULL, plot.type = c("multiple", "single"),
        xy.labels, xy.lines, panel = lines, nc, yax.flip = FALSE,
        mar.multi = c(0, 5.1, 0, if(yax.flip) 5.1 else 2.1),
        oma.multi = c(6, 0, 5, 0), axes = TRUE, ...)

## S3 method for class 'ts'
lines(x, ...)
```

Arguments

x, y	time series objects, usually inheriting from class "ts".
plot.type	for multivariate time series, should the series by plotted separately (with a common time axis) or on a single plot? Can be abbreviated.
xy.labels	logical, indicating if text() labels should be used for an x-y plot, *or* character, supplying a vector of labels to be used. The default is to label for up to 150 points, and not for more.
xy.lines	logical, indicating if lines should be drawn for an x-y plot. Defaults to the value of xy.labels if that is logical, otherwise to TRUE.
panel	a function(x, col, bg, pch, type, ...) which gives the action to be carried out in each panel of the display for plot.type = "multiple". The default is lines.
nc	the number of columns to use when type = "multiple". Defaults to 1 for up to 4 series, otherwise to 2.
yax.flip	logical indicating if the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series when type = "multiple".
mar.multi, oma.multi	the (default) par settings for plot.type = "multiple". Modify with care!
axes	logical indicating if x- and y- axes should be drawn.
...	additional graphical arguments, see plot, plot.default and par.

Details

If y is missing, this function creates a time series plot, for multivariate series of one of two kinds depending on plot.type.

If y is present, both x and y must be univariate, and a scatter plot y ~ x will be drawn, enhanced by using text if xy.labels is TRUE or character, and lines if xy.lines is TRUE.

See Also

ts for basic time series construction and access functionality.

Examples

```
require(graphics)

## Multivariate
z <- ts(matrix(rt(200 * 8, df = 3), 200, 8),
        start = c(1961, 1), frequency = 12)
plot(z, yax.flip = TRUE)
plot(z, axes = FALSE, ann = FALSE, frame.plot = TRUE,
     mar.multi = c(0,0,0,0), oma.multi = c(1,1,5,1))
title("plot(ts(..), axes=FALSE, ann=FALSE, frame.plot=TRUE, mar..., oma...)")

z <- window(z[,1:3], end = c(1969,12))
plot(z, type = "b")    # multiple
plot(z, plot.type = "single", lty = 1:3, col = 4:2)
```

```
## A phase plot:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col = "blue",
     main = "Lag plot of New Haven temperatures")

## xy.lines and xy.labels are FALSE for large series:
plot(lag(sunspots, 1), sunspots, pch = ".")

SMI <- EuStockMarkets[, "SMI"]
plot(lag(SMI,  1), SMI, pch = ".")
plot(lag(SMI, 20), SMI, pch = ".", log = "xy",
     main = "4 weeks lagged SMI stocks -- log scale", xy.lines =  TRUE)
```

`Poisson` *The Poisson Distribution*

Description

Density, distribution function, quantile function and random generation for the Poisson distribution with parameter `lambda`.

Usage

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

Arguments

`x`	vector of (non-negative integer) quantiles.
`q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of random values to return.
`lambda`	vector of (non-negative) means.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Poisson distribution has density

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

for $x = 0, 1, 2, \ldots$. The mean and variance are $E(X) = Var(X) = \lambda$.

Note that $\lambda = 0$ is really a limit case (setting $0^0 = 1$) resulting in a point mass at 0, see also the example.

If an element of x is not integer, the result of dpois is zero, with a warning. $p(x)$ is computed using Loader's algorithm, see the reference in dbinom.

The quantile is right continuous: qpois(p, lambda) is the smallest integer x such that $P(X \leq x) \geq p$.

Setting lower.tail = FALSE allows to get much more precise results when the default, lower.tail = TRUE would return 1, see the example below.

Value

dpois gives the (log) density, ppois gives the (log) distribution function, qpois gives the quantile function, and rpois generates random deviates.

Invalid lambda will result in return value NaN, with a warning.

The length of the result is determined by n for rpois, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

rpois returns a vector of type integer unless generated values exceed the maximum representable integer when double values are returned.

Source

dpois uses C code contributed by Catherine Loader (see dbinom).

ppois uses pgamma.

qpois uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rpois uses

Ahrens, J. H. and Dieter, U. (1982). Computer generation of Poisson deviates from modified normal distributions. *ACM Transactions on Mathematical Software*, **8**, 163–179.

See Also

Distributions for other standard distributions, including dbinom for the binomial and dnbinom for the negative binomial distribution.

poisson.test.

Examples

```
require(graphics)

-log(dpois(0:7, lambda = 1) * gamma(1+ 0:7)) # == 1
Ni <- rpois(50, lambda = 4); table(factor(Ni, 0:max(Ni)))

1 - ppois(10*(15:25), lambda = 100)  # becomes 0 (cancellation)
    ppois(10*(15:25), lambda = 100, lower.tail = FALSE)  # no cancellation

par(mfrow = c(2, 1))
x <- seq(-0.01, 5, 0.01)
```

```
plot(x, ppois(x, 1), type = "s", ylab = "F(x)", main = "Poisson(1) CDF")
plot(x, pbinom(x, 100, 0.01), type = "s", ylab = "F(x)",
     main = "Binomial(100, 0.01) CDF")

## The (limit) case  lambda = 0 :
stopifnot(identical(dpois(0,0), 1),
   identical(ppois(0,0), 1),
   identical(qpois(1,0), 0))
```

`poisson.test` *Exact Poisson tests*

Description

Performs an exact test of a simple null hypothesis about the rate parameter in Poisson distribution, or for the ratio between two rate parameters.

Usage

```
poisson.test(x, T = 1, r = 1,
    alternative = c("two.sided", "less", "greater"),
    conf.level = 0.95)
```

Arguments

`x`	number of events. A vector of length one or two.
`T`	time base for event count. A vector of length one or two.
`r`	hypothesized rate or rate ratio
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter.
`conf.level`	confidence level for the returned confidence interval.

Details

Confidence intervals are computed similarly to those of `binom.test` in the one-sample case, and using `binom.test` in the two sample case.

Value

A list with class `"htest"` containing the following components:

`statistic`	the number of events (in the first sample if there are two.)
`parameter`	the corresponding expected count
`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the rate or rate ratio.
`estimate`	the estimated rate or rate ratio.

null.value	the rate or rate ratio under the null, r.
alternative	a character string describing the alternative hypothesis.
method	the character string "Exact Poisson test" or "Comparison of Poisson rates" as appropriate.
data.name	a character string giving the names of the data.

Note

The rate parameter in Poisson data is often given based on a "time on test" or similar quantity (person-years, population size, or expected number of cases from mortality tables). This is the role of the T argument.

The one-sample case is effectively the binomial test with a very large n. The two sample case is converted to a binomial test by conditioning on the total event count, and the rate ratio is directly related to the odds in that binomial distribution.

See Also

binom.test

Examples

```
### These are paraphrased from data sets in the ISwR package

## SMR, Welsh Nickel workers
poisson.test(137, 24.19893)

## eba1977, compare Fredericia to other three cities for ages 55-59
poisson.test(c(11, 6+8+7), c(800, 1083+1050+878))
```

---

poly *Compute Orthogonal Polynomials*

Description

Returns or evaluates orthogonal polynomials of degree 1 to degree over the specified set of points x: these are all orthogonal to the constant polynomial of degree 0. Alternatively, evaluate raw polynomials.

Usage

```
poly(x, ..., degree = 1, coefs = NULL, raw = FALSE, simple = FALSE)
polym  (..., degree = 1, coefs = NULL, raw = FALSE)

## S3 method for class 'poly'
predict(object, newdata, ...)
```

Arguments

x, newdata	a numeric vector or an object with mode "numeric" (such as a Date) at which to evaluate the polynomial. x can also be a matrix. Missing values are not allowed in x.
degree	the degree of the polynomial. Must be less than the number of unique points when raw is false, as by default.
coefs	for prediction, coefficients from a previous fit.
raw	if true, use raw and not orthogonal polynomials.
simple	logical indicating if a simple matrix (with no further attributes but dimnames) should be returned. For speedup only.
object	an object inheriting from class "poly", normally the result of a call to poly with a single vector argument.
...	poly, polym: further vectors. predict.poly: arguments to be passed to or from other methods.

Details

Although formally degree should be named (as it follows ...), an unnamed second argument of length 1 will be interpreted as the degree, such that poly(x, 3) can be used in formulas.

The orthogonal polynomial is summarized by the coefficients, which can be used to evaluate it via the three-term recursion given in Kennedy & Gentle (1980, pp. 343–4), and used in the predict part of the code.

poly using ... is just a convenience wrapper for polym: coef is ignored. Conversely, if polym is called with a single argument in ... it is a wrapper for poly.

Value

For poly and polym() (when simple=FALSE and coefs=NULL as per default):
A matrix with rows corresponding to points in x and columns corresponding to the degree, with attributes "degree" specifying the degrees of the columns and (unless raw = TRUE) "coefs" which contains the centering and normalization constants used in constructing the orthogonal polynomials and class c("poly", "matrix").

For poly(*, simple=TRUE), polym(*, coefs=<non-NULL>), and predict.poly(): a matrix.

Note

This routine is intended for statistical purposes such as contr.poly: it does not attempt to orthogonalize to machine accuracy.

Author(s)

R Core Team. Keith Jewell (Campden BRI Group, UK) contributed improvements for correct prediction on subsets.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Kennedy, W. J. Jr and Gentle, J. E. (1980) *Statistical Computing* Marcel Dekker.

See Also

`contr.poly`.

`cars` for an example of polynomial regression.

Examples

```
od <- options(digits = 3) # avoid too much visual clutter
(z <- poly(1:10, 3))
predict(z, seq(2, 4, 0.5))
zapsmall(poly(seq(4, 6, 0.5), 3, coefs = attr(z, "coefs")))

 zm <- zapsmall(polym (    1:4, c(1, 4:6),  degree = 3)) # or just poly():
(z1 <- zapsmall(poly(cbind(1:4, c(1, 4:6)), degree = 3)))
## they are the same :
stopifnot(all.equal(zm, z1, tolerance = 1e-15))

## poly(<matrix>, df) --- used to fail till July 14 (vive la France!), 2017:
m2 <- cbind(1:4, c(1, 4:6))
pm2 <- zapsmall(poly(m2, 3)) # "unnamed degree = 3"
stopifnot(all.equal(pm2, zm, tolerance = 1e-15))

options(od)
```

`power` *Create a Power Link Object*

Description

Creates a link object based on the link function $\eta = \mu^\lambda$.

Usage

```
power(lambda = 1)
```

Arguments

`lambda`	a real number.

Details

If `lambda` is non-positive, it is taken as zero, and the log link is obtained. The default `lambda = 1` gives the identity link.

Value

A list with components `linkfun`, `linkinv`, `mu.eta`, and `valideta`. See `make.link` for information on their meaning.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`make.link`, `family`

To raise a number to a power, see `Arithmetic`.

To calculate the power of a test, see various functions in the **stats** package, e.g., `power.t.test`.

Examples

```
power()
quasi(link = power(1/3))[c("linkfun", "linkinv")]
```

---

power.anova.test *Power Calculations for Balanced One-Way Analysis of Variance Tests*

Description

Compute power of test or determine parameters to obtain target power.

Usage

```
power.anova.test(groups = NULL, n = NULL,
                 between.var = NULL, within.var = NULL,
                 sig.level = 0.05, power = NULL)
```

Arguments

`groups`	Number of groups
`n`	Number of observations (per group)
`between.var`	Between group variance
`within.var`	Within group variance
`sig.level`	Significance level (Type I error probability)
`power`	Power of test (1 minus Type II error probability)

Details

Exactly one of the parameters `groups`, `n`, `between.var`, `power`, `within.var`, and `sig.level` must be passed as NULL, and that parameter is determined from the others. Notice that `sig.level` has non-NULL default so NULL must be explicitly passed if you want it computed.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Claus Ekstrøm

See Also

anova, lm, uniroot

Examples

```
power.anova.test(groups = 4, n = 5, between.var = 1, within.var = 3)
# Power = 0.3535594

power.anova.test(groups = 4, between.var = 1, within.var = 3,
                 power = .80)
# n = 11.92613

## Assume we have prior knowledge of the group means:
groupmeans <- c(120, 130, 140, 150)
power.anova.test(groups = length(groupmeans),
                 between.var = var(groupmeans),
                 within.var = 500, power = .90) # n = 15.18834
```

---

power.prop.test *Power Calculations for Two-Sample Test for Proportions*

Description

Compute the power of the two-sample test for proportions, or determine parameters to obtain a target power.

Usage

```
power.prop.test(n = NULL, p1 = NULL, p2 = NULL, sig.level = 0.05,
                power = NULL,
                alternative = c("two.sided", "one.sided"),
                strict = FALSE, tol = .Machine$double.eps^0.25)
```

Arguments

n	number of observations (per group)
p1	probability in one group
p2	probability in other group
sig.level	significance level (Type I error probability)
power	power of test (1 minus Type II error probability)
alternative	one- or two-sided test. Can be abbreviated.
strict	use strict interpretation in two-sided case
tol	numerical tolerance used in root finding, the default providing (at least) four significant digits.

Details

Exactly one of the parameters n, p1, p2, power, and sig.level must be passed as NULL, and that parameter is determined from the others. Notice that sig.level has a non-NULL default so NULL must be explicitly passed if you want it computed.

If strict = TRUE is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Note that not all conditions can be satisfied, e.g., for

```
power.prop.test(n=30, p1=0.90, p2=NULL, power=0.8, strict=TRUE)
```

there is no proportion p2 between p1 = 0.9 and 1, as you'd need a sample size of at least $n = 74$ to yield the desired power for $(p1, p2) = (0.9, 1)$.

For these impossible conditions, currently a warning (warning) is signalled which may become an error (stop) in the future.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given. If one of p1 and p2 is computed, then $p1 < p2$ is assumed and will hold, but if you specify both, $p2 \leq p1$ is allowed.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

prop.test, uniroot

Examples

```
power.prop.test(n = 50, p1 = .50, p2 = .75)      ## => power = 0.740
power.prop.test(p1 = .50, p2 = .75, power = .90) ## =>     n = 76.7
power.prop.test(n = 50, p1 = .5, power = .90)    ## =>    p2 = 0.8026
power.prop.test(n = 50, p1 = .5, p2 = 0.9, power = .90, sig.level=NULL)
                                                 ## => sig.l = 0.00131
power.prop.test(p1 = .5, p2 = 0.501, sig.level=.001, power=0.90)
                                                 ## => n = 10451937
try(
 power.prop.test(n=30, p1=0.90, p2=NULL, power=0.8)
) # a warning  (which may become an error)
## Reason:
power.prop.test(      p1=0.90, p2= 1.0, power=0.8) ##-> n = 73.37
```

---

power.t.test *Power calculations for one and two sample t tests*

Description

Compute the power of the one- or two- sample t test, or determine parameters to obtain a target power.

Usage

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"),
             strict = FALSE, tol = .Machine$double.eps^0.25)
```

Arguments

n	number of observations (per group)
delta	true difference in means
sd	standard deviation
sig.level	significance level (Type I error probability)
power	power of test (1 minus Type II error probability)
type	string specifying the type of t test. Can be abbreviated.
alternative	one- or two-sided test. Can be abbreviated.
strict	use strict interpretation in two-sided case
tol	numerical tolerance used in root finding, the default providing (at least) four significant digits.

Details

Exactly one of the parameters `n`, `delta`, `power`, `sd`, and `sig.level` must be passed as `NULL`, and that parameter is determined from the others. Notice that the last two have non-NULL defaults, so NULL must be explicitly passed if you want to compute them.

If `strict = TRUE` is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Value

Object of class `"power.htest"`, a list of the arguments (including the computed one) augmented with `method` and `note` elements.

Note

`uniroot` is used to solve the power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

`t.test`, `uniroot`

Examples

```
power.t.test(n = 20, delta = 1)
power.t.test(power = .90, delta = 1)
power.t.test(power = .90, delta = 1, alternative = "one.sided")
```

PP.test *Phillips-Perron Test for Unit Roots*

Description

Computes the Phillips-Perron test for the null hypothesis that `x` has a unit root against a stationary alternative.

Usage

```
PP.test(x, lshort = TRUE)
```

Arguments

`x`	a numeric vector or univariate time series.
`lshort`	a logical indicating whether the short or long version of the truncation lag parameter is used.

Details

The general regression equation which incorporates a constant and a linear trend is used and the corrected t-statistic for a first order autoregressive coefficient equals one is computed. To estimate `sigma^2` the Newey-West estimator is used. If `lshort` is `TRUE`, then the truncation lag parameter is set to `trunc(4*(n/100)^0.25)`, otherwise `trunc(12*(n/100)^0.25)` is used. The p-values are interpolated from Table 4.2, page 103 of Banerjee et al. (1993).

Missing values are not handled.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the truncation lag parameter.
`p.value`	the p-value of the test.
`method`	a character string indicating what type of test was performed.
`data.name`	a character string giving the name of the data.

Author(s)

A. Trapletti

References

A. Banerjee, J. J. Dolado, J. W. Galbraith, and D. F. Hendry (1993). *Cointegration, Error Correction, and the Econometric Analysis of Non-Stationary Data*. Oxford University Press, Oxford.

P. Perron (1988). Trends and random walks in macroeconomic time series. *Journal of Economic Dynamics and Control*, **12**, 297–332. doi:10.1016/01651889(88)900437.

Examples

```
x <- rnorm(1000)
PP.test(x)
y <- cumsum(x) # has unit root
PP.test(y)
```

---

ppoints *Ordinates for Probability Plotting*

Description

Generates the sequence of probability points `(1:m - a)/(m + (1-a)-a)` where `m` is either `n`, if `length(n)==1`, or `length(n)`.

Usage

```
ppoints(n, a = if(n <= 10) 3/8 else 1/2)
```

Arguments

n	either the number of points generated or a vector of observations.
a	the offset fraction to be used; typically in $(0, 1)$.

Details

If $0 < a < 1$, the resulting values are within $(0, 1)$ (excluding boundaries). In any case, the resulting sequence is symmetric in $[0, 1]$, i.e., `p + rev(p) == 1`.

`ppoints()` is used in `qqplot` and `qqnorm` to generate the set of probabilities at which to evaluate the inverse distribution.

The choice of `a` follows the documentation of the function of the same name in Becker et al. (1988), and appears to have been motivated by results from Blom (1958) on approximations to expect normal order statistics (see also `quantile`).

The probability points for the continuous sample quantile types 5 to 9 (see `quantile`) can be obtained by taking `a` as, respectively, 1/2, 0, 1, 1/3, and 3/8.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Blom, G. (1958) *Statistical Estimates and Transformed Beta Variables.* Wiley

See Also

`qqplot`, `qqnorm`.

Examples

```
ppoints(4) # the same as  ppoints(1:4)
ppoints(10)
ppoints(10, a = 1/2)

## Visualize including the fractions :
require(graphics)
p.ppoints <- function(n, ..., add = FALSE, col = par("col")) {
  pn <- ppoints(n, ...)
  if(add)
      points(pn, pn, col = col)
  else {
      tit <- match.call(); tit[[1]] <- quote(ppoints)
      plot(pn,pn, main = deparse(tit), col=col,
           xlim = 0:1, ylim = 0:1, xaxs = "i", yaxs = "i")
      abline(0, 1, col = adjustcolor(1, 1/4), lty = 3)
  }
  if(!add && requireNamespace("MASS", quietly = TRUE))
    text(pn, pn, as.character(MASS::fractions(pn)),
         adj = c(0,0)-1/4, cex = 3/4, xpd = NA, col=col)
  abline(h = pn, v = pn, col = adjustcolor(col, 1/2), lty = 2, lwd = 1/2)
}
```

```
p.ppoints(4)
p.ppoints(10)
p.ppoints(10, a = 1/2)
p.ppoints(21)
p.ppoints(8) ; p.ppoints(8, a = 1/2, add=TRUE, col="tomato")
```

---

ppr *Projection Pursuit Regression*

Description

Fit a projection pursuit regression model.

Usage

```
ppr(x, ...)

## S3 method for class 'formula'
ppr(formula, data, weights, subset, na.action,
    contrasts = NULL, ..., model = FALSE)

## Default S3 method:
ppr(x, y, weights = rep(1, n),
    ww = rep(1, q), nterms, max.terms = nterms, optlevel = 2,
    sm.method = c("supsmu", "spline", "gcvspline"),
    bass = 0, span = 0, df = 5, gcvpen = 1, trace = FALSE, ...)
```

Arguments

formula	a formula specifying one or more numeric response variables and the explanatory variables.
x	numeric matrix of explanatory variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
y	numeric matrix of response variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
nterms	number of terms to include in the final model.
data	a data frame (or similar: see model.frame) from which variables specified in formula are preferentially to be taken.
weights	a vector of weights w_i for each *case*.
ww	a vector of weights for each *response*, so the fit criterion is the sum over case i and responses j of w_i ww_j (y_ij - fit_ij)^2 divided by the sum of w_i.
subset	an index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)

na.action	a function to specify the action to be taken if NAs are found. The default action is given by getOption("na.action"). (NOTE: If given, this argument must be named.)
contrasts	the contrasts to be used when any factor explanatory variables are coded.
max.terms	maximum number of terms to choose from when building the model.
optlevel	integer from 0 to 3 which determines the thoroughness of an optimization routine in the SMART program. See the 'Details' section.
sm.method	the method used for smoothing the ridge functions. The default is to use Friedman's super smoother supsmu. The alternatives are to use the smoothing spline code underlying smooth.spline, either with a specified (equivalent) degrees of freedom for each ridge functions, or to allow the smoothness to be chosen by GCV. Can be abbreviated.
bass	super smoother bass tone control used with automatic span selection (see supsmu); the range of values is 0 to 10, with larger values resulting in increased smoothing.
span	super smoother span control (see supsmu). The default, 0, results in automatic span selection by local cross validation. span can also take a value in (0, 1].
df	if sm.method is "spline" specifies the smoothness of each ridge term via the requested equivalent degrees of freedom.
gcvpen	if sm.method is "gcvspline" this is the penalty used in the GCV selection for each degree of freedom used.
trace	logical indicating if each spline fit should produce diagnostic output (about lambda and df), and the supsmu fit about its steps.
...	arguments to be passed to or from other methods.
model	logical. If true, the model frame is returned.

Details

The basic method is given by Friedman (1984) and based on his code. This code has been shown to be extremely sensitive to the Fortran compiler used.

The algorithm first adds up to max.terms ridge terms one at a time; it will use less if it is unable to find a term to add that makes sufficient difference. It then removes the least important term at each step until nterms terms are left.

The levels of optimization (argument optlevel) differ in how thoroughly the models are refitted during this process. At level 0 the existing ridge terms are not refitted. At level 1 the projection directions are not refitted, but the ridge functions and the regression coefficients are. Levels 2 and 3 refit all the terms and are equivalent for one response; level 3 is more careful to re-balance the contributions from each regressor at each step and so is a little less likely to converge to a saddle point of the sum of squares criterion.

Value

A list with the following components, many of which are for use by the method functions.

call	the matched call
p	the number of explanatory variables (after any coding)
q	the number of response variables
mu	the argument nterms
ml	the argument max.terms
gof	the overall residual (weighted) sum of squares for the selected model
gofn	the overall residual (weighted) sum of squares against the number of terms, up to max.terms. Will be invalid (and zero) for less than nterms.
df	the argument df
edf	if sm.method is "spline" or "gcvspline" the equivalent number of degrees of freedom for each ridge term used.
xnames	the names of the explanatory variables
ynames	the names of the response variables
alpha	a matrix of the projection directions, with a column for each ridge term
beta	a matrix of the coefficients applied for each response to the ridge terms: the rows are the responses and the columns the ridge terms
yb	the weighted means of each response
ys	the overall scale factor used: internally the responses are divided by ys to have unit total weighted sum of squares.
fitted.values	the fitted values, as a matrix if q > 1.
residuals	the residuals, as a matrix if q > 1.
smod	internal work array, which includes the ridge functions evaluated at the training set points.
model	(only if model = TRUE) the model frame.

Source

Friedman (1984): converted to double precision and added interface to smoothing splines by B. D. Ripley, originally for the **MASS** package.

References

Friedman, J. H. and Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American Statistical Association*, **76**, 817–823. doi:10.2307/2287576.

Friedman, J. H. (1984). SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer.

See Also

plot.ppr, supsmu, smooth.spline

Examples

```
require(graphics)

# Note: your numerical values may differ
attach(rock)
area1 <- area/10000; peri1 <- peri/10000
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock, nterms = 2, max.terms = 5)
rock.ppr
# Call:
# ppr.formula(formula = log(perm) ~ area1 + peri1 + shape, data = rock,
#     nterms = 2, max.terms = 5)
#
# Goodness of fit:
#  2 terms  3 terms  4 terms  5 terms
# 8.737806 5.289517 4.745799 4.490378

summary(rock.ppr)
# .....  (same as above)
# .....
#
# Projection direction vectors ('alpha'):
#       term 1      term 2
# area1  0.34357179  0.37071027
# peri1 -0.93781471 -0.61923542
# shape  0.04961846  0.69218595
#
# Coefficients of ridge terms:
#    term 1    term 2
# 1.6079271 0.5460971

par(mfrow = c(3,2))   # maybe: , pty = "s")
plot(rock.ppr, main = "ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass = 5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method = "gcv", gcvpen = 2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
cbind(perm = rock$perm, prediction = round(exp(predict(rock.ppr)), 1))
detach()
```

`prcomp` *Principal Components Analysis*

Description

Performs a principal components analysis on the given data matrix and returns the results as an object of class `prcomp`.

Usage

```
prcomp(x, ...)

## S3 method for class 'formula'
prcomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
prcomp(x, retx = TRUE, center = TRUE, scale. = FALSE,
       tol = NULL, rank. = NULL, ...)

## S3 method for class 'prcomp'
predict(object, newdata, ...)
```

Arguments

formula	a formula with no response variable, referring only to numeric variables.
data	an optional data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector used to select rows (observations) of the data matrix x.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit.
...	arguments passed to or from other methods. If x is a formula one might specify scale. or tol.
x	a numeric or complex matrix (or data frame) which provides the data for the principal components analysis.
retx	a logical value indicating whether the rotated variables should be returned.
center	a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
scale.	a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is FALSE for consistency with S, but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
tol	a value indicating the magnitude below which components should be omitted. (Components are omitted if their standard deviations are less than or equal to tol times the standard deviation of the first component.) With the default null setting, no components are omitted (unless rank. is specified less than min(dim(x)).). Other settings for tol could be tol = 0 or tol = sqrt(.Machine$double.eps), which would omit essentially constant components.
rank.	optionally, a number specifying the maximal rank, i.e., maximal number of principal components to be used. Can be set as alternative or in addition to tol, useful notably when the desired rank is considerably smaller than the dimensions of the matrix.

object	object of class inheriting from "prcomp"
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

The calculation is done by a singular value decomposition of the (centered and possibly scaled) data matrix, not by using eigen on the covariance matrix. This is generally the preferred method for numerical accuracy. The print method for these objects prints the results in a nice format and the plot method produces a scree plot.

Unlike princomp, variances are computed with the usual divisor $N - 1$.

Note that scale = TRUE cannot be used if there are zero or constant (for center = TRUE) variables.

Value

prcomp returns a list with class "prcomp" containing the following components:

sdev	the standard deviations of the principal components (i.e., the square roots of the eigenvalues of the covariance/correlation matrix, though the calculation is actually done with the singular values of the data matrix).
rotation	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). The function princomp returns this in the element loadings.
x	if retx is true the value of the rotated data (the centred (and scaled if requested) data multiplied by the rotation matrix) is returned. Hence, cov(x) is the diagonal matrix diag(sdev^2). For the formula method, napredict() is applied to handle the treatment of values omitted by the na.action.
center, scale	the centering and scaling used, or FALSE.

Note

The signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002) *Modern Applied Statistics with S*, Springer-Verlag.

See Also

biplot.prcomp, screeplot, princomp, cor, cov, svd, eigen.

Examples

```
C <- chol(S <- toeplitz(.9 ^ (0:31))) # Cov.matrix and its root
all.equal(S, crossprod(C))
set.seed(17)
X <- matrix(rnorm(32000), 1000, 32)
Z <- X %*% C  ## ==>  cov(Z) ~=  C'C = S
all.equal(cov(Z), S, tolerance = 0.08)
pZ <- prcomp(Z, tol = 0.1)
summary(pZ) # only ~14 PCs (out of 32)
## or choose only 3 PCs more directly:
pz3 <- prcomp(Z, rank. = 3)
summary(pz3) # same numbers as the first 3 above
stopifnot(ncol(pZ$rotation) == 14, ncol(pz3$rotation) == 3,
          all.equal(pz3$sdev, pZ$sdev, tolerance = 1e-15)) # exactly equal typically

## signs are random
require(graphics)
## the variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
prcomp(USArrests)  # inappropriate
prcomp(USArrests, scale. = TRUE)
prcomp(~ Murder + Assault + Rape, data = USArrests, scale. = TRUE)
plot(prcomp(USArrests))
summary(prcomp(USArrests, scale. = TRUE))
biplot(prcomp(USArrests, scale. = TRUE))
```

`predict` *Model Predictions*

Description

`predict` is a generic function for predictions from the results of various model fitting functions. The function invokes particular *methods* which depend on the `class` of the first argument.

Usage

```
predict (object, ...)
```

Arguments

`object`	a model object for which prediction is desired.
`...`	additional arguments affecting the predictions produced.

Details

Most prediction methods which are similar to those for linear models have an argument `newdata` specifying the first place to look for explanatory variables to be used for prediction. Some considerable attempts are made to match up the columns in `newdata` to those used for fitting, for example that they are of comparable types and that any factors have the same level set in the same order (or can be transformed to be so).

Time series prediction methods in package **stats** have an argument `n.ahead` specifying how many time steps ahead to predict.

Many methods have a logical argument `se.fit` saying if standard errors are to returned.

Value

The form of the value returned by `predict` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`predict.glm`, `predict.lm`, `predict.loess`, `predict.nls`, `predict.poly`, `predict.princomp`, `predict.smooth.spline`.

SafePrediction for prediction from (univariable) polynomial and spline fits.

For time-series prediction, `predict.ar`, `predict.Arima`, `predict.arima0`, `predict.HoltWinters`, `predict.StructTS`.

Examples

```
require(utils)

## All the "predict" methods found
## NB most of the methods in the standard packages are hidden.
## Output will depend on what namespaces are (or have been) loaded.

for(fn in methods("predict"))
   try({
       f <- eval(substitute(getAnywhere(fn)$objs[[1]], list(fn = fn)))
       cat(fn, ":\n\t", deparse(args(f)), "\n")
       }, silent = TRUE)
```

predict.Arima *Forecast from ARIMA fits*

Description

Forecast from models fitted by arima.

Usage

```
## S3 method for class 'Arima'
predict(object, n.ahead = 1, newxreg = NULL,
        se.fit = TRUE, ...)
```

Arguments

object	The result of an arima fit.
n.ahead	The number of steps ahead for which prediction is required.
newxreg	New values of xreg to be used for prediction. Must have at least n.ahead rows.
se.fit	Logical: should standard errors of prediction be returned?
...	arguments passed to or from other methods.

Details

Finite-history prediction is used, via KalmanForecast. This is only statistically efficient if the MA part of the fit is invertible, so predict.Arima will give a warning for non-invertible MA models.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients. According to Harvey (1993, pp. 58–9) the effect is small.

Value

A time series of predictions, or if se.fit = TRUE, a list with components pred, the predictions, and se, the estimated standard errors. Both components are time series.

References

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

Harvey, A. C. and McKenzie, C. R. (1982). Algorithm AS 182: An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics*, **31**, 180–187. doi:10.2307/2347987.

Harvey, A. C. (1993). *Time Series Models*, 2nd Edition. Harvester Wheatsheaf. Sections 3.3 and 4.4.

See Also

arima

Examples

```
od <- options(digits = 5) # avoid too much spurious accuracy
predict(arima(lh, order = c(3,0,0)), n.ahead = 12)

(fit <- arima(USAccDeaths, order = c(0,1,1),
              seasonal = list(order = c(0,1,1))))
predict(fit, n.ahead = 6)
options(od)
```

---

predict.glm *Predict Method for GLM Fits*

Description

Obtains predictions and optionally estimates standard errors of those predictions from a fitted generalized linear model object.

Usage

```
## S3 method for class 'glm'
predict(object, newdata = NULL,
            type = c("link", "response", "terms"),
            se.fit = FALSE, dispersion = NULL, terms = NULL,
            na.action = na.pass, ...)
```

Arguments

object	a fitted object of class inheriting from "glm".
newdata	optionally, a data frame in which to look for variables with which to predict. If omitted, the fitted linear predictors are used.
type	the type of prediction required. The default is on the scale of the linear predictors; the alternative "response" is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and type = "response" gives the predicted probabilities. The "terms" option returns a matrix giving the fitted values of each term in the model formula on the linear predictor scale. The value of this argument can be abbreviated.
se.fit	logical switch indicating if standard errors are required.
dispersion	the dispersion of the GLM fit to be assumed in computing the standard errors. If omitted, that returned by summary applied to the object is used.
terms	with type = "terms" by default all terms are returned. A character vector specifies which terms are to be returned
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
...	further arguments passed to or from other methods.

Details

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear (in predictions and standard errors), with residual value NA. See also napredict.

Value

If se.fit = FALSE, a vector or matrix of predictions. For type = "terms" this is a matrix with a column per term, and may have an attribute "constant".

If se.fit = TRUE, a list with components

fit	Predictions, as for se.fit = FALSE.
se.fit	Estimated standard errors.
residual.scale	A scalar giving the square root of the dispersion used in computing the standard errors.

Note

Variables are first looked for in newdata and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in newdata if it was supplied.

See Also

glm, SafePrediction

Examples

```
require(graphics)

## example from Venables and Ripley (2002, pp. 190-2.)
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive = 20-numdead)
budworm.lg <- glm(SF ~ sex*ldose, family = binomial)
summary(budworm.lg)

plot(c(1,32), c(0,1), type = "n", xlab = "dose",
     ylab = "prob", log = "x")
text(2^ldose, numdead/20, as.character(sex))
ld <- seq(0, 5, 0.1)
lines(2^ld, predict(budworm.lg, data.frame(ldose = ld,
   sex = factor(rep("M", length(ld)), levels = levels(sex))),
   type = "response"))
lines(2^ld, predict(budworm.lg, data.frame(ldose = ld,
   sex = factor(rep("F", length(ld)), levels = levels(sex))),
   type = "response"))
```

`predict.HoltWinters` *Prediction Function for Fitted Holt-Winters Models*

Description

Computes predictions and prediction intervals for models fitted by the Holt-Winters method.

Usage

```
## S3 method for class 'HoltWinters'
predict(object, n.ahead = 1, prediction.interval = FALSE,
        level = 0.95, ...)
```

Arguments

`object`	An object of class `HoltWinters`.
`n.ahead`	Number of future periods to predict.
`prediction.interval`	logical. If `TRUE`, the lower and upper bounds of the corresponding prediction intervals are computed.
`level`	Confidence level for the prediction interval.
`...`	arguments passed to or from other methods.

Value

A time series of the predicted values. If prediction intervals are requested, a multiple time series is returned with columns `fit`, `lwr` and `upr` for the predicted values and the lower and upper bounds respectively.

Author(s)

David Meyer <David.Meyer@wu.ac.at>

References

C. C. Holt (1957) Forecasting trends and seasonals by exponentially weighted moving averages, *ONR Research Memorandum, Carnegie Institute of Technology* **52**.

P. R. Winters (1960). Forecasting sales by exponentially weighted moving averages. *Management Science*, **6**, 324–342. doi:10.1287/mnsc.6.3.324.

See Also

`HoltWinters`

Examples

```
require(graphics)

m <- HoltWinters(co2)
p <- predict(m, 50, prediction.interval = TRUE)
plot(m, p)
```

`predict.lm` *Predict method for Linear Model Fits*

Description

Predicted values based on linear model object.

Usage

```
## S3 method for class 'lm'
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, na.action = na.pass,
        pred.var = res.var/weights, weights = 1,
        rankdeficient = c("warnif", "simple", "non-estim", "NA", "NAwarn"),
        tol = 1e-6, verbose = FALSE,
        ...)
```

Arguments

`object`	Object of class inheriting from `"lm"`
`newdata`	An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.
`se.fit`	A switch indicating if standard errors are required.
`scale`	Scale parameter for std.err. calculation.
`df`	Degrees of freedom for scale.
`interval`	Type of interval calculation. Can be abbreviated.
`level`	Tolerance/confidence level.
`type`	Type of prediction (response or model term). Can be abbreviated.
`terms`	If `type = "terms"`, which terms (default is all terms), a `character` vector.
`na.action`	function determining what should be done with missing values in `newdata`. The default is to predict `NA`.
`pred.var`	the variance(s) for future observations to be assumed for prediction intervals. See 'Details'.

weights	variance weights for prediction. This can be a numeric vector or a one-sided model formula. In the latter case, it is interpreted as an expression evaluated in newdata.
rankdeficient	a character string specifying what should happen in the case of a rank deficient model, i.e., when object$rank < ncol(model.matrix(object)).
	"warnif": gives a warning only in case of predicting 'non-estimable' cases, i.e., vectors not in the same predictor subspace as the original data (with tolerance tol). In that case, the non-estimable indices are also returned as attribute "non-estim" (see rankdeficient="non-estim").
	"simple": is back compatible to R < 4.3.0, possibly giving dubious predictions in non-estimable cases, and always signalling a warning.
	"non-estim": gives the same predictions without warning, and with an attribute attr(*, "non-estim") with indices in 1:nrow(newdata) of new data observations which are deemed non-estimable.
	"NA": predicts NA for non-estimable new data, silently. Often recommended in new code.
	"NAwarn": predicts NA for non-estimable new data with a warning.
tol	non-negative number determining how non-estimability is determined in rank deficient cases.
verbose	logical indicating if messages should be produced about rank deficiency handling.
...	further arguments passed to or from other methods.

Details

predict.lm produces predicted values, obtained by evaluating the regression function in the frame newdata (which defaults to model.frame(object)). If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting intervals specifies computation of confidence or prediction (tolerance) intervals at the specified level, sometimes referred to as narrow vs. wide intervals.

If the fit is rank-deficient, some of the columns of the design matrix will have been dropped during the lm computations, and corresponding coef() components set to NA. Prediction from such a fit only makes sense if newdata is contained in the same subspace as the original data. Other newdata entries (rows) are non-estimable. This is now checked (up to numerical tolerance tol) unless rankdeficient == "simple", which corresponds to previous behaviour, warns always and predicts using the non-NA coefficients with the corresponding columns of the design matrix. The new default option, rankdeficient == "warnif" checks if there are "non-estimable" cases (up to tolerance tol) and only warns in that case. All further rankdeficient options also check and either predict NA or mark the non-estimable cases differently.

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit are handled is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the predictions, whereas if na.action = na.exclude they will appear (in predictions, standard errors or interval limits), with value NA. See also napredict.

The prediction intervals are for a single observation at each case in `newdata` (or by default, the data used for the fit) with error variance(s) `pred.var`. This can be a multiple of `res.var`, the estimated value of σ^2: the default is to assume that future observations have the same error variance as those used for fitting. If `weights` is supplied, the inverse of this is used as a scale factor. For a weighted fit, if the prediction is for the original data frame, `weights` defaults to the weights used for the model fit, with a warning since it might not be the intended result. If the fit was weighted and `newdata` is given, the default is to assume constant prediction variance, with a warning.

Value

`predict.lm` produces a vector of predictions or a matrix of predictions and bounds with column names `fit`, `lwr`, and `upr` if `interval` is set. For `type = "terms"` this is a matrix with a column per term and may have an attribute `"constant"`.

If `se.fit` is `TRUE`, a list with the following components is returned:

`fit`	vector or matrix as above
`se.fit`	standard error of predicted means
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

Notice that prediction variances and prediction intervals always refer to *future* observations, possibly corresponding to the same predictors as used for the fit. The variance of the *residuals* will be smaller.

Strictly speaking, the formula used for prediction limits assumes that the degrees of freedom for the fit are the same as those for the residual variance. This may not be the case if `res.var` is not obtained from the fit.

See Also

The model fitting function `lm`, `predict`.

SafePrediction for prediction from (univariable) polynomial and spline fits.

Examples

```
require(graphics)

## Predictions
x <- rnorm(15)
y <- x + rnorm(15)
predict(lm(y ~ x))
new <- data.frame(x = seq(-3, 3, 0.5))
predict(lm(y ~ x), new, se.fit = TRUE)
pred.w.plim <- predict(lm(y ~ x), new, interval = "prediction")
```

```
pred.w.clim <- predict(lm(y ~ x), new, interval = "confidence")
matplot(new$x, cbind(pred.w.clim, pred.w.plim[,-1]),
        lty = c(1,2,2,3,3), type = "l", ylab = "predicted y")

## Prediction intervals, special cases
##  The first three of these throw warnings
w <- 1 + x^2
fit <- lm(y ~ x)
wfit <- lm(y ~ x, weights = w)
predict(fit, interval = "prediction")
predict(wfit, interval = "prediction")
predict(wfit, new, interval = "prediction")
predict(wfit, new, interval = "prediction", weights = (new$x)^2)
predict(wfit, new, interval = "prediction", weights = ~x^2)

##-- From  aov(.) example ---- predict(.. terms)
npk.aov <- aov(yield ~ block + N*P*K, npk)
(termL <- attr(terms(npk.aov), "term.labels"))
(pt <- predict(npk.aov, type = "terms"))
pt. <- predict(npk.aov, type = "terms", terms = termL[1:4])
stopifnot(all.equal(pt[,1:4], pt.,
                    tolerance = 1e-12, check.attributes = FALSE))
```

`predict.loess` *Predict LOESS Curve or Surface*

Description

Predictions from a `loess` fit, optionally with standard errors.

Usage

```
## S3 method for class 'loess'
predict(object, newdata = NULL, se = FALSE,
        na.action = na.pass, ...)
```

Arguments

`object`	an object fitted by `loess`.
`newdata`	an optional data frame in which to look for variables with which to predict, or a matrix or vector containing exactly the variables needs for prediction. If missing, the original data points are used.
`se`	should standard errors be computed?
`na.action`	function determining what should be done with missing values in data frame `newdata`. The default is to predict `NA`.
`...`	arguments passed to or from other methods.

Details

The standard errors calculation se = TRUE is slower than prediction, notably as it needs a relatively large workspace (memory), notably matrices of dimension $N \times Nf$ where f =span, i.e., se = TRUE is $O(N^2)$ and hence stops when the sample size N is larger than about 40'600 (for default span = 0.75).

When the fit was made using surface = "interpolate" (the default), predict.loess will not extrapolate – so points outside an axis-aligned hypercube enclosing the original data will have missing (NA) predictions and standard errors.

Value

If se = FALSE, a vector giving the prediction for each row of newdata (or the original data). If se = TRUE, a list containing components

fit	the predicted values.
se	an estimated standard error for each predicted value.
residual.scale	the estimated scale of the residuals used in computing the standard errors.
df	an estimate of the effective degrees of freedom used in estimating the residual scale, intended for use with t-based confidence intervals.

If newdata was the result of a call to expand.grid, the predictions (and s.e.'s if requested) will be an array of the appropriate dimensions.

Predictions from infinite inputs will be NA since loess does not support extrapolation.

Note

Variables are first looked for in newdata and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in newdata if it was supplied.

Author(s)

B. D. Ripley, based on the cloess package of Cleveland, Grosse and Shyu.

See Also

loess

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed = seq(5, 30, 1)), se = TRUE)
# to get extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control = loess.control(surface = "direct"))
predict(cars.lo2, data.frame(speed = seq(5, 30, 1)), se = TRUE)
```

predict.nls *Predicting from Nonlinear Least Squares Fits*

Description

predict.nls produces predicted values, obtained by evaluating the regression function in the frame newdata. If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting intervals specifies computation of confidence or prediction (tolerance) intervals at the specified level.

At present se.fit and interval are ignored.

Usage

```
## S3 method for class 'nls'
predict(object, newdata , se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, ...)
```

Arguments

object	An object that inherits from class nls.
newdata	A named list or data frame in which to look for variables with which to predict. If newdata is missing the fitted values at the original data points are returned.
se.fit	A logical value indicating if the standard errors of the predictions should be calculated. Defaults to FALSE. At present this argument is ignored.
scale	A numeric scalar. If it is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this information is extracted from the model fit. At present this argument is ignored.
df	A positive numeric scalar giving the number of degrees of freedom for the scale estimate. At present this argument is ignored.
interval	A character string indicating if prediction intervals or a confidence interval on the mean responses are to be calculated. At present this argument is ignored.
level	A numeric scalar between 0 and 1 giving the confidence level for the intervals (if any) to be calculated. At present this argument is ignored.
...	Additional optional arguments. At present no optional arguments are used.

Value

predict.nls produces a vector of predictions. When implemented, interval will produce a matrix of predictions and bounds with column names fit, lwr, and upr. When implemented, if se.fit is TRUE, a list with the following components will be returned:

fit	vector or matrix as above

se.fit	standard error of predictions
residual.scale	residual standard deviations
df	degrees of freedom for residual

Note

Variables are first looked for in newdata and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in newdata if it was supplied.

See Also

The model fitting function nls, predict.

Examples

```
require(graphics)

fm <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
predict(fm)              # fitted values at observed times
## Form data plot and smooth line for the predictions
opar <- par(las = 1)
plot(demand ~ Time, data = BOD, col = 4,
     main = "BOD data and fitted first-order curve",
     xlim = c(0,7), ylim = c(0, 20) )
tt <- seq(0, 8, length.out = 101)
lines(tt, predict(fm, list(Time = tt)))
par(opar)
```

---

predict.smooth.spline *Predict from Smoothing Spline Fit*

Description

Predict a smoothing spline fit at new points, return the derivative if desired. The predicted fit is linear beyond the original data.

Usage

```
## S3 method for class 'smooth.spline'
predict(object, x, deriv = 0, ...)
```

Arguments

object	a fit from smooth.spline.
x	the new values of x.
deriv	integer; the order of the derivative required.
...	further arguments passed to or from other methods.

Value

A list with components

x	The input x.
y	The fitted values or derivatives at x.

See Also

smooth.spline

Examples

```
require(graphics)

attach(cars)
cars.spl <- smooth.spline(speed, dist, df = 6.4)


## "Proof" that the derivatives are okay, by comparing with approximation
diff.quot <- function(x, y) {
  ## Difference quotient (central differences where available)
  n <- length(x); i1 <- 1:2; i2 <- (n-1):n
  c(diff(y[i1]) / diff(x[i1]), (y[-i1] - y[-i2]) / (x[-i1] - x[-i2]),
    diff(y[i2]) / diff(x[i2]))
}

xx <- unique(sort(c(seq(0, 30, by = .2), kn <- unique(speed))))
i.kn <- match(kn, xx)   # indices of knots within xx
op <- par(mfrow = c(2,2))
plot(speed, dist, xlim = range(xx), main = "Smooth.spline & derivatives")
lines(pp <- predict(cars.spl, xx), col = "red")
points(kn, pp$y[i.kn], pch = 3, col = "dark red")
mtext("s(x)", col = "red")
for(d in 1:3){
  n <- length(pp$x)
  plot(pp$x, diff.quot(pp$x,pp$y), type = "l", xlab = "x", ylab = "",
       col = "blue", col.main = "red",
       main = paste0("s" ,paste(rep("'", d), collapse = ""), "(x)"))
  mtext("Difference quotient approx.(last)", col = "blue")
  lines(pp <- predict(cars.spl, xx, deriv = d), col = "red")

  points(kn, pp$y[i.kn], pch = 3, col = "dark red")
  abline(h = 0, lty = 3, col = "gray")
}
detach(); par(op)
```

preplot *Pre-computations for a Plotting Object*

Description

Compute an object to be used for plots relating to the given model object.

Usage

```
preplot(object, ...)
```

Arguments

`object`	a fitted model object.
`...`	additional arguments for specific methods.

Details

Only the generic function is currently provided in base R, but some add-on packages have methods. Principally here for S compatibility.

Value

An object set up to make a plot that describes `object`.

princomp *Principal Components Analysis*

Description

`princomp` performs a principal components analysis on the given numeric data matrix and returns the results as an object of class `princomp`.

Usage

```
princomp(x, ...)

## S3 method for class 'formula'
princomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
princomp(x, cor = FALSE, scores = TRUE, covmat = NULL,
         subset = rep_len(TRUE, nrow(as.matrix(x))), fix_sign = TRUE, ...)

## S3 method for class 'princomp'
predict(object, newdata, ...)
```

Arguments

formula	a formula with no response variable, referring only to numeric variables.
data	an optional data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector used to select rows (observations) of the data matrix x.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit.
x	a numeric matrix or data frame which provides the data for the principal components analysis.
cor	a logical value indicating whether the calculation should use the correlation matrix or the covariance matrix. (The correlation matrix can only be used if there are no constant variables.)
scores	a logical value indicating whether the score on each principal component should be calculated.
covmat	a covariance matrix, or a covariance list as returned by cov.wt (and cov.mve or cov.mcd from package **MASS**). If supplied, this is used rather than the covariance matrix of x.
fix_sign	Should the signs of the loadings and scores be chosen so that the first element of each loading is non-negative?
...	arguments passed to or from other methods. If x is a formula one might specify cor or scores.
object	Object of class inheriting from "princomp".
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

princomp is a generic function with "formula" and "default" methods.

The calculation is done using eigen on the correlation or covariance matrix, as determined by cor. (This was done for compatibility with the S-PLUS result.) A preferred method of calculation is to use svd on x, as is done in prcomp.

Note that the default calculation uses divisor N for the covariance matrix.

The print method for these objects prints the results in a nice format and the plot method produces a scree plot (screeplot). There is also a biplot method.

If x is a formula then the standard NA-handling is applied to the scores (if requested): see napredict.

princomp only handles so-called R-mode PCA, that is feature extraction of variables. If a data matrix is supplied (possibly via a formula) it is required that there are at least as many units as variables. For Q-mode PCA use prcomp.

Value

princomp returns a list with class "princomp" containing the following components:

sdev	the standard deviations of the principal components.
loadings	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). This is of class "loadings": see loadings for its print method.
center	the means that were subtracted.
scale	the scalings applied to each variable.
n.obs	the number of observations.
scores	if scores = TRUE, the scores of the supplied data on the principal components. These are non-null only if x was supplied, and if covmat was also supplied if it was a covariance list. For the formula method, napredict() is applied to handle the treatment of values omitted by the na.action.
call	the matched call.
na.action	If relevant.

Note

The signs of the columns of the loadings and scores are arbitrary, and so may differ between different programs for PCA, and even between different builds of R: fix_sign = TRUE alleviates that.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

summary.princomp, screeplot, biplot.princomp, prcomp, cor, cov, eigen.

Examples

```
require(graphics)

## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests))  # inappropriate
princomp(USArrests, cor = TRUE) # =^= prcomp(USArrests, scale=TRUE)
## Similar, but different:
## The standard deviations differ by a factor of sqrt(49/50)

summary(pc.cr <- princomp(USArrests, cor = TRUE))
loadings(pc.cr)  # note that blank entries are small but not zero
## The signs of the columns of the loadings are arbitrary
plot(pc.cr) # shows a screeplot.
biplot(pc.cr)

## Formula interface
```

```
princomp(~ ., data = USArrests, cor = TRUE)

## NA-handling
USArrests[1, 2] <- NA
pc.cr <- princomp(~ Murder + Assault + UrbanPop,
                  data = USArrests, na.action = na.exclude, cor = TRUE)
pc.cr$scores[1:5, ]

## (Simple) Robust PCA:
## Classical:
(pc.cl  <- princomp(stackloss))
## Robust:
(pc.rob <- princomp(stackloss, covmat = MASS::cov.rob(stackloss)))
```

`print.power.htest` *Print Methods for Hypothesis Tests and Power Calculation Objects*

Description

Printing objects of class `"htest"` or `"power.htest"`, respectively, by simple `print` methods.

Usage

```
## S3 method for class 'htest'
print(x, digits = getOption("digits"), prefix = "\t", ...)

## S3 method for class 'power.htest'
print(x, digits = getOption("digits"), ...)
```

Arguments

`x`	object of class `"htest"` or `"power.htest"`.
`digits`	number of significant digits to be used.
`prefix`	string, passed to `strwrap` for displaying the `method` component of the `htest` object.
`...`	further arguments to be passed to or from methods.

Details

Both `print` methods traditionally have not obeyed the `digits` argument properly. They now do, the `htest` method mostly in expressions like `max(1, digits - 2)`.

A `power.htest` object is just a named list of numbers and character strings, supplemented with `method` and `note` elements. The `method` is displayed as a title, the `note` as a footnote, and the remaining elements are given in an aligned 'name = value' format.

Value

the argument `x`, invisibly, as for all `print` methods.

Author(s)

Peter Dalgaard

See Also

power.t.test, power.prop.test

Examples

```
(ptt <- power.t.test(n = 20, delta = 1))
print(ptt, digits =  4) # using less digits than default
print(ptt, digits = 12) # using more  "       "     "
```

---

print.ts *Printing and Formatting of Time-Series Objects*

Description

Notably for calendar related time series objects, format and print methods showing years, months and or quarters respectively.

Usage

```
## S3 method for class 'ts'
print(x, calendar, ...)
.preformat.ts(x, calendar, ...)
```

Arguments

x	a time series object.
calendar	enable/disable the display of information about month names, quarter names or year when printing. The default is TRUE for a frequency of 4 or 12, FALSE otherwise.
...	additional arguments to print (or format methods).

Details

The print method for "ts" objects prints a header (basically of tsp(x)), if calendar is false, and then prints the result of .preformat.ts(x, *), which is typically a matrix with rownames built from the calendar times where applicable.

See Also

print, ts.

Examples

```
print(ts(1:10, frequency = 7, start = c(12, 2)), calendar = TRUE)

print(sunsp.1 <- window(sunspot.month, end=c(1756, 12)))
m <- .preformat.ts(sunsp.1) # a character matrix
```

`printCoefmat` *Print Coefficient Matrices*

Description

Utility function to be used in higher-level `print` methods, such as those for `summary.lm`, `summary.glm` and `anova`. The goal is to provide a flexible interface with smart defaults such that often, only `x` needs to be specified.

Usage

```
printCoefmat(x, digits = max(3, getOption("digits") - 2),
             signif.stars = getOption("show.signif.stars"),
             signif.legend = signif.stars,
             dig.tst = max(1, min(5, digits - 1)),
             cs.ind = 1L:k, tst.ind = k + 1L,
             zap.ind = integer(), P.values = NULL,
             has.Pvalue = nc >= 4L && length(cn <- colnames(x)) &&
                          substr(cn[nc], 1L, 3L) %in% c("Pr(", "p-v"),
             eps.Pvalue = .Machine$double.eps,
             na.print = "NA", quote = FALSE, right = TRUE, ...)
```

Arguments

`x`	a numeric matrix like object, to be printed.
`digits`	minimum number of significant digits to be used for most numbers.
`signif.stars`	logical; if `TRUE`, P-values are additionally encoded visually as 'significance stars' in order to help scanning of long coefficient tables. It defaults to the `show.signif.stars` slot of `options`.
`signif.legend`	logical; if `TRUE`, a legend for the 'significance stars' is printed provided `signif.stars` = `TRUE`.
`dig.tst`	minimum number of significant digits for the test statistics, see `tst.ind`.
`cs.ind`	indices (integer) of column numbers which are (like) **c**oefficients and **s**tandard errors to be formatted together.
`tst.ind`	indices (integer) of column numbers for test statistics.
`zap.ind`	indices (integer) of column numbers which should be formatted by `zapsmall`, i.e., by 'zapping' values close to 0.

P.values	logical or NULL; if TRUE, the last column of x is formatted by format.pval as P values. If P.values = NULL, the default, it is set to TRUE only if options("show.coef.Pvalue") is TRUE *and* x has at least 4 columns *and* the last column name of x starts with "Pr(".
has.Pvalue	logical; if TRUE, the last column of x contains P values; in that case, it is printed if and only if P.values (above) is true.
eps.Pvalue	number, passed to format.pval() as eps.
na.print	a character string to code NA values in printed output.
quote, right, ...	further arguments passed to print.default.

Value

Invisibly returns its argument, x.

Author(s)

Martin Maechler

See Also

print.summary.lm, format.pval, format.

Examples

```
cmat <- cbind(rnorm(3, 10), sqrt(rchisq(3, 12)))
cmat <- cbind(cmat, cmat[, 1]/cmat[, 2])
cmat <- cbind(cmat, 2*pnorm(-cmat[, 3]))
colnames(cmat) <- c("Estimate", "Std.Err", "Z value", "Pr(>z)")
printCoefmat(cmat[, 1:3])
printCoefmat(cmat)
op <- options(show.coef.Pvalues = FALSE)
printCoefmat(cmat, digits = 2)
printCoefmat(cmat, digits = 2, P.values = TRUE)
options(op) # restore
```

---

profile *Generic Function for Profiling Models*

Description

Investigates the behavior of the objective function near the solution represented by fitted.

See documentation on method functions for further details.

Usage

```
profile(fitted, ...)
```

Arguments

fitted	the original fitted model object.
...	additional parameters. See documentation on individual methods.

Value

A list with an element for each parameter being profiled. See the individual methods for further details.

See Also

profile.nls, profile.glm ...

plot.profile.

For profiling R code, see Rprof.

---

profile.glm *Method for Profiling* glm *Objects*

Description

Investigates the profile log-likelihood function for a fitted model of class "glm".

Usage

```
## S3 method for class 'glm'
profile(fitted, which = 1:p, alpha = 0.01, maxsteps = 10,
        del = zmax/5, trace = FALSE, test = c("LRT", "Rao"), ...)
```

Arguments

fitted	the original fitted model object.
which	the original model parameters which should be profiled. This can be a numeric or character vector. By default, all parameters are profiled.
alpha	highest significance level allowed for the profile z-statistics.
maxsteps	maximum number of points to be used for profiling each parameter.
del	suggested change on the scale of the profile t-statistics. Default value chosen to allow profiling at about 10 parameter values.
trace	logical: should the progress of profiling be reported?
test	profile Likelihood Ratio test or Rao Score test.
...	further arguments passed to or from other methods.

Details

The profile z-statistic is defined either as (case `test = "LRT"`) the square root of change in deviance with an appropriate sign, or (case `test = "Rao"`) as the similarly signed square root of the Rao Score test statistic. The latter is defined as the squared gradient of the profile log likelihood divided by the profile Fisher information, but more conveniently calculated via the deviance of a Gaussian GLM fitted to the residuals of the profiled model.

Value

A list of classes `"profile.glm"` and `"profile"` with an element for each parameter being profiled. The elements are data-frames with two variables

`par.vals`	a matrix of parameter values for each fitted model.
`tau` or `z`	the profile t or z-statistics (the name depends on whether there is an estimated dispersion parameter.)

Author(s)

Originally, D. M. Bates and W. N. Venables. (For S in 1996.)

See Also

`glm`, `profile`, `plot.profile`

Examples

```
options(contrasts = c("contr.treatment", "contr.poly"))
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive = 20 - numdead)
budworm.lg <- glm(SF ~ sex*ldose, family = binomial)
pr1 <- profile(budworm.lg)
plot(pr1)
pairs(pr1)
```

`profile.nls` *Method for Profiling* `nls` *Objects*

Description

Investigates the profile log-likelihood function for a fitted model of class `"nls"`.

Usage

```
## S3 method for class 'nls'
profile(fitted, which = 1:npar, maxpts = 100, alphamax = 0.01,
        delta.t = cutoff/5, ...)
```

Arguments

`fitted`	the original fitted model object.
`which`	the original model parameters which should be profiled. This can be a numeric or character vector. By default, all non-linear parameters are profiled.
`maxpts`	maximum number of points to be used for profiling each parameter.
`alphamax`	highest significance level allowed for the profile t-statistics.
`delta.t`	suggested change on the scale of the profile t-statistics. Default value chosen to allow profiling at about 10 parameter values.
`...`	further arguments passed to or from other methods.

Details

The profile t-statistics is defined as the square root of change in sum-of-squares divided by residual standard error with an appropriate sign.

Value

A list with an element for each parameter being profiled. The elements are data-frames with two variables

`par.vals`	a matrix of parameter values for each fitted model.
`tau`	the profile t-statistics.

Author(s)

Of the original version, Douglas M. Bates and Saikat DebRoy

References

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6).

See Also

`nls`, `profile`, `plot.profile.nls`

Examples

```
# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
# get the profile for the fitted model: default level is too extreme
pr1 <- profile(fm1, alphamax = 0.05)
# profiled values for the two parameters

pr1$A
pr1$lrc

# see also example(plot.profile.nls)
```

proj *Projections of Models*

Description

proj returns a matrix or list of matrices giving the projections of the data onto the terms of a linear model. It is most frequently used for aov models.

Usage

```
proj(object, ...)

## S3 method for class 'aov'
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## S3 method for class 'aovlist'
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## Default S3 method:
proj(object, onedf = TRUE, ...)

## S3 method for class 'lm'
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)
```

Arguments

object	An object of class "lm" or a class inheriting from it, or an object with a similar structure including in particular components qr and effects.
onedf	A logical flag. If TRUE, a projection is returned for all the columns of the model matrix. If FALSE, the single-column projections are collapsed by terms of the model (as represented in the analysis of variance table).
unweighted.scale	If the fit producing object used weights, this determines if the projections correspond to weighted or unweighted observations.
...	Swallow and ignore any other arguments.

Details

A projection is given for each stratum of the object, so for aov models with an Error term the result is a list of projections.

Value

A projection matrix or (for multi-stratum objects) a list of projection matrices.

Each projection is a matrix with a row for each observations and either a column for each term (onedf = FALSE) or for each coefficient (onedf = TRUE). Projection matrices from the default

method have orthogonal columns representing the projection of the response onto the column space of the Q matrix from the QR decomposition. The fitted values are the sum of the projections, and the sum of squares for each column is the reduction in sum of squares from fitting that column (after those to the left of it).

The methods for `lm` and `aov` models add a column to the projection matrix giving the residuals (the projection of the data onto the orthogonal complement of the model space).

Strictly, when `onedf = FALSE` the result is not a projection, but the columns represent sums of projections onto the columns of the model matrix corresponding to that term. In this case the matrix does not depend on the coding used.

Author(s)

The design was inspired by the S function of the same name described in Chambers et al. (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`aov`, `lm`, `model.tables`

Examples

```
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
proj(npk.aov)

## as a test, not particularly sensible
options(contrasts = c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
proj(npk.aovE)
```

---

prop.test *Test of Equal or Given Proportions*

Description

`prop.test` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

Usage

```
prop.test(x, n, p = NULL,
          alternative = c("two.sided", "less", "greater"),
          conf.level = 0.95, correct = TRUE)
```

Arguments

`x`	a vector of counts of successes, a one-dimensional table with two entries, or a two-dimensional table (or matrix) with 2 columns, giving the counts of successes and failures, respectively.
`n`	a vector of counts of trials; ignored if `x` is a matrix or a table.
`p`	a vector of probabilities of success. The length of `p` must be the same as the number of groups specified by `x`, and its elements must be greater than 0 and less than 1.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter. Only used for testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`conf.level`	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Only used when testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`correct`	a logical indicating whether Yates' continuity correction should be applied where possible.

Details

Only groups with finite numbers of successes and failures are used. Counts of successes and failures must be nonnegative and hence not greater than the corresponding numbers of trials which must be positive. All finite counts should be integers.

If `p` is `NULL` and there is more than one group, the null tested is that the proportions in each group are the same. If there are two groups, the alternatives are that the probability of success in the first group is less than, not equal to, or greater than the probability of success in the second group, as specified by `alternative`. A confidence interval for the difference of proportions with confidence level as specified by `conf.level` and clipped to $[-1, 1]$ is returned. Continuity correction is used only if it does not exceed the difference of the sample proportions in absolute value. Otherwise, if there are more than 2 groups, the alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

If there is only one group, then the null tested is that the underlying probability of success is `p`, or .5 if `p` is not given. The alternative is that the probability of success is less than, not equal to, or greater than `p` or 0.5, respectively, as specified by `alternative`. A confidence interval for the underlying proportion with confidence level as specified by `conf.level` and clipped to $[0, 1]$ is returned. Continuity correction is used only if it does not exceed the difference between sample and null proportions in absolute value. The confidence interval is computed by inverting the score test.

Finally, if `p` is given and there are more than 2 groups, the null tested is that the underlying probabilities of success are those given by `p`. The alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

Value

A list with class "htest" containing the following components:

statistic	the value of Pearson's chi-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
estimate	a vector with the sample proportions x/n.
conf.int	a confidence interval for the true proportion if there is one group, or for the difference in proportions if there are 2 groups and p is not given, or NULL otherwise. In the cases where it is not NULL, the returned confidence interval has an asymptotic confidence level as specified by conf.level, and is appropriate to the specified alternative hypothesis.
null.value	the value of p if specified by the null, or NULL otherwise.
alternative	a character string describing the alternative.
method	a character string indicating the method used, and whether Yates' continuity correction was applied.
data.name	a character string giving the names of the data.

References

Wilson, E.B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, **22**, 209–212. doi:10.2307/2276774.

Newcombe R.G. (1998). Two-Sided Confidence Intervals for the Single Proportion: Comparison of Seven Methods. *Statistics in Medicine*, **17**, 857–872. doi:10.1002/(SICI)1097-0258(19980430)17:8<857::AIDSIM777>3.0.CO;2E.

Newcombe R.G. (1998). Interval Estimation for the Difference Between Independent Proportions: Comparison of Eleven Methods. *Statistics in Medicine*, **17**, 873–890. doi:10.1002/(SICI)1097-0258(19980430)17:8<873::AIDSIM779>3.0.CO;2I.

See Also

binom.test for an *exact* test of a binomial hypothesis.

Examples

```
heads <- rbinom(1, size = 100, prob = .5)
prop.test(heads, 100)          # continuity correction TRUE by default
prop.test(heads, 100, correct = FALSE)

## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers  <- c( 83, 90, 129, 70 )
```

```
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
```

`prop.trend.test` *Test for trend in proportions*

Description

Performs chi-squared test for trend in proportions, i.e., a test asymptotically optimal for local alternatives where the log odds vary in proportion with `score`. By default, `score` is chosen as the group numbers.

Usage

```
prop.trend.test(x, n, score = seq_along(x))
```

Arguments

`x`	Number of events
`n`	Number of trials
`score`	Group score

Value

An object of class `"htest"` with title, test statistic, p-value, etc.

Note

This really should get integrated with `prop.test`

Author(s)

Peter Dalgaard

See Also

`prop.test`

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
prop.trend.test(smokers, patients)
prop.trend.test(smokers, patients, c(0,0,0,1))
```

qqnorm *Quantile-Quantile Plots*

Description

qqnorm is a generic function the default method of which produces a normal QQ plot of the values in y. qqline adds a line to a "theoretical", by default normal, quantile-quantile plot which passes through the probs quantiles, by default the first and third quartiles.

qqplot produces a QQ plot of two datasets. If conf.level is given, a confidence band for a function transforming the distribution of x into the distribution of y is plotted based on Switzer (1976). The QQ plot can be understood as an estimate of such a treatment function. If exact = NULL (the default), an exact confidence band is computed if the product of the sample sizes is less than 10000, with or without ties. Otherwise, asymptotic distributions are used whose approximations may be inaccurate in small samples. Monte-Carlo approximations based on B random permutations are computed when simulate = TRUE. Confidence bands are in agreement with Smirnov's test, that is, the bisecting line is covered by the band iff the null of both samples coming from the same distribution cannot be rejected at the same level.

Graphical parameters may be given as arguments to qqnorm, qqplot and qqline.

Usage

```
qqnorm(y, ...)
## Default S3 method:
qqnorm(y, ylim, main = "Normal Q-Q Plot",
       xlab = "Theoretical Quantiles", ylab = "Sample Quantiles",
       plot.it = TRUE, datax = FALSE, ...)

qqline(y, datax = FALSE, distribution = qnorm,
       probs = c(0.25, 0.75), qtype = 7, ...)

qqplot(x, y, plot.it = TRUE,
       xlab = deparse1(substitute(x)),
       ylab = deparse1(substitute(y)), ...,
       conf.level = NULL,
       conf.args = list(exact = NULL, simulate.p.value = FALSE,
                        B = 2000, col = NA, border = NULL))
```

Arguments

x	The first sample for qqplot.
y	The second or only data sample.
xlab, ylab, main	plot labels. The xlab and ylab refer to the y and x axes respectively if datax = TRUE.
plot.it	logical. Should the result be plotted?
datax	logical. Should data values be on the x-axis?

distribution	quantile function for reference theoretical distribution.
probs	numeric vector of length two, representing probabilities. Corresponding quantile pairs define the line drawn.
qtype	the type of quantile computation used in quantile.
ylim, ...	graphical parameters.
conf.level	confidence level of the band. The default, NULL, does not lead to the computation of a confidence band.
conf.args	list of arguments defining confidence band computation and visualisation: exact is NULL (see details) or a logical indicating whether an exact p-value should be computed, simulate.p.value is a logical indicating whether to compute p-values by Monte Carlo simulation, B defines the number of replicates used in the Monte Carlo test, col and border define the color for filling and border of the confidence band (the default, NA and NULL, is to leave the band unfilled with black borders.

Value

For qqnorm and qqplot, a list with components

x	The x coordinates of the points that were/would be plotted
y	The original y vector, i.e., the corresponding y coordinates *including* NAs. If conf.level was specified to qqplot, the list contains additional components lwr and upr defining the confidence band.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Switzer, P. (1976). Confidence procedures for two-sample problems. *Biometrika*, **63**(1), 13–25. doi:10.1093/biomet/63.1.13.

See Also

ppoints, used by qqnorm to generate approximations to expected order statistics for a normal distribution.

Examples

```
require(graphics)

y <- rt(200, df = 5)
qqnorm(y); qqline(y, col = 2)
qqplot(y, rt(300, df = 5))

qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")

## "QQ-Chisquare" : --------------------------
y <- rchisq(500, df = 3)
```

```
## Q-Q plot for Chi^2 data against true theoretical distribution:
qqplot(qchisq(ppoints(500), df = 3), y,
       main = expression("Q-Q plot for" ~~ {chi^2}[nu == 3]))
qqline(y, distribution = function(p) qchisq(p, df = 3),
       probs = c(0.1, 0.6), col = 2)
mtext("qqline(*, dist = qchisq(., df=3), prob = c(0.1, 0.6))")
## (Note that the above uses ppoints() with a = 1/2, giving the
## probability points for quantile type 5: so theoretically, using
## qqline(qtype = 5) might be preferable.)

## Figure 1 in Switzer (1976), knee angle data
switzer <- data.frame(
    angle = c(-31, -30, -25, -25, -23, -23, -22, -20, -20, -18,
              -18, -18, -16, -15, -15, -14, -13, -11, -10, - 9,
              - 8, - 7, - 7, - 7, - 6, - 6, - 4, - 4, - 3, - 2,
              - 2, - 1,   1,   1,   4,   5,  11,  12,  16,  34,
              -31, -20, -18, -16, -16, -16, -15, -14, -14, -14,
              -14, -13, -13, -11, -11, -10, - 9, - 9, - 8, - 7,
              - 7, - 6, - 6,  -5, - 5, - 5, - 4, - 2, - 2, - 2,
                0,   0,   1,   1,   2,   4,   5,   5,   6,  17),
    sex = gl(2, 40, labels = c("Female", "Male")))

ks.test(angle ~ sex, data = switzer)
d <- with(switzer, split(angle, sex))
with(d, qqplot(Female, Male, pch = 19, xlim = c(-31, 31), ylim = c(-31, 31),
               conf.level = 0.945,
               conf.args = list(col = "lightgrey", exact = TRUE))
)
abline(a = 0, b = 1)

## agreement with ks.test
set.seed(1)
x <- rnorm(50)
y <- rnorm(50, mean = .5, sd = .95)
ex <- TRUE
### p = 0.112
(pval <- ks.test(x, y, exact = ex)$p.value)
## 88.8% confidence band with bisecting line
## touching the lower bound
qqplot(x, y, pch = 19, conf.level = 1 - pval,
       conf.args = list(exact = ex, col = "lightgrey"))
abline(a = 0, b = 1)
```

---

quade.test *Quade Test*

Description

Performs a Quade test with unreplicated blocked data.

Usage

```
quade.test(y, ...)

## Default S3 method:
quade.test(y, groups, blocks, ...)

## S3 method for class 'formula'
quade.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

quade.test can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in y for each combination of levels of groups and blocks) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of blocks, the location parameter of y is the same in each of the groups.

If y is a matrix, groups and blocks are obtained from the column and row indices, respectively. NA's are not allowed in groups or blocks; if y contains NA's, corresponding blocks are removed.

Value

A list with class "htest" containing the following components:

statistic	the value of Quade's F statistic.
parameter	a vector with the numerator and denominator degrees of freedom of the approximate F distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Quade test".
data.name	a character string giving the names of the data.

References

D. Quade (1979), Using weighted rankings in the analysis of complete blocks with additive block effects. *Journal of the American Statistical Association* **74**, 680–683.

William J. Conover (1999), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 373–380.

See Also

friedman.test.

Examples

```
## Conover (1999, p. 375f):
## Numbers of five brands of a new hand lotion sold in seven stores
## during one week.
y <- matrix(c( 5,  4,  7, 10, 12,
               1,  3,  1,  0,  2,
              16, 12, 22, 22, 35,
               5,  4,  3,  5,  4,
              10,  9,  7, 13, 10,
              19, 18, 28, 37, 58,
              10,  7,  6,  8,  7),
            nrow = 7, byrow = TRUE,
            dimnames =
            list(Store = as.character(1:7),
                 Brand = LETTERS[1:5]))
y
(qTst <- quade.test(y))

## Show equivalence of different versions of test :
utils::str(dy <- as.data.frame(as.table(y)))
qT. <- quade.test(Freq ~ Brand|Store, data = dy)
qT.$data.name <- qTst$data.name
stopifnot(all.equal(qTst, qT., tolerance = 1e-15))
dys <- dy[order(dy[,"Freq"]),]
qTs <- quade.test(Freq ~ Brand|Store, data = dys)
qTs$data.name <- qTst$data.name
stopifnot(all.equal(qTst, qTs, tolerance = 1e-15))
```

---

quantile *Sample Quantiles*

Description

The generic function quantile produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

Usage

```
quantile(x, ...)

## Default S3 method:
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE,
         names = TRUE, type = 7, digits = 7, ...)
```

Arguments

x	numeric vector whose sample quantiles are wanted, or an object of a class for which a method has been defined (see also 'details'). NA and NaN values are not allowed in numeric vectors unless na.rm is TRUE.
probs	numeric vector of probabilities with values in $[0, 1]$. (Values up to '2e-14' outside that range are accepted and moved to the nearby endpoint.)
na.rm	logical; if true, any NA and NaN's are removed from x before the quantiles are computed.
names	logical; if true, the result has a names attribute. Set to FALSE for speedup with many probs.
type	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.
digits	used only when names is true: the precision to use when formatting the percentages. In R versions up to 4.0.x, this had been set to max(2, getOption("digits")), internally.
...	further arguments passed to or from other methods.

Details

A vector of length length(probs) is returned; if names = TRUE, it has a names attribute.

NA and NaN values in probs are propagated to the result.

The default method works with classed objects sufficiently like numeric vectors that sort and (not needed by types 1 and 3) addition of elements and multiplication by a number work correctly. Note that as this is in a namespace, the copy of sort in **base** will be used, not some S4 generic of that name. Also note that that is no check on the 'correctly', and so e.g. quantile can be applied to complex vectors which (apart from ties) will be ordered on their real parts.

There is a method for the date-time classes (see "POSIXt"). Types 1 and 3 can be used for class "Date" and for ordered factors.

Types

quantile returns estimates of underlying distribution quantiles based on one or two order statistics from the supplied elements in x at probabilities in probs. One of the nine quantile algorithms discussed in Hyndman and Fan (1996), selected by type, is employed.

All sample quantiles are defined as weighted averages of consecutive order statistics. Sample quantiles of type i are defined by:

$$Q_i(p) = (1 - \gamma)x_j + \gamma x_{j+1}$$

where $1 \le i \le 9$, $\frac{j-m}{n} \le p < \frac{j-m+1}{n}$, x_j is the j-th order statistic, n is the sample size, the value of γ is a function of $j = \lfloor np + m \rfloor$ and $g = np + m - j$, and m is a constant determined by the sample quantile type.

Discontinuous sample quantile types 1, 2, and 3

For types 1, 2 and 3, $Q_i(p)$ is a discontinuous function of p, with $m = 0$ when $i = 1$ and $i = 2$, and $m = -1/2$ when $i = 3$.

Type 1 Inverse of empirical distribution function. $\gamma = 0$ if $g = 0$, and 1 otherwise.

Type 2 Similar to type 1 but with averaging at discontinuities. $\gamma = 0.5$ if $g = 0$, and 1 otherwise (SAS default, see Wicklin (2017)).

Type 3 Nearest even order statistic (SAS default till ca. 2010). $\gamma = 0$ if $g = 0$ and j is even, and 1 otherwise.

Continuous sample quantile types 4 through 9

For types 4 through 9, $Q_i(p)$ is a continuous function of p, with $\gamma = g$ and m given below. The sample quantiles can be obtained equivalently by linear interpolation between the points (p_k, x_k) where x_k is the k-th order statistic. Specific expressions for p_k are given below.

Type 4 $m = 0$. $p_k = \frac{k}{n}$. That is, linear interpolation of the empirical cdf.

Type 5 $m = 1/2$. $p_k = \frac{k-0.5}{n}$. That is a piecewise linear function where the knots are the values midway through the steps of the empirical cdf. This is popular amongst hydrologists.

Type 6 $m = p$. $p_k = \frac{k}{n+1}$. Thus $p_k = \mathrm{E}[F(x_k)]$. This is used by Minitab and by SPSS.

Type 7 $m = 1 - p$. $p_k = \frac{k-1}{n-1}$. In this case, $p_k = \mathrm{mode}[F(x_k)]$. This is used by S.

Type 8 $m = (p+1)/3$. $p_k = \frac{k-1/3}{n+1/3}$. Then $p_k \approx \mathrm{median}[F(x_k)]$. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x.

Type 9 $m = p/4 + 3/8$. $p_k = \frac{k-3/8}{n+1/4}$. The resulting quantile estimates are approximately unbiased for the expected order statistics if x is normally distributed.

Further details are provided in Hyndman and Fan (1996) who recommended type 8. The default method is type 7, as used by S and by R < 2.0.0. Makkonen argues for type 6, also as already proposed by Weibull in 1939. The Wikipedia page contains further information about availability of these 9 types in software.

Author(s)

of the version used in R >= 2.0.0, Ivan Frohne and Rob J Hyndman.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, *American Statistician* **50**, 361–365. doi:10.2307/2684934.

Wicklin, R. (2017) Sample quantiles: A comparison of 9 definitions; SAS Blog. https://blogs.sas.com/content/iml/2017/05/24/definitions-sample-quantiles.html

Wikipedia: https://en.wikipedia.org/wiki/Quantile#Estimating_quantiles_from_a_sample

See Also

ecdf for empirical distributions of which quantile is an inverse; boxplot.stats and fivenum for computing other versions of quartiles, etc.

Examples

```
quantile(x <- rnorm(1001)) # Extremes & Quartiles by default
quantile(x,  probs = c(0.1, 0.5, 1, 2, 5, 10, 50, NA)/100)

### Compare different types
quantAll <- function(x, prob, ...)
  t(vapply(1:9, function(typ) quantile(x, probs = prob, type = typ, ...),
           quantile(x, prob, type=1, ...)))
p <- c(0.1, 0.5, 1, 2, 5, 10, 50)/100
signif(quantAll(x, p), 4)

## 0% and 100% are equal to min(), max() for all types:
stopifnot(t(quantAll(x, prob=0:1)) == range(x))

## for complex numbers:
z <- complex(real = x, imaginary = -10*x)
signif(quantAll(z, p), 4)
```

---

r2dtable *Random 2-way Tables with Given Marginals*

Description

Generate random 2-way tables with given marginals using Patefield's algorithm.

Usage

```
r2dtable(n, r, c)
```

Arguments

n	a non-negative numeric giving the number of tables to be drawn.
r	a non-negative vector of length at least 2 giving the row totals, to be coerced to integer. Must sum to the same as c.
c	a non-negative vector of length at least 2 giving the column totals, to be coerced to integer.

Value

A list of length n containing the generated tables as its components.

References

Patefield, W. M. (1981). Algorithm AS 159: An efficient method of generating r x c tables with given row and column totals. *Applied Statistics*, **30**, 91–97. doi:10.2307/2346669.

Examples

```
## Fisher's Tea Drinker data.
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nrow = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
## Simulate permutation test for independence based on the maximum
## Pearson residuals (rather than their sum).
rowTotals <- rowSums(TeaTasting)
colTotals <- colSums(TeaTasting)
nOfCases <- sum(rowTotals)
expected <- outer(rowTotals, colTotals) / nOfCases
maxSqResid <- function(x) max((x - expected) ^ 2 / expected)
simMaxSqResid <-
    sapply(r2dtable(1000, rowTotals, colTotals), maxSqResid)
sum(simMaxSqResid >= maxSqResid(TeaTasting)) / 1000
## Fisher's exact test gives p = 0.4857 ...
```

---

read.ftable *Manipulate Flat Contingency Tables*

Description

Read, write and coerce 'flat' (contingency) tables, aka `ftables`.

Usage

```
read.ftable(file, sep = "", quote = "\"",
            row.var.names, col.vars, skip = 0)

write.ftable(x, file = "", quote = TRUE, append = FALSE,
             digits = getOption("digits"), sep = " ", ...)

## S3 method for class 'ftable'
format(x, quote = TRUE, digits = getOption("digits"),
       method = c("non.compact", "row.compact", "col.compact", "compact"),
       lsep = " | ",
       justify = c("left", "right"),
       ...)

## S3 method for class 'ftable'
print(x, digits = getOption("digits"), ...)
```

Arguments

file	either a character string naming a file or a connection which the data are to be read from or written to. "" indicates input from the console for reading and output to the console for writing.
sep	the field separator string. Values on each line of the file are separated by this string.
quote	a character string giving the set of quoting characters for read.ftable; to disable quoting altogether, use quote="". For write.table, a logical indicating whether strings in the data will be surrounded by double quotes.
row.var.names	a character vector with the names of the row variables, in case these cannot be determined automatically.
col.vars	a list giving the names and levels of the column variables, in case these cannot be determined automatically.
skip	the number of lines of the data file to skip before beginning to read data.
x	an object of class "ftable".
append	logical. If TRUE and file is the name of a file (and not a connection or "|cmd"), the output from write.ftable is appended to the file. If FALSE, the contents of file will be overwritten.
digits	an integer giving the number of significant digits to use for (the cell entries of) x.
method	string specifying how the "ftable" object is formatted (and printed if used as in write.ftable() or the print method). Can be abbreviated. Available methods are (see the examples): **"non.compact"** the default representation of an "ftable" object. **"row.compact"** a row-compact version without empty cells below the column labels. **"col.compact"** a column-compact version without empty cells to the right of the row labels. **"compact"** a row- and column-compact version. This may imply a row and a column label sharing the same cell. They are then separated by the string lsep.
lsep	only for method = "compact", the separation string for row and column labels.
justify	character vector of length (one or) two, specifying how string justification should happen in format(..), first for the labels, then the table entries.
...	further arguments to be passed to or from methods; for write() and print(), notably arguments such as method, passed to format().

Details

read.ftable reads in a flat-like contingency table from a file. If the file contains the written representation of a flat table (more precisely, a header with all information on names and levels of column variables, followed by a line with the names of the row variables), no further arguments are needed. Similarly, flat tables with only one column variable the name of which is the only entry in the first line are handled automatically. Other variants can be dealt with by skipping all header

information using `skip`, and providing the names of the row variables and the names and levels of the column variable using `row.var.names` and `col.vars`, respectively. See the examples below.

Note that flat tables are characterized by their 'ragged' display of row (and maybe also column) labels. If the full grid of levels of the row variables is given, one should instead use `read.table` to read in the data, and create the contingency table from this using `xtabs`.

`write.ftable` writes a flat table to a file, which is useful for generating 'pretty' ASCII representations of contingency tables. Different versions are available via the `method` argument, which may be useful, for example, for constructing LaTeX tables.

References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

See Also

`ftable` for more information on flat contingency tables.

Examples

```
## Agresti (1990), page 157, Table 5.8.
## Not in ftable standard format, but o.k.
file <- tempfile()
cat("             Intercourse\n",
    "Race  Gender     Yes  No\n",
    "White Male        43 134\n",
    "      Female      26 149\n",
    "Black Male        29  23\n",
    "      Female      22  36\n",
    file = file)
file.show(file)
ft1 <- read.ftable(file)
ft1
unlink(file)

## Agresti (1990), page 297, Table 8.16.
## Almost o.k., but misses the name of the row variable.
file <- tempfile()
cat("                      \"Tonsil Size\"\n",
    "            \"Not Enl.\" \"Enl.\" \"Greatly Enl.\"\n",
    "Noncarriers       497     560           269\n",
    "Carriers           19      29            24\n",
    file = file)
file.show(file)
ft <- read.ftable(file, skip = 2,
                  row.var.names = "Status",
                  col.vars = list("Tonsil Size" =
                      c("Not Enl.", "Enl.", "Greatly Enl.")))
ft
unlink(file)

ft22 <- ftable(Titanic, row.vars = 2:1, col.vars = 4:3)
```

```
write.ftable(ft22, quote = FALSE) # is the same as
print(ft22)#method="non.compact" is default
print(ft22, method="row.compact")
print(ft22, method="col.compact")
print(ft22, method="compact")

## using 'justify' and 'quote' :
format(ftable(wool + tension ~ breaks, warpbreaks),
       justify = "none", quote = FALSE)
```

`rect.hclust` *Draw Rectangles Around Hierarchical Clusters*

Description

Draws rectangles around the branches of a dendrogram highlighting the corresponding clusters. First the dendrogram is cut at a certain level, then a rectangle is drawn around selected branches.

Usage

```
rect.hclust(tree, k = NULL, which = NULL, x = NULL, h = NULL,
            border = 2, cluster = NULL)
```

Arguments

`tree`	an object of the type produced by `hclust`.
`k`, `h`	Scalar. Cut the dendrogram such that either exactly `k` clusters are produced or by cutting at height `h`.
`which`, `x`	A vector selecting the clusters around which a rectangle should be drawn. `which` selects clusters by number (from left to right in the tree), `x` selects clusters containing the respective horizontal coordinates. Default is `which = 1:k`.
`border`	Vector with border colors for the rectangles.
`cluster`	Optional vector with cluster memberships as returned by `cutree(hclust.obj, k = k)`, can be specified for efficiency if already computed.

Value

(Invisibly) returns a list where each element contains a vector of data points contained in the respective cluster.

See Also

`hclust`, `identify.hclust`.

Examples

```
require(graphics)

hca <- hclust(dist(USArrests))
plot(hca)
rect.hclust(hca, k = 3, border = "red")
x <- rect.hclust(hca, h = 50, which = c(2,7), border = 3:4)
x
```

---

relevel *Reorder Levels of Factor*

Description

The levels of a factor are re-ordered so that the level specified by `ref` is first and the others are moved down. This is useful for `contr.treatment` contrasts which take the first level as the reference.

Usage

```
relevel(x, ref, ...)
```

Arguments

`x`	an unordered factor.
`ref`	the reference level, typically a string.
`...`	additional arguments for future methods.

Details

This, as `reorder()`, is a special case of simply calling `factor(x, levels = levels(x)[...])`.

Value

A factor of the same length as `x`.

See Also

`factor`, `contr.treatment`, `levels`, `reorder`.

Examples

```
warpbreaks$tension <- relevel(warpbreaks$tension, ref = "M")
summary(lm(breaks ~ wool + tension, data = warpbreaks))
```

`reorder.default` *Reorder Levels of a Factor*

Description

`reorder` is a generic function. The `"default"` method treats its first argument as a categorical variable, and reorders its levels based on the values of a second variable, usually numeric.

Usage

```
reorder(x, ...)

## Default S3 method:
reorder(x, X, FUN = mean, ...,
        order = is.ordered(x), decreasing = FALSE)
```

Arguments

`x`	an atomic vector, usually a `factor` (possibly ordered). The vector is treated as a categorical variable whose levels will be reordered. If `x` is not a factor, its unique values will be used as the implicit levels.
`X`	a vector of the same length as `x`, whose subset of values for each unique level of `x` determines the eventual order of that level.
`FUN`	a `function` whose first argument is a vector and returns a scalar, to be applied to each subset of `X` determined by the levels of `x`.
`...`	optional: extra arguments supplied to `FUN`
`order`	logical, whether return value will be an ordered factor rather than a factor.
`decreasing`	logical, whether the levels will be ordered in increasing or decreasing order.

Details

This, as `relevel()`, is a special case of simply calling `factor(x, levels = levels(x)[....])`.

Value

A factor or an ordered factor (depending on the value of `order`), with the order of the levels determined by `FUN` applied to `X` grouped by `x`. By default, the levels are ordered such that the values returned by `FUN` are in increasing order. Empty levels will be dropped.

Additionally, the values of `FUN` applied to the subsets of `X` (in the original order of the levels of `x`) is returned as the `"scores"` attribute.

Author(s)

Deepayan Sarkar <deepayan.sarkar@r-project.org>

See Also

reorder.dendrogram, levels, relevel.

Examples

```
require(graphics)

bymedian <- with(InsectSprays, reorder(spray, count, median))
boxplot(count ~ bymedian, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE,
        col = "lightgray")

bymedianR <- with(InsectSprays, reorder(spray, count, median, decreasing=TRUE))
stopifnot(exprs = {
    identical(attr(bymedian, "scores") -> sc,
              attr(bymedianR,"scores"))
    identical(nms <- names(sc), LETTERS[1:6])
    identical(levels(bymedian ), nms[isc <- order(sc)])
    identical(levels(bymedianR), nms[rev(isc)])
})
```

---

reorder.dendrogram *Reorder a Dendrogram*

Description

A method for the generic function reorder.

There are many different orderings of a dendrogram that are consistent with the structure imposed. This function takes a dendrogram and a vector of values and reorders the dendrogram in the order of the supplied vector, maintaining the constraints on the dendrogram.

Usage

```
## S3 method for class 'dendrogram'
reorder(x, wts, agglo.FUN = sum, ...)
```

Arguments

x	the (dendrogram) object to be reordered
wts	numeric weights (arbitrary values) for reordering.
agglo.FUN	a function for weights agglomeration, see below.
...	additional arguments

Details

Using the weights `wts`, the leaves of the dendrogram are reordered so as to be in an order as consistent as possible with the weights. At each node, the branches are ordered in increasing weights where the weight of a branch is defined as $f(w_j)$ where f is `agglo.FUN` and w_j is the weight of the j-th sub branch.

Value

A dendrogram where each node has a further attribute `value` with its corresponding weight.

Author(s)

R. Gentleman and M. Maechler

See Also

`reorder`.

`rev.dendrogram` which simply reverses the nodes' order; `heatmap`, `cophenetic`.

Examples

```
require(graphics)

set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
dd <- as.dendrogram(hc)
dd.reorder <- reorder(dd, 10:1)
plot(dd, main = "random dendrogram 'dd'")

op <- par(mfcol = 1:2)
plot(dd.reorder, main = "reorder(dd, 10:1)")
plot(reorder(dd, 10:1, agglo.FUN = mean), main = "reorder(dd, 10:1, mean)")
par(op)
```

`replications` *Number of Replications of Terms*

Description

Returns a vector or a list of the number of replicates for each term in the formula.

Usage

```
replications(formula, data = NULL, na.action)
```

Arguments

formula	a formula or a terms object or a data frame.
data	a data frame used to find the objects in formula.
na.action	function for handling missing values. Defaults to a na.action attribute of data, then a setting of the option na.action, or na.fail if that is not set.

Details

If formula is a data frame and data is missing, formula is used for data with the formula ~ ..

Any character vectors in the formula are coerced to factors.

Value

A vector or list with one entry for each term in the formula giving the number(s) of replications for each level. If all levels are balanced (have the same number of replications) the result is a vector, otherwise it is a list with a component for each terms, as a vector, matrix or array as required.

A test for balance is !is.list(replications(formula,data)).

Author(s)

The design was inspired by the S function of the same name described in Chambers et al. (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

model.tables

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
replications(~ . - yield, npk)
```

reshape *Reshape Grouped Data*

Description

This function reshapes a data frame between 'wide' format (with repeated measurements in separate columns of the same row) and 'long' format (with the repeated measurements in separate rows).

Usage

```
reshape(data, varying = NULL, v.names = NULL, timevar = "time",
        idvar = "id", ids = 1:NROW(data),
        times = seq_along(varying[[1]]),
        drop = NULL, direction, new.row.names = NULL,
        sep = ".",
        split = if (sep == "") {
            list(regexp = "[A-Za-z][0-9]", include = TRUE)
        } else {
            list(regexp = sep, include = FALSE, fixed = TRUE)}
        )

### Typical usage for converting from long to wide format:

# reshape(data, direction = "wide",
#         idvar = "___", timevar = "___", # mandatory
#         v.names = c(___),    # time-varying variables
#         varying = list(___)) # auto-generated if missing

### Typical usage for converting from wide to long format:

### If names of wide-format variables are in a 'nice' format

# reshape(data, direction = "long",
#         varying = c(___), # vector
#         sep)              # to help guess 'v.names' and 'times'

### To specify long-format variable names explicitly

# reshape(data, direction = "long",
#         varying = ___,  # list / matrix / vector (use with care)
#         v.names = ___,  # vector of variable names in long format
#         timevar, times, # name / values of constructed time variable
#         idvar, ids)     # name / values of constructed id variable
```

Arguments

data	a data frame

varying	names of sets of variables in the wide format that correspond to single variables in long format ('time-varying'). This is canonically a list of vectors of variable names, but it can optionally be a matrix of names, or a single vector of names. In each case, when direction = "long", the names can be replaced by indices which are interpreted as referring to names(data). See 'Details' for more details and options.
v.names	names of variables in the long format that correspond to multiple variables in the wide format. See 'Details'.
timevar	the variable in long format that differentiates multiple records from the same group or individual. If more than one record matches, the first will be taken (with a warning).
idvar	Names of one or more variables in long format that identify multiple records from the same group/individual. These variables may also be present in wide format.
ids	the values to use for a newly created idvar variable in long format.
times	the values to use for a newly created timevar variable in long format. See 'Details'.
drop	a vector of names of variables to drop before reshaping.
direction	character string, partially matched to either "wide" to reshape to wide format, or "long" to reshape to long format.
new.row.names	character or NULL: a non-null value will be used for the row names of the result.
sep	A character vector of length 1, indicating a separating character in the variable names in the wide format. This is used for guessing v.names and times arguments based on the names in varying. If sep == "", the split is just before the first numeral that follows an alphabetic character. This is also used to create variable names when reshaping to wide format.
split	A list with three components, regexp, include, and (optionally) fixed. This allows an extended interface to variable name splitting. See 'Details'.

Details

Although reshape() can be used in a variety of contexts, the motivating application is data from longitudinal studies, and the arguments of this function are named and described in those terms. A longitudinal study is characterized by repeated measurements of the same variable(s), e.g., height and weight, on each unit being studied (e.g., individual persons) at different time points (which are assumed to be the same for all units). These variables are called time-varying variables. The study may include other variables that are measured only once for each unit and do not vary with time (e.g., gender and race); these are called time-constant variables.

A 'wide' format representation of a longitudinal dataset will have one record (row) for each unit, typically with some time-constant variables that occupy single columns, and some time-varying variables that occupy multiple columns (one column for each time point). A 'long' format representation of the same dataset will have multiple records (rows) for each individual, with the time-constant variables being constant across these records and the time-varying variables varying across the records. The 'long' format dataset will have two additional variables: a 'time' variable identifying which time point each record comes from, and an 'id' variable showing which records refer to the same unit.

The type of conversion (long to wide or wide to long) is determined by the direction argument, which is mandatory unless the data argument is the result of a previous call to reshape. In that case, the operation can be reversed simply using reshape(data) (the other arguments are stored as attributes on the data frame).

Conversion from long to wide format with direction = "wide" is the simpler operation, and is mainly useful in the context of multivariate analysis where data is often expected as a wide-format matrix. In this case, the time variable timevar and id variable idvar must be specified. All other variables are assumed to be time-varying, unless the time-varying variables are explicitly specified via the v.names argument. A warning is issued if time-constant variables are not actually constant.

Each time-varying variable is expanded into multiple variables in the wide format. The names of these expanded variables are generated automatically, unless they are specified as the varying argument in the form of a list (or matrix) with one component (or row) for each time-varying variable. If varying is a vector of names, it is implicitly converted into a matrix, with one row for each time-varying variable. Use this option with care if there are multiple time-varying variables, as the ordering (by column, the default in the matrix constructor) may be unintuitive, whereas the explicit list or matrix form is unambiguous.

Conversion from wide to long with direction = "long" is the more common operation as most (univariate) statistical modeling functions expect data in the long format. In the simpler case where there is only one time-varying variable, the corresponding columns in the wide format input can be specified as the varying argument, which can be either a vector of column names or the corresponding column indices. The name of the corresponding variable in the long format output combining these columns can be optionally specified as the v.names argument, and the name of the time variables as the timevar argument. The values to use as the time values corresponding to the different columns in the wide format can be specified as the times argument. If v.names is unspecified, the function will attempt to guess v.names and times from varying (an explicitly specified times argument is unused in that case). The default expects variable names like x.1, x.2, where sep = "." specifies to split at the dot and drop it from the name. To have alphabetic followed by numeric times use sep = "".

Multiple time-varying variables can be specified in two ways, either with varying as an atomic vector as above, or as a list (or a matrix). The first form is useful (and mandatory) if the automatic variable name splitting as described above is used; this requires the names of all time-varying variables to be suitably formatted in the same manner, and v.names to be unspecified. If varying is a list (with one component for each time-varying variable) or a matrix (one row for each time-varying variable), variable name splitting is not attempted, and v.names and times will generally need to be specified, although they will default to, respectively, the first variable name in each set, and sequential times.

Also, guessing is not attempted if v.names is given explicitly, even if varying is an atomic vector. In that case, the number of time-varying variables is taken to be the length of v.names, and varying is implicitly converted into a matrix, with one row for each time-varying variable. As in the case of long to wide conversion, the matrix is filled up by column, so careful attention needs to be paid to the order of variable names (or indices) in varying, which is taken to be like x.1, y.1, x.2, y.2 (i.e., variables corresponding to the same time point need to be grouped together).

The split argument should not usually be necessary. The split$regexp component is passed to either strsplit or regexpr, where the latter is used if split$include is TRUE, in which case the splitting occurs after the first character of the matched string. In the strsplit case, the separator is not included in the result, and it is possible to specify fixed-string matching using split$fixed.

Value

The reshaped data frame with added attributes to simplify reshaping back to the original form.

See Also

stack, aperm; relist for reshaping the result of unlist. xtabs and as.data.frame.table for creating contingency tables and converting them back to data frames.

Examples

```
summary(Indometh) # data in long format

## long to wide (direction = "wide") requires idvar and timevar at a minimum
reshape(Indometh, direction = "wide", idvar = "Subject", timevar = "time")

## can also explicitly specify name of combined variable
wide <- reshape(Indometh, direction = "wide", idvar = "Subject",
                timevar = "time", v.names = "conc", sep= "_")
wide

## reverse transformation
reshape(wide, direction = "long")
reshape(wide, idvar = "Subject", varying = list(2:12),
        v.names = "conc", direction = "long")

## times need not be numeric
df <- data.frame(id = rep(1:4, rep(2,4)),
                 visit = I(rep(c("Before","After"), 4)),
                 x = rnorm(4), y = runif(4))
df
reshape(df, timevar = "visit", idvar = "id", direction = "wide")
## warns that y is really varying
reshape(df, timevar = "visit", idvar = "id", direction = "wide", v.names = "x")


##  unbalanced 'long' data leads to NA fill in 'wide' form
df2 <- df[1:7, ]
df2
reshape(df2, timevar = "visit", idvar = "id", direction = "wide")

## Alternative regular expressions for guessing names
df3 <- data.frame(id = 1:4, age = c(40,50,60,50), dose1 = c(1,2,1,2),
                  dose2 = c(2,1,2,1), dose4 = c(3,3,3,3))
reshape(df3, direction = "long", varying = 3:5, sep = "")


## an example that isn't longitudinal data
state.x77 <- as.data.frame(state.x77)
long <- reshape(state.x77, idvar = "state", ids = row.names(state.x77),
                times = names(state.x77), timevar = "Characteristic",
                varying = list(names(state.x77)), direction = "long")
```

```
reshape(long, direction = "wide")

reshape(long, direction = "wide", new.row.names = unique(long$state))

## multiple id variables
df3 <- data.frame(school = rep(1:3, each = 4), class = rep(9:10, 6),
                  time = rep(c(1,1,2,2), 3), score = rnorm(12))
wide <- reshape(df3, idvar = c("school", "class"), direction = "wide")
wide
## transform back
reshape(wide)
```

`residuals` *Extract Model Residuals*

Description

`residuals` is a generic function which extracts model residuals from objects returned by modeling functions.

`resid` is an alias for `residuals`, abbreviated to encourage users to access object components through an accessor function rather than by directly referencing an object slot.

All object classes which are returned by model fitting functions should provide a `residuals` method. (Note that the method is for '`residuals`' and not '`resid`'.)

Methods can make use of `naresid` methods to compensate for the omission of missing values. The default, `nls` and `smooth.spline` methods do.

Usage

```
residuals(object, ...)
resid(object, ...)
```

Arguments

`object`	an object for which the extraction of model residuals is meaningful.
`...`	other arguments.

Value

Residuals extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`coefficients`, `fitted.values`, `glm`, `lm`.

influence.measures for standardized (`rstandard`) and studentized (`rstudent`) residuals.

runmed *Running Medians – Robust Scatter Plot Smoothing*

Description

Compute running medians of odd span. This is the 'most robust' scatter plot smoothing possible. For efficiency (and historical reason), you can use one of two different algorithms giving identical results.

Usage

```
runmed(x, k, endrule = c("median", "keep", "constant"),
       algorithm = NULL,
       na.action = c("+Big_alternate", "-Big_alternate", "na.omit", "fail"),
       print.level = 0)
```

Arguments

x	numeric vector, the 'dependent' variable to be smoothed.
k	integer width of median window; must be odd. Turlach had a default of `k <- 1 + 2 * min((n-1)%/% 2, ceiling(0.1*n))`. Use `k = 3` for 'minimal' robust smoothing eliminating isolated outliers.
endrule	character string indicating how the values at the beginning and the end (of the data) should be treated. Can be abbreviated. Possible values are: `"keep"` keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[j] = x[j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$; `"constant"` copies `median(y[1:k2])` to the first values and analogously for the last ones making the smoothed ends *constant*; `"median"` the default, smooths the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see `smoothEnds`.
algorithm	character string (partially matching `"Turlach"` or `"Stuetzle"`) or the default `NULL`, specifying which algorithm should be applied. The default choice depends on `n = length(x)` and `k` where `"Turlach"` will be used for larger problems.
na.action	character string determining the behavior in the case of `NA` or `NaN` in `x`, (partially matching) one of `"+Big_alternate"` Here, all the NAs in `x` are first replaced by alternating $\pm B$ where B is a "Big" number (with $2B < M*$, where $M* =$ `.Machine $ double.xmax`). The replacement values are "from left" $(+B, -B, +B, \ldots)$, i.e. start with `"+"`. `"-Big_alternate"` almost the same as `"+Big_alternate"`, just starting with $-B$ (`"-Big..."`). `"na.omit"` the result is the same as `runmed(x[!is.na(x)], k, ..)`.

`"fail"` the presence of NAs in `x` will raise an error.

`print.level` integer, indicating verboseness of algorithm; should rarely be changed by average users.

Details

Apart from the end values, the result `y = runmed(x, k)` simply has `y[j] = median(x[(j-k2):(j+k2)])` (`k = 2*k2+1`), computed very efficiently.

The two algorithms are internally entirely different:

`"Turlach"` is the Härdle–Steiger algorithm (see Ref.) as implemented by Berwin Turlach. A tree algorithm is used, ensuring performance $O(n \log k)$ where `n = length(x)` which is asymptotically optimal.

`"Stuetzle"` is the (older) Stuetzle–Friedman implementation which makes use of median *updating* when one observation enters and one leaves the smoothing window. While this performs as $O(n \times k)$ which is slower asymptotically, it is considerably faster for small k or n.

Note that, both algorithms (and the `smoothEnds()` utility) now "work" also when `x` contains non-finite entries ($\pm$`Inf`, `NaN`, and `NA`):

`"Turlach"` .....

`"Stuetzle"` currently simply works by applying the underlying math library ('`libm`') arithmetic for the non-finite numbers; this may optionally change in the future.

Currently long vectors are only supported for `algorithm = "Stuetzle"`.

Value

vector of smoothed values of the same length as `x` with an `attr`ibute `k` containing (the 'oddified') `k`.

Author(s)

Martin Maechler <maechler@stat.math.ethz.ch>, based on Fortran code from Werner Stuetzle and S-PLUS and C code from Berwin Turlach.

References

Härdle, W. and Steiger, W. (1995) Algorithm AS 296: Optimal median smoothing, *Applied Statistics* **44**, 258–264. doi:10.2307/2986349.

Jerome H. Friedman and Werner Stuetzle (1982) *Smoothing of Scatterplots*; Report, Dep. Statistics, Stanford U., Project Orion 003.

See Also

`smoothEnds` which implements Tukey's end point rule and is called by default from `runmed(*, endrule = "median")`. `smooth` uses running medians of 3 for its compound smoothers.

Examples

```
require(graphics)

utils::example(nhtemp)
myNHT <- as.vector(nhtemp)
myNHT[20] <- 2 * nhtemp[20]
plot(myNHT, type = "b", ylim = c(48, 60), main = "Running Medians Example")
lines(runmed(myNHT, 7), col = "red")

## special: multiple y values for one x
plot(cars, main = "'cars' data and runmed(dist, 3)")
lines(cars, col = "light gray", type = "c")
with(cars, lines(speed, runmed(dist, k = 3), col = 2))

## nice quadratic with a few outliers
y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <- c(150, 30, 400, 450)
all(y == runmed(y, 1)) # 1-neighbourhood <==> interpolation
plot(y) ## lines(y, lwd = .1, col = "light gray")
lines(lowess(seq(y), y, f = 0.3), col = "brown")
lines(runmed(y, 7), lwd = 2, col = "blue")
lines(runmed(y, 11), lwd = 2, col = "red")

## Lowess is not robust
y <- ys ; y[21] <- 6666 ; x <- seq(y)
col <- c("black", "brown","blue")
plot(y, col = col[1])
lines(lowess(x, y, f = 0.3), col = col[2])
lines(runmed(y, 7),      lwd = 2, col = col[3])
legend(length(y),max(y), c("data", "lowess(y, f = 0.3)", "runmed(y, 7)"),
       xjust = 1, col = col, lty = c(0, 1, 1), pch = c(1,NA,NA))

## An example with initial NA's - used to fail badly (notably for "Turlach"):
x15 <- c(rep(NA, 4), c(9, 9, 4, 22, 6, 1, 7, 5, 2, 8, 3))
rS15 <- cbind(Sk.3 = runmed(x15, k = 3, algorithm="S"),
              Sk.7 = runmed(x15, k = 7, algorithm="S"),
              Sk.11= runmed(x15, k =11, algorithm="S"))
rT15 <- cbind(Tk.3 = runmed(x15, k = 3, algorithm="T", print.level=1),
              Tk.7 = runmed(x15, k = 7, algorithm="T", print.level=1),
              Tk.9 = runmed(x15, k = 9, algorithm="T", print.level=1),
              Tk.11= runmed(x15, k =11, algorithm="T", print.level=1))
cbind(x15, rS15, rT15) # result for k=11  maybe a bit surprising ..
Tv <- rT15[-(1:3),]
stopifnot(3 <= Tv, Tv <= 9, 5 <= Tv[1:10,])
matplot(y = cbind(x15, rT15), type = "b", ylim = c(1,9), pch=1:5, xlab = NA,
        main = "runmed(x15, k, algo = \"Turlach\")")
mtext(paste("x15 <-", deparse(x15)))
points(x15, cex=2)
legend("bottomleft", legend=c("data", paste("k = ", c(3,7,9,11))),
       bty="n", col=1:5, lty=1:5, pch=1:5)
```

`rWishart` *Random Wishart Distributed Matrices*

Description

Generate `n` random matrices, distributed according to the Wishart distribution with parameters `Sigma` and `df`, $W_p(\Sigma, m),\ m = \texttt{df},\ \Sigma = \texttt{Sigma}$.

Usage

```
rWishart(n, df, Sigma)
```

Arguments

`n`	integer sample size.
`df`	numeric parameter, "degrees of freedom".
`Sigma`	positive definite ($p \times p$) "scale" matrix, the matrix parameter of the distribution.

Details

If $X_1, \ldots, X_m,\ X_i \in \mathbf{R}^p$ is a sample of m independent multivariate Gaussians with mean (vector) 0, and covariance matrix Σ, the distribution of $M = X'X$ is $W_p(\Sigma, m)$.

Consequently, the expectation of M is

$$E[M] = m \times \Sigma.$$

Further, if `Sigma` is scalar ($p = 1$), the Wishart distribution is a scaled chi-squared (χ^2) distribution with `df` degrees of freedom, $W_1(\sigma^2, m) = \sigma^2 \chi^2_m$.

The component wise variance is

$$\mathrm{Var}(M_{ij}) = m(\Sigma^2_{ij} + \Sigma_{ii}\Sigma_{jj}).$$

Value

a numeric `array`, say `R`, of dimension $p \times p \times n$, where each `R[,,i]` is a positive definite matrix, a realization of the Wishart distribution $W_p(\Sigma, m),\ \ m = \texttt{df},\ \Sigma = \texttt{Sigma}$.

Author(s)

Douglas Bates

References

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

See Also

`cov`, `rnorm`, `rchisq`.

Examples

```
## Artificial
S <- toeplitz((10:1)/10)
set.seed(11)
R <- rWishart(1000, 20, S)
dim(R)  #  10 10  1000
mR <- apply(R, 1:2, mean)  # ~= E[ Wish(S, 20) ] = 20 * S
stopifnot(all.equal(mR, 20*S, tolerance = .009))

## See Details, the variance is
Va <- 20*(S^2 + tcrossprod(diag(S)))
vR <- apply(R, 1:2, var)
stopifnot(all.equal(vR, Va, tolerance = 1/16))
```

---

scatter.smooth *Scatter Plot with Smooth Curve Fitted by* loess

---

Description

Plot and add a smooth curve computed by `loess` to a scatter plot.

Usage

```
scatter.smooth(x, y = NULL, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"),
    xlab = NULL, ylab = NULL,
    ylim = range(y, pred$y, na.rm = TRUE),
    evaluation = 50, ..., lpars = list())

loess.smooth(x, y, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"), evaluation = 50, ...)
```

Arguments

x, y	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details.
span	smoothness parameter for `loess`.
degree	degree of local polynomial used.
family	if `"gaussian"` fitting is by least-squares, and if `family = "symmetric"` a redescending M estimator is used. Can be abbreviated.
xlab	label for x axis.
ylab	label for y axis.
ylim	the y limits of the plot.
evaluation	number of points at which to evaluate the smooth curve.

...	For scatter.smooth(), graphical parameters, passed to plot() only. For loess.smooth, control parameters passed to loess.control.
lpars	a list of arguments to be passed to lines().

Details

loess.smooth is an auxiliary function which evaluates the loess smooth at evaluation equally spaced points covering the range of x.

Value

For scatter.smooth, none.

For loess.smooth, a list with two components, x (the grid of evaluation points) and y (the smoothed values at the grid points).

See Also

loess; smoothScatter for scatter plots with smoothed *density* color representation.

Examples

```
require(graphics)

with(cars, scatter.smooth(speed, dist))
## or with dotted thick smoothed line results :
with(cars, scatter.smooth(speed, dist, lpars =
                    list(col = "red", lwd = 3, lty = 3)))
```

---

screeplot *Scree Plots*

Description

screeplot.default plots the variances against the number of the principal component. This is also the plot method for classes "princomp" and "prcomp".

Usage

```
screeplot(x, ...)
## Default S3 method:
screeplot(x, npcs = min(10, length(x$sdev)),
          type = c("barplot", "lines"),
          main = deparse1(substitute(x)), ...)
```

Arguments

x	an object containing a sdev component, such as that returned by princomp() and prcomp().
npcs	the number of components to be plotted.
type	the type of plot. Can be abbreviated.
main, ...	graphics parameters.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

princomp and prcomp.

Examples

```
require(graphics)

## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests, cor = TRUE))  # inappropriate
screeplot(pc.cr)

fit <- princomp(covmat = Harman74.cor)
screeplot(fit)
screeplot(fit, npcs = 24, type = "lines")
```

---

sd *Standard Deviation*

Description

This function computes the standard deviation of the values in x. If na.rm is TRUE then missing values are removed before computation proceeds.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

x	a numeric vector or an R object but not a factor coercible to numeric by as.double(x).
na.rm	logical. Should missing values be removed?

Details

Like var this uses denominator $n - 1$.

The standard deviation of a length-one or zero-length vector is NA.

See Also

var for its square, and mad, the most robust alternative.

Examples

```
sd(1:2) ^ 2
```

---

se.contrast *Standard Errors for Contrasts in Model Terms*

Description

Returns the standard errors for one or more contrasts in an aov object.

Usage

```
se.contrast(object, ...)
## S3 method for class 'aov'
se.contrast(object, contrast.obj,
            coef = contr.helmert(ncol(contrast))[, 1],
            data = NULL, ...)
```

Arguments

object	A suitable fit, usually from aov.
contrast.obj	The contrasts for which standard errors are requested. This can be specified via a list or via a matrix. A single contrast can be specified by a list of logical vectors giving the cells to be contrasted. Multiple contrasts should be specified by a matrix, each column of which is a numerical contrast vector (summing to zero).
coef	used when contrast.obj is a list; it should be a vector of the same length as the list with zero sum. The default value is the first Helmert contrast, which contrasts the first and second cell means specified by the list.
data	The data frame used to evaluate contrast.obj.
...	further arguments passed to or from other methods.

Details

Contrasts are usually used to test if certain means are significantly different; it can be easier to use `se.contrast` than compute them directly from the coefficients.

In multistratum models, the contrasts can appear in more than one stratum, in which case the standard errors are computed in the lowest stratum and adjusted for efficiencies and comparisons between strata. (See the comments in the note in the help for `aov` about using orthogonal contrasts.) Such standard errors are often conservative.

Suitable matrices for use with `coef` can be found by calling `contrasts` and indexing the columns by a factor.

Value

A vector giving the standard errors for each contrast.

See Also

`contrasts`, `model.tables`

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
## Set suitable contrasts.
options(contrasts = c("contr.helmert", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, data = npk)
se.contrast(npk.aov1, list(N == "0", N == "1"), data = npk)
# or via a matrix
cont <- matrix(c(-1,1), 2, 1, dimnames = list(NULL, "N"))
se.contrast(npk.aov1, cont[N, , drop = FALSE]/12, data = npk)

## test a multi-stratum model
npk.aov2 <- aov(yield ~ N + K + Error(block/(N + K)), data = npk)
se.contrast(npk.aov2, list(N == "0", N == "1"))


## an example looking at an interaction contrast
## Dataset from R.E. Kirk (1995)
## 'Experimental Design: procedures for the behavioral sciences'
score <- c(12, 8,10, 6, 8, 4,10,12, 8, 6,10,14, 9, 7, 9, 5,11,12,
           7,13, 9, 9, 5,11, 8, 7, 3, 8,12,10,13,14,19, 9,16,14)
A <- gl(2, 18, labels = c("a1", "a2"))
B <- rep(gl(3, 6, labels = c("b1", "b2", "b3")), 2)
fit <- aov(score ~ A*B)
```

```
cont <- c(1, -1)[A] * c(1, -1, 0)[B]
sum(cont)       # 0
sum(cont*score) # value of the contrast
se.contrast(fit, as.matrix(cont))
(t.stat <- sum(cont*score)/se.contrast(fit, as.matrix(cont)))
summary(fit, split = list(B = 1:2), expand.split = TRUE)
## t.stat^2 is the F value on the A:B: C1 line (with Helmert contrasts)
## Now look at all three interaction contrasts
cont <- c(1, -1)[A] * cbind(c(1, -1, 0), c(1, 0, -1), c(0, 1, -1))[B,]
se.contrast(fit, cont)  # same, due to balance.
rm(A, B, score)


## multi-stratum example where efficiencies play a role
## An example from Yates (1932),
## a 2^3 design in 2 blocks replicated 4 times

Block <- gl(8, 4)
A <- factor(c(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
              0,1,0,1,0,1,0,1,0,1,0,1))
B <- factor(c(0,0,1,1,0,0,1,1,0,1,0,1,1,0,1,0,0,0,1,1,
              0,0,1,1,0,0,1,1,0,0,1,1))
C <- factor(c(0,1,1,0,1,0,0,1,0,0,1,1,0,0,1,1,0,1,0,1,
              1,0,1,0,0,0,1,1,1,1,0,0))
Yield <- c(101, 373, 398, 291, 312, 106, 265, 450, 106, 306, 324, 449,
           272, 89, 407, 338, 87, 324, 279, 471, 323, 128, 423, 334,
           131, 103, 445, 437, 324, 361, 302, 272)
aovdat <- data.frame(Block, A, B, C, Yield)
fit <- aov(Yield ~ A + B * C + Error(Block), data = aovdat)
cont1 <- c(-1, 1)[A]/32  # Helmert contrasts
cont2 <- c(-1, 1)[B] * c(-1, 1)[C]/32
cont <- cbind(A = cont1, BC = cont2)
colSums(cont*Yield) # values of the contrasts
se.contrast(fit, as.matrix(cont))
# comparison with lme
library(nlme)
fit2 <- lme(Yield ~ A + B*C, random = ~1 | Block, data = aovdat)
summary(fit2)$tTable # same estimates, similar (but smaller) se's.
```

`selfStart` *Construct Self-starting Nonlinear Models*

Description

Construct self-starting nonlinear models to be used in `nls`, etc. Via function `initial` to compute approximate parameter values from data, such models are "self-starting", i.e., do not need a `start` argument in, e.g., `nls()`.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

model	a function object defining a nonlinear model or a nonlinear formula object of the form ~ expression.
initial	a function object, taking arguments mCall, data, and LHS, *and* ..., representing, respectively, a matched call to the function model, a data frame in which to interpret the variables in mCall, and the expression from the left-hand side of the model formula in the call to nls. This function should return initial values for the parameters in model. The ... is used by nls() to pass its control and trace arguments for the cases where initial() itself calls nls() as it does for the ten self-starting nonlinear models in R's **stats** package.
parameters	a character vector specifying the terms on the right hand side of model for which initial estimates should be calculated. Passed as the namevec argument to the deriv function.
template	an optional prototype for the calling sequence of the returned object, passed as the function.arg argument to the deriv function. By default, a template is generated with the covariates in model coming first and the parameters in model coming last in the calling sequence.

Details

nls() calls getInitial and the initial function for these self-starting models.

This function is generic; methods functions can be written to handle specific classes of objects.

Value

a function object of class "selfStart", for the formula method obtained by applying deriv to the right hand side of the model formula. An initial attribute (defined by the initial argument) is added to the function to calculate starting estimates for the parameters in the model automatically.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, getInitial.

Each of the following are "selfStart" models (with examples) SSasymp, SSasympOff, SSasympOrig, SSbiexp, SSfol, SSfpl, SSgompertz, SSlogis, SSmicmen, SSweibull.

Further, package **nlme**'s nlsList.

Examples

```
## self-starting logistic model

## The "initializer" (finds initial values for parameters from data):
initLogis <- function(mCall, data, LHS, ...) {
    xy <- sortedXyData(mCall[["x"]], LHS, data)
    if(nrow(xy) < 4)
```

```
        stop("too few distinct input values to fit a logistic model")
    z <- xy[["y"]]
    ## transform to proportion, i.e. in (0,1) :
    rng <- range(z); dz <- diff(rng)
    z <- (z - rng[1L] + 0.05 * dz)/(1.1 * dz)
    xy[["z"]] <- log(z/(1 - z))  # logit transformation
    aux <- coef(lm(x ~ z, xy))
    pars <- coef(nls(y ~ 1/(1 + exp((xmid - x)/scal)),
                     data = xy,
                     start = list(xmid = aux[[1L]], scal = aux[[2L]]),
                     algorithm = "plinear", ...))
    setNames(pars [c(".lin", "xmid", "scal")],
             mCall[c("Asym", "xmid", "scal")])
}

mySSlogis <- selfStart(~ Asym/(1 + exp((xmid - x)/scal)),
                       initial = initLogis,
                       parameters = c("Asym", "xmid", "scal"))

getInitial(weight ~ mySSlogis(Time, Asym, xmid, scal),
           data = subset(ChickWeight, Chick == 1))


# 'first.order.log.model' is a function object defining a first order
# compartment model
# 'first.order.log.initial' is a function object which calculates initial
# values for the parameters in 'first.order.log.model'
#
# self-starting first order compartment model
## Not run:
SSfol <- selfStart(first.order.log.model, first.order.log.initial)

## End(Not run)

## Explore the self-starting models already available in R's  "stats":
pos.st <- which("package:stats" == search())
mSS <- apropos("^SS..", where = TRUE, ignore.case = FALSE)
(mSS <- unname(mSS[names(mSS) == pos.st]))
fSS <- sapply(mSS, get, pos = pos.st, mode = "function")
all(sapply(fSS, inherits, "selfStart"))  # -> TRUE

## Show the argument list of each self-starting function:
str(fSS, give.attr = FALSE)
```

`setNames` *Set the Names in an Object*

Description

This is a convenience function that sets the names on an object and returns the object. It is most useful at the end of a function definition where one is creating the object to be returned and would

prefer not to store it under a name just so the names can be assigned.

Usage

```
setNames(object = nm, nm)
```

Arguments

object	an object for which a names attribute will be meaningful
nm	a character vector of names to assign to the object

Value

An object of the same sort as object with the new names assigned.

Author(s)

Douglas M. Bates and Saikat DebRoy

See Also

unname for removing names.

Examples

```
setNames( 1:3, c("foo", "bar", "baz") )
# this is just a short form of
tmp <- 1:3
names(tmp) <-  c("foo", "bar", "baz")
tmp

## special case of character vector, using default
setNames(nm = c("First", "2nd"))
```

---

shapiro.test *Shapiro-Wilk Normality Test*

Description

Performs the Shapiro-Wilk test of normality.

Usage

```
shapiro.test(x)
```

Arguments

x	a numeric vector of data values. Missing values are allowed, but the number of non-missing values must be between 3 and 5000.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the Shapiro-Wilk statistic.
`p.value`	an approximate p-value for the test. This is said in Royston (1995) to be adequate for `p.value < 0.1`.
`method`	the character string `"Shapiro-Wilk normality test"`.
`data.name`	a character string giving the name(s) of the data.

Source

The algorithm used is a C translation of the Fortran code described in Royston (1995). The calculation of the p value is exact for $n = 3$, otherwise approximations are used, separately for $4 \leq n \leq 11$ and $n \geq 12$.

References

Patrick Royston (1982). An extension of Shapiro and Wilk's W test for normality to large samples. *Applied Statistics*, **31**, 115–124. doi:10.2307/2347973.

Patrick Royston (1982). Algorithm AS 181: The W test for Normality. *Applied Statistics*, **31**, 176–180. doi:10.2307/2347986.

Patrick Royston (1995). Remark AS R94: A remark on Algorithm AS 181: The W test for normality. *Applied Statistics*, **44**, 547–551. doi:10.2307/2986146.

See Also

`qqnorm` for producing a normal quantile-quantile plot.

Examples

```
shapiro.test(rnorm(100, mean = 5, sd = 3))
shapiro.test(runif(100, min = 2, max = 4))
```

`sigma` *Extract Residual Standard Deviation 'Sigma'*

Description

Extract the estimated standard deviation of the errors, the "residual standard deviation" (misnamed also "residual standard error", e.g., in `summary.lm()`'s output, from a fitted model).

Many classical statistical models have a *scale parameter*, typically the standard deviation of a zero-mean normal (or Gaussian) random variable which is denoted as σ. `sigma(.)` extracts the *estimated* parameter from a fitted model, i.e., $\hat{\sigma}$.

Usage

```
sigma(object, ...)

## Default S3 method:
sigma(object, use.fallback = TRUE, ...)
```

Arguments

`object`	an R object, typically resulting from a model fitting function such as `lm`.
`use.fallback`	logical, passed to `nobs`.
`...`	potentially further arguments passed to and from methods. Passed to `deviance(*, ...)` for the default method.

Details

The **stats** package provides the S3 generic, a default method, and a method for objects of class `"glm"`. The default method is correct typically for (asymptotically / approximately) generalized gaussian ("least squares") problems, since it is defined as

```
sigma.default <- function (object, use.fallback = TRUE, ...)
    sqrt( deviance(object, ...) / (NN - PP) )
```

where `NN <- nobs(object, use.fallback = use.fallback)` and `PP <- sum(!is.na(coef(object)))` – where in older R versions this was `length(coef(object))` which is too large in case of undetermined coefficients, e.g., for rank deficient model fits.

Value

Typically a number, the estimated standard deviation of the errors ("residual standard deviation") for Gaussian models, and—less interpretably—the square root of the residual deviance per degree of freedom in more general models.

Very strictly speaking, $\hat{\sigma}$ ("σ hat") is actually $\sqrt{\widehat{\sigma^2}}$.

For generalized linear models (class `"glm"`), the `sigma.glm` method returns the square root of the dispersion parameter (See `summary.glm`). For families with free dispersion parameter, $sigma$ is estimated from the root mean square of the Pearson residuals. For families with fixed dispersion, `sigma` is not estimated from the residuals but extracted directly from the family of the fitted model. Consequently, for binomial or Poisson GLMs, `sigma` is exactly 1.

For multivariate linear models (class `"mlm"`), a *vector* of sigmas is returned, each corresponding to one column of Y.

Note

The misnomer "Residual standard **error**" has been part of too many R (and S) outputs to be easily changed there.

See Also

deviance, nobs, vcov, summary.glm.

Examples

```
## -- lm() ------------------------------
lm1 <- lm(Fertility ~ . , data = swiss)
sigma(lm1) # ~= 7.165  = "Residual standard error"  printed from summary(lm1)
stopifnot(all.equal(sigma(lm1), summary(lm1)$sigma, tolerance=1e-15))

## -- nls() -----------------------------
DNase1 <- subset(DNase, Run == 1)
fm.DN1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
sigma(fm.DN1) # ~= 0.01919  as from summary(..)
stopifnot(all.equal(sigma(fm.DN1), summary(fm.DN1)$sigma, tolerance=1e-15))

## -- glm() -----------------------------
## -- a) Binomial -- Example from MASS
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive = 20-numdead)
sigma(budworm.lg <- glm(SF ~ sex*ldose, family = binomial))

## -- b) Poisson -- from ?glm :
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
sigma(glm.D93 <- glm(counts ~ outcome + treatment, family = poisson()))
## equal to
sqrt(summary(glm.D93)$dispersion) # == 1
## and the *Quasi*poisson's dispersion
sigma(glm.qD93 <- update(glm.D93, family = quasipoisson()))
sigma (glm.qD93)^2 # 1.2933 equal to
summary(glm.qD93)$dispersion # == 1.2933

## -- Multivariate lm() "mlm" -----------
utils::example("SSD", echo=FALSE)
sigma(mlmfit) # is the same as {but more efficient than}
sqrt(diag(estVar(mlmfit)))
```

SignRank *Distribution of the Wilcoxon Signed Rank Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon Signed Rank statistic obtained from a sample with size n.

Usage

```
dsignrank(x, n, log = FALSE)
psignrank(q, n, lower.tail = TRUE, log.p = FALSE)
qsignrank(p, n, lower.tail = TRUE, log.p = FALSE)
rsignrank(nn, n)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
nn	number of observations. If length(nn) > 1, the length is taken to be the number required.
n	number(s) of observations in the sample(s). A positive integer, or a vector of such integers.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let x be a sample of size n from a continuous distribution symmetric about the origin. Then the Wilcoxon signed rank statistic is the sum of the ranks of the absolute values x[i] for which x[i] is positive. This statistic takes values between 0 and $n(n+1)/2$, and its mean and variance are $n(n+1)/4$ and $n(n+1)(2n+1)/24$, respectively.

If either of the first two arguments is a vector, the recycling rule is used to do the calculations for all combinations of the two up to the length of the longer vector.

Value

dsignrank gives the density, psignrank gives the distribution function, qsignrank gives the quantile function, and rsignrank generates random deviates.

The length of the result is determined by nn for rsignrank, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than nn are recycled to the length of the result. Only the first elements of the logical arguments are used.

Author(s)

Kurt Hornik; efficiency improvement by Ivo Ugrina.

See Also

wilcox.test to calculate the statistic from data, find p values and so on.

Distributions for standard distributions, including dwilcox for the distribution of *two-sample* Wilcoxon rank sum statistic.

Examples

```
require(graphics)

par(mfrow = c(2,2))
for(n in c(4:5,10,40)) {
  x <- seq(0, n*(n+1)/2, length.out = 501)
  plot(x, dsignrank(x, n = n), type = "l",
       main = paste0("dsignrank(x, n = ", n, ")"))
}
```

`simulate` *Simulate Responses*

Description

Simulate one or more responses from the distribution corresponding to a fitted model object.

Usage

```
simulate(object, nsim = 1, seed = NULL, ...)
```

Arguments

`object`	an object representing a fitted model.
`nsim`	number of response vectors to simulate. Defaults to `1`.
`seed`	an object specifying if and how the random number generator should be initialized ('seeded'). For the `"lm"` method, either `NULL` or an integer that will be used in a call to `set.seed` before simulating the response vectors. If set, the value is saved as the `"seed"` attribute of the returned value. The default, `NULL` will not change the random generator state, and return `.Random.seed` as the `"seed"` attribute, see 'Value'.
`...`	additional optional arguments.

Details

This is a generic function. Consult the individual modeling functions for details on how to use this function.

Package **stats** has a method for `"lm"` objects which is used for `lm` and `glm` fits. There is a method for fits from `glm.nb` in package **MASS**, and hence the case of negative binomial families is not covered by the `"lm"` method.

The methods for linear models fitted by `lm` or `glm(family = "gaussian")` assume that any weights which have been supplied are inversely proportional to the error variance. For other GLMs the (optional) `simulate` component of the `family` object is used—there is no appropriate simulation method for 'quasi' models as they are specified only up to two moments.

For binomial and Poisson GLMs the dispersion is fixed at one. Integer prior weights w_i can be interpreted as meaning that observation i is an average of w_i observations, which is natural for binomials specified as proportions but less so for a Poisson, for which prior weights are ignored with a warning.

For a gamma GLM the shape parameter is estimated by maximum likelihood (using function `gamma.shape` in package **MASS**). The interpretation of weights is as multipliers to a basic shape parameter, since dispersion is inversely proportional to shape.

For an inverse gaussian GLM the model assumed is $IG(\mu_i, \lambda w_i)$ (see `https://en.wikipedia.org/wiki/Inverse_Gaussian_distribution`) where λ is estimated by the inverse of the dispersion estimate for the fit. The variance is $\mu_i^3/(\lambda w_i)$ and hence inversely proportional to the prior weights. The simulation is done by function `rinvGauss` from the **SuppDists** package, which must be installed.

Value

Typically, a list of length `nsim` of simulated responses. Where appropriate the result can be a data frame (which is a special type of list).

For the `"lm"` method, the result is a data frame with an attribute `"seed"`. If argument `seed` is `NULL`, the attribute is the value of `.Random.seed` before the simulation was started; otherwise it is the value of the argument with a `"kind"` attribute with value `as.list(RNGkind())`.

See Also

`RNG` about random number generation in R, `fitted.values` and `residuals` for related methods; `glm`, `lm` for model fitting.

There are further examples in the 'simulate.R' tests file in the sources for package **stats**.

Examples

```
x <- 1:5
mod1 <- lm(c(1:3, 7, 6) ~ x)
S1 <- simulate(mod1, nsim = 4)
## repeat the simulation:
.Random.seed <- attr(S1, "seed")
identical(S1, simulate(mod1, nsim = 4))

S2 <- simulate(mod1, nsim = 200, seed = 101)
rowMeans(S2) # should be about the same as
fitted(mod1)

## repeat identically:
(sseed <- attr(S2, "seed")) # seed; RNGkind as attribute
stopifnot(identical(S2, simulate(mod1, nsim = 200, seed = sseed)))

## To be sure about the proper RNGkind, e.g., after
RNGversion("2.7.0")
## first set the RNG kind, then simulate
do.call(RNGkind, attr(sseed, "kind"))
identical(S2, simulate(mod1, nsim = 200, seed = sseed))
```

```
## Binomial GLM examples
yb1 <- matrix(c(4, 4, 5, 7, 8, 6, 6, 5, 3, 2), ncol = 2)
modb1 <- glm(yb1 ~ x, family = binomial)
S3 <- simulate(modb1, nsim = 4)
# each column of S3 is a two-column matrix.

x2 <- sort(runif(100))
yb2 <- rbinom(100, prob = plogis(2*(x2-1)), size = 1)
yb2 <- factor(1 + yb2, labels = c("failure", "success"))
modb2 <- glm(yb2 ~ x2, family = binomial)
S4 <- simulate(modb2, nsim = 4)
# each column of S4 is a factor
```

`Smirnov` *Distribution of the Smirnov Statistic*

Description

Distribution function, quantile function and random generation for the distribution of the Smirnov statistic.

Usage

```
psmirnov(q, sizes, z = NULL,
         alternative = c("two.sided", "less", "greater"),
         exact = TRUE, simulate = FALSE, B = 2000,
         lower.tail = TRUE, log.p = FALSE)
qsmirnov(p, sizes, z = NULL,
         alternative = c("two.sided", "less", "greater"),
         exact = TRUE, simulate = FALSE, B = 2000)
rsmirnov(n, sizes, z = NULL,
         alternative = c("two.sided", "less", "greater"))
```

Arguments

`q`	a numeric vector of quantiles.
`p`	a numeric vector of probabilities.
`sizes`	an integer vector of length two giving the sample sizes.
`z`	a numeric vector of the pooled data values in both samples when the exact conditional distribution of the Smirnov statistic given the data shall be computed.
`alternative`	one of `"two.sided"` (default), `"less"`, or `"greater"` indicating whether absolute (two-sided, default) or raw (one-sided) differences of frequencies define the test statistic. See 'Details'.
`exact`	`NULL` or a logical indicating whether the exact (conditional on the pooled data values in `z`) distribution or the asymptotic distribution should be used.
`simulate`	a logical indicating whether to compute the distribution function by Monte Carlo simulation.

`B`	an integer specifying the number of replicates used in the Monte Carlo test.
`lower.tail`	a logical, if `TRUE` (default), probabilities are $P[D < q]$, otherwise, $P[D \geq q]$.
`log.p`	a logical, if `TRUE` (default), probabilities are given as log-probabilities.
`n`	an integer giving number of observations.

Details

For samples x and y with respective sizes n_x and n_y and empirical cumulative distribution functions F_{x,n_x} and F_{y,n_y}, the Smirnov statistic is

$$D = \sup_c |F_{x,n_x}(c) - F_{y,n_y}(c)|$$

in the two-sided case,

$$D^+ = \sup_c (F_{x,n_x}(c) - F_{y,n_y}(c))$$

in the one-sided `"greater"` case, and

$$D^- = \sup_c (F_{y,n_y}(c) - F_{x,n_x}(c))$$

in the one-sided `"less"` case.

These statistics are used in the Smirnov test of the null that x and y were drawn from the same distribution, see `ks.test`.

If the underlying common distribution function F is continuous, the distribution of the test statistics does not depend on F, and has a simple asymptotic approximation. For arbitrary F, one can compute the conditional distribution given the pooled data values z of x and y, either exactly (feasible provided that the product $n_x n_y$ of the sample sizes is "small enough") or approximately Monte Carlo simulation. If the pooled data values z are not specified, a pooled sample without ties is assumed.

Value

`psmirnov` gives the distribution function, `qsmirnov` gives the quantile function, and `rsmirnov` generates random deviates.

See Also

`ks.test` for references on the algorithms used for computing exact distributions.

smooth *Tukey's (Running Median) Smoothing*

Description

Tukey's smoothers, *3RS3R*, *3RSS*, *3R*, etc.

Usage

```
smooth(x, kind = c("3RS3R", "3RSS", "3RSR", "3R", "3", "S"),
       twiceit = FALSE, endrule = c("Tukey", "copy"), do.ends = FALSE)
```

Arguments

x	a vector or time series
kind	a character string indicating the kind of smoother required; defaults to "3RS3R".
twiceit	logical, indicating if the result should be 'twiced'. Twicing a smoother $S(y)$ means $S(y) + S(y - S(y))$, i.e., adding smoothed residuals to the smoothed values. This decreases bias (increasing variance).
endrule	a character string indicating the rule for smoothing at the boundary. Either "Tukey" (default) or "copy".
do.ends	logical, indicating if the 3-splitting of ties should also happen at the boundaries (ends). This is only used for kind = "S".

Details

3 is Tukey's short notation for running medians of length **3**,
3R stands for **R**epeated 3 until convergence, and
S for **S**plitting of horizontal stretches of length 2 or 3.

Hence, 3RS3R is a concatenation of 3R, S and 3R, 3RSS similarly, whereas 3RSR means first 3R and then (S and 3) **R**epeated until convergence – which can be bad.

Value

An object of class "tukeysmooth" (which has print and summary methods) and is a vector or time series containing the smoothed values with additional attributes.

Note

Note that there are other smoothing methods which provide rather better results. These were designed for hand calculations and may be used mainly for didactical purposes.

Since R version 1.2, smooth *does* really implement Tukey's end-point rule correctly (see argument endrule).

kind = "3RSR" had been the default till R-1.1, but it can have very bad properties, see the examples.

Note that repeated application of smooth(*) *does* smooth more, for the "3RS*" kinds.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

`runmed` for running medians; `lowess` and `loess`; `supsmu` and `smooth.spline`.

Examples

```
require(graphics)

## see also   demo(smooth) !

x1 <- c(4, 1, 3, 6, 6, 4, 1, 6, 2, 4, 2) # very artificial
(x3R <- smooth(x1, "3R")) # 2 iterations of "3"
smooth(x3R, kind = "S")

sm.3RS <- function(x, ...)
   smooth(smooth(x, "3R", ...), "S", ...)

y <- c(1, 1, 19:1)
plot(y, main = "misbehaviour of \"3RSR\"", col.main = 3)
lines(sm.3RS(y))
lines(smooth(y))
lines(smooth(y, "3RSR"), col = 3, lwd = 2)  # the horror

x <- c(8:10, 10, 0, 0, 9, 9)
plot(x, main = "breakdown of  3R  and  S  and hence  3RSS")
matlines(cbind(smooth(x, "3R"), smooth(x, "S"), smooth(x, "3RSS"), smooth(x)))

presidents[is.na(presidents)] <- 0 # silly
summary(sm3 <- smooth(presidents, "3R"))
summary(sm2 <- smooth(presidents,"3RSS"))
summary(sm  <- smooth(presidents))

all.equal(c(sm2), c(smooth(smooth(sm3, "S"), "S")))  # 3RSS  === 3R S S
all.equal(c(sm),  c(smooth(smooth(sm3, "S"), "3R"))) # 3RS3R === 3R S 3R

plot(presidents, main = "smooth(presidents0, *) :  3R and default 3RS3R")
lines(sm3, col = 3, lwd = 1.5)
lines(sm, col = 2, lwd = 1.25)
```

`smooth.spline` *Fit a Smoothing Spline*

Description

Fits a cubic smoothing spline to the supplied data.

Usage

```
smooth.spline(x, y = NULL, w = NULL, df, spar = NULL, lambda = NULL, cv = FALSE,
              all.knots = FALSE, nknots = .nknots.smspl,
              keep.data = TRUE, df.offset = 0, penalty = 1,
              control.spar = list(), tol = 1e-6 * IQR(x), keep.stuff = FALSE)

.nknots.smspl(n)
```

Arguments

x	a vector giving the values of the predictor variable, or a list or a two-column matrix specifying x and y.
y	responses. If y is missing or NULL, the responses are assumed to be specified by x, with x the index vector.
w	optional vector of weights of the same length as x; defaults to all 1.
df	the desired equivalent number of degrees of freedom (trace of the smoother matrix). Must be in $(1, n_x]$, n_x the number of unique x values, see below.
spar	smoothing parameter, typically (but not necessarily) in $(0, 1]$. When spar is specified, the coefficient λ of the integral of the squared second derivative in the fit (penalized log likelihood) criterion is a monotone function of spar, see the details below. Alternatively lambda may be specified instead of the *scale free* spar=s.
lambda	if desired, the internal (design-dependent) smoothing parameter λ can be specified instead of spar. This may be desirable for resampling algorithms such as cross validation or the bootstrap.
cv	ordinary leave-one-out (TRUE) or 'generalized' cross-validation (GCV) when FALSE; is used for smoothing parameter computation only when both spar and df are not specified; it is used however to determine cv.crit in the result. Setting it to NA for speedup skips the evaluation of leverages and any score.
all.knots	if TRUE, all distinct points in x are used as knots. If FALSE (default), a subset of x[] is used, specifically x[j] where the nknots indices are evenly spaced in 1:n, see also the next argument nknots. Alternatively, a strictly increasing numeric vector specifying "all the knots" to be used; must be rescaled to $[0, 1]$ already such that it corresponds to the ans $ fit$knots sequence returned, not repeating the boundary knots.
nknots	integer or function giving the number of knots to use when all.knots = FALSE. If a function (as by default), the number of knots is nknots(nx). By default using .nknots.smspl(), for $n_x > 49$ this is less than n_x, the number of unique x values, see the Note.
keep.data	logical specifying if the input data should be kept in the result. If TRUE (as per default), fitted values and residuals are available from the result.
df.offset	allows the degrees of freedom to be increased by df.offset in the GCV criterion.
penalty	the coefficient of the penalty for degrees of freedom in the GCV criterion.

`control.spar` optional list with named components controlling the root finding when the smoothing parameter `spar` is computed, i.e., missing or `NULL`, see below.

Note that this is partly *experimental* and may change with general spar computation improvements!

low: lower bound for `spar`; defaults to -1.5 (used to implicitly default to 0 in R versions earlier than 1.4).

high: upper bound for `spar`; defaults to +1.5.

tol: the absolute precision (**tol**erance) used; defaults to 1e-4 (formerly 1e-3).

eps: the relative precision used; defaults to 2e-8 (formerly 0.00244).

trace: logical indicating if iterations should be traced.

maxit: integer giving the maximal number of iterations; defaults to 500.

Note that `spar` is only searched for in the interval $[low, high]$.

`tol` a tolerance for sameness or uniqueness of the `x` values. The values are binned into bins of size `tol` and values which fall into the same bin are regarded as the same. Must be strictly positive (and finite).

`keep.stuff` an experimental `logical` indicating if the result should keep extras from the internal computations. Should allow to reconstruct the X matrix and more.

`n` for `.nknots.smspl`; typically the number of unique `x` values (aka n_x).

Details

Neither `x` nor `y` are allowed to containing missing or infinite values.

The `x` vector should contain at least four distinct values. 'Distinct' here is controlled by `tol`: values which are regarded as the same are replaced by the first of their values and the corresponding `y` and `w` are pooled accordingly.

Unless `lambda` has been specified instead of `spar`, the computational λ used (as a function of $s = spar$) is $\lambda = r \cdot 256^{3s-1}$ where $r = tr(X'WX)/tr(\Sigma)$, Σ is the matrix given by $\Sigma_{ij} = \int B_i''(t)B_j''(t)dt$, X is given by $X_{ij} = B_j(x_i)$, W is the diagonal matrix of weights (scaled such that its trace is n, the original number of observations) and $B_k(.)$ is the k-th B-spline.

Note that with these definitions, $f_i = f(x_i)$, and the B-spline basis representation $f = Xc$ (i.e., c is the vector of spline coefficients), the penalized log likelihood is $L = (y - f)'W(y - f) + \lambda c'\Sigma c$, and hence c is the solution of the (ridge regression) $(X'WX + \lambda\Sigma)c = X'Wy$.

If `spar` and `lambda` are missing or `NULL`, the value of `df` is used to determine the degree of smoothing. If `df` is missing as well, leave-one-out cross-validation (ordinary or 'generalized' as determined by `cv`) is used to determine λ.

Note that from the above relation, `spar` is $s = s0 + 0.0601 \cdot \log \lambda$.

Note however that currently the results may become very unreliable for `spar` values smaller than about -1 or -2. The same may happen for values larger than 2 or so. Don't think of setting `spar` or the controls `low` and `high` outside such a safe range, unless you know what you are doing! Similarly, specifying `lambda` instead of `spar` is delicate, notably as the range of "safe" values for `lambda` is not scale-invariant and hence entirely data dependent.

The 'generalized' cross-validation method GCV will work correctly when there are duplicated points in `x`. However, it is ambiguous what leave-one-out cross-validation means with duplicated points, and the internal code uses an approximation that involves leaving out groups of duplicated points. `cv = TRUE` is best avoided in that case.

Value

An object of class "smooth.spline" with components

x	the *distinct* x values in increasing order, see the 'Details' above.
y	the fitted values corresponding to x.
w	the weights used at the unique values of x.
yin	the y values used at the unique y values.
tol	the tol argument (whose default depends on x).
data	only if keep.data = TRUE: itself a list with components x, y and w of the same length. These are the original $(x_i, y_i, w_i), i = 1, \ldots, n$, values where data$x may have repeated values and hence be longer than the above x component; see details.
n	an integer; the (original) sample size.
lev	(when cv was not NA) leverages, the diagonal values of the smoother matrix.
cv	the cv argument used; i.e., FALSE, TRUE, or NA.
cv.crit	cross-validation score, 'generalized' or true, depending on cv. The CV score is often called "PRESS" (and labeled on print()), for '**PRE**diction **S**um of **S**quares'. Note that this is *not* the same as the (CV or GCV) score which is minimized during fitting (and returned in crit), e.g., in the case of nx < n (where nx= n_x is the number of unique x values).
pen.crit	the penalized criterion, a non-negative number; simply the (weighted) residual sum of squares (RSS), sum(.$w * residuals(.)^2) .
crit	the criterion value minimized in the underlying .Fortran routine 'sslvrg'. When df has been specified, the criterion is $3 + (tr(S_\lambda) - df)^2$, where the $3+$ is there for numerical (and historical) reasons.
df	equivalent degrees of freedom used. Note that (currently) this value may become quite imprecise when the true df is between and 1 and 2.
spar	the value of spar computed or given, unless it has been given as c(lambda = *), when it set to NA here.
ratio	(when spar above is not NA), the value r, the ratio of two matrix traces.
lambda	the value of λ corresponding to spar, see the details above.
iparms	named integer(3) vector where ..$ipars["iter"] gives number of spar computing iterations used.
auxMat	experimental; when keep.stuff was true, a "flat" numeric vector containing parts of the internal computations.
fit	list for use by predict.smooth.spline, with components **knot:** the knot sequence (including the repeated boundary knots), scaled into $[0, 1]$ (via min and range). **nk:** number of coefficients or number of 'proper' knots plus 2. **coef:** coefficients for the spline basis used. **min, range:** numbers giving the corresponding quantities of x.
call	the matched call.

method(class = "smooth.spline") shows a hatvalues() method based on the lev vector above.

Note

The number of unique x values, nx $= n_x$, are determined by the tol argument, equivalently to

```
nx <- length(x) - sum(duplicated( round((x - mean(x)) / tol) ))
```

The default all.knots = FALSE and nknots = .nknots.smspl, entails using only $O({n_x}^{0.2})$ knots instead of n_x for $n_x > 49$. This cuts speed and memory requirements, but not drastically anymore since R version 1.5.1 where it is only $O(n_k) + O(n)$ where n_k is the number of knots.

In this case where not all unique x values are used as knots, the result is a *regression spline* rather than a smoothing spline in the strict sense, but very close unless a small smoothing parameter (or large df) is used.

Author(s)

R implementation by B. D. Ripley and Martin Maechler (spar/lambda, etc).

Source

This function is based on code in the GAMFIT Fortran program by T. Hastie and R. Tibshirani (originally taken from http://lib.stat.cmu.edu/general/gamfit) which makes use of spline code by Finbarr O'Sullivan. Its design parallels the smooth.spline function of Chambers & Hastie (1992).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

Green, P. J. and Silverman, B. W. (1994) *Nonparametric Regression and Generalized Linear Models: A Roughness Penalty Approach.* Chapman and Hall.

Hastie, T. J. and Tibshirani, R. J. (1990) *Generalized Additive Models.* Chapman and Hall.

See Also

predict.smooth.spline for evaluating the spline and its derivatives.

Examples

```
require(graphics)
plot(dist ~ speed, data = cars, main = "data(cars)  &  smoothing splines")
cars.spl <- with(cars, smooth.spline(speed, dist))
cars.spl
## This example has duplicate points, so avoid cv = TRUE

lines(cars.spl, col = "blue")
ss10 <- smooth.spline(cars[,"speed"], cars[,"dist"], df = 10)
lines(ss10, lty = 2, col = "red")
legend(5,120,c(paste("default [C.V.] => df =",round(cars.spl$df,1)),
               "s( * , df = 10)"), col = c("blue","red"), lty = 1:2,
       bg = 'bisque')
```

```
## Residual (Tukey Anscombe) plot:
plot(residuals(cars.spl) ~ fitted(cars.spl))
abline(h = 0, col = "gray")

## consistency check:
stopifnot(all.equal(cars$dist,
                    fitted(cars.spl) + residuals(cars.spl)))
## The chosen inner knots in original x-scale :
with(cars.spl$fit, min + range * knot[-c(1:3, nk+1 +1:3)]) # == unique(cars$speed)

## Visualize the behavior of  .nknots.smspl()
nKnots <- Vectorize(.nknots.smspl) ; c.. <- adjustcolor("gray20",.5)
curve(nKnots, 1, 250, n=250)
abline(0,1, lty=2, col=c..); text(90,90,"y = x", col=c.., adj=-.25)
abline(h=100,lty=2); abline(v=200, lty=2)

n <- c(1:799, seq(800, 3490, by=10), seq(3500, 10000, by = 50))
plot(n, nKnots(n), type="l", main = "Vectorize(.nknots.smspl) (n)")
abline(0,1, lty=2, col=c..); text(180,180,"y = x", col=c..)
n0 <- c(50, 200, 800, 3200); c0 <- adjustcolor("blue3", .5)
lines(n0, nKnots(n0), type="h", col=c0)
axis(1, at=n0, line=-2, col.ticks=c0, col=NA, col.axis=c0)
axis(4, at=.nknots.smspl(10000), line=-.5, col=c..,col.axis=c.., las=1)

##-- artificial example
y18 <- c(1:3, 5, 4, 7:3, 2*(2:5), rep(10, 4))
xx  <- seq(1, length(y18), length.out = 201)
(s2   <- smooth.spline(y18)) # GCV
(s02  <- smooth.spline(y18, spar = 0.2))
(s02. <- smooth.spline(y18, spar = 0.2, cv = NA))
plot(y18, main = deparse(s2$call), col.main = 2)
lines(s2, col = "gray"); lines(predict(s2, xx), col = 2)
lines(predict(s02, xx), col = 3); mtext(deparse(s02$call), col = 3)

## Specifying 'lambda' instead of usual spar :
(s2. <- smooth.spline(y18, lambda = s2$lambda, tol = s2$tol))



## The following shows the problematic behavior of 'spar' searching:
(s2  <- smooth.spline(y18, control =
                      list(trace = TRUE, tol = 1e-6, low = -1.5)))
(s2m <- smooth.spline(y18, cv = TRUE, control =
                      list(trace = TRUE, tol = 1e-6, low = -1.5)))
## both above do quite similarly (Df = 8.5 +- 0.2)
```

---

smoothEnds *End Points Smoothing (for Running Medians)*

Description

Smooth end points of a vector `y` using subsequently smaller medians and Tukey's end point rule at the very end. (of odd span),

Usage

```
smoothEnds(y, k = 3)
```

Arguments

`y`	dependent variable to be smoothed (vector).
`k`	width of largest median window; must be odd.

Details

`smoothEnds` is used to only do the 'end point smoothing', i.e., change at most the observations closer to the beginning/end than half the window `k`. The first and last value are computed using *Tukey's end point rule*, i.e., `sm[1] = median(y[1], sm[2], 3*sm[2] - 2*sm[3], na.rm=TRUE)`.

In R versions 3.6.0 and earlier, missing values (`NA`) in `y` typically lead to an error, whereas now the equivalent of `median(*, na.rm=TRUE)` is used.

Value

vector of smoothed values, the same length as `y`.

Author(s)

Martin Maechler

References

John W. Tukey (1977) *Exploratory Data Analysis*, Addison.

Velleman, P.F., and Hoaglin, D.C. (1981) *ABC of EDA (Applications, Basics, and Computing of Exploratory Data Analysis)*; Duxbury.

See Also

`runmed(*, endrule = "median")` which calls `smoothEnds()`.

Examples

```
require(graphics)

y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <-  c(100, 30, 400, 470)
s7k <- runmed(y, 7, endrule = "keep")
s7. <- runmed(y, 7, endrule = "const")
s7m <- runmed(y, 7)
col3 <- c("midnightblue","blue","steelblue")
```

```
plot(y, main = "Running Medians -- runmed(*, k=7, endrule = X)")
lines(ys, col = "light gray")
matlines(cbind(s7k, s7.,s7m), lwd = 1.5, lty = 1, col = col3)
eRules <- c("keep","constant","median")
legend("topleft", paste("endrule", eRules, sep = " = "),
       col = col3, lwd = 1.5, lty = 1, bty = "n")

stopifnot(identical(s7m, smoothEnds(s7k, 7)))

## With missing values (for R >= 3.6.1):
yN <- y; yN[c(2,40)] <- NA
rN <- sapply(eRules, function(R) runmed(yN, 7, endrule=R))
matlines(rN, type = "b", pch = 4, lwd = 3, lty=2,
         col = adjustcolor(c("red", "orange4", "orange1"), 0.5))
yN[c(1, 20:21)] <- NA # additionally
rN. <- sapply(eRules, function(R) runmed(yN, 7, endrule=R))
head(rN., 4); tail(rN.) # more NA's too, still not *so* many:
stopifnot(exprs = {
   !anyNA(rN[,2:3])
   identical(which(is.na(rN[,"keep"])), c(2L, 40L))
   identical(which(is.na(rN.), arr.ind=TRUE, useNames=FALSE),
             cbind(c(1:2,40L), 1L))
   identical(rN.[38:41, "median"], c(289,289, 397, 470))
})
```

`sortedXyData` *Create a* `sortedXyData` *Object*

Description

This is a constructor function for the class of `sortedXyData` objects. These objects are mostly used in the `initial` function for a self-starting nonlinear regression model, which will be of the `selfStart` class.

Usage

```
sortedXyData(x, y, data)
```

Arguments

`x`	a numeric vector or an expression that will evaluate in `data` to a numeric vector
`y`	a numeric vector or an expression that will evaluate in `data` to a numeric vector
`data`	an optional data frame in which to evaluate expressions for `x` and `y`, if they are given as expressions

Value

A `sortedXyData` object. This is a data frame with exactly two numeric columns, named `x` and `y`. The rows are sorted so the `x` column is in increasing order. Duplicate `x` values are eliminated by averaging the corresponding `y` values.

Author(s)

José Pinheiro and Douglas Bates

See Also

selfStart, NLSstClosestX, NLSstLfAsymptote, NLSstRtAsymptote

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
sortedXyData( expression(log(conc)), expression(density), DNase.2 )
```

---

spec.ar *Estimate Spectral Density of a Time Series from AR Fit*

Description

Fits an AR model to x (or uses the existing fit) and computes (and by default plots) the spectral density of the fitted model.

Usage

```
spec.ar(x, n.freq, order = NULL, plot = TRUE, na.action = na.fail,
        method = "yule-walker", ...)
```

Arguments

x	A univariate (not yet:or multivariate) time series or the result of a fit by ar.
n.freq	The number of points at which to plot.
order	The order of the AR model to be fitted. If omitted, the order is chosen by AIC.
plot	Plot the periodogram?
na.action	NA action function.
method	method for ar fit.
...	Graphical arguments passed to plot.spec.

Value

An object of class "spec". The result is returned invisibly if plot is true.

Warning

Some authors, for example Thomson (1990), warn strongly that AR spectra can be misleading.

Note

The multivariate case is not yet implemented.

References

Thompson, D.J. (1990). Time series analysis of Holocene climate data. *Philosophical Transactions of the Royal Society of London Series A*, **330**, 601–616. doi:10.1098/rsta.1990.0041.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer. (Especially page 402.)

See Also

ar, spectrum.

Examples

```
require(graphics)

spec.ar(lh)

spec.ar(ldeaths)
spec.ar(ldeaths, method = "burg")

spec.ar(log(lynx))
spec.ar(log(lynx), method = "burg", add = TRUE, col = "purple")
spec.ar(log(lynx), method = "mle", add = TRUE, col = "forest green")
spec.ar(log(lynx), method = "ols", add = TRUE, col = "blue")
```

---

spec.pgram — *Estimate Spectral Density of a Time Series by a Smoothed Periodogram*

Description

spec.pgram calculates the periodogram using a fast Fourier transform, and optionally smooths the result with a series of modified Daniell smoothers (moving averages giving half weight to the end values).

Usage

```
spec.pgram(x, spans = NULL, kernel, taper = 0.1,
           pad = 0, fast = TRUE, demean = FALSE, detrend = TRUE,
           plot = TRUE, na.action = na.fail, ...)
```

Arguments

x	univariate or multivariate time series.
spans	vector of odd integers giving the widths of modified Daniell smoothers to be used to smooth the periodogram.
kernel	alternatively, a kernel smoother of class "tskernel".

taper	specifies the proportion of data to taper. A split cosine bell taper is applied to this proportion of the data at the beginning and end of the series.
pad	proportion of data to pad. Zeros are added to the end of the series to increase its length by the proportion pad.
fast	logical; if TRUE, pad the series to a highly composite length.
demean	logical. If TRUE, subtract the mean of the series.
detrend	logical. If TRUE, remove a linear trend from the series. This will also remove the mean.
plot	plot the periodogram?
na.action	NA action function.
...	graphical arguments passed to plot.spec.

Details

The raw periodogram is not a consistent estimator of the spectral density, but adjacent values are asymptotically independent. Hence a consistent estimator can be derived by smoothing the raw periodogram, assuming that the spectral density is smooth.

The series will be automatically padded with zeros until the series length is a highly composite number in order to help the Fast Fourier Transform. This is controlled by the fast and not the pad argument.

The periodogram at zero is in theory zero as the mean of the series is removed (but this may be affected by tapering): it is replaced by an interpolation of adjacent values during smoothing, and no value is returned for that frequency.

Value

A list object of class "spec" (see spectrum) with the following additional components:

kernel	The kernel argument, or the kernel constructed from spans.
df	The distribution of the spectral density estimate can be approximated by a (scaled) chi square distribution with df degrees of freedom.
bandwidth	The equivalent bandwidth of the kernel smoother as defined by Bloomfield (1976, page 201).
taper	The value of the taper argument.
pad	The value of the pad argument.
detrend	The value of the detrend argument.
demean	The value of the demean argument.

The result is returned invisibly if plot is true.

Author(s)

Originally Martyn Plummer; kernel smoothing by Adrian Trapletti, synthesis by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer. (Especially pp. 392–7.)

See Also

`spectrum`, `spec.taper`, `plot.spec`, `fft`

Examples

```
require(graphics)

## Examples from Venables & Ripley
spectrum(ldeaths)
spectrum(ldeaths, spans = c(3,5))
spectrum(ldeaths, spans = c(5,7))
spectrum(mdeaths, spans = c(3,3))
spectrum(fdeaths, spans = c(3,3))

## bivariate example
mfdeaths.spc <- spec.pgram(ts.union(mdeaths, fdeaths), spans = c(3,3))
# plots marginal spectra: now plot coherency and phase
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

## now impose a lack of alignment
mfdeaths.spc <- spec.pgram(ts.intersect(mdeaths, lag(fdeaths, 4)),
   spans = c(3,3), plot = FALSE)
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

stocks.spc <- spectrum(EuStockMarkets, kernel("daniell", c(30,50)),
                       plot = FALSE)
plot(stocks.spc, plot.type = "marginal") # the default type
plot(stocks.spc, plot.type = "coherency")
plot(stocks.spc, plot.type = "phase")

sales.spc <- spectrum(ts.union(BJsales, BJsales.lead),
                      kernel("modified.daniell", c(5,7)))
plot(sales.spc, plot.type = "coherency")
plot(sales.spc, plot.type = "phase")
```

`spec.taper` *Taper a Time Series by a Cosine Bell*

Description

Apply a cosine-bell taper to a time series.

Usage

```
spec.taper(x, p = 0.1)
```

Arguments

x	A univariate or multivariate time series
p	The proportion to be tapered at each end of the series, either a scalar (giving the proportion for all series) or a vector of the length of the number of series (giving the proportion for each series).

Details

The cosine-bell taper is applied to the first and last `p[i]` observations of time series `x[, i]`.

Value

A new time series object.

See Also

`spec.pgram`, `cpgram`

`spectrum` *Spectral Density Estimation*

Description

The `spectrum` function estimates the spectral density of a time series.

Usage

```
spectrum(x, ..., method = c("pgram", "ar"))
```

Arguments

x	A univariate or multivariate time series.
method	String specifying the method used to estimate the spectral density. Allowed methods are `"pgram"` (the default) and `"ar"`. Can be abbreviated.
...	Further arguments to specific spec methods or `plot.spec`.

Details

`spectrum` is a wrapper function which calls the methods `spec.pgram` and `spec.ar`.

The spectrum here is defined (for historical compatibility) with scaling `1/frequency(x)`. This makes the spectral density a density over the range `(-frequency(x)/2, +frequency(x)/2]`, whereas a more common scaling is 2π and range $(-0.5, 0.5]$ (e.g., Bloomfield) or 1 and range $(-\pi, \pi]$.

If available, a confidence interval will be plotted by `plot.spec`: this is asymmetric, and the width of the centre mark indicates the equivalent bandwidth.

Value

An object of class `"spec"`, which is a list containing at least the following components:

`freq`	vector of frequencies at which the spectral density is estimated. (Possibly approximate Fourier frequencies.) The units are the reciprocal of cycles per unit time (and not per observation spacing): see 'Details' below.
`spec`	Vector (for univariate series) or matrix (for multivariate series) of estimates of the spectral density at frequencies corresponding to `freq`.
`coh`	`NULL` for univariate series. For multivariate time series, a matrix containing the *squared* coherency between different series. Column $i + (j-1) * (j-2)/2$ of `coh` contains the squared coherency between columns i and j of `x`, where $i < j$.
`phase`	`NULL` for univariate series. For multivariate time series a matrix containing the cross-spectrum phase between different series. The format is the same as `coh`.
`series`	The name of the time series.
`snames`	For multivariate input, the names of the component series.
`method`	The method used to calculate the spectrum.

The result is returned invisibly if `plot` is true.

Note

The default plot for objects of class `"spec"` is quite complex, including an error bar and default title, subtitle and axis labels. The defaults can all be overridden by supplying the appropriate graphical parameters.

Author(s)

Martyn Plummer, B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS.* Fourth edition. Springer. (Especially pages 392–7.)

See Also

spec.ar, spec.pgram; plot.spec.

Examples

```
require(graphics)

## Examples from Venables & Ripley
## spec.pgram
par(mfrow = c(2,2))
spectrum(lh)
spectrum(lh, spans = 3)
spectrum(lh, spans = c(3,3))
spectrum(lh, spans = c(3,5))

spectrum(ldeaths)
spectrum(ldeaths, spans = c(3,3))
spectrum(ldeaths, spans = c(3,5))
spectrum(ldeaths, spans = c(5,7))
spectrum(ldeaths, spans = c(5,7), log = "dB", ci = 0.8)

# for multivariate examples see the help for spec.pgram

## spec.ar
spectrum(lh, method = "ar")
spectrum(ldeaths, method = "ar")
```

---

splinefun *Interpolating Splines*

Description

Perform cubic (or Hermite) spline interpolation of given data points, returning either a list of points obtained by the interpolation or a *function* performing the interpolation.

Usage

```
splinefun(x, y = NULL,
          method = c("fmm", "periodic", "natural", "monoH.FC", "hyman"),
          ties = mean)

spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x), xout, ties = mean)

splinefunH(x, y, m)
```

Arguments

x, y	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see xy.coords. y must be increasing or decreasing for method = "hyman".
m	(for splinefunH()): vector of *slopes* m_i at the points (x_i, y_i); these together determine the **H**ermite "spline" which is piecewise cubic, (only) *once* differentiable continuously.
method	specifies the type of spline to be used. Possible values are "fmm", "natural", "periodic", "monoH.FC" and "hyman". Can be abbreviated.
n	if xout is left unspecified, interpolation takes place at n equally spaced points spanning the interval [xmin, xmax].
xmin, xmax	left-hand and right-hand endpoint of the interpolation interval (when xout is unspecified).
xout	an optional set of values specifying where interpolation is to take place.
ties	handling of tied x values. The string "ordered" or a function (or the name of a function) taking a single vector argument and returning a single number or a length-2 list of both, see approx and its 'Details' section, and the example below.

Details

The inputs can contain missing values which are deleted, so at least one complete (x, y) pair is required. If method = "fmm", the spline used is that of Forsythe, Malcolm and Moler (an exact cubic is fitted through the four points at each end of the data, and this is used to determine the end conditions). Natural splines are used when method = "natural", and periodic splines when method = "periodic".

The method "monoH.FC" computes a *monotone* Hermite spline according to the method of Fritsch and Carlson. It does so by determining slopes such that the Hermite spline, determined by (x_i, y_i, m_i), is monotone (increasing or decreasing) **iff** the data are.

Method "hyman" computes a *monotone* cubic spline using Hyman filtering of an method = "fmm" fit for strictly monotonic inputs.

These interpolation splines can also be used for extrapolation, that is prediction at points outside the range of x. Extrapolation makes little sense for method = "fmm"; for natural splines it is linear using the slope of the interpolating curve at the nearest data point.

Value

spline returns a list containing components x and y which give the ordinates where interpolation took place and the interpolated values.

splinefun returns a function with formal arguments x and deriv, the latter defaulting to zero. This function can be used to evaluate the interpolating cubic spline (deriv = 0), or its derivatives (deriv = 1, 2, 3) at the points x, where the spline function interpolates the data points originally specified. It uses data stored in its environment when it was created, the details of which are subject to change.

Warning

The value returned by `splinefun` contains references to the code in the current version of R: it is not intended to be saved and loaded into a different R session. This is safer in R >= 3.0.0.

Author(s)

R Core Team.

Simon Wood for the original code for Hyman filtering.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole.

Dougherty, R. L., Edelman, A. and Hyman, J. M. (1989) Positivity-, monotonicity-, or convexity-preserving cubic and quintic Hermite interpolation. *Mathematics of Computation*, **52**, 471–494. doi:10.1090/S0025571819890922091.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B. (1977). *Computer Methods for Mathematical Computations*. Wiley.

Fritsch, F. N. and Carlson, R. E. (1980). Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, **17**, 238–246. doi:10.1137/0717021.

Hyman, J. M. (1983). Accurate monotonicity preserving cubic interpolation. *SIAM Journal on Scientific and Statistical Computing*, **4**, 645–654. doi:10.1137/0904045.

See Also

`approx` and `approxfun` for constant and linear interpolation.

Package **splines**, especially `interpSpline` and `periodicSpline` for interpolation splines. That package also generates spline bases that can be used for regression splines.

`smooth.spline` for smoothing splines.

Examples

```
require(graphics)

op <- par(mfrow = c(2,1), mgp = c(2,.8,0), mar = 0.1+c(3,3,3,1))
n <- 9
x <- 1:n
y <- rnorm(n)
plot(x, y, main = paste("spline[fun](.) through", n, "points"))
lines(spline(x, y))
lines(spline(x, y, n = 201), col = 2)

y <- (x-6)^2
plot(x, y, main = "spline(.) -- 3 methods")
lines(spline(x, y, n = 201), col = 2)
lines(spline(x, y, n = 201, method = "natural"), col = 3)
lines(spline(x, y, n = 201, method = "periodic"), col = 4)
legend(6, 25, c("fmm","natural","periodic"), col = 2:4, lty = 1)
```

```
y <- sin((x-0.5)*pi)
f <- splinefun(x, y)
ls(envir = environment(f))
splinecoef <- get("z", envir = environment(f))
curve(f(x), 1, 10, col = "green", lwd = 1.5)
points(splinecoef, col = "purple", cex = 2)
curve(f(x, deriv = 1), 1, 10, col = 2, lwd = 1.5)
curve(f(x, deriv = 2), 1, 10, col = 2, lwd = 1.5, n = 401)
curve(f(x, deriv = 3), 1, 10, col = 2, lwd = 1.5, n = 401)
par(op)

## Manual spline evaluation --- demo the coefficients :
.x <- splinecoef$x
u <- seq(3, 6, by = 0.25)
(ii <- findInterval(u, .x))
dx <- u - .x[ii]
f.u <- with(splinecoef,
            y[ii] + dx*(b[ii] + dx*(c[ii] + dx* d[ii])))
stopifnot(all.equal(f(u), f.u))

## An example with ties (non-unique  x values):
set.seed(1); x <- round(rnorm(30), 1); y <- sin(pi * x) + rnorm(30)/10
plot(x, y, main = "spline(x,y)  when x has ties")
lines(spline(x, y, n = 201), col = 2)
## visualizes the non-unique ones:
tx <- table(x); mx <- as.numeric(names(tx[tx > 1]))
ry <- matrix(unlist(tapply(y, match(x, mx), range, simplify = FALSE)),
             ncol = 2, byrow = TRUE)
segments(mx, ry[, 1], mx, ry[, 2], col = "blue", lwd = 2)

## Another example with sorted x, but ties:
set.seed(8); x <- sort(round(rnorm(30), 1)); y <- round(sin(pi * x) + rnorm(30)/10, 3)
summary(diff(x) == 0) # -> 7 duplicated x-values
str(spline(x, y, n = 201, ties="ordered")) # all '$y' entries are NaN
## The default (ties=mean) is ok, but most efficient to use instead is
sxyo <- spline(x, y, n = 201, ties= list("ordered", mean))
sapply(sxyo, summary)# all fine now
plot(x, y, main = "spline(x,y, ties=list(\"ordered\", mean))  for when x has ties")
lines(sxyo, col="blue")

## An example of monotone interpolation
n <- 20
set.seed(11)
x. <- sort(runif(n)) ; y. <- cumsum(abs(rnorm(n)))
plot(x., y.)
curve(splinefun(x., y.)(x), add = TRUE, col = 2, n = 1001)
curve(splinefun(x., y., method = "monoH.FC")(x), add = TRUE, col = 3, n = 1001)
curve(splinefun(x., y., method = "hyman")   (x), add = TRUE, col = 4, n = 1001)
legend("topleft",
       paste0("splinefun( \"", c("fmm", "monoH.FC", "hyman"), "\" )"),
       col = 2:4, lty = 1, bty = "n")
```

```
## and one from Fritsch and Carlson (1980), Dougherty et al (1989)
x. <- c(7.09, 8.09, 8.19, 8.7, 9.2, 10, 12, 15, 20)
f <- c(0, 2.76429e-5, 4.37498e-2, 0.169183, 0.469428, 0.943740,
       0.998636, 0.999919, 0.999994)
s0 <- splinefun(x., f)
s1 <- splinefun(x., f, method = "monoH.FC")
s2 <- splinefun(x., f, method = "hyman")
plot(x., f, ylim = c(-0.2, 1.2))
curve(s0(x), add = TRUE, col = 2, n = 1001) -> m0
curve(s1(x), add = TRUE, col = 3, n = 1001)
curve(s2(x), add = TRUE, col = 4, n = 1001)
legend("right",
       paste0("splinefun( \"", c("fmm", "monoH.FC", "hyman"), "\" )"),
       col = 2:4, lty = 1, bty = "n")

## they seem identical, but are not quite:
xx <- m0$x
plot(xx, s1(xx) - s2(xx), type = "l",  col = 2, lwd = 2,
     main = "Difference   monoH.FC - hyman"); abline(h = 0, lty = 3)

x <- xx[xx < 10.2] ## full range: x <- xx .. does not show enough
ccol <- adjustcolor(2:4, 0.8)
matplot(x, cbind(s0(x, deriv = 2), s1(x, deriv = 2), s2(x, deriv = 2))^2,
        lwd = 2, col = ccol, type = "l", ylab = quote({{f*second}(x)}^2),
        main = expression({{f*second}(x)}^2 ~" for the three 'splines'"))
legend("topright",
       paste0("splinefun( \"", c("fmm", "monoH.FC", "hyman"), "\" )"),
       lwd = 2, col  =  ccol, lty = 1:3, bty = "n")
## --> "hyman" has slightly smaller  Integral f''(x)^2 dx  than "FC",
## here, and both are 'much worse' than the regular fmm spline.
```

SSasymp *Self-Starting* nls *Asymptotic Model*

Description

This selfStart model evaluates the asymptotic regression function and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Asym, R0, and lrc for a given set of data.

Note that SSweibull() generalizes this asymptotic model with an extra parameter.

Usage

```
SSasymp(input, Asym, R0, lrc)
```

Arguments

input	a numeric vector of values at which to evaluate the model.

Asym	a numeric parameter representing the horizontal asymptote on the right side (very large values of input).
R0	a numeric parameter representing the response when input is zero.
lrc	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as input. It is the value of the expression Asym+(R0-Asym)*exp(-exp(lrc)*input). If all of the arguments Asym, R0, and lrc are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasymp( Lob.329$age, 100, -8.5, -3.2 )   # response only
local({
  Asym <- 100 ; resp0 <- -8.5 ; lrc <- -3.2
  SSasymp( Lob.329$age, Asym, resp0, lrc) # response _and_ gradient
})
getInitial(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
summary(fm1)

## Visualize the SSasymp()  model  parametrization :

  xx <- seq(-.3, 5, length.out = 101)
  ##  Asym + (R0-Asym) * exp(-exp(lrc)* x) :
  yy <- 5 - 4 * exp(-xx / exp(3/4))
  stopifnot( all.equal(yy, SSasymp(xx, Asym = 5, R0 = 1, lrc = -3/4)) )
  require(graphics)
  op <- par(mar = c(0, .2, 4.1, 0))
  plot(xx, yy, type = "l", axes = FALSE, ylim = c(0,5.2), xlim = c(-.3, 5),
       xlab = "", ylab = "", lwd = 2,
       main = quote("Parameters in the SSasymp model " ~
                    {f[phi](x) == phi[1] + (phi[2]-phi[1])*~e^{-e^{phi[3]}*~x}}))
  mtext(quote(list(phi[1] == "Asym", phi[2] == "R0", phi[3] == "lrc")))
  usr <- par("usr")
  arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
  arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
  text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
  text(     -0.1, usr[4], "y", adj = c(1, 1))
  abline(h = 5, lty = 3)
```

```
arrows(c(0.35, 0.65), 1,
       c(0  ,  1   ), 1, length = 0.08, angle = 25); text(0.5, 1, quote(1))
y0 <- 1 + 4*exp(-3/4) ; t.5 <- log(2) / exp(-3/4) ; AR2 <- 3 # (Asym + R0)/2
segments(c(1, 1), c( 1, y0),
         c(1, 0), c(y0,  1),  lty = 2, lwd = 0.75)
text(1.1, 1/2+y0/2, quote((phi[1]-phi[2])*e^phi[3]), adj = c(0,.5))
axis(2, at = c(1,AR2,5), labels= expression(phi[2], frac(phi[1]+phi[2],2), phi[1]),
     pos=0, las=1)
arrows(c(.6,t.5-.6), AR2,
       c(0, t.5   ), AR2, length = 0.08, angle = 25)
text(   t.5/2,   AR2, quote(t[0.5]))
text(   t.5 +.4, AR2,
     quote({f(t[0.5]) == frac(phi[1]+phi[2],2)}~{} %=>% {}~~
             {t[0.5] == frac(log(2), e^{phi[3]})}), adj = c(0, 0.5))
par(op)
```

`SSasympOff` *Self-Starting* `nls` *Asymptotic Model with an Offset*

Description

This `selfStart` model evaluates an alternative parametrization of the asymptotic regression function and the gradient with respect to those parameters. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `lrc`, and `c0`.

Usage

```
SSasympOff(input, Asym, lrc, c0)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the horizontal asymptote on the right side (very large values of `input`).
`lrc`	a numeric parameter representing the natural logarithm of the rate constant.
`c0`	a numeric parameter representing the `input` for which the response is zero.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym*(1 - exp(-exp(lrc)*(input - c0)))`. If all of the arguments `Asym`, `lrc`, and `c0` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart; example(SSasympOff) gives graph showing the SSasympOff parametrization.

Examples

```
CO2.Qn1 <- CO2[CO2$Plant == "Qn1", ]
SSasympOff(CO2.Qn1$conc, 32, -4, 43)  # response only
local({  Asym <- 32; lrc <- -4; c0 <- 43
  SSasympOff(CO2.Qn1$conc, Asym, lrc, c0) # response and gradient
})
getInitial(uptake ~ SSasympOff(conc, Asym, lrc, c0), data = CO2.Qn1)
## Initial values are in fact the converged values
fm1 <- nls(uptake ~ SSasympOff(conc, Asym, lrc, c0), data = CO2.Qn1)
summary(fm1)

## Visualize the SSasympOff()  model  parametrization :

  xx <- seq(0.25, 8,  by=1/16)
  yy <- 5 * (1 -  exp(-(xx - 3/4)*0.4))
  stopifnot( all.equal(yy, SSasympOff(xx, Asym = 5, lrc = log(0.4), c0 = 3/4)) )
  require(graphics)
  op <- par(mar = c(0, 0, 4.0, 0))
  plot(xx, yy, type = "l", axes = FALSE, ylim = c(-.5,6), xlim = c(-1, 8),
       xlab = "", ylab = "", lwd = 2,
       main = "Parameters in the SSasympOff model")
  mtext(quote(list(phi[1] == "Asym", phi[2] == "lrc", phi[3] == "c0")))
  usr <- par("usr")
  arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
  arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
  text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
  text(     -0.1, usr[4], "y", adj = c(1, 1))
  abline(h = 5, lty = 3)
  arrows(-0.8, c(2.1, 2.9),
         -0.8, c(0  , 5  ), length = 0.1, angle = 25)
  text  (-0.8, 2.5, quote(phi[1]))
  segments(3/4, -.2, 3/4, 1.6, lty = 2)
  text    (3/4,    c(-.3, 1.7), quote(phi[3]))
  arrows(c(1.1, 1.4), -.15,
         c(3/4, 7/4), -.15, length = 0.07, angle = 25)
  text    (3/4 + 1/2, -.15, quote(1))
  segments(c(3/4, 7/4, 7/4), c(0, 0, 2),   # 5 * exp(log(0.4)) = 2
           c(7/4, 7/4, 3/4), c(0, 2, 0),  lty = 2, lwd = 2)
  text(      7/4 +.1, 2./2, quote(phi[1]*e^phi[2]), adj = c(0, .5))
  par(op)
```

SSasympOrig *Self-Starting* nls *Asymptotic Model through the Origin*

Description

This selfStart model evaluates the asymptotic regression function through the origin and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Asym and lrc for a given set of data.

Usage

```
SSasympOrig(input, Asym, lrc)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote.
lrc	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*(1 - exp(-exp(lrc)*input)). If all of the arguments Asym and lrc are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasympOrig(Lob.329$age, 100, -3.2)  # response only
local({   Asym <- 100; lrc <- -3.2
  SSasympOrig(Lob.329$age, Asym, lrc) # response and gradient
})
getInitial(height ~ SSasympOrig(age, Asym, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasympOrig(age, Asym, lrc), data = Lob.329)
summary(fm1)


## Visualize the SSasympOrig()  model  parametrization :

  xx <- seq(0, 5, length.out = 101)
  yy <- 5 * (1- exp(-xx * log(2)))
  stopifnot( all.equal(yy, SSasympOrig(xx, Asym = 5, lrc = log(log(2)))) )

  require(graphics)
  op <- par(mar = c(0, 0, 3.5, 0))
```

```
plot(xx, yy, type = "l", axes = FALSE, ylim = c(0,5), xlim = c(-1/4, 5),
     xlab = "", ylab = "", lwd = 2,
     main = quote("Parameters in the SSasympOrig model"~~ f[phi](x)))
mtext(quote(list(phi[1] == "Asym", phi[2] == "lrc")))
usr <- par("usr")
arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
text(   -0.1,   usr[4], "y", adj = c(1, 1))
abline(h = 5, lty = 3)
axis(2, at = 5*c(1/2,1), labels= expression(frac(phi[1],2), phi[1]), pos=0, las=1)
arrows(c(.3,.7), 5/2,
       c(0, 1 ), 5/2, length = 0.08, angle = 25)
text(   0.5,     5/2, quote(t[0.5]))
text(   1 +.4,   5/2,
   quote({f(t[0.5]) == frac(phi[1],2)}~{} %=>% {}~~{t[0.5] == frac(log(2), e^{phi[2]})}),
     adj = c(0, 0.5))
par(op)
```

SSbiexp *Self-Starting* nls *Biexponential Model*

Description

This selfStart model evaluates the biexponential model function and its gradient. It has an initial attribute that creates initial estimates of the parameters A1, lrc1, A2, and lrc2.

Usage

```
SSbiexp(input, A1, lrc1, A2, lrc2)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
A1	a numeric parameter representing the multiplier of the first exponential.
lrc1	a numeric parameter representing the natural logarithm of the rate constant of the first exponential.
A2	a numeric parameter representing the multiplier of the second exponential.
lrc2	a numeric parameter representing the natural logarithm of the rate constant of the second exponential.

Value

a numeric vector of the same length as input. It is the value of the expression A1*exp(-exp(lrc1)*input)+A2*exp(-exp(lrc2)*input). If all of the arguments A1, lrc1, A2, and lrc2 are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Indo.1 <- Indometh[Indometh$Subject == 1, ]
SSbiexp( Indo.1$time, 3, 1, 0.6, -1.3 )  # response only
A1 <- 3; lrc1 <- 1; A2 <- 0.6; lrc2 <- -1.3
SSbiexp( Indo.1$time, A1, lrc1, A2, lrc2 ) # response and gradient
print(getInitial(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1),
      digits = 5)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
summary(fm1)

## Show the model components visually
  require(graphics)

  xx <- seq(0, 5, length.out = 101)
  y1 <- 3.5 * exp(-4*xx)
  y2 <- 1.5 * exp(-xx)
  plot(xx, y1 + y2, type = "l", lwd=2, ylim = c(-0.2,6), xlim = c(0, 5),
       main = "Components of the SSbiexp model")
  lines(xx, y1, lty = 2, col="tomato"); abline(v=0, h=0, col="gray40")
  lines(xx, y2, lty = 3, col="blue2" )
  legend("topright", c("y1+y2", "y1 = 3.5 * exp(-4*x)", "y2 = 1.5 * exp(-x)"),
         lty=1:3, col=c("black","tomato","blue2"), bty="n")
  axis(2, pos=0, at = c(3.5, 1.5), labels = c("A1","A2"), las=2)

## and how you could have got their sum via SSbiexp():
  ySS <- SSbiexp(xx, 3.5, log(4), 1.5, log(1))
  ##                      ---          ---
  stopifnot(all.equal(y1+y2, ySS, tolerance = 1e-15))

## Show a no-noise example
datN <- data.frame(time = (0:600)/64)
datN$conc <- predict(fm1, newdata=datN)
plot(conc ~ time, data=datN) # perfect, no noise

## Fails by default (scaleOffset=0) on most platforms {also after increasing maxiter !}
## Not run:
        nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = datN, trace=TRUE)
## End(Not run)

fmX1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = datN,
            control = list(scaleOffset=1))
fmX  <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = datN,
        control = list(scaleOffset=1, printEval=TRUE, tol=1e-11, nDcentral=TRUE), trace=TRUE)
```

```
all.equal(coef(fm1), coef(fmX1), tolerance=0) # ... rel.diff.: 1.57e-6
all.equal(coef(fm1), coef(fmX),  tolerance=0) # ... rel.diff.: 1.03e-12

stopifnot(all.equal(coef(fm1), coef(fmX1), tolerance = 6e-6),
          all.equal(coef(fm1), coef(fmX ), tolerance = 1e-11))
```

SSD *SSD Matrix and Estimated Variance Matrix in Multivariate Models*

Description

Functions to compute matrix of residual sums of squares and products, or the estimated variance matrix for multivariate linear models.

Usage

```
# S3 method for class 'mlm'
SSD(object, ...)

# S3 methods for class 'SSD' and 'mlm'
estVar(object, ...)
```

Arguments

object	object of class "mlm", or "SSD" in the case of estVar.
...	Unused

Value

SSD() returns a list of class "SSD" containing the following components

SSD	The residual sums of squares and products matrix
df	Degrees of freedom
call	Copied from object

estVar returns a matrix with the estimated variances and covariances.

See Also

mauchly.test, anova.mlm

Examples

```
# Lifted from Baron+Li:
# "Notes on the use of R for psychology experiments and questionnaires"
# Maxwell and Delaney, p. 497
reacttime <- matrix(c(
420, 420, 480, 480, 600, 780,
420, 480, 480, 360, 480, 600,
480, 480, 540, 660, 780, 780,
420, 540, 540, 480, 780, 900,
540, 660, 540, 480, 660, 720,
360, 420, 360, 360, 480, 540,
480, 480, 600, 540, 720, 840,
480, 600, 660, 540, 720, 900,
540, 600, 540, 480, 720, 780,
480, 420, 540, 540, 660, 780),
ncol = 6, byrow = TRUE,
dimnames = list(subj = 1:10,
              cond = c("deg0NA", "deg4NA", "deg8NA",
                       "deg0NP", "deg4NP", "deg8NP")))

mlmfit <- lm(reacttime ~ 1)
SSD(mlmfit)
estVar(mlmfit)
```

SSfol *Self-Starting* nls *First-order Compartment Model*

Description

This selfStart model evaluates the first-order compartment function and its gradient. It has an initial attribute that creates initial estimates of the parameters lKe, lKa, and lCl.

Usage

```
SSfol(Dose, input, lKe, lKa, lCl)
```

Arguments

Dose	a numeric value representing the initial dose.
input	a numeric vector at which to evaluate the model.
lKe	a numeric parameter representing the natural logarithm of the elimination rate constant.
lKa	a numeric parameter representing the natural logarithm of the absorption rate constant.
lCl	a numeric parameter representing the natural logarithm of the clearance.

Value

a numeric vector of the same length as `input`, which is the value of the expression

```
Dose * exp(lKe+lKa-lCl) * (exp(-exp(lKe)*input) - exp(-exp(lKa)*input))
    / (exp(lKa) - exp(lKe))
```

If all of the arguments `lKe`, `lKa`, and `lCl` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

José Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Theoph.1 <- Theoph[ Theoph$Subject == 1, ]
with(Theoph.1, SSfol(Dose, Time, -2.5, 0.5, -3)) # response only
with(Theoph.1, local({  lKe <- -2.5; lKa <- 0.5; lCl <- -3
  SSfol(Dose, Time, lKe, lKa, lCl) # response _and_ gradient
}))
getInitial(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
summary(fm1)
```

SSfpl *Self-Starting* `nls` *Four-Parameter Logistic Model*

Description

This `selfStart` model evaluates the four-parameter logistic function and its gradient. It has an `initial` attribute computing initial estimates of the parameters `A`, `B`, `xmid`, and `scal` for a given set of data.

Usage

```
SSfpl(input, A, B, xmid, scal)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
A	a numeric parameter representing the horizontal asymptote on the left side (very small values of input).
B	a numeric parameter representing the horizontal asymptote on the right side (very large values of input).
xmid	a numeric parameter representing the input value at the inflection point of the curve. The value of SSfpl will be midway between A and B at xmid.
scal	a numeric scale parameter on the input axis.

Value

a numeric vector of the same length as input. It is the value of the expression A+(B-A)/(1+exp((xmid-input)/scal)). If all of the arguments A, B, xmid, and scal are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSfpl(Chick.1$Time, 13, 368, 14, 6)  # response only
local({
  A <- 13; B <- 368; xmid <- 14; scal <- 6
  SSfpl(Chick.1$Time, A, B, xmid, scal) # response _and_ gradient
})
print(getInitial(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1),
      digits = 5)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
summary(fm1)

## Visualizing the  SSfpl()  parametrization
  xx <- seq(-0.5, 5, length.out = 101)
  yy <- 1 + 4 / (1 + exp((2-xx))) # == SSfpl(xx, *) :
  stopifnot( all.equal(yy, SSfpl(xx, A = 1, B = 5, xmid = 2, scal = 1)) )
  require(graphics)
  op <- par(mar = c(0, 0, 3.5, 0))
  plot(xx, yy, type = "l", axes = FALSE, ylim = c(0,6), xlim = c(-1, 5),
       xlab = "", ylab = "", lwd = 2,
       main = "Parameters in the SSfpl model")
  mtext(quote(list(phi[1] == "A", phi[2] == "B", phi[3] == "xmid", phi[4] == "scal")))
  usr <- par("usr")
  arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
```

```
arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
text(     -0.1, usr[4], "y", adj = c(1, 1))
abline(h = c(1, 5), lty = 3)
arrows(-0.8, c(2.1, 2.9),
       -0.8, c(0,   5  ), length = 0.1, angle = 25)
text  (-0.8, 2.5, quote(phi[1]))
arrows(-0.3, c(1/4, 3/4),
       -0.3, c(0,   1  ), length = 0.07, angle = 25)
text  (-0.3, 0.5, quote(phi[2]))
text(2, -.1, quote(phi[3]))
segments(c(2,3,3), c(0,3,4), # SSfpl(x = xmid = 2) = 3
         c(2,3,2), c(3,4,3),    lty = 2, lwd = 0.75)
arrows(c(2.3, 2.7), 3,
       c(2.0, 3  ), 3, length = 0.08, angle = 25)
text(      2.5,     3, quote(phi[4])); text(3.1, 3.5, "1")
par(op)
```

`SSgompertz` *Self-Starting* `nls` *Gompertz Growth Model*

Description

This `selfStart` model evaluates the Gompertz growth model and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `b2`, and `b3`.

Usage

```
SSgompertz(x, Asym, b2, b3)
```

Arguments

`x`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the asymptote.
`b2`	a numeric parameter related to the value of the function at `x = 0`
`b3`	a numeric parameter related to the scale the `x` axis.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym*exp(-b2*b3^x)`. If all of the arguments `Asym`, `b2`, and `b3` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Douglas Bates

See Also

nls, selfStart

Examples

```
DNase.1 <- subset(DNase, Run == 1)
SSgompertz(log(DNase.1$conc), 4.5, 2.3, 0.7)  # response only
local({  Asym <- 4.5; b2 <- 2.3; b3 <- 0.7
  SSgompertz(log(DNase.1$conc), Asym, b2, b3) # response _and_ gradient
})
print(getInitial(density ~ SSgompertz(log(conc), Asym, b2, b3),
                 data = DNase.1), digits = 5)
## Initial values are in fact the converged values
fm1 <- nls(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
summary(fm1)
plot(density ~ log(conc), DNase.1, # xlim = c(0, 21),
     main = "SSgompertz() fit to DNase.1")
ux <- par("usr")[1:2]; x <- seq(ux[1], ux[2], length.out=250)
lines(x, do.call(SSgompertz, c(list(x=x), coef(fm1))), col = "red", lwd=2)
As <- coef(fm1)[["Asym"]]; abline(v = 0, h = 0, lty = 3)
axis(2, at= exp(-coef(fm1)[["b2"]]), quote(e^{-b[2]}), las=1, pos=0)
```

SSlogis *Self-Starting* nls *Logistic Model*

Description

This selfStart model evaluates the logistic function and its gradient. It has an initial attribute that creates initial estimates of the parameters Asym, xmid, and scal. In R 3.4.2 and earlier, that init function failed when min(input) was exactly zero.

Usage

```
SSlogis(input, Asym, xmid, scal)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the asymptote.
xmid	a numeric parameter representing the x value at the inflection point of the curve. The value of SSlogis will be Asym/2 at xmid.
scal	a numeric scale parameter on the input axis.

Value

a numeric vector of the same length as input. It is the value of the expression Asym/(1+exp((xmid-input)/scal)). If all of the arguments Asym, xmid, and scal are names of objects the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSlogis(Chick.1$Time, 368, 14, 6)  # response only
local({
  Asym <- 368; xmid <- 14; scal <- 6
  SSlogis(Chick.1$Time, Asym, xmid, scal) # response _and_ gradient
})
getInitial(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
## Initial values are in fact the converged one here, "Number of iter...: 0" :
fm1 <- nls(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
summary(fm1)
## but are slightly improved here:
fm2 <- update(fm1, control=nls.control(tol = 1e-9, warnOnly=TRUE), trace = TRUE)
all.equal(coef(fm1), coef(fm2)) # "Mean relative difference: 9.6e-6"
str(fm2$convInfo) # 3 iterations


dwlg1 <- data.frame(Prop = c(rep(0,5), 2, 5, rep(9, 9)), end = 1:16)
iPar <- getInitial(Prop ~ SSlogis(end, Asym, xmid, scal), data = dwlg1)
## failed in R <= 3.4.2 (because of the '0's in 'Prop')
stopifnot(all.equal(tolerance = 1e-6,
   iPar, c(Asym = 9.0678, xmid = 6.79331, scal = 0.499934)))

## Visualize the SSlogis()  model  parametrization :
  xx <- seq(-0.75, 5, by=1/32)
  yy <- 5 / (1 + exp((2-xx)/0.6)) # == SSlogis(xx, *):
  stopifnot( all.equal(yy, SSlogis(xx, Asym = 5, xmid = 2, scal = 0.6)) )
  require(graphics)
  op <- par(mar = c(0.5, 0, 3.5, 0))
  plot(xx, yy, type = "l", axes = FALSE, ylim = c(0,6), xlim = c(-1, 5),
       xlab = "", ylab = "", lwd = 2,
       main = "Parameters in the SSlogis model")
  mtext(quote(list(phi[1] == "Asym", phi[2] == "xmid", phi[3] == "scal")))
  usr <- par("usr")
  arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
  arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
  text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
  text(     -0.1, usr[4], "y", adj = c(1, 1))
  abline(h = 5, lty = 3)
  arrows(-0.8, c(2.1, 2.9),
         -0.8, c(0,   5  ), length = 0.1, angle = 25)
  text  (-0.8, 2.5, quote(phi[1]))
  segments(c(2,2.6,2.6), c(0,  2.5,3.5),   # NB.  SSlogis(x = xmid = 2) = 2.5
           c(2,2.6,2  ), c(2.5,3.5,2.5), lty = 2, lwd = 0.75)
```

```
text(2, -.1, quote(phi[2]))
arrows(c(2.2, 2.4), 2.5,
       c(2.0, 2.6), 2.5, length = 0.08, angle = 25)
text(      2.3,     2.5, quote(phi[3])); text(2.7, 3, "1")
par(op)
```

SSmicmen *Self-Starting* nls *Michaelis-Menten Model*

Description

This selfStart model evaluates the Michaelis-Menten model and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Vm and K

Usage

```
SSmicmen(input, Vm, K)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Vm	a numeric parameter representing the maximum value of the response.
K	a numeric parameter representing the input value at which half the maximum response is attained. In the field of enzyme kinetics this is called the Michaelis parameter.

Value

a numeric vector of the same length as input. It is the value of the expression Vm*input/(K+input). If both the arguments Vm and K are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

José Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
SSmicmen(PurTrt$conc, 200, 0.05)  # response only
local({  Vm <- 200; K <- 0.05
  SSmicmen(PurTrt$conc, Vm, K)    # response _and_ gradient
})
print(getInitial(rate ~ SSmicmen(conc, Vm, K), data = PurTrt), digits = 3)
```

```
## Initial values are in fact the converged values
fm1 <- nls(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
summary(fm1)
## Alternative call using the subset argument
fm2 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm2) # The same indeed:
stopifnot(all.equal(coef(summary(fm1)), coef(summary(fm2))))

## Visualize the SSmicmen()  Michaelis-Menton model parametrization :

  xx <- seq(0, 5, length.out = 101)
  yy <- 5 * xx/(1+xx)
  stopifnot(all.equal(yy, SSmicmen(xx, Vm = 5, K = 1)))
  require(graphics)
  op <- par(mar = c(0, 0, 3.5, 0))
  plot(xx, yy, type = "l", lwd = 2, ylim = c(-1/4,6), xlim = c(-1, 5),
       ann = FALSE, axes = FALSE, main = "Parameters in the SSmicmen model")
  mtext(quote(list(phi[1] == "Vm", phi[2] == "K")))
  usr <- par("usr")
  arrows(usr[1], 0, usr[2], 0, length = 0.1, angle = 25)
  arrows(0, usr[3], 0, usr[4], length = 0.1, angle = 25)
  text(usr[2] - 0.2, 0.1, "x", adj = c(1, 0))
  text(     -0.1, usr[4], "y", adj = c(1, 1))
  abline(h = 5, lty = 3)
  arrows(-0.8, c(2.1, 2.9),
         -0.8, c(0,   5  ),  length = 0.1, angle = 25)
  text(  -0.8,     2.5, quote(phi[1]))
  segments(1, 0, 1, 2.7, lty = 2, lwd = 0.75)
  text(1, 2.7, quote(phi[2]))
  par(op)
```

SSweibull *Self-Starting* nls *Weibull Growth Curve Model*

Description

This selfStart model evaluates the Weibull model for growth curve data and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Asym, Drop, lrc, and pwr for a given set of data.

Usage

```
SSweibull(x, Asym, Drop, lrc, pwr)
```

Arguments

x	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote on the right side (very small values of x).

Drop	a numeric parameter representing the change from Asym to the y intercept.
lrc	a numeric parameter representing the natural logarithm of the rate constant.
pwr	a numeric parameter representing the power to which x is raised.

Details

This model is a generalization of the SSasymp model in that it reduces to SSasymp when pwr is unity.

Value

a numeric vector of the same length as x. It is the value of the expression Asym-Drop*exp(-exp(lrc)*x^pwr). If all of the arguments Asym, Drop, lrc, and pwr are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Douglas Bates

References

Ratkowsky, David A. (1983), *Nonlinear Regression Modeling*, Dekker. (section 4.4.5)

See Also

nls, selfStart, SSasymp

Examples

```
Chick.6 <- subset(ChickWeight, (Chick == 6) & (Time > 0))
SSweibull(Chick.6$Time, 160, 115, -5.5, 2.5)   # response only
local({ Asym <- 160; Drop <- 115; lrc <- -5.5; pwr <- 2.5
  SSweibull(Chick.6$Time, Asym, Drop, lrc, pwr) # response _and_ gradient
})

getInitial(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
summary(fm1)

## Data and Fit:
plot(weight ~ Time, Chick.6, xlim = c(0, 21), main = "SSweibull() fit to Chick.6")
ux <- par("usr")[1:2]; x <- seq(ux[1], ux[2], length.out=250)
lines(x, do.call(SSweibull, c(list(x=x), coef(fm1))), col = "red", lwd=2)
As <- coef(fm1)[["Asym"]]; abline(v = 0, h = c(As, As - coef(fm1)[["Drop"]]), lty = 3)
```

start *Encode the Terminal Times of Time Series*

Description

Extract and encode the times the first and last observations were taken. Provided only for compatibility with S version 2.

Usage

```
start(x, ...)
end(x, ...)
```

Arguments

x	a univariate or multivariate time-series, or a vector or matrix.
...	extra arguments for future methods.

Details

These are generic functions, which will use the tsp attribute of x if it exists. Their default methods decode the start time from the original time units, so that for a monthly series 1995.5 is represented as c(1995, 7). For a series of frequency f, time n+i/f is presented as c(n, i+1) (even for i = 0 and f = 1).

Warning

The representation used by start and end has no meaning unless the frequency is supplied.

See Also

ts, time, tsp.

stat.anova *GLM ANOVA Statistics*

Description

This is a utility function, used in lm and glm methods for anova(..., test != NULL) and should not be used by the average user.

Usage

```
stat.anova(table, test = c("Rao","LRT", "Chisq", "F", "Cp"),
           scale, df.scale, n)
```

Arguments

table	numeric matrix as results from anova.glm(..., test = NULL).
test	a character string, partially matching one of "Rao", "LRT", "Chisq", "F" or "Cp".
scale	a residual mean square or other scale estimate to be used as the denominator in an F test.
df.scale	degrees of freedom corresponding to scale.
n	number of observations.

Value

A matrix which is the original table, augmented by a column of test statistics, depending on the test argument.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

anova.lm, anova.glm.

Examples

```
##-- Continued from '?glm':

print(ag <- anova(glm.D93))
stat.anova(ag$table, test = "Cp",
           scale = sum(resid(glm.D93, "pearson")^2)/4,
           df.scale = 4, n = 9)
```

---

stats-deprecated *Deprecated Functions in Package* **stats**

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Details

There are currently no deprecated functions in this package.

See Also

Deprecated

`step` *Choose a model by AIC in a Stepwise Algorithm*

Description

Select a formula-based model by AIC.

Usage

```
step(object, scope, scale = 0,
     direction = c("both", "backward", "forward"),
     trace = 1, keep = NULL, steps = 1000, k = 2, ...)
```

Arguments

`object`	an object representing a model of an appropriate class (mainly `"lm"` and `"glm"`). This is used as the initial model in the stepwise search.
`scope`	defines the range of models examined in the stepwise search. This should be either a single formula, or a list containing components `upper` and `lower`, both formulae. See the details for how to specify the formulae and how they are used.
`scale`	used in the definition of the AIC statistic for selecting the models, currently only for `lm`, `aov` and `glm` models. The default value, `0`, indicates the scale should be estimated: see `extractAIC`.
`direction`	the mode of stepwise search, can be one of `"both"`, `"backward"`, or `"forward"`, with a default of `"both"`. If the `scope` argument is missing the default for `direction` is `"backward"`. Values can be abbreviated.
`trace`	if positive, information is printed during the running of `step`. Larger values may give more detailed information.
`keep`	a filter function whose input is a fitted model object and the associated `AIC` statistic, and whose output is arbitrary. Typically `keep` will select a subset of the components of the object and return them. The default is not to keep anything.
`steps`	the maximum number of steps to be considered. The default is 1000 (essentially as many as required). It is typically used to stop the process early.
`k`	the multiple of the number of degrees of freedom used for the penalty. Only `k = 2` gives the genuine AIC: `k = log(n)` is sometimes referred to as BIC or SBC.
`...`	any additional arguments to `extractAIC`.

Details

`step` uses `add1` and `drop1` repeatedly; it will work for any method for which they work, and that is determined by having a valid method for `extractAIC`. When the additive constant can be chosen so that AIC is equal to Mallows' C_p, this is done and the tables are labelled appropriately.

The set of models searched is determined by the `scope` argument. The right-hand-side of its `lower` component is always included in the model, and right-hand-side of the model is included in the

upper component. If scope is a single formula, it specifies the upper component, and the lower model is empty. If scope is missing, the initial model is used as the upper model.

Models specified by scope can be templates to update object as used by update.formula. So using . in a scope formula means 'what is already there', with .^2 indicating all interactions of existing terms.

There is a potential problem in using glm fits with a variable scale, as in that case the deviance is not simply related to the maximized log-likelihood. The "glm" method for function extractAIC makes the appropriate adjustment for a gaussian family, but may need to be amended for other cases. (The binomial and poisson families have fixed scale by default and do not correspond to a particular maximum-likelihood problem for variable scale.)

Value

the stepwise-selected model is returned, with up to two additional components. There is an "anova" component corresponding to the steps taken in the search, as well as a "keep" component if the keep= argument was supplied in the call. The "Resid. Dev" column of the analysis of deviance table refers to a constant minus twice the maximized log likelihood: it will be a deviance only in cases where a saturated model is well-defined (thus excluding lm, aov and survreg fits, for example).

Warning

The model fitting must apply the models to the same dataset. This may be a problem if there are missing values and R's default of na.action = na.omit is used. We suggest you remove the missing values first.

Calls to the function nobs are used to check that the number of observations involved in the fitting process remains unchanged.

Note

This function differs considerably from the function in S, which uses a number of approximations and does not in general compute the correct AIC.

This is a minimal implementation. Use stepAIC in package **MASS** for a wider range of object classes.

Author(s)

B. D. Ripley: step is a slightly simplified version of stepAIC in package **MASS** (Venables & Ripley, 2002 and earlier editions).

The idea of a step function follows that described in Hastie & Pregibon (1992); but the implementation in R is more general.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

stepAIC in **MASS**, add1, drop1

Examples

```
## following on from example(lm)

step(lm.D9)

summary(lm1 <- lm(Fertility ~ ., data = swiss))
slm1 <- step(lm1)
summary(slm1)
slm1$anova
```

---

stepfun *Step Functions - Creation and Class*

Description

Given the vectors $(x_1, \ldots, x_n)$ and $(y_0, y_1, \ldots, y_n)$ (one value more!), `stepfun(x, y, ...)` returns an interpolating 'step' function, say `fn`. I.e., $fn(t) = c_i$ (constant) for $t \in (x_i, x_{i+1})$ and at the abscissa values, if (by default) `right = FALSE`, $fn(x_i) = y_i$ and for `right = TRUE`, $fn(x_i) = y_{i-1}$, for $i = 1, \ldots, n$.

The value of the constant c_i above depends on the 'continuity' parameter `f`. For the default, `right = FALSE, f = 0`, `fn` is a *cadlag* function, i.e., continuous from the right, limits from the left, so that the function is piecewise constant on intervals that include their *left* endpoint. In general, c_i is interpolated in between the neighbouring y values, $c_i = (1 - f)y_i + f \cdot y_{i+1}$. Therefore, for non-0 values of `f`, `fn` may no longer be a proper step function, since it can be discontinuous from both sides, unless `right = TRUE, f = 1` which is left-continuous (i.e., constant pieces contain their right endpoint).

Usage

```
stepfun(x, y, f = as.numeric(right), ties = "ordered",
        right = FALSE)

is.stepfun(x)
knots(Fn, ...)
as.stepfun(x, ...)

## S3 method for class 'stepfun'
print(x, digits = getOption("digits") - 2, ...)

## S3 method for class 'stepfun'
summary(object, ...)
```

Arguments

x	numeric vector giving the knots or jump locations of the step function for stepfun(). For the other functions, x is as object below.
y	numeric vector one longer than x, giving the heights of the function values *between* the x values.
f	a number between 0 and 1, indicating how interpolation outside the given x values should happen. See approxfun.
ties	Handling of tied x values. Either a function or the string "ordered". See approxfun.
right	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
Fn, object	an R object inheriting from "stepfun".
digits	number of significant digits to use, see print.
...	potentially further arguments (required by the generic).

Value

A function of class "stepfun", say fn.

There are methods available for summarizing ("summary(.)"), representing ("print(.)") and plotting ("plot(.)", see plot.stepfun) "stepfun" objects.

The environment of fn contains all the information needed;

"x", "y"	the original arguments
"n"	number of knots (x values)
"f"	continuity parameter
"yleft", "yright"	the function values *outside* the knots
"method"	(always == "constant", from approxfun(.)).

The knots are also available via knots(fn).

Note

The objects of class "stepfun" are not intended to be used for permanent storage and may change structure between versions of R (and did at R 3.0.0). They can usually be re-created by

```
eval(attr(old_obj, "call"), environment(old_obj))
```

since the data used is stored as part of the object's environment.

Author(s)

Martin Maechler, <maechler@stat.math.ethz.ch> with some basic code from Thomas Lumley.

See Also

ecdf for empirical distribution functions as special step functions and plot.stepfun for *plotting* step functions.

approxfun and splinefun.

Examples

```
y0 <- c(1., 2., 4., 3.)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = 0.2)
sfun1  <- stepfun(1:3, y0, f = 1)
sfun1c <- stepfun(1:3, y0, right = TRUE) # hence f=1
sfun0
summary(sfun0)
summary(sfun.2)

## look at the internal structure:
unclass(sfun0)
ls(envir = environment(sfun0))

x0 <- seq(0.5, 3.5, by = 0.25)
rbind(x = x0, f.f0 = sfun0(x0), f.f02 = sfun.2(x0),
      f.f1 = sfun1(x0), f.f1c = sfun1c(x0))
## Identities :
stopifnot(identical(y0[-1], sfun0 (1:3)), # right = FALSE
          identical(y0[-4], sfun1c(1:3))) # right = TRUE
```

---

stl *Seasonal Decomposition of Time Series by Loess*

Description

Decompose a time series into seasonal, trend and irregular components using loess, acronym STL.

Usage

```
stl(x, s.window, s.degree = 0,
    t.window = NULL, t.degree = 1,
    l.window = nextodd(period), l.degree = t.degree,
    s.jump = ceiling(s.window/10),
    t.jump = ceiling(t.window/10),
    l.jump = ceiling(l.window/10),
    robust = FALSE,
    inner = if(robust)  1 else 2,
    outer = if(robust) 15 else 0,
    na.action = na.fail)
```

Arguments

x	univariate time series to be decomposed. This should be an object of class "ts" with a frequency greater than one.
s.window	either the character string "periodic" or the span (in lags) of the loess window for seasonal extraction, which should be odd and at least 7, according to Cleveland et al. This has no default.
s.degree	degree of locally-fitted polynomial in seasonal extraction. Should be zero or one.
t.window	the span (in lags) of the loess window for trend extraction, which should be odd. If NULL, the default, nextodd(ceiling((1.5*period) / (1-(1.5/s.window)))), is taken.
t.degree	degree of locally-fitted polynomial in trend extraction. Should be zero or one.
l.window	the span (in lags) of the loess window of the low-pass filter used for each subseries. Defaults to the smallest odd integer greater than or equal to frequency(x) which is recommended since it prevents competition between the trend and seasonal components. If not an odd integer its given value is increased to the next odd one.
l.degree	degree of locally-fitted polynomial for the subseries low-pass filter. Must be 0 or 1.
s.jump, t.jump, l.jump	integers at least one to increase speed of the respective smoother. Linear interpolation happens between every *.jump-th value.
robust	logical indicating if robust fitting be used in the loess procedure.
inner	integer; the number of 'inner' (backfitting) iterations; usually very few (2) iterations suffice.
outer	integer; the number of 'outer' robustness iterations.
na.action	action on missing values.

Details

The seasonal component is found by *loess* smoothing the seasonal sub-series (the series of all January values, ...); if s.window = "periodic" smoothing is effectively replaced by taking the mean. The seasonal values are removed, and the remainder smoothed to find the trend. The overall level is removed from the seasonal component and added to the trend component. This process is iterated a few times. The remainder component is the residuals from the seasonal plus trend fit.

Several methods for the resulting class "stl" objects, see, plot.stl.

Value

stl returns an object of class "stl" with components

time.series	a multiple time series with columns seasonal, trend and remainder.
weights	the final robust weights (all one if fitting is not done robustly).
call	the matched call.

win	integer (length 3 vector) with the spans used for the "s", "t", and "l" smoothers.
deg	integer (length 3) vector with the polynomial degrees for these smoothers.
jump	integer (length 3) vector with the 'jumps' (skips) used for these smoothers.
ni	number of **i**nner iterations
no	number of **o**uter robustness iterations

Author(s)

B.D. Ripley; Fortran code by Cleveland et al. (1990) from 'netlib'.

References

R. B. Cleveland, W. S. Cleveland, J.E. McRae, and I. Terpenning (1990) STL: A Seasonal-Trend Decomposition Procedure Based on Loess. *Journal of Official Statistics*, **6**, 3–73.

See Also

plot.stl for stl methods; loess in package **stats** (which is not actually used in stl).

StructTS for different kind of decomposition.

Examples

```
require(graphics)

plot(stl(nottem, "per"))
plot(stl(nottem, s.window = 7, t.window = 50, t.jump = 1))

plot(stllc <- stl(log(co2), s.window = 21))
summary(stllc)
## linear trend, strict period.
plot(stl(log(co2), s.window = "per", t.window = 1000))

## Two STL plotted side by side :
        stmd <- stl(mdeaths, s.window = "per") # non-robust
summary(stmR <- stl(mdeaths, s.window = "per", robust = TRUE))
op <- par(mar = c(0, 4, 0, 3), oma = c(5, 0, 4, 0), mfcol = c(4, 2))
plot(stmd, set.pars = NULL, labels  =  NULL,
     main = "stl(mdeaths, s.w = \"per\",  robust = FALSE / TRUE )")
plot(stmR, set.pars = NULL)
# mark the 'outliers' :
(iO <- which(stmR $ weights  < 1e-8)) # 10 were considered outliers
sts <- stmR$time.series
points(time(sts)[iO], 0.8* sts[,"remainder"][iO], pch = 4, col = "red")
par(op)   # reset
```

stlmethods *Methods for STL Objects*

Description

Methods for objects of class `stl`, typically the result of `stl`. The `plot` method does a multiple figure plot with some flexibility.

There are also (non-visible) `print` and `summary` methods.

Usage

```
## S3 method for class 'stl'
plot(x, labels = colnames(X),
     set.pars = list(mar = c(0, 6, 0, 6), oma = c(6, 0, 4, 0),
                     tck = -0.01, mfrow = c(nplot, 1)),
     main = NULL, range.bars = TRUE, ...,
     col.range = "light gray")
```

Arguments

`x`	`stl` object.
`labels`	character of length 4 giving the names of the component time-series.
`set.pars`	settings for `par(.)` when setting up the plot.
`main`	plot main title.
`range.bars`	logical indicating if each plot should have a bar at its right side which are of equal heights in user coordinates.
`...`	further arguments passed to or from other methods.
`col.range`	colour to be used for the range bars, if plotted. Note this appears after `...` and so cannot be abbreviated.

See Also

`plot.ts` and `stl`, particularly for examples.

StructTS *Fit Structural Time Series*

Description

Fit a structural model for a time series by maximum likelihood.

Usage

```
StructTS(x, type = c("level", "trend", "BSM"), init = NULL,
         fixed = NULL, optim.control = NULL)
```

Arguments

x	a univariate numeric time series. Missing values are allowed.
type	the class of structural model. If omitted, a BSM is used for a time series with frequency(x) > 1, and a local trend model otherwise. Can be abbreviated.
init	initial values of the variance parameters.
fixed	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. Probably most useful for setting variances to zero.
optim.control	List of control parameters for optim. Method "L-BFGS-B" is used.

Details

Structural time series models are (linear Gaussian) state-space models for (univariate) time series based on a decomposition of the series into a number of components. They are specified by a set of error variances, some of which may be zero.

The simplest model is the *local level* model specified by type = "level". This has an underlying level μ_t which evolves by

$$\mu_{t+1} = \mu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

The observations are

$$x_t = \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

There are two parameters, σ_ξ^2 and σ_ϵ^2. It is an ARIMA(0,1,1) model, but with restrictions on the parameter set.

The *local linear trend model*, type = "trend", has the same measurement equation, but with a time-varying slope in the dynamics for μ_t, given by

$$\mu_{t+1} = \mu_t + \nu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

$$\nu_{t+1} = \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2)$$

with three variance parameters. It is not uncommon to find $\sigma_\zeta^2 = 0$ (which reduces to the local level model) or $\sigma_\xi^2 = 0$, which ensures a smooth trend. This is a restricted ARIMA(0,2,2) model.

The *basic structural model*, type = "BSM", is a local trend model with an additional seasonal component. Thus the measurement equation is

$$x_t = \mu_t + \gamma_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

where γ_t is a seasonal component with dynamics

$$\gamma_{t+1} = -\gamma_t + \cdots + \gamma_{t-s+2} + \omega_t, \qquad \omega_t \sim N(0, \sigma_\omega^2)$$

The boundary case $\sigma_\omega^2 = 0$ corresponds to a deterministic (but arbitrary) seasonal pattern. (This is sometimes known as the 'dummy variable' version of the BSM.)

Value

A list of class "StructTS" with components:

coef	the estimated variances of the components.
loglik	the maximized log-likelihood. Note that as all these models are non-stationary this includes a diffuse prior for some observations and hence is not comparable to arima nor different types of structural models.
loglik0	the maximized log-likelihood with the constant used prior to R 3.0.0, for backwards compatibility.
data	the time series x.
residuals	the standardized residuals.
fitted	a multiple time series with one component for the level, slope and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
call	the matched call.
series	the name of the series x.
code	the convergence code returned by optim.
model, model0	Lists representing the Kalman filter used in the fitting. See KalmanLike. model0 is the initial state of the filter, model its final state.
xtsp	the tsp attributes of x.

Note

Optimization of structural models is a lot harder than many of the references admit. For example, the AirPassengers data are considered in Brockwell & Davis (1996): their solution appears to be a local maximum, but nowhere near as good a fit as that produced by StructTS. It is quite common to find fits with one or more variances zero, and this can include σ_ϵ^2.

References

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 8.2 and 8.5.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Harvey, A. C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf.

See Also

KalmanLike, tsSmooth; stl for different kind of (seasonal) decomposition.

Examples

```
## see also JohnsonJohnson, Nile and AirPassengers
require(graphics)

trees <- window(treering, start = 0)
(fit <- StructTS(trees, type = "level"))
plot(trees)
lines(fitted(fit), col = "green")
tsdiag(fit)

(fit <- StructTS(log10(UKgas), type = "BSM"))
par(mfrow = c(4, 1)) # to give appropriate aspect ratio for next plot.
plot(log10(UKgas))
plot(cbind(fitted(fit), resids=resid(fit)), main = "UK gas consumption")

## keep some parameters fixed; trace optimizer:
StructTS(log10(UKgas), type = "BSM", fixed = c(0.1,0.001,NA,NA),
         optim.control = list(trace = TRUE))
```

---

summary.aov *Summarize an Analysis of Variance Model*

Description

Summarize an analysis of variance model.

Usage

```
## S3 method for class 'aov'
summary(object, intercept = FALSE, split,
        expand.split = TRUE, keep.zero.df = TRUE, ...)

## S3 method for class 'aovlist'
summary(object, ...)
```

Arguments

object	An object of class "aov" or "aovlist".
intercept	logical: should intercept terms be included?
split	an optional named list, with names corresponding to terms in the model. Each component is itself a list with integer components giving contrasts whose contributions are to be summed.
expand.split	logical: should the split apply also to interactions involving the factor?
keep.zero.df	logical: should terms with no degrees of freedom be included?
...	Arguments to be passed to or from other methods, for summary.aovlist including those for summary.aov.

Value

An object of class `c("summary.aov", "listof")` or `"summary.aovlist"` respectively.

For fits with a single stratum the result will be a list of ANOVA tables, one for each response (even if there is only one response): the tables are of class `"anova"` inheriting from class `"data.frame"`. They have columns `"Df"`, `"Sum Sq"`, `"Mean Sq"`, as well as `"F value"` and `"Pr(>F)"` if there are non-zero residual degrees of freedom. There is a row for each term in the model, plus one for `"Residuals"` if there are any.

For multistratum fits the return value is a list of such summaries, one for each stratum.

Note

The use of `expand.split = TRUE` is little tested: it is always possible to set it to `FALSE` and specify exactly all the splits required.

See Also

`aov`, `summary`, `model.tables`, `TukeyHSD`

Examples

```
## For a simple example see example(aov)

# Cochran and Cox (1957, p.164)
# 3x3 factorial with ordered factors, each is average of 12.
CC <- data.frame(
    y = c(449, 413, 326, 409, 358, 291, 341, 278, 312)/12,
    P = ordered(gl(3, 3)), N = ordered(gl(3, 1, 9))
)
CC.aov <- aov(y ~ N * P, data = CC , weights = rep(12, 9))
summary(CC.aov)

# Split both main effects into linear and quadratic parts.
summary(CC.aov, split = list(N = list(L = 1, Q = 2),
                             P = list(L = 1, Q = 2)))

# Split only the interaction
summary(CC.aov, split = list("N:P" = list(L.L = 1, Q = 2:4)))

# split on just one var
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)))
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)),
        expand.split = FALSE)
```

`summary.glm` *Summarizing Generalized Linear Model Fits*

Description

These functions are all `methods` for class `glm` or `summary.glm` objects.

Usage

```
## S3 method for class 'glm'
summary(object, dispersion = NULL, correlation = FALSE,
        symbolic.cor = FALSE, ...)

## S3 method for class 'summary.glm'
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"),
      show.residuals = FALSE, ...)
```

Arguments

object	an object of class "glm", usually, a result of a call to glm.
x	an object of class "summary.glm", usually, a result of a call to summary.glm.
dispersion	the dispersion parameter for the family used. Either a single numerical value or NULL (the default), when it is inferred from object (see 'Details').
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, 'significance stars' are printed for each coefficient.
show.residuals	logical. If TRUE then a summary of the deviance residuals is printed at the head of the output.
...	further arguments passed to or from other methods.

Details

print.summary.glm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE. The coefficients component of the result gives the estimated coefficients and their estimated standard errors, together with their ratio. This third column is labelled t ratio if the dispersion is estimated, and z ratio if the dispersion is known (or fixed by the family). A fourth column gives the two-tailed p-value corresponding to the t or z ratio based on a Student t or Normal reference distribution. (It is possible that the dispersion is not known and there are no residual degrees of freedom from which to estimate it. In that case the estimate is NaN.)

Aliased coefficients are omitted in the returned object but restored by the print method.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

The dispersion of a GLM is not used in the fitting process, but it is needed to find standard errors. If dispersion is not supplied or NULL, the dispersion is taken as 1 for the binomial and Poisson families, and otherwise estimated by the residual Chi-squared statistic (calculated from cases with non-zero weights) divided by the residual degrees of freedom.

summary can be used with Gaussian glm fits to handle the case of a linear regression with known error variance, something not handled by summary.lm.

Value

summary.glm returns an object of class "summary.glm", a list with components

call	the component from object.
family	the component from object.
deviance	the component from object.
contrasts	the component from object.
df.residual	the component from object.
null.deviance	the component from object.
df.null	the component from object.
deviance.resid	the deviance residuals: see residuals.glm.
coefficients	the matrix of coefficients, standard errors, z-values and p-values. Aliased coefficients are omitted.
aliased	named logical vector showing if the original coefficients are aliased.
dispersion	either the supplied argument or the inferred/estimated dispersion if the former is NULL.
df	a 3-vector of the rank of the model and the number of residual degrees of freedom, plus number of coefficients (including aliased ones).
cov.unscaled	the unscaled (dispersion = 1) estimated covariance matrix of the estimated coefficients.
cov.scaled	ditto, scaled by dispersion.
correlation	(only if correlation is true.) The estimated correlations of the estimated coefficients.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.

See Also

glm, summary.

Examples

```
## For examples see example(glm)
```

---

summary.lm *Summarizing Linear Model Fits*

Description

summary method for class "lm".

Usage

```
## S3 method for class 'lm'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.lm'
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "lm", usually, a result of a call to lm.
x	an object of class "summary.lm", usually, a result of a call to summary.lm.
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, 'significance stars' are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

print.summary.lm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE.

Aliased coefficients are omitted in the returned object but restored by the print method.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

Value

The function summary.lm computes and returns a list of summary statistics of the fitted linear model given in object, using the components (list elements) "call" and "terms" from its argument, plus

residuals	the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to lm.
coefficients	a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value. Aliased coefficients are omitted.
aliased	named logical vector showing if the original coefficients are aliased.
sigma	the square root of the estimated variance of the random error

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_i w_i R_i^2,$$

where R_i is the i-th residual, residuals[i].

df	degrees of freedom, a 3-vector $(p, n-p, p*)$, the first being the number of non-aliased coefficients, the last being the total number of coefficients.
fstatistic	(for models including non-intercept terms) a 3-vector with the value of the F-statistic with its numerator and denominator degrees of freedom.
r.squared	R^2, the 'fraction of variance explained by the model', $$R^2 = 1 - \frac{\sum_i R_i^2}{\sum_i (y_i - y^*)^2},$$ where y^* is the mean of y_i if there is an intercept and zero otherwise.
adj.r.squared	the above R^2 statistic '*adjusted*', penalizing for higher p.
cov.unscaled	a $p \times p$ matrix of (unscaled) covariances of the $\hat{\beta}_j$, $j = 1, \ldots, p$.
correlation	the correlation matrix corresponding to the above cov.unscaled, if correlation = TRUE is specified.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.
na.action	from object, if present there.

See Also

The model fitting function lm, summary.

Function coef will extract the matrix of coefficients with standard errors, t-statistics and p-values.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90)  # the bare coefficients
sld90 <- summary(lm.D90 <- lm(weight ~ group -1))  # omitting intercept
sld90
coef(sld90)  # much more

## model with *aliased* coefficient:
lm.D9. <- lm(weight ~ group + I(group != "Ctl"))
Sm.D9. <- summary(lm.D9.)
Sm.D9. #  shows the NA NA NA NA  line
stopifnot(length(cc <- coef(lm.D9.)) == 3, is.na(cc[3]),
          dim(coef(Sm.D9.)) == c(2,4), Sm.D9.$df == c(2, 18, 3))
```

---

summary.manova *Summary Method for Multivariate Analysis of Variance*

Description

A summary method for class "manova".

Usage

```
## S3 method for class 'manova'
summary(object,
        test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy"),
        intercept = FALSE, tol = 1e-7, ...)
```

Arguments

object	An object of class "manova" or an aov object with multiple responses.
test	The name of the test statistic to be used. Partial matching is used so the name can be abbreviated.
intercept	logical. If TRUE, the intercept term is included in the table.
tol	tolerance to be used in deciding if the residuals are rank-deficient: see qr.
...	further arguments passed to or from other methods.

Details

The summary.manova method uses a multivariate test statistic for the summary table. Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987).

The table gives a transformation of the test statistic which has approximately an F distribution. The approximations used follow S-PLUS and SAS (the latter apart from some cases of the Hotelling–Lawley statistic), but many other distributional approximations exist: see Anderson (1984) and Krzanowski and Marriott (1994) for further references. All four approximate F statistics are the same when the term being tested has one degree of freedom, but in other cases that for the Roy statistic is an upper bound.

The tolerance tol is applied to the QR decomposition of the residual correlation matrix (unless some response has essentially zero residuals, when it is unscaled). Thus the default value guards against very highly correlated responses: it can be reduced but doing so will allow rather inaccurate results and it will normally be better to transform the responses to remove the high correlation.

Value

An object of class "summary.manova". If there is a positive residual degrees of freedom, this is a list with components

row.names	The names of the terms, the row names of the stats table if present.
SS	A named list of sums of squares and product matrices.
Eigenvalues	A matrix of eigenvalues.
stats	A matrix of the statistics, approximate F value, degrees of freedom and P value.

otherwise components row.names, SS and Df (degrees of freedom) for the terms (and not the residuals).

References

Anderson, T. W. (1994) *An Introduction to Multivariate Statistical Analysis.* Wiley.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Krzanowski, W. J. and Marriott, F. H. C. (1994) *Multivariate Analysis. Part I: Distributions, Ordination and Inference.* Edward Arnold.

See Also

manova, aov

Examples

```
## Example on producing plastic film from Krzanowski (1998, p. 381)
tear <- c(6.5, 6.2, 5.8, 6.5, 6.5, 6.9, 7.2, 6.9, 6.1, 6.3,
          6.7, 6.6, 7.2, 7.1, 6.8, 7.1, 7.0, 7.2, 7.5, 7.6)
gloss <- c(9.5, 9.9, 9.6, 9.6, 9.2, 9.1, 10.0, 9.9, 9.5, 9.4,
           9.1, 9.3, 8.3, 8.4, 8.5, 9.2, 8.8, 9.7, 10.1, 9.2)
opacity <- c(4.4, 6.4, 3.0, 4.1, 0.8, 5.7, 2.0, 3.9, 1.9, 5.7,
             2.8, 4.1, 3.8, 1.6, 3.4, 8.4, 5.2, 6.9, 2.7, 1.9)
Y <- cbind(tear, gloss, opacity)
rate     <- gl(2,10, labels = c("Low", "High"))
additive <- gl(2, 5, length = 20, labels = c("Low", "High"))

fit <- manova(Y ~ rate * additive)
summary.aov(fit)             # univariate ANOVA tables
summary(fit, test = "Wilks") # ANOVA table of Wilks' lambda
summary(fit)                 # same F statistics as single-df terms
```

---

summary.nls *Summarizing Non-Linear Least-Squares Model Fits*

Description

summary method for class "nls".

Usage

```
## S3 method for class 'nls'
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.nls'
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "nls".
x	an object of class "summary.nls", usually the result of a call to summary.nls.
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, 'significance stars' are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

The distribution theory used to find the distribution of the standard errors and of the residual standard error (for t ratios) is based on linearization and is approximate, maybe very approximate.

print.summary.nls tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

Value

The function summary.nls computes and returns a list of summary statistics of the fitted model given in object, using the component "formula" from its argument, plus

residuals	the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to nls.
coefficients	a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value.
sigma	the square root of the estimated variance of the random error $$\hat{\sigma}^2 = \frac{1}{n-p}\sum_i R_i^2,$$ where R_i is the i-th weighted residual.
df	degrees of freedom, a 2-vector $(p, n-p)$. (Here and elsewhere n omits observations with zero weights.)
cov.unscaled	a $p \times p$ matrix of (unscaled) covariances of the parameter estimates.
correlation	the correlation matrix corresponding to the above cov.unscaled, if correlation = TRUE is specified and there are a non-zero number of residual degrees of freedom.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.

See Also

The model fitting function nls, summary.

Function coef will extract the matrix of coefficients with standard errors, t-statistics and p-values.

summary.princomp *Summary method for Principal Components Analysis*

Description

The summary method for class "princomp".

Usage

```
## S3 method for class 'princomp'
summary(object, loadings = FALSE, cutoff = 0.1, ...)

## S3 method for class 'summary.princomp'
print(x, digits = 3, loadings = x$print.loadings,
      cutoff = x$cutoff, ...)
```

Arguments

object	an object of class "princomp", as from princomp().
loadings	logical. Should loadings be included?
cutoff	numeric. Loadings below this cutoff in absolute value are shown as blank in the output.
x	an object of class "summary.princomp".
digits	the number of significant digits to be used in listing loadings.
...	arguments to be passed to or from other methods.

Value

object with additional components cutoff and print.loadings.

See Also

princomp

Examples

```
summary(pc.cr <- princomp(USArrests, cor = TRUE))
## The signs of the loading columns are arbitrary
print(summary(princomp(USArrests, cor = TRUE),
              loadings = TRUE, cutoff = 0.2), digits = 2)
```

`supsmu` *Friedman's SuperSmoother*

Description

Smooth the (x, y) values by Friedman's 'super smoother'.

Usage

```
supsmu(x, y, wt =, span = "cv", periodic = FALSE, bass = 0, trace = FALSE)
```

Arguments

`x`	x values for smoothing
`y`	y values for smoothing
`wt`	case weights, by default all equal
`span`	the fraction of the observations in the span of the running lines smoother, or `"cv"` to choose this by leave-one-out cross-validation.
`periodic`	if `TRUE`, the x values are assumed to be in `[0, 1]` and of period 1.
`bass`	controls the smoothness of the fitted curve. Values of up to 10 indicate increasing smoothness.
`trace`	logical, if true, prints one line of info "per spar", notably useful for `"cv"`.

Details

`supsmu` is a running lines smoother which chooses between three spans for the lines. The running lines smoothers are symmetric, with `k/2` data points each side of the predicted point, and values of `k` as `0.5 * n`, `0.2 * n` and `0.05 * n`, where `n` is the number of data points. If `span` is specified, a single smoother with span `span * n` is used.

The best of the three smoothers is chosen by cross-validation for each prediction. The best spans are then smoothed by a running lines smoother and the final prediction chosen by linear interpolation.

The FORTRAN code says: "For small samples (`n < 40`) or if there are substantial serial correlations between observations close in x-value, then a pre-specified fixed span smoother (`span > 0`) should be used. Reasonable span values are 0.2 to 0.4."

Cases with non-finite values of `x`, `y` or `wt` are dropped, with a warning.

Value

A list with components

`x`	the input values in increasing order with duplicates removed.
`y`	the corresponding y values on the fitted curve.

References

Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Friedman, J. H. (1984) A variable span scatterplot smoother. Laboratory for Computational Statistics, Stanford University Technical Report No. 5.

See Also

ppr

Examples

```
require(graphics)

with(cars, {
    plot(speed, dist)
    lines(supsmu(speed, dist))
    lines(supsmu(speed, dist, bass = 7), lty = 2)
    })
```

---

symnum *Symbolic Number Coding*

Description

Symbolically encode a given numeric or logical vector or array. Particularly useful for visualization of structured matrices, e.g., correlation, sparse, or logical ones.

Usage

```
symnum(x, cutpoints = c(0.3, 0.6, 0.8, 0.9, 0.95),
       symbols = if(numeric.x) c(" ", ".", ",", "+", "*", "B")
                 else c(".", "|"),
       legend = length(symbols) >= 3,
       na = "?", eps = 1e-5, numeric.x = is.numeric(x),
       corr = missing(cutpoints) && numeric.x,
       show.max = if(corr) "1", show.min = NULL,
       abbr.colnames = has.colnames,
       lower.triangular = corr && is.numeric(x) && is.matrix(x),
       diag.lower.tri   = corr && !is.null(show.max))
```

Arguments

x	numeric or logical vector or array.
cutpoints	numeric vector whose values cutpoints[j] $= c_j$ (*after* augmentation, see corr below) are used for intervals.

symbols	character vector, one shorter than (the *augmented*, see corr below) cutpoints. symbols[j]= s_j are used as 'code' for the (half open) interval $(c_j, c_{j+1}]$. When numeric.x is FALSE, i.e., by default when argument x is logical, the default is c(".","|") (graphical 0 / 1 s).
legend	logical indicating if a "legend" attribute is desired.
na	character or logical. How NAs are coded. If na == FALSE, NAs are coded invisibly, *including* the "legend" attribute below, which otherwise mentions NA coding.
eps	absolute precision to be used at left and right boundary.
numeric.x	logical indicating if x should be treated as numbers, otherwise as logical.
corr	logical. If TRUE, x contains correlations. The cutpoints are augmented by 0 and 1 and abs(x) is coded.
show.max	if TRUE, or of mode character, the maximal cutpoint is coded especially.
show.min	if TRUE, or of mode character, the minimal cutpoint is coded especially.
abbr.colnames	logical, integer or NULL indicating how column names should be abbreviated (if they are); if NULL (or FALSE and x has no column names), the column names will all be empty, i.e., ""; otherwise if abbr.colnames is false, they are left unchanged. If TRUE or integer, existing column names will be abbreviated to abbreviate(*, minlength = abbr.colnames).
lower.triangular	logical. If TRUE and x is a matrix, only the *lower triangular* part of the matrix is coded as non-blank.
diag.lower.tri	logical. If lower.triangular *and* this are TRUE, the *diagonal* part of the matrix is shown.

Value

An atomic character object of class noquote and the same dimensions as x.

If legend is TRUE (as by default when there are more than two classes), the result has an attribute "legend" containing a legend of the returned character codes, in the form

$$c_1 s_1 c_2 s_2 \ldots s_n c_{n+1}$$

where c_j = cutpoints[j] and s_j = symbols[j].

Note

The optional (mostly logical) arguments all try to use smart defaults. Specifying them explicitly may lead to considerably improved output in many cases.

Author(s)

Martin Maechler <maechler@stat.math.ethz.ch>

See Also

as.character; image

Examples

```
ii <- setNames(0:8, 0:8)
symnum(ii, cutpoints =  2*(0:4), symbols = c(".", "-", "+", "$"))
symnum(ii, cutpoints =  2*(0:4), symbols = c(".", "-", "+", "$"), show.max = TRUE)

symnum(1:12 %% 3 == 0)  # --> "|" = TRUE, "." = FALSE  for logical

## Pascal's Triangle modulo 2 -- odd and even numbers:
N <- 38
pascal <- t(sapply(0:N, function(n) round(choose(n, 0:N - (N-n)%/%2))))
rownames(pascal) <- rep("", 1+N) # <-- to improve "graphic"
symnum(pascal %% 2, symbols = c(" ", "A"), numeric.x = FALSE)

##-- Symbolic correlation matrices:
symnum(cor(attitude), diag.lower.tri = FALSE)
symnum(cor(attitude), abbr.colnames = NULL)
symnum(cor(attitude), abbr.colnames = FALSE)
symnum(cor(attitude), abbr.colnames = 2)

symnum(cor(rbind(1, rnorm(25), rnorm(25)^2)))
symnum(cor(matrix(rexp(30, 1), 5, 18))) # <<-- PATTERN ! --
symnum(cm1 <- cor(matrix(rnorm(90) ,  5, 18))) # < White Noise SMALL n
symnum(cm1, diag.lower.tri = FALSE)
symnum(cm2 <- cor(matrix(rnorm(900), 50, 18))) # < White Noise "BIG" n
symnum(cm2, lower.triangular = FALSE)

## NA's:
Cm <- cor(matrix(rnorm(60),  10, 6)); Cm[c(3,6), 2] <- NA
symnum(Cm, show.max = NULL)

## Graphical P-values (aka "significance stars"):
pval <- rev(sort(c(outer(1:6, 10^-(1:3)))))
symp <- symnum(pval, corr = FALSE,
               cutpoints = c(0,  .001,.01,.05, .1, 1),
               symbols = c("***","**","*",".","  "))
noquote(cbind(P.val = format(pval), Signif = symp))
```

`t.test` *Student's t-Test*

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)

## Default S3 method:
```

```
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula'
t.test(formula, data, subset, na.action = na.pass, ...)
```

Arguments

x	a (non-empty) numeric vector of data values.
y	an optional (non-empty) numeric vector of data values.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
paired	a logical indicating whether you want a paired t-test.
var.equal	a logical variable indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs either 1 for a one-sample or paired test or a factor with two levels giving the corresponding groups. If lhs is of class "Pair" and rhs is 1, a paired test is done, see Examples.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs.
...	further arguments to be passed to or from methods. For the formula method, this includes arguments of the default method, but not paired.

Details

alternative = "greater" is the alternative that x has a larger mean than y. For the one-sample case: that the mean is positive.

If paired is TRUE then both x and y must be specified and they must be the same length. Missing values are silently removed (in pairs if paired is TRUE). If var.equal is TRUE then the pooled estimate of the variance is used. By default, if var.equal is FALSE then the variance is estimated separately for both groups and the Welch modification to the degrees of freedom is used.

If the input data are effectively constant (compared to the larger of the two means) an error is generated.

Value

A list with class "htest" containing the following components:

statistic	the value of the t-statistic.
parameter	the degrees of freedom for the t-statistic.
p.value	the p-value for the test.
conf.int	a confidence interval for the mean appropriate to the specified alternative hypothesis.
estimate	the estimated mean or difference in means depending on whether it was a one-sample test or a two-sample test.
null.value	the specified hypothesized value of the mean or mean difference depending on whether it was a one-sample test or a two-sample test.
stderr	the standard error of the mean (difference), used as denominator in the t-statistic formula.
alternative	a character string describing the alternative hypothesis.
method	a character string indicating what type of t-test was performed.
data.name	a character string giving the name(s) of the data.

See Also

prop.test

Examples

```
## Two-sample t-test
t.test(1:10, y = c(7:20))      # P = .00001855
t.test(1:10, y = c(7:20, 200)) # P = .1245    -- NOT significant anymore

## Traditional interface
with(mtcars, t.test(mpg[am == 0], mpg[am == 1]))

## Formula interface
t.test(mpg ~ am, data = mtcars)

## One-sample t-test
## Traditional interface
t.test(sleep$extra)

## Formula interface
t.test(extra ~ 1, data = sleep)

## Paired t-test
## The sleep data is actually paired, so could have been in wide format:
sleep2 <- reshape(sleep, direction = "wide",
                  idvar = "ID", timevar = "group")

## Traditional interface
t.test(sleep2$extra.1, sleep2$extra.2, paired = TRUE)
```

```
## Formula interface
t.test(Pair(extra.1, extra.2) ~ 1, data = sleep2)
```

TDist *The Student t Distribution*

Description

Density, distribution function, quantile function and random generation for the t distribution with df degrees of freedom (and optional non-centrality parameter ncp).

Usage

```
dt(x, df, ncp, log = FALSE)
pt(q, df, ncp, lower.tail = TRUE, log.p = FALSE)
qt(p, df, ncp, lower.tail = TRUE, log.p = FALSE)
rt(n, df, ncp)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
df	degrees of freedom (> 0, maybe non-integer). df = Inf is allowed.
ncp	non-centrality parameter δ; currently except for rt(), accurate only for abs(ncp) <= 37.62. If omitted, use the central t distribution.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The t distribution with df $= \nu$ degrees of freedom has density

$$f(x) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}(1 + x^2/\nu)^{-(\nu+1)/2}$$

for all real x. It has mean 0 (for $\nu > 1$) and variance $\frac{\nu}{\nu-2}$ (for $\nu > 2$).

The general *non-central* t with parameters (ν, δ) = (df, ncp) is defined as the distribution of $T_\nu(\delta) := (U + \delta)/\sqrt{V/\nu}$ where U and V are independent random variables, $U \sim \mathcal{N}(0, 1)$ and $V \sim \chi^2_\nu$ (see Chisquare).

The most used applications are power calculations for t-tests:
Let $T = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$ where $\bar{X}$ is the mean and S the sample standard deviation (sd) of $X_1, X_2, \ldots, X_n$ which are i.i.d. $\mathcal{N}(\mu, \sigma^2)$ Then T is distributed as non-central t with df $= n-1$ degrees of freedom and **n**on-**c**entrality **p**arameter ncp $= (\mu - \mu_0)\sqrt{n}/\sigma$.

The t distribution's cumulative distribution function (cdf), F_ν fulfills $F_\nu(t) = \frac{1}{2} I_x(\frac{\nu}{2}, \frac{1}{2})$, for $t \leq 0$, and $F_\nu(t) = 1 - \frac{1}{2} I_x(\frac{\nu}{2}, \frac{1}{2})$, for $t \geq 0$, where $x := \nu/(\nu + t^2)$, and $I_x(a, b)$ is the incomplete beta function, in R this is `pbeta(x, a,b)`.

Value

`dt` gives the density, `pt` gives the distribution function, `qt` gives the quantile function, and `rt` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rt`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

Supplying `ncp = 0` uses the algorithm for the non-central distribution, which is not the same algorithm used if `ncp` is omitted. This is to give consistent behaviour in extreme cases with values of `ncp` very near zero.

The code for non-zero `ncp` is principally intended to be used for moderate values of `ncp`: it will not be highly accurate, especially in the tails, for large values.

Source

The central `dt` is computed via an accurate formula provided by Catherine Loader (see the reference in `dbinom`).

For the non-central case of `dt`, C code contributed by Claus Ekstrøm based on the relationship (for $x \neq 0$) to the cumulative distribution.

For the central case of `pt`, a normal approximation in the tails, otherwise via `pbeta`.

For the non-central case of `pt` based on a C translation of

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central t distribution, *Applied Statistics* **38**, 185–189.

This computes the lower tail only, so the upper tail currently suffers from cancellation and a warning will be given when this is likely to be significant.

For central `qt`, a C translation of

Hill, G. W. (1970) Algorithm 396: Student's t-quantiles. *Communications of the ACM*, **13(10)**, 619–620.

altered to take account of

Hill, G. W. (1981) Remark on Algorithm 396, *ACM Transactions on Mathematical Software*, **7**, 250–1.

The non-central case is done by inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Except non-central versions.)

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 28 and 31. Wiley, New York.

See Also

Distributions for other standard distributions, including `df` for the F distribution.

Examples

```
require(graphics)

1 - pt(1:5, df = 1)
qt(.975, df = c(1:10,20,50,100,1000))

tt <- seq(0, 10, length.out = 21)
ncp <- seq(0, 6, length.out = 31)
ptn <- outer(tt, ncp, function(t, d) pt(t, df = 3, ncp = d))
t.tit <- "Non-central t - Probabilities"
image(tt, ncp, ptn, zlim = c(0,1), main = t.tit)
persp(tt, ncp, ptn, zlim = 0:1, r = 2, phi = 20, theta = 200, main = t.tit,
      xlab = "t", ylab = "non-centrality parameter",
      zlab = "Pr(T <= t)")

plot(function(x) dt(x, df = 3, ncp = 2), -3, 11, ylim = c(0, 0.32),
     main = "Non-central t - Density", yaxs = "i")

## Relation between F_t(.) = pt(x, n) and pbeta():
ptBet <- function(t, n) {
    x <- n/(n + t^2)
    r <- pb <- pbeta(x, n/2, 1/2) / 2
    pos <- t > 0
    r[pos] <- 1 - pb[pos]
    r
}
x <- seq(-5, 5, by = 1/8)
nu <- 3:10
pt. <- outer(x, nu, pt)
ptB <- outer(x, nu, ptBet)
## matplot(x, pt., type = "l")
stopifnot(all.equal(pt., ptB, tolerance = 1e-15))
```

`termplot` *Plot Regression Terms*

Description

Plots regression terms against their predictors, optionally with standard errors and partial residuals added.

Usage

```
termplot(model, data = NULL, envir = environment(formula(model)),
         partial.resid = FALSE, rug = FALSE,
         terms = NULL, se = FALSE,
         xlabs = NULL, ylabs = NULL, main = NULL,
         col.term = 2, lwd.term = 1.5,
         col.se = "orange", lty.se = 2, lwd.se = 1,
         col.res = "gray", cex = 1, pch = par("pch"),
         col.smth = "darkred", lty.smth = 2, span.smth = 2/3,
         ask = dev.interactive() && nb.fig < n.tms,
         use.factor.levels = TRUE, smooth = NULL, ylim = "common",
         plot = TRUE, transform.x = FALSE, ...)
```

Arguments

model	fitted model object
data	data frame in which variables in model can be found
envir	environment in which variables in model can be found
partial.resid	logical; should partial residuals be plotted?
rug	add rugplots (jittered 1-d histograms) to the axes?
terms	which terms to plot (default NULL means all terms); a vector passed to predict(.., type = "terms", terms = *).
se	plot pointwise standard errors?
xlabs	vector of labels for the x axes
ylabs	vector of labels for the y axes
main	logical, or vector of main titles; if TRUE, the model's call is taken as main title, NULL or FALSE mean no titles.
col.term, lwd.term	color and line width for the 'term curve', see lines.
col.se, lty.se, lwd.se	color, line type and line width for the 'twice-standard-error curve' when se = TRUE.
col.res, cex, pch	color, plotting character expansion and type for partial residuals, when partial.resid = TRUE, see points.
ask	logical; if TRUE, the user is *ask*ed before each plot, see par(ask=.).
use.factor.levels	Should x-axis ticks use factor levels or numbers for factor terms?
smooth	NULL or a function with the same arguments as panel.smooth to draw a smooth through the partial residuals for non-factor terms

`lty.smth`, `col.smth`, `span.smth`
: Passed to `smooth`

`ylim`
: an optional range for the y axis, or `"common"` when a range sufficient for all the plot will be computed, or `"free"` when limits are computed for each plot.

`plot`
: if set to `FALSE` plots are not produced: instead a list is returned containing the data that would have been plotted.

`transform.x`
: logical vector; if an element (recycled as necessary) is `TRUE`, partial residuals for the corresponding term are plotted against transformed values. The model response is then a straight line, allowing a ready comparison against the data or against the curve obtained from `smooth-panel.smooth`.

`...`
: other graphical parameters.

Details

The `model` object must have a `predict` method that accepts `type = "terms"`, e.g., `glm` in the **stats** package, `coxph` and `survreg` in the **survival** package.

For the `partial.resid = TRUE` option `model` must have a `residuals` method that accepts `type = "partial"`, which `lm` and `glm` do.

The `data` argument should rarely be needed, but in some cases `termplot` may be unable to reconstruct the original data frame. Using `na.action=na.exclude` makes these problems less likely.

Nothing sensible happens for interaction terms, and they may cause errors.

The `plot = FALSE` option is useful when some special action is needed, e.g. to overlay the results of two different models or to plot confidence bands.

Value

For `plot = FALSE`, a list with one element for each plot which would have been produced. Each element of the list is a data frame with variables `x`, `y`, and optionally the pointwise standard errors `se`. For continuous predictors `x` will contain the ordered unique values and for a factor it will be a factor containing one instance of each level. The list has attribute `"constant"` copied from the predicted terms object.

Otherwise, the number of terms, invisibly.

See Also

For (generalized) linear models, `plot.lm` and `predict.glm`.

Examples

```
require(graphics)

had.splines <- "package:splines" %in% search()
if(!had.splines) rs <- require(splines)
x <- 1:100
z <- factor(rep(LETTERS[1:4], 25))
y <- rnorm(100, sin(x/10)+as.numeric(z))
model <- glm(y ~ ns(x, 6) + z)
```

```
par(mfrow = c(2,2)) ## 2 x 2 plots for same model :
termplot(model, main = paste("termplot( ", deparse(model$call)," ...)"))
termplot(model, rug = TRUE)
termplot(model, partial.resid = TRUE, se = TRUE, main = TRUE)
termplot(model, partial.resid = TRUE, smooth = panel.smooth, span.smth = 1/4)
if(!had.splines && rs) detach("package:splines")

if(requireNamespace("MASS", quietly = TRUE)) {
hills.lm <- lm(log(time) ~ log(climb)+log(dist), data = MASS::hills)
termplot(hills.lm, partial.resid = TRUE, smooth = panel.smooth,
        terms = "log(dist)", main = "Original")
termplot(hills.lm, transform.x = TRUE,
         partial.resid = TRUE, smooth = panel.smooth,
  terms = "log(dist)", main = "Transformed")

}
```

`terms` *Model Terms*

Description

The function `terms` is a generic function which can be used to extract *terms* objects from various kinds of R data objects.

Usage

```
terms(x, ...)
```

Arguments

`x`	object used to select a method to dispatch.
`...`	further arguments passed to or from other methods.

Details

There are methods for classes `"aovlist"`, and `"terms" "formula"` (see `terms.formula`): the default method just extracts the `terms` component of the object, or failing that a `"terms"` attribute (as used by `model.frame`).

There are `print` and `labels` methods for class `"terms"`: the latter prints the term labels (see `terms.object`).

Value

An object of class `c("terms", "formula")` which contains the *terms* representation of a symbolic model. See `terms.object` for its structure.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

terms.object, terms.formula, lm, glm, formula.

---

terms.formula *Construct a terms Object from a Formula*

Description

This function takes a formula and some optional arguments and constructs a terms object. The terms object can then be used to construct a model.matrix.

Usage

```
## S3 method for class 'formula'
terms(x, specials = NULL, abb = NULL, data = NULL, neg.out = TRUE,
      keep.order = FALSE, simplify = FALSE, ...,
      allowDotAsName = FALSE)
```

Arguments

x	a formula.
specials	which functions in the formula should be marked as special in the terms object? A character vector or NULL.
abb	Not implemented in R; deprecated.
data	a data frame from which the meaning of the special symbol . can be inferred. It is used only if there is a . in the formula.
neg.out	Not implemented in R; deprecated.
keep.order	a logical value indicating whether the terms should keep their positions. By default, when FALSE, the terms are reordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on. Effects of a given order are kept in the order specified.
simplify	should the formula be expanded and simplified, the pre-1.7.0 behaviour?
...	further arguments passed to or from other methods.
allowDotAsName	normally . in a formula refers to the remaining variables contained in data. Exceptionally, . can be treated as a name for non-standard uses of formulae.

Details

Not all of the options work in the same way that they do in S and not all are implemented.

Value

A terms object is returned. It is the re-ordered formula (unless keep.order = TRUE) with several attributes, see terms.object for details. In all cases variables within an interaction term in the formula are re-ordered by the ordering of the "variables" attribute, which is the order in which the variables occur in the formula.

See Also

terms, terms.object, also for examples.

---

terms.object *Description of Terms Objects*

Description

An object of class terms holds information about a model. Usually the model was specified in terms of a formula and that formula was used to determine the terms object.

Details

The object itself is simply the result of terms.formula(<formula>). It has a number of attributes and they are used to construct the model frame:

factors An integer matrix of variables by terms showing which variables appear in which terms. The entries are

- **0** if the variable does not occur in the term,
- **1** if it does occur and should be coded by contrasts, and
- **2** if it occurs and should be coded via dummy variables for all levels (as when a lower-order term is missing).

Note that variables in main effects always receive 1, even if the intercept is missing (in which case the first one should be coded with dummy variables). If there are no terms other than an intercept and offsets, this is integer(0).

term.labels A character vector containing the labels for each of the terms in the model, except for offsets. Note that these are after possible re-ordering of terms.

Non-syntactic names will be quoted by backticks: this makes it easier to re-construct the formula from the term labels.

variables A call to list of the variables in the model.

intercept Either 0, indicating no intercept is to be fit, or 1 indicating that an intercept is to be fit.

order A vector of the same length as term.labels indicating the order of interaction for each term.

response The index of the variable (in variables) of the response (the left hand side of the formula). Zero, if there is no response.

offset If the model contains offset terms there is an offset attribute indicating which variable(s) are offsets

`specials` If a `specials` argument was given to `terms.formula` there is a `specials` attribute, a pairlist of vectors (one for each specified special function) giving numeric indices of the arguments of the list returned as the `variables` attribute which contain these special functions.

`dataClasses` optional. A named character vector giving the classes (as given by `.MFclass`) of the variables used in a fit.

`predvars` optional. An expression to help in computing predictions at new covariate values; see `makepredictcall`.

The object has class `c("terms", "formula")`.

Note

These objects are different from those found in S. In particular there is no `formula` attribute: instead the object is itself a formula. (Thus, the mode of a terms object is different.)

Examples of the `specials` argument can be seen in the `aov` and `coxph` functions, the latter from package **survival**.

See Also

`terms`, `formula`.

Examples

```
## use of specials (as used for gam() in packages mgcv and gam)
(tf <- terms(y ~ x + x:z + s(x), specials = "s"))
## Note that the "factors" attribute has variables as row names
## and term labels as column names, both as character vectors.
attr(tf, "specials")    # index 's' variable(s)
rownames(attr(tf, "factors"))[attr(tf, "specials")$s]

## we can keep the order by
terms(y ~ x + x:z + s(x), specials = "s", keep.order = TRUE)
```

---

time *Sampling Times of Time Series*

Description

`time` creates the vector of times at which a time series was sampled.

`cycle` gives the positions in the cycle of each observation.

`frequency` returns the number of samples per unit time and `deltat` the time interval between observations (see `ts`).

Usage

```
time(x, ...)
## Default S3 method:
time(x, offset = 0, ts.eps = getOption("ts.eps"), ...)

cycle(x, ...)
frequency(x, ...)
deltat(x, ...)
```

Arguments

x	a univariate or multivariate time-series, or a vector or matrix.
offset	can be used to indicate when sampling took place in the time unit. 0 (the default) indicates the start of the unit, 0.5 the middle and 1 the end of the interval.
ts.eps	time series comparison tolerance, used in time() to determine if values close than ts.eps to an integer should be round()ed to it in order to preserve the "year".
...	extra arguments for future methods.

Details

These are all generic functions, which will use the tsp attribute of x if it exists. time and cycle have methods for class ts that coerce the result to that class.

time() round()s values close to an integer, i.e., closer than ts.eps, since R 4.3.0. For previous behaviour, you can call it with ts.eps = 0.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ts, start, tsp, window.

date for clock time, system.time for CPU usage.

Examples

```
require(graphics)

cycle(presidents)
# a simple series plot
plot(as.vector(time(presidents)), as.vector(presidents), type = "l")
```

toeplitz *Create Symmetric and Asymmetric Toeplitz Matrix*

Description

In its simplest use, `toeplitz()` forms a symmetric Toeplitz matrix given its first column (or row). For the general case, asymmetric and non-square Toeplitz matrices are formed either by specifying the first column and row separately,

```
T1 <- toeplitz(col, row)
```

or by

```
T <- toeplitz2(x, nr, nc)
```

where only one of `(nr, nc)` needs to be specified. In the latter case, the simple equivalence $T_{i,j} = x_{i-j+n_c}$ is fulfilled where n_c =`ncol(T)`.

Usage

```
toeplitz (x, r = NULL, symmetric = is.null(r))
toeplitz2(x, nrow = length(x) +1 - ncol, ncol = length(x) +1 - nrow)
```

Arguments

`x`	for `toeplitz(x, *)`: the first column of the Toeplitz matrix; for `toeplitz2(x, *)` it is the upper-and-left border of the Toeplitz matrix, i.e., from top-right to bottom-left, such that `T[i,j] == x[i-j + ncol]`.
`r`	the first row of the target Toeplitz matrix; only needed in asymmetric cases.
`symmetric`	optional `logical` indicating if the matrix should be symmetric.
`nrow, ncol`	the number of rows and columns; only one needs to be specified.

Value

The $n \times m$ Toeplitz matrix T; for

`toeplitz():` `dim(T)` is `(n,m)` and `m == length(x)` and `n == m` in the symmetric case or `n == length(r)` otherwise.

`toeplitz2():` `dim(T) == c(nrow, ncol)`.

Author(s)

A. Trapletti and Martin Maechler (speedup and asymmetric extensions)

Examples

```
x <- 1:5
toeplitz (x)

T. <- toeplitz (1:5, 11:13) # with a  *Warning* x[1] != r[1]
T2 <- toeplitz2(c(13:12, 1:5), 5, 3)# this is the same matrix:
stopifnot(identical(T., T2))

# Matrix of character (could also have logical, raw, complex ..) {also warning}:
noquote(toeplitz(letters[1:4], LETTERS[20:26]))

## A convolution/smoother weight matrix :
m <- 17
k <- length(wts <- c(76, 99, 60, 20, 1))
n <- m-k+1
## Convolution
W <- toeplitz2(c(rep(0, m-k), wts, rep(0, m-k)), ncol=n)

## "display" nicely :
if(requireNamespace("Matrix"))
   print(Matrix::Matrix(W))    else {
   colnames(W) <- paste0(",", if(n <= 9) 1:n else c(1:9, letters[seq_len(n-9)]))
   print(W)
}

## scale W to have column sums 1:
W. <- W / sum(wts)
all.equal(rep(1, ncol(W.)), colSums(W.), check.attributes = FALSE)
## Visualize "mass-preserving" convolution
x <- 1:n; f <- function(x) exp(-((x - .4*n)/3)^2)
y <- f(x) + rep_len(3:-2, n)/10
## Smoothing convolution:
y.hat <- W. %*% y # y.hat := smoothed(y) ("mass preserving" -> longer than y)
stopifnot(length(y.hat) == m, m == n + (k-1))
plot(x,y, type="b", xlim=c(1,m)); curve(f(x), 1,n, col="gray", lty=2, add=TRUE)
lines(1:m, y.hat, col=2, lwd=3)
rbind(sum(y), sum(y.hat)) ## mass preserved

## And, yes, convolve(y, *) does the same when called appropriately:
all.equal(c(y.hat), convolve(y, rev(wts/sum(wts)), type="open"))
```

---

ts *Time-Series Objects*

Description

The function `ts` is used to create time-series objects.

`as.ts` and `is.ts` coerce an object to a time-series and test whether an object is a time series.

Usage

```
ts(data = NA, start = 1, end = numeric(), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"),
   class = if(nseries > 1) c("mts", "ts", "matrix", "array") else "ts",
   names = )
as.ts(x, ...)
is.ts(x)

is.mts(x)
```

Arguments

`data`	a vector or matrix of the observed time-series values. A data frame will be coerced to a numeric matrix via `data.matrix`. (See also 'Details'.)
`start`	the time of the first observation. Either a single number or a vector of two numbers (the second of which is an integer), which specify a natural time unit and a (1-based) number of samples into the time unit. See the examples for the use of the second form.
`end`	the time of the last observation, specified in the same way as `start`.
`frequency`	the number of observations per unit of time.
`deltat`	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of `frequency` or `deltat` should be provided.
`ts.eps`	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than `ts.eps`.
`class`	class to be given to the result, or none if `NULL` or `"none"`. The default is `"ts"` for a single series, or `c("mts", "ts", "matrix", "array")` for multiple series.
`names`	a character vector of names for the series in a multiple series: defaults to the colnames of `data`, or `"Series 1"`, `"Series 2"`, ...
`x`	an arbitrary R object.
`...`	arguments passed to methods (unused for the default method).

Details

The function `ts` is used to create time-series objects. These are vectors or matrices which inherit from class `"ts"` (and have additional attributes) which represent data which has been sampled at equispaced points in time. In the matrix case, each column of the matrix `data` is assumed to contain a single (univariate) time series. Time series must have at least one observation, and although they need not be numeric there is very limited support for non-numeric series.

Class `"ts"` has a number of methods. In particular arithmetic will attempt to align time axes, and subsetting to extract subsets of series can be used (e.g., `EuStockMarkets[, "DAX"]`). However, subsetting the first (or only) dimension will return a matrix or vector, as will matrix subsetting. Subassignment can be used to replace values but not to extend a series (see `window`). There is a method for `t` that transposes the series as a matrix (a one-column matrix if a vector) and hence returns a result that does not inherit from class `"ts"`.

Argument `frequency` indicates the sampling frequency of the time series, with the default value `1` indicating one sample in each unit time interval. For example, one could use a value of `7` for `frequency` when the data are sampled daily, and the natural time period is a week, or `12` when the data are sampled monthly and the natural time period is a year. Values of `4` and `12` are assumed in (e.g.) `print` methods to imply a quarterly and monthly series respectively. `frequency` need not be a whole number: for example, `frequency = 0.2` would imply sampling once every five time units.

`as.ts` is generic. Its default method will use the `tsp` attribute of the object if it has one to set the start and end times and frequency.

`is.ts()` tests if an object is a time series, i.e., inherits from `"ts"` and is of positive length.

`is.mts(x)` tests if an object `x` is a multivariate time series, i.e., fulfills `is.ts(x)`, `is.matrix(x)` and inherits from class `"mts"`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`tsp`, `frequency`, `start`, `end`, `time`, `window`; `print.ts`, the print method for time series objects; `plot.ts`, the plot method for time series objects.

For other definitions of 'time series' (e.g., time-ordered observations) see the CRAN task view at https://CRAN.R-project.org/view=TimeSeries.

Examples

```
require(graphics)

ts(1:10, frequency = 4, start = c(1959, 2)) # 2nd Quarter of 1959
print( ts(1:10, frequency = 7, start = c(12, 2)), calendar = TRUE)
# print.ts(.)
## Using July 1954 as start date:
gnp <- ts(cumsum(1 + round(rnorm(100), 2)),
          start = c(1954, 7), frequency = 12)
plot(gnp) # using 'plot.ts' for time-series plot

## Multivariate
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1), frequency = 12)
class(z)
is.mts(z)
head(z) # as "matrix"
plot(z)
plot(z, plot.type = "single", lty = 1:3)

## A phase plot:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col = "blue",
     main = "Lag plot of New Haven temperatures")
```

ts-methods *Methods for Time Series Objects*

Description

Methods for objects of class "ts", typically the result of ts.

Usage

```
## S3 method for class 'ts'
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'ts'
na.omit(object, ...)
```

Arguments

x	an object of class "ts" containing the values to be differenced.
lag	an integer indicating which lag to use.
differences	an integer indicating the order of the difference.
object	a univariate or multivariate time series.
...	further arguments to be passed to or from methods.

Details

The na.omit method omits initial and final segments with missing values in one or more of the series. 'Internal' missing values will lead to failure.

Value

For the na.omit method, a time series without missing values. The class of object will be preserved.

See Also

diff; na.omit, na.fail, na.contiguous.

`ts.plot` *Plot Multiple Time Series*

Description

Plot several time series on a common plot. Unlike `plot.ts` the series can have a different time bases, but they should have the same frequency.

Usage

```
ts.plot(..., gpars = list())
```

Arguments

`...`	one or more univariate or multivariate time series.
`gpars`	list of named graphics parameters to be passed to the plotting functions. Those commonly used can be supplied directly in `....`

Value

None.

Note

Although this can be used for a single time series, `plot` is easier to use and is preferred.

See Also

`plot.ts`

Examples

```
require(graphics)

ts.plot(ldeaths, mdeaths, fdeaths,
        gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))
```

ts.union *Bind Two or More Time Series*

Description

Bind time series which have a common frequency. `ts.union` pads with NAs to the total time coverage, `ts.intersect` restricts to the time covered by all the series.

Usage

```
ts.intersect(..., dframe = FALSE)
ts.union(..., dframe = FALSE)
```

Arguments

`...`	two or more univariate or multivariate time series, or objects which can coerced to time series.
`dframe`	logical; if `TRUE` return the result as a data frame.

Details

As a special case, `...` can contain vectors or matrices of the same length as the combined time series of the time series present, as well as those of a single row.

Value

A time series object if `dframe` is `FALSE`, otherwise a data frame.

See Also

`cbind`.

Examples

```
ts.union(mdeaths, fdeaths)
cbind(mdeaths, fdeaths) # same as the previous line
ts.intersect(window(mdeaths, 1976), window(fdeaths, 1974, 1978))

sales1 <- ts.union(BJsales, lead = BJsales.lead)
ts.intersect(sales1, lead3 = lag(BJsales.lead, -3))
```

`tsdiag` *Diagnostic Plots for Time-Series Fits*

Description

A generic function to plot time-series diagnostics.

Usage

```
tsdiag(object, gof.lag, ...)
```

Arguments

`object`	a fitted time-series model
`gof.lag`	the maximum number of lags for a Portmanteau goodness-of-fit test
`...`	further arguments to be passed to particular methods

Details

This is a generic function. It will generally plot the residuals, often standardized, the autocorrelation function of the residuals, and the p-values of a Portmanteau test for all lags up to `gof.lag`.

The methods for `arima` and `StructTS` objects plots residuals scaled by the estimate of their (individual) variance, and use the Ljung–Box version of the portmanteau test.

Value

None. Diagnostics are plotted.

See Also

`arima`, `StructTS`, `Box.test`

Examples

```
require(graphics)

fit <- arima(lh, c(1,0,0))
tsdiag(fit)

## see also examples(arima)

(fit <- StructTS(log10(JohnsonJohnson), type = "BSM"))
tsdiag(fit)
```

`tsp` *Tsp Attribute of Time-Series-like Objects*

Description

`tsp` returns the `tsp` attribute (or `NULL`). It is included for compatibility with S version 2. `tsp<-` sets the `tsp` attribute. `hasTsp` ensures `x` has a `tsp` attribute, by adding one if needed.

Usage

```
tsp(x)
tsp(x) <- value
hasTsp(x)
```

Arguments

`x`	a vector or matrix or univariate or multivariate time-series.
`value`	a numeric vector of length 3 or `NULL`.

Details

The `tsp` attribute gives the start time *in time units*, the end time and the frequency (the number of observations per unit of time, e.g. 12 for a monthly series).

Assignments are checked for consistency.

Assigning `NULL` which removes the `tsp` attribute *and* any `"ts"` (or `"mts"`) class of `x`.

Value

An object which differs from `x` only in the `tsp` attribute (unless `NULL` is assigned).

`hasTsp` adds, if needed, an attribute with a start time and frequency of 1 and end time `NROW(x)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `time`, `start`.

`tsSmooth` *Use Fixed-Interval Smoothing on Time Series*

Description

Performs fixed-interval smoothing on a univariate time series via a state-space model. Fixed-interval smoothing gives the best estimate of the state at each time point based on the whole observed series.

Usage

```
tsSmooth(object, ...)
```

Arguments

`object`	a time-series fit. Currently only class `"StructTS"` is supported
`...`	possible arguments for future methods.

Value

A time series, with as many dimensions as the state space and results at each time point of the original series. (For seasonal models, only the current seasonal component is returned.)

Author(s)

B. D. Ripley

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

See Also

`KalmanSmooth`, `StructTS`.

For examples consult `AirPassengers`, `JohnsonJohnson` and `Nile`.

Tukey *The Studentized Range Distribution*

Description

Functions of the distribution of the studentized range, R/s, where R is the range of a standard normal sample and $df \times s^2$ is independently distributed as chi-squared with df degrees of freedom, see pchisq.

Usage

```
ptukey(q, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
qtukey(p, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles.
p	vector of probabilities.
nmeans	sample size for range (same for each group).
df	degrees of freedom for s (see below).
nranges	number of *groups* whose **maximum** range is considered.
log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If $n_g =$nranges is greater than one, R is the *maximum* of n_g groups of nmeans observations each.

Value

ptukey gives the distribution function and qtukey its inverse, the quantile function.

The length of the result is the maximum of the lengths of the numerical arguments. The other numerical arguments are recycled to that length. Only the first elements of the logical arguments are used.

Note

A Legendre 16-point formula is used for the integral of ptukey. The computations are relatively expensive, especially for qtukey which uses a simple secant method for finding the inverse of ptukey. qtukey will be accurate to the 4th decimal place.

Source

qtukey is in part adapted from Odeh and Evans (1974).

References

Copenhaver, Margaret Diponzio and Holland, Burt S. (1988). Computation of the distribution of the maximum studentized range statistic with application to multiple significance testing of simple effects. *Journal of Statistical Computation and Simulation*, **30**, 1–15. doi:10.1080/00949658808811082.

Odeh, R. E. and Evans, J. O. (1974). Algorithm AS 70: Percentage Points of the Normal Distribution. *Applied Statistics*, **23**, 96–97. doi:10.2307/2347061.

See Also

Distributions for standard distributions, including pnorm and qnorm for the corresponding functions for the normal distribution.

Examples

```
if(interactive())
  curve(ptukey(x, nm = 6, df = 5), from = -1, to = 8, n = 101)
(ptt <- ptukey(0:10, 2, df =  5))
(qtt <- qtukey(.95, 2, df =  2:11))
## The precision may be not much more than about 8 digits:
summary(abs(.95 - ptukey(qtt, 2, df = 2:11)))
```

TukeyHSD *Compute Tukey Honest Significant Differences*

Description

Create a set of confidence intervals on the differences between the means of the levels of a factor with the specified family-wise probability of coverage. The intervals are based on the Studentized range statistic, Tukey's 'Honest Significant Difference' method.

Usage

```
TukeyHSD(x, which, ordered = FALSE, conf.level = 0.95, ...)
```

Arguments

x	A fitted model object, usually an aov fit.
which	A character vector listing terms in the fitted model for which the intervals should be calculated. Defaults to all the terms.
ordered	A logical value indicating if the levels of the factor should be ordered according to increasing average in the sample before taking differences. If ordered is true then the calculated differences in the means will all be positive. The significant differences will be those for which the lwr end point is positive.
conf.level	A numeric value between zero and one giving the family-wise confidence level to use.
...	Optional additional arguments. None are used at present.

Details

This is a generic function: the description here applies to the method for fits of class `"aov"`.

When comparing the means for the levels of a factor in an analysis of variance, a simple comparison using t-tests will inflate the probability of declaring a significant difference when it is not in fact present. This because the intervals are calculated with a given coverage probability for each interval but the interpretation of the coverage is usually with respect to the entire family of intervals.

John Tukey introduced intervals based on the range of the sample means rather than the individual differences. The intervals returned by this function are based on this Studentized range statistics.

The intervals constructed in this way would only apply exactly to balanced designs where there are the same number of observations made at each level of the factor. This function incorporates an adjustment for sample size that produces sensible intervals for mildly unbalanced designs.

If `which` specifies non-factor terms these will be dropped with a warning: if no terms are left this is an error.

Value

A list of class `c("multicomp", "TukeyHSD")`, with one component for each term requested in `which`. Each component is a matrix with columns `diff` giving the difference in the observed means, `lwr` giving the lower end point of the interval, `upr` giving the upper end point and `p adj` giving the p-value after adjustment for the multiple comparisons.

There are `print` and `plot` methods for class `"TukeyHSD"`. The `plot` method does not accept `xlab`, `ylab` or `main` arguments and creates its own values for each plot.

Author(s)

Douglas Bates

References

Miller, R. G. (1981) *Simultaneous Statistical Inference*. Springer.

Yandell, B. S. (1997) *Practical Data Analysis for Designed Experiments*. Chapman & Hall.

See Also

`aov`, `qtukey`, `model.tables`, `glht` in package **multcomp**.

Examples

```
require(graphics)

summary(fm1 <- aov(breaks ~ wool + tension, data = warpbreaks))
TukeyHSD(fm1, "tension", ordered = TRUE)
plot(TukeyHSD(fm1, "tension"))
```

`Uniform` *The Uniform Distribution*

Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Arguments

`x`, `q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`min`, `max`	lower and upper limits of the distribution. Must be finite.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `min` or `max` are not specified they assume the default values of `0` and `1` respectively.

The uniform distribution has density

$$f(x) = \frac{1}{max - min}$$

for $min \leq x \leq max$.

For the case of $u := min == max$, the limit case of $X \equiv u$ is assumed, although there is no density in that case and `dunif` will return `NaN` (the error condition).

`runif` will not generate either of the extreme values unless `max = min` or `max-min` is small compared to `min`, and in particular not for the default arguments.

Value

`dunif` gives the density, `punif` gives the distribution function, `qunif` gives the quantile function, and `runif` generates random deviates.

The length of the result is determined by `n` for `runif`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The characteristics of output from pseudo-random number generators (such as precision and periodicity) vary widely. See `.Random.seed` for more information on R's random number generation algorithms.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`RNG` about random number generation in R.

Distributions for other standard distributions.

Examples

```
u <- runif(20)

## The following relations always hold :
punif(u) == u
dunif(u) == 1

var(runif(10000))  #- ~ = 1/12 = .08333
```

`uniroot` *One Dimensional Root (Zero) Finding*

Description

The function `uniroot` searches the interval from `lower` to `upper` for a root (i.e., zero) of the function `f` with respect to its first argument.

Setting `extendInt` to a non-"no" string, means searching for the correct `interval = c(lower,upper)` if `sign(f(x))` does not satisfy the requirements at the interval end points; see the 'Details' section.

Usage

```
uniroot(f, interval, ...,
        lower = min(interval), upper = max(interval),
        f.lower = f(lower, ...), f.upper = f(upper, ...),
        extendInt = c("no", "yes", "downX", "upX"), check.conv = FALSE,
        tol = .Machine$double.eps^0.25, maxiter = 1000, trace = 0)
```

Arguments

f	the function for which the root is sought.
interval	a vector containing the end-points of the interval to be searched for the root.
...	additional named or unnamed arguments to be passed to f
lower, upper	the lower and upper end points of the interval to be searched.
f.lower, f.upper	the same as f(upper) and f(lower), respectively. Passing these values from the caller where they are often known is more economical as soon as f() contains non-trivial computations.
extendInt	character string specifying if the interval c(lower,upper) should be extended or directly produce an error when f() does not have differing signs at the endpoints. The default, "no", keeps the search interval and hence produces an error. Can be abbreviated.
check.conv	logical indicating whether a convergence warning of the underlying uniroot should be caught as an error and if non-convergence in maxiter iterations should be an error instead of a warning.
tol	the desired accuracy (convergence tolerance).
maxiter	the maximum number of iterations.
trace	integer number; if positive, tracing information is produced. Higher values giving more details.

Details

Note that arguments after ... must be matched exactly.

Either interval or both lower and upper must be specified: the upper endpoint must be strictly larger than the lower endpoint. The function values at the endpoints must be of opposite signs (or zero), for extendInt="no", the default. Otherwise, if extendInt="yes", the interval is extended on both sides, in search of a sign change, i.e., until the search interval $[l, u]$ satisfies $f(l) \cdot f(u) \leq 0$.

If it is *known how* f changes sign at the root x_0, that is, if the function is increasing or decreasing there, extendInt can (and typically should) be specified as "upX" (for "upward crossing") or "downX", respectively. Equivalently, define $S := \pm 1$, to require $S = \text{sign}(f(x_0 + \epsilon))$ at the solution. In that case, the search interval $[l, u]$ possibly is extended to be such that $S \cdot f(l) \leq 0$ and $S \cdot f(u) \geq 0$.

uniroot() uses Fortran subroutine zeroin (from Netlib) based on algorithms given in the reference below. They assume a continuous function (which then is known to have at least one root in the interval).

Convergence is declared either if f(x) == 0 or the change in x for one step of the algorithm is less than tol (plus an allowance for representation error in x).

If the algorithm does not converge in maxiter steps, a warning is printed and the current approximation is returned.

f will be called as f(x, ...) for a numeric value of x.

The argument passed to f has special semantics and used to be shared between calls. The function should not copy it.

Value

A list with at least five components: `root` and `f.root` give the location of the root and the value of the function evaluated at that point. `iter` and `estim.prec` give the number of iterations used and an approximate estimated precision for `root`. (If the root occurs at one of the endpoints, the estimated precision is `NA`.) `init.it` contains the number of *init*ial `extendInt` *it*erations if there were any and is `NA` otherwise. In the case of such `extendInt` iterations, `iter` contains the `sum` of these and the `zeroin` iterations.

Further components may be added in the future.

Source

Based on 'zeroin.c' in https://netlib.org/c/brent.shar.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs, NJ: Prentice-Hall.

See Also

`polyroot` for all complex roots of a polynomial; `optimize`, `nlm`.

Examples

```
require(utils) # for str

## some platforms hit zero exactly on the first step:
## if so the estimated precision is 2/3.
f <- function (x, a) x - a
str(xmin <- uniroot(f, c(0, 1), tol = 0.0001, a = 1/3))

## handheld calculator example: fixed point of cos(.):
uniroot(function(x) cos(x) - x, lower = -pi, upper = pi, tol = 1e-9)$root

str(uniroot(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
            tol = 0.0001))
str(uniroot(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
            tol = 1e-10))

## Find the smallest value x for which exp(x) > 0 (numerically):
r <- uniroot(function(x) 1e80*exp(x) - 1e-300, c(-1000, 0), tol = 1e-15)
str(r, digits.d = 15) # around -745, depending on the platform.

exp(r$root)     # = 0, but not for r$root * 0.999...
minexp <- r$root * (1 - 10*.Machine$double.eps)
exp(minexp)     # typically denormalized


##--- uniroot() with new interval extension + checking features: --------------

f1 <- function(x) (121 - x^2)/(x^2+1)
```

```
f2 <- function(x) exp(-x)*(x - 12)

try(uniroot(f1, c(0,10)))
try(uniroot(f2, c(0, 2)))
##--> error: f() .. end points not of opposite sign

## where as  'extendInt="yes"'  simply first enlarges the search interval:
u1 <- uniroot(f1, c(0,10),extendInt="yes", trace=1)
u2 <- uniroot(f2, c(0,2), extendInt="yes", trace=2)
stopifnot(all.equal(u1$root, 11, tolerance = 1e-5),
          all.equal(u2$root, 12, tolerance = 6e-6))

## The *danger* of interval extension:
## No way to find a zero of a positive function, but
## numerically, f(-|M|) becomes zero :
u3 <- uniroot(exp, c(0,2), extendInt="yes", trace=TRUE)

## Nonsense example (must give an error):
tools::assertCondition( uniroot(function(x) 1, 0:1, extendInt="yes"),
                       "error", verbose=TRUE)

## Convergence checking :
sinc <- function(x) ifelse(x == 0, 1, sin(x)/x)
curve(sinc, -6,18); abline(h=0,v=0, lty=3, col=adjustcolor("gray", 0.8))

uniroot(sinc, c(0,5), extendInt="yes", maxiter=4) #-> "just" a warning


## now with  check.conv=TRUE, must signal a convergence error :

uniroot(sinc, c(0,5), extendInt="yes", maxiter=4, check.conv=TRUE)


### Weibull cumulative hazard (example origin, Ravi Varadhan):
cumhaz <- function(t, a, b) b * (t/b)^a
froot <- function(x, u, a, b) cumhaz(x, a, b) - u

n <- 1000
u <- -log(runif(n))
a <- 1/2
b <- 1
## Find failure times
ru <- sapply(u, function(x)
   uniroot(froot, u=x, a=a, b=b, interval= c(1.e-14, 1e04),
           extendInt="yes")$root)
ru2 <- sapply(u, function(x)
   uniroot(froot, u=x, a=a, b=b, interval= c(0.01,  10),
           extendInt="yes")$root)
stopifnot(all.equal(ru, ru2, tolerance = 6e-6))

r1 <- uniroot(froot, u= 0.99, a=a, b=b, interval= c(0.01, 10),
              extendInt="up")
stopifnot(all.equal(0.99, cumhaz(r1$root, a=a, b=b)))
```

```
## An error if 'extendInt' assumes "wrong zero-crossing direction":

uniroot(froot, u= 0.99, a=a, b=b, interval= c(0.1, 10), extendInt="down")
```

---

update *Update and Re-fit a Model Call*

Description

update will update and (by default) re-fit a model. It does this by extracting the call stored in the object, updating the call and (by default) evaluating that call. Sometimes it is useful to call update with only one argument, for example if the data frame has been corrected.

"Extracting the call" in update() and similar functions uses getCall() which itself is a (S3) generic function with a default method that simply gets x$call.

Because of this, update() will often work (via its default method) on new model classes, either automatically, or by providing a simple getCall() method for that class.

Usage

```
update(object, ...)
## Default S3 method:
update(object, formula., ..., evaluate = TRUE)

getCall(x, ...)
```

Arguments

object, x	An existing fit from a model function such as lm, glm and many others.
formula.	Changes to the formula – see update.formula for details.
...	Additional arguments to the call, or arguments with changed values. Use name = NULL to remove the argument name.
evaluate	If true evaluate the new call else return the call.

Value

If evaluate = TRUE the fitted object, otherwise the updated call.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

update.formula

Examples

```
oldcon <- options(contrasts = c("contr.treatment", "contr.poly"))
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D9
summary(lm.D90 <- update(lm.D9, . ~ . - 1))
options(contrasts = c("contr.helmert", "contr.poly"))
update(lm.D9)
getCall(lm.D90)  # "through the origin"

options(oldcon)
```

`update.formula` *Model Updating*

Description

`update.formula` is used to update model formulae. This typically involves adding or dropping terms, but updates can be more general.

Usage

```
## S3 method for class 'formula'
update(old, new, ...)
```

Arguments

`old`	a model formula to be updated.
`new`	a formula giving a template which specifies how to update.
`...`	further arguments passed to or from other methods.

Details

Either or both of `old` and `new` can be objects such as length-one character vectors which can be coerced to a formula via `as.formula`.

The function works by first identifying the *left-hand side* and *right-hand side* of the `old` formula. It then examines the `new` formula and substitutes the *lhs* of the `old` formula for any occurrence of '.' on the left of `new`, and substitutes the *rhs* of the `old` formula for any occurrence of '.' on the right of `new`. The result is then simplified *via* `terms.formula(simplify = TRUE)`.

Value

The updated formula is returned. The environment of the result is that of `old`.

See Also

terms, model.matrix.

Examples

```
update(y ~ x,    ~ . + x2) #> y ~ x + x2
update(y ~ x, log(.) ~ . ) #> log(y) ~ x
update(. ~ u+v, res  ~ . ) #> res ~ u + v
```

---

var.test *F Test to Compare Two Variances*

Description

Performs an F test to compare the variances of two samples from normal populations.

Usage

```
var.test(x, ...)

## Default S3 method:
var.test(x, y, ratio = 1,
         alternative = c("two.sided", "less", "greater"),
         conf.level = 0.95, ...)

## S3 method for class 'formula'
var.test(formula, data, subset, na.action, ...)
```

Arguments

x, y	numeric vectors of data values, or fitted linear model objects (inheriting from class "lm").
ratio	the hypothesized ratio of the population variances of x and y.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
conf.level	confidence level for the returned confidence interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The null hypothesis is that the ratio of the variances of the populations from which `x` and `y` were drawn, or in the data to which the linear models `x` and `y` were fitted, is equal to `ratio`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the F test statistic.
`parameter`	the degrees of the freedom of the F distribution of the test statistic.
`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the ratio of the population variances.
`estimate`	the ratio of the sample variances of `x` and `y`.
`null.value`	the ratio of population variances under the null.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"F test to compare two variances"`.
`data.name`	a character string giving the names of the data.

See Also

`bartlett.test` for testing homogeneity of variances in more than two samples from normal distributions; `ansari.test` and `mood.test` for two rank based (nonparametric) two-sample tests for difference in scale.

Examples

```
x <- rnorm(50, mean = 0, sd = 2)
y <- rnorm(30, mean = 1, sd = 1)
var.test(x, y)                  # Do x and y have the same variance?
var.test(lm(x ~ 1), lm(y ~ 1))  # The same.
```

`varimax` *Rotation Methods for Factor Analysis*

Description

These functions 'rotate' loading matrices in factor analysis.

Usage

```
varimax(x, normalize = TRUE, eps = 1e-5)
promax(x, m = 4)
```

Arguments

x	A loadings matrix, with p rows and $k < p$ columns
m	The power used the target for promax. Values of 2 to 4 are recommended.
normalize	logical. Should Kaiser normalization be performed? If so the rows of x are re-scaled to unit length before rotation, and scaled back afterwards.
eps	The tolerance for stopping: the relative change in the sum of singular values.

Details

These seek a 'rotation' of the factors x %*% T that aims to clarify the structure of the loadings matrix. The matrix T is a rotation (possibly with reflection) for varimax, but a general linear transformation for promax, with the variance of the factors being preserved.

Value

A list with components

loadings	The 'rotated' loadings matrix, x %*% rotmat, of class "loadings".
rotmat	The 'rotation' matrix.

References

Hendrickson, A. E. and White, P. O. (1964). Promax: a quick method for rotation to orthogonal oblique structure. *British Journal of Statistical Psychology*, **17**, 65–70. doi:10.1111/j.2044-8317.1964.tb00244.x.

Horst, P. (1965). *Factor Analysis of Data Matrices*. Holt, Rinehart and Winston. Chapter 10.

Kaiser, H. F. (1958). The varimax criterion for analytic rotation in factor analysis. *Psychometrika*, **23**, 187–200. doi:10.1007/BF02289233.

Lawley, D. N. and Maxwell, A. E. (1971). *Factor Analysis as a Statistical Method*, second edition. Butterworths.

See Also

factanal, Harman74.cor.

Examples

```
## varimax with normalize = TRUE is the default
fa <- factanal( ~., 2, data = swiss)
varimax(loadings(fa), normalize = FALSE)
promax(loadings(fa))
```

vcov *Calculate Variance-Covariance Matrix for a Fitted Model Object*

Description

Returns the variance-covariance matrix of the main parameters of a fitted model object. The "main" parameters of model correspond to those returned by coef, and typically do not contain a nuisance scale parameter (sigma).

Usage

```
vcov(object, ...)
## S3 method for class 'lm'
vcov(object, complete = TRUE, ...)
## and also for '[summary.]glm' and 'mlm'
## S3 method for class 'aov'
vcov(object, complete = FALSE, ...)

.vcov.aliased(aliased, vc, complete = TRUE)
```

Arguments

object	a fitted model object, typically. Sometimes also a summary() object of such a fitted model.
complete	for the aov, lm, glm, mlm, and where applicable summary.lm etc methods: logical indicating if the full variance-covariance matrix should be returned also in case of an over-determined system where some coefficients are undefined and coef(.) contains NAs correspondingly. When complete = TRUE, vcov() is compatible with coef() also in this singular case.
...	additional arguments for method functions. For the glm method this can be used to pass a dispersion parameter.
aliased	a logical vector typically identical to is.na(coef(.)) indicating which coefficients are 'aliased'.
vc	a variance-covariance matrix, typically "incomplete", i.e., with no rows and columns for aliased coefficients.

Details

vcov() is a generic function and functions with names beginning in vcov. will be methods for this function. Classes with methods for this function include: lm, mlm, glm, nls, summary.lm, summary.glm, negbin, polr, rlm (in package **MASS**), multinom (in package **nnet**) gls, lme (in package **nlme**), coxph and survreg (in package **survival**).

(vcov() methods for summary objects allow more efficient and still encapsulated access when both summary(mod) and vcov(mod) are needed.)

.vcov.aliased() is an auxiliary function useful for vcov method implementations which have to deal with singular model fits encoded via NA coefficients: It augments a vcov–matrix vc by NA

rows and columns where needed, i.e., when some entries of `aliased` are true and `vc` is of smaller dimension than `length(aliased)`.

Value

A matrix of the estimated covariances between the parameter estimates in the linear or non-linear predictor of the model. This should have row and column names corresponding to the parameter names given by the `coef` method.

When some coefficients of the (linear) model are undetermined and hence `NA` because of linearly dependent terms (or an "over specified" model), also called "aliased", see `alias`, then since R version 3.5.0, `vcov()` (iff `complete = TRUE`, i.e., by default for `lm` etc, but not for `aov`) contains corresponding rows and columns of `NA`s, wherever `coef()` has always contained such `NA`s.

`Weibull` *The Weibull Distribution*

Description

Density, distribution function, quantile function and random generation for the Weibull distribution with parameters `shape` and `scale`.

Usage

```
dweibull(x, shape, scale = 1, log = FALSE)
pweibull(q, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
qweibull(p, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
rweibull(n, shape, scale = 1)
```

Arguments

`x`, `q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`shape`, `scale`	shape and scale parameters, the latter defaulting to 1.
`log`, `log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Weibull distribution with `shape` parameter a and `scale` parameter σ has density given by

$$f(x) = (a/\sigma)(x/\sigma)^{a-1} \exp(-(x/\sigma)^a)$$

for $x > 0$. The cumulative distribution function is $F(x) = 1 - \exp(-(x/\sigma)^a)$ on $x > 0$, the mean is $E(X) = \sigma\Gamma(1 + 1/a)$, and the $Var(X) = \sigma^2(\Gamma(1 + 2/a) - (\Gamma(1 + 1/a))^2)$.

Value

`dweibull` gives the density, `pweibull` gives the distribution function, `qweibull` gives the quantile function, and `rweibull` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rweibull`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is

```
-pweibull(t, a, b, lower = FALSE, log = TRUE)
```

which is just $H(t) = (t/b)^a$.

Source

`[dpq]weibull` are calculated directly from the definitions. `rweibull` uses inversion.

References

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 21. Wiley, New York.

See Also

Distributions for other standard distributions, including the Exponential which is a special case of the Weibull distribution.

Examples

```
x <- c(0, rlnorm(50))
all.equal(dweibull(x, shape = 1), dexp(x))
all.equal(pweibull(x, shape = 1, scale = pi), pexp(x, rate = 1/pi))
## Cumulative hazard H():
all.equal(pweibull(x, 2.5, pi, lower.tail = FALSE, log.p = TRUE),
          -(x/pi)^2.5, tolerance = 1e-15)
all.equal(qweibull(x/11, shape = 1, scale = pi), qexp(x/11, rate = 1/pi))
```

`weighted.mean` *Weighted Arithmetic Mean*

Description

Compute a weighted mean.

Usage

```
weighted.mean(x, w, ...)

## Default S3 method:
weighted.mean(x, w, ..., na.rm = FALSE)
```

Arguments

`x`	an object containing the values whose weighted mean is to be computed.
`w`	a numerical vector of weights the same length as `x` giving the weights to use for elements of `x`.
`...`	arguments to be passed to or from methods.
`na.rm`	a logical value indicating whether `NA` values in `x` should be stripped before the computation proceeds.

Details

This is a generic function and methods can be defined for the first argument `x`: apart from the default methods there are methods for the date-time classes `"POSIXct"`, `"POSIXlt"`, `"difftime"` and `"Date"`. The default method will work for any numeric-like object for which `[`, multiplication, division and `sum` have suitable methods, including complex vectors.

If `w` is missing then all elements of `x` are given the same weight, otherwise the weights are normalized to sum to one (if possible: if their sum is zero or infinite the value is likely to be `NaN`).

Missing values in `w` are not handled specially and so give a missing value as the result. However, zero weights *are* handled specially and the corresponding `x` values are omitted from the sum.

Value

For the default method, a length-one numeric vector.

See Also

`mean`

Examples

```
## GPA from Siegel 1994
wt <- c(5,  5,  4,  1)/15
x <- c(3.7,3.3,3.5,2.8)
xm <- weighted.mean(x, wt)
```

`weighted.residuals` *Compute Weighted Residuals*

Description

Computed weighted residuals from a linear model fit.

Usage

```
weighted.residuals(obj, drop0 = TRUE)
```

Arguments

`obj`	R object, typically of class `lm` or `glm`.
`drop0`	logical. If `TRUE`, drop all cases with `weights == 0`.

Details

Weighted residuals are based on the deviance residuals, which for a `lm` fit are the raw residuals R_i multiplied by $\sqrt{w_i}$, where w_i are the `weights` as specified in `lm`'s call.

Dropping cases with weights zero is compatible with `influence` and related functions.

Value

Numeric vector of length n', where n' is the number of non-0 weights (`drop0 = TRUE`) or the number of observations, otherwise.

See Also

`residuals`, `lm.influence`, etc.

Examples

```
## following on from example(lm)

all.equal(weighted.residuals(lm.D9),
          residuals(lm.D9))
x <- 1:10
w <- 0:9
y <- rnorm(x)
weighted.residuals(lmxy <- lm(y ~ x, weights = w))
weighted.residuals(lmxy, drop0 = FALSE)
```

weights *Extract Model Weights*

Description

`weights` is a generic function which extracts fitting weights from objects returned by modeling functions.

Methods can make use of `napredict` methods to compensate for the omission of missing values. The default methods does so.

Usage

```
weights(object, ...)
```

Arguments

`object`	an object for which the extraction of model weights is meaningful.
`...`	other arguments passed to methods.

Value

Weights extracted from the object `object`: the default method looks for component `"weights"` and if not `NULL` calls `napredict` on it.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`weights.glm`

wilcox.test *Wilcoxon Rank Sum and Signed Rank Tests*

Description

Performs one- and two-sample Wilcoxon tests on vectors of data; the latter is also known as 'Mann-Whitney' test.

Usage

```
wilcox.test(x, ...)

## Default S3 method:
wilcox.test(x, y = NULL,
            alternative = c("two.sided", "less", "greater"),
            mu = 0, paired = FALSE, exact = NULL, correct = TRUE,
            conf.int = FALSE, conf.level = 0.95,
            tol.root = 1e-4, digits.rank = Inf, ...)

## S3 method for class 'formula'
wilcox.test(formula, data, subset, na.action = na.pass, ...)
```

Arguments

x	numeric vector of data values. Non-finite (e.g., infinite or missing) values will be omitted.
y	an optional numeric vector of data values: as with x non-finite values will be omitted.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number specifying an optional parameter used to form the null hypothesis. See 'Details'.
paired	a logical indicating whether you want a paired test.
exact	a logical indicating whether an exact p-value should be computed.
correct	a logical indicating whether to apply continuity correction in the normal approximation for the p-value.
conf.int	a logical indicating whether a confidence interval should be computed.
conf.level	confidence level of the interval.
tol.root	(when conf.int is true:) a positive numeric tolerance, used in uniroot(*, tol=tol.root) calls.
digits.rank	a number; if finite, rank(signif(r, digits.rank)) will be used to compute ranks for the test statistic instead of (the default) rank(r).
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs either 1 for a one-sample or paired test or a factor with two levels giving the corresponding groups. If lhs is of class "Pair" and rhs is 1, a paired test is done, see Examples.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs.
...	further arguments to be passed to or from methods. For the formula method, this includes arguments of the default method, but not paired.

Details

The formula interface is only applicable for the 2-sample tests.

If only `x` is given, or if both `x` and `y` are given and `paired` is `TRUE`, a Wilcoxon signed rank test of the null that the distribution of `x` (in the one sample case) or of `x - y` (in the paired two sample case) is symmetric about `mu` is performed.

Otherwise, if both `x` and `y` are given and `paired` is `FALSE`, a Wilcoxon rank sum test (equivalent to the Mann-Whitney test: see the Note) is carried out. In this case, the null hypothesis is that the distributions of `x` and `y` differ by a location shift of `mu` and the alternative is that they differ by some other location shift (and the one-sided alternative `"greater"` is that `x` is shifted to the right of `y`).

By default (if `exact` is not specified), an exact p-value is computed if the samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

For stability reasons, it may be advisable to use rounded data or to set `digits.rank = 7`, say, such that determination of ties does not depend on very small numeric differences (see the example).

Optionally (if argument `conf.int` is true), a nonparametric confidence interval and an estimator for the pseudomedian (one-sample case) or for the difference of the location parameters `x-y` is computed. (The pseudomedian of a distribution F is the median of the distribution of $(u + v)/2$, where u and v are independent, each with distribution F. If F is symmetric, then the pseudomedian and median coincide. See Hollander & Wolfe (1973), page 34.) Note that in the two-sample case the estimator for the difference in location parameters does **not** estimate the difference in medians (a common misconception) but rather the median of the difference between a sample from `x` and a sample from `y`.

If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations. These are continuity-corrected for the interval but *not* the estimate (as the correction depends on the `alternative`).

With small samples it may not be possible to achieve very high confidence interval coverages. If this happens a warning will be given and an interval with lower coverage will be substituted.

When `x` (and `y` if applicable) are valid, the function now always returns, also in the `conf.int = TRUE` case when a confidence interval cannot be computed, in which case the interval boundaries and sometimes the `estimate` now contain `NaN`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic with a name describing it.
`parameter`	the parameter(s) for the exact distribution of the test statistic.
`p.value`	the p-value for the test.
`null.value`	the location parameter `mu`.
`alternative`	a character string describing the alternative hypothesis.
`method`	the type of test applied.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the location parameter. (Only present if argument `conf.int = TRUE`.)

`estimate` an estimate of the location parameter. (Only present if argument `conf.int = TRUE`.)

Warning

This function can use large amounts of memory and stack (and even crash R if the stack limit is exceeded) if `exact = TRUE` and one sample is large (several thousands or more).

Note

The literature is not unanimous about the definitions of the Wilcoxon rank sum and Mann-Whitney tests. The two most common definitions correspond to the sum of the ranks of the first sample with the minimum value ($m(m+1)/2$ for a first sample of size m) subtracted or not: R subtracts. It seems Wilcoxon's original paper used the unadjusted sum of the ranks but subsequent tables subtracted the minimum.

R's value can also be computed as the number of all pairs `(x[i], y[j])` for which `y[j]` is not greater than `x[i]`, the most common definition of the Mann-Whitney test.

References

David F. Bauer (1972). Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690. doi:10.1080/01621459.1972.10481279.

Myles Hollander and Douglas A. Wolfe (1973). *Nonparametric Statistical Methods*. New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample).
Or second edition (1999).

See Also

`psignrank`, `pwilcox`.

`wilcox_test` in package **coin** for exact, asymptotic and Monte Carlo *conditional* p-values, including in the presence of ties.

`kruskal.test` for testing homogeneity in location parameters in the case of two or more samples; `t.test` for an alternative under normality assumptions [or large samples]

Examples

```
require(graphics)
## One-sample test.
## Hollander & Wolfe (1973), 29f.
## Hamilton depression scale factor measurements in 9 patients with
##  mixed anxiety and depression, taken at the first (x) and second
##  (y) visit after initiation of a therapy (administration of a
##  tranquilizer).
x <- c(1.83,  0.50,  1.62,  2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
wilcox.test(x, y, paired = TRUE, alternative = "greater")
wilcox.test(y - x, alternative = "less")    # The same.
wilcox.test(y - x, alternative = "less",
            exact = FALSE, correct = FALSE) # H&W large sample
```

```
                                        # approximation

## Formula interface to one-sample and paired tests

depression <- data.frame(first = x, second = y, change = y - x)
wilcox.test(change ~ 1, data = depression)
wilcox.test(Pair(first, second) ~ 1, data = depression)

## Two-sample test.
## Hollander & Wolfe (1973), 69f.
## Permeability constants of the human chorioamnion (a placental
##  membrane) at term (x) and between 12 to 26 weeks gestational
##  age (y).  The alternative of interest is greater permeability
##  of the human chorioamnion for the term pregnancy.
x <- c(0.80, 0.83, 1.89, 1.04, 1.45, 1.38, 1.91, 1.64, 0.73, 1.46)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
wilcox.test(x, y, alternative = "g")        # greater
wilcox.test(x, y, alternative = "greater",
            exact = FALSE, correct = FALSE) # H&W large sample
                                        # approximation

wilcox.test(rnorm(10), rnorm(10, 2), conf.int = TRUE)

## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
wilcox.test(Ozone ~ Month, data = airquality,
            subset = Month %in% c(5, 8))

## accuracy in ties determination via 'digits.rank':
wilcox.test( 4:2,    3:1,   paired=TRUE) # Warning:  cannot compute exact p-value with ties
wilcox.test((4:2)/10, (3:1)/10, paired=TRUE) # no ties => *no* warning
wilcox.test((4:2)/10, (3:1)/10, paired=TRUE, digits.rank = 9) # same ties as (4:2, 3:1)
```

`Wilcoxon` *Distribution of the Wilcoxon Rank Sum Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon rank sum statistic obtained from samples with size `m` and `n`, respectively.

Usage

```
dwilcox(x, m, n, log = FALSE)
pwilcox(q, m, n, lower.tail = TRUE, log.p = FALSE)
qwilcox(p, m, n, lower.tail = TRUE, log.p = FALSE)
rwilcox(nn, m, n)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
nn	number of observations. If length(nn) > 1, the length is taken to be the number required.
m, n	numbers of observations in the first and second sample, respectively. Can be vectors of positive integers.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let x and y be two random, independent samples of size m and n. Then the Wilcoxon rank sum statistic is the number of all pairs (x[i], y[j]) for which y[j] is not greater than x[i]. This statistic takes values between 0 and m * n, and its mean and variance are m * n / 2 and m * n * (m + n + 1) / 12, respectively.

If any of the first three arguments are vectors, the recycling rule is used to do the calculations for all combinations of the three up to the length of the longest vector.

Value

dwilcox gives the density, pwilcox gives the distribution function, qwilcox gives the quantile function, and rwilcox generates random deviates.

The length of the result is determined by nn for rwilcox, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than nn are recycled to the length of the result. Only the first elements of the logical arguments are used.

Warning

These functions can use large amounts of memory and stack (and even crash R if the stack limit is exceeded and stack-checking is not in place) if one sample is large (several thousands or more).

Note

S-PLUS used a different (but equivalent) definition of the Wilcoxon statistic: see wilcox.test for details.

Author(s)

Kurt Hornik

Source

These ("d","p","q") are calculated via recursion, based on `cwilcox(k, m, n)`, the number of choices with statistic `k` from samples of size `m` and `n`, which is itself calculated recursively and the results cached. Then `dwilcox` and `pwilcox` sum appropriate values of `cwilcox`, and `qwilcox` is based on inversion.

`rwilcox` generates a random permutation of ranks and evaluates the statistic. Note that it is based on the same C code as `sample()`, and hence is determined by `.Random.seed`, notably from `RNGkind(sample.kind = ..)` which changed with R version 3.6.0.

See Also

`wilcox.test` to calculate the statistic from data, find p values and so on.

Distributions for standard distributions, including `dsignrank` for the distribution of the *one-sample* Wilcoxon signed rank statistic.

Examples

```
require(graphics)

x <- -1:(4*6 + 1)
fx <- dwilcox(x, 4, 6)
Fx <- pwilcox(x, 4, 6)

layout(rbind(1,2), widths = 1, heights = c(3,2))
plot(x, fx, type = "h", col = "violet",
     main =  "Probabilities (density) of Wilcoxon-Statist.(n=6, m=4)")
plot(x, Fx, type = "s", col = "blue",
     main =  "Distribution of Wilcoxon-Statist.(n=6, m=4)")
abline(h = 0:1, col = "gray20", lty = 2)
layout(1) # set back

N <- 200
hist(U <- rwilcox(N, m = 4,n = 6), breaks = 0:25 - 1/2,
     border = "red", col = "pink", sub = paste("N =",N))
mtext("N * f(x),  f() = true \"density\"", side = 3, col = "blue")
 lines(x, N*fx, type = "h", col = "blue", lwd = 2)
points(x, N*fx, cex = 2)

## Better is a Quantile-Quantile Plot
qqplot(U, qw <- qwilcox((1:N - 1/2)/N, m = 4, n = 6),
       main = paste("Q-Q-Plot of empirical and theoretical quantiles",
                    "Wilcoxon Statistic,  (m=4, n=6)", sep = "\n"))
n <- as.numeric(names(print(tU <- table(U))))
text(n+.2, n+.5, labels = tU, col = "red")
```

window *Time (Series) Windows*

Description

window is a generic function which extracts the subset of the object x observed between the times start and end. If a frequency is specified, the series is then re-sampled at the new frequency.

Usage

```
window(x, ...)
## S3 method for class 'ts'
window(x, ...)
## Default S3 method:
window(x, start = NULL, end = NULL,
    frequency = NULL, deltat = NULL, extend = FALSE, ts.eps = getOption("ts.eps"), ...)

window(x, ...) <- value
## S3 replacement method for class 'ts'
window(x, start, end, frequency, deltat, ...) <- value
```

Arguments

x	a time-series (or other object if not replacing values).
start	the start time of the period of interest.
end	the end time of the period of interest.
frequency, deltat	the new frequency can be specified by either (or both if they are consistent).
extend	logical. If true, the start and end values are allowed to extend the series. If false, attempts to extend the series give a warning and are ignored.
ts.eps	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than ts.eps and boundaries (length-1 versions of start and end) are checked with fuzz ts.eps/frequency(x).
...	further arguments passed to or from other methods.
value	replacement values.

Details

The start and end times can be specified as for ts. If there is no observation at the new start or end, the immediately following (start) or preceding (end) observation time is used.

The replacement function has a method for ts objects, and is allowed to extend the series (with a warning). There is no default method.

Value

The value depends on the method. `window.default` will return a vector or matrix with an appropriate `tsp` attribute.

`window.ts` differs from `window.default` only in ensuring the result is a `ts` object.

If `extend = TRUE` the series will be padded with `NA`s if needed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`time`, `ts`.

Examples

```
window(presidents, 1960, c(1969,4)) # values in the 1960's
window(presidents, deltat = 1)  # All Qtr1s
window(presidents, start = c(1945,3), deltat = 1)  # All Qtr3s
window(presidents, 1944, c(1979,2), extend = TRUE)

pres <- window(presidents, 1945, c(1949,4)) # values in the 1940's
window(pres, 1945.25, 1945.50) <- c(60, 70)
window(pres, 1944, 1944.75) <- 0 # will generate a warning
window(pres, c(1945,4), c(1949,4), frequency = 1) <- 85:89
pres
```

---

xtabs *Cross Tabulation*

Description

Create a contingency table (optionally a sparse matrix) from cross-classifying factors, usually contained in a data frame, using a formula interface.

Usage

```
xtabs(formula = ~., data = parent.frame(), subset, sparse = FALSE,
      na.action, na.rm = FALSE, addNA = FALSE,
      exclude = if(!addNA) c(NA, NaN), drop.unused.levels = FALSE)

## S3 method for class 'xtabs'
print(x, na.print = "", ...)
```

Arguments

formula	a formula object with the cross-classifying variables (separated by +) on the right-hand side (or an object which can be coerced to a formula). Interactions are not allowed. On the left-hand side, one may optionally give a vector or a matrix of counts; in the latter case, the columns are interpreted as corresponding to the levels of a variable. This is useful if the data have already been tabulated, see the examples below.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
sparse	logical specifying if the result should be a *sparse* matrix, i.e., inheriting from sparseMatrix Only works for two factors (since there are no higher-order sparse array classes yet).
na.action	a function which indicates what should happen when the variables in formula (or subset) contain NAs. Defaults to na.pass, so na.rm and addNA, respectively, control the handling of missing values for the two sides of the formula. Using na.omit removes any incomplete cases.
na.rm	logical: should missing values on the left-hand side of the formula be treated as zero when computing the sum?
addNA	logical indicating if NAs in the factors should get a separate level and be counted, using addNA(*, ifany=TRUE). This has no effect if na.action = na.omit.
exclude	a vector of values to be excluded when forming the set of levels of the classifying factors.
drop.unused.levels	a logical indicating whether to drop unused levels in the classifying factors. If this is FALSE and there are unused levels, the table will contain zero marginals, and a subsequent chi-squared test for independence of the factors will not work.
x	an object of class "xtabs".
na.print	character string (or NULL) indicating how NA are printed. The default ("") does not show NAs clearly, and na.print = "NA" maybe advisable instead.
...	further arguments passed to or from other methods.

Details

There is a summary method for contingency table objects created by table or xtabs(*, sparse = FALSE), which gives basic information and performs a chi-squared test for independence of factors (note that the function chisq.test currently only handles 2-d tables).

If a left-hand side is given in formula, its entries are simply summed over the cells corresponding to the right-hand side; this also works if the LHS does not give counts.

For variables in formula which are factors, exclude must be specified explicitly; the default exclusions will not be used.

In R versions before 3.4.0, e.g., when na.action = na.pass, sometimes zeroes (0) were returned instead of NAs.

In R versions before 4.4.0, when !addNA as by default, the default na.action was na.omit, effectively treating missing counts as zero.

Value

By default, when sparse = FALSE, a contingency table in array representation of S3 class c("xtabs", "table"), with a "call" attribute storing the matched call.

When sparse = TRUE, a sparse numeric matrix, specifically an object of S4 class dgTMatrix from package **Matrix**.

See Also

table for traditional cross-tabulation, and as.data.frame.table which is the inverse operation of xtabs (see the DF example below).

sparseMatrix on sparse matrices in package **Matrix**.

Examples

```
## 'esoph' has the frequencies of cases and controls for all levels of
## the variables 'agegp', 'alcgp', and 'tobgp'.
xtabs(cbind(ncases, ncontrols) ~ ., data = esoph)
## Output is not really helpful ... flat tables are better:
ftable(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph))
## In particular if we have fewer factors ...
ftable(xtabs(cbind(ncases, ncontrols) ~ agegp, data = esoph))

## This is already a contingency table in array form.
DF <- as.data.frame(UCBAdmissions)
## Now 'DF' is a data frame with a grid of the factors and the counts
## in variable 'Freq'.
DF
## Nice for taking margins ...
xtabs(Freq ~ Gender + Admit, DF)
## And for testing independence ...
summary(xtabs(Freq ~ ., DF))

## with NA's
DN <- DF; DN[cbind(6:9, c(1:2,4,1))] <- NA
DN # 'Freq' is missing only for (Rejected, Female, B)
(xtNA <- xtabs(Freq ~ Gender + Admit, DN))     # NA prints 'invisibly'
print(xtNA, na.print = "NA")                   # show NA's better
xtabs(Freq ~ Gender + Admit, DN, na.rm = TRUE) # ignore missing Freq
## Use addNA = TRUE to tabulate missing factor levels:
xtabs(Freq ~ Gender + Admit, DN, addNA = TRUE)
xtabs(Freq ~ Gender + Admit, DN, addNA = TRUE, na.rm = TRUE)
## na.action = na.omit removes all rows with NAs right from the start:
xtabs(Freq ~ Gender + Admit, DN, na.action = na.omit)

## Create a nice display for the warp break data.
warpbreaks$replicate <- rep_len(1:9, 54)
ftable(xtabs(breaks ~ wool + tension + replicate, data = warpbreaks))

### ---- Sparse Examples ----

if(require("Matrix")) withAutoprint({
```

```
## similar to "nlme"s  'ergoStool' :
d.ergo <- data.frame(Type = paste0("T", rep(1:4, 9*4)),
                     Subj = gl(9, 4, 36*4))
xtabs(~ Type + Subj, data = d.ergo) # 4 replicates each
set.seed(15) # a subset of cases:
xtabs(~ Type + Subj, data = d.ergo[sample(36, 10), ], sparse = TRUE)

## Hypothetical two-level setup:
inner <- factor(sample(letters[1:25], 100, replace = TRUE))
inout <- factor(sample(LETTERS[1:5], 25, replace = TRUE))
fr <- data.frame(inner = inner, outer = inout[as.integer(inner)])
xtabs(~ inner + outer, fr, sparse = TRUE)
})
```

Chapter 11

The stats4 package

---

stats4-package *Statistical Functions using S4 Classes*

Description

Statistical Functions using S4 classes.

Details

This package contains functions and classes for statistics using the S version 4 class system.

The methods currently support maximum likelihood (function `mle()` returning class `"mle"`), including methods for `logLik` for use with `AIC`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

---

coef-methods *Methods for Function* coef *in Package* **stats4**

Description

Extract the coefficient vector from `"mle"` objects.

Methods

`signature(object = "ANY")` Generic function: see `coef`.

`signature(object = "mle")` Extract the full coefficient vector (including any fixed coefficients) from the fit.

`signature(object = "summary.mle")` Extract the coefficient vector and standard errors from the summary of the fit.

confint-methods *Methods for Function* confint *in Package* **stats4**

Description

Generate confidence intervals

Methods

signature(object = "ANY") Generic function: see confint.

signature(object = "mle") First generate profile and then confidence intervals from the profile.

signature(object = "profile.mle") Generate confidence intervals based on likelihood profile.

logLik-methods *Methods for Function* logLik *in Package* **stats4**

Description

Extract the maximized log-likelihood from "mle" objects.

Methods

signature(object = "ANY") Generic function: see logLik.

signature(object = "mle") Extract log-likelihood from the fit.

Note

The mle method does not know about the number of observations unless nobs was specified on the call and so may not be suitable for use with BIC.

mle *Maximum Likelihood Estimation*

Description

Estimate parameters by the method of maximum likelihood.

Usage

```
mle(minuslogl, start,
       optim = stats::optim,
       method = if(!useLim) "BFGS" else "L-BFGS-B",
       fixed = list(), nobs, lower, upper, ...)
```

Arguments

minuslogl	Function to calculate negative log-likelihood.
start	Named list of vectors or single vector. Initial values for optimizer. By default taken from the default arguments of minuslogl
optim	Optimizer function. (Experimental)
method	Optimization method to use. See optim.
fixed	Named list of vectors or single vector. Parameter values to keep fixed during optimization.
nobs	optional integer: the number of observations, to be used for e.g. computing BIC.
lower, upper	Named lists of vectors or single vectors. Bounds for optim, if relevant.
...	Further arguments to pass to optim.

Details

The optim optimizer is used to find the minimum of the negative log-likelihood. An approximate covariance matrix for the parameters is obtained by inverting the Hessian matrix at the optimum. By default, optim from the stats package is used; other optimizers need to be plug-compatible, both with respect to arguments and return values.

The function minuslogl should take one or several arguments, each of which can be a vector. The optimizer optimizes a function which takes a single vector argument, containing the concatenation of the arguments to minuslogl, removing any values that should be held fixed. This function internally unpacks the argument vector, inserts the fixed values and calls minuslogl.

The vector arguments start, fixed, upper, and lower, can be given in both packed and unpacked form, either as a single vector or as a list of vectors. In the latter case, you only need to specify those list elements that are actually affected. For vector arguments, including those inside lists, use a default marker for those values that you don't want to set: NA for fixed and start, and +Inf, -Inf for upper, and lower.

Value

An object of class mle-class.

Note

Notice that the mll argument should calculate -log L (not -2 log L). It is for the user to ensure that the likelihood is correct, and that asymptotic likelihood inference is valid.

See Also

mle-class

Examples

```
## Avoid printing to unwarranted accuracy
od <- options(digits = 5)

## Simulated EC50 experiment with count data
```

```
x <- 0:10
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)

## Easy one-dimensional MLE:
nLL <- function(lambda) -sum(stats::dpois(y, lambda, log = TRUE))
fit0 <- mle(nLL, start = list(lambda = 5), nobs = NROW(y))

## sanity check --- notice that "nobs" must be input
## (not guaranteed to be meaningful for any likelihood)
stopifnot(nobs(fit0) == length(y))


# For 1D, this is preferable:
fit1 <- mle(nLL, start = list(lambda = 5), nobs = NROW(y),
            method = "Brent", lower = 1, upper = 20)

## This needs a constrained parameter space: most methods will accept NA
ll <- function(ymax = 15, xhalf = 6) {
    if(ymax > 0 && xhalf > 0)
      -sum(stats::dpois(y, lambda = ymax/(1+x/xhalf), log = TRUE))
    else NA
}
(fit <- mle(ll, nobs = length(y)))
mle(ll, fixed = list(xhalf = 6))

## Alternative using bounds on optimization
ll2 <- function(ymax = 15, xhalf = 6)
    -sum(stats::dpois(y, lambda = ymax/(1+x/xhalf), log = TRUE))
mle(ll2, lower = rep(0, 2))

AIC(fit)
BIC(fit)

summary(fit)
logLik(fit)
vcov(fit)
plot(profile(fit), absVal = FALSE)
confint(fit)

## Use bounded optimization
## The lower bounds are really > 0,
## but we use >=0 to stress-test profiling
(fit2 <- mle(ll2, lower = c(0, 0)))
plot(profile(fit2), absVal = FALSE)

## A better parametrization:
ll3 <- function(lymax = log(15), lxhalf = log(6))
    -sum(stats::dpois(y, lambda = exp(lymax)/(1+x/exp(lxhalf)), log = TRUE))
(fit3 <- mle(ll3))
plot(profile(fit3), absVal = FALSE)
exp(confint(fit3))

# Regression tests for bounded cases (this was broken in R 3.x)
```

```
fit4 <- mle(ll, lower = c(0, 4)) # has max on boundary
confint(fit4)

## direct check that fixed= and constraints work together
mle(ll, lower = c(0, 4), fixed=list(ymax=23)) # has max on boundary

## Linear regression using MLE
x <- 1:10
y <- c(0.48, 2.24, 2.22, 5.15, 4.64, 5.53, 7, 8.8, 7.67, 9.23)

LM_mll <- function(formula, data = environment(formula))
{
     y <- model.response(model.frame(formula, data))
     X <- model.matrix(formula, data)
     b0 <- numeric(NCOL(X))
     names(b0) <- colnames(X)
     function(b=b0, sigma=1)
         -sum(dnorm(y, X %*% b, sigma, log=TRUE))
}

mll <- LM_mll(y ~ x)

summary(lm(y~x)) # for comparison -- notice variance bias in MLE
summary(mle(mll, lower=c(-Inf,-Inf, 0.01)))
summary(mle(mll, lower=list(sigma = 0.01))) # alternative specification

confint(mle(mll, lower=list(sigma = 0.01)))
plot(profile(mle(mll, lower=list(sigma = 0.01))))

Binom_mll <- function(x, n)
{
    force(x); force(n) ## beware lazy evaluation
    function(p=.5) -dbinom(x, n, p, log=TRUE)
}

## Likelihood functions for different x.
## This code goes wrong, if force(x) is not used in Binom_mll:

curve(Binom_mll(0, 10)(p), xname="p", ylim=c(0, 10))
mll_list <- list(10)
for (x in 1:10)
    mll_list[[x]] <- Binom_mll(x, 10)
for (mll in mll_list)
    curve(mll(p), xname="p", add=TRUE)

mll <- Binom_mll(4,10)
mle(mll, lower = 1e-16, upper = 1-1e-16) # limits must be inside (0,1)

## Boundary case: This works, but fails if limits are set closer to 0 and 1
mll <- Binom_mll(0, 10)
mle(mll, lower=.005, upper=.995)

## Not run:
```

```
## We can use limits closer to the boundaries if we use the
## drop-in replacement optimr() from the optimx package.

mle(mll, lower = 1e-16, upper = 1-1e-16, optim=optimx::optimr)

## End(Not run)


options(od)
```

---

mle-class *Class* "mle" *for Results of Maximum Likelihood Estimation*

Description

This class encapsulates results of a generic maximum likelihood procedure.

Objects from the Class

Objects can be created by calls of the form `new("mle", ...)`, but most often as the result of a call to `mle`.

Slots

`call`: Object of class `"language"`. The call to `mle`.

`coef`: Object of class `"numeric"`. Estimated parameters.

`fullcoef`: Object of class `"numeric"`. Full parameter set of fixed and estimated parameters.

`fixed`: Object of class `"numeric"`. Fixed parameter values (`NA` for non-fixed parameters).

`vcov`: Object of class `"matrix"`. Approximate variance-covariance matrix.

`min`: Object of class `"numeric"`. Minimum value of objective function.

`details`: a `"list"`, as returned from `optim`.

`minuslogl`: Object of class `"function"`. The negative loglikelihood function.

`nobs`: `"integer"` of length one. The number of observations (often `NA`, when not set in call explicitly).

`method`: Object of class `"character"`. The optimization method used.

Methods

confint `signature(object = "mle")`: Confidence intervals from likelihood profiles.

logLik `signature(object = "mle")`: Extract maximized log-likelihood.

profile `signature(fitted = "mle")`: Likelihood profile generation.

nobs `signature(object = "mle")`: Number of observations, here simply accessing the `nobs` slot mentioned above.

show `signature(object = "mle")`: Display object briefly.

summary `signature(object = "mle")`: Generate object summary.

update `signature(object = "mle")`: Update fit.

vcov `signature(object = "mle")`: Extract variance-covariance matrix.

`plot-methods` *Methods for Function* `plot` *in Package* **stats4**

Description

Plot profile likelihoods for `"mle"` objects.

Usage

```
## S4 method for signature 'profile.mle,missing'
plot(x, levels, conf = c(99, 95, 90, 80, 50)/100, nseg = 50,
     absVal = TRUE, ...)
```

Arguments

`x`	an object of class `"profile.mle"`
`levels`	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually `conf` is used instead of giving `levels` explicitly.
`conf`	a numeric vector of confidence levels for profile-based confidence intervals on the parameters.
`nseg`	an integer value giving the number of segments to use in the spline interpolation of the profile t curves.
`absVal`	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to `TRUE`.
`...`	other arguments to the `plot` function can be passed here.

Methods

`signature(x = "ANY", y = "ANY")` Generic function: see `plot`.

`signature(x = "profile.mle", y = "missing")` Plot likelihood profiles for `x`.

profile-methods *Methods for Function* profile *in Package* **stats4**

Description

Profile likelihood for "mle" objects.

Usage

```
## S4 method for signature 'mle'
profile(fitted, which = 1:p, maxsteps = 100, alpha = 0.01,
        zmax = sqrt(qchisq(1 - alpha, 1L)), del = zmax/5,
        trace = FALSE, ...)
```

Arguments

fitted	Object to be profiled
which	Optionally select subset of parameters to profile.
maxsteps	Maximum number of steps to bracket zmax.
alpha	Significance level corresponding to zmax, based on a Scheffe-style multiple testing interval. Ignored if zmax is specified.
zmax	Cutoff for the profiled value of the signed root-likelihood.
del	Initial stepsize on root-likelihood scale.
trace	Logical. Print intermediate results.
...	Currently unused.

Details

The profiling algorithm tries to find an approximately evenly spaced set of at least five parameter values (in each direction from the optimum) to cover the root-likelihood function. Some care is taken to try and get sensible results in cases of high parameter curvature. Notice that it may not always be possible to obtain the cutoff value, since the likelihood might level off.

Value

An object of class "profile.mle", see "profile.mle-class".

Methods

signature(fitted = "ANY") Generic function: see profile.

signature(fitted = "mle") Profile the likelihood in the vicinity of the optimum of an "mle" object.

profile.mle-class *Class "profile.mle"; Profiling information for "mle" object*

Description

Likelihood profiles along each parameter of likelihood function

Objects from the Class

Objects can be created by calls of the form `new("profile.mle", ...)`, but most often by invoking `profile` on an `"mle"` object.

Slots

`profile`: Object of class `"list"`. List of profiles, one for each requested parameter. Each profile is a data frame with the first column called `z` being the signed square root of the -2 log likelihood ratio, and the others being the parameters with names prefixed by `par.vals`.

`summary`: Object of class `"summary.mle"`. Summary of object being profiled.

Methods

confint `signature(object = "profile.mle")`: Use profile to generate approximate confidence intervals for parameters.

plot `signature(x = "profile.mle", y = "missing")`: Plot profiles for each parameter.

See Also

`mle`, `mle-class`, `summary.mle-class`

show-methods *Methods for Function show in Package* **stats4**

Description

Show objects of classes `mle` and `summary.mle`

Methods

`signature(object = "mle")` Print simple summary of `mle` object. Just the coefficients and the call.

`signature(object = "summary.mle")` Shows call, table of coefficients and standard errors, and $-2 \log L$.

summary-methods *Methods for Function* summary *in Package* **stats4**

Description

Summarize objects

Methods

`signature(object = "ANY")` Generic function

`signature(object = "mle")` Generate a summary as an object of class `"summary.mle"`, containing estimates, asymptotic SE, and value of $-2 \log L$.

summary.mle-class *Class* "summary.mle", *Summary of* "mle" *Objects*

Description

Extract of `"mle"` object

Objects from the Class

Objects can be created by calls of the form `new("summary.mle", ...)`, but most often by invoking `summary` on an `"mle"` object. They contain values meant for printing by `show`.

Slots

`call`: Object of class `"language"` The call that generated the `"mle"` object.

`coef`: Object of class `"matrix"`. Estimated coefficients and standard errors

`m2logL`: Object of class `"numeric"`. Minus twice the log likelihood.

Methods

show `signature(object = "summary.mle")`: Pretty-prints `object`

coef `signature(object = "summary.mle")`: Extracts the contents of the `coef` slot

See Also

`summary`, `mle`, `mle-class`

update-methods *Methods for Function* update *in Package* **stats4**

Description

Update "mle" objects.

Usage

```
## S4 method for signature 'mle'
update(object, ..., evaluate = TRUE)
```

Arguments

object	An existing fit.
...	Additional arguments to the call, or arguments with changed values. Use name = NULL to remove the argument name.
evaluate	If true evaluate the new call else return the call.

Methods

signature(object = "ANY") Generic function: see update.

signature(object = "mle") Update a fit.

Examples

```
x <- 0:10
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax = 15, xhalf = 6)
    -sum(stats::dpois(y, lambda = ymax/(1+x/xhalf), log = TRUE))
fit <- mle(ll)
## note the recorded call contains ..1, a problem with S4 dispatch
update(fit, fixed = list(xhalf = 3))
```

vcov-methods *Methods for Function* vcov *in Package* **stats4**

Description

Extract the approximate variance-covariance matrix from "mle" objects.

Methods

signature(object = "ANY") Generic function: see vcov.

signature(object = "mle") Extract the estimated variance-covariance matrix for the estimated parameters (if any).

Chapter 12

The `tcltk` package

`tcltk-package` *Tcl/Tk Interface*

Description

Interface and language bindings to Tcl/Tk GUI elements.

Details

This package provides access to the platform-independent Tcl scripting language and Tk GUI elements. See TkWidgets for a list of supported widgets, TkWidgetcmds for commands to work with them, and references in those files for more.

The Tcl/Tk documentation is in the system man pages.

For a complete list of functions, use `ls("package:tcltk")`.

Note that Tk will not be initialized if there is no `DISPLAY` variable set, but Tcl can still be used. This is most useful to allow the loading of a package which depends on **tcltk** in a session that does not actually use it (e.g., during installation).

Author(s)

R Core Team

Maintainer: R Core Team <R-core@r-project.org>

TclInterface *Low-level Tcl/Tk Interface*

Description

These functions and variables provide the basic glue between R and the Tcl interpreter and Tk GUI toolkit. Tk windows may be represented via R objects. Tcl variables can be accessed via objects of class `tclVar` and the C level interface to Tcl objects is accessed via objects of class `tclObj`.

Usage

```
.Tcl(...)
.Tcl.objv(objv)
.Tcl.args(...)
.Tcl.args.objv(...)
.Tcl.callback(...)
.Tk.ID(win)
.Tk.newwin(ID)
.Tk.subwin(parent)
.TkRoot
.TkUp

tkdestroy(win)
is.tkwin(x)

tclvalue(x)
tclvalue(x) <- value

tclVar(init = "")
## S3 method for class 'tclVar'
as.character(x, ...)
## S3 method for class 'tclVar'
tclvalue(x)
## S3 replacement method for class 'tclVar'
tclvalue(x) <- value

tclArray()
## S3 method for class 'tclArray'
x[[...]]
## S3 replacement method for class 'tclArray'
x[[...]] <- value
## S3 method for class 'tclArray'
x$i
## S3 replacement method for class 'tclArray'
x$i <- value

## S3 method for class 'tclArray'
```

```
names(x)
## S3 method for class 'tclArray'
length(x)

tclObj(x)
tclObj(x) <- value
## S3 method for class 'tclVar'
tclObj(x)
## S3 replacement method for class 'tclVar'
tclObj(x) <- value

as.tclObj(x, drop = FALSE)
is.tclObj(x)

## S3 method for class 'tclObj'
as.character(x, ...)
## S3 method for class 'tclObj'
as.integer(x, ...)
## S3 method for class 'tclObj'
as.double(x, ...)
## S3 method for class 'tclObj'
as.logical(x, ...)
## S3 method for class 'tclObj'
as.raw(x, ...)
## S3 method for class 'tclObj'
tclvalue(x)

## Default S3 method:
tclvalue(x)
## Default S3 replacement method:
tclvalue(x) <- value


addTclPath(path = ".")
tclRequire(package, warn = TRUE)
tclVersion()
```

Arguments

objv	a named vector of Tcl objects
win	a window structure
x	an object
i	character or (unquoted) name
drop	logical. Indicates whether a single-element vector should be made into a simple Tcl object or a list of length one
value	For tclvalue assignments, a character string. For tclObj assignments, an object of class tclObj

ID	a window ID
parent	a window which becomes the parent of the resulting window
path	path to a directory containing Tcl packages
package	a Tcl package name
warn	logical. Warn if not found?
...	Additional arguments. See below.
init	initialization value

Details

Many of these functions are not intended for general use but are used internally by the commands that create and manipulate Tk widgets and Tcl objects. At the lowest level `.Tcl` sends a command as a text string to the Tcl interpreter and returns the result as an object of class `tclObj` (see below). A newer variant `.Tcl.objv` accepts arguments in the form of a named list of `tclObj` objects.

`.Tcl.args` converts an R argument list of `tag = value` pairs to the Tcl `-option value` style, thus enabling a simple translation between the two languages. To send a value with no preceding option flag to Tcl, just use an untagged argument. In the rare case one needs an option with no subsequent value `tag = NULL` can be used. Most values are just converted to character mode and inserted in the command string, but window objects are passed using their ID string, and callbacks are passed via the result of `.Tcl.callback`. Tags are converted to option flags simply by prepending a `-`

`.Tcl.args.objv` serves a similar purpose as `.Tcl.args` but produces a list of `tclObj` objects suitable for passing to `.Tcl.objv`. The names of the list are converted to Tcl option style internally by `.Tcl.objv`.

Callbacks can be either *atomic callbacks* handled by `.Tcl.callback` or expressions. An expression is treated as a list of atomic callbacks, with the following exceptions: if an element is a name, it is first evaluated in the callers frame, and likewise if it is an explicit function definition; the `break` expression is translated directly to the Tcl counterpart. `.Tcl.callback` converts R functions and unevaluated calls to Tcl command strings. The argument must be either a function closure or an object of mode `"call"` followed by an environment. The return value in the first case is of the form `R_call 0x408b94d4` in which the hexadecimal number is the memory address of the function. In the second case it will be of the form `R_call_lang 0x8a95904 0x819bfd0`. For expressions, a sequence of similar items is generated, separated by semicolons. `.Tcl.args` takes special precautions to ensure that functions or calls will continue to exist at the specified address by assigning the callback into the relevant window environment (see below).

Tk windows are represented as objects of class `tkwin` which are lists containing a `ID` field and an `env` field which is an R environments, enclosed in the global environment. The value of the `ID` field is identical to the Tk window name. The `env` environment contains a `parent` variable and a `num.subwin` variable. If the window obtains sub-windows and callbacks, they are added as variables to the environment. `.TkRoot` is the top window with ID "."; this window is not displayed in order to avoid ill effects of closing it via window manager controls. The `parent` variable is undefined for `.TkRoot`.

`.Tk.ID` extracts the `ID` of a window, `.Tk.newwin` creates a new window environment with a given ID and `.Tk.subwin` creates a new window which is a sub-window of a given parent window.

`.TkUp` is a logical flag to indicate whether the Tk widget system is active; if `FALSE`, only the Tcl interpreter is available.

tkdestroy destroys a window and also removes the reference to a window from its parent.

is.tkwin can be used to test whether a given object is a window environment.

tclVar creates a new Tcl variable and initializes it to init. An R object of class tclVar is created to represent it. Using as.character on the object returns the Tcl variable name. Accessing the Tcl variable from R is done using the tclvalue function, which can also occur on the left-hand side of assignments. If tclvalue is passed an argument which is not a tclVar object, then it will assume that it is a character string explicitly naming global Tcl variable. Tcl variables created by tclVar are uniquely named and automatically unset by the garbage collector when the representing object is no longer in use.

tclArray creates a new Tcl array and initializes it to the empty array. An R object of class tclArray and inheriting from class tclVar is created to represent it. You can access elements of the Tcl array using indexing with [[or $, which also allow replacement forms. Notice that Tcl arrays are associative by nature and hence unordered; indexing with a numeric index i refers to the element with the *name* as.character(i). Multiple indices are pasted together separated by commas to form a single name. You can query the length and the set of names in an array using methods for length and names, respectively; these cannot meaningfully be set so assignment forms exist only to print an error message.

It is possible to access Tcl's 'dual-ported' objects directly, thus avoiding parsing and deparsing of their string representation. This works by using objects of class tclObj. The string representation of such objects can be extracted (but not set) using tclvalue and conversion to vectors of mode "character", "double", "integer", "logical", and "raw" is performed using the standard coercion functions as.character, etc. Conversely, such vectors can be converted using as.tclObj. There is an ambiguity as to what should happen for length one vectors, controlled by the drop argument; there are cases where the distinction matters to Tcl, although mostly it treats them equivalently. Notice that tclvalue and as.character differ on an object whose string representation has embedded spaces, the former is sometimes to be preferred, in particular when applied to the result of tclread, tkgetOpenFile, and similar functions. The as.raw method returns a raw vector or a list of raw vectors and can be used to return binary data from Tcl.

The object behind a tclVar object is extracted using tclObj(x) which also allows an assignment form, in which the right hand side of the assignment is automatically converted using as.tclObj. There is a print method for tclObj objects; it prints '<Tcl>' followed by the string representation of the object. Notice that as.character on a tclVar object is the *name* of the corresponding Tcl variable and not the value.

Tcl packages can be loaded with tclRequire; it may be necessary to add the directory where they are found to the Tcl search path with addTclPath. The return value is a class "tclObj" object if it succeeds, or FALSE if it fails (when a warning is issued). To see the current search path as an R character vector, use:

```
strsplit(tclvalue('auto_path'), " ")[[1]]
```

The Tcl version (including patch level) is returned as a character string (such as "8.6.3").

Note

Strings containing unbalanced braces are currently not handled well in many circumstances.

See Also

TkWidgets, TkCommands, TkWidgetcmds.

capabilities("tcltk") to see if Tcl/Tk support was compiled into this build of R.

Examples

```
tclVersion()

.Tcl("format \"%s\n\" \"Hello, World!\"")

f <- function() cat("HI!\n")
.Tcl.callback(f)
.Tcl.args(text = "Push!", command = f) # NB: Different address


xyzzy <- tclVar(7913)
tclvalue(xyzzy)
tclvalue(xyzzy) <- "foo"
as.character(xyzzy)
tcl("set", as.character(xyzzy))

## Not run:
## These cannot be run by example() but should be OK when pasted
## into an interactive R session with the tcltk package loaded
top <- tktoplevel() # a Tk widget, see Tk-widgets
ls(envir = top$env, all.names = TRUE)

## End(Not run)

ls(envir = .TkRoot$env, all.names = TRUE) # .Tcl.args put a callback ref in here
```

---

tclServiceMode *Allow Tcl events to be serviced or not*

Description

This function controls or reports on the Tcl service mode, i.e., whether Tcl will respond to events.

Usage

```
tclServiceMode(on = NULL)
```

Arguments

on	(logical) Whether event servicing is turned on.

Details

If called with on == NULL (the default), no change is made.

Note that this blocks all Tcl/Tk activity, including for widgets from other packages. It may be better to manage mapping of windows individually.

Value

The value of the Tcl service mode before the call.

Examples

```
## see demo(tkcanvas) for an example
oldmode <- tclServiceMode(FALSE)
# Do some work to create a nice picture.
# Nothing will be displayed until...
tclServiceMode(oldmode)
## another idea is to use tkwm.withdraw() ... tkwm.deiconify()
```

TkCommands *Tk non-widget commands*

Description

These functions interface to Tk non-widget commands, such as the window manager interface commands and the geometry managers.

Usage

```
tcl(...)
tktitle(x)

tktitle(x) <- value

tkbell(...)
tkbind(...)
tkbindtags(...)
tkfocus(...)
tklower(...)
tkraise(...)

tkclipboard.append(...)
tkclipboard.clear(...)

tkevent.add(...)
tkevent.delete(...)
tkevent.generate(...)
tkevent.info(...)
```

```
tkfont.actual(...)
tkfont.configure(...)
tkfont.create(...)
tkfont.delete(...)
tkfont.families(...)
tkfont.measure(...)
tkfont.metrics(...)
tkfont.names(...)

tkgrab(...)
tkgrab.current(...)
tkgrab.release(...)
tkgrab.set(...)
tkgrab.status(...)

tkimage.create(...)
tkimage.delete(...)
tkimage.height(...)
tkimage.inuse(...)
tkimage.names(...)
tkimage.type(...)
tkimage.types(...)
tkimage.width(...)

## NB: some widgets also have a selection.clear command,
## hence the "X".

tkXselection.clear(...)
tkXselection.get(...)
tkXselection.handle(...)
tkXselection.own(...)

tkwait.variable(...)
tkwait.visibility(...)
tkwait.window(...)

## winfo actually has a large number of subcommands,
## but it's rarely used,
## so use tkwinfo("atom", ...) etc. instead.

tkwinfo(...)

# Window manager interface

tkwm.aspect(...)
tkwm.client(...)
tkwm.colormapwindows(...)
```

```
tkwm.command(...)
tkwm.deiconify(...)
tkwm.focusmodel(...)
tkwm.frame(...)
tkwm.geometry(...)
tkwm.grid(...)
tkwm.group(...)
tkwm.iconbitmap(...)
tkwm.iconify(...)
tkwm.iconmask(...)
tkwm.iconname(...)
tkwm.iconposition(...)
tkwm.iconwindow(...)
tkwm.maxsize(...)
tkwm.minsize(...)
tkwm.overrideredirect(...)
tkwm.positionfrom(...)
tkwm.protocol(...)
tkwm.resizable(...)
tkwm.sizefrom(...)
tkwm.state(...)
tkwm.title(...)
tkwm.transient(...)
tkwm.withdraw(...)


### Geometry managers

tkgrid(...)
tkgrid.bbox(...)
tkgrid.columnconfigure(...)
tkgrid.configure(...)
tkgrid.forget(...)
tkgrid.info(...)
tkgrid.location(...)
tkgrid.propagate(...)
tkgrid.rowconfigure(...)
tkgrid.remove(...)
tkgrid.size(...)
tkgrid.slaves(...)

tkpack(...)
tkpack.configure(...)
tkpack.forget(...)
tkpack.info(...)
tkpack.propagate(...)
tkpack.slaves(...)
```

```
tkplace(...)
tkplace.configure(...)
tkplace.forget(...)
tkplace.info(...)
tkplace.slaves(...)

## Standard dialogs
tkgetOpenFile(...)
tkgetSaveFile(...)
tkchooseDirectory(...)
tkmessageBox(...)
tkdialog(...)
tkpopup(...)


## File handling functions
tclfile.tail(...)
tclfile.dir(...)
tclopen(...)
tclclose(...)
tclputs(...)
tclread(...)
```

Arguments

x	A window object
value	For tktitle assignments, a character string.
...	Handled via .Tcl.args

Details

tcl provides a generic interface to calling any Tk or Tcl command by simply running .Tcl.args.objv on the argument list and passing the result to .Tcl.objv. Most of the other commands simply call tcl with a particular first argument and sometimes also a second argument giving the subcommand.

tktitle and its assignment form provides an alternate interface to Tk's wm title

There are far too many of these commands to describe them and their arguments in full. Please refer to the Tcl/Tk documentation for details. With a few exceptions, the pattern is that Tk subcommands like pack configure are converted to function names like tkpack.configure, and Tcl subcommands are like tclfile.dir.

See Also

TclInterface, TkWidgets, TkWidgetcmds

Examples

```
## Not run:
```

```
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
tkpack(l1 <- tklabel(tt, text = "Heave"), l2 <- tklabel(tt, text = "Ho"))
tkpack.configure(l1, side = "left")

## Try stretching the window and then

tkdestroy(tt)

## End(Not run)
```

---

tkpager *Page file using Tk text widget*

Description

This plugs into `file.show`, showing files in separate windows.

Usage

```
tkpager(file, header, title, delete.file)
```

Arguments

`file`	character vector containing the names of the files to be displayed
`header`	headers to use for each file
`title`	common title to use for the window(s). Pasted together with the `header` to form actual window title.
`delete.file`	logical. Should file(s) be deleted after display?

Note

The `"\b_"` string used for underlining is currently quietly removed. The font and background colour are currently hardcoded to Courier and gray90.

See Also

`file.show`

tkProgressBar *Progress Bars via Tk*

Description

Put up a Tk progress bar widget.

Usage

```
tkProgressBar(title = "R progress bar", label = "",
              min = 0, max = 1, initial = 0, width = 300)

getTkProgressBar(pb)
setTkProgressBar(pb, value, title = NULL, label = NULL)
## S3 method for class 'tkProgressBar'
close(con, ...)
```

Arguments

title, label	character strings, giving the window title and the label on the dialog box respectively.
min, max	(finite) numeric values for the extremes of the progress bar.
initial, value	initial or new value for the progress bar.
width	the width of the progress bar in pixels: the dialog box will be 40 pixels wider (plus frame).
pb, con	an object of class "tkProgressBar".
...	for consistency with the generic.

Details

tkProgressBar will display a widget containing a label and progress bar.

setTkProgessBar will update the value and for non-NULL values, the title and label (provided there was one when the widget was created). Missing (NA) and out-of-range values of value will be (silently) ignored.

The progress bar should be closed when finished with.

This will use the ttk::progressbar widget for Tk version 8.5 or later, otherwise R's copy of BWidget's progressbar.

Value

For tkProgressBar an object of class "tkProgressBar".

For getTkProgressBar and setTkProgressBar, a length-one numeric vector giving the previous value (invisibly for setTkProgressBar).

See Also

txtProgressBar

Examples

```
pb <- tkProgressBar("test progress bar", "Some information in %",
                    0, 100, 50)
Sys.sleep(0.5)
u <- c(0, sort(runif(20, 0, 100)), 100)
for(i in u) {
    Sys.sleep(0.1)
    info <- sprintf("%d%% done", round(i))
    setTkProgressBar(pb, i, sprintf("test (%s)", info), info)
}
Sys.sleep(5)
close(pb)
```

---

tkStartGUI *Tcl/Tk GUI startup*

Description

Starts up the Tcl/Tk GUI

Usage

```
tkStartGUI()
```

Details

Starts a GUI console implemented via a Tk text widget. This should probably be called at most once per session. Also redefines the file pager (as used by `help()`) to be the Tk pager.

Note

`tkStartGUI()` saves its evaluation environment as `.GUIenv`. This means that the user interface elements can be accessed in order to extend the interface. The three main objects are named `Term`, `Menu`, and `Toolbar`, and the various submenus and callback functions can be seen with `ls(envir = .GUIenv)`.

Author(s)

Peter Dalgaard

TkWidgetcmds *Tk widget commands*

Description

These functions interface to Tk widget commands.

Usage

```
tkactivate(widget, ...)
tkadd(widget, ...)
tkaddtag(widget, ...)
tkbbox(widget, ...)
tkcanvasx(widget, ...)
tkcanvasy(widget, ...)
tkcget(widget, ...)
tkcompare(widget, ...)
tkconfigure(widget, ...)
tkcoords(widget, ...)
tkcreate(widget, ...)
tkcurselection(widget, ...)
tkdchars(widget, ...)
tkdebug(widget, ...)
tkdelete(widget, ...)
tkdelta(widget, ...)
tkdeselect(widget, ...)
tkdlineinfo(widget, ...)
tkdtag(widget, ...)
tkdump(widget, ...)
tkentrycget(widget, ...)
tkentryconfigure(widget, ...)
tkfind(widget, ...)
tkflash(widget, ...)
tkfraction(widget, ...)
tkget(widget, ...)
tkgettags(widget, ...)
tkicursor(widget, ...)
tkidentify(widget, ...)
tkindex(widget, ...)
tkinsert(widget, ...)
tkinvoke(widget, ...)
tkitembind(widget, ...)
tkitemcget(widget, ...)
tkitemconfigure(widget, ...)
tkitemfocus(widget, ...)
tkitemlower(widget, ...)
tkitemraise(widget, ...)
```

```
tkitemscale(widget, ...)
tkmark.gravity(widget, ...)
tkmark.names(widget, ...)
tkmark.next(widget, ...)
tkmark.previous(widget, ...)
tkmark.set(widget, ...)
tkmark.unset(widget, ...)
tkmove(widget, ...)
tknearest(widget, ...)
tkpost(widget, ...)
tkpostcascade(widget, ...)
tkpostscript(widget, ...)
tkscan.mark(widget, ...)
tkscan.dragto(widget, ...)
tksearch(widget, ...)
tksee(widget, ...)
tkselect(widget, ...)
tkselection.adjust(widget, ...)
tkselection.anchor(widget, ...)
tkselection.clear(widget, ...)
tkselection.from(widget, ...)
tkselection.includes(widget, ...)
tkselection.present(widget, ...)
tkselection.range(widget, ...)
tkselection.set(widget, ...)
tkselection.to(widget, ...)
tkset(widget, ...)
tksize(widget, ...)
tktoggle(widget, ...)
tktag.add(widget, ...)
tktag.bind(widget, ...)
tktag.cget(widget, ...)
tktag.configure(widget, ...)
tktag.delete(widget, ...)
tktag.lower(widget, ...)
tktag.names(widget, ...)
tktag.nextrange(widget, ...)
tktag.prevrange(widget, ...)
tktag.raise(widget, ...)
tktag.ranges(widget, ...)
tktag.remove(widget, ...)
tktype(widget, ...)
tkunpost(widget, ...)
tkwindow.cget(widget, ...)
tkwindow.configure(widget, ...)
tkwindow.create(widget, ...)
tkwindow.names(widget, ...)
tkxview(widget, ...)
```

```
tkxview.moveto(widget, ...)
tkxview.scroll(widget, ...)
tkyposition(widget, ...)
tkyview(widget, ...)
tkyview.moveto(widget, ...)
tkyview.scroll(widget, ...)
```

Arguments

widget	The widget this applies to
...	Handled via .Tcl.args

Details

There are far too many of these commands to describe them and their arguments in full. Please refer to the Tcl/Tk documentation for details. Except for a few exceptions, the pattern is that Tcl widget commands possibly with subcommands like .a.b selection clear are converted to function names like tkselection.clear and the widget is given as the first argument.

See Also

TclInterface, TkWidgets, TkCommands

Examples

```
## Not run:
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
tkpack(txt.w <- tktext(tt))
tkinsert(txt.w, "0.0", "plot(1:10)")

# callback function
eval.txt <- function() eval(str2lang(tclvalue(tkget(txt.w, "0.0", "end"))))
tkpack(but.w <- tkbutton(tt, text = "Submit", command = eval.txt))

## Try pressing the button, edit the text and when finished:

tkdestroy(tt)

## End(Not run)
```

TkWidgets *Tk widgets*

Description

Create Tk widgets and associated R objects.

Usage

```
tkwidget(parent, type, ...)

tkbutton(parent, ...)
tkcanvas(parent, ...)
tkcheckbutton(parent, ...)
tkentry(parent, ...)
ttkentry(parent, ...)
tkframe(parent, ...)
tklabel(parent, ...)
tklistbox(parent, ...)
tkmenu(parent, ...)
tkmenubutton(parent, ...)
tkmessage(parent, ...)
tkradiobutton(parent, ...)
tkscale(parent, ...)
tkscrollbar(parent, ...)
tktext(parent, ...)
tktoplevel(parent = .TkRoot, ...)

ttkbutton(parent, ...)
ttkcheckbutton(parent, ...)
ttkcombobox(parent, ...)
ttkframe(parent, ...)
ttklabel(parent, ...)
ttklabelframe(parent, ...)
ttkmenubutton(parent, ...)
ttknotebook(parent, ...)
ttkpanedwindow(parent, ...)
ttkprogressbar(parent, ...)
ttkradiobutton(parent, ...)
ttkscale(parent, ...)
ttkscrollbar(parent, ...)
ttkseparator(parent, ...)
ttksizegrip(parent, ...)
ttkspinbox(parent, ...)
ttktreeview(parent, ...)
```

Arguments

parent	Parent of widget window.
type	string describing the type of widget desired.
...	handled via .Tcl.args.

Details

These functions create Tk widgets. tkwidget creates a widget of a given type, the others simply call tkwidget with the respective type argument.

The functions starting ttk are for the themed widget set for Tk 8.5 or later. A tutorial can be found at https://tkdocs.com/.

It is not possible to describe the widgets and their arguments in full. Please refer to the Tcl/Tk documentation.

See Also

TclInterface, TkCommands, TkWidgetcmds

Examples

```
## Not run:
## These cannot be run by examples() but should be OK when pasted
## into an interactive R session with the tcltk package loaded

tt <- tktoplevel()
label.widget <- tklabel(tt, text = "Hello, World!")
button.widget <- tkbutton(tt, text = "Push",
                          command = function()cat("OW!\n"))
tkpack(label.widget, button.widget) # geometry manager
                                    # see Tk-commands

## Push the button and then...

tkdestroy(tt)

## test for themed widgets
if(as.character(tcl("info", "tclversion")) >= "8.5") {
  # make use of themed widgets
  # list themes
  themes <- as.character(tcl("ttk::style", "theme", "names"))
  themes
  # select a theme -- for pre-XP windows
  #   tcl("ttk::style", "theme", "use", "winnative")
  tcl("ttk::style", "theme", "use", themes[1])
} else {
  # use Tk 8.0 widgets
}

## End(Not run)
```

tk_choose.dir *Choose a Folder Interactively*

Description

Use a Tk widget to choose a directory interactively.

Usage

```
tk_choose.dir(default = "", caption = "Select directory")
```

Arguments

default	which directory to show initially.
caption	the caption on the selection dialog.

Value

A length-one character vector, character NA if 'Cancel' was selected.

See Also

tk_choose.files

Examples

```
if (interactive()) tk_choose.dir(getwd(), "Choose a suitable folder")
```

tk_choose.files *Choose a List of Files Interactively*

Description

Use a Tk file dialog to choose a list of zero or more files interactively.

Usage

```
tk_choose.files(default = "", caption = "Select files",
                multi = TRUE, filters = NULL, index = 1)
```

Arguments

default	which filename to show initially.
caption	the caption on the file selection dialog.
multi	whether to allow multiple files to be selected.
filters	two-column character matrix of filename filters.
index	unused.

Details

Unlike file.choose, tk_choose.files will always attempt to return a character vector giving a list of files. If the user cancels the dialog, then zero files are returned, whereas file.choose would signal an error.

The format of filters can be seen from the example. File patterns are specified via extensions, with "*" meaning any file, and "" any file without an extension (a filename not containing a period). (Other forms may work on specific platforms.) Note that the way to have multiple extensions for one file type is to have multiple rows with the same name in the first column, and that whether the extensions are named in file chooser widget is platform-specific. **The format may change before release.**

Value

A character vector giving zero or more file paths.

Note

A bug in Tk 8.5.0–8.5.4 prevented multiple selections being used.

See Also

file.choose, tk_choose.dir

Examples

```
Filters <- matrix(c("R code", ".R", "R code", ".s",
                    "Text", ".txt", "All files", "*"),
                  4, 2, byrow = TRUE)
Filters
if(interactive()) tk_choose.files(filter = Filters)
```

---

tk_messageBox *Tk Message Box*

Description

An implementation of a generic message box using Tk.

Usage

```
tk_messageBox(type = c("ok", "okcancel", "yesno", "yesnocancel",
                       "retrycancel", "abortretryignore"),
              message, caption = "", default = "", ...)
```

Arguments

type	character. The type of dialog box. It will have the buttons implied by its name. Can be abbreviated.
message	character. The information field of the dialog box.
caption	the caption on the widget displayed.
default	character. The name of the button to be used as the default.
...	additional named arguments to be passed to the Tk function of this name. An example is icon = "warning".

Value

A character string giving the name of the button pressed.

See Also

tkmessageBox for a 'raw' interface.

---

tk_select.list *Select Items from a List*

Description

Select item(s) from a character vector using a Tk listbox.

Usage

```
tk_select.list(choices, preselect = NULL, multiple = FALSE,
               title = NULL)
```

Arguments

choices	a character vector of items.
preselect	a character vector, or NULL. If non-null and if the string(s) appear in the list, the item(s) are selected initially.
multiple	logical: can more than one item be selected?
title	optional character string for window title, or NULL for no title.

Details

This is a version of select.list implemented as a Tk list box plus OK and Cancel buttons. There will be a scrollbar if the list is too long to fit comfortably on the screen.

The dialog box is *modal*, so a selection must be made or cancelled before the R session can proceed. Double-clicking on an item is equivalent to selecting it and then clicking OK.

If Tk is version 8.5 or later, themed widgets will be used.

Value

A character vector of selected items. If `multiple` is false and no item was selected (or `Cancel` was used), `""` is returned. If `multiple` is true and no item was selected (or `Cancel` was used) then a character vector of length 0 is returned.

See Also

`select.list` (a text version except on Windows and the macOS GUI), `menu` (whose `graphics = TRUE` mode uses this on most Unix-alikes).

Chapter 13

The `tools` package

`tools-package` *Tools for Package Development*

Description

Tools for package development, administration and documentation.

Details

This package contains tools for manipulating R packages and their documentation.

For a complete list of functions, use `library(help = "tools")`.

Author(s)

Kurt Hornik and Friedrich Leisch

Maintainer: R Core Team <R-core@r-project.org>

`.print.via.format` *Printing Utilities*

Description

`.print.via.format` is a "prototype" `print()` method, useful, at least as a start, by a simple

```
print.<myS3class>  <-  .print.via.format
```

Usage

```
.print.via.format(x, ...)
```

Arguments

x	object to be printed.
...	optional further arguments, passed to format.

Value

x, invisibly (by invisible()), as print methods should.

See Also

The print generic; its default method print.default (used for many basic implicit classes such as "numeric", "character" and arrays of them, lists etc).

Examples

```
## The function is simply defined as
 function (x, ...) {
    writeLines(format(x, ...))
    invisible(x)
 }

## is used for simple print methods in R, and as prototype for new methods.
```

---

add_datalist *Add a 'datalist' File to a Source Package*

Description

The data() command with no arguments lists all the datasets available via data in attached packages, and to do so a per-package list is installed. Creating that list at install time can be slow for packages with huge datasets, and can be expedited by a supplying 'data/datalist' file.

Usage

```
add_datalist(pkgpath, force = FALSE, small.size = 1024^2)
```

Arguments

pkgpath	The path to a (source) package.
force	logical: can an existing 'data/datalist' file be over-written?
small.size	number: a 'data/datalist' file is created only if the total size of the data files is larger than small.size bytes.

Details

R CMD build will call this function to add a data list to packages with 1MB or more of file in the 'data' directory.

It can also be also helpful to give a 'data/datalist' file in packages whose datasets have many dependencies, including loading the packages itself (and maybe others).

See Also

data.

The 'Writing R Extensions' manual.

---

assertCondition *Asserting Error Conditions*

Description

When testing code, it is not sufficient to check that results are correct, but also that errors or warnings are signalled in appropriate situations. The functions described here provide a convenient facility for doing so. The three functions check that evaluating the supplied expression produces an error, a warning or one of a specified list of conditions, respectively. If the assertion fails, an error is signalled.

Usage

```
assertError(expr, classes = "error", verbose = FALSE)
assertWarning(expr, classes = "warning", verbose = FALSE)
assertCondition(expr, ..., .exprString = , verbose = FALSE)
```

Arguments

expr	an unevaluated R expression which will be evaluated via tryCatch(expr, ..).
classes, ...	character strings corresponding to the classes of the conditions that would satisfy the assertion; e.g., "error" or "warning". If none are specified, any condition will satisfy the assertion. See the details section.
.exprString	The string to be printed corresponding to expr. By default, the actual expr will be deparsed. Will be omitted if the function is supplied with the actual expression to be tested. If assertCondition() is called from another function, with the actual expression passed as an argument to that function, supply the deparsed version.
verbose	If TRUE, a message is printed when the condition is satisfied.

Details

`assertCondition()` uses the general condition mechanism to check all the conditions generated in evaluating `expr`. The occurrence of any of the supplied condition classes among these satisfies the assertion regardless of what other conditions may be signalled.

`assertError()` is a convenience function for asserting errors; it calls `assertCondition()`.

`assertWarning()` asserts that a warning will be signalled, but *not* an error, whereas `assertCondition(expr, "warning")` will be satisfied even if an error follows the warning. See the examples.

Value

If the assertion is satisfied, a list of all the condition objects signalled is returned, invisibly. See `conditionMessage` for the interpretation of these objects. Note that *all* conditions signalled during the evaluation are returned, whether or not they were among the requirements.

Author(s)

John Chambers and Martin Maechler

See Also

`stop`, `warning`; `signalCondition`, `tryCatch`.

Examples

```
  assertError(sqrt("abc"))
  assertWarning(matrix(1:8, 4,3))

  assertCondition( ""-1 ) # ok, any condition would satisfy this

try( assertCondition(sqrt(2), "warning") )
## .. Failed to get warning in evaluating sqrt(2)
     assertCondition(sqrt("abc"), "error")   # ok
try( assertCondition(sqrt("abc"), "warning") )# -> error: had no warning
     assertCondition(sqrt("abc"), "error")
  ## identical to assertError() call above

assertCondition(matrix(1:5, 2,3), "warning")
try( assertCondition(matrix(1:8, 4,3), "error") )
## .. Failed to get expected error ....

## either warning or worse:
assertCondition(matrix(1:8, 4,3), "error","warning") # OK
assertCondition(matrix(1:8, 4, 3), "warning") # OK

## when both are signalled:
ff <- function() { warning("my warning"); stop("my error") }
    assertCondition(ff(), "warning")
## but assertWarning does not allow an error to follow
try(assertWarning(ff()))
```

```
    assertCondition(ff(), "error")          # ok
assertCondition(ff(), "error", "warning") # ok (quietly, catching warning)

## assert that assertC..() does not assert [and use *one* argument only]
assertCondition( assertCondition(sqrt( 2   ), "warning") )
assertCondition( assertCondition(sqrt("abc"), "warning"), "error")
assertCondition( assertCondition(matrix(1:8, 4,3), "error"),
                "error")
```

`bibstyle` *Select or Define a Bibliography Style*

Description

This function defines and registers styles for rendering `bibentry` objects into 'Rd' format, for later conversion to text, HTML, etc.

Usage

```
bibstyle(style, envir, ..., .init = FALSE, .default = TRUE)
getBibstyle(all = FALSE)
```

Arguments

`style`	A character string naming the style.
`envir`	(optional) An environment holding the functions to implement the style.
`...`	Named arguments to add to the environment.
`.init`	Whether to initialize the environment from the default style `"JSS"`.
`.default`	Whether to set the specified style as the default style.
`all`	Whether to return the names of all registered styles.

Details

Rendering of `bibentry` objects may be done using routines modelled after those used by BibTeX. This function allows environments to be created and manipulated to contain those routines.

There are two ways to create a new style environment. The easiest is to set `.init = TRUE`, in which case the environment will be initialized with a copy of the default `"JSS"` environment. (This style is modelled after the '`jss.bst`' style used by the *Journal of Statistical Software*.) Alternatively, the `envir` argument can be used to specify a completely new style environment.

To find the name of the default style, use `getBibstyle()`. To retrieve an existing style without setting it as the default, use `bibstyle(style, .default = FALSE)`. To modify an existing style, specify `style` and some named entries via `...`. (Modifying the default `"JSS"` style is discouraged.) Setting `style` to `NULL` or leaving it missing will retrieve the default style, but modifications will not be allowed.

At a minimum, the environment should contain routines to render each of the 12 types of bibliographic entry supported by bibentry as well as several other routines described below. The former must be named formatArticle, formatBook, formatInbook, formatIncollection, formatInProceedings, formatManual, formatMastersthesis, formatMisc, formatPhdthesis, formatProceedings, formatTechreport and formatUnpublished. Each of these takes one argument, a single unclass'ed entry from the bibentry vector passed to the renderer, and should produce a single element character vector (possibly containing newlines).

The other routines are as follows. sortKeys, a function to produce a sort key to sort the entries, is passed the original bibentry vector and should produce a sortable vector of the same length to define the sort order. Finally, the optional function cite should have the same argument list as utils::cite, and should produce a citation to be used in text.

The format method for "bibentry" objects adds a field named ".index" to each entry after sorting and before formatting. This is a 1-based index within the complete object that can be used in styles that require numbering. Although the "JSS" style doesn't use numbers, it includes a fmtPrefix() stub function that may be used to display them. See the example below.

Value

bibstyle returns the environment which has been selected or created.

getBibstyle returns the name of the default style, or all style names.

Author(s)

Duncan Murdoch

See Also

bibentry

Examples

```
refs <-
c(bibentry(bibtype = "manual",
    title = "R: A Language and Environment for Statistical Computing",
    author = person("R Core Team"),
    organization = "R Foundation for Statistical Computing",
    address = "Vienna, Austria",
    year = 2013,
    url = "https://www.R-project.org"),
  bibentry(bibtype = "article",
    author = c(person(c("George", "E.", "P."), "Box"),
               person(c("David",  "R."),      "Cox")),
    year = 1964,
    title = "An Analysis of Transformations",
    journal = "Journal of the Royal Statistical Society, Series B",
    volume = 26, number = 2, pages = "211--243",
    doi = "10.1111/j.2517-6161.1964.tb00553.x"))

bibstyle("unsorted", sortKeys = function(refs) seq_along(refs),
```

```
    fmtPrefix = function(paper) paste0("[", paper$.index, "]"),
       .init = TRUE)
print(refs, .bibstyle = "unsorted")
```

---

buildVignette *Build One Vignette*

Description

Run Sweave (or other custom weave function), texi2pdf, and/or Stangle (or other custom tangle function) on one vignette.

This is the workhorse of R CMD Sweave.

Usage

```
buildVignette(file, dir = ".", weave = TRUE, latex = TRUE, tangle = TRUE,
              quiet = TRUE, clean = TRUE, keep = character(),
              engine = NULL, buildPkg = NULL, encoding, ...)
```

Arguments

file	character; the vignette source file.
dir	character; the working directory in which the intermediate and output files will be produced.
weave	logical; should weave be run?
latex	logical; should texi2pdf be run if weaving produces a '.tex' file?
tangle	logical; should tangle be run?
quiet	logical; run in quiet mode?
clean	logical; whether to remove some newly created, often intermediate, files. See details below.
keep	a list of file names to keep in any case when cleaning. Note that "target" files are kept anyway.
engine	NULL or character; name of vignette engine to use. Overrides any '\VignetteEngine{}' markup in the vignette.
buildPkg	NULL or a character vector; optional packages in which to find the vignette engine.
encoding	the encoding to assume for the file. If not specified, it will be read if possible from the file's contents. Note that if the vignette is part of a package, buildVignettes reads the package's encoding from the 'DESCRIPTION' file but this function does not.
...	Additional arguments passed to weave and tangle.

Details

This function determines the vignette engine for the vignette (default `utils::Sweave`), then weaves and/or tangles the vignette using that engine. Finally, if `clean` is `TRUE`, newly created intermediate files (non "targets", where these depend on the engine, etc, and not any in `keep`) will be deleted. If `clean` is `NA`, and `weave` is true, newly created intermediate output files (e.g., '`.tex`') will not be deleted even if a '`.pdf`' file has been produced from them.

If `buildPkg` is specified, those packages will be loaded before the vignette is processed and will be used as the default packages in the search for a vignette engine, but an explicitly specified package in the vignette source (e.g., using '`\VignetteEngine{utils::Sweave}`' to specify the `Sweave` engine in the **utils** package) will override it. In contrast, if the `engine` argument is given, it will override the vignette source.

Value

A character vector naming the files that have been produced.

Author(s)

Henrik Bengtsson and Duncan Murdoch

See Also

`buildVignettes` for building all vignettes in a package.

---

buildVignettes *List and Build Package Vignettes*

Description

Run `Sweave` (or other custom weave function) and `texi2pdf` on all vignettes of a package, or list the vignettes.

Usage

```
buildVignettes(package, dir, lib.loc = NULL, quiet = TRUE,
               clean = TRUE, tangle = FALSE, skip = NULL,
               ser_elibs = NULL)

pkgVignettes(package, dir, subdirs = NULL, lib.loc = NULL,
      output = FALSE, source = FALSE, check = FALSE)
```

Arguments

package	a character string naming an installed package. If given, vignette source files are by default looked for in subdirectory 'doc'.
dir	a character string specifying the path to a package's root source directory. If given, vignette source files are by default looked for in subdirectory 'vignettes'.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
quiet	logical. Weave and run texi2pdf in quiet mode.
clean	Remove all files generated by the build, even if there were copies there before.
tangle	logical. Do tangling as well as weaving.
skip	a character vector of names of vignettes (without file extension, matching the names returned from pkgVignettes) which should be skipped, or TRUE to skip those with unavailable '\VignetteDepends' (from vignetteInfo).
ser_elibs	For use from R CMD check.
subdirs	a character vector of subdirectories of dir in which to look for vignettes. The first which exists is used. Defaults to "doc" if package is supplied, otherwise "vignettes".
output	logical indicating if the output filenames for each vignette should be returned (in component outputs).
source	logical indicating if the *tangled* output filenames for each vignette should be returned (in component sources).
check	logical. If TRUE, check whether all files that have vignette-like filenames have an identifiable vignette engine. This may be a false positive if a file is not a vignette but has a filename matching a pattern defined by one of the vignette engines.

Details

buildVignettes is used by R CMD build and R CMD check to (re-)build vignette outputs from their sources.

As from R 3.4.1, both of these functions ignore files that are listed in the '.Rbuildignore' file in dir.

Value

buildVignettes is called for its side effect of creating the outputs of all vignettes, and if tangle = TRUE, extracting the R code.

pkgVignettes returns an object of class "pkgVignettes" if a vignette directory is found, otherwise NULL.

Examples

```
gVigns <- pkgVignettes("grid")
str(gVigns)
```

charsets *Conversion Tables between Character Sets*

Description

charset_to_Unicode is a matrix of Unicode code points with columns for the common 8-bit encodings.

Adobe_glyphs is a data frame which gives Adobe glyph names for Unicode code points. It has two character columns, "adobe" and "unicode" (a 4-digit hex representation).

Usage

```
charset_to_Unicode

Adobe_glyphs
```

Details

charset_to_Unicode is an integer matrix of class c("noquote", "hexmode") so prints in hexadecimal. The mappings are those used by libiconv: there are differences in the way quotes and minus/hyphen are mapped between sources (and the postscript encoding files use a different mapping).

Adobe_glyphs includes all the Adobe glyph names which correspond to single Unicode characters. It is sorted by Unicode code point and within a point alphabetically on the glyph (there can be more than one name for a Unicode code point). The data are in the file '*R_HOME*/share/encodings/Adobe_glyphlist'.

Examples

```
## find Adobe names for ISOLatin2 chars.
latin2 <- charset_to_Unicode[, "ISOLatin2"]
aUnicode <- as.hexmode(paste0("0x", Adobe_glyphs$unicode))
keep <- aUnicode %in% latin2
aUnicode <- aUnicode[keep]
aAdobe <- Adobe_glyphs[keep, 1]
## first match
aLatin2 <- aAdobe[match(latin2, aUnicode)]
## all matches
bLatin2 <- lapply(1:256, function(x) aAdobe[aUnicode == latin2[x]])
format(bLatin2, justify = "none")
```

checkFF *Check Foreign Function Calls*

Description

Performs checks on calls to compiled code from R code. Currently only checks whether the interface functions such as `.C` and `.Fortran` are called with a "`NativeSymbolInfo`" first argument or with argument `PACKAGE` specified, which is highly recommended to avoid name clashes in foreign function calls.

Usage

```
checkFF(package, dir, file, lib.loc = NULL,
        registration = FALSE, check_DUP = FALSE,
        verbose = getOption("verbose"))
```

Arguments

`package`	a character string naming an installed package. If given, the installed R code of the package is checked.
`dir`	a character string specifying the path to a package's root source directory. This should contain the subdirectory '`R`' (for R code). Only used if `package` is not given.
`file`	the name of a file containing R code to be checked. Used if neither `package` nor `dir` are given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.
`registration`	a logical. If `TRUE`, checks the registration information on the call (if available).
`check_DUP`	a logical. If `TRUE`, `.C` and `.Fortran` calls with `DUP = FALSE` are reported.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed (and the result is returned invisibly).

Details

Note that we can only check if the `name` argument is a symbol or a character string, not what class of object the symbol resolves to at run-time.

If the package has a namespace which contains a `useDynLib` directive, calls in top-level functions in the package are not reported as their symbols will be preferentially looked up in the DLL named in the first `useDynLib` directive.

This checks that calls with `PACKAGE` specified are to the same package, and reports separately those which are in base packages and those which are in other packages (and if those packages are specified in the '`DESCRIPTION`' file).

Value

An object of class "checkFF".

There are format and print methods to display the information contained in such objects.

See Also

.C, .Fortran; Foreign.

Examples

```
 # order is pretty much random
checkFF(package = "stats", verbose = TRUE)
```

---

checkMD5sums *Check and Create MD5 Checksum Files*

Description

checkMD5sums checks the files against a file 'MD5'.

Usage

```
checkMD5sums(package, dir)
```

Arguments

package	the name of an installed package
dir	the path to the top-level directory of an installed package.

Details

The file 'MD5' which is created is in a format which can be checked by md5sum -c MD5 if a suitable command-line version of md5sum is available. (For Windows, one is supplied in the bundle at https://cran.r-project.org/bin/windows/Rtools/.)

If dir is missing, an installed package of name package is searched for.

The private function tools:::.installMD5sums is used to create MD5 files in the Windows build.

Value

checkMD5sums returns a logical, NA if there is no 'MD5' file to be checked.

See Also

md5sum

checkPoFiles *Check Translation Files for Inconsistent Format Strings*

Description

These functions compare formats embedded in English messages with translated strings to check for consistency. `checkPoFile` checks one file, while `checkPoFiles` checks all files for specified or all languages.

Usage

```
checkPoFile(f, strictPlural = FALSE)
checkPoFiles(language, dir = ".")
```

Arguments

`f`	a character string giving a single filepath.
`strictPlural`	whether to compare formats of singular and plural forms in a strict way.
`language`	a character string giving a language code, or a 'regexp' (regular expression), to match languages in `dir`. Use `""` to denote all languages in the `dir` path.
`dir`	a path to a directory in which to check files.

Details

Part of R's internationalization depends on translations of messages in '`.po`' files. In these files an 'English' message taken from the R sources is followed by a translation into another language. Many of these messages are format strings for C or R `sprintf` and related functions. In these cases, the translation must give a compatible format or an error will be generated when the message is displayed.

The rules for compatibility differ between C and R in several ways. C supports several conversions not supported by R, namely `c`, `u`, `p`, `n`. It is allowed in C's `sprintf()` function to have more arguments than are needed by the format string, but in R the counts must match exactly. R requires types of arguments to match, whereas C will do the display whether it makes sense or not.

These functions compromise on the testing as follows. The additional formats allowed in C are accepted, and all differences in argument type or count are reported. As a consequence some reported differences are not errors.

If the `strictPlural` argument is `TRUE`, then argument lists must agree exactly between singular and plural forms of messages; if `FALSE`, then translations only need to match one or the other of the two forms. When `checkPoFiles` calls `checkPoFile`, the `strictPlural` argument is set to `TRUE` for files with names starting '`R-`', and to `FALSE` otherwise.

Items marked as 'fuzzy' in the '`.po`' file are not processed (as they are ignored by the message compiler).

If a difference is found, the translated string is checked for variant percent signs (e.g., the wide percent sign `"\uFF05"`). Such signs will not be recognized as format specifiers, and are likely to be errors.

Value

Both functions return an object of S3 class "check_po_files". A print method is defined for this class to display a report on the differences.

Author(s)

Duncan Murdoch

References

See the GNU gettext manual for the '.po' file format:
https://www.gnu.org/software/gettext/manual/gettext.html.

See Also

update_pkg_po() which calls checkPoFile(); xgettext, sprintf.

Examples

```
## Not run:
checkPoFiles("de", "/path/to/R/src/directory")

## End(Not run)
```

---

checkRd *Check an Rd Object*

Description

Check an help file or the output of the parse_Rd function.

Usage

```
checkRd(Rd, defines = .Platform$OS.type, stages = "render",
        unknownOK = TRUE, listOK = TRUE, ..., def_enc = FALSE)
```

Arguments

Rd	a filename or Rd object to use as input.
defines	string(s) to use in #ifdef tests.
stages	at which stage ("build", "install", or "render") should \Sexpr macros be executed? See the notes below.
unknownOK	unrecognized macros are treated as errors if FALSE, otherwise warnings.
listOK	unnecessary non-empty braces (e.g., around text, not as an argument) are treated as errors if FALSE, otherwise warnings ("Lost braces").

...	additional parameters to pass to parse_Rd when Rd is a filename. One that is often useful is encoding.
def_enc	logical: has the package declared an encoding, so tests for non-ASCII text are suppressed?

Details

checkRd performs consistency checks on an Rd file, confirming that required sections are present, etc.

It accepts a filename for an Rd file, and will use parse_Rd to parse it before applying the checks. If so, warnings from parse_Rd are collected, together with those from the internal function prepare_Rd, which does the #ifdef and \Sexpr processing, drops sections that would not be rendered or are duplicated (and should not be) and removes empty sections.

An Rd object is passed through prepare_Rd, but it may already have been (and installed Rd objects have).

Warnings are given a 'level': those from prepare_Rd have level 0. These include

\Sexpr expects R code; found ...
Unprocessed '*stage*' macro from *stage*-stage \Sexpr
All text must be in a section
Only one *tag name* section is allowed: the first will be used
docType *type* is unrecognized
Section *name* is unrecognized and will be dropped
Dropping empty section *name*

checkRd itself can show

7 Tag *tag name* not recognized
7 Unrecognized format: ...
7 \tabular format must be simple text
7 Unrecognized \tabular format: ...
7 Only *n* columns allowed in this table
7 Tag *tag name* is invalid in a *block name* block
7 \method not valid outside a code block
7 Tag \method is only valid in \usage
7 Tag \dontrun is only valid in \examples
7 Invalid email address: ...
7 Invalid URL: ...
5 \name should not contain !, | or @
5 \item in *block name* must have non-empty label
3 Empty section *tag name*
-1 \name should only contain printable ASCII characters
-1 Non-ASCII contents without declared encoding
-1 Non-ASCII contents in second part of \enc
-1 Escaped LaTeX specials: ...
-1|-3 Lost braces ...

-3 Tag \ldots is invalid in a code block
-5 \title should not end in a period

and variations with \method replaced by \S3method or \S4method, \dontrun replaced by \donttest, \dontdiff or \dontshow, and \title replaced by \section or \subsection name. "Lost braces" are uprated to warning level -1 when they match common markup mistakes, e.g., 'code{text}' rendered as 'codetext' due to the missing backslash escape for the macro name.

Note that both prepare_Rd and checkRd have tests for an empty section: that in checkRd is stricter (essentially that nothing is output).

Value

This may fail through an R error, but otherwise warnings are collected as returned as an object of class "checkRd", a character vector of messages. This class has a print method which only prints unique messages, and has argument minlevel that can be used to select only more serious messages. (This is set to -1 in R CMD check.)

Possible fatal errors are those from !unknownOK or !listOK, from invalid \if or \ifelse conditions, from running the parser (e.g., a non-existent file, unclosed quoted string, non-ASCII input without a specified encoding, an invalid value for an \Sexpr option), or from prepare_Rd (multiple \Rdversion declarations, invalid \encoding or \docType or \name sections, and missing or duplicate \name or \title sections), including errors from parsing/running code from \Sexpr macros (if covered by stages).

Author(s)

Duncan Murdoch, Brian Ripley

See Also

parse_Rd, Rd2HTML.

Examples

```
## parsed Rd from the installed version of _this_ help file
rd <- Rd_db("tools")[["checkRd.Rd"]]
rd
stopifnot(length(checkRd(rd)) == 0)  # there should be no issues

## make up \tabular issues
bad <- r"(\name{bad}\title{bad}\description{\tabular{p}{1 \tab 2}})"
(res <- checkRd(parse_Rd(textConnection(bad))))
stopifnot(length(res) > 0)
```

checkRdaFiles *Report on Details of Saved Images or Re-saves them*

Description

This reports for each of the files produced by save the size, if it was saved in ASCII or XDR binary format, and if it was compressed (and if so in what format).

Usually such files have extension '.rda' or '.RData', hence the name of the function.

Usage

```
checkRdaFiles(paths)
resaveRdaFiles(paths, compress = c("auto", "gzip", "bzip2", "xz"),
               compression_level, version = NULL)
```

Arguments

paths	A character vector of paths to save files. If this specifies a single directory, it is taken to refer to all '.rda' and '.RData' files in that directory.
compress, compression_level	Type and level of compression: see save. Values of compress can be abbreviated.
version	The format to be used when re-saving: see save.

Details

compress = "auto" asks R to choose the compression and ignores compression_level. It will try "gzip", "bzip2" and if the "gzip" compressed size is over 10Kb, "xz" and choose the smallest compressed file (but with a 10% bias towards "gzip"). This can be slow.

For back-compatibility, version = NULL is interpreted to mean version 2: however version-3 files will only be saved as version 3.

Value

For checkRdaFiles, a data frame with rows names paths and columns

size	numeric: file size in bytes, NA if the file does not exist.
ASCII	logical: true for save(ASCII = TRUE), NA if the format is not that of an R save file.
compress	character: type of compression. One of "gzip", "bzip2", "xz", "none" or "unknown" (which means that if this is an R save file it is from a later version of R).
version	integer: positive with the version(s) of the save(), see there on which versions have been default in which versions of R, and NA for non-Rda files.

Examples

```
## Not run:
## from a package top-level source directory
paths <- sort(Sys.glob(c("data/*.rda", "data/*.RData")))
(res <- checkRdaFiles(paths))
## pick out some that may need attention
bad <- is.na(res$ASCII) | res$ASCII | (res$size > 1e4 & res$compress == "none")
res[bad, ]

## End(Not run)
```

---

checkTnF *Check R Packages or Code for T/F*

Description

Checks the specified R package or code file for occurrences of T or F, and gathers the expression containing these. This is useful as in R T and F are just variables which are set to the logicals TRUE and FALSE by default, but are not reserved words and hence can be overwritten by the user. Hence, one should always use TRUE and FALSE for the logicals.

Usage

```
checkTnF(package, dir, file, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package. If given, the installed R code and the examples in the documentation files of the package are checked. R code installed as an image file cannot be checked.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'R' (for R code), and should also contain 'man' (for documentation). Only used if package is not given. If used, the R code files and the examples in the documentation files are checked.
file	the name of a file containing R code to be checked. Used if neither package nor dir are given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

Value

An object of class "checkTnF" which is a list containing, for each file where occurrences of T or F were found, a list with the expressions containing these occurrences. The names of the list are the corresponding file names.

There is a print method for nicely displaying the information contained in such objects.

checkVignettes *Check Package Vignettes*

Description

Check all vignettes of a package by running Sweave (or other custom weave function) and/or Stangle (or other custom tangle function) on them. All R source code files found after the tangling step are sourceed to check whether all code can be executed without errors.

Usage

```
checkVignettes(package, dir, lib.loc = NULL,
               tangle = TRUE, weave = TRUE, latex = FALSE,
               workdir = c("tmp", "src", "cur"),
               keepfiles = FALSE)
```

Arguments

package	a character string naming an installed package. If given, vignette source files are looked for in subdirectory 'doc'.
dir	a character string specifying the path to a package's root source directory. If given, vignette source files are looked for in subdirectory 'vignettes'.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
tangle	Perform a tangle and source the extracted code?
weave	Perform a weave?
latex	logical: if weave and latex are TRUE and there is no 'Makefile' in the vignettes directory, run the intermediate '.tex' outputs from weaving through texi2pdf.
workdir	Directory used as working directory while checking the vignettes. If "tmp" then a temporary directory is created, this is the default. If "src" then the directory containing the vignettes itself is used, if "cur" then the current working directory of R is used.
keepfiles	Delete files in the temporary directory? This option is ignored when workdir != "tmp".

Details

This function first uses pkgVignettes to find the package vignettes, and in particular their vignette engines (see vignetteEngine).

If tangle is true, it then runs Stangle (or other custom tangle function provided by the engine) to produce (one or more) R code files from each vignette, then sources each code file in turn.

If weave is true, the vignettes are run through Sweave (or other custom weave function provided by the engine). If latex is also true and there is no 'Makefile' in the vignettes directory, texi2pdf is run on the intermediate '.tex' files from weaving for those vignettes which did not give errors in the previous steps.

Value

An object of class "checkVignettes", which is a list with the error messages found during the tangle, source, weave and latex steps. There is a print method for displaying the information contained in such objects.

---

check_packages_in_dir *Check Source Packages and Their Reverse Dependencies*

Description

Check source packages in a given directory, optionally with their reverse dependencies.

Usage

```
check_packages_in_dir(dir,
                      pfiles = Sys.glob("*.tar.gz"),
                      check_args = character(),
                      check_args_db = list(),
                      reverse = NULL,
                      check_env = character(),
                      xvfb = FALSE,
                      Ncpus = getOption("Ncpus", 1L),
                      clean = TRUE,
                      install_args = list(),
                      parallel_args = list(),
                      ...)

summarize_check_packages_in_dir_results(dir, all = TRUE,
                                        full = FALSE, ...)
summarize_check_packages_in_dir_timings(dir, all = FALSE,
                                        full = FALSE)
summarize_check_packages_in_dir_depends(dir, all = FALSE,
                                        which = c("Depends",
                                                  "Imports",
                                                  "LinkingTo"))

check_packages_in_dir_changes(dir, old,
                              outputs = FALSE, sources = FALSE, ...)
check_packages_in_dir_details(dir, logs = NULL, drop_ok = TRUE, ...)
```

Arguments

dir	a character string giving the path to the directory with the source '.tar.gz' files to be checked.
pfiles	(optional) character vector of tarball files to be checked. Useful for choosing a subset of the '*.tar.gz' files in dir.

check_args	a character vector with arguments to be passed to R CMD check, or a list of length two of such character vectors to be used for checking packages and reverse dependencies, respectively.
check_args_db	a named list of character vectors with arguments to be passed to R CMD check, with names the respective package names.
reverse	a list with names partially matching "repos", "which", or "recursive", giving the repositories to use for locating reverse dependencies (a subset of getOption("repos"), the default), the types of reverse dependencies (default: c("Depends", "Imports", "LinkingTo"), with shorthands "most" and "all" as for package_dependencies), and indicating whether to also check reverse dependencies of reverse dependencies and so on (default: FALSE), or NULL (default), in which case no reverse dependencies are checked.
check_env	a character vector of name=value strings to set environment variables for checking, or a list of length two of such character vectors to be used for checking packages and reverse dependencies, respectively.
xvfb	a logical indicating whether to perform checking inside a virtual framebuffer X server (Unix only), or a character vector of Xvfb options for doing so.
Ncpus	the number of parallel processes to use for parallel installation and checking.
clean	a logical indicating whether to remove the downloaded reverse dependency sources.
install_args	list of arguments to be passed to underlying install.packages call.
parallel_args	list of arguments to be passed to underlying calls of parLapply (on Windows) or mclapply (on other OS).
...	passed to readLines, e.g. for reading log files produced in a different encoding; currently not used by check_packages_in_dir.
all	a logical indicating whether to also summarize the reverse dependencies checked.
full	a logical indicating whether to also give details for checks with non-ok results, or summarize check example timings (if available).
which	see package_dependencies.
old	a character string giving the path to the directory of a previous check_packages_in_dir run.
outputs	a logical indicating whether to analyze changes in the outputs of the checks performed, or only (default) the status of the checks.
sources	a logical indicating whether to also investigate the changes in the source files checked (default: FALSE).
logs	a character vector with the paths of '00check.log' to analyze. Only used if dir was not given.
drop_ok	a logical indicating whether to drop checks with 'ok' status, or a character vector with the 'ok' status tags to drop. The default corresponds to tags 'OK', 'NONE' and 'SKIPPED'.

Details

check_packages_in_dir allows to conveniently check source package '.tar.gz' files in the given directory dir, along with their reverse dependencies as controlled by reverse.

The "which" component of reverse can also be a list, in which case reverse dependencies are obtained for each element of the list and the corresponding element of the "recursive" component of reverse (which is recycled as needed).

If needed, the source '.tar.gz' files of the reverse dependencies to be checked as well are downloaded into dir (and removed at the end if clean is true). Next, all packages (additionally) needed for checking are installed to the 'Library' subdirectory of dir. Then, all '.tar.gz' files are checked using the given arguments and environment variables, with outputs and messages to files in the 'Outputs' subdirectory of dir. The '*.Rcheck' directories with the check results of the reverse dependencies are renamed by prefixing their base names with 'rdepends_'.

Results and timings can conveniently be summarized using summarize_check_packages_in_dir_results and summarize_check_packages_in_dir_timings, respectively.

Installation and checking is performed in parallel if Ncpus is greater than one: this will use mclapply on Unix and parLapply on Windows.

check_packages_in_dir returns an object inheriting from class "check_packages_in_dir" which has print and summary methods.

check_packages_in_dir_changes allows to analyze the effect of changing (some of) the sources. With dir and old the paths to the directories with the new and old sources, respectively, and the corresponding check results, possible changes in the check results can conveniently be analyzed as controlled via options outputs and sources. The changes object returned can be subscripted according to change in severity from the old to the new results by using one of "==", "!=", "<", "<=", ">" or ">=" as row index.

check_packages_in_dir_details analyzes check log files to obtain check details as a data frame which can be used for further processing, providing check name, status and output for every check performed and not dropped according to status tag (via variables Check, Status and Output, respectively).

Environment variable _R_CHECK_ELAPSED_TIMEOUT_ can be used to set a limit on the elapsed time of each check run. See the 'R Internals' manual for how the value is interpreted and for other environment variables which can be used for finer-grained control on timeouts within a check run.

Note

This functionality is still experimental: interfaces may change in future versions.

Examples

```
## Not run:
## Check packages in dir without reverse dependencies:
check_packages_in_dir(dir)
## Check packages in dir and their reverse dependencies using the
## defaults (all repositories in getOption("repos"), all "strong"
## reverse dependencies, no recursive reverse dependencies):
check_packages_in_dir(dir, reverse = list())
```

```
## Check packages in dir with their reverse dependencies from CRAN,
## using all strong reverse dependencies and reverse suggests:
check_packages_in_dir(dir,
                      reverse = list(repos = getOption("repos")["CRAN"],
                                     which = "most"))
## Check packages in dir with their reverse dependencies from CRAN,
## using '--as-cran' for the former but not the latter:
check_packages_in_dir(dir,
                      check_args = c("--as-cran", ""),
                      reverse = list(repos = getOption("repos")["CRAN"]))

## End(Not run)
```

---

codoc *Check Code/Documentation Consistency*

Description

Find inconsistencies between actual and documented 'structure' of R objects in a package. codoc compares names and optionally also corresponding positions and default values of the arguments of functions. codocClasses and codocData compare slot names of S4 classes and variable names of data sets, respectively.

Usage

```
codoc(package, dir, lib.loc = NULL,
      use.values = NULL, verbose = getOption("verbose"))
codocClasses(package, lib.loc = NULL)
codocData(package, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectories 'man' with R documentation sources (in Rd format) and 'R' with R code. Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
use.values	if FALSE, do not use function default values when comparing code and docs. Otherwise, compare *all* default values if TRUE, and only the ones documented in the usage otherwise (default).
verbose	a logical. If TRUE, additional diagnostics are printed.

Details

The purpose of codoc is to check whether the documented usage of function objects agrees with their formal arguments as defined in the R code. This is not always straightforward, in particular as the usage information for methods to generic functions often employs the name of the generic rather than the method.

The following algorithm is used. If an installed package is used, it is loaded (unless it is the **base** package), after possibly detaching an already loaded version of the package. Otherwise, if the sources are used, the R code files of the package are collected and sourced in a new environment. Then, the usage sections of the Rd files are extracted and parsed 'as much as possible' to give the formals documented. For interpreted functions in the code environment, the formals are compared between code and documentation according to the values of the argument `use.values`.

If a package has a namespace both exported and unexported objects are checked, as well as registered S3 methods. (In the unlikely event of differences the order is exported objects in the package, registered S3 methods and finally objects in the namespace and only the first found is checked.)

Currently, the R documentation format has no high-level markup for the basic 'structure' of classes and data sets (similar to the usage sections for function synopses). Variable names for data frames in documentation objects obtained by suitably editing 'templates' created by `prompt` are recognized by `codocData` and used provided that the documentation object is for a single data frame (i.e., only has one alias). `codocClasses` analogously handles slot names for classes in documentation objects obtained by editing shells created by `promptClass`.

Help files named '*pkgname*`-defunct.Rd`' for the appropriate *pkgname* are checked more loosely, as they may have undocumented arguments.

Value

`codoc` returns an object of class `"codoc"`. Currently, this is a list which, for each Rd object in the package where an inconsistency was found, contains an element with a list of the mismatches (which in turn are lists with elements `code` and `docs`, giving the corresponding arguments obtained from the function's code and documented usage).

`codocClasses` and `codocData` return objects of class `"codocClasses"` and `"codocData"`, respectively, with a structure similar to class `"codoc"`.

There are `print` methods for nicely displaying the information contained in such objects.

Note

The default for `use.values` has been changed from `FALSE` to `NULL`, for R versions 1.9.0 and later.

See Also

`undoc`, `QC`

compactPDF *Compact PDF Files*

Description

Re-save PDF files (especially vignettes) more compactly. Support function for `R CMD build --compact-vignettes`.

Usage

```
compactPDF(paths,
           qpdf = Sys.which(Sys.getenv("R_QPDF", "qpdf")),
           gs_cmd = Sys.getenv("R_GSCMD", ""),
           gs_quality = Sys.getenv("GS_QUALITY", "none"),
           gs_extras = character(),
           verbose = FALSE)

## S3 method for class 'compactPDF'
format(x, ratio = 0.9, diff = 1e4, ...)
```

Arguments

paths	A character vector of paths to PDF files, or a length-one character vector naming a directory, when all '.pdf' files in that directory will be used.
qpdf	Character string giving the path to the qpdf command. If empty, qpdf will not be used.
gs_cmd	Character string giving the path to the GhostScript executable, if that is to be used. On Windows this is the path to 'gswin32c.exe' or 'gswin64c.exe'. If "" (the default), the function will try to find a platform-specific path to GhostScript where required.
gs_quality	A character string indicating the quality required: the options are "none" (so GhostScript is not used), "printer" (300dpi), "ebook" (150dpi) and "screen" (72dpi). Can be abbreviated.
gs_extras	An optional character vector of further options to be passed to GhostScript.
verbose	logical or non-negative integer indicating if and how much of the compression utilities' output should be shown.
x	An object of class "compactPDF".
ratio, diff	Limits for reporting: files are only reported whose sizes are reduced both by a factor of ratio and by diff bytes.
...	Further arguments to be passed to or from other methods.

Details

This by default makes use of `qpdf`, available from https://qpdf.sourceforge.io/ (including as a Windows binary) and included with the CRAN macOS distribution of R. If `gs_cmd` is non-empty and `gs_quality != "none"`, GhostScript will used first, then `qpdf` if it is available. If `gs_quality != "none"` and `gs_cmd` is `""`, an attempt will be made to find a GhostScript executable.

`qpdf` and/or `gs_cmd` are run on all PDF files found, and those which are reduced in size by at least 10% and 10Kb are replaced.

The strategy of our use of `qpdf` is to (losslessly) compress both PDF streams and objects. GhostScript compresses streams and more (including downsampling and compressing embedded images) and consequently is much slower and may lose quality (but can also produce much smaller PDF files). However, quality `"ebook"` is perfectly adequate for screen viewing and printing on laser printers.

Where PDF files are changed they will become PDF version 1.5 files: these have been supported by Acrobat Reader since version 6 in 2003, so this is very unlikely to cause difficulties.

Stream compression is what most often has large gains. Most PDF documents are generated with object compression, but this does not seem to be the default for MiKTeX's `pdflatex`. For some PDF files (and especially package vignettes), using GhostScript can dramatically reduce the space taken by embedded images (often screenshots).

Where both GhostScript and `qpdf` are selected (when `gs_quality != "none"` and both executables are found), they are run in that order and the size reductions apply to the total compression achieved.

Value

An object of class `c("compactPDF", "data.frame")`. This has two columns, the old and new sizes in bytes for the files that were changed.

There are `format` and `print` methods: the latter passes `...` to the format method, so will accept `ratio` and `diff` arguments.

Note

The external tools used may change in future releases.

Frequently, updates to GhostScript have produced better compression (up to several times better), so if possible use the latest version available.

See Also

`resaveRdaFiles`.

For other tools to compact PDF files, see the 'Writing R Extensions' manual.

CRANtools *CRAN Package Repository Tools*

Description

Tools for obtaining information about current packages in the CRAN package repository, and their check status.

Usage

```
CRAN_package_db()

CRAN_check_results(flavors = NULL)
CRAN_check_details(flavors = NULL)
CRAN_check_issues()
summarize_CRAN_check_status(packages,
                            results = NULL,
                            details = NULL,
                            issues = NULL)
```

Arguments

packages	a character vector of package names.
flavors	a character vector of CRAN check flavor names, or NULL (default), corresponding to all available flavors.
results	the return value of CRAN_check_results() (default), or a subset of this.
details	the return value of CRAN_check_details() (default), or a subset of this.
issues	the return value of CRAN_check_issues() (default), or a subset of this.

Details

CRAN_package_db() returns a data frame with character columns containing most 'DESCRIPTION' metadata for the current packages in the CRAN package repository, including in particular the Description and Maintainer information not provided by utils::available.packages().

CRAN_check_results() returns a data frame with the basic CRAN package check results including timings, with columns Package, Flavor and Status giving the package name, check flavor, and overall check status, respectively.

CRAN_check_details() returns a data frame inheriting from class "check_details" (which has useful print and format methods) with details on the check results, providing check name, status and output for every non-OK check (*via* columns Check, Status and Output, respectively). Packages with all-OK checks are indicated via a * Check wildcard name and OK Status.

CRAN_check_issues() returns a character frame with additional check issues (including the memory-access check results made available from https://www.stats.ox.ac.uk/pub/bdr/memtests/) as a character frame with variables Package, Version, kind (an identifier for the issue) and href (a URL with information on the issue).

Value

See 'Details'. Note that the results are collated on CRAN: currently this is done in a locale which sorts aAbB ....

Which CRAN?

The main functions access a CRAN mirror specified by the environment variable R_CRAN_WEB, defaulting to one specified in the "repos" option. Otherwise the entry in the 'repositories' file (see setRepositories) is used: if that specifies '@CRAN@' (the default) or does not contain an entry for CRAN then https://CRAN.R-project.org is used.

The mirror to be used is reported by utils::findCRANmirror("web").

Note that these functions access parts of CRAN under 'web/contrib' and 'web/packages' so if you have specified a mirror of just 'src/contrib' for installing packages you will need to set R_CRAN_WEB to point to a full mirror.

Internal functions CRAN_aliases_db, CRAN_archive_db, CRAN_current_db and CRAN_rdxrefs_db (used by R CMD check) use R_CRAN_SRC rather than R_CRAN_WEB.

Examples

```
## This can be rather slow  with a non-local CRAN mirror
## and might fail (slowly) without Internet access in that case.

set.seed(11)  # but the packages chosen will change as soon as CRAN does.
pdb <- CRAN_package_db()
dim(pdb)
## DESCRIPTION fields included:
colnames(pdb)
## Summarize publication dates:
summary(as.Date(pdb$Published))
## Summarize numbers of packages according to maintainer:
summary(lengths(split(pdb$Package, pdb$Maintainer)))
## Packages with 'LASSO' in their Description:
pdb$Package[grepl("LASSO", pdb$Description)]

results <- CRAN_check_results()
## Available variables:
names(results)
## Tabulate overall check status according to flavor:
with(results, table(Flavor, Status))

details <- CRAN_check_details()
## Available variables:
names(details)
## Tabulate checks according to their status:
tab <- with(details, table(Check, Status))
## Inspect some installation problems:
bad <- subset(details,
              ((Check == "whether package can be installed") &
               (Status != "OK")))
## Show a random sample of up to 6
```

```
head(bad[sample(seq_len(NROW(bad)), NROW(bad)), ])

issues <- CRAN_check_issues()
head(issues)
## Show counts of issues according to kind:
table(issues[, "kind"])

## Summarize CRAN check status for 10 randomly-selected packages
## (reusing the information already read in):
pos <- sample(seq_len(NROW(pdb)), 10L)
summarize_CRAN_check_status(pdb[pos, "Package"],
                            results, details, issues)
```

`delimMatch` *Delimited Pattern Matching*

Description

Match delimited substrings in a character vector, with proper nesting.

Usage

```
delimMatch(x, delim = c("{", "}"), syntax = "Rd")
```

Arguments

`x`	a character vector.
`delim`	a character vector of length 2 giving the start and end delimiters. Future versions might allow for arbitrary regular expressions.
`syntax`	currently, always the string `"Rd"` indicating Rd syntax (i.e., '`%`' starts a comment extending till the end of the line, and '`\`' escapes). Future versions might know about other syntax, perhaps via 'syntax tables' allowing to flexibly specify comment, escape, and quote characters.

Value

An integer vector of the same length as `x` giving the starting position (in characters) of the first match, or -1 if there is none, with attribute `"match.length"` giving the length (in characters) of the matched text (or -1 for no match).

See Also

`regexpr` for 'simple' pattern matching.

Examples

```
x <- c("\\value{foo}", "function(bar)")
delimMatch(x)
delimMatch(x, c("(", ")"))
```

dependsOnPkgs *Find Reverse Dependencies*

Description

Find 'reverse' dependencies of packages, that is those packages which depend on this one, and (optionally) so on recursively.

Usage

```
dependsOnPkgs(pkgs,
              dependencies = "strong",
              recursive = TRUE, lib.loc = NULL,
              installed =
              utils::installed.packages(lib.loc, fields = "Enhances"))
```

Arguments

pkgs	a character vector of package names.
dependencies	a character vector listing the types of dependencies, a subset of `c("Depends", "Imports", "LinkingTo", "Suggests", "Enhances")`. Character string `"all"` is shorthand for that vector, character string `"most"` for the same vector without `"Enhances"`, character string `"strong"` (default) for the first three elements of that vector.
recursive	logical: should reverse dependencies of reverse dependencies (and so on) be included?
lib.loc	a character vector of R library trees, or `NULL` for all known trees (see `.libPaths`).
installed	a result of calling `installed.packages`.

Value

A character vector of package names, which does not include any from `pkgs`.

See Also

`package_dependencies()` to get the regular ("forward") dependencies of a package.

Examples

```
## there are few dependencies in a vanilla R installation:
## lattice may not be installed
dependsOnPkgs("lattice")
```

encoded_text_to_latex *Translate non-ASCII Text to LaTeX Escapes*

Description

Translate non-ASCII characters in text to LaTeX escape sequences.

Usage

```
encoded_text_to_latex(x,
                      encoding = c("latin1", "latin2", "latin9",
                                   "UTF-8", "utf8"))
```

Arguments

x	a character vector.
encoding	the encoding to be assumed. "latin9" is officially ISO-8859-15 or Latin-9, but known as latin9 to LaTeX's inputenc package.

Details

Non-ASCII characters in x are replaced by an appropriate LaTeX escape sequence, or '?' if there is no appropriate sequence.

Even if there is an appropriate sequence, it may not be supported by the font in use. Hyphen is mapped to '\-'.

Value

A character vector of the same length as x.

See Also

iconv

Examples

```
x <- "fran\xE7ais"
encoded_text_to_latex(x, "latin1")
## Not run:
## create a tex file to show the upper half of 8-bit charsets
x <- rawToChar(as.raw(160:255), multiple = TRUE)
(x <- matrix(x, ncol = 16, byrow = TRUE))
xx <- x
xx[] <- encoded_text_to_latex(x, "latin1") # or latin2 or latin9
xx <- apply(xx, 1, paste, collapse = "&")
con <- file("test-encoding.tex", "w")
header <- c(
"\\documentclass{article}",
```

```
"\\usepackage[T1]{fontenc}",
"\\usepackage{Rd}",
"\\begin{document}",
"\\HeaderA{test}{}{test}",
"\\begin{Details}\\relax",
"\\Tabular{ccccccccccccccc}{")
trailer <- c("}", "\\end{Details}", "\\end{document}")
writeLines(header, con)
writeLines(paste0(xx, "\\\\"), con)
writeLines(trailer, con)
close(con)
## and some UTF_8 chars
x <- intToUtf8(as.integer(
    c(160:383,0x0192,0x02C6,0x02C7,0x02CA,0x02D8,
      0x02D9, 0x02DD, 0x200C, 0x2018, 0x2019, 0x201C,
      0x201D, 0x2020, 0x2022, 0x2026, 0x20AC)),
               multiple = TRUE)
x <- matrix(x, ncol = 16, byrow = TRUE)
xx <- x
xx[] <- encoded_text_to_latex(x, "UTF-8")
xx <- apply(xx, 1, paste, collapse = "&")
con <- file("test-utf8.tex", "w")
writeLines(header, con)
writeLines(paste(xx, "\\\\", sep = ""), con)
writeLines(trailer, con)
close(con)

## End(Not run)
```

`fileutils` *File Utilities*

Description

Utilities for listing files, and manipulating file paths.

Usage

```
file_ext(x)
file_path_as_absolute(x)
file_path_sans_ext(x, compression = FALSE)

list_files_with_exts(dir, exts, all.files = FALSE,
                     full.names = TRUE)
list_files_with_type(dir, type, all.files = FALSE,
                     full.names = TRUE, OS_subdirs = .OStype())
```

Arguments

x	character vector giving file paths.
compression	logical: should compression extension '.gz', '.bz2' or '.xz' be removed first?
dir	a character string with the path name to a directory.
exts	a character vector of possible file extensions (excluding the leading dot).
all.files	a logical. If FALSE (default), only visible files are considered; if TRUE, all files are used.
full.names	a logical indicating whether the full paths of the files found are returned (default), or just the file names.
type	a character string giving the 'type' of the files to be listed, as characterized by their extensions. Currently, possible values are "code" (R code), "data" (data sets), "demo" (demos), "docs" (R documentation), and "vignette" (vignettes).
OS_subdirs	a character vector with the names of OS-specific subdirectories to possibly include in the listing of R code and documentation files. By default, the value of the environment variable R_OSTYPE, or if this is empty, the value of .Platform$OS.type, is used.

Details

file_ext returns the file (name) extensions (excluding the leading dot). (Only purely alphanumeric extensions are recognized.)

file_path_as_absolute turns a possibly relative file path absolute, performing tilde expansion if necessary. This is a wrapper for normalizePath. Currently, x must be a single existing path.

file_path_sans_ext returns the file paths without extensions (and the leading dot). (Only purely alphanumeric extensions are recognized.)

list_files_with_exts returns the paths or names of the files in directory dir with extension matching one of the elements of exts. Note that by default, full paths are returned, and that only visible files are used.

list_files_with_type returns the paths of the files in dir of the given 'type', as determined by the extensions recognized by R. When listing R code and documentation files, files in OS-specific subdirectories are included if present according to the value of OS_subdirs. Note that by default, full paths are returned, and that only visible files are used.

See Also

file.path, file.info, list.files

Examples

```
dir <- file.path(R.home(), "library", "stats")
list_files_with_exts(file.path(dir, "demo"), "R")
list_files_with_type(file.path(dir, "demo"), "demo") # the same
file_path_sans_ext(list.files(file.path(R.home("modules"))))
```

find_gs_cmd *Find a GhostScript Executable*

Description

Find a GhostScript executable in a cross-platform way.

Usage

```
find_gs_cmd(gs_cmd = "")
```

Arguments

gs_cmd	The name, full or partial path of a GhostScript executable.

Details

The details differ by platform.

On a Unix-alike, the GhostScript executable is usually called gs. The name (and possibly path) of the command is taken first from argument gs_cmd then from the environment variable R_GSCMD and default gs. This is then looked for on the system path and the value returned if a match is found.

On Windows, the name of the command is taken from argument gs_cmd then from the environment variables R_GSCMD and GSC. If neither of those produces a suitable command name, gswin64c and gswin32c are tried in turn. In all cases the command is looked for on the system PATH.

Note that on Windows (and some other OSes) there are separate GhostScript executables to display Postscript/PDF files and to manipulate them: this function looks for the latter.

Value

A character string giving the full path to a GhostScript executable if one was found, otherwise an empty string.

Examples

```
## Not run:
## Suppose a Solaris system has GhostScript 9.00 on the path and
## 9.07 in /opt/csw/bin.  Then one might set
Sys.setenv(R_GSCMD = "/opt/csw/bin/gs")

## End(Not run)
```

getVignetteInfo *Get Information on Installed Vignettes*

Description

This function gets information on installed vignettes.

Usage

```
getVignetteInfo(package = NULL, lib.loc = NULL, all = TRUE)
```

Arguments

package	Which package to look in, or NULL for all packages.
lib.loc	Which library to look in.
all	Whether to search all installed packages, or just attached packages.

Value

A matrix with columns

Package	the name of the package
Dir	the directory where the package is installed
Topic	the name of the vignette
File	the base filename of the source of the vignette
Title	the title of the vignette
R	the tangled R source from the vignette
PDF	the PDF or HTML file for display

Note

The last column of the result is named PDF for historical reasons, but it may contain a filename of a PDF or HTML document.

See Also

pkgVignettes is a similar function that can work on an uninstalled package.

Examples

```
getVignetteInfo("grid")
```

HTMLheader *Generate a Standard HTML Header for R Help*

Description

This function generates the standard HTML header used on R help pages.

Usage

```
HTMLheader(title = "R", logo = TRUE, up = NULL,
           top = file.path(Rhome, "doc/html/index.html"),
           Rhome = "",
           css = file.path(Rhome, "doc/html/R.css"),
           headerTitle = paste("R:", title),
           outputEncoding = "UTF-8")
```

Arguments

`title`	The title to display and use in the HTML headers. Should have had any HTML escaping already done.
`logo`	Whether to display the R logo after the title.
`up`	Which page (if any) to link to on the "up" button.
`top`	Which page (if any) to link to on the "top" button.
`Rhome`	A **relative** path to the R home directory. See the 'Details'.
`css`	The relative URL for the Cascading Style Sheet.
`headerTitle`	The title used in the headers.
`outputEncoding`	The declared encoding for the whole page.

Details

The `up` and `top` links should be relative to the current page. The `Rhome` path default works with dynamic help; for static help, a relative path (e.g., '`../..`') to it should be used.

Value

A character vector containing the lines of an HTML header which can be used to start a page in the R help system.

Examples

```
cat(HTMLheader("This is a sample header"), sep="\n")
```

HTMLlinks *Collect HTML Links from Package Documentation*

Description

Compute relative file paths for URLs to other package's installed HTML documentation.

Usage

```
findHTMLlinks(pkgDir = "", lib.loc = NULL, level = 0:2)
```

Arguments

pkgDir	the top-level directory of an installed package. The default indicates no package.
lib.loc	character vector describing the location of R library trees to scan: the default indicates .libPaths().
level	Which level(s) to include.

Details

findHTMLlinks tries to resolve links from one help page to another. It uses in decreasing priority

- The package in pkgDir: this is used when converting HTML help for that package (level 0).
- The base and recommended packages (level 1).
- Other packages found in the library trees specified by lib.loc in the order of the trees and alphabetically within a library tree (level 2).

Value

A named character vector of file paths, relative to the 'html' directory of an installed package. So these are of the form '"../../*somepkg*/html/*sometopic*.html"'.

Author(s)

Duncan Murdoch, Brian Ripley

`loadRdMacros` *Load User-defined Rd Help System Macros*

Description

Loads macros from an '`.Rd`' file, or from several '`.Rd`' files contained in a package.

Usage

```
loadRdMacros(file, macros = TRUE)
loadPkgRdMacros(pkgdir, macros = NULL)
```

Arguments

`file`	A file in Rd format containing macro definitions.
`macros`	optionally, a previous set of macro definitions, in the format expected by the `parse_Rd` `macros` argument. `loadPkgRdMacros` loads the system Rd macros by default.
`pkgdir`	The base directory of a source package or an installed package.

Details

The Rd files parsed by these functions should contain only macro definitions; a warning will be issued if anything else other than comments or white space is found.

The `macros` argument may be a filename of a base set of macros, or the result of a previous call to `loadRdMacros` or `loadPkgRdMacros` in the same session. These results should be assumed to be valid only within the current session.

The `loadPkgRdMacros` function first looks for an `"RdMacros"` entry in the package '`DESCRIPTION`' file. If present, it should contain a comma-separated list of other package names; their macros will be loaded before those of the current package. It will then look in the current package for '`.Rd`' files in the '`man/macros`' or '`help/macros`' subdirectories, and load those.

Value

These functions each return an environment containing objects with the names of the newly defined macros from the last file processed. The parent environment will be macros from the previous file, and so on. The first file processed will have `emptyenv()` as its parent.

Author(s)

Duncan Murdoch

References

See the 'Writing R Extensions' manual for the syntax of Rd files, or https://developer.r-project.org/parseRd.pdf for a technical discussion.

See Also

parse_Rd

Examples

```
f <- tempfile()
writeLines(r"(
\newcommand{\Rlogo}{
  \if{html}{\figure{Rlogo.svg}{options: width=100 alt="R logo"}}
  \if{latex}{\figure{Rlogo.pdf}{options: width=0.5in}}
}
)", f)
m <- loadRdMacros(f)
ls(m)
ls(parent.env(m))
ls(parent.env(parent.env(m)))
parse_Rd(textConnection(r"(\Rlogo)"), fragment = TRUE, macros = m)
```

---

makevars *User and Site Compilation Variables*

Description

Determine the location of the user and site specific 'Makevars' files for customizing package compilation.

Usage

```
makevars_user()
makevars_site()
```

Details

Package maintainers can use these functions to employ user and site specific compilation settings also for compilations not using R's mechanisms (in particular, custom compilations in subdirectories of 'src'), e.g., by adding configure code calling R with `cat(tools::makevars_user())` or `cat(tools::makevars_site())`, and if non-empty passing this with '-f' to custom Make invocations.

Value

A character string with the path to the user or site specific 'Makevars' file, or an empty character vector if there is no such file.

See Also

Section 'Customizing package compilation' in the 'R Installation and Administration' manual.

Examples

```
makevars_user()
makevars_site()
```

`make_translations_pkg` *Package the Current Translations in the* R *Sources*

Description

A utility for R Core members to prepare a package of updated translations.

Usage

```
make_translations_pkg(srcdir, outDir = ".", append = "-1")
```

Arguments

`srcdir`	The R source directory.
`outDir`	The directory into which to place the prepared package.
`append`	The suffix for the package version number, e.g. '`3.0.0-1`' will be the default in R 3.0.0.

Details

This extracts the translations in a current R source distribution and packages them as a source package called **translations** which can be distributed on CRAN and installed by `update.packages`. This allows e.g. the translations shipped in R 3.x.y to be updated to those currently in '`R-patched`', even by a user without administrative privileges.

The package has a '`Depends`' field which restricts it to versions '`3.x.*`' for a single `x`.

`matchConcordance` *Concordance between source and target lines*

Description

The Rd parser records locations in '`.Rd`' files from which components of the file are read. Output generators `Rd2HTML` and `Rd2latex` can output information about these locations as "concordances" between source and output lines.

`matchConcordance` converts from output locations to source locations. The `"Rconcordance"` method of `as.character` produces strings to embed in output files, and the default method of `as.Rconcordance` converts these back to objects that can be interpreted by `matchConcordance`.

Usage

```
matchConcordance(linenum, concordance)
## S3 method for class 'Rconcordance'
as.character(x, targetfile = "", ...)
as.Rconcordance(x, ...)
followConcordance(concordance, prevConcordance)
```

Arguments

linenum	One or more line numbers being queried.
concordance	The concordance data for the file containing the lines: an object of class "Rconcordance".
prevConcordance	A concordance object retrieved from the current file.
targetfile	The output filename.
x	The object to convert: for as.character, an "Rconcordance" object; for as.concordance, a character vector which contains as.character output, typically in comments.
...	Further arguments passed to other methods.

Details

The correspondence between target lines and source lines in Rd file conversion is not one to one. Often a single source line can lead to the generation of multiple output lines, and sometimes more than one source line triggers output on the same output line.

matchConcordance converts from target lines to source lines. This can be used to help in understanding how particular output lines depend on the source, e.g. when an error is found in the output file. When more than one line contributes to the output, the last one will be returned.

The "Rconcordance" method of as.character converts a concordance object to strings suitable for embedding (e.g. in comments) in an output file.

The default method of as.Rconcordance searches for strings matching the pattern of as.character.Rconcordance output, then converts those lines back to a single concordance.

followConcordance is used when a file is transformed more than once. The first transformation records a concordance in the file which is read as prevConcordance. followConcordance chains this with the current concordance, relating the final result to the original source.

There are 3 kinds of objects used to hold concordances.

Objects of class "activeConcordance" are internal to **tools**; they are used by Rd2HTML and Rd2latex while building the output file and saving links to the source file.

Objects of class "Rconcordance" are visible to users. They are list objects with the following three fields:

offset The number of lines of output before the first one corresponding to this concordance.

srcLine For each line of output after the offset, the corresponding input file line number. There may be more lines of output than the length of srcLine, in which case nothing can be inferred about the source of those lines.

`srcFile` A vector of filenames of length 1 or the same length as `srcLine` giving the source file(s) for each output line.

Concordance strings are produced by the `"Rconcordance"` method of `as.character`; they are simply character vectors encoding the concordance data. The default method of the `as.concordance` generic function converts them to `"Rconcordance"` objects.

Value

`matchConcordance` returns a character array with one row per input `linenum` entry and two columns, `"srcFile"` and `"srcLine"`.

For the `"Rconcordance"` method of `as.character`, a character vector used (e.g. in `Sweave`) to embed the concordance in a file.

For `as.concordance`, an `"Rconcordance"` object, or `NULL` if no concordance strings are found.

Author(s)

Duncan Murdoch

See Also

`Rd2HTML`, `Rd2latex`

`md5sum` *Compute MD5 Checksums*

Description

Compute the 32-byte MD5 hashes of one or more files.

Usage

```
md5sum(files)
```

Arguments

`files`	character. The paths of file(s) whose contents are to be hashed.

Details

A MD5 'hash' or 'checksum' or 'message digest' is a 128-bit summary of the file contents represented by 32 hexadecimal digits. Files with different MD5 sums are different: only very exceptionally (and usually with the intent to deceive) are those with the same sums different.

On Windows all files are read in binary mode (as the `md5sum` utilities there do): on other OSes the files are read in the default mode (almost always text mode where there is more than one).

MD5 sums are used as a check that R packages have been unpacked correctly and not subsequently accidentally modified.

Value

A character vector of the same length as `files`, with names equal to `files` (possibly expanded). The elements will be `NA` for non-existent or unreadable files, otherwise a 32-character string of hexadecimal digits.

Source

The underlying C code was written by Ulrich Drepper and extracted from a 2001 release of `glibc`.

See Also

`checkMD5sums`

Examples

```
as.vector(md5sum(dir(R.home(), pattern = "^COPY", full.names = TRUE)))
```

---

package_dependencies *Computations on the Dependency Hierarchy of Packages*

Description

Find (recursively) dependencies or reverse dependencies of packages.

Usage

```
package_dependencies(packages = NULL, db = NULL, which = "strong",
                     recursive = FALSE, reverse = FALSE,
                            verbose = getOption("verbose"))
```

Arguments

`packages`	a character vector of package names.
`db`	character matrix as from `available.packages()` (with the default `NULL` the results of this call) or data frame variants thereof. Alternatively, a package database like the one available from `https://cran.r-project.org/web/packages/packages.rds`.
`which`	a character vector listing the types of dependencies, a subset of `c("Depends", "Imports", "LinkingTo", "Suggests", "Enhances")`. Character string `"all"` is shorthand for that vector, character string `"most"` for the same vector without `"Enhances"`, character string `"strong"` (default) for the first three elements of that vector.
`recursive`	a logical indicating whether (reverse) dependencies of (reverse) dependencies (and so on) should be included, or a character vector like `which` indicating the type of (reverse) dependencies to be added recursively.
`reverse`	logical: if `FALSE` (default), regular dependencies are calculated, otherwise reverse dependencies.
`verbose`	logical indicating if output should monitor the package search cycles.

Value

Named list with one element for each package in argument `packages`, each consists of a character vector naming the (recursive) (reverse) dependencies of that package.

For given packages which are not found in the db, `NULL` entries are returned, as opposed to `character(0)` entries which indicate no dependencies.

See Also

`dependsOnPkgs`.

Examples

```
myPkgs <- c("MASS", "Matrix", "KernSmooth", "class", "cluster", "codetools")
pdb <- available.packages(repos = findCRANmirror("web"))
system.time(
dep1 <- package_dependencies(myPkgs, db = pdb) # all arguments at default
) # very fast
utils::str(dep1, vec.len=10)

system.time( ## reverse dependencies, recursively --- takes much longer:
deps <- package_dependencies(myPkgs, db = pdb, which = "most",
                             recursive = TRUE, reverse = TRUE)
) # seen ~ 10 seconds

lengths(deps) # 2020-05-03: all are 16053, but codetools with 16057

## install.packages(dependencies = TRUE) installs 'most' dependencies
## and the strong recursive dependencies of these: these dependencies
## can be obtained using 'which = "most"' and 'recursive = "strong"'.
## To illustrate on the first packages with non-missing Suggests:
packages <- pdb[head(which(!is.na(pdb[, "Suggests"]))), "Package"]
package_dependencies(packages, db = pdb,
                     which = "most", recursive = "strong")
```

---

package_native_routine_registration_skeleton

Write Skeleton for Adding Native Routine Registration to a Package

---

Description

Write a skeleton for adding native routine registration to a package.

Usage

```
package_native_routine_registration_skeleton(dir, con = stdout(),
    align = TRUE, character_only = TRUE, include_declarations = TRUE)
```

Arguments

dir	Top-level directory of a package.
con	Connection on which to write the skeleton: can be specified as a file path.
align	Logical: should the registration tables be lined up in three columns each?
character_only	Logical: should only .NAME arguments specified by character strings (and not as names of R objects nor expressions) be extracted?
include_declarations	Logical: should the output include declarations (also known as 'prototypes') for the registered routines?

Details

Registration is described in section 'Registering native routines' of 'Writing R Extensions'. This function produces a skeleton of the C code which needs to be added to enable registration, conventionally as file 'src/init.c' or appended to the sole C file of the package.

This function examines the code in the 'R' directory of the package for calls to .C, .Fortran, .Call and .External and creates registration information for those it can make sense of. If the number of arguments used cannot be determined it will be recorded as -1: such values should be corrected.

Optionally the skeleton will include declarations for the registered routines: they should be checked against the C/Fortran source code, not least as the number of arguments is taken from the R code. For .Call and .External calls they will often suffice, but for .C and .Fortran calls the 'void *' arguments would ideally be replaced by the actual types. Otherwise declarations need to be included (they may exist earlier in that file if appending to a file, or in a header file which can be included in 'init.c').

The default value of character_only is appropriate when working on a package without any existing registration: character_only = FALSE can be used to suggest updates for a package which has been extended since registration. For the default value, if .NAME values are found which are not character strings (e.g. names or expressions) this is noted via a comment in the output.

Packages which used the earlier form of creating R objects for native symbols *via* additional arguments in a useDynLib directive will probably most easily be updated to use registration with character_only = FALSE.

If an entry point is used with different numbers of arguments in the package's R code, an entry in the table (and optionally, a declaration) is made for each number, and a comment placed in the output. This needs to be resolved: only .External calls can have a variable number of arguments, which should be declared as -1.

A surprising number of CRAN packages had calls in R code to native routines not included in the package, which will lead to a 'loading failed' error during package installation when the registration C code is added.

Calls which do not name a routine such as .Call(...) will be silently ignored.

Value

None: the output is written to the connection con.

Extracting C/C++ prototypes

There are several tools available to extract function declarations from C or C++ code.

For C code one can use `cproto` (https://invisible-island.net/cproto/cproto.html; Windows executables are available), for example

```
cproto -I/path/to/R/include -e *.c
```

`ctags` (commonly distributed with the OS) covers C and C++., using something like

```
ctags -x *.c
```

to list all function usages. (The 'Exuberant' version allows a lot more control.)

Extracting Fortran prototypes

`gfortran` 9.2 and later can extract C prototypes for Fortran subroutines with a special flag:

```
gfortran -c -fc-prototypes-external file.f
```

although ironically not for functions declared `bind(C)`.

Note

This only examines the 'R' directory: it will not find e.g. `.Call` calls used directly in examples, tests *etc*.

Static code analysis is used to find the `.C` etc calls: it *will* find those in parts of the R code 'commented out' by inclusion in `if(FALSE) { ... }`. On the other hand, it will fail to find the entry points in constructs like

```
.Call(if(int) "rle_i" else "rle_d", i, force)
```

and does not know the value of variables in calls like

```
.Call (cfunction, ...)
.Call(..., PACKAGE="sparseLTSEigen")
```

(but if `character_only` is false, will extract the first as `"cfunction"`). Calls which have not been fully resolved will be noted *via* comments in the output file.

Call to entry points in other packages will be ignored if they have an explicit (character string) `PACKAGE` argument.

See Also

`package.skeleton`.

Examples

```
## Not run:
## with a completed splines/DESCRIPTION file,
tools::package_native_routine_registration_skeleton('splines',,,FALSE)
## produces
#include <R.h>
#include <Rinternals.h>
#include <stdlib.h> // for NULL
#include <R_ext/Rdynload.h>

/* FIXME:
   Check these declarations against the C/Fortran source code.
*/

/* .Call calls */
extern SEXP spline_basis(SEXP, SEXP, SEXP, SEXP);
extern SEXP spline_value(SEXP, SEXP, SEXP, SEXP, SEXP);

static const R_CallMethodDef CallEntries[] = {
    {"spline_basis", (DL_FUNC) &spline_basis, 4},
    {"spline_value", (DL_FUNC) &spline_value, 5},
    {NULL, NULL, 0}
};

void R_init_splines(DllInfo *dll)
{
    R_registerRoutines(dll, NULL, CallEntries, NULL, NULL);
    R_useDynamicSymbols(dll, FALSE);
}

## End(Not run)
```

---

parseLatex *Experimental Functions to Work with LaTeX Code*

Description

The `parseLatex` function parses LaTeX source, producing a structured object; `deparseLatex` reverses the process. The `latexToUtf8` function takes a LaTeX object, and processes a number of different macros to convert them into the corresponding UTF-8 characters.

Usage

```
parseLatex(text, filename = deparse1(substitute(text)),
           verbose = FALSE,
           verbatim = c("verbatim", "verbatim*",
                        "Sinput", "Soutput"),
           verb = "\\Sexpr")
deparseLatex(x, dropBraces = FALSE)
latexToUtf8(x)
```

Arguments

text	A character vector containing LaTeX source code.
filename	A filename to use in syntax error messages.
verbose	If TRUE, print debug error messages.
verbatim	A character vector containing the names of LaTeX environments holding verbatim text.
verb	A character vector containing LaTeX macros that should be assumed to hold verbatim text.
x	A "LaTeX" object.
dropBraces	Drop unnecessary braces when displaying a "LaTeX" object.

Details

The parser does not recognize all legal LaTeX code, only relatively simple examples. It does not associate arguments with macros, that needs to be done after parsing, with knowledge of the definitions of each macro. The main intention for this function is to process simple LaTeX code used in bibliographic references, not fully general LaTeX documents.

Verbose text is allowed in three forms: the \verb macro (with single character delimiters), environments whose names are listed in the verbatim argument, and other macros (with brace delimiters) whose names are listed in the verb argument.

Value

The parseLatex() function returns a recursive object of class "LaTeX". Each of the entries in this object will have a "latex_tag" attribute identifying its syntactic role.

The deparseLatex() function returns a single element character vector, possibly containing embedded newlines.

The latexToUtf8() function returns a modified version of the "LaTeX" object that was passed to it.

Author(s)

Duncan Murdoch

Examples

```
latex <- parseLatex("fran\\c{c}ais")
deparseLatex(latexToUtf8(latex))
```

`parse_Rd` *Parse an Rd File*

Description

This function reads an R documentation (Rd) file and parses it, for processing by other functions.

Usage

```
parse_Rd(file, srcfile = NULL, encoding = "unknown",
         verbose = FALSE, fragment = FALSE, warningCalls = TRUE,
  macros = file.path(R.home("share"), "Rd", "macros", "system.Rd"),
         permissive = FALSE)
## S3 method for class 'Rd'
print(x, deparse = FALSE, ...)
## S3 method for class 'Rd'
as.character(x, deparse = FALSE, ...)
```

Arguments

`file`	A filename or text-mode connection. At present filenames work best.
`srcfile`	`NULL`, or a `"srcfile"` object. See the 'Details' section.
`encoding`	Encoding to be assumed for input strings.
`verbose`	Logical indicating whether detailed parsing information should be printed.
`fragment`	Logical indicating whether file represents a complete Rd file, or a fragment.
`warningCalls`	Logical: should parser warnings include the call?
`macros`	Filename or environment from which to load additional macros, or a logical value. See the Details below.
`permissive`	Logical indicating that unrecognized macros should be treated as text with no warning.
`x`	An object of class Rd.
`deparse`	If `TRUE`, attempt to reinstate the escape characters so that the resulting characters will parse to the same object.
`...`	Further arguments to be passed to or from other methods.

Details

This function parses 'Rd' files according to the specification given in https://developer.r-project.org/parseRd.pdf.

It generates a warning for each parse error and attempts to continue parsing. In order to continue, it is generally necessary to drop some parts of the file, so such warnings should not be ignored.

Files without a marked encoding are by default assumed to be in the native encoding. An alternate default can be set using the `encoding` argument. All text in files is translated to the UTF-8 encoding in the parsed object.

As from R version 3.2.0, User-defined macros may be given in a separate file using '\newcommand' or '\renewcommand'. An environment may also be given: it would be produced by loadRdMacros, loadPkgRdMacros, or by a previous call to parse_Rd. If a logical value is given, only the default built-in macros will be used; FALSE indicates that no "macros" attribute will be returned with the result.

The permissive argument allows text to be parsed that is not completely in Rd format. Typically it would be LaTeX code, used in an Rd fragment, e.g. in a bibentry. With permissive = TRUE, this will be passed through as plain text. Since parse_Rd doesn't know how many arguments belong in LaTeX macros, it will guess based on the presence of braces after the macro; this is not infallible.

Value

parse_Rd returns an object of class "Rd". The internal format of this object is subject to change. The as.character() and print() methods defined for the class return character vectors and print them, respectively.

Unless macros = FALSE, the object will have an attribute named "macros", which is an environment containing the macros defined in file, in a format that can be used for further parse_Rd calls in the same session. It is not guaranteed to work if saved to a file and reloaded in a different session.

Author(s)

Duncan Murdoch

References

https://developer.r-project.org/parseRd.pdf

See Also

Rd2HTML for the converters that use the output of parse_Rd().

---

pkg2HTML *Rd Converters*

Description

Generate a single-page HTML reference manual from the Rd sources contained in an installed or source R package.

Usage

```
pkg2HTML(package, dir = NULL, lib.loc = NULL,
         outputEncoding = "UTF-8",
         stylesheet = file.path(R.home("doc"), "html", "R-nav.css"),
         hooks = list(pkg_href = function(pkg) sprintf("%s.html", pkg)),
         texmath = getOption("help.htmlmath"),
         prism = TRUE,
```

```
          out = NULL,
          toc_entry = c("title", "name"),
          ...,
          Rhtml = FALSE,
         mathjax_config = file.path(R.home("doc"), "html", "mathjax-config.js"),
          include_description = TRUE)
```

Arguments

package	a character string, typically giving the name of an installed package. Can also be a file path or URL pointing to a source tarball (this feature is experimental).
dir	character string giving the path to a directory containing an installed or source package.
lib.loc	a character vector describing the location of R library trees to search through, or NULL. Passed on to find.package and Rd_db.
outputEncoding	character string; see Rd2HTML.
stylesheet	character string giving URL containing CSS style information.
hooks	A list of functions controlling details of output. Currently the only component used is pkg_href, which is used to determine the output HTML file path given a package name as input.
texmath	character string controlling math rendering library to be used, either "katex" or "mathjax". The default is to use "katex", unless the package appears to use the mathjaxr package in at least one of its documentation files. The support for MathJax is experimental and may not work well.
prism	logical flag controlling code highlighting, as described in Rd2HTML.
out	a filename or connection object to which to write the output. By default (NULL), the filename is inferred from hooks$pkg_href, which defaults to '*pkg*.html'.
toc_entry	Determines whether the entry for a help page in the table of contents is the name of the help page or its title.
...	additional arguments, passed on to Rd2HTML. The stages argument, if specified, is passed on to Rd_db.
Rhtml	logical: whether the output is intended to be a Rhtml file that can be processed using **knitr**. If TRUE, the examples section is wrapped inside a rcode block.
mathjax_config	character string giving path of file containing configuration instructions for mathjax. Relevant only if texmath = "mathjax".
include_description	logical flag indicating whether the output should begin with the contents of the DESCRIPTION file.

Details

The pkg2HTML function is intended to produce a single-page HTML reference manual for a given package, with links to other packages. The URLs of links to external packages are controlled by the provided hooks.

The handling of \Sexpr-s are necessarily incomplete, but can be controlled to some extent by specifying the stages argument. Best results are likely for installed packages.

Value

The name of the output file (invisibly).

Author(s)

Deepayan Sarkar

See Also

parse_Rd, Rd_db, Rd2HTML.

Examples

```
pkg2HTML("tools", out = tempfile(fileext = ".html")) |> browseURL()
```

---

pskill *Kill a Process*

Description

pskill sends a signal to a process, usually to terminate it.

Usage

```
pskill(pid, signal = SIGTERM)

SIGHUP
SIGINT
SIGQUIT
SIGKILL
SIGTERM
SIGSTOP
SIGTSTP
SIGCHLD
SIGUSR1
SIGUSR2
```

Arguments

pid	positive integers: one or more process IDs as returned by Sys.getpid.
signal	integer, most often one of the symbolic constants.

Details

Signals are a C99 concept, but only a small number are required to be supported (of those listed, only SIGINT and SIGTERM). They are much more widely used on POSIX operating systems (which should define all of those listed here), which also support a kill system call to send a signal to a process, most often to terminate it. Function pskill provides a wrapper: it silently ignores invalid values of its arguments, including zero or negative pids.

In normal use on a Unix-alike, Ctrl-C sends SIGINT, Ctrl-\ sends SIGQUIT and Ctrl-Z sends SIGTSTP: that and SIGSTOP suspend a process which can be resumed by SIGCONT.

The signals are small integers, but the actual numeric values are not standardized (and most do differ between OSes). The SIG* objects contain the appropriate integer values for the current platform (or NA_INTEGER_ if the signal is not defined).

Only SIGINT and SIGTERM will be defined on Windows, and pskill will always use the Windows system call TerminateProcess.

Value

A logical vector of the same length as pid, TRUE (for success) or FALSE, invisibly.

See Also

Package **parallel** has several means to launch child processes which record the process IDs.

psnice

Examples

```
## Not run:
pskill(c(237, 245), SIGKILL)

## End(Not run)
```

---

psnice *Get or Set the Priority (Niceness) of a Process*

Description

Get or set the 'niceness' of the current process, or one or more other processes.

Usage

```
psnice(pid = Sys.getpid(), value = NA_integer_)
```

Arguments

pid	positive integers: the process IDs of one of more processes: defaults to the R session process.
value	The niceness to be set, or NA for an enquiry.

Details

POSIX operating systems have a concept of process priorities, usually from 0 to 39 (or 40) with 20 being a normal priority and (somewhat confusingly) larger numeric values denoting lower priority. To add to the confusion, there is a 'niceness' value, the amount by which the priority numerically exceeds 20 (which can be negative). Processes with high niceness will receive less CPU time than those with normal priority. On some OSes, processes with niceness `+19` are only run when the system would otherwise be idle.

On many OSes utilities such as `top` report the priority and not the niceness. Niceness is used by the utility '`/usr/bin/renice`': '`/usr/bin/nice`' (and `/usr/bin/renice -n`) specifies an *increment* in niceness.

Only privileged users (usually super-users) can lower the niceness.

Windows has a slightly different concept of 'priority classes'. We have mapped the idle priority to niceness `19`, 'below normal' to `15`, normal to `0`, 'above normal' to `-5` and 'realtime' to `-10`. Unlike Unix-alikes, a non-privileged user can increase the priority class on Windows (but using 'realtime' is inadvisable).

Value

An integer vector of *previous* niceness values, `NA` if unknown for any reason.

See Also

Various functions in package **parallel** create child processes whose priority may need to be changed.

`pskill`.

`QC` *QC Checks for R Code and/or Documentation*

Description

Functions for performing various quality control (QC) checks on R code and documentation, notably on R packages.

Usage

```
checkDocFiles   (package, dir, lib.loc = NULL, chkInternal = NULL)
checkDocStyle   (package, dir, lib.loc = NULL)
checkReplaceFuns(package, dir, lib.loc = NULL)
checkS3methods  (package, dir, lib.loc = NULL)
checkRdContents (package, dir, lib.loc = NULL, chkInternal = NULL)

langElts
nonS3methods(package)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source (or *installed* in some cases) directory. This should contain the subdirectories 'R' (for R code) and 'man' with R documentation sources (in Rd format). Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
chkInternal	logical indicating if Rd files marked with keyword internal should be checked as well. If NULL (default), these are checked "specially", ignoring missing documentation of arguments.

Details

checkDocFiles checks, for all Rd files in a package, whether all arguments shown in the usage sections of the Rd file are documented in its arguments section. It also reports duplicated entries in the arguments section, and 'over-documented' arguments which are given in the arguments section but not in the usage.

checkDocStyle investigates how (S3) methods are shown in the usages of the Rd files in a package. It reports the methods shown by their full name rather than using the Rd \method markup for indicating S3 methods. Earlier versions of R also reported about methods shown along with their generic, which typically caused problems for the documentation of the primary argument in the generic and its methods. With \method now being expanded in a way that class information is preserved, joint documentation is no longer necessarily a problem. (The corresponding information is still contained in the object returned by checkDocStyle.)

checkReplaceFuns checks whether replacement functions or S3/S4 replacement methods in the package R code have their final argument named value.

checkS3methods checks whether all S3 methods defined in the package R code have all arguments of the corresponding generic, with positional arguments of the generics in the same positions for the method. As an exception, the first argument of a formula method *may* be called formula even if this is not the name used by the generic. The rules when ... is involved are subtle: see the source code. Functions recognized as S3 generics are those with a call to UseMethod in their body, internal S3 generics (see InternalMethods), and S3 group generics (see Math). Possible dispatch under a different name is not taken into account. The generics are sought first in the given package, then (if given an installed package) in the package imports, and finally the namespace environment for the **base** package.

checkRdContents() checks Rd content, e.g., whether arguments of functions in the usage section have non empty descriptions.

nonS3methods(package) returns a character vector with the names of the functions in package which 'look' like S3 methods, but are not. Using package = NULL returns all known examples.

langElts is a character vector of names of "language elements" of R. These are implemented as "very primitive" functions (no argument list; print()ing as .Primitive("*name*")).

If using an installed package, the checks needing access to all R objects of the package will load the package (unless it is the **base** package), after possibly detaching an already loaded version of the package.

Value

The functions return objects of class the same as the respective function names containing the information about problems detected. There are `print` methods for nicely displaying the information contained in such objects.

`Rcmd` *R CMD Interface*

Description

Invoke R CMD tools from within R.

Usage

```
Rcmd(args, ...)
```

Arguments

`args`	a character vector of arguments to R CMD.
`...`	arguments to be passed to `system2`.

Details

Provides a portable convenience interface to the R CMD mechanism by invoking the corresponding system commands (using the version of R currently used) via `system2`.

Value

See section "Value" in `system2`.

`Rd2HTML` *Rd Converters*

Description

These functions take the output of `parse_Rd()`, an Rd object, and produce a help page from it. As they are mainly intended for internal use, their interfaces are subject to change.

Usage

```
Rd2HTML(Rd, out = "", package = "", defines = .Platform$OS.type,
        Links = NULL, Links2 = NULL,
        stages = "render", outputEncoding = "UTF-8",
        dynamic = FALSE, no_links = FALSE, fragment = FALSE,
        stylesheet = if (dynamic) "/doc/html/R.css" else "R.css",
        texmath = getOption("help.htmlmath"),
        concordance = FALSE,
        standalone = TRUE,
        hooks = list(),
        toc = isTRUE(getOption("help.htmltoc")),
        Rhtml = FALSE,
        ...)

Rd2txt(Rd, out = "", package = "", defines = .Platform$OS.type,
       stages = "render", outputEncoding = "",
       fragment = FALSE, options, ...)

Rd2latex(Rd, out = "", defines = .Platform$OS.type,
         stages = "render", outputEncoding = "UTF-8",
         fragment = FALSE, ..., writeEncoding = TRUE,
         concordance = FALSE)

Rd2ex(Rd, out = "", defines = .Platform$OS.type,
      stages = "render", outputEncoding = "UTF-8",
      commentDontrun = TRUE, commentDonttest = FALSE, ...)
```

Arguments

`Rd`	a filename or `Rd` object to use as input.
`out`	a filename or connection object to which to write the output. The default `out = ""` is equivalent to `out = stdout()`.
`package`	the package to list in the output.
`defines`	string(s) to use in `#ifdef` tests.
`stages`	at which stage (`"build"`, `"install"`, or `"render"`) should `\Sexpr` macros be executed? See the notes below.
`outputEncoding`	see the 'Encodings' section below.
`dynamic`	logical: set links for render-time resolution by dynamic help system.
`no_links`	logical: suppress hyperlinks to other help topics. Used by `R CMD Rdconv`.
`fragment`	logical: should fragments of Rd files be accepted? See the notes below.
`stylesheet`	character: a URL for a stylesheet to be used in the header of the HTML output page.
`texmath`	character: controls how mathematics in `\eqn` and `\deqn` commands are typeset in HTML output. Useful values are `"katex"` (default) and `"mathjax"` to use KaTeX or MathJax respectively, otherwise basic substitutions are used. May be

	ignored under certain circumstances, e.g., if the help page already uses macros from the mathjaxr package.
concordance	Whether concordance data should be embedded in the output file and attached to the return value.
standalone	logical: whether the output is intended to be a standalone HTML file. If FALSE, the header and footer are omitted, so that the output can be combined with other fragments.
hooks	A list of functions controlling details of output. Currently the only component used is pkg_href, which is used by pkg2HTML to determine the output HTML file path given a package name as input.
toc	logical: whether the HTML output should include a table of contents. Ignored unless standalone = TRUE.
Rhtml	logical: whether the output is intended to be a Rhtml file that can be processed using **knitr**. If TRUE, the examples section is wrapped inside a rcode block.
Links, Links2	NULL or a named (by topics) character vector of links, as returned by findHTMLlinks.
options	An optional named list of options to pass to Rd2txt_options.
...	additional parameters to pass to parse_Rd when Rd is a filename.
writeEncoding	should \inputencoding lines be written in the file for non-ASCII encodings?
commentDontrun	should \dontrun sections be commented out?
commentDonttest	should \donttest sections be commented out?

Details

These functions convert help documents: Rd2HTML produces HTML, Rd2txt produces plain text, Rd2latex produces LaTeX. Rd2ex extracts the examples in the format used by example and R utilities.

Each of the functions accepts a filename for an Rd file, and will use parse_Rd to parse it before applying the conversions or checks.

The difference between arguments Link and Link2 is that links are looked in them in turn, so lazy-evaluation can be used to only do a second-level search for links if required.

Before R 3.6.0, the default for Rd2latex was outputEncoding = "ASCII", including using the second option of \enc markup, because LaTeX versions did not provide enough coverage of UTF-8 glyphs for a long time.

Rd2txt will format text paragraphs to a width determined by width, with appropriate margins. The default is to be close to the rendering in versions of R < 2.10.0.

Rd2txt will use directional quotes (see sQuote) if option "useFancyQuotes" is true (usually the default, see sQuote) and the current encoding is UTF-8.

Various aspects of formatting by Rd2txt are controlled by the options argument, documented with the Rd2txt_options function. Changes made using options are temporary, those made with Rd2txt_options are persistent.

When fragment = TRUE, the Rd file will be rendered with no processing of \Sexpr elements or conditional defines using #ifdef or #ifndef. Normally a fragment represents text within a section,

but if the first element of the fragment is a section macro, the whole fragment will be rendered as a series of sections, without the usual sorting.

Value

These functions are executed mainly for the side effect of writing the converted help page. Their value is the name of the output file (invisibly). For `Rd2latex`, the output name is given an attribute `"latexEncoding"` giving the encoding of the file in a form suitable for use with the LaTeX 'inputenc' package. For `Rd2HTML` with `standalone = FALSE`, an attribute `"info"` gives supplementary information such as the contents of the `name` and `title` fields. This is currently experimental, and the details are subject to change.

For `Rd2HTML` and `Rd2latex` with `concordance = TRUE`, a `"concordance"` attribute is added, containing an `Rconcordance` object.

Encodings

Rd files are normally intended to be rendered on a wide variety of systems, so care must be taken in the encoding of non-ASCII characters. In general, any such encoding should be declared using the 'encoding' section for there to be any hope of correct rendering.

For output, the `outputEncoding` argument will be used: `outputEncoding = ""` will choose the native encoding for the current system.

If the text cannot be converted to the `outputEncoding`, byte substitution will be used (see `iconv`): `Rd2latex` and `Rd2ex` give a warning.

Note

The `\Sexpr` macro includes R code that will be executed at one of three times: *build* time (when a package's source code is built into a tarball), *install* time (when the package is installed or built into a binary package), and *render* time (when the man page is converted to a readable format).

For example, this man page was:

1. built on 2024-06-14 at 11:54:19,
2. installed on 2024-06-14 at 11:54:19, and
3. rendered on 2024-06-14 at 11:54:44.

Author(s)

Duncan Murdoch, Brian Ripley

References

https://developer.r-project.org/parseRd.pdf

See Also

`parse_Rd`, `checkRd`, `findHTMLlinks`, `Rd2txt_options`, `matchConcordance`.

Examples

```
## Simulate rendering of this (installed) page in HTML and text format
Rd <- Rd_db("tools")[["Rd2HTML.Rd"]]

outfile <- tempfile(fileext = ".html")
Rd2HTML(Rd, outfile, package = "tools") |> browseURL()

outfile <- tempfile(fileext = ".txt")
Rd2txt(Rd, outfile, package = "tools") |> file.show()
```

Rd2txt_options *Set Formatting Options for Text Help*

Description

This function sets various options for displaying text help.

Usage

```
Rd2txt_options(...)
```

Arguments

...	A list containing named options, or options passed as individual named arguments. See below for currently defined ones.

Details

This function persistently sets various formatting options for the Rd2txt function which is used in displaying text format help. Currently defined options are:

width (default 80): The width of the output page.

minIndent (default 10): The minimum indent to use in a list.

extraIndent (default 4): The extra indent to use in each level of nested lists.

sectionIndent (default 5): The indent level for a section.

sectionExtra (default 2): The extra indentation for each nested section level.

itemBullet (default `"* "`, with the asterisk replaced by a Unicode bullet in UTF-8 and most Windows locales): The symbol to use as a bullet in itemized lists.

enumFormat : A function to format item numbers in enumerated lists.

showURLs (default `FALSE`): Whether to show URLs when expanding `\href` tags.

code_quote (default `TRUE`): Whether to render `\code` and similar with single quotes.

underline_titles (default `TRUE`): Whether to render section titles with underlines (via backspacing).

Value

If called with no arguments, returns all option settings in a list. Otherwise, it changes the named settings and invisibly returns their previous values.

Author(s)

Duncan Murdoch

See Also

Rd2txt

Examples

```
saveOpts <- Rd2txt_options()
saveOpts
Rd2txt_options(minIndent = 4)
Rd2txt_options()[["minIndent"]]
Rd2txt_options(saveOpts)
stopifnot(identical(Rd2txt_options(), saveOpts))
```

Rdiff *Difference R Output Files*

Description

Given two R output files, compute differences ignoring headers, footers and some other differences.

Usage

```
Rdiff(from, to, useDiff = FALSE, forEx = FALSE,
      nullPointers = TRUE, Log = FALSE)
```

Arguments

from, to	filepaths to be compared
useDiff	should diff be used to compare results? Overridden to false if the command is not available.
forEx	logical: extra pruning for '-Ex.Rout' files to exclude headers and footers of examples, code and results for '"--timings"', etc.
nullPointers	logical: should the displayed addresses of pointers be set to 0x00000000 before comparison?
Log	logical: should the returned value include a log of differences found?

Details

The R startup banner and any timing information from R CMD BATCH are removed from both files, together with lines about loading packages. UTF-8 fancy quotes (see sQuote) and on Windows, Windows' so-called 'smart quotes', are mapped to a simple quote. Addresses of environments, compiled bytecode and other exotic types expressed as hex addresses (e.g., '<environment: 0x12345678>') are mapped to 0x00000000. The files are then compared line-by-line. If there are the same number of lines and useDiff is false, a simple diff -b -like display of differences is printed (which ignores trailing spaces and differences in numbers of consecutive spaces), otherwise diff -bw is called on the edited files. (This tries to ignore all differences in whitespace: note that flag '-w' is not required by POSIX but is supported by GNU, Solaris and FreeBSD versions – macOS uses an old GNU version.)

There is limited support for comparing PDF files produced by pdf(compress = FALSE), mainly for use in make check – this requires a diff command and useDiff = TRUE.

Mainly for use in examples and tests, text from marker '> ## IGNORE_RDIFF_BEGIN' up to (but not including) '> ## IGNORE_RDIFF_END' is ignored.

Value

If Log is true, a list with components status (see below) and out, a character vector of descriptions of differences, possibly of zero length.

Otherwise, a status indicator (invisibly), 0L if and only if no differences were found.

See Also

The shell script run as R CMD Rdiff, which uses useDiff = TRUE.

Rdindex *Generate Index from Rd Files*

Description

Print a 2-column index table with names and titles from given R documentation files to a given output file or connection. The titles are nicely formatted between two column positions (typically 25 and 72, respectively).

Usage

```
Rdindex(RdFiles, outFile = "", type = NULL,
        width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

RdFiles	a character vector specifying the Rd files to be used for creating the index, either by giving the paths to the files, or the path to a single directory with the sources of a package.

outFile	a connection, or a character string naming the output file to print to. "" (the default) indicates output to the console.
type	a character string giving the documentation type of the Rd files to be included in the index, or NULL (the default). The type of an Rd file is typically specified via the \docType tag; if type is "data", Rd files whose *only* keyword is datasets are included as well.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a positive integer specifying the indentation of the second column. Must not be greater than width/2, and defaults to width/3.

Details

If a name is not a valid alias, the first alias (or the empty string if there is none) is used instead.

RdTextFilter *Select Text in an Rd File*

Description

This function blanks out all non-text in an Rd file, for spell checking or other uses.

Usage

```
RdTextFilter(ifile, encoding = "unknown", keepSpacing = TRUE,
             drop = character(), keep = character(),
             macros = file.path(R.home("share"), "Rd", "macros", "system.Rd"))
```

Arguments

ifile	An input file specified as a filename or connection, or an "Rd" object from parse_Rd.
encoding	An encoding name to pass to parse_Rd.
keepSpacing	Whether to try to leave the text in the same lines and columns as in the original file.
drop	Additional sections of the Rd to drop.
keep	Sections of the Rd file to keep.
macros	Macro definitions to assume when parsing. See parse_Rd.

Details

This function parses the Rd file, then walks through it, element by element. Items with tag "TEXT" are kept in the same position as they appeared in the original file, while other parts of the file are replaced with blanks, so a spell checker such as aspell can check only the text and report the position in the original file. (If keepSpacing is FALSE, blank filling will not occur, and text will not be output in its original location.)

By default, the tags \S3method, \S4method, \command, \docType, \email, \encoding, \file, \keyword, \link, \linkS4class, \method, \pkg, and \var are skipped. Additional tags can be skipped by listing them in the drop argument; listing tags in the keep argument will stop them from being skipped. It is also possible to keep any of the c("RCODE", "COMMENT", "VERB") tags, which correspond to R-like code, comments, and verbatim text respectively, or to drop "TEXT".

Value

A character vector which if written to a file, one element per line, would duplicate the text elements of the original Rd file.

Note

The filter attempts to merge text elements into single words when markup in the Rd file is used to highlight just the start of a word.

Author(s)

Duncan Murdoch

See Also

aspell, for which this is an acceptable filter.

Rdutils *Rd Utilities*

Description

Utilities for computing on the information in Rd objects.

Usage

```
Rd_db(package, dir, lib.loc = NULL, stages = "build")
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This should contain the subdirectory 'man' with R documentation sources (in Rd format). Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
stages	if dir is specified and the database is being built from source, which stages of \Sexpr processing should be processed?

Details

Rd_db builds a simple database of all Rd objects in a package, as a list of the results of running parse_Rd on the Rd source files in the package and processing platform conditionals and some \Sexpr macros.

See Also

parse_Rd

Examples

```
## Build the Rd db for the (installed) base package.
db <- Rd_db("base")

## Keyword metadata per Rd object.
keywords <- lapply(db, tools:::.Rd_get_metadata, "keyword")
## Tabulate the keyword entries.
kw_table <- sort(table(unlist(keywords)))
## The 5 most frequent ones:
rev(kw_table)[1 : 5]
## The "most informative" ones:
kw_table[kw_table == 1]

## Concept metadata per Rd file.
concepts <- lapply(db, tools:::.Rd_get_metadata, "concept")
## How many files already have \concept metadata?
sum(sapply(concepts, length) > 0)
## How many concept entries altogether?
length(unlist(concepts))
```

---

read.00Index *Read 00Index-style Files*

Description

Read item/description information from '00Index'-like files. Such files are description lists rendered in tabular form, and currently used for the 'INDEX' and 'demo/00Index' files of add-on packages.

Usage

```
read.00Index(file)
```

Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (in this case input can be terminated by a blank line). Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call.

Value

A character matrix with 2 columns named "Item" and "Description" which hold the items and descriptions.

See Also

formatDL for the inverse operation of creating a 00Index-style file from items and their descriptions.

---

showNonASCII *Pick Out Non-ASCII Characters*

Description

This function prints elements of a character vector which contain non-ASCII bytes, printing such bytes as a escape like '<fc>'.

Usage

```
showNonASCII(x)

showNonASCIIfile(file)
```

Arguments

x	a character vector.
file	path to a file.

Details

This was originally written to help detect non-portable text in files in packages.

It prints all elements of x which contain non-ASCII characters, preceded by the element number and with non-ASCII bytes highlighted *via* iconv(sub = "byte").

However, this rendering depends on iconv(to = "ASCII") failing to convert, and macOS 14 no longer does so reliably so for example permille ('\u2030') is rendered as o/oo.

Value

The elements of x containing non-ASCII characters will be returned invisibly.

Examples

```
out <- c(
"fran\xE7ais: test of showNonASCII():",
"\\details{",
"   This is a good line",
"   This has an \xfcmlaut in it.",
"   OK again.",
"}")
f <- tempfile()
cat(out, file = f, sep = "\n")

showNonASCIIfile(f)
unlink(f)
```

`startDynamicHelp` *Start the Dynamic HTML Help System*

Description

This function starts the internal help server, so that HTML help pages are rendered when requested.

Usage

```
startDynamicHelp(start = TRUE)
```

Arguments

`start`	logical: whether to start or shut down the dynamic help system. If `NA`, the server is started if not already running.

Details

This function starts the internal HTTP server, which runs on the loopback interface ('`127.0.0.1`'). If `options("help.ports")` is set to a vector of non-zero integer values, `startDynamicHelp` will try those ports in order; otherwise, it tries up to 10 random ports to find one not in use. It can be disabled by setting the environment variable `R_DISABLE_HTTPD` to a non-empty value or `options("help.ports")` to `0`.

`startDynamicHelp` is called by functions that need to use the server, so would rarely be called directly by a user.

Note that `options(help_type = "html")` must be set to actually make use of HTML help, although it might be the default for an R installation.

If the server cannot be started or is disabled, `help.start` will be unavailable and requests for HTML help will give text help (with a warning).

The browser in use does need to be able to connect to the loopback interface: occasionally it is set to use a proxy for HTTP on all interfaces, which will not work – the solution is to add an exception for '`127.0.0.1`'.

Value

The chosen port number is returned invisibly (which will be `0` if the server has been stopped).

See Also

`help.start` and `help(help_type = "html")` will attempt to start the HTTP server if required

`Rd2HTML` is used to render the package help pages.

`SweaveTeXFilter` *Strip R Code out of Sweave File*

Description

This function blanks out code chunks and Noweb markup in an Sweave input file, for spell checking or other uses.

Usage

```
SweaveTeXFilter(ifile, encoding = "unknown")
```

Arguments

`ifile`	Input file or connection.
`encoding`	Text encoding to pass to `readLines`.

Details

This function blanks out all Noweb markup and code chunks from an Sweave input file, leaving behind the LaTeX source, so that a LaTeX-aware spelling checker can check it and report errors in their original locations.

Value

A character vector which if written to a file, one element per line, would duplicate the text elements of the original Sweave input file.

Author(s)

Duncan Murdoch

See Also

`aspell`, for which this is used with `filter = "Sweave"`.

testInstalledPackage *Test Installed Packages*

Description

These functions allow an installed package to be tested, or all base and recommended packages.

Usage

```
testInstalledPackage(pkg, lib.loc = NULL, outDir = ".",
                     types = c("examples", "tests", "vignettes"),
                     srcdir = NULL, Ropts = "", ...)

testInstalledPackages(outDir = ".", errorsAreFatal = TRUE,
                      scope = c("both", "base", "recommended"),
                      types = c("examples", "tests", "vignettes"),
                      srcdir = NULL, Ropts = "", ...)

testInstalledBasic(scope = c("basic", "devel", "both", "internet", "all"),
                   outDir = file.path(R.home(), "tests"),
                   testSrcdir = getTestSrcdir(outDir))

standard_package_names()
```

Arguments

pkg	name of an installed package.
lib.loc	library path(s) in which to look for the package. See library.
outDir	the directory into which to write the output files: this should already exist. The default, "." is the current working directory. Often a subdirectory is preferable.
types	type(s) of tests to be done.
srcdir	Optional directory to look for .save files.
Ropts	Additional options such as '-d valgrind' to be passed to R CMD BATCH when running examples or tests.
errorsAreFatal	logical: should testing terminate at the first error?
scope	a string indicating which set(s) should be tested. "both" includes "basic" and "devel"; "all" adds "internet". Can be abbreviated.
...	additional arguments use when preparing the files to be run, e.g. commentDontrun and commentDonttest.
testSrcdir	optional directory where the test R scripts are found.

Details

The testInstalledPackage{s}() tests depend on having the package example files installed (which is the default).

If package-specific tests are found in a 'tests' directory they can be tested: these are not installed by default, but will be if R CMD INSTALL --install-tests was used. Finally, the R code in any vignettes can be extracted and tested.
The package-specific tests are run in a '*pkg*-tests' subdirectory of 'outDir', and leave their output there.

testInstalledBasic runs the basic tests, if installed or inside testSrcdir. This should be run with LC_COLLATE=C set: the function tries to set this but it may not work on all OSes. For non-English locales it may be desirable to set environment variables LANGUAGE to 'en' and LC_TIME to 'C' to reduce the number of differences from reference results.

Except on Windows, if the environment variable TEST_MC_CORES is set to an integer greater than one, testInstalledPackages will run the package tests in parallel using its value as the maximum number of parallel processes.

The package-specific tests for the base and recommended packages are not normally installed, but make install-tests is provided to do so (as well as the basic tests).

Value

Invisibly 0L for success, 1L for failure.

standard_package_names() returns a list with components named

base	a character vector with the 'base' package names.
recommended	a character vector with the 'Recommended' package names in historical order.

Examples

```
str(stPkgs <- standard_package_names())

## consistency of packageDescription and standard_package_names :
(pNms <- unlist(stPkgs, FALSE))
(prio <- sapply(as.vector(pNms), packageDescription, fields = "Priority"))
stopifnot(identical(unname(prio),
                    sub("[0-9]+$", '', names(pNms))))
```

---

texi2dvi *Compile LaTeX Files*

Description

Run latex/pdflatex, makeindex and bibtex until all cross-references are resolved to create a DVI or a PDF file.

Usage

```
texi2dvi(file, pdf = FALSE, clean = FALSE, quiet = TRUE,
         texi2dvi = getOption("texi2dvi"),
         texinputs = NULL, index = TRUE)

texi2pdf(file, clean = FALSE, quiet = TRUE,
         texi2dvi = getOption("texi2dvi"),
         texinputs = NULL, index = TRUE)
```

Arguments

file	character string. Name of the LaTeX source file.
pdf	logical. If TRUE, a PDF file is produced instead of the default DVI file (texi2dvi command line option '--pdf').
clean	logical. If TRUE, all auxiliary files created during the conversion are removed.
quiet	logical. No output unless an error occurs.
texi2dvi	character string (or NULL). Script or program used to compile a TeX file to DVI or PDF. The default (selected by "" or "texi2dvi" or NULL) is to look for a program or script named texi2dvi on the path and otherwise emulate the script with system2 calls (which can be selected by the value "emulation"). See also 'Details'.
texinputs	NULL or a character vector of paths to add to the LaTeX and BibTeX input search paths.
index	logical: should indices be prepared?

Details

texi2pdf is a wrapper for the common case of texi2dvi(pdf = TRUE).

Despite the name, this is used in R to compile LaTeX files, specifically those generated from vignettes and by the Rd2pdf script (used for package reference manuals). It ensures that the '*R_HOME*/share/texmf' directory is in the TEXINPUTS path, so R style files such as 'Sweave.sty' and 'Rd.sty' will be found. The TeX search path used is first the existing TEXINPUTS setting (or the current directory if unset), then elements of argument texinputs, then '*R_HOME*/share/texmf' and finally the default path. Analogous changes are made to BIBINPUTS and BSTINPUTS settings.

The default option for texi2dvi is set from environment variable R_TEXI2DVICMD, and the default for that is set from environment variable TEXI2DVI or if that is unset, from a value chosen when R is configured.

A shell script texi2dvi is part of GNU's **texinfo**. Several issues have been seen with released versions, so if yours does not work correctly try R_TEXI2DVICMD=emulation.

Occasionally indices contain special characters which cause indexing to fail (particularly when using the 'hyperref' LaTeX package) even on valid input. The argument index = FALSE is provided to allow package manuals to be made when this happens: it uses emulation.

Value

Invisible NULL. Used for the side effect of creating a DVI or PDF file in the current working directory (and maybe other files, especially if clean = FALSE).

Note

There are various versions of the `texi2dvi` script on Unix-alikes and quite a number of bugs have been seen, some of which this R wrapper works around.

One that was present with `texi2dvi` version `4.8` (as supplied by macOS) is that it will not work correctly for paths which contain spaces, nor if the absolute path to a file would contain spaces.

The three possible approaches all have their quirks. For example the Unix-alike `texi2dvi` script removes ancillary files that already exist but the other two approaches do not (and may get confused by such files).

Where supported (`texi2dvi` 5.0 and later; `texify.exe` from MiKTeX), option '`--max-iterations=20`' is used to avoid infinite retries.

The emulation mode supports `quiet = TRUE` from R 3.2.3 only. Currently `clean = TRUE` only cleans up in this mode if the conversion was successful—this gives users a chance to examine log files in the event of error.

All the approaches should respect the values of environment variables `LATEX`, `PDFLATEX`, `MAKEINDEX` and `BIBTEX` for the full paths to the corresponding commands.

Author(s)

Originally Achim Zeileis but largely rewritten by R-core.

---

toHTML *Display an Object in HTML*

Description

This generic function generates a complete HTML page from an object.

Usage

```
toHTML(x, ...)
## S3 method for class 'packageIQR'
toHTML(x, ...)
## S3 method for class 'news_db'
toHTML(x, ...)
```

Arguments

`x`	An object to display.
`...`	Optional parameters for methods; the `"packageIQR"` and `"news_db"` methods pass these to `HTMLheader`.

Value

A character vector to display the object `x`. The `"packageIQR"` method is designed to display lists in the R help system.

See Also

HTMLheader

Examples

```
cat(toHTML(demo(package = "base")), sep = "\n")
```

---

tools-deprecated *Deprecated Objects in Package* **tools**

Description

(Currently none)

The functions or variables listed here are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

See Also

Deprecated, Defunct

---

toRd *Generic Function to Convert Object to a Fragment of Rd Code*

Description

Methods for this function render their associated classes as a fragment of Rd code, which can then be rendered into text, HTML, or LaTeX.

Usage

```
toRd(obj, ...)
## S3 method for class 'bibentry'
toRd(obj, style = NULL, ...)
```

Arguments

obj	The object to be rendered.
style	The style to be used in converting a bibentry object.
...	Additional arguments used by methods.

Details

See bibstyle for a discussion of styles. The default style = NULL value gives the default style.

Value

Returns a character vector containing a fragment of Rd code that could be parsed and rendered. The default method converts `obj` to mode `character`, then escapes any Rd markup within it. The `bibentry` method converts an object of that class to markup appropriate for use in a bibliography.

`toTitleCase` *Convert Titles to Title Case*

Description

Convert a character vector to title case, especially package titles.

Usage

```
toTitleCase(text)
```

Arguments

`text`	a character vector.

Details

This is intended for English text only.

No definition of'title case' is universally accepted: all agree that 'principal' words are capitalized and common words like 'for' are not, but not which words fall into each category.

Generally words in all capitals are left alone: this implementation knows about conventional mixed-case words such as 'LaTeX' and 'OpenBUGS' and a few technical terms which are not usually capitalized such as 'jar' and 'xls'. However, unknown technical terms will be capitalized unless they are single words enclosed in single quotes: names of packages and libraries should be quoted in titles.

Value

A character vector of the same length as `text`, without names.

undoc *Find Undocumented Objects*

Description

Finds the objects in a package which are undocumented, in the sense that they are visible to the user (or data objects or S4 classes provided by the package), but no documentation entry exists.

Usage

```
undoc(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'man' with R documentation sources (in Rd format), and at least one of the 'R' or 'data' subdirectories with R code or data objects, respectively.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

Details

This function is useful for package maintainers mostly. In principle, *all* user-level R objects should be documented.

The **base** package is special as it contains the primitives and these do not have definitions available at code level. We provide equivalent closures in environments .ArgsEnv and .GenericArgsEnv in the **base** package that are used for various purposes: undoc("base") checks that all the primitives that are not language constructs are prototyped in those environments and no others are.

Value

An object of class "undoc" which is a list of character vectors containing the names of the undocumented objects split according to documentation type.

There is a print method for nicely displaying the information contained in such objects.

See Also

codoc, QC

Examples

```
undoc("tools")                # Undocumented objects in 'tools'
```

update_PACKAGES *Update Existing PACKAGES Files*

Description

Update an existing repository by reading the PACKAGES file, retaining entries which are still valid, removing entries which are no longer valid, and only processing built package tarballs which do not match existing entries.

update_PACKAGES can be much faster than write_PACKAGES for small-moderate changes to large repository indexes, particularly in non-strict mode (see Details).

Usage

```
update_PACKAGES(dir = ".", fields = NULL, type = c("source",
  "mac.binary", "win.binary"), verbose.level = as.integer(dryrun),
  latestOnly = TRUE, addFiles = FALSE, rds_compress = "xz",
  strict = TRUE, dryrun = FALSE)
```

Arguments

dir	see write_PACKAGES.
fields	see write_PACKAGES.
type	see write_PACKAGES.
verbose.level	one of {0, 1, 2}, the level of informative messages displayed throughout the process. Defaults to 0 if dryrun is FALSE (the default) and 1 otherwise. See Details for more information.
latestOnly	see write_PACKAGES.
addFiles	see write_PACKAGES.
rds_compress	see write_PACKAGES.
strict	logical. Should "strict mode" be used when checking existing PACKAGES entries. See Details. Defaults to TRUE.
dryrun	logical. Should the updates to existing PACKAGES files be computed but NOT applied. Defaults to FALSE.

Details

Throughout this section, *package tarball* is defined to mean any archive file in dir whose name can be interpreted as '*package*_*version*.*ext*' – with *ext* the appropriate extension for built packages of type type – (or that is pointed to by the File field of an existing PACKAGES entry). *Novel package tarballs* are those which do not match an existing PACKAGES file entry.

update_PACKAGES calls directly down to write_PACKAGES with a warning (and thus all package tarballs will be processed), if any of the following conditions hold:

- type is "win.binary" and strict is TRUE (no MD5 checksums are included in win.binary PACKAGES files)

- No PACKAGES file exists under dir
- A PACKAGES file exists under dir but is empty
- fields is not NULL and one or more specified fields are not present in the existing PACKAGES file

update_PACKAGES avoids (re)processing package tarballs in cases where a PACKAGES file entry already exists and appears to remain valid. The logic for detecting still-valid entries is as follows:

Any package tarball which was last modified more recently than the existing PACKAGES file is considered novel; existing PACKAGES entries appearing to correspond to such tarballs are *always* considered stale and replaced by newly generated ones. Similarly, all PACKAGES entries that do not correspond to any package tarball found in dir are considered invalid and are excluded from the resulting updated PACKAGES files.

When strict is TRUE, PACKAGES entries that match a package tarball (by package name and version) are confirmed via MD5 checksum; only those that pass are retained as valid. All novel package tarballs are fully processed by the standard machinery underlying write_PACKAGES and the resulting entries are added. Finally, if latestOnly is TRUE, package-version pruning is performed across the entries.

When strict is FALSE, package tarballs are assumed to encode correct metadata in their filenames. PACKAGES entries which appear to match a package tarball are retained as valid (No MD5 checksum testing occurs). If latestOnly is TRUE, package-version pruning is performed across the full set of retained entries and novel package tarballs *before* the processing of the novel tarballs, at significant computational and time savings in some situations. After the optional pruning, any relevant novel package tarballs are processed via the standard machinery and added to the set of retained entries.

In both cases, after the above process concludes, entries are sorted alphabetically by the string concatenation of Package and Version. This should match the entry order write_PACKAGES outputs.

The fields within the entries are ordered as follows: canonical fields - i.e., those appearing as columns when available.packages is called on a CRAN mirror - appear first in their canonical order, followed by any non-canonical fields.

After entry and field reordering, the final database of PACKAGES entries is written to all three PACKAGES files, overwriting the existing versions.

When verbose.level is 0, no extra messages are displayed to the user. When it is 1, detailed information about what is happening is conveyed via messages, but underlying machinery from write_PACKAGES is invoked with verbose = FALSE. Behavior when verbose.level is 2 is identical to verbose.level 1 with the exception that underlying machinery from write_PACKAGES is invoked with verbose = TRUE, which will individually list every processed tarball.

Note

While both strict and non-strict modes can offer speedups when updating small percentages of large repositories, non-strict mode is *much* faster and is recommended in situations where the assumption it makes about tarballs' filenames encoding accurate information is safe.

Note

Users should expect significantly smaller speedups over write_PACKAGES in the type == "win.binary" case on at least some operating systems. This is due to write_PACKAGES being significantly faster in this context, rather than update_PACKAGES being slower.

Author(s)

Gabriel Becker (adapted from previous, related work by him in the **switchr** package which is copyright Genentech, Inc.)

See Also

write_PACKAGES

Examples

```
## Not run:
write_PACKAGES("c:/myFolder/myRepository") # on Windows
update_PACKAGES("c:/myFolder/myRepository") # on Windows
write_PACKAGES("/pub/RWin/bin/windows/contrib/2.9",
type = "win.binary") # on Linux
update_PACKAGES("/pub/RWin/bin/windows/contrib/2.9",
type = "win.binary") # on Linux

## End(Not run)
```

---

update_pkg_po *Prepare Translations for a Package*

Description

Prepare the 'po' directory of a package and optionally compile and install the translations.

Usage

```
update_pkg_po(pkgdir, pkg = NULL, version = NULL,
              pot_make = TRUE, mo_make = TRUE,
              verbose = getOption("verbose"),
              mergeOpts = "", copyright, bugs)
```

Arguments

pkgdir	The path to the package directory.
pkg	The package name: if NULL it is read from the package's 'DESCRIPTION' file.
version	The package version: if NULL it is read from the package's 'DESCRIPTION' file.
pot_make, mo_make	logicals indicating if a new '*.pot' file or new binary translations '*.mo' should be (re)created.
verbose	logical indicating if extra information about the updating process should be printed to the console.
mergeOpts	a string, by default empty, of space-separated options to msgmerge in addition to "--update". Since R 4.2.0, "--no-wrap" is used when called from file.path(R.home("po"), "Makefile").

copyright, bugs optional character strings for the 'Copyright' and 'Report-Msgid-Bugs-To' details in the template files.

Details

This performs a series of steps to prepare or update messages in the package.

- If the package sources do not already have a 'po' directory, one is created.
- xgettext2pot is called to create/update a file 'po/R-*pkgname*.pot' containing the translatable messages in the package.
- All existing files in directory po with names 'R-*lang*.po' are updated from 'R-*pkgname*.pot', checkPoFile is called on the updated file, and if there are no problems the file is compiled and installed under 'inst/po'.
- In a UTF-8 locale, a 'translation' 'R-en@quot.po' is created with UTF-8 directional quotes, compiled and installed under 'inst/po'.
- The remaining steps are done only if file 'po/*pkgname*.pot' already exists. The 'src/*.{c,cc,cpp,m,mm}' files in the package are examined to create a file 'po/*pkgname*.pot' containing the translatable messages in the C/C++ files. If there is a src/windows directory, files within it are also examined.
- All existing files in directory po with names '*lang*.po' are updated from '*pkgname*.pot', checkPoFile is called on the updated file, and if there are no problems the file is compiled and installed under 'inst/po'.
- In a UTF-8 locale, a 'translation' 'en@quot.po' is created with UTF-8 directional quotes, compiled and installed under 'inst/po'.

Note that C/C++ messages are not automatically prepared for translation as they need to be explicitly marked for translation in the source files. Once that has been done, create an empty file 'po/*pkgname*.pot' in the package sources and run this function again.

pkg = "base" is special (and for use by R developers only): the C files are not in the package directory but in the main sources.

System requirements

This function requires the following tools from the GNU gettext-tools: xgettext, msgmerge, msgfmt, msginit and msgconv. These are part of most Linux distributions and easily compiled from the sources on Unix-alikes (including macOS). Pre-compiled versions for Windows are available in https://www.stats.ox.ac.uk/pub/Rtools/goodies/gettext-tools.zip.

It will probably not work correctly for en@quot translations except in a UTF-8 locale, so these are skipped elsewhere.

See Also

xgettext2pot.

`userdir` *R User Directories*

Description

Directories for storing R-related user-specific data, configuration and cache files.

Usage

```
R_user_dir(package, which = c("data", "config", "cache"))
```

Arguments

`package`	a character string giving the name of an R package
`which`	a character string indicating the kind of file(s) of interest. Can be abbreviated.

Details

For desktop environments using X Windows, the freedesktop.org project (formerly X Desktop Group, XDG) developed the XDG Base Directory Specification (https://specifications.freedesktop.org/basedir-spec) for standardizing the location where certain files should be placed. CRAN package **rappdirs** provides these general locations with appropriate values for all platforms for which R is available.

`R_user_dir` specializes the general mechanism to R package specific locations for user files, by providing package specific subdirectories inside a 'R' subdirectory inside the "base" directories appropriate for user-specific data, configuration and cache files (see the examples), with the intent that packages will not interfere if they work within their respective subdirectories.

The locations of these base directories can be customized via the specific environment variables `R_USER_DATA_DIR`, `R_USER_CONFIG_DIR` and `R_USER_CACHE_DIR`. If these are not set, the general XDG-style environment variables `XDG_DATA_HOME`, `XDG_CONFIG_HOME` and `XDG_CACHE_HOME` are used if set, and otherwise, defaults appropriate for the R platform in use are employed.

Examples

```
R_user_dir("FOO", "cache")

## Create one, platform agnostically, must work if <normal> :
(Rdb <- R_user_dir("base"))
if(newD <- !dir.exists(Rdb)) # should work user specifically:
    newD <- dir.create(Rdb, recursive=TRUE)
dir(Rdb) # typically empty
if(newD) unlink(Rdb) # cleaning up

list.files(R_user_dir("grid"), full.names = TRUE)
```

vignetteEngine *Set or Get a Vignette Processing Engine*

Description

Vignettes are normally processed by Sweave, but package writers may choose to use a different engine (e.g., one provided by the **knitr**, **noweb** or **R.rsp** packages). This function is used by those packages to register their engines, and internally by R to retrieve them.

Usage

```
vignetteEngine(name, weave, tangle, pattern = NULL,
        package = NULL, aspell = list())
```

Arguments

name	the name of the engine.
weave	a function to convert vignette source files to PDF/HTML or intermediate LaTeX output.
tangle	a function to convert vignette source files to R code.
pattern	a regular expression pattern for the filenames handled by this engine, or NULL for the default pattern.
package	the package registering the engine. By default, this is the package calling vignetteEngine.
aspell	a list with element names filter and/or control giving the respective arguments to be used when spell checking the text in the vignette source file with aspell.

Details

If weave is missing, vignetteEngine will return the currently registered engine matching name and package.

If weave is NULL, the specified engine will be deleted.

Other settings define a new engine. The weave and tangle functions must be defined with argument lists compatible with function(file, ...). Currently the ... arguments may include logical argument quiet and character argument encoding; others may be added in future. These are described in the documentation for Sweave and Stangle.

The weave and tangle functions should return the filename of the output file that has been produced. Currently the weave function, when operating on a file named '<name><pattern>' must produce a file named '<name>[.](tex|pdf|html)'. The '.tex' files will be processed by pdflatex to produce '.pdf' output for display to the user; the others will be displayed as produced. The tangle function must produce a file named '<name>[.][rRsS]' containing the executable R code from the vignette. The tangle function may support a split = TRUE argument, and then it should produce files named '<name>.*[.][rRsS]'.

The pattern argument gives a regular expression to match the extensions of files which are to be processed as vignette input files. If set to NULL, the default pattern "[.][RrSs](nw|tex)$" is used.

Value

If the engine is being deleted, NULL. Otherwise a list containing components

name	The name of the engine
package	The name of its package
pattern	The pattern for vignette input files
weave	The weave function
tangle	The tangle function

Author(s)

Duncan Murdoch and Henrik Bengtsson.

See Also

Sweave and the 'Writing R Extensions' manual.

Examples

```
str(vignetteEngine("Sweave"))
```

---

vignetteInfo *Basic Information about a Vignette*

Description

Extract metadata from a vignette source file.

Usage

```
vignetteInfo(file)
```

Arguments

file	file name of the vignette.

Value

A list with the following character components:

file	the basename of the file.
title	the vignette title from '\VignetteIndexEntry', possibly an empty string.
depends	a vector of package dependencies from '\VignetteDepends', possibly of length 0.
keywords	a vector of keywords from '\VignetteKeyword', possibly of length 0.
engine	the vignetteEngine, such as "utils::Sweave" or "knitr::knitr".

See Also

package_dependencies for recursive dependencies.

Examples

```
gridEx <- system.file("doc", "grid.Rnw", package = "grid")
vi <- vignetteInfo(gridEx)
str(vi)
```

---

write_PACKAGES *Generate PACKAGES Files*

Description

Generate 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files for a repository of source or Mac/Windows binary packages.

Usage

```
write_PACKAGES(dir = ".", fields = NULL,
               type = c("source", "mac.binary", "win.binary"),
               verbose = FALSE, unpacked = FALSE, subdirs = FALSE,
               latestOnly = TRUE, addFiles = FALSE, rds_compress = "xz",
               validate = FALSE)
```

Arguments

dir	Character vector describing the location of the repository (directory including source or binary packages) to generate the 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files from and write them to.
fields	a character vector giving the fields to be used in the 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files in addition to the default ones, or NULL (default). The default corresponds to the fields needed by available.packages: "Package", "Version", "Priority", "Depends", "Imports", "LinkingTo", "Suggests", "Enhances", "OS_type", "License" and "Archs", and those fields will always be included, plus the file name in field "File" if addFile = TRUE and the path to the subdirectory in field "Path" if subdirectories are used.
type	Type of packages: currently source '.tar.{gz,bz2,xz}' archives, and macOS or Windows binary ('.tgz' or '.zip', respectively) packages are supported. Defaults to "win.binary" on Windows and to "source" otherwise.
verbose	logical. Should packages be listed as they are processed?
unpacked	a logical indicating whether the package contents are available in unpacked form or not (default).
subdirs	either logical (to indicate if subdirectories should be included, recursively) or a character vector of names of subdirectories to include (which are not recursed).

latestOnly	logical: if multiple versions of a package are available should only the latest version be included?
addFiles	logical: should the filenames be included as field 'File' in the 'PACKAGES' file.
rds_compress	The type of compression to be used for 'PACKAGES.rds': see saveRDS. The default is the one found to give maximal compression, and is as used on CRAN.
validate	a logical indicating whether 'DESCRIPTION' files should be validated, and the corresponding packages skipped in case this finds problems.

Details

write_PACKAGES scans the named directory for R packages, extracts information from each package's 'DESCRIPTION' file, and writes this information into the 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files, where the first two represent the information in DCF format, and the third serializes it via saveRDS.

Including non-latest versions of packages is only useful if they have less constraining version requirements, so for example latestOnly = FALSE could be used for a source repository when 'foo_1.0' depends on 'R >= 2.15.0' but 'foo_0.9' is available which depends on 'R >= 2.11.0'.

Support for repositories with subdirectories and hence for subdirs != FALSE depends on recording a "Path" field in the 'PACKAGES' files.

Support for more general file names (e.g., other types of compression) *via* a "File" field in the 'PACKAGES' files can be used by download.packages. If the file names are not of the standard form, use addFiles = TRUE.

type = "win.binary" uses unz connections to read all 'DESCRIPTION' files contained in the (zipped) binary packages for Windows in the given directory dir, and builds files 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files from this information.

For a remote repository there is a tradeoff between download speed and time spent by available.packages processing the downloaded file(s). For large repositories it is likely to be beneficial to use rds_compress = "xz".

Value

Invisibly returns the number of packages described in the resulting 'PACKAGES', 'PACKAGES.gz' and 'PACKAGES.rds' files. If 0, no packages were found and no files were written.

Note

Processing '.tar.gz' archives to extract the 'DESCRIPTION' files is quite slow.

This function can be useful on other OSes to prepare a repository to be accessed by Windows machines, so type = "win.binary" should work on all OSes.

Author(s)

Uwe Ligges and R-core.

See Also

See read.dcf and write.dcf for reading 'DESCRIPTION' files and writing the 'PACKAGES' and 'PACKAGES.gz' files. See update_PACKAGES for efficiently updating existing 'PACKAGES' and 'PACKAGES.gz' files.

Examples

```
## Not run:
write_PACKAGES("c:/myFolder/myRepository")  # on Windows
write_PACKAGES("/pub/RWin/bin/windows/contrib/2.9",
               type = "win.binary")  # on Linux

## End(Not run)
```

---

xgettext *Extract Translatable Messages from R Files in a Package*

Description

For each file in the 'R' directory (including system-specific subdirectories) of a *source* package, extract the unique arguments passed to these "message generating" calls;

for `xgettext()`**:** to stop, warning, message, packageStartupMessage, gettext and gettextf,

for `xngettext()`**:** to ngettext.

`xgettext2pot()` calls both `xgettext()` and then `xngettext()`.

Usage

```
xgettext(dir, verbose = FALSE, asCall = TRUE)

xngettext(dir, verbose = FALSE)

xgettext2pot(dir, potFile, name = "R", version, bugs)
```

Arguments

`dir`	the directory of a source package, i.e., with a '`./R`' sub directory.
`verbose`	logical: should each file be listed as it is processed?
`asCall`	logical: if `TRUE` each argument is converted to string and returned whole, otherwise the string literals within each argument are extracted (recursively). See Examples.
`potFile`	name of po template file to be produced. Defaults to '`R-`*pkgname*`.pot`' where *pkgname* is the basename of '`dir`'.
`name, version, bugs`	as recorded in the template file: `version` defaults the version number of the currently running R, and `bugs` to `"bugs.r-project.org"`.

Details

Leading and trailing white space (space, tab and linefeed (aka newline, i.e., '\n')) is removed for all the calls extracted by `xgettext()`, see 'Description' above, as it is by the internal code that passes strings for translation.

We look to see if the matched functions were called with `domain = NA`. If so, when `asCall` is true, the whole call is omitted. Note that a call might contain a nested call to `gettext` (or `warning`, etc.) whose strings would be visible if `asCall` is false.

`xgettext2pot` calls `xgettext` and then `xngettext`, and writes a PO template file (to `potFile`) for use with the **GNU Gettext** tools. This ensures that the strings for simple translation are unique in the file (as **GNU Gettext** requires), but does not do so for `ngettext` calls (and the rules are not stated in the Gettext manual, but `msgfmt` complains if there is duplication between the sets.).

If applied to the **base** package, this also looks in the '`.R`' files in '*R_HOME*`/share/R`'.

Value

For `xgettext`, a list of objects of class `"xgettext"` (which has a `print` method), one per source file that contains potentially translatable strings.

For `xngettext`, a list of objects of class `"xngettext"`, which are themselves lists of length-2 character vectors.

See Also

`update_pkg_po()` which calls `xgettext2pot()`.

Examples

```
## Not run: ## in a source-directory build (not typical!) of R;
## otherwise, download and unpack the R sources, and replace
## R.home()  by  "<my_path_to_source_R>" :
xgettext(file.path(R.home(), "src", "library", "splines"))

## End(Not run)

## Create source package-like  <tmp>/R/foo.R  and get text from it:
tmpPkg <- tempdir()
tmpRDir <- file.path(tmpPkg, "R")
dir.create(tmpRDir, showWarnings = FALSE)
fnChar <- paste(sep = "\n",
  "foo <- function(x) {",
  "  if (x < -1)  stop('too small')",
  "  # messages unduplicated (not so for ngettext)",
  "  if (x < -.5) stop('too small')",
  "  if (x < 0) {",
  "    warning(",
  "      'sqrt(x) is', sqrt(as.complex(x)),",
  "      ', which may be too small'",
  "    )",
  "  }",
  "  # calls with domain=NA are skipped",
```

```
  "  if (x == 0) cat(gettext('x is 0!\n', domain=NA))",
  "  # gettext strings may be ignored due to 'outer' domain=NA",
  "  if (x > 10) warning('x is ', gettextf('%.2f', x), domain=NA)",
  "  # using a custom condition class",
  "  if (x == 42)",
  "    stop(errorCondition(gettext('needs Deep Thought'), class='myError'))",
  "  x",
  "}")

writeLines(fnChar, con = file.path(tmpRDir, "foo.R"))

## [[1]] : suppressing (tmpfile) name to make example Rdiff-able
xgettext(tmpPkg, asCall=TRUE )[[1]] # default; shows calls
xgettext(tmpPkg, asCall=FALSE)[[1]] # doesn't ; but then ' %.2f '

unlink(tmpRDir, recursive=TRUE)
```

Chapter 14

The `utils` package

`utils-package` *The R Utils Package*

Description

R utility functions

Details

This package contains a collection of utility functions.

For a complete list, use `library(help = "utils")`.

Author(s)

R Core Team and contributors worldwide

Maintainer: R Core Team <R-core@r-project.org>

`adist` *Approximate String Distances*

Description

Compute the approximate string distance between character vectors. The distance is a generalized Levenshtein (edit) distance, giving the minimal possibly weighted number of insertions, deletions and substitutions needed to transform one string into another.

Usage

```
adist(x, y = NULL, costs = NULL, counts = FALSE, fixed = TRUE,
      partial = !fixed, ignore.case = FALSE, useBytes = FALSE)
```

Arguments

x	a character vector. Long vectors are not supported.
y	a character vector, or NULL (default) indicating taking x as y.
costs	a numeric vector or list with names partially matching 'insertions', 'deletions' and 'substitutions' giving the respective costs for computing the Levenshtein distance, or NULL (default) indicating using unit cost for all three possible transformations.
counts	a logical indicating whether to optionally return the transformation counts (numbers of insertions, deletions and substitutions) as the "counts" attribute of the return value.
fixed	a logical. If TRUE (default), the x elements are used as string literals. Otherwise, they are taken as regular expressions and partial = TRUE is implied (corresponding to the approximate string distance used by agrep with fixed = FALSE).
partial	a logical indicating whether the transformed x elements must exactly match the complete y elements, or only substrings of these. The latter corresponds to the approximate string distance used by agrep (by default).
ignore.case	a logical. If TRUE, case is ignored for computing the distances.
useBytes	a logical. If TRUE distance computations are done byte-by-byte rather than character-by-character.

Details

The (generalized) Levenshtein (or edit) distance between two strings *s* and *t* is the minimal possibly weighted number of insertions, deletions and substitutions needed to transform *s* into *t* (so that the transformation exactly matches *t*). This distance is computed for partial = FALSE, currently using a dynamic programming algorithm (see, e.g., https://en.wikipedia.org/wiki/Levenshtein_distance) with space and time complexity $O(mn)$, where m and n are the lengths of *s* and *t*, respectively. Additionally computing the transformation sequence and counts is $O(\max(m, n))$.

The generalized Levenshtein distance can also be used for approximate (fuzzy) string matching, in which case one finds the substring of *t* with minimal distance to the pattern *s* (which could be taken as a regular expression, in which case the principle of using the leftmost and longest match applies), see, e.g., https://en.wikipedia.org/wiki/Approximate_string_matching. This distance is computed for partial = TRUE using 'tre' by Ville Laurikari (https://github.com/laurikari/tre) and corresponds to the distance used by agrep. In this case, the given cost values are coerced to integer.

Note that the costs for insertions and deletions can be different, in which case the distance between *s* and *t* can be different from the distance between *t* and *s*.

Value

A matrix with the approximate string distances of the elements of x and y, with rows and columns corresponding to x and y, respectively.

If counts is TRUE, the transformation counts are returned as the "counts" attribute of this matrix, as a 3-dimensional array with dimensions corresponding to the elements of x, the elements of y, and

the type of transformation (insertions, deletions and substitutions), respectively. Additionally, if `partial = FALSE`, the transformation sequences are returned as the `"trafos"` attribute of the return value, as character strings with elements 'M', 'I', 'D' and 'S' indicating a match, insertion, deletion and substitution, respectively. If `partial = TRUE`, the offsets (positions of the first and last element) of the matched substrings are returned as the `"offsets"` attribute of the return value (with both offsets -1 in case of no match).

See Also

`agrep` for approximate string matching (fuzzy matching) using the generalized Levenshtein distance.

Examples

```
## Cf. https://en.wikipedia.org/wiki/Levenshtein_distance
adist("kitten", "sitting")
## To see the transformation counts for the Levenshtein distance:
drop(attr(adist("kitten", "sitting", counts = TRUE), "counts"))
## To see the transformation sequences:
attr(adist(c("kitten", "sitting"), counts = TRUE), "trafos")

## Cf. the examples for agrep:
adist("lasy", "1 lazy 2")
## For a "partial approximate match" (as used for agrep):
adist("lasy", "1 lazy 2", partial = TRUE)
```

`alarm` *Alert the User*

Description

Gives an audible or visual signal to the user.

Usage

```
alarm()
```

Details

`alarm()` works by sending a `"\a"` character to the console. On most platforms this will ring a bell, beep, or give some other signal to the user (unless standard output has been redirected).

It attempts to flush the console (see `flush.console`).

Value

No useful value is returned.

Examples

```
alarm()
```

apropos *Find Objects by (Partial) Name*

Description

`apropos()` returns a character vector giving the names of objects in the search list matching (as a regular expression) `what`.

`find()` returns where objects of a given name can be found.

Usage

```
apropos(what, where = FALSE, ignore.case = TRUE,
        dot_internals = FALSE, mode = "any")

find(what, mode = "any", numeric = FALSE, simple.words = TRUE)
```

Arguments

`what`	character string. For `simple.words = FALSE` the name of an object; otherwise a regular expression to match object names against.
`where, numeric`	a logical indicating whether positions in the search list should also be returned
`ignore.case`	logical indicating if the search should be case-insensitive, `TRUE` by default.
`dot_internals`	logical indicating if the search result should show **base** internal objects, `FALSE` by default.
`mode`	character; if not `"any"`, only objects whose `mode` equals `mode` are searched.
`simple.words`	logical; if `TRUE`, the `what` argument is only searched as a whole word.

Details

If `mode != "any"` only those objects which are of mode `mode` are considered.

`find` is a different user interface for a similar task to `apropos`. By default (`simple.words == TRUE`), only whole names are matched. Unlike `apropos`, matching is always case-sensitive.

Unlike the default behaviour of `ls`, names which begin with a '.' are included, but base 'internal' objects are included only when `dot_internals` is true.

Value

For `apropos`, a character vector sorted by name. For `where = TRUE` this has names giving the (numerical) positions on the search path.

For `find`, either a character vector of environment names or (for `numeric = TRUE`) a numerical vector of positions on the search path with names the names of the corresponding environments.

Author(s)

Originally, Kurt Hornik and Martin Maechler (May 1997).

See Also

glob2rx to convert wildcard patterns to regular expressions.

objects for listing objects from one place, help.search for searching the help system, search for the search path.

Examples

```
require(stats)

## Not run: apropos("lm")
apropos("GLM")                      # several
apropos("GLM", ignore.case = FALSE) # not one
apropos("lq")

cor <- 1:pi
find("cor")                         #> ".GlobalEnv"   "package:stats"
find("cor", numeric = TRUE)                     # numbers with these names
find("cor", numeric = TRUE, mode = "function")  # only the second one
rm(cor)

## Not run: apropos(".", mode = "list")  # includes many datasets

# extraction/replacement methods (need a DOUBLE backslash '\\')
apropos("\\[")

# everything % not diff-able
length(apropos("."))

# those starting with 'pr'
apropos("^pr")

# the 1-letter things
apropos("^.$")
# the 1-2-letter things
apropos("^..?$")
# the 2-to-4 letter things
apropos("^.{2,4}$")
# frequencies of 8-and-more letter things
table(nchar(apropos("^.{8,}$")))
```

---

aregexec *Approximate String Match Positions*

Description

Determine positions of approximate string matches.

Usage

```
aregexec(pattern, text, max.distance = 0.1, costs = NULL,
         ignore.case = FALSE, fixed = FALSE, useBytes = FALSE)
```

Arguments

pattern	a non-empty character string or a character string containing a regular expression (for fixed = FALSE) to be matched. Coerced by as.character to a string if possible.
text	character vector where matches are sought. Coerced by as.character to a character vector if possible.
max.distance	maximum distance allowed for a match. See agrep.
costs	cost of transformations. See agrep.
ignore.case	a logical. If TRUE, case is ignored for computing the distances.
fixed	If TRUE, the pattern is matched literally (as is). Otherwise (default), it is matched as a regular expression.
useBytes	a logical. If TRUE comparisons are byte-by-byte rather than character-by-character.

Details

aregexec provides a different interface to approximate string matching than agrep (along the lines of the interfaces to exact string matching provided by regexec and grep).

Note that by default, agrep performs literal matches, whereas aregexec performs regular expression matches.

See agrep and adist for more information about approximate string matching and distances.

Comparisons are byte-by-byte if pattern or any element of text is marked as "bytes".

Value

A list of the same length as text, each element of which is either -1 if there is no match, or a sequence of integers with the starting positions of the match and all substrings corresponding to parenthesized subexpressions of pattern, with attribute "match.length" an integer vector giving the lengths of the matches (or -1 for no match).

See Also

regmatches for extracting the matched substrings.

Examples

```
## Cf. the examples for agrep.
x <- c("1 lazy", "1", "1 LAZY")
aregexec("laysy", x, max.distance = 2)
aregexec("(lay)(sy)", x, max.distance = 2)
aregexec("(lay)(sy)", x, max.distance = 2, ignore.case = TRUE)
m <- aregexec("(lay)(sy)", x, max.distance = 2)
regmatches(x, m)
```

arrangeWindows *Rearrange Windows on MS Windows*

Description

This function allows you to tile or cascade windows, or to minimize or restore them (on Windows, i.e. when (.Platform$OS.type == "windows")). This may include windows not "belonging" to R.

Usage

```
arrangeWindows(action, windows, preserve = TRUE, outer = FALSE)
```

Arguments

action	a character string, the action to perform on the windows. The choices are c("vertical", "horizontal", "cascade", "minimize", "restore") with default "vertical"; see the 'Details' for the interpretation. Abbreviations may be used.
windows	a list of window handles, by default produced by getWindowsHandles().
preserve	If TRUE, when tiling preserve the outer boundary of the collection of windows; otherwise make them as large as will fit.
outer	This argument is only used in MDI mode. If TRUE, tile the windows on the system desktop. Otherwise, tile them within the MDI frame.

Details

The actions are as follows:

"vertical" Tile vertically.

"horizontal" Tile horizontally.

"cascade" Cascade the windows.

"minimize" Minimize all of the windows.

"restore" Restore all of the windows to normal size (not minimized, not maximized).

The tiling and cascading are done by the standard Windows API functions, but unlike those functions, they will apply to all of the windows in the windows list.

By default, windows is set to the result of getWindowsHandles() (with one exception described below). This will select windows belonging to the current R process. However, if the global environment contains a variable named .arrangeWindowsDefaults, it will be used as the argument list instead. See the getWindowsHandles man page for a discussion of the optional arguments to that function.

When action = "restore" is used with windows unspecified, minimized = TRUE is added to the argument list of getWindowsHandles so that minimized windows will be restored.

In MDI mode, by default tiling and cascading will happen within the R GUI frame. However, if outer = TRUE, tiling is done on the system desktop. This will generally not give desirable results if any R child windows are included within windows.

Value

This function is called for the side effect of arranging the windows. The list of window handles is returned invisibly.

Note

This is only available on Windows.

Author(s)

Duncan Murdoch

See Also

getWindowsHandles

Examples

```
## Not run: ## Only available on Windows :
arrangeWindows("v")
# This default is useful only in SDI mode:  it will tile any Firefox window
# along with the R windows
.arrangeWindowsDefaults <- list(c("R", "all"), pattern = c("", "Firefox"))
arrangeWindows("v")

## End(Not run)
```

---

askYesNo *Ask a Yes/No Question*

Description

askYesNo provides a standard way to ask the user a yes/no question. It provides a way for front-ends to substitute their own dialogs.

Usage

```
askYesNo(msg, default = TRUE,
         prompts = getOption("askYesNo", gettext(c("Yes", "No", "Cancel"))),
         ...)
```

Arguments

msg	The prompt message for the user.
default	The default response.
prompts	Any of: a character vector containing 3 prompts corresponding to return values of TRUE, FALSE, or NA, or a single character value containing the prompts separated by / characters, or a function to call.
...	Additional parameters, ignored by the default function.

Details

askYesNo will accept case-independent partial matches to the prompts. If no response is given the value of default will be returned; if a non-empty string that doesn't match any of the prompts is entered, an error will be raised.

If a function or single character string naming a function is given for prompts, it will be called as fn(msg = msg, default = default, prompts = prompts, ...). On Windows, the GUI uses the unexported utils:::askYesNoWinDialog function for this purpose.

If strings (or a string such as "Y/N/C") are given as prompts, the choices will be mapped to lower-case for the non-default choices, and left as-is for the default choice.

Value

TRUE for yes, FALSE for no, and NA for cancel.

See Also

readline for more general user input.

Examples

```
if (interactive())
    askYesNo("Do you want to use askYesNo?")
```

---

aspell *Spell Check Interface*

Description

Spell check given files via Aspell, Hunspell or Ispell.

Usage

```
aspell(files, filter, control = list(), encoding = "unknown",
       program = NULL, dictionaries = character())
```

Arguments

files	a character vector with the names of files to be checked.
filter	an optional filter for processing the files before spell checking, given as either a function (with formals ifile and encoding), or a character string specifying a built-in filter, or a list with the name of a built-in filter and additional arguments to be passed to it. See **Details** for available filters. If missing or NULL, no filtering is performed.
control	a list or character vector of control options for the spell checker.
encoding	the encoding of the files. Recycled as needed.

`program` a character string giving the name (if on the system path) or full path of the spell check program to be used, or `NULL` (default). By default, the system path is searched for `aspell`, `hunspell` and `ispell` (in that order), and the first one found is used.

`dictionaries` a character vector of names or file paths of additional R level dictionaries to use. Elements with no path separator specify R system dictionaries (in subdirectory '`share/dictionaries`' of the R home directory). The file extension (currently, only '`.rds`') can be omitted.

Details

The spell check programs employed must support the so-called Ispell pipe interface activated via command line option '`-a`'. In addition to the programs, suitable dictionaries need to be available. See `http://aspell.net`, `https://hunspell.github.io/` and `https://www.cs.hmc.edu/~geoff/ispell.html`, respectively, for obtaining the Aspell, Hunspell and (International) Ispell programs and dictionaries.

On Windows, Aspell is available via MSYS2. One should use a non-Cygwin version, e.g. package `mingw-w64-x86_64-aspell`. The version built against the Cygwin runtime (package `aspell`) requires Unix line endings in files and Unix-style paths, which is incompatible with `aspell()`.

The currently available built-in filters are `"Rd"` (corresponding to `RdTextFilter`, with additional argument `ignore` allowing to give regular expressions for parts of the text to be ignored for spell checking), `"Sweave"` (corresponding to `SweaveTeXFilter`), `"R"`, `"pot"`, `"dcf"` and `"md"`.

Filter `"R"` is for R code and extracts the message string constants in calls to `message`, `warning`, `stop`, `packageStartupMessage`, `gettext`, `gettextf`, and `ngettext` (the unnamed string constants for the first five, and `fmt` and `msg1`/`msg2` string constants, respectively, for the latter two).

Filter `"pot"` is for message string catalog '`.pot`' files. Both have an argument `ignore` allowing to give regular expressions for parts of message strings to be ignored for spell checking: e.g., using `"[\t]'[^']*'[\t[:punct:]]"` ignores all text inside single quotes.

Filter `"dcf"` is for files in Debian Control File format. The fields to keep can be controlled by argument `keep` (a character vector with the respective field names). By default, '`Title`' and '`Description`' fields are kept.

Filter `"md"` is for files in Markdown format ('`.md`' and '`.Rmd`' files), and needs packages **commonmark** and **xml2** to be available.

The print method for the objects returned by `aspell` has an `indent` argument controlling the indentation of the positions of possibly misspelled words. The default is 2; Emacs users may find it useful to use an indentation of 0 and visit output in grep-mode. It also has a `verbose` argument: when this is true, suggestions for replacements are shown as well.

It is possible to employ additional R level dictionaries. Currently, these are files with extension '`.rds`' obtained by serializing character vectors of word lists using `saveRDS`. If such dictionaries are employed, they are combined into a single word list file which is then used as the spell checker's personal dictionary (option '`-p`'): hence, the default personal dictionary is not used in this case.

Value

A data frame inheriting from `aspell` (which has a useful print method) with the information about possibly misspelled words.

References

Kurt Hornik and Duncan Murdoch (2011). "Watch your spelling!" *The R Journal*, **3**(2), 22–28. doi:10.32614/RJ2011014.

See Also

aspell-utils for utilities for spell checking packages.

Examples

```
## Not run:
## To check all Rd files in a directory, (additionally) skipping the
## \references sections.
files <- Sys.glob("*.Rd")
aspell(files, filter = list("Rd", drop = "\\references"))

## To check all Sweave files
files <- Sys.glob(c("*.Rnw", "*.Snw", "*.rnw", "*.snw"))
aspell(files, filter = "Sweave", control = "-t")

## To check all Texinfo files (Aspell only)
files <- Sys.glob("*.texi")
aspell(files, control = "--mode=texinfo")

## End(Not run)

## List the available R system dictionaries.
Sys.glob(file.path(R.home("share"), "dictionaries", "*.rds"))
```

---

aspell-utils *Spell Check Utilities*

Description

Utilities for spell checking packages via Aspell, Hunspell or Ispell.

Usage

```
aspell_package_Rd_files(dir,
                        drop = c("\\abbr", "\\acronym",
                                 "\\author", "\\references"),
                        control = list(), program = NULL,
                        dictionaries = character())
aspell_package_vignettes(dir,
                         control = list(), program = NULL,
                         dictionaries = character())
aspell_package_R_files(dir, ignore = character(), control = list(),
                       program = NULL, dictionaries = character())
```

```
aspell_package_C_files(dir, ignore = character(), control = list(),
                       program = NULL, dictionaries = character())

aspell_write_personal_dictionary_file(x, out, language = "en",
                                      program = NULL)
```

Arguments

dir	a character string specifying the path to a package's root directory.
drop	a character vector naming additional Rd sections to drop when selecting text via RdTextFilter.
control	a list or character vector of control options for the spell checker.
program	a character string giving the name (if on the system path) or full path of the spell check program to be used, or NULL (default). By default, the system path is searched for aspell, hunspell and ispell (in that order), and the first one found is used.
dictionaries	a character vector of names or file paths of additional R level dictionaries to use. See aspell.
ignore	a character vector with regular expressions to be replaced by blanks when filtering the message strings.
x	a character vector, or the result of a call to aspell().
out	a character string naming the personal dictionary file to write to.
language	a character string indicating a language as used by Aspell.

Details

Functions aspell_package_Rd_files, aspell_package_vignettes, aspell_package_R_files and aspell_package_C_files perform spell checking on the Rd files, vignettes, R files, and C-level messages of the package with root directory dir. They determine the respective files, apply the appropriate filters, and run the spell checker.

See aspell for details on filters.

The C-level message string are obtained from the 'po/*PACKAGE*.pot' message catalog file, with *PACKAGE* the basename of dir. See the section on 'C-level messages' in 'Writing R Extensions' for more information.

When using Aspell, the vignette checking skips parameters and/or options of commands \Sexpr, \citep, \code, \pkg, \proglang and \samp (in addition to the what the Aspell TeX/LaTeX filter skips by default). Further commands can be skipped by adding --add-tex-command options to the control argument. E.g., to skip both option and parameter of \mycmd, add --add-tex-command='mycmd op'.

Suitable values for control, program, dictionaries, drop and ignore can also be specified using a package defaults file which should go as 'defaults.R' into the '.aspell' subdirectory of dir, and provides defaults via assignments of suitable named lists, e.g.,

```
vignettes <- list(control = "--add-tex-command='mycmd op'")
```

for vignettes (when using Aspell) and similarly assigning to `Rd_files`, `R_files` and `C_files` for Rd files, R files and C level message defaults.

Maintainers of packages using both English and American spelling will find it convenient to pass control options '`--master=en_US`' and '`--add-extra-dicts=en_GB`' to Aspell and control options '`-d en_US,en_GB`' to Hunspell (provided that the corresponding dictionaries are installed).

Older versions of R had no support for R level dictionaries, and hence provided the function `aspell_write_personal_dictionary_file` to create (spell check) program-specific personal dictionary files from words to be accepted. The new mechanism is to use R level dictionaries, i.e., '`.rds`' files obtained by serializing character vectors of such words using `saveRDS`. For such dictionaries specified via the package defaults mechanism, elements with no path separator can be R system dictionaries or dictionaries in the '`.aspell`' subdirectory.

See Also

`aspell`

---

available.packages *List Available Packages at CRAN-like Repositories*

Description

`available.packages` returns a matrix of details corresponding to packages currently available at one or more repositories. The current list of packages is downloaded over the internet (or copied from a local mirror).

Usage

```
available.packages(contriburl = contrib.url(repos, type), method,
                   fields = NULL, type = getOption("pkgType"),
                   filters = NULL, repos = getOption("repos"),
                   ignore_repo_cache = FALSE, max_repo_cache_age,
                   quiet = TRUE, ...)
```

Arguments

`contriburl`	URL(s) of the '`contrib`' sections of the repositories. Specify this argument only if your repository mirror is incomplete, e.g., because you mirrored only the '`contrib`' section.
`method`	download method, see `download.file`.
`type`	character string, indicate which type of packages: see `install.packages`. If `type = "both"` this will use the source repository.
`fields`	a character vector giving the fields to extract from the '`PACKAGES`' file(s) in addition to the default ones, or `NULL` (default). Unavailable fields result in `NA` values.
`filters`	a character vector or list or `NULL` (default). See 'Details'.
`repos`	character vector, the base URL(s) of the repositories to use.

`ignore_repo_cache`
: logical. If true, the repository cache is never used (see 'Details').

`max_repo_cache_age`
: any cached values older than this in seconds will be ignored. See 'Details'.

`quiet`
: logical, passed to `download.file()`; change only if you know what you are doing.

`...`
: allow additional arguments to be passed from callers (which might be arguments to future versions of this function). Currently these are all passed to `download.file()`.

Details

The list of packages is either copied from a local mirror (specified by a '`file://`' URI) or downloaded. If downloaded and `ignore_repo_cache` is false (the default), the list is cached for the R session in a per-repository file in `tempdir()` with a name like

```
repos_http%3a%2f%2fcran.r-project.org%2fsrc%2fcontrib.rds
```

The cached values are renewed when found to be too old, with the age limit controlled *via* argument `max_repo_cache_age`. This defaults to the current value of the environment variable `R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE`, or if unset, to `3600` (one hour).

By default, the return value includes only packages whose version and OS requirements are met by the running version of R, and only gives information on the latest versions of packages.

Argument `filters` can be used to select which of the packages on the repositories are reported. It is called with its default value (`NULL`) by functions such as `install.packages`: this value corresponds to `getOption("available_packages_filters")` and to `c("R_version", "OS_type", "subarch", "duplicates")` if that is unset or set to `NULL`.

The built-in filters are

`"R_version"` Exclude packages whose R version requirements are not met.

`"OS_type"` Exclude packages whose OS requirement is incompatible with this version of R: that is exclude Windows-only packages on a Unix-alike platform and *vice versa*.

`"subarch"` For binary packages, exclude those with compiled code that is not available for the current sub-architecture, e.g. exclude packages only compiled for 32-bit Windows on a 64-bit Windows R.

`"duplicates"` Only report the latest version where more than one version is available, and only report the first-named repository (in `contriburl`) with the latest version if that is in more than one repository.

`"license/FOSS"` Include only packages for which installation can proceed solely based on packages which can be verified as Free or Open Source Software (FOSS, e.g., https://en.wikipedia.org/wiki/FOSS) employing the available license specifications. Thus both the package and any packages that it depends on to load need to be *known to be* FOSS.

Note that this does depend on the repository supplying license information.

`"license/restricts_use"` Include only packages for which installation can proceed solely based on packages which are known not to restrict use.

"CRAN" Use CRAN versions in preference to versions from other repositories (even if these have a higher version number). This needs to be applied *before* the default "duplicates" filter, so cannot be used with add = TRUE.

If all the filters are from this set, then they can be specified as a character vector; otherwise filters should be a list with elements which are character strings, user-defined functions or add = TRUE (see below).

User-defined filters are functions which take a single argument, a matrix of the form returned by available.packages, and return a matrix consisting of a subset of the rows of the argument.

The special 'filter' add = TRUE appends the other elements of the filter list to the default filters.

Value

A character matrix with one row per package, row names the package names and column names including "Package", "Version", "Priority", "Depends", "Imports", "LinkingTo", "Suggests", "Enhances", "File" and "Repository". Additional columns can be specified using the fields argument.

Where provided by the repository, fields "OS_type", "License", "License_is_FOSS", "License_restricts_use", "Archs", "MD5sum" and "NeedsCompilation" are reported for use by the filters and package management tools, including install.packages.

See Also

packageStatus, update.packages, install.packages, download.packages, contrib.url.

The 'R Installation and Administration' manual for how to set up a repository.

Examples

```
## Count package licenses
db <- available.packages(repos = findCRANmirror("web"), filters = "duplicates")
table(db[,"License"])

## Use custom filter function to only keep recommended packages
## which do not require compilation
available.packages(repos = findCRANmirror("web"),
  filters = list(
    add = TRUE,
    function (db) db[db[,"Priority"] %in% "recommended" &
                     db[,"NeedsCompilation"] == "no", ]
  ))

## Not run:
## Restrict install.packages() (etc) to known-to-be-FOSS packages
options(available_packages_filters =
  c("R_version", "OS_type", "subarch", "duplicates", "license/FOSS"))
## or
options(available_packages_filters = list(add = TRUE, "license/FOSS"))

## Give priority to released versions on CRAN, rather than development
## versions on R-Forge etc.
```

```
options(available_packages_filters =
     c("R_version", "OS_type", "subarch", "CRAN", "duplicates"))

## End(Not run)
```

BATCH *Batch Execution of R*

Description

Run R non-interactively with input from `infile` and send output (stdout/stderr) to another file.

Usage

```
R CMD BATCH [options] infile [outfile]
```

Arguments

`infile`	the name of a file with R code to be executed.
`options`	a list of R command line options, e.g., for setting the amount of memory available and controlling the load/save process. If `infile` starts with a '`-`', use '`--`' as the final option. The default options are '`--restore --save --no-readline`'. (Without '`--no-readline`' on Windows.)
`outfile`	the name of a file to which to write output. If not given, the name used is that of `infile`, with a possible '`.R`' extension stripped, and '`.Rout`' appended.

Details

Use `R CMD BATCH --help` to be reminded of the usage.

By default, the input commands are printed along with the output. To suppress this behavior, add `options(echo = FALSE)` at the beginning of `infile`, or use option '`--no-echo`'.

The `infile` can have end of line marked by LF or CRLF (but not just CR), and files with an incomplete last line (missing end of line (EOL) mark) are processed correctly.

A final expression '`proc.time()`' will be executed after the input script unless the latter calls `q(runLast = FALSE)` or is aborted. This can be suppressed by the option '`--no-timing`'.

Additional options can be set by the environment variable `R_BATCH_OPTIONS`: these come after the default options (see the description of the `options` argument) and before any options given on the command line.

Note

On Unix-alikes only: Unlike `Splus BATCH`, this does not run the R process in the background. In most shells,

```
R CMD BATCH [options] infile [outfile] &
```

will do so.

bibentry *Bibliography Entries*

Description

Functionality for representing and manipulating bibliographic information in enhanced BibTeX style.

Usage

```
bibentry(bibtype, textVersion = NULL, header = NULL, footer = NULL,
         key = NULL, ..., other = list(),
         mheader = NULL, mfooter = NULL)

## S3 method for class 'bibentry'
print(x, style = "text", .bibstyle,
      bibtex = length(x) <= getOption("citation.bibtex.max", 1),
      ...)

## S3 method for class 'bibentry'
format(x, style = "text", .bibstyle = NULL,
       bibtex = length(x) <= 1,
       citMsg = missing(bibtex),
       sort = FALSE, macros = NULL, ...)

## S3 method for class 'bibentry'
sort(x, decreasing = FALSE, .bibstyle = NULL, drop = FALSE, ...)

## S3 method for class 'citation'
 print(x, style = "citation", ...)
## S3 method for class 'citation'
format(x, style = "citation", ...)

## S3 method for class 'bibentry'
toBibtex(object, escape = FALSE, ...)
```

Arguments

bibtype	a character string with a BibTeX entry type. See **Entry Types** for details.
textVersion	a character string with a text representation of the reference to optionally be employed for printing. It is recommended to leave this unspecified if `format(x, style = "text")` works correctly. Only if special LaTeX macros (e.g., math formatting) or special characters (e.g., with accents) are necessary, a `textVersion` should be provided.
header	a character string with optional header text.
footer	a character string with optional footer text.

key	a character string giving the citation key for the entry.
...	for bibentry: arguments of the form *tag=value* giving the fields of the entry, with *tag* and *value* the name and value of the field, respectively. Arguments with empty values are dropped. Field names are case-insensitive. See **Entry Fields** for details. For the print() method, extra arguments to pass to the renderer which typically includes the format() method. For the citation class methods, arguments passed to the next method, i.e., the corresponding bibentry one. For the toBibtex() method, currently not used.
other	a list of arguments as in ... (useful in particular for fields named the same as formals of bibentry).
mheader	a character string with optional "outer" header text.
mfooter	a character string with optional "outer" footer text.
x	an object inheriting from class "bibentry".
style	an optional character string specifying the print style. If present, must be a unique abbreviation (with case ignored) of the available styles, see **Details**.
decreasing	logical, passed to order indicating the sort direction.
.bibstyle	a character string naming a bibliography style, see bibstyle.
bibtex	logical indicating if BibTeX code should be given additionally; currently applies only to style = "citation". The default for the print() method depends on the number of (bib) entries and getOption("citation.bibtex.max") (which itself is 1 by default). For example, to see no BibTeX at all, you can change the default by options(citation.bibtex.max = 0).
citMsg	logical indicating if a "message" should be added (to the footer) about how to get BibTeX code when bibtex is false *and* style = "citation".
sort	logical indicating if bibentries should be sorted, using bibstyle(.bibstyle)$sortKeys(x).
macros	a character string or an object with already loaded Rd macros, see **Details**.
drop	logical used as x[..., drop=drop] inside the sort() method.
object	an object inheriting from class "bibentry".
escape	a logical indicating whether non-ASCII characters should be translated to LaTeX escape sequences.

Details

The bibentry objects created by bibentry can represent an arbitrary positive number of references. One can use c() to combine bibentry objects, and hence in particular build a multiple reference object from single reference ones. Alternatively, one can use bibentry to directly create a multiple reference object by specifying the arguments as lists of character strings.

The print method for bibentry objects is based on a corresponding format method and provides a choice between seven different styles: plain text (style "text"), BibTeX ("bibtex"), a mixture of plain text and BibTeX as traditionally used for citations ("citation"), HTML ("html"), LaTeX ("latex"), R code ("R"), and a simple copy of the textVersion elements (style "textVersion").

The "text", "html" and "latex" styles make use of the .bibstyle argument: a style defined by the bibstyle function for rendering the bibentry into (intermediate) Rd format. The Rd format uses markup commands documented in the 'Rd format' section of the 'Writing R Extensions' manual, e.g. \bold. In addition, one can use the macros argument to provide additional (otherwise unknown, presumably LaTeX-style) Rd macros, either by giving the path to a file with Rd macros to be loaded via loadRdMacros, or an object with macros already loaded. Note that the "latex" result may contain commands from the LaTeX style file 'Rd.sty' shipped with R; put \usepackage{Rd} in the preamble of a LaTeX document to make these available when compiling, e.g. with texi2pdf.

When printing bibentry objects in citation style, a header/footer for each item can be displayed as well as a mheader/mfooter for the whole vector of references.

For formatting as R code, a choice between giving a character vector with one bibentry() call for each bibentry (as commonly used in 'CITATION' files), or a character string with one collapsed call, obtained by combining the individual calls with c() if there is more than one bibentry. This can be controlled by passing the argument collapse=FALSE (default) or TRUE, respectively, to the format() method. (*Printing* in R style always collapses to a single call.)

It is possible to subscript bibentry objects by their keys (which are used for character subscripts if the names are NULL).

There is also a toBibtex method for direct conversion to BibTeX.

As of R 4.3.0, there is also a transform method which allows to directly use the current fields, see the examples.

Value

bibentry produces an object of class "bibentry".

Entry Types

bibentry creates "bibentry" objects, which are modeled after BibTeX entries. The entry should be a valid BibTeX entry type, e.g.,

Article: An article from a journal or magazine.

Book: A book with an explicit publisher.

InBook: A part of a book, which may be a chapter (or section or whatever) and/or a range of pages.

InCollection: A part of a book having its own title.

InProceedings: An article in a conference proceedings.

Manual: Technical documentation like a software manual.

MastersThesis: A Master's thesis.

Misc: Use this type when nothing else fits.

PhdThesis: A PhD thesis.

Proceedings: The proceedings of a conference.

TechReport: A report published by a school or other institution, usually numbered within a series.

Unpublished: A document having an author and title, but not formally published.

Entry Fields

The ... argument of bibentry can be any number of BibTeX fields, including

address: The address of the publisher or other type of institution.

author: The name(s) of the author(s), either as a person object, or as a character string which as.person correctly coerces to such.

booktitle: Title of a book, part of which is being cited.

chapter: A chapter (or section or whatever) number.

doi: The DOI (https://en.wikipedia.org/wiki/Digital_Object_Identifier) for the reference.

editor: Name(s) of editor(s), same format as author.

institution: The publishing institution of a technical report.

journal: A journal name.

note: Any additional information that can help the reader. The first word should be capitalized.

number: The number of a journal, magazine, technical report, or of a work in a series.

pages: One or more page numbers or range of numbers.

publisher: The publisher's name.

school: The name of the school where a thesis was written.

series: The name of a series or set of books.

title: The work's title.

url: A URL for the reference. (If the URL is an expanded DOI, we recommend to use the 'doi' field with the unexpanded DOI instead.)

volume: The volume of a journal or multi-volume book.

year: The year of publication.

See Also

person

Examples

```
## R reference
rref <- bibentry(
   bibtype = "Manual",
   title = "R: A Language and Environment for Statistical Computing",
   author = person("R Core Team"),
   organization = "R Foundation for Statistical Computing",
   address = "Vienna, Austria",
   year = 2014,
   url = "https://www.R-project.org/")

## Different printing styles
print(rref)
print(rref, style = "bibtex")
print(rref, style = "citation")
```

```
print(rref, style = "html")
print(rref, style = "latex")
print(rref, style = "R")

## References for boot package and associated book
bref <- c(
   bibentry(
     bibtype = "Manual",
     title = "boot: Bootstrap R (S-PLUS) Functions",
     author = c(
       person("Angelo", "Canty", role = "aut",
         comment = "S original"),
       person(c("Brian", "D."), "Ripley", role = c("aut", "trl", "cre"),
         comment = "R port, author of parallel support",
         email = "ripley@stats.ox.ac.uk")
     ),
     year = "2012",
     note = "R package version 1.3-4",
     url = "https://CRAN.R-project.org/package=boot",
     key = "boot-package"
   ),

   bibentry(
     bibtype = "Book",
     title = "Bootstrap Methods and Their Applications",
     author = as.person("Anthony C. Davison [aut], David V. Hinkley [aut]"),
     year = "1997",
     publisher = "Cambridge University Press",
     address = "Cambridge",
     isbn = "0-521-57391-2",
     url = "http://statwww.epfl.ch/davison/BMA/",
     key = "boot-book"
   )
)

## Combining and subsetting
c(rref, bref)
bref[2]
bref["boot-book"]

## Extracting fields
bref$author
bref[1]$author
bref[1]$author[2]$email

## Field names are case-insensitive
rref$Year
rref$Year <- R.version$year
stopifnot(identical(rref$year, R.version$year))

## Convert to BibTeX
toBibtex(bref)
```

```
## Transform
transform(rref, address = paste0(address, ", Europe"))

## BibTeX reminder message (in case of >= 2 refs):
print(bref, style = "citation")

## Format in R style
## One bibentry() call for each bibentry:
writeLines(paste(format(bref, "R"), collapse = "\n\n"))
## One collapsed call:
writeLines(format(bref, "R", collapse = TRUE))
```

---

browseEnv *Browse Objects in Environment*

Description

The browseEnv function opens a browser with list of objects currently in sys.frame() environment.

Usage

```
browseEnv(envir = .GlobalEnv, pattern,
          excludepatt = "^last\\.warning",
          html = .Platform$GUI != "AQUA",
          expanded = TRUE, properties = NULL,
          main = NULL, debugMe = FALSE)
```

Arguments

envir	an environment the objects of which are to be browsed.
pattern	a regular expression for object subselection is passed to the internal ls() call.
excludepatt	a regular expression for *dropping* objects with matching names.
html	is used to display the workspace on a HTML page in your favorite browser. The default except when running from R.app on macOS.
expanded	whether to show one level of recursion. It can be useful to switch it to FALSE if your workspace is large. This option is ignored if html is set to FALSE.
properties	a named list of global properties (of the objects chosen) to be showed in the browser; when NULL (as per default), user, date, and machine information is used.
main	a title string to be used in the browser; when NULL (as per default) a title is constructed.
debugMe	logical switch; if true, some diagnostic output is produced.

Details

Very experimental code: displays a static HTML page on all platforms except `R.app` on macOS.

Only allows one level of recursion into object structures.

It can be generalized. See sources for details. Most probably, this should rather work through using the **tkWidget** package (from https://www.bioconductor.org).

See Also

`str`, `ls`.

Examples

```
if(interactive()) {
   ## create some interesting objects :
   ofa <- ordered(4:1)
   ex1 <- expression(1+ 0:9)
   ex3 <- expression(u, v, 1+ 0:9)
   example(factor, echo = FALSE)
   example(table, echo = FALSE)
   example(ftable, echo = FALSE)
   example(lm, echo = FALSE, ask = FALSE)
   example(str, echo = FALSE)

   ## and browse them:
   browseEnv()

   ## a (simple) function's environment:
   af12 <- approxfun(1:2, 1:2, method = "const")
   browseEnv(envir = environment(af12))
 }
```

`browseURL` *Load URL into an HTML Browser*

Description

Load a given URL into an HTML browser.

Usage

```
browseURL(url, browser = getOption("browser"),
          encodeIfNeeded = FALSE)
```

Arguments

`url`	a non-empty character string giving the URL to be loaded. Some platforms also accept file paths.
`browser`	a non-empty character string giving the name of the program to be used as the HTML browser. It should be in the PATH, or a full path specified. Alternatively, an R function to be called to invoke the browser. Under Windows `NULL` is also allowed (and is the default), and implies that the file association mechanism will be used.
`encodeIfNeeded`	Should the URL be encoded by `URLencode` before passing to the browser? This is not needed (and might be harmful) if the `browser` program/function itself does encoding, and can be harmful for '`file://`' URLs on some systems and for '`http://`' URLs passed to some CGI applications. Fortunately, most URLs do not need encoding.

Details

On Unix-alikes: The default browser is set by option `"browser"`, in turn set by the environment variable `R_BROWSER` which is by default set in file '*R_HOME*`/etc/Renviron`' to a choice made manually or automatically when R was configured. (See `Startup` for where to override that default value.) To suppress showing URLs altogether, use the value `"false"`.

On many platforms it is best to set option `"browser"` to a generic program/script and let that invoke the user's choice of browser. For example, on macOS use `open` and on many other Unix-alikes use `xdg-open`.

If `browser` supports remote control and R knows how to perform it, the URL is opened in any already-running browser or a new one if necessary. This mechanism currently is available for browsers which support the `"-remote openURL(...)"` interface (which includes Mozilla and Opera), Galeon, KDE konqueror (*via* kfmclient) and the GNOME interface to Mozilla. (Firefox has dropped support, but defaults to using an already-running browser.) Note that the type of browser is determined from its name, so this mechanism will only be used if the browser is installed under its canonical name.

Because `"-remote"` will use any browser displaying on the X server (whatever machine it is running on), the remote control mechanism is only used if `DISPLAY` points to the local host. This may not allow displaying more than one URL at a time from a remote host.

It is the caller's responsibility to encode `url` if necessary (see `URLencode`).

To suppress showing URLs altogether, set `browser = "false"`.

The behaviour for arguments `url` which are not URLs is platform-dependent. Some platforms accept absolute file paths; fewer accept relative file paths.

On Windows: The default browser is set by option `"browser"`, in turn set by the environment variable `R_BROWSER` if that is set, otherwise to `NULL`. To suppress showing URLs altogether, use the value `"false"`.

Some browsers have required '`:`' be replaced by '`|`' in file paths: others do not accept that. All seem to accept '`\`' as a path separator even though the RFC1738 standard requires '`/`'.

To suppress showing URLs altogether, set `browser = "false"`.

URL schemes

Which URL schemes are accepted is platform-specific: expect 'http://', 'https://' and 'ftp://' to work, but 'mailto:' may or may not (and if it does may not use the user's preferred email client). However, modern browsers are unlikely to handle 'ftp://'.

For the 'file://' scheme the format accepted (if any) can depend on both browser and OS.

Examples

```
## Not run:
## for KDE users who want to open files in a new tab
options(browser = "kfmclient newTab")

browseURL("https://www.r-project.org")

## On Windows-only, something like
browseURL("file://d:/R/R-2.5.1/doc/html/index.html",
          browser = "C:/Program Files/Mozilla Firefox/firefox.exe")

## End(Not run)
```

---

browseVignettes *List Vignettes in an HTML Browser*

Description

List available vignettes in an HTML browser with links to PDF, LaTeX/Noweb source, and (tangled) R code (if available).

Usage

```
browseVignettes(package = NULL, lib.loc = NULL, all = TRUE)

## S3 method for class 'browseVignettes'
print(x, ...)
```

Arguments

package	a character vector with the names of packages to search through, or NULL in which "all" packages (as defined by argument all) are searched.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known.
all	logical; if TRUE search all available packages in the library trees specified by lib.loc, and if FALSE, search only attached packages.
x	Object of class browseVignettes.
...	Further arguments, ignored by the print method.

Details

Function `browseVignettes` returns an object of the same class; the print method displays it as an HTML page in a browser (using `browseURL`).

See Also

`browseURL`, `vignette`

Examples

```
## List vignettes from all *attached* packages
browseVignettes(all = FALSE)

## List vignettes from a specific package
browseVignettes("grid")
```

`bug.report` *Send a Bug Report*

Description

Invokes an editor or email program to write a bug report or opens a web page for bug submission. Some standard information on the current version and configuration of R are included automatically.

Usage

```
bug.report(subject = "",  address,
           file = "R.bug.report", package = NULL, lib.loc = NULL,
           ...)
```

Arguments

`subject`	Subject of the email.
`address`	Recipient's email address, where applicable: for package bug reports sent by email this defaults to the address of the package maintainer (the first if more than one is listed).
`file`	filename to use (if needed) for setting up the email.
`package`	Optional character vector naming a single package which is the subject of the bug report.
`lib.loc`	A character vector describing the location of R library trees in which to search for the package, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`...`	additional named arguments such as `method` and `ccaddress` to pass to `create.post`.

Details

If package is NULL or a base package, this opens the R bugs tracker at https://bugs.r-project.org/.

If package is specified, it is assumed that the bug report is about that package, and parts of its 'DESCRIPTION' file are added to the standard information. If the package has a non-empty BugReports field in the 'DESCRIPTION' file specifying the URL of a webpage, that URL will be opened using browseURL, otherwise an email directed to the package maintainer will be generated using create.post. If there is any other form of BugReports field or a Contact field, this is examined as it may provide a preferred email address.

Value

Nothing useful.

When is there a bug?

If R executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

Taking forever to complete a command can be a bug, but you must make certain that it was really R's fault. Some commands simply take a long time. If the input was such that you KNOW it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an R error message in a case where its usual definition ought to be reasonable, it is probably a bug. If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for statistical analysis. This is a very important sort of problem, but it is also a matter of judgement. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. The mailing list r-devel@r-project.org is a better place for discussions of this sort than the bug list.

If you are not sure what the command is supposed to do after a careful reading of the manual this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the online argument list of a function disagrees with the manual, one of them must be wrong, so report the bug.

How to report a bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, from when you start R until the problem happens. Always include the version of R, machine, and operating system that you are using; type version in R to print this. To help us keep track of which bugs have been fixed and which are still open please send a separate report for each bug.

The most important principle in reporting a bug is to report FACTS, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how R is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that on a data set which you know to be quite large the command `data.frame(x, y, z, monday, tuesday)` never returns. Do not report that `data.frame()` fails for large data sets. Perhaps it fails when a variable name is a day of the week. If this is so then when we got your report we would try out the `data.frame()` command on a large data set, probably with no day of the week variable name, and not see any problem. There is no way in the world that we could guess that we should try a day of the week variable name.

Or perhaps the command fails because the last command you used was a `[` method that had a bug causing R's internal data structures to be corrupted and making the `data.frame()` command fail from then on. This is why we need to know what other commands you have typed (or read from your startup file).

It is very useful to try and find simple examples that produce apparently the same bug, and somewhat useful to find simple examples that might be expected to produce the bug but actually do not. If you want to debug the problem and find exactly what caused it, that is wonderful. You should still report the facts as well as any explanations or solutions.

Invoking R with the '`--vanilla`' option may help in isolating a bug. This ensures that the site profile and saved data files are not read.

A bug report can be generated using the function `bug.report()`. For reports on R this will open the Web page at `https://bugs.r-project.org/`: for a contributed package it will open the package's bug tracker Web page or help you compose an email to the maintainer.

Bug reports on **contributed packages** should not be sent to the R bug tracker: rather make use of the `package` argument.

Author(s)

This help page is adapted from the Emacs manual and the R FAQ

See Also

`help.request` which you possibly should try *before* `bug.report`.

`create.post`, which handles emailing reports.

The R FAQ, also `sessionInfo()` from which you may add to the bug report.

`capture.output` *Send Output to a Character String or File*

Description

Evaluates its arguments with the output being returned as a character string or sent to a file. Related to `sink` similarly to how `with` is related to `attach`.

Usage

```
capture.output(..., file = NULL, append = FALSE,
               type = c("output", "message"), split = FALSE)
```

Arguments

...	Expressions to be evaluated.
file	A file name or a connection, or NULL to return the output as a character vector. If the connection is not open, it will be opened initially and closed on exit.
append	logical. If file a file name or unopened connection, append or overwrite?
type, split	are passed to sink(), see there.

Details

It works via sink(<file connection>) and hence the R code in dots must *not* interfere with the connection (e.g., by calling closeAllConnections()).

An attempt is made to write output as far as possible to file if there is an error in evaluating the expressions, but for file = NULL all output will be lost.

Messages sent to stderr() (including those from message, warning and stop) are captured by type = "message". Note that this can be "unsafe" and should only be used with care.

Value

A character string (if file = NULL), or invisible NULL.

See Also

sink, textConnection

Examples

```
require(stats)
glmout <- capture.output(summary(glm(case ~ spontaneous+induced,
                                     data = infert, family = binomial())))
glmout[1:5]
capture.output(1+1, 2+2)
capture.output({1+1; 2+2})

## Not run: ## on Unix-alike with a2ps available
op <- options(useFancyQuotes=FALSE)
pdf <- pipe("a2ps -o - | ps2pdf - tempout.pdf", "w")
capture.output(example(glm), file = pdf)
close(pdf); options(op) ; system("evince tempout.pdf &")

## End(Not run)
```

changedFiles *Detect which Files Have Changed*

Description

fileSnapshot takes a snapshot of a selection of files, recording summary information about each. changedFiles compares two snapshots, or compares one snapshot to the current state of the file system. The snapshots need not be the same directory; this could be used to compare two directories.

Usage

```
fileSnapshot(path = ".", file.info = TRUE, timestamp = NULL,
     md5sum = FALSE, digest = NULL, full.names = length(path) > 1,
     ...)

changedFiles(before, after, path = before$path, timestamp = before$timestamp,
     check.file.info = c("size", "isdir", "mode", "mtime"),
     md5sum = before$md5sum, digest = before$digest,
     full.names = before$full.names, ...)

## S3 method for class 'fileSnapshot'
print(x, verbose = FALSE, ...)

## S3 method for class 'changedFiles'
print(x, verbose = FALSE, ...)
```

Arguments

path	character vector; the path(s) to record.
file.info	logical; whether to record file.info values for each file.
timestamp	character string or NULL; the name of a file to write at the time the snapshot is taken. This gives a quick test for modification, but may be unreliable; see the Details.
md5sum	logical; whether MD5 summaries of each file should be taken as part of the snapshot.
digest	a function or NULL; a function with header function(filename) which will take a vector of filenames and produce a vector of values of the same length, or a matrix with that number of rows.
full.names	logical; whether full names (as in list.files) should be recorded. Must be TRUE if length(path) > 1.
...	additional parameters to pass to list.files to control the set of files in the snapshots.
before, after	objects produced by fileSnapshot; two snapshots to compare. If after is missing, a new snapshot of the current file system will be produced for comparison, using arguments recorded in before as defaults.

check.file.info
: character vector; which columns from file.info should be compared.

x
: the object to print.

verbose
: logical; whether to list all data when printing.

Details

The fileSnapshot function uses list.files to obtain a list of files, and depending on the file.info, md5sum, and digest arguments, records information about each file.

The changedFiles function compares two snapshots.

If the timestamp argument to fileSnapshot is length 1, a file with that name is created. If it is length 1 in changedFiles, the file_test function is used to compare the age of all files common to both before and after to it. This test may be unreliable: it compares the current modification time of the after files to the timestamp; that may not be the same as the modification time when the after snapshot was taken. It may also give incorrect results if the clock on the file system holding the timestamp differs from the one holding the snapshot files.

If the check.file.info argument contains a non-empty character vector, the indicated columns from the result of a call to file.info will be compared.

If md5sum is TRUE, fileSnapshot will call the tools::md5sum function to record the 32 byte MD5 checksum for each file, and changedFiles will compare the values. The digest argument allows users to provide their own digest function.

Value

fileSnapshot returns an object of class "fileSnapshot". This is a list containing the fields

info
: a data frame whose rownames are the filenames, and whose columns contain the requested snapshot data

path
: the normalized path from the call

timestamp, file.info, md5sum, digest, full.names
: a record of the other arguments from the call

args
: other arguments passed via ... to list.files.

changedFiles produces an object of class "changedFiles". This is a list containing

added, deleted, changed, unchanged
: character vectors of filenames from the before and after snapshots, with obvious meanings

changes
: a logical matrix with a row for each common file, and a column for each comparison test. TRUE indicates a change in that test.

print methods are defined for each of these types. The print method for "fileSnapshot" objects displays the arguments used to produce them, while the one for "changedFiles" displays the added, deleted and changed fields if non-empty, and a submatrix of the changes matrix containing all of the TRUE values.

Author(s)

Duncan Murdoch, using suggestions from Karl Millar and others.

See Also

`file.info`, `file_test`, `md5sum`.

Examples

```
# Create some files in a temporary directory
dir <- tempfile()
dir.create(dir)
writeBin(1L, file.path(dir, "file1"))
writeBin(2L, file.path(dir, "file2"))
dir.create(file.path(dir, "dir"))

# Take a snapshot
snapshot <- fileSnapshot(dir, timestamp = tempfile("timestamp"), md5sum=TRUE)

# Change one of the files.
writeBin(3L:4L, file.path(dir, "file2"))

# Display the detected changes.  We may or may not see mtime change...
changedFiles(snapshot)
changedFiles(snapshot)$changes
```

---

charClass *Character Classification*

Description

An interface to the (C99) wide character classification functions in use.

Usage

```
charClass(x, class)
```

Arguments

x	**Either** a UTF-8-encoded length-1 character vector **or** an integer vector of Unicode points (or a vector coercible to integer).
class	A character string, one of those given in the 'Details' section.

Details

The classification into character classes is platform-dependent. The classes are determined by internal tables on Windows and (optionally but by default) on macOS and AIX.

The character classes are interpreted as follows:

`"alnum"` Alphabetic or numeric.

`"alpha"` Alphabetic.

`"blank"` Space or tab.

`"cntrl"` Control characters.

`"digit"` Digits `0-9`.

`"graph"` Graphical characters (printable characters except whitespace).

`"lower"` Lower-case alphabetic.

`"print"` Printable characters.

`"punct"` Punctuation characters. Some platforms treat all non-alphanumeric graphical characters as punctuation.

`"space"` Whitespace, including tabs, form and line feeds and carriage returns. Some OSes include non-breaking spaces, some exclude them.

`"upper"` Upper-case alphabetic.

`"xdigit"` Hexadecimal character, one of `0-9A-fa-f`.

Alphabetic characters contain all lower- and upper-case ones and some others (for example, those in 'title case').

Whether a character is printable is used to decide whether to escape it when printing – see the help for `print.default`.

If `x` is a character string it should either be ASCII or declared as UTF-8 – see `Encoding`.

`charClass` was added in R 4.1.0. A less direct way to examine character classes which also worked in earlier versions is to use something like `grepl("[[:print:]]", intToUtf8(x))` – however, the regular-expression code might not use the same classification functions as printing and on macOS used not to.

Value

A logical vector of the length the number of characters or integers in `x`.

Note

Non-ASCII digits are excluded by the C99 standard from the class `"digit"`: most platforms will have them as alphabetic.

It is an assumption that the system's wide character classification functions are coded in Unicode points, but this is known to be true for all recent platforms.

The classification may depend on the locale even on one platform.

See Also

Character classes are used in regular expressions.

The OS's man pages for `iswctype` and `wctype`.

Examples

```
x <- c(48:70, 32, 0xa0) # Last is non-breaking space
cl <- c("alnum", "alpha", "blank", "digit", "graph", "punct", "upper", "xdigit")
X <- lapply(cl, function(y) charClass(x,y)); names(X) <- cl
X <- as.data.frame(X); row.names(X) <- sQuote(intToUtf8(x, multiple = TRUE))
X

charClass("ABC123", "alpha")
## Some accented capital Greek characters
(x <- "\u0386\u0388\u0389")
charClass(x, "upper")

## How many printable characters are there? (Around 280,000 in Unicode 13.)
## There are 2^21-1 possible Unicode points (most not yet assigned).
pr <- charClass(1:0x1fffff, "print")
table(pr)
```

---

choose.dir *Choose a Folder Interactively on MS Windows*

Description

Use a Windows shell folder widget to choose a folder interactively.

Usage

```
choose.dir(default = "", caption = "Select folder")
```

Arguments

`default`	which folder to show initially.
`caption`	the caption on the selection dialog.

Details

This brings up the Windows shell folder selection widget. With the default `default = ""`, 'My Computer' (or similar) is initially selected.

To workaround a bug, on Vista and later only folders under 'Computer' are accessible via the widget.

Value

A length-one character vector, character `NA` if 'Cancel' was selected.

Note

This is only available on Windows.

See Also

choose.files (on Windows) and file.choose (on all platforms).

Examples

```
if (interactive() && .Platform$OS.type == "windows")
      choose.dir(getwd(), "Choose a suitable folder")
```

---

choose.files *Choose a List of Files Interactively on MS Windows*

Description

Use a Windows file dialog to choose a list of zero or more files interactively.

Usage

```
choose.files(default = "", caption = "Select files",
             multi = TRUE, filters = Filters,
             index = nrow(Filters))

Filters
```

Arguments

default	which filename to show initially
caption	the caption on the file selection dialog
multi	whether to allow multiple files to be selected
filters	a matrix of filename filters (see Details)
index	which row of filters to use by default

Details

Unlike file.choose, choose.files will always attempt to return a character vector giving a list of files. If the user cancels the dialog, then zero files are returned, whereas file.choose would signal an error. choose.dir chooses a directory.

Windows file dialog boxes include a list of 'filters', which allow the file selection to be limited to files of specific types. The filters argument to choose.files allows the list of filters to be set. It should be an n by 2 character matrix. The first column gives, for each filter, the description the user will see, while the second column gives the mask(s) to select those files. If more than one mask is used, separate them by semicolons, with no spaces. The index argument chooses which filter will be used initially.

`Filters` is a matrix giving the descriptions and masks for the file types that R knows about. Print it to see typical formats for filter specifications. The examples below show how particular filters may be selected.

If you would like to display files in a particular directory, give a fully qualified file mask (e.g., `"c:\\*.*"`) in the `default` argument. If a directory is not given, the dialog will start in the current directory the first time, and remember the last directory used on subsequent invocations.

There is a buffer limit on the total length of the selected filenames: it is large but this function is not intended to select thousands of files, when the limit might be reached.

Value

A character vector giving zero or more file paths.

Note

This is only available on Windows.

See Also

`file.choose`, `choose.dir`.

`Sys.glob` or `list.files` to select multiple files by pattern.

Examples

```
if (interactive() && .Platform$OS.type == "windows")
     choose.files(filters = Filters[c("zip", "All"),])
```

---

chooseBioCmirror *Select a Bioconductor Mirror*

Description

Interact with the user to choose a Bioconductor mirror.

Usage

```
chooseBioCmirror(graphics = getOption("menu.graphics"), ind = NULL,
                 local.only = FALSE)
```

Arguments

`graphics`	Logical. If true, use a graphical list: on Windows or the macOS GUI use a list box, and on a Unix-alike use a Tk widget if package **tcltk** and an X server are available. Otherwise use a text `menu`.
`ind`	Optional numeric value giving which entry to select.
`local.only`	Logical, try to get most recent list from the Bioconductor master or use file on local disk only.

Details

This sets the option "BioC_mirror": it is used before a call to setRepositories. The out-of-the-box default for that option is NULL, which currently corresponds to the mirror https://bioconductor.org.

The 'Bioconductor (World-wide)' 'mirror' is a network of mirrors providing reliable world-wide access; other mirrors may provide faster access on a geographically local scale.

ind chooses a row in '*R_HOME*/doc/BioC_mirrors.csv', by number.

Value

None: this function is invoked for its side effect of updating options("BioC_mirror").

See Also

setRepositories, chooseCRANmirror.

---

chooseCRANmirror *Select a CRAN Mirror*

Description

Interact with the user to choose a CRAN mirror.

Usage

```
chooseCRANmirror(graphics = getOption("menu.graphics"), ind = NULL,
                 local.only = FALSE)

getCRANmirrors(all = FALSE, local.only = FALSE)
```

Arguments

graphics	Logical. If true, use a graphical list: on Windows or the macOS GUI use a list box, and on a Unix-alike use a Tk widget if package **tcltk** and an X server are available. Otherwise use a text menu.
ind	Optional numeric value giving which entry to select.
all	Logical, get all known mirrors or only the ones flagged as OK.
local.only	Logical, try to get most recent list from the CRAN master or use file on local disk only.

Details

A list of mirrors is stored in file '*R_HOME*/doc/CRAN_mirrors.csv', but first an on-line list of current mirrors is consulted, and the file copy used only if the on-line list is inaccessible.

chooseCRANmirror is called by a Windows GUI menu item and by contrib.url if it finds the initial dummy value of options("repos").

HTTPS mirrors with mirroring over ssh will be offered in preference to other mirrors (which are listed in a sub-menu).

ind chooses a row in the list of current mirrors, by number. It is best used with local.only = TRUE and row numbers in '*R_HOME*/doc/CRAN_mirrors.csv'.

Value

None for chooseCRANmirror(), this function is invoked for its side effect of updating options("repos").

getCRANmirrors() returns a data frame with mirror information.

See Also

setRepositories, findCRANmirror, chooseBioCmirror, contrib.url.

---

citation *Citing R and R Packages in Publications*

Description

How to cite R and R packages in publications.

Usage

```
citation(package = "base", lib.loc = NULL, auto = NULL)

readCitationFile(file, meta = NULL)
citHeader(...)
citFooter(...)
```

Arguments

package	a character string with the name of a single package. An error occurs if more than one package name is given.
lib.loc	a character vector with path names of R libraries, or the directory containing the source for package, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

auto	a logical indicating whether the default citation auto-generated from the package 'DESCRIPTION' metadata should be used or not, or NULL (default), indicating that a 'CITATION' file is used if it exists, or an object of class "packageDescription" with package metadata (see below).
file	a file name.
meta	a list of package metadata as obtained by packageDescription, or NULL (the default).
...	character strings (which will be pasted).

Details

The R core development team and the very active community of package authors have invested a lot of time and effort in creating R as it is today. Please give credit where credit is due and cite R and R packages when you use them for data analysis.

Execute function citation() for information on how to cite the base R system in publications. If the name of a non-base package is given, the function either returns the information contained in the 'CITATION' file of the package (using readCitationFile with meta equal to packageDescription(package, lib.loc)) or auto-generates citation information from the 'DESCRIPTION' file.

Packages can use an 'Authors@R' field in their 'DESCRIPTION' to provide (R code giving) a person object with a refined, machine-readable description of the package "authors" (in particular specifying their precise roles). Only those with an author role will be included in the auto-generated citation.

If the object returned by citation() contains only one reference, the associated print method shows both a text version and a BibTeX entry for it. If a package has more than one reference then only the text versions are shown. This threshold is controlled by options("citation.bibtex.max"). The BibTeX versions can also be obtained using function toBibtex() (see the examples below).

The 'CITATION' file of an R package should be placed in the 'inst' subdirectory of the package source. The file is an R source file and may contain arbitrary R commands including conditionals and computations. Function readCitationFile() is used by citation() to extract the information in 'CITATION' files. The file is source()ed by the R parser in a temporary environment and all resulting bibliographic objects (specifically, inheriting from "bibentry") are collected. These are typically produced by one or more bibentry() calls, optionally preceded by a citHeader() and followed by a citFooter() call. One can include an auto-generated package citation in the 'CITATION' file via citation(auto = meta).

readCitationFile makes use of the Encoding element (if any) of meta to determine the encoding of the file.

Value

An object of class "citation", inheriting from class "bibentry"; see there, notably for the print and format methods.

citHeader and citFooter return an empty "bibentry" storing "outer" header/footer text for the package citation.

See Also

bibentry

Examples

```
## the basic R reference
citation()

## extract the BibTeX entry from the return value
x <- citation()
toBibtex(x)

## references for a package
citation("lattice")
citation("lattice", auto = TRUE)  # request the Manual-type reference
citation("foreign")

## a CITATION file with more than one bibentry:
file.show(system.file("CITATION", package="mgcv"))
cm <- citation("mgcv")
cm # header, text references, plus "reminder" about getting BibTeX
print(cm, bibtex = TRUE) # each showing its bibtex code

## a CITATION file including citation(auto = meta)
file.show(system.file("CITATION", package="nlme"))
citation("nlme")
```

---

cite *Cite a Bibliography Entry*

Description

Cite a bibentry object in text. The cite() function uses the cite() function from the default bibstyle if present, or citeNatbib() if not. citeNatbib() uses a style similar to that used by the LaTeX package **natbib**.

Usage

```
cite(keys, bib, ...)
citeNatbib(keys, bib, textual = FALSE, before = NULL, after = NULL,
           mode = c("authoryear", "numbers", "super"),
           abbreviate = TRUE, longnamesfirst = TRUE,
           bibpunct = c("(", ")", ";", "a", "", ","), previous)
```

Arguments

keys	A character vector of keys of entries to cite. May contain multiple keys in a single entry, separated by commas.
bib	A "bibentry" object containing the list of documents in which to find the keys.
...	Additional arguments to pass to the cite() function for the default style.
textual	Produce a "textual" style of citation, i.e. what '\citet' would produce in LaTeX.
before	Optional text to display before the citation.
after	Optional text to display after the citation.
mode	The "mode" of citation.
abbreviate	Whether to abbreviate long author lists.
longnamesfirst	If abbreviate == TRUE, whether to leave the first citation long.
bibpunct	A vector of punctuation to use in the citation, as used in **natbib**. See the Details section.
previous	A list of keys that have been previously cited, to be used when abbreviate == TRUE and longnamesfirst == TRUE

Details

Argument names are chosen based on the documentation for the LaTeX **natbib** package. See that documentation for the interpretation of the bibpunct entries.

The entries in bibpunct are as follows:

1. The left delimiter.
2. The right delimiter.
3. The separator between references within a citation.
4. An indicator of the "mode": "n" for numbers, "s" for superscripts, anything else for author-year.
5. Punctuation to go between the author and year.
6. Punctuation to go between years when authorship is suppressed.

Note that if mode is specified, it overrides the mode specification in bibpunct[4]. Partial matching is used for mode.

The defaults for citeNatbib have been chosen to match the JSS style, and by default these are used in cite. See bibstyle for how to set a different default style.

Value

A single element character string is returned, containing the citation.

Author(s)

Duncan Murdoch

Examples

```
## R reference
rref <- bibentry(
   bibtype = "Manual",
   title = "R: A Language and Environment for Statistical Computing",
   author = person("R Core Team"),
   organization = "R Foundation for Statistical Computing",
   address = "Vienna, Austria",
   year = 2013,
   url = "https://www.R-project.org/",
   key = "R")

## References for boot package and associated book
bref <- c(
   bibentry(
     bibtype = "Manual",
     title = "boot: Bootstrap R (S-PLUS) Functions",
     author = c(
       person("Angelo", "Canty", role = "aut",
         comment = "S original"),
       person(c("Brian", "D."), "Ripley", role = c("aut", "trl", "cre"),
         comment = "R port, author of parallel support",
         email = "ripley@stats.ox.ac.uk")
     ),
     year = "2012",
     note = "R package version 1.3-4",
     url = "https://CRAN.R-project.org/package=boot",
     key = "boot-package"
   ),

   bibentry(
     bibtype = "Book",
     title = "Bootstrap Methods and Their Applications",
     author = as.person("Anthony C. Davison [aut], David V. Hinkley [aut]"),
     year = "1997",
     publisher = "Cambridge University Press",
     address = "Cambridge",
     isbn = "0-521-57391-2",
     url = "http://statwww.epfl.ch/davison/BMA/",
     key = "boot-book"
   )
)

## Combine and cite
refs <- c(rref, bref)
cite("R, boot-package", refs)

## Cite numerically
savestyle <- tools::getBibstyle()
tools::bibstyle("JSSnumbered", .init = TRUE,
         fmtPrefix = function(paper) paste0("[", paper$.index, "]"),
         cite = function(key, bib, ...)
```

```
           citeNatbib(key, bib, mode = "numbers",
               bibpunct = c("[", "]", ";", "n", "", ","), ...)
         )
cite("R, boot-package", refs, textual = TRUE)
refs

## restore the old style
tools::bibstyle(savestyle, .default = TRUE)
```

---

citEntry *Bibliography Entries (Older Interface)*

Description

Old interface providing functionality for specifying bibliographic information in enhanced BibTeX style. Since R 2.14.0 this has been superseded by bibentry.

Usage

```
citEntry(entry, textVersion = NULL, header = NULL, footer = NULL, ...)
```

Arguments

entry	a character string with a BibTeX entry type. See section **Entry Types** in bibentry for details.
textVersion	a character string with a text representation of the reference to optionally be employed for printing.
header	a character string with optional header text.
footer	a character string with optional footer text.
...	for citEntry, arguments of the form *tag=value* giving the fields of the entry, with *tag* and *value* the name and value of the field, respectively. See section **Entry Fields** in bibentry for details.

Value

citEntry produces an object of class "bibentry".

See Also

citation for more information about citing R and R packages and 'CITATION' files; bibentry for the newer functionality for representing and manipulating bibliographic information.

clipboard *Read/Write to/from the Clipboard in MS Windows*

Description

Transfer text between a character vector and the Windows clipboard in MS Windows (only).

Usage

```
getClipboardFormats(numeric = FALSE)
readClipboard(format = 13, raw = FALSE)
writeClipboard(str, format = 13)
```

Arguments

`numeric`	logical: should the result be in human-readable form (the default) or raw numbers?
`format`	an integer giving the desired format.
`raw`	should the value be returned as a raw vector rather than as a character vector?
`str`	a character vector or a raw vector.

Details

The Windows clipboard offers data in a number of formats: see e.g. https://docs.microsoft.com/en-gb/windows/desktop/dataxchg/clipboard-formats.

The standard formats include

CF_TEXT	1	Text in the machine's locale
CF_BITMAP	2	
CF_METAFILEPICT	3	Metafile picture
CF_SYLK	4	Symbolic link
CF_DIF	5	Data Interchange Format
CF_TIFF	6	Tagged-Image File Format
CF_OEMTEXT	7	Text in the OEM codepage
CF_DIB	8	Device-Independent Bitmap
CF_PALETTE	9	
CF_PENDATA	10	
CF_RIFF	11	Audio data
CF_WAVE	12	Audio data
CF_UNICODETEXT	13	Text in Unicode (UCS-2)
CF_ENHMETAFILE	14	Enhanced metafile
CF_HDROP	15	Drag-and-drop data
CF_LOCALE	16	Locale for the text on the clipboard
CF_MAX	17	Shell-oriented formats

Applications normally make data available in one or more of these and possibly additional private formats. Use `raw = TRUE` to read binary formats, `raw = FALSE` (the default) for text formats. The current codepage is used to convert text to Unicode text, and information on that is contained in the `CF_LOCALE` format. (Take care if you are running R in a different locale from Windows. It is recommended to read as Unicode text, so that Windows does the conversion based on `CF_LOCALE`, if available.)

The `writeClipboard` function will write a character vector as text or Unicode text with standard CRLF line terminators. It will copy a raw vector directly to the clipboard without any changes. It is recommended to use Unicode text (the default) instead of text to avoid interoperability problems. (Note that R 4.2 and newer on recent systems uses UTF-8 as the native encoding but the machine's locale uses a different encoding.)

Value

For `getClipboardFormats`, a character or integer vector of available formats, in numeric order. If non human-readable character representation is known, the number is returned.

For `readClipboard`, a character vector by default, a raw vector if `raw` is `TRUE`, or `NULL`, if the format is unavailable.

For `writeClipboard` an invisible logical indicating success or failure.

Note

This is only available on Windows.

See Also

`file` which can be used to set up a connection to a clipboard.

---

close.socket *Close a Socket*

Description

Closes the socket and frees the space in the file descriptor table. The port may not be freed immediately.

Usage

```
close.socket(socket, ...)
```

Arguments

`socket`	a `socket` object
`...`	further arguments passed to or from other methods.

Value

logical indicating success or failure

Author(s)

Thomas Lumley

See Also

make.socket, read.socket

Compiling in support for sockets was optional prior to R 3.3.0: see capabilities("sockets") to see if it is available.

---

combn *Generate All Combinations of n Elements, Taken m at a Time*

Description

Generate all combinations of the elements of x taken m at a time. If x is a positive integer, returns all combinations of the elements of seq(x) taken m at a time. If argument FUN is not NULL, applies a function given by the argument to each point. If simplify is FALSE, returns a list; otherwise returns an array, typically a matrix. ... are passed unchanged to the FUN function, if specified.

Usage

```
combn(x, m, FUN = NULL, simplify = TRUE, ...)
```

Arguments

x	vector source for combinations, or integer n for x <- seq_len(n).
m	number of elements to choose.
FUN	function to be applied to each combination; default NULL means the identity, i.e., to return the combination (vector of length m).
simplify	logical indicating if the result should be simplified to an array (typically a matrix); if FALSE, the function returns a list. Note that when simplify = TRUE as by default, the dimension of the result is simply determined from FUN(*1st combination*) (for efficiency reasons). This will badly fail if FUN(u) is not of constant length.
...	optionally, further arguments to FUN.

Details

Factors x are accepted.

Value

A list or array, see the simplify argument above. In the latter case, the identity dim(combn(n, m)) == c(m, choose(n, m)) holds.

Author(s)

Scott Chasalow wrote the original in 1994 for S; R package **combinat** and documentation by Vince Carey <stvjc@channing.harvard.edu>; small changes by the R core team, notably to return an array in all cases of `simplify = TRUE`, e.g., for `combn(5,5)`.

References

Nijenhuis, A. and Wilf, H.S. (1978) *Combinatorial Algorithms for Computers and Calculators*; Academic Press, NY.

See Also

`choose` for fast computation of the *number* of combinations. `expand.grid` for creating a data frame from all combinations of factors or vectors.

Examples

```
combn(letters[1:4], 2)
(m <- combn(10, 5, min))   # minimum value in each combination
mm <- combn(15, 6, function(x) matrix(x, 2, 3))
stopifnot(round(choose(10, 5)) == length(m), is.array(m), # 1-dimensional
          c(2,3, round(choose(15, 6))) == dim(mm))

## Different way of encoding points:
combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate, nbins = 4)

## Compute support points and (scaled) probabilities for a
## Multivariate-Hypergeometric(n = 3, N = c(4,3,2,1)) p.f.:
# table.mat(t(combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate, nbins = 4)))

## Assuring the identity
for(n in 1:7)
 for(m in 0:n) stopifnot(is.array(cc <- combn(n, m)),
                         dim(cc) == c(m, choose(n, m)),
                         identical(cc, combn(n, m, identity)) || m == 1)
```

---

compareVersion *Compare Two Package Version Numbers*

Description

Compare two package version numbers to see which is later.

Usage

```
compareVersion(a, b)
```

Arguments

a, b	Character strings representing package version numbers.

Details

R package version numbers are of the form `x.y-z` for integers `x`, `y` and `z`, with components after `x` optionally missing (in which case the version number is older than those with the components present).

Value

`0` if the numbers are equal, `-1` if `b` is later and `1` if `a` is later (analogous to the C function `strcmp`).

See Also

`package_version`, `library`, `packageStatus`.

Examples

```
compareVersion("1.0", "1.0-1")
compareVersion("7.2-0","7.1-12")
```

COMPILE *Compile Files for Use with R on Unix-alikes*

Description

Compile given source files so that they can subsequently be collected into a shared object using `R CMD SHLIB` or an executable program using `R CMD LINK`. Not available on Windows.

Usage

```
R CMD COMPILE [options] srcfiles
```

Arguments

`srcfiles`	A list of the names of source files to be compiled. Currently, C, C++, Objective C, Objective C++ and Fortran are supported; the corresponding files should have the extensions '`.c`', '`.cc`' (or '`.cpp`'), '`.m`', '`.mm`' (or '`.M`'), '`.f`' and '`.f90`' or '`.f95`', respectively.
`options`	A list of compile-relevant settings, or for obtaining information about usage and version of the utility.

Details

R CMD SHLIB can both compile and link files into a shared object: since it knows what run-time libraries are needed when passed C++, Fortran and Objective C(++) sources, passing source files to R CMD SHLIB is more reliable.

Objective C and Objective C++ support is optional and will work only if the corresponding compilers were available at R configure time: their main usage is on macOS.

Compilation arranges to include the paths to the R public C/C++ headers.

As this compiles code suitable for incorporation into a shared object, it generates PIC code: that might occasionally be undesirable for the main code of an executable program.

This is a make-based facility, so will not compile a source file if a newer corresponding '.o' file is present.

Note

Some binary distributions of R have COMPILE in a separate bundle, e.g. an R-devel RPM.

This is not available on Windows.

See Also

LINK, SHLIB, dyn.load

The section on "Customizing package compilation" in the 'R Administration and Installation' manual: RShowDoc("R-admin").

---

contrib.url *Find Appropriate Paths in CRAN-like Repositories*

Description

contrib.url adds the appropriate type-specific path within a repository to each URL in repos.

Usage

```
contrib.url(repos, type = getOption("pkgType"))
```

Arguments

repos	character vector, the base URL(s) of the repositories to use.
type	character string, indicating which type of packages: see install.packages.

Details

If type = "both" this will use the source repository.

Value

A character vector of the same length as repos.

See Also

setRepositories to set options("repos"), the most common value used for argument repos.

available.packages, download.packages, install.packages.

The 'R Installation and Administration' manual for how to set up a repository.

---

count.fields *Count the Number of Fields per Line*

Description

count.fields counts the number of fields, as separated by sep, in each of the lines of file read.

Usage

```
count.fields(file, sep = "", quote = "\"'", skip = 0,
             blank.lines.skip = TRUE, comment.char = "#")
```

Arguments

file	a character string naming an ASCII data file, or a connection, which will be opened if necessary, and if so closed at the end of the function call.
sep	the field separator character. Values on each line of the file are separated by this character. By default, arbitrary amounts of whitespace can separate fields.
quote	the set of quoting characters
skip	the number of lines of the data file to skip before beginning to read data.
blank.lines.skip	logical: if TRUE blank lines in the input are ignored.
comment.char	character: a character vector of length one containing a single character or an empty string.

Details

This used to be used by read.table and can still be useful in discovering problems in reading a file by that function.

For the handling of comments, see scan.

Consistent with scan, count.fields allows quoted strings to contain newline characters. In such a case the starting line will have the field count recorded as NA, and the ending line will include the count of all fields from the beginning of the record.

Value

A vector with the numbers of fields found.

See Also

read.table

Examples

```
fil <- tempfile()
cat("NAME", "1:John", "2:Paul", file = fil, sep = "\n")
count.fields(fil, sep = ":")
unlink(fil)
```

---

create.post *Ancillary Function for Preparing Emails and Postings*

Description

An ancillary function used by bug.report and help.request to prepare emails for submission to package maintainers or to R mailing lists.

Usage

```
create.post(instructions = character(), description = "post",
            subject = "",
            method = getOption("mailer"),
            address = "the relevant mailing list",
            ccaddress = getOption("ccaddress", ""),
            filename = "R.post", info = character())
```

Arguments

instructions	Character vector of instructions to put at the top of the template email.
description	Character string: a description to be incorporated into messages.
subject	Subject of the email. Optional except for the "mailx" method.
method	Submission method, one of "none", "mailto", "gnudoit", "ess" or (Unix only) "mailx". See 'Details'.
address	Recipient's email address, where applicable: for package bug reports sent by email this defaults to the address of the package maintainer (the first if more than one is listed).
ccaddress	Optional email address for copies with the "mailx" and "mailto" methods. Use ccaddress = "" for no copy.
filename	Filename to use for setting up the email (or storing it when method is "none" or sending mail fails).
info	character vector of information to include in the template email below the 'please do not edit the information below' line.

Details

What this does depends on the `method`. The function first creates a template email body.

`none` A file editor (see `file.edit`) is opened with instructions and the template email. When this returns, the completed email is in file `file` ready to be read/pasted into an email program.

`mailto` This opens the default email program with a template email (including address, Cc: address and subject) for you to edit and send.

This works where default mailers are set up (usual on macOS and Windows, and where `xdg-open` is available and configured on other Unix-alikes: if that fails it tries the browser set by `R_BROWSER`).

This is the 'factory-fresh' default method.

`mailx` (Unix-alikes only.) A file editor (see `file.edit`) is opened with instructions and the template email. When this returns, it is mailed using a Unix command line mail utility such as `mailx`, to the address (and optionally, the Cc: address) given.

`gnudoit` An (X)emacs mail buffer is opened for the email to be edited and sent: this requires the `gnudoit` program to be available. Currently `subject` is ignored.

`ess` The body of the template email is sent to `stdout`.

Value

Invisible `NULL`.

See Also

`bug.report`, `help.request`.

---

data *Data Sets*

Description

Loads specified data sets, or list the available data sets.

Usage

```
data(..., list = character(), package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"), envir = .GlobalEnv,
     overwrite = TRUE)
```

Arguments

`...`	literal character strings or names.
`list`	a character vector.
`package`	a character vector giving the package(s) to look in for data sets, or `NULL`. By default, all packages in the search path are used, then the '`data`' subdirectory (if present) of the current working directory.

`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.
`envir`	the environment where the data should be loaded.
`overwrite`	logical: should existing objects of the same name in `envir` be replaced?

Details

Currently, four formats of data files are supported:

1. files ending '`.R`' or '`.r`' are `source()`d in, with the R working directory changed temporarily to the directory containing the respective file. (`data` ensures that the **utils** package is attached, in case it had been run *via* `utils::data`.)
2. files ending '`.RData`' or '`.rda`' are `load()`ed.
3. files ending '`.tab`', '`.txt`' or '`.TXT`' are read using `read.table(..., header = TRUE, as.is=FALSE)`, and hence result in a data frame.
4. files ending '`.csv`' or '`.CSV`' are read using `read.table(..., header = TRUE, sep = ";", as.is=FALSE)`, and also result in a data frame.

If more than one matching file name is found, the first on this list is used. (Files with extensions '`.txt`', '`.tab`' or '`.csv`' can be compressed, with or without further extension '`.gz`', '`.bz2`' or '`.xz`'.)

The data sets to be loaded can be specified as a set of character strings or names, or as the character vector `list`, or as both.

For each given data set, the first two types ('`.R`' or '`.r`', and '`.RData`' or '`.rda`' files) can create several variables in the load environment, which might all be named differently from the data set. The third and fourth types will always result in the creation of a single variable with the same name (without extension) as the data set.

If no data sets are specified, `data` lists the available data sets. For each package, it looks for a data index in the '`Meta`' subdirectory or, if this is not found, scans the '`data`' subdirectory for data files using `list_files_with_type`. The information about available data sets is returned in an object of class `"packageIQR"`. The structure of this class is experimental. Where the datasets have a different name from the argument that should be used to retrieve them the index will have an entry like `beaver1 (beavers)` which tells us that dataset `beaver1` can be retrieved by the call `data(beavers)`.

If `lib.loc` and `package` are both `NULL` (the default), the data sets are searched for in all the currently loaded packages then in the '`data`' directory (if any) of the current working directory.

If `lib.loc = NULL` but `package` is specified as a character vector, the specified package(s) are searched for first amongst loaded packages and then in the default libraries (see `.libPaths`).

If `lib.loc` *is* specified (and not `NULL`), packages are searched for in the specified libraries, even if they are already loaded from another library.

To just look in the '`data`' directory of the current working directory, set `package = character(0)` (and `lib.loc = NULL`, the default).

Value

A character vector of all data sets specified (whether found or not), or information about all available data sets in an object of class `"packageIQR"` if none were specified.

Good practice

There is no requirement for `data(`*foo*`)` to create an object named *foo* (nor to create one object), although it much reduces confusion if this convention is followed (and it is enforced if datasets are lazy-loaded).

`data()` was originally intended to allow users to load datasets from packages for use in their examples, and as such it loaded the datasets into the workspace `.GlobalEnv`. This avoided having large datasets in memory when not in use: that need has been almost entirely superseded by lazy-loading of datasets.

The ability to specify a dataset by name (without quotes) is a convenience: in programming the datasets should be specified by character strings (with quotes).

Use of `data` within a function without an `envir` argument has the almost always undesirable side-effect of putting an object in the user's workspace (and indeed, of replacing any object of that name already there). It would almost always be better to put the object in the current evaluation environment by `data(..., envir = environment())`. However, two alternatives are usually preferable, both described in the 'Writing R Extensions' manual.

- For sets of data, set up a package to use lazy-loading of data.
- For objects which are system data, for example lookup tables used in calculations within the function, use a file '`R/sysdata.rda`' in the package sources or create the objects by R code at package installation time.

A sometimes important distinction is that the second approach places objects in the namespace but the first does not. So if it is important that the function sees `mytable` as an object from the package, it is system data and the second approach should be used. In the unusual case that a package uses a lazy-loaded dataset as a default argument to a function, that needs to be specified by `::`, e.g., `survival::survexp.us`.

Warning

This function creates objects in the `envir` environment (by default the user's workspace) replacing any which already existed. `data("foo")` can silently create objects other than `foo`: there have been instances in published packages where it created/replaced `.Random.seed` and hence change the seed for the session.

Note

One can take advantage of the search order and the fact that a '`.R`' file will change directory. If raw data are stored in '`mydata.txt`' then one can set up '`mydata.R`' to read '`mydata.txt`' and pre-process it, e.g., using `transform()`. For instance one can convert numeric vectors to factors with the appropriate labels. Thus, the '`.R`' file can effectively contain a metadata specification for the plaintext formats.

See Also

help for obtaining documentation on data sets, save for *creating* the second ('.rda') kind of data, typically the most efficient one.

The 'Writing R Extensions' manual for considerations in preparing the 'data' directory of a package.

Examples

```
require(utils)
data()                          # list all available data sets
try(data(package = "rpart"), silent = TRUE) # list the data sets in the rpart package
data(USArrests, "VADeaths")     # load the data sets 'USArrests' and 'VADeaths'
## Not run: ## Alternatively
ds <- c("USArrests", "VADeaths"); data(list = ds)
## End(Not run)
help(USArrests)                 # give information on data set 'USArrests'
```

---

dataentry *Spreadsheet Interface for Entering Data*

Description

A spreadsheet-like editor for entering or editing data.

Usage

```
data.entry(..., Modes = NULL, Names = NULL)
dataentry(data, modes)
de(..., Modes = list(), Names = NULL)
```

Arguments

...	A list of variables: currently these should be numeric or character vectors or list containing such vectors.
Modes	The modes to be used for the variables.
Names	The names to be used for the variables.
data	A list of numeric and/or character vectors.
modes	A list of length up to that of data giving the modes of (some of) the variables. list() is allowed.

Details

The data entry editor is only available on some platforms and GUIs. Where available it provides a means to visually edit a matrix or a collection of variables (including a data frame) as described in the Notes section.

`data.entry` has side effects, any changes made in the spreadsheet are reflected in the variables. Function `de` and the internal functions `de.ncols`, `de.setup` and `de.restore` are designed to help achieve these side effects. If the user passes in a matrix, `X` say, then the matrix is broken into columns before `dataentry` is called. Then on return the columns are collected and glued back together and the result assigned to the variable `X`. If you don't want this behaviour use `dataentry` directly.

The primitive function is `dataentry`. It takes a list of vectors of possibly different lengths and modes (the second argument) and opens a spreadsheet with these variables being the columns. The columns of the data entry window are returned as vectors in a list when the spreadsheet is closed.

`de.ncols` counts the number of columns which are supplied as arguments to `data.entry`. It attempts to count columns in lists, matrices and vectors. `de.setup` sets things up so that on return the columns can be regrouped and reassigned to the correct name. This is handled by `de.restore`.

Value

`de` and `dataentry` return the edited value of their arguments. `data.entry` invisibly returns a vector of variable names but its main value is its side effect of assigning new version of those variables in the user's workspace.

Resources

The data entry window responds to X resources of class `R_dataentry`. Resources `foreground`, `background` and `geometry` are utilized.

Note

The details of interface to the data grid may differ by platform and GUI. The following description applies to the X11-based implementation under Unix.

You can navigate around the grid using the cursor keys or by clicking with the (left) mouse button on any cell. The active cell is highlighted by thickening the surrounding rectangle. Moving to the right or down will scroll the grid as needed: there is no constraint to the rows or columns currently in use.

There are alternative ways to navigate using the keys. Return and (keypad) Enter and LineFeed all move down. Tab moves right and Shift-Tab move left. Home moves to the top left.

PageDown or Control-F moves down a page, and PageUp or Control-B up by a page. End will show the last used column and the last few rows used (in any column).

Using any other key starts an editing process on the currently selected cell: moving away from that cell enters the edited value whereas Esc cancels the edit and restores the previous value. When the editing process starts the cell is cleared. In numerical columns (the default) only letters making up a valid number (including `-.eE`) are accepted, and entering an invalid edited value (such as blank) enters `NA` in that cell. The last entered value can be deleted using the BackSpace or Del(ete) key. Only a limited number of characters (currently 29) can be entered in a cell, and if necessary only

the start or end of the string will be displayed, with the omissions indicated by > or <. (The start is shown except when editing.)

Entering a value in a cell further down a column than the last used cell extends the variable and fills the gap (if any) by NAs (not shown on screen).

The column names can only be selected by clicking in them. This gives a popup menu to select the column type (currently Real (numeric) or Character) or to change the name. Changing the type converts the current contents of the column (and converting from Character to Real may generate NAs.) If changing the name is selected the header cell becomes editable (and is cleared). As with all cells, the value is entered by moving away from the cell by clicking elsewhere or by any of the keys for moving down (only).

New columns are created by entering values in them (and not by just assigning a new name). The mode of the column is auto-detected from the first value entered: if this is a valid number it gives a numeric column. Unused columns are ignored, so adding data in `var5` to a three-column grid adds one extra variable, not two.

The Copy button copies the currently selected cell: `paste` copies the last copied value to the current cell, and right-clicking selects a cell *and* copies in the value. Initially the value is blank, and attempts to paste a blank value will have no effect.

Control-L will refresh the display, recalculating field widths to fit the current entries.

In the default mode the column widths are chosen to fit the contents of each column, with a default of 10 characters for empty columns. you can specify fixed column widths by setting option `de.cellwidth` to the required fixed width (in characters). (set it to zero to return to variable widths). The displayed width of any field is limited to 600 pixels (and by the window width).

See Also

`vi`, `edit`: `edit` uses `dataentry` to edit data frames.

Examples

```
# call data entry with variables x and y
## Not run: data.entry(x, y)
```

---

debugcall *Debug a Call*

Description

Set or unset debugging flags based on a call to a function. Takes into account S3/S4 method dispatch based on the classes of the arguments in the call.

Usage

```
debugcall(call, once = FALSE)
undebugcall(call)
```

Arguments

call	An R expression calling a function. The called function will be debugged. See Details.
once	logical; if TRUE, debugging only occurs once, as via debugonce. Defaults to FALSE

Details

debugcall debugs the non-generic function, S3 method or S4 method that would be called by evaluating call. Thus, the user does not need to specify the signature when debugging methods. Although the call is actually to the generic, it is the method that is debugged, not the generic, except for non-standard S3 generics (see isS3stdGeneric).

Value

debugcall invisibly returns the debugged call expression.

Note

Non-standard evaluation is used to retrieve the call (via substitute). For this reason, passing a variable containing a call expression, rather than the call expression itself, will not work.

See Also

debug for the primary debugging interface

Examples

```
## Not run:
## Evaluate call after setting debugging
##
f <- factor(1:10)
res <- eval(debugcall(summary(f)))

## End(Not run)
```

---

debugger *Post-Mortem Debugging*

Description

Functions to dump the evaluation environments (frames) and to examine dumped frames.

Usage

```
dump.frames(dumpto = "last.dump", to.file = FALSE,
            include.GlobalEnv = FALSE)
debugger(dump = last.dump)

limitedLabels(value, maxwidth = getOption("width") - 5L)
```

Arguments

dumpto	a character string. The name of the object or file to dump to.
to.file	logical. Should the dump be to an R object or to a file?
include.GlobalEnv	logical indicating if a *copy* of the .GlobalEnv environment should be included in addition to the sys.frames(). Will be particularly useful when used in a batch job.
dump	an R dump object created by dump.frames.
value	a list of calls to be formatted, e.g., for user menus.
maxwidth	optional length to which to trim the result of limitedLabels(); values smaller than 40 or larger than 1000 are winsorized.

Details

To use post-mortem debugging, set the option error to be a call to dump.frames. By default this dumps to an R object last.dump in the workspace, but it can be set to dump to a file (a dump of the object produced by a call to save). The dumped object contain the call stack, the active environments and the last error message as returned by geterrmessage.

When dumping to file, dumpto gives the name of the dumped object and the file name has '.rda' appended.

A dump object of class "dump.frames" can be examined by calling debugger. This will give the error message and a list of environments from which to select repeatedly. When an environment is selected, it is copied and the browser called from within the copy. Note that not all the information in the original frame will be available, e.g. promises which have not yet been evaluated and the contents of any ... argument.

If dump.frames is installed as the error handler, execution will continue even in non-interactive sessions. See the examples for how to dump and then quit.

limitedLabels(v) takes a list of calls whose elements may have a srcref attribute and returns a vector that pastes a formatted version of those attributes onto the formatted version of the elements, all finally strtrim()med to maxwidth.

Value

Invisible NULL.

Note

Functions such as sys.parent and environment applied to closures will not work correctly inside debugger.

If the error occurred when computing the default value of a formal argument the debugger will report "recursive default argument reference" when trying to examine that environment.

Of course post-mortem debugging will not work if R is too damaged to produce and save the dump, for example if it has run out of workspace.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

browser for the actions available at the Browse prompt.

options for setting error options; recover is an interactive debugger working similarly to debugger but directly after the error occurs.

Examples

```
## Not run:
options(error = quote(dump.frames("testdump", TRUE)))

f <- function() {
    g <- function() stop("test dump.frames")
    g()
}
f()   # will generate a dump on file "testdump.rda"
options(error = NULL)

## possibly in another R session
load("testdump.rda")
debugger(testdump)
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
f()
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "g"
Browse[1]> g
function() stop("test dump.frames")
<environment: 759818>
Browse[1]>
```

```
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 0

## A possible setting for non-interactive sessions
options(error = quote({dump.frames(to.file = TRUE); q(status = 1)}))

## End(Not run)
```

---

demo *Demonstrations of R Functionality*

Description

demo is a user-friendly interface to running some demonstration R scripts. demo() gives the list of available topics.

Usage

```
demo(topic, package = NULL, lib.loc = NULL,
     character.only = FALSE, verbose = getOption("verbose"),
     type = c("console", "html"), echo = TRUE,
     ask = getOption("demo.ask"),
     encoding = getOption("encoding"))
```

Arguments

topic	the topic which should be demonstrated, given as a name or literal character string, or a character string, depending on whether character.only is FALSE (default) or TRUE. If omitted, the list of available topics is displayed.
package	a character vector giving the packages to look into for demos, or NULL. By default, all packages in the search path are used.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
character.only	logical; if TRUE, use topic as character string.
verbose	a logical. If TRUE, additional diagnostics are printed.
type	character: whether to show output in the console or a browser (using the dynamic help system). The latter is honored only in interactive sessions and if the knitr package is installed. Several other arguments are silently ignored in that case, including lib.loc.
echo	a logical. If TRUE, show the R input when sourcing.

ask	a logical (or "default") indicating if devAskNewPage(ask = TRUE) should be called before graphical output happens from the demo code. The value "default" (the factory-fresh default) means to ask if echo == TRUE and the graphics device appears to be interactive. This parameter applies both to any currently opened device and to any devices opened by the demo code. If this is evaluated to TRUE and the session is interactive, the user is asked to press RETURN to start.
encoding	See source. If the package has a declared encoding, that takes preference.

Details

If no topics are given, demo lists the available demos. For type = "console", the corresponding information is returned in an object of class "packageIQR".

See Also

source and devAskNewPage which are called by demo. example to run code in the Examples section of help pages.

Examples

```
demo() # for attached packages

## All available demos:
demo(package = .packages(all.available = TRUE))


## Display a demo, pausing between pages
demo(lm.glm, package = "stats", ask = TRUE)

## Display it without pausing
demo(lm.glm, package = "stats", ask = FALSE)


## Not run:
 ch <- "scoping"
 demo(ch, character = TRUE)

## End(Not run)

## Find the location of a demo
system.file("demo", "lm.glm.R", package = "stats")
```

DLL.version *DLL Version Information on MS Windows*

Description

On MS Windows only, return the version of the package and the version of R used to build the DLL, if available.

Usage

```
DLL.version(path)
```

Arguments

path	character vector of length one giving the complete path to the DLL.

Value

If the DLL does not exist, NULL.

A character vector of length two, giving the DLL version and the version of R used to build the DLL. If the information is not available, the corresponding string is empty.

Note

This is only available on Windows.

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
  DLL.version(file.path(R.home("bin"), "R.dll"))
  DLL.version(file.path(R.home(), "library/stats/libs", .Platform$r_arch, "stats.dll"))
})
```

---

download.file *Download File from the Internet*

Description

This function can be used to download a file from the Internet.

Usage

```
download.file(url, destfile, method, quiet = FALSE, mode = "w",
              cacheOK = TRUE,
              extra = getOption("download.file.extra"),
              headers = NULL, ...)
```

Arguments

url	a character string (or longer vector for the "libcurl" method) naming the URL of a resource to be downloaded.
destfile	a character string (or vector, see the url argument) with the file path where the downloaded file is to be saved. Tilde-expansion is performed.

method	Method to be used for downloading files. Current download methods are "internal", "libcurl", "wget", "curl" and "wininet" (Windows only), and there is a value "auto": see 'Details' and 'Note'. The method can also be set through the option "download.file.method": see options().
quiet	If TRUE, suppress status messages (if any), and the progress bar.
mode	character. The mode with which to write the file. Useful values are "w", "wb" (binary), "a" (append) and "ab". Not used for methods "wget" and "curl". See also 'Details', notably about using "wb" for Windows.
cacheOK	logical. Is a server-side cached value acceptable?
extra	character vector of additional command-line arguments for the "wget" and "curl" methods.
headers	named character vector of additional HTTP headers to use in HTTP[S] requests. It is ignored for non-HTTP[S] URLs. The User-Agent header taken from the HTTPUserAgent option (see options) is automatically used as the first header.
...	allow additional arguments to be passed, unused.

Details

The function download.file can be used to download a single file as described by url from the internet and store it in destfile.

The url must start with a scheme such as 'http://', 'https://' or 'file://'. Which methods support which schemes varies by R version, but method = "auto" will try to find a method which supports the scheme.

For method = "auto" (the default) currently the "internal" method is used for 'file://' URLs and "libcurl" for all others.

Support for method "libcurl" was optional on Windows prior to R 4.2.0: use capabilities("libcurl") to see if it is supported on an earlier version. It uses an external library of that name (https://curl.se/libcurl/) against which R can be compiled.

When method "libcurl" is used, there is support for simultaneous downloads, so url and destfile can be character vectors of the same length greater than one (but the method has to be specified explicitly and not *via* "auto"). For a single URL and quiet = FALSE a progress bar is shown in interactive use.

Nowadays the "internal" method only supports the 'file://' scheme (for which it is the default). On Windows the "wininet" method currently supports 'file://' and (but deprecated with a warning) 'http://' and 'https://' schemes.

For methods "wget" and "curl" a system call is made to the tool given by method, and the respective program must be installed on your system and be in the search path for executables. They will block all other activity on the R process until they complete: this may make a GUI unresponsive.

cacheOK = FALSE is useful for 'http://' and 'https://' URLs: it will attempt to get a copy directly from the site rather than from an intermediate cache. It is used by available.packages.

The "libcurl" and "wget" methods follow 'http://' and 'https://' redirections to any scheme they support. (For method "curl" use argument extra = "-L". To disable redirection in wget, use extra = "--max-redirect=0".) The "wininet" method supports some redirections but not all. (For method "libcurl", messages will quote the endpoint of redirections.)

See url for how 'file://' URLs are interpreted, especially on Windows. The "internal" and "wininet" methods do not percent-decode, but the "libcurl" and "curl" methods do: method "wget" does not support them.

Most methods do not percent-encode special characters such as spaces in URLs (see URLencode), but it seems the "wininet" method does.

The remaining details apply to the "wininet" and "libcurl" methods only.

The timeout for many parts of the transfer can be set by the option timeout which defaults to 60 seconds. This is often insufficient for downloads of large files (50MB or more) and so should be increased when download.file is used in packages to do so. Note that the user can set the default timeout by the environment variable R_DEFAULT_INTERNET_TIMEOUT in recent versions of R, so to ensure that this is not decreased packages should use something like

```
options(timeout = max(300, getOption("timeout")))
```

(It is unrealistic to require download times of less than 1s/MB.)

The level of detail provided during transfer can be set by the quiet argument and the internet.info option: the details depend on the platform and scheme. For the "libcurl" method values of the option less than 2 give verbose output.

A progress bar tracks the transfer platform-specifically:

On Windows If the file length is known, the full width of the bar is the known length. Otherwise the initial width represents 100 Kbytes and is doubled whenever the current width is exceeded. (In non-interactive use this uses a text version. If the file length is known, an equals sign represents 2% of the transfer completed: otherwise a dot represents 10Kb.)

On a Unix-alike If the file length is known, an equals sign represents 2% of the transfer completed: otherwise a dot represents 10Kb.

The choice of binary transfer (mode = "wb" or "ab") is important on Windows, since unlike Unix-alikes it does distinguish between text and binary files and for text transfers changes '\n' line endings to '\r\n' (aka 'CRLF').

On Windows, if mode is not supplied (missing()) and url ends in one of '.gz', '.bz2', '.xz', '.tgz', '.zip', '.jar', '.rda', '.rds', '.RData' or '.pdf', mode = "wb" is set so that a binary transfer is done to help unwary users.

Code written to download binary files must use mode = "wb" (or "ab"), but the problems incurred by a text transfer will only be seen on Windows.

Value

An (invisible) integer code, 0 for success and non-zero for failure. For the "wget" and "curl" methods this is the status code returned by the external program. The "internal" method can return 1, but will in most cases throw an error.

What happens to the destination file(s) in the case of error depends on the method and R version. Currently the "internal", "wininet" and "libcurl" methods will remove the file if the URL is unavailable except when mode specifies appending when the file should be unchanged.

Setting Proxies

For the Windows-only method "wininet", the 'Internet Options' of the system are used to choose proxies and so on; these are set in the Control Panel and are those used for system browsers.

For the "libcurl" and "curl" methods, proxies can be set *via* the environment variables http_proxy or ftp_proxy. See https://curl.se/libcurl/c/libcurl-tutorial.html for further details.

Secure URLs

Methods which access 'https://' and (where supported) 'ftps://' URLs should try to verify the site certificates. This is usually done using the CA root certificates installed by the OS (although we have seen instances in which these got removed rather than updated). For further information see https://curl.se/docs/sslcerts.html.

On Windows with method = "libcurl", the CA root certificates are provided by the OS when R was linked with libcurl with Schannel enabled, which is the current default in Rtools. This can be verified by checking that libcurlVersion() returns a version string containing '"Schannel"'. If it does not, for verification to be on the environment variable CURL_CA_BUNDLE must be set to a path to a certificate bundle file, usually named 'ca-bundle.crt' or 'curl-ca-bundle.crt'. (This is normally done automatically for a binary installation of R, which installs '*R_HOME*/etc/curl-ca-bundle.crt' and sets CURL_CA_BUNDLE to point to it if that environment variable is not already set.) For an updated certificate bundle, see https://curl.se/docs/sslcerts.html. Currently one can download a copy from https://raw.githubusercontent.com/bagder/ca-bundle/master/ca-bundle.crt and set CURL_CA_BUNDLE to the full path to the downloaded file.

On Windows with method = "libcurl", when R was linked with libcurl with Schannel enabled, the connection fails if it cannot be established that the certificate has not been revoked. Some MITM proxies present particularly in corporate environments do not work with this behavior. It can be changed by setting environment variable R_LIBCURL_SSL_REVOKE_BEST_EFFORT to TRUE, with the consequence of reducing security.

Note that the root certificates used by R may or may not be the same as used in a browser, and indeed different browsers may use different certificate bundles (there is typically a build option to choose either their own or the system ones).

Good practice

Setting the method should be left to the end user. Neither of the wget nor curl commands is widely available: you can check if one is available *via* Sys.which, and should do so in a package or script.

If you use download.file in a package or script, you must check the return value, since it is possible that the download will fail with a non-zero status but not an R error.

The supported methods do change: method libcurl was introduced in R 3.2.0 and was optional on Windows until R 4.2.0 – use capabilities("libcurl") in a program to see if it is available.

'ftp://' URLs

Most modern browsers do not support such URLs, and 'https://' ones are much preferred for use in R. 'ftps://' URLs have always been rare, and are nowadays even less supported.

It is intended that R will continue to allow such URLs for as long as `libcurl` does, but as they become rarer this is increasingly untested. What 'protocols' the version of `libcurl` being used supports can be seen by calling `libcurlVersion()`.

These URLs are accessed using the FTP protocol which has a number of variants. One distinction is between 'active' and '(extended) passive' modes: which is used is chosen by the client. The `"libcurl"` method uses passive mode which was almost universally used by browsers before they dropped support altogether.

Note

Files of more than 2GB are supported on 64-bit builds of R; they may be truncated on some 32-bit builds.

Methods `"wget"` and `"curl"` are mainly for historical compatibility but provide may provide capabilities not supported by the `"libcurl"` or `"wininet"` methods.

Method `"wget"` can be used with proxy firewalls which require user/password authentication if proper values are stored in the configuration file for `wget`.

`wget` (`https://www.gnu.org/software/wget/`) is commonly installed on Unix-alikes (but not macOS). Windows binaries are available from MSYS2 and elsewhere.

`curl` (`https://curl.se/`) is installed on macOS and increasingly commonly on Unix-alikes. Windows binaries are available at that URL.

See Also

`options` to set the `HTTPUserAgent`, `timeout` and `internet.info` options used by some of the methods.

`url` for a finer-grained way to read data from URLs.

`url.show`, `available.packages`, `download.packages` for applications.

Contributed packages **RCurl** and **curl** provide more comprehensive facilities to download from URLs.

`download.packages` *Download Packages from CRAN-like Repositories*

Description

These functions can be used to automatically compare the version numbers of installed packages with the newest available version on the repositories and update outdated packages on the fly.

Usage

```
download.packages(pkgs, destdir, available = NULL,
                  repos = getOption("repos"),
                  contriburl = contrib.url(repos, type),
                  method, type = getOption("pkgType"), ...)
```

Arguments

pkgs	character vector of the names of packages whose latest available versions should be downloaded from the repositories.
destdir	directory where downloaded packages are to be stored.
available	an object as returned by available.packages listing packages available at the repositories, or NULL which makes an internal call to available.packages.
repos	character vector, the base URL(s) of the repositories to use, i.e., the URL of the CRAN master such as "https://cran.r-project.org" or ones of its mirrors, "https://cloud.r-project.org".
contriburl	URL(s) of the contrib sections of the repositories. Use this argument only if your repository mirror is incomplete, e.g., because you burned only the 'contrib' section on a CD. Overrides argument repos.
method	Download method, see download.file.
type	character string, indicate which type of packages: see install.packages and 'Details'.
...	additional arguments to be passed to download.file and available.packages.

Details

download.packages takes a list of package names and a destination directory, downloads the newest versions and saves them in destdir. If the list of available packages is not given as argument, it is obtained from repositories. If a repository is local, i.e. the URL starts with "file:", then the packages are not downloaded but used directly. Both "file:" and "file:///" are allowed as prefixes to a file path. Use the latter only for URLs: see url for their interpretation. (Other forms of 'file://' URLs are not supported.)

For download.packages, type = "both" looks at source packages only.

Value

A two-column matrix of names and destination file names of those packages successfully downloaded. If packages are not available or there is a problem with the download, suitable warnings are given.

See Also

available.packages, contrib.url.

The main use is by install.packages.

See download.file for how to handle proxies and other options to monitor file transfers.

The 'R Installation and Administration' manual for how to set up a repository.

edit *Invoke a Text Editor*

Description

Invoke a text editor on an R object.

Usage

```
edit(name, ...)
## Default S3 method:
edit(name = NULL, file = "", title = NULL,
     editor = getOption("editor"), ...)

vi(name = NULL, file = "")
emacs(name = NULL, file = "")
pico(name = NULL, file = "")
xemacs(name = NULL, file = "")
xedit(name = NULL, file = "")
```

Arguments

name	a named object that you want to edit. For the default method, if name is missing then the file specified by file is opened for editing.
file	a string naming the file to write the edited version to.
title	a display name for the object being edited.
editor	usually a character string naming (or giving the path to) the text editor you want to use. On Unix the default is set from the environment variables EDITOR or VISUAL if either is set, otherwise vi is used. On Windows it defaults to "internal", the script editor. On the macOS GUI the argument is ignored and the document editor is always used. editor can also be an R function, in which case it is called with the arguments name, file, and title. Note that such a function will need to independently implement all desired functionality.
...	further arguments to be passed to or from methods.

Details

edit invokes the text editor specified by editor with the object name to be edited. It is a generic function, currently with a default method and one for data frames and matrices.

data.entry can be used to edit data, and is used by edit to edit matrices and data frames on systems for which data.entry is available.

It is important to realize that edit does not change the object called name. Instead, a copy of name is made and it is that copy which is changed. Should you want the changes to apply to the object name you must assign the result of edit to name. (Try fix if you want to make permanent changes to an object.)

In the form `edit(name)`, `edit` deparses `name` into a temporary file and invokes the editor `editor` on this file. Quitting from the editor causes `file` to be parsed and that value returned. Should an error occur in parsing, possibly due to incorrect syntax, no value is returned. Calling `edit()`, with no arguments, will result in the temporary file being reopened for further editing.

Note that deparsing is not perfect, and the object recreated after editing can differ in subtle ways from that deparsed: see `dput` and `.deparseOpts`. (The deparse options used are the same as the defaults for `dump`.) Editing a function will preserve its environment. See `edit.data.frame` for further changes that can occur when editing a data frame or matrix.

Currently only the internal editor in Windows makes use of the `title` option; it displays the given name in the window header.

Note

The functions `vi`, `emacs`, `pico`, `xemacs`, `xedit` rely on the corresponding editor being available and being on the path. This is system-dependent.

See Also

`edit.data.frame`, `data.entry`, `fix`.

Examples

```
## Not run:
# use xedit on the function mean and assign the changes
mean <- edit(mean, editor = "xedit")

# use vi on mean and write the result to file mean.out
vi(mean, file = "mean.out")

## End(Not run)
```

---

edit.data.frame *Edit Data Frames and Matrices*

Description

Use data editor on data frame or matrix contents.

Usage

```
## S3 method for class 'data.frame'
edit(name, factor.mode = c("character", "numeric"),
     edit.row.names = any(row.names(name) != 1:nrow(name)), ...)

## S3 method for class 'matrix'
edit(name, edit.row.names = !is.null(dn[[1]]), ...)
```

Arguments

name	A data frame or (numeric, logical or character) matrix.
factor.mode	How to handle factors (as integers or using character levels) in a data frame. Can be abbreviated.
edit.row.names	logical. Show the row names (if they exist) be displayed as a separate editable column? It is an error to ask for this on a matrix with NULL row names.
...	further arguments passed to or from other methods.

Details

At present, this only works on simple data frames containing numeric, logical or character vectors and factors, and numeric, logical or character matrices. Any other mode of matrix will give an error, and a warning is given when the matrix has a class (which will be discarded).

Data frame columns are coerced on input to *character* unless numeric (in the sense of `is.numeric`), logical or factor. A warning is given when classes are discarded. Special characters (tabs, non-printing ASCII, etc.) will be displayed as escape sequences.

Factors columns are represented in the spreadsheet as either numeric vectors (which are more suitable for data entry) or character vectors (better for browsing). After editing, vectors are padded with NA to have the same length and factor attributes are restored. The set of factor levels can not be changed by editing in numeric mode; invalid levels are changed to NA and a warning is issued. If new factor levels are introduced in character mode, they are added at the end of the list of levels in the order in which they encountered.

It is possible to use the data-editor's facilities to select the mode of columns to swap between numerical and factor columns in a data frame. Changing any column in a numerical matrix to character will cause the result to be coerced to a character matrix. Changing the mode of logical columns is not supported.

For a data frame, the row names will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged, and from the edited output if `edit.row.names = TRUE` and there are no duplicates. (If the `row.names` column is incomplete, it is extended by entries like `row223`.) In all other cases the row names are replaced by `seq(length = nrows)`.

For a matrix, colnames will be added (of the form `col7`) if needed. The rownames will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged (otherwise NULL), and from the edited output if `edit.row.names = TRUE`. (If the `row.names` column is incomplete, it is extended by entries like `row223`.)

Editing a matrix or data frame will lose all attributes apart from the row and column names.

Value

The edited data frame or matrix.

Note

`fix(dataframe)` works for in-place editing by calling this function.

If the data editor is not available, a dump of the object is presented for editing using the default method of `edit`.

At present the data editor is limited to 65535 rows.

Author(s)

Peter Dalgaard

See Also

data.entry, edit

Examples

```
## Not run:
edit(InsectSprays)
edit(InsectSprays, factor.mode = "numeric")

## End(Not run)
```

---

example *Run an Examples Section from the Online Help*

Description

Run all the R code from the **Examples** part of R's online help topic topic, with possible exceptions due to \dontrun, \dontshow, and \donttest tags, see 'Details' below.

Usage

```
example(topic, package = NULL, lib.loc = NULL,
        character.only = FALSE, give.lines = FALSE, local = FALSE,
        type = c("console", "html"), echo = TRUE,
        verbose = getOption("verbose"),
        setRNG = FALSE, ask = getOption("example.ask"),
        prompt.prefix = abbreviate(topic, 6),
        catch.aborts = FALSE,
        run.dontrun = FALSE, run.donttest = interactive())
```

Arguments

topic	name or literal character string: the online help topic the examples of which should be run.
package	a character vector giving the package names to look into for the topic, or NULL (the default), when all packages on the search path are used.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
character.only	a logical indicating whether topic can be assumed to be a character string.
give.lines	logical: if true, the *lines* of the example source code are returned as a character vector.

local	logical: if TRUE evaluate locally, if FALSE evaluate in the workspace.
type	character: whether to show output in the console or a browser (using the dynamic help system). The latter is honored only in interactive sessions and if the knitr package is installed. Several other arguments are silently ignored in that case, including setRNG and lib.loc.
echo	logical; if TRUE, show the R input when sourcing.
verbose	logical; if TRUE, show even more when running example code.
setRNG	logical or expression; if not FALSE, the random number generator state is saved, then initialized to a specified state, the example is run and the (saved) state is restored. setRNG = TRUE sets the same state as R CMD check does for running a package's examples. This is currently equivalent to setRNG = {RNGkind("default", "default", "default"); set.seed(1)}.
ask	logical (or "default") indicating if devAskNewPage(ask = TRUE) should be called before graphical output happens from the example code. The value "default" (the factory-fresh default) means to ask if echo is true and the graphics device appears to be interactive. This parameter applies both to any currently opened device and to any devices opened by the example code.
prompt.prefix	character; prefixes the prompt to be used if echo is true (as it is by default).
catch.aborts	logical, passed on to source(), indicating that "abort"ing errors should be caught.
run.dontrun	logical indicating that \dontrun should be ignored, i.e., do run the enclosed code.
run.donttest	logical indicating that \donttest should be ignored, i.e., do run the enclosed code.

Details

If lib.loc is not specified, the packages are searched for amongst those already loaded, then in the libraries given by .libPaths(). If lib.loc is specified, packages are searched for only in the specified libraries, even if they are already loaded from another library. The search stops at the first package found that has help on the topic.

An attempt is made to load the package before running the examples, but this will not replace a package loaded from another location.

If local = TRUE objects are not created in the workspace and so not available for examination after example completes: on the other hand they cannot overwrite objects of the same name in the workspace.

As detailed in the 'Writing R Extensions' manual, the author of the help page can tag parts of the examples with the following exception rules:

\dontrun encloses code that should not be run.

\dontshow encloses code that is invisible on help pages, but will be run both by the package checking tools, and the example() function. This was previously \testonly, and that form is still accepted.

\donttest encloses code that typically should be run, but not during package checking. The default run.donttest = interactive() leads example() use in other help page examples to skip \donttest sections appropriately.

The additional \dontdiff tag (in R ≥ 4.4.0) produces special comments in the code run by example (for Rdiff-based testing of example output), but does not affect which code is run or displayed on the help page.

Value

The value of the last evaluated expression, unless give.lines is true, where a character vector is returned.

Author(s)

Martin Maechler and others

See Also

demo

Examples

```
example(InsectSprays)
## force use of the standard package 'stats':
example("smooth", package = "stats", lib.loc = .Library)

## set RNG *before* example as when R CMD check is run:

r1 <- example(quantile, setRNG = TRUE)
x1 <- rnorm(1)
u <- runif(1)
## identical random numbers
r2 <- example(quantile, setRNG = TRUE)
x2 <- rnorm(1)
stopifnot(identical(r1, r2))
## but x1 and x2 differ since the RNG state from before example()
## differs and is restored!
x1; x2

## Exploring examples code:
## How large are the examples of "lm...()" functions?
lmex <- sapply(apropos("^lm", mode = "function"),
               example, character.only = TRUE, give.lines = TRUE)
lengths(lmex)
```

---

file.edit *Edit One or More Files*

Description

Edit one or more files in a text editor.

Usage

```
file.edit(..., title = file, editor = getOption("editor"),
          fileEncoding = "")
```

Arguments

`...`	one or more character vectors containing the names of the files to be displayed. These will be tilde-expanded: see `path.expand`.
`title`	the title to use in the editor; defaults to the filename.
`editor`	the text editor to be used, usually as a character string naming (or giving the path to) the text editor you want to use See 'Details'.
`fileEncoding`	the encoding to assume for the file: the default is to assume the native encoding. See the 'Encoding' section of the help for `file`.

Details

The behaviour of this function is very system-dependent. Currently files can be opened only one at a time on Unix; on Windows, the internal editor allows multiple files to be opened, but has a limit of 50 simultaneous edit windows.

The `title` argument is used for the window caption in Windows, and is currently ignored on other platforms.

Any error in re-encoding the files to the native encoding will cause the function to fail.

The default for `editor` is system-dependent. On Windows it defaults to `"internal"`, the script editor, and in the macOS GUI the document editor is used whatever the value of `editor`. On Unix the default is set from the environment variables `EDITOR` or `VISUAL` if either is set, otherwise `vi` is used.

`editor` can also be an R function, in which case it is called with the arguments `name`, `file`, and `title`. Note that such a function will need to independently implement all desired functionality.

On Windows, UTF-8-encoded paths not valid in the current locale can be used.

See Also

`files`, `file.show`, `edit`, `fix`,

Examples

```
## Not run:
# open two R scripts for editing
file.edit("script1.R", "script2.R")

## End(Not run)
```

`file_test` *Shell-style Tests on Files*

Description

Utility for shell-style file tests.

Usage

```
file_test(op, x, y)
```

Arguments

`op`	a character string specifying the test to be performed. Unary tests (only `x` is used) are `"-f"` (existence and not being a directory), `"-d"` (existence and directory), `"-L"` or `"-h"` (existence and symbolic link), `"-x"` (executable as a file or searchable as a directory), `"-w"` (writable) and `"-r"` (readable). Binary tests are `"-nt"` (strictly newer than, using the modification dates) and `"-ot"` (strictly older than): in both cases the test is false unless both files exist.
`x`, `y`	character vectors giving file paths.

Details

'Existence' here means being on the file system and accessible by the `stat` system call (or a 64-bit extension) – on a Unix-alike this requires execute permission on all of the directories in the path that leads to the file, but no permissions on the file itself.

For the meaning of `"-x"` on Windows see `file.access`.

See Also

`file.exists` which only tests for existence (`test -e` on some systems) but not for not being a directory.

`file.path`, `file.info`

Examples

```
dir <- file.path(R.home(), "library", "stats")
file_test("-d", dir)
file_test("-nt", file.path(dir, "R"), file.path(dir, "demo"))
```

findCRANmirror *Find CRAN Mirror Preference*

Description

Find out if a CRAN mirror has been selected for the current session.

Usage

```
findCRANmirror(type = c("src", "web"))
```

Arguments

type	Is the mirror to be used for package sources or web information?

Details

Find out if a CRAN mirror has been selected for the current session. If so, return its URL else return '"https://CRAN.R-project.org"'.

The mirror is looked for in several places.

- The value of the environment variable R_CRAN_SRC or R_CRAN_WEB (depending on type), if set.
- An entry in getOption("repos") named 'CRAN' which is not the default '"@CRAN@")'.
- The 'CRAN' URL entry in the 'repositories' file (see setRepositories), if it is not the default '"@CRAN@"'.

The two types allow for partial local CRAN mirrors, for example those mirroring only the package sources where getOption("repos") might point to the partial mirror and R_CRAN_WEB point to a full (remote) mirror.

Value

A character string.

See Also

setRepositories, chooseCRANmirror

Examples

```
c(findCRANmirror("src"), findCRANmirror("web"))

Sys.setenv(R_CRAN_WEB = "https://cloud.r-project.org")
c(findCRANmirror("src"), findCRANmirror("web"))
```

`findLineNum` *Find the Location of a Line of Source Code, or Set a Breakpoint There*

Description

These functions locate objects containing particular lines of source code, using the information saved when the code was parsed with `keep.source = TRUE`.

Usage

```
findLineNum(srcfile, line, nameonly = TRUE,
            envir = parent.frame(), lastenv)

setBreakpoint(srcfile, line, nameonly = TRUE,
              envir = parent.frame(), lastenv, verbose = TRUE,
              tracer, print = FALSE, clear = FALSE, ...)
```

Arguments

`srcfile`	The name of the file containing the source code.
`line`	The line number within the file. See Details for an alternate way to specify this.
`nameonly`	If `TRUE` (the default), we require only a match to `basename(srcfile)`, not to the full path.
`envir`	Where do we start looking for function objects?
`lastenv`	Where do we stop? See the Details.
`verbose`	Should we print information on where breakpoints were set?
`tracer`	An optional `tracer` function to pass to `trace`. By default, a call to `browser` is inserted.
`print`	The `print` argument to pass to `trace`.
`clear`	If `TRUE`, call `untrace` rather than `trace`.
`...`	Additional arguments to pass to `trace`.

Details

The `findLineNum` function searches through all objects in environment `envir`, its parent, grandparent, etc., all the way back to `lastenv`.

`lastenv` defaults to the global environment if `envir` is not specified, and to the root environment `emptyenv()` if `envir` is specified. (The first default tends to be quite fast, and will usually find all user code other than S4 methods; the second one is quite slow, as it will typically search all attached system libraries.)

For convenience, `envir` may be specified indirectly: if it is not an environment, it will be replaced with `environment(envir)`.

`setBreakpoint` is a simple wrapper function for `trace` and `untrace`. It will set or clear breakpoints at the locations found by `findLineNum`.

The srcfile is normally a filename entered as a character string, but it may be a "srcfile" object, or it may include a suffix like "filename.R#nn", in which case the number nn will be used as a default value for line.

As described in the description of the where argument on the man page for trace, the R package system uses a complicated scheme that may include more than one copy of a function in a package. The user will typically see the public one on the search path, while code in the package will see a private one in the package namespace. If you set envir to the environment of a function in the package, by default findLineNum will find both versions, and setBreakpoint will set the breakpoint in both. (This can be controlled using lastenv; e.g., envir = environment(foo), lastenv = globalenv() will find only the private copy, as the search is stopped before seeing the public copy.)

S version 4 methods are also somewhat tricky to find. They are stored with the generic function, which may be in the **base** or other package, so it is usually necessary to have lastenv = emptyenv() in order to find them. In some cases transformations are done by R when storing them and findLineNum may not be able to find the original code. Many special cases, e.g. methods on primitive generics, are not yet supported.

Value

findLineNum returns a list of objects containing location information. A print method is defined for them.

setBreakpoint has no useful return value; it is called for the side effect of calling trace or untrace.

Author(s)

Duncan Murdoch

See Also

trace

Examples

```
## Not run:
# Find what function was defined in the file mysource.R at line 100:
findLineNum("mysource.R#100")

# Set a breakpoint in both copies of that function, assuming one is in the
# same namespace as myfunction and the other is on the search path
setBreakpoint("mysource.R#100", envir = myfunction)

## End(Not run)
```

`fix` *Fix an Object*

Description

`fix` invokes `edit` on `x` and then assigns the new (edited) version of `x` in the user's workspace.

Usage

```
fix(x, ...)
```

Arguments

`x`	the name of an R object, as a name or a character string.
`...`	arguments to pass to editor: see `edit`.

Details

The name supplied as `x` need not exist as an R object, in which case a function with no arguments and an empty body is supplied for editing.

Editing an R object may change it in ways other than are obvious: see the comment under `edit`. See `edit.data.frame` for changes that can occur when editing a data frame or matrix.

See Also

`edit`, `edit.data.frame`

Examples

```
## Not run:
 ## Assume 'my.fun' is a user defined function :
 fix(my.fun)
 ## now my.fun is changed
 ## Also,
 fix(my.data.frame) # calls up data editor
 fix(my.data.frame, factor.mode="char") # use of ...

## End(Not run)
```

flush.console *Flush Output to a Console*

Description

This does nothing except on console-based versions of R. On the macOS and Windows GUIs, it ensures that the display of output in the console is current, even if output buffering is on.

Usage

```
flush.console()
```

format *Format Unordered and Ordered Lists*

Description

Format unordered (itemize) and ordered (enumerate) lists.

Usage

```
formatUL(x, label = "*", offset = 0,
         width = 0.9 * getOption("width"))
formatOL(x, type = "arabic", offset = 0, start = 1,
         width = 0.9 * getOption("width"))
```

Arguments

x	a character vector of list items.
label	a character string used for labelling the items.
offset	a non-negative integer giving the offset (indentation) of the list.
width	a positive integer giving the target column for wrapping lines in the output.
type	a character string specifying the 'type' of the labels in the ordered list. If "arabic" (default), arabic numerals are used. For "Alph" or "alph", single upper or lower case letters are employed (in this case, the number of the last item must not exceed 26). Finally, for "Roman" or "roman", the labels are given as upper or lower case roman numerals (with the number of the last item maximally 3999). type can be given as a unique abbreviation of the above, or as one of the HTML style tokens "1" (arabic), "A"/"a" (alphabetic), or "I"/"i" (roman), respectively.
start	a positive integer specifying the starting number of the first item in an ordered list.

Value

A character vector with the formatted entries.

See Also

formatDL for formatting description lists.

Examples

```
## A simpler recipe.
x <- c("Mix dry ingredients thoroughly.",
       "Pour in wet ingredients.",
       "Mix for 10 minutes.",
       "Bake for one hour at 300 degrees.")
## Format and output as an unordered list.
writeLines(formatUL(x))
## Format and output as an ordered list.
writeLines(formatOL(x))
## Ordered list using lower case roman numerals.
writeLines(formatOL(x, type = "i"))
## Ordered list using upper case letters and some offset.
writeLines(formatOL(x, type = "A", offset = 5))
```

---

getAnywhere *Retrieve an R Object, Including from a Namespace*

Description

These functions locate all objects with name matching their argument, whether visible on the search path, registered as an S3 method or in a namespace but not exported. `getAnywhere()` returns the objects and `argsAnywhere()` returns the arguments of any objects that are functions.

Usage

```
getAnywhere(x)
argsAnywhere(x)
```

Arguments

x	a character string or name.

Details

These functions look at all loaded namespaces, whether or not they are associated with a package on the search list.

They do not search literally "anywhere": for example, local evaluation frames and namespaces that are not loaded will not be searched.

Where functions are found as registered S3 methods, an attempt is made to find which namespace registered them. This may not be correct, especially if namespaces have been unloaded.

Value

For `getAnywhere()` an object of class `"getAnywhere"`. This is a list with components

`name`	the name searched for
`objs`	a list of objects found
`where`	a character vector explaining where the object(s) were found
`visible`	logical: is the object visible
`dups`	logical: is the object identical to one earlier in the list.

In computing whether objects are identical, their environments are ignored.

Normally the structure will be hidden by the `print` method. There is a `[` method to extract one or more of the objects found.

For `argsAnywhere()` one or more argument lists as returned by `args`.

See Also

`getS3method` to find the method which would be used: this might not be the one of those returned by `getAnywhere` since it might have come from a namespace which was unloaded or be registered under another name.

`get`, `getFromNamespace`, `args`

Examples

```
getAnywhere("format.dist")
getAnywhere("simpleLoess") # not exported from stats
argsAnywhere(format.dist)
```

`getFromNamespace` *Utility Functions for Developing Namespaces*

Description

Utility functions to access and replace the non-exported functions in a namespace, for use in developing packages with namespaces.

They should not be used in production code (except perhaps `assignInMyNamespace`, but see the 'Note').

Usage

```
getFromNamespace(x, ns, pos = -1, envir = as.environment(pos))

assignInNamespace(x, value, ns, pos = -1,
                  envir = as.environment(pos))

assignInMyNamespace(x, value)

fixInNamespace(x, ns, pos = -1, envir = as.environment(pos), ...)
```

Arguments

x	an object name (given as a character string).
value	an R object.
ns	a namespace, or character string giving the namespace.
pos	where to look for the object: see get.
envir	an alternative way to specify an environment to look in.
...	arguments to pass to the editor: see edit.

Details

assignInMyNamespace is intended to be called from functions within a package, and chooses the namespace as the environment of the function calling it.

The namespace can be specified in several ways. Using, for example, ns = "stats" is the most direct, but a loaded package can be specified via any of the methods used for get: ns can also be the environment printed as '<namespace:foo>'.

getFromNamespace is similar to (but predates) the ::: operator: it is more flexible in how the namespace is specified.

fixInNamespace invokes edit on the object named x and assigns the revised object in place of the original object. For compatibility with fix, x can be unquoted.

Value

getFromNamespace returns the object found (or gives an error).

assignInNamespace, assignInMyNamespace and fixInNamespace are invoked for their side effect of changing the object in the namespace.

Warning

assignInNamespace should not be used in final code, and will in future throw an error if called from a package. Already certain uses are disallowed.

Note

assignInNamespace, assignInMyNamespace and fixInNamespace change the copy in the namespace, but not any copies already exported from the namespace, in particular an object of that name in the package (if already attached) and any copies already imported into other namespaces. They are really intended to be used *only* for objects which are not exported from the namespace. They do attempt to alter a copy registered as an S3 method if one is found.

They can only be used to change the values of objects in the namespace, not to create new objects.

See Also

get, fix, getS3method

Examples

```
getFromNamespace("findGeneric", "utils")
## Not run:
fixInNamespace("predict.ppr", "stats")
stats:::predict.ppr
getS3method("predict", "ppr")
## alternatively
fixInNamespace("predict.ppr", pos = 3)
fixInNamespace("predict.ppr", pos = "package:stats")

## End(Not run)
```

---

getParseData *Get Detailed Parse Information from Object*

Description

If the "keep.source" option is TRUE, R's parser will attach detailed information on the object it has parsed. These functions retrieve that information.

Usage

```
getParseData(x, includeText = NA)
getParseText(parseData, id)
```

Arguments

x	an expression returned from parse, or a function or other object with source reference information
includeText	logical; whether to include the text of parsed items in the result
parseData	a data frame returned from getParseData
id	a vector of item identifiers whose text is to be retrieved

Details

In version 3.0.0, the R parser was modified to include code written by Romain Francois in his **parser** package. This constructs a detailed table of information about every token and higher level construct in parsed code. This table is stored in the srcfile record associated with source references in the parsed code, and retrieved by the getParseData function.

Value

For getParseData:
If parse data is not present, NULL. Otherwise a data frame is returned, containing the following columns:

line1	integer. The line number where the item starts. This is the parsed line number called "parse" in getSrcLocation, which ignores #line directives.

col1	integer. The column number where the item starts. The first character is column 1. This corresponds to "column" in getSrcLocation.
line2	integer. The line number where the item ends.
col2	integer. The column number where the item ends.
id	integer. An identifier associated with this item.
parent	integer. The id of the parent of this item.
token	character string. The type of the token.
terminal	logical. Whether the token is "terminal", i.e. a leaf in the parse tree.
text	character string. If includeText is TRUE, the text of all tokens; if it is NA (the default), the text of terminal tokens. If includeText == FALSE, this column is not included. Very long strings (with source of 1000 characters or more) will not be stored; a message giving their length and delimiter will be included instead.

The rownames of the data frame will be equal to the id values, and the data frame will have a "srcfile" attribute containing the srcfile record which was used. The rows will be ordered by starting position within the source file, with parent items occurring before their children.

For getParseText:
A character vector of the same length as id containing the associated text items. If they are not included in parseData, they will be retrieved from the original file.

Note

There are a number of differences in the results returned by getParseData relative to those in the original **parser** code:

- Fewer columns are kept.
- The internal token number is not returned.
- col1 starts counting at 1, not 0.
- The id values are not attached to the elements of the parse tree, they are only retained in the table returned by getParseData.
- #line directives are identified, but other comment markup (e.g., **roxygen** comments) are not.

Parse data by design explore details of the parser implementation, which are subject to change without notice. Applications computing on the parse data may require updates for each R release.

Author(s)

Duncan Murdoch

References

Romain Francois (2012). parser: Detailed R source code parser. R package version 0.0-16. https://github.com/halpo/parser.

See Also

parse, srcref

Examples

```
fn <- function(x) {
  x + 1 # A comment, kept as part of the source
}

d <- getParseData(fn)
if (!is.null(d)) {
  plus <- which(d$token == "'+'")
  sum <- d$parent[plus]
  print(d[as.character(sum),])
  print(getParseText(d, sum))
}
```

`getS3method` *Get an S3 Method*

Description

Get a method for an S3 generic, possibly from a namespace or the generic's registry.

Usage

```
getS3method(f, class, optional = FALSE, envir = parent.frame())
```

Arguments

`f`	a character string giving the name of the generic.
`class`	a character string giving the name of the class.
`optional`	logical: should failure to find the generic or a method be allowed?
`envir`	the `environment` in which the method and its generic are searched first.

Details

S3 methods may be hidden in namespaces, and will not then be found by `get`: this function can retrieve such functions, primarily for debugging purposes.

Further, S3 methods can be registered on the generic when a namespace is loaded, and the registered method will be used if none is visible (using namespace scoping rules).

It is possible that which S3 method will be used may depend on where the generic `f` is called from: `getS3method` returns the method found if `f` were called from the same environment.

Value

The function found, or `NULL` if no function is found and `optional = TRUE`.

See Also

`methods`, `get`, `getAnywhere`

Examples

```
require(stats)
exists("predict.ppr") # false
getS3method("predict", "ppr")
```

`getWindowsHandle` *Get a Windows Handle*

Description

Get the Windows handle of a window or of the R process in MS Windows.

Usage

```
getWindowsHandle(which = "Console")
```

Arguments

`which`	a string (see below), or the number of a graphics device window (which must a `windows` one).

Details

`getWindowsHandle` gets the Windows handle. Possible choices for `which` are:

`"Console"`	The console window handle.
`"Frame"`	The MDI frame window handle.
`"Process"`	The process pseudo-handle.
A device number	The window handle of a graphics device

These values are not normally useful to users, but may be used by developers making addons to R.

`NULL` is returned for the Frame handle if not running in MDI mode, for the Console handle when running Rterm, for any unrecognized string for `which`, or for a graphics device with no corresponding window.

Other windows (help browsers, etc.) are not accessible through this function.

Value

An external pointer holding the Windows handle, or `NULL`.

Note

This is only available on Windows.

See Also

`getIdentification`, `getWindowsHandles`

Examples

```
if(.Platform$OS.type == "windows")
  print( getWindowsHandle() )
```

`getWindowsHandles` *Get handles of Windows in the MS Windows RGui*

Description

This function gets the Windows handles of visible top level windows or windows within the R MDI frame (when using the `Rgui`).

Usage

```
getWindowsHandles(which = "R", pattern = "", minimized = FALSE)
```

Arguments

`which`	A vector of strings "R" or "all" (possibly with repetitions). See the Details section.
`pattern`	A vector of patterns that the titles of the windows must match.
`minimized`	A logical vector indicating whether minimized windows should be considered.

Details

This function will search for Windows handles, for passing to external GUIs or to the `arrangeWindows` function. Each of the arguments may be a vector of values. These will be treated as follows:

- The arguments will all be recycled to the same length.
- The corresponding elements of each argument will be applied in separate searches.
- The final result will be the union of the windows identified in each of the searches.

If an element of `which` is `"R"`, only windows belonging to the current R process will be returned. In MDI mode, those will be the child windows within the R GUI (`Rgui`) frame. In SDI mode, all windows belonging to the process will be included.

If the element is `"all"`, then top level windows will be returned.

The elements of `pattern` will be used to make a subset of windows whose title text matches (according to `grep`) the pattern.

If `minimized = FALSE`, minimized windows will be ignored.

Value

A list of external pointers containing the window handles.

Note

This is only available on Windows.

Author(s)

Duncan Murdoch

See Also

arrangeWindows, getWindowsHandle (singular).

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
  getWindowsHandles()
  getWindowsHandles("all")
})
```

---

glob2rx *Change Wildcard or Globbing Pattern into Regular Expression*

Description

Change *wildcard* aka *globbing* patterns into the corresponding regular expressions (regexp).

This is also a practical didactical example for the use of sub() and regular expressions.

Usage

```
glob2rx(pattern, trim.head = FALSE, trim.tail = TRUE)
```

Arguments

pattern	character vector
trim.head	logical specifying if leading "^.*" should be trimmed from the result.
trim.tail	logical specifying if trailing ".*$" should be trimmed from the result.

Details

This takes a wildcard as used by most shells and returns an equivalent regular expression. '?' is mapped to '.' (match a single character), '*' to '.*' (match any string, including an empty one), and the pattern is anchored (it must start at the beginning and end at the end). Optionally, the resulting regexp is simplified.

Note that now even '(', '[' and '{' can be used in pattern, but glob2rx() may not work correctly with arbitrary characters in pattern, for example escaped special characters.

Value

A character vector of the same length as the input `pattern` where each wildcard is translated to the corresponding regular expression.

Author(s)

Martin Maechler, Unix/sed based version, 1991; current: 2004

See Also

`regexp` about regular expression, `sub`, etc about substitutions using regexps. `Sys.glob` does wildcard expansio, i.e., "globbing" on file paths more subtly, e.g., allowing to escape special characters.

Examples

```
stopifnot(glob2rx("abc.*") == "^abc\\.",
          glob2rx("a?b.*") == "^a.b\\.",
          glob2rx("a?b.*", trim.tail = FALSE) == "^a.b\\..*$",
          glob2rx("*.doc") == "^.*\\.doc$",
          glob2rx("*.doc", trim.head = TRUE) == "\\.doc$",
          glob2rx("*.t*")  == "^.*\\.t",
          glob2rx("*.t??") == "^.*\\.t..$",
          glob2rx("*[*")  == "^.*\\["
)
```

---

globalVariables *Declarations Used in Checking a Package*

Description

For `globalVariables`, the names supplied are of functions or other objects that should be regarded as defined globally when the `check` tool is applied to this package. The call to `globalVariables` will be included in the package's source. Repeated calls in the same package accumulate the names of the global variables.

Typical examples are the fields and methods in reference classes, which appear to be global objects to `codetools`. (This case is handled automatically by `setRefClass()` and friends, using the supplied field and method names.)

For `suppressForeignCheck`, the names supplied are of variables used as `.NAME` in foreign function calls which should not be checked by `checkFF(registration = TRUE)`. Without this declaration, expressions other than simple character strings are assumed to evaluate to registered native symbol objects. The type of call (`.Call`, `.External`, etc.) and argument counts will be checked. With this declaration, checks on those names will usually be suppressed. (If the code uses an expression that should only be evaluated at runtime, the message can be suppressed by wrapping it in a `dontCheck` function call, or by saving it to a local variable, and suppressing messages about that variable. See the example below.)

Usage

```
globalVariables(names, package, add = TRUE)
suppressForeignCheck(names, package, add = TRUE)
```

Arguments

names	The character vector of object names. If omitted, the current list of global variables declared in the package will be returned, unchanged.
package	The relevant package, usually the character string name of the package but optionally its corresponding namespace environment. When the call to globalVariables or suppressForeignCheck comes in the package's source file, the argument is normally omitted, as in the example below.
add	Should the contents of names be added to the current global variables or replace it?

Details

The lists of declared global variables and native symbol objects are stored in a metadata object in the package's namespace, assuming the globalVariables or suppressForeignCheck call(s) occur as top-level calls in the package's source code.

The check command, as implemented in package tools, queries the list before checking the R source code in the package for possible problems.

globalVariables was introduced in R 2.15.1 and suppressForeignCheck was introduced in R 3.1.0 so both should be used conditionally: see the example.

Value

globalVariables returns the current list of declared global variables, possibly modified by this call.

suppressForeignCheck returns the current list of native symbol objects which are not to be checked.

Note

The global variables list really belongs to a restricted scope (a function or a group of method definitions, for example) rather than the package as a whole. However, implementing finer control would require changes in check and/or in codetools, so in this version the information is stored at the package level.

Author(s)

John Chambers and Duncan Murdoch

See Also

dontCheck.

Examples

```
## Not run:
## assume your package has some code that assigns ".obj1" and ".obj2"
## but not in a way that codetools can find.
## In the same source file (to remind you that you did it) add:
if(getRversion() >= "2.15.1")  utils::globalVariables(c(".obj1", "obj2"))

## To suppress messages about a run-time calculated native symbol,
## save it to a local variable.

## At top level, put this:
if(getRversion() >= "3.1.0") utils::suppressForeignCheck("localvariable")

## Within your function, do the call like this:
localvariable <- if (condition) entry1 else entry2
.Call(localvariable, 1, 2, 3)

## HOWEVER, it is much better practice to write code
## that can be checked thoroughly, e.g.
if(condition) .Call(entry1, 1, 2, 3) else .Call(entry2, 1, 2, 3)

## End(Not run)
```

---

hashtab *Hash Tables (Experimental)*

Description

Create and manipulate mutable hash tables.

Usage

```
hashtab(type = c("identical", "address"), size)
gethash(h, key, nomatch = NULL)
sethash(h, key, value)
remhash(h, key)
numhash(h)
typhash(h)
maphash(h, FUN)
clrhash(h)
is.hashtab(x)
## S3 method for class 'hashtab'
h[[key, nomatch = NULL, ...]]
## S3 replacement method for class 'hashtab'
h[[key, ...]] <- value
## S3 method for class 'hashtab'
print(x, ...)
## S3 method for class 'hashtab'
```

```
format(x, ...)
## S3 method for class 'hashtab'
length(x)
## S3 method for class 'hashtab'
str(object, ...)
```

Arguments

type	character string specifying the hash table type.
size	an integer specifying the expected number of entries.
h, object	a hash table.
key	an R object to use as a key.
nomatch	value to return if key does not match.
value	new value to associate with key.
FUN	a function of two arguments, the key and the value, to call for each entry.
x	object to be tested, printed, or formatted.
...	additional arguments.

Details

Hash tables are a data structure for efficiently associating keys with values. Hash tables are similar to environments, but keys can be arbitrary objects. Like environments, and unlike named lists and most other objects in R, hash tables are mutable, i.e., they are *not* copied when modified and assignment means just giving a new name to the same object.

New hash tables are created by hashtab. Two variants are available: keys can be considered to match if they are identical() (type = "identical", the default), or if their addresses in memory are equal (type = "address"). The default "identical" type is almost always the right choice. The size argument provides a hint for setting the initial hash table size. The hash table will grow if necessary, but specifying an expected size can be more efficient.

gethash returns the value associated with key. If key is not present in the table, then the value of nomatch is returned.

sethash adds a new key/value association or changes the current value for an existing key. remhash removes the entry for key, if there is one.

maphash calls FUN for each entry in the hash table with two arguments, the entry key and the entry value. The order in which the entries are processed is not predictable. The consequence of FUN adding entries to the table or deleting entries from the table is also not predictable, except that removing the entry currently being processed will have the desired effect.

clrhash removes all entries from the hash table.

Value

hashtab returns a new hash table of the specified type.

gethash returns the value associated with key, or nomatch if there is no such value.

sethash returns value invisibly.

remhash invisibly returns TRUE if an entry for key was found and removed, and FALSE if no entry was found.

numhash returns the current number of entries in the table.

typhash returns a character string specifying the type of the hash table, one of "identical" or "address".

maphash and clrhash return NULL invisibly.

Notes

The interface design is based loosely on hash table support in Common Lisp.

The hash function and equality test used for "identical" hash tables are the same as the ones used internally by duplicated and unique, with two exceptions:

- Closure environments are not ignored when comparing closures. This corresponds to calling identical() with ignore.environment = FALSE, which is the default for identical().
- External pointer objects are compared as reference objects, corresponding to calling identical() with extptr.as.ref = TRUE. This ensures that hash tables with keys containing external pointers behave reasonably when serialized and unserialized.

As an experimental feature, the element operator [[can also be used get or set hash table entries, and length can be used to obtain the number of entries. It is not yet clear whether this is a good idea.

Examples

```
## Create a new empty hash table.
h1 <- hashtab()
h1

## Add some key/value pairs.
sethash(h1, NULL, 1)
sethash(h1, .GlobalEnv, 2)
for (i in seq_along(LETTERS)) sethash(h1, LETTERS[i], i)

## Look up values for some keys.
gethash(h1, NULL)
gethash(h1, .GlobalEnv)
gethash(h1, "Q")

## Remove an entry.
(remhash(h1, NULL))
gethash(h1, NULL)
(remhash(h1, "XYZ"))

## Using the element operator.
h1[["ABC"]]
h1[["ABC", nomatch = 77]]
h1[["ABC"]] <- "DEF"
h1[["ABC"]]
```

```
## Integers and real numbers that are equal are considered different
## (not identical) as keys:
identical(3, 3L)
sethash(h1, 3L, "DEF")
gethash(h1, 3L)
gethash(h1, 3)

## Two variables can refer to the same hash table.
h2 <- h1
identical(h1, h2)
## set in one, see in the "other"  <==> really one object with 2 names
sethash(h2, NULL, 77)
gethash(h1, NULL)
str(h1)

## An example of using  maphash():  get all hashkeys of a hash table:
hashkeys <- function(h) {
  val <- vector("list", numhash(h))
  idx <- 0
  maphash(h, function(k, v) { idx <<- idx + 1
                              val[idx] <<- list(k) })
  val
}

kList <- hashkeys(h1)
str(kList) # the *order* is "arbitrary" & cannot be "known"
```

---

hasName *Check for Name*

Description

hasName is a convenient way to test for one or more names in an R object.

Usage

```
hasName(x, name)
```

Arguments

x	Any object.
name	One or more character values to look for.

Details

hasName(x, name) is defined to be equivalent to name %in% names(x), though it will evaluate slightly more quickly. It is intended to replace the common idiom !is.null(x$name). The latter can be unreliable due to partial name matching; see the example below.

Value

A logical vector of the same length as `name` containing `TRUE` if the corresponding entry is in `names(x)`.

See Also

`%in%`, `exists`

Examples

```
x <- list(abc = 1, def = 2)
!is.null(x$abc) # correct
!is.null(x$a)   # this is the wrong test!
hasName(x, "abc")
hasName(x, "a")
```

---

head *Return the First or Last Parts of an Object*

Description

Returns the first or last parts of a vector, matrix, table, data frame or function. Since `head()` and `tail()` are generic functions, they may also have been extended to other classes.

Usage

```
head(x, ...)
## Default S3 method:
head(x, n = 6L, ...)
## S3 method for class 'matrix'
head(x, n = 6L, ...) # is exported as head.matrix()
## NB: The methods for 'data.frame' and 'array'  are identical to the 'matrix' one

## S3 method for class 'ftable'
head(x, n = 6L, ...)
## S3 method for class 'function'
head(x, n = 6L, ...)


tail(x, ...)
## Default S3 method:
tail(x, n = 6L, keepnums = FALSE, addrownums, ...)
## S3 method for class 'matrix'
tail(x, n = 6L, keepnums = TRUE, addrownums, ...) # exported as tail.matrix()
## NB: The methods for 'data.frame', 'array', and 'table'
##     are identical to the  'matrix'  one
```

```
## S3 method for class 'ftable'
tail(x, n = 6L, keepnums = FALSE, addrownums, ...)
## S3 method for class 'function'
tail(x, n = 6L, ...)
```

Arguments

x	an object
n	an integer vector of length up to dim(x) (or 1, for non-dimensioned objects). A logical is silently coerced to integer. Values specify the indices to be selected in the corresponding dimension (or along the length) of the object. A positive value of n[i] includes the first/last n[i] indices in that dimension, while a negative value excludes the last/first abs(n[i]), including all remaining indices. NA or non-specified values (when length(n) < length(dim(x))) select all indices in that dimension. Must contain at least one non-missing value.
keepnums	in each dimension, if no names in that dimension are present, create them using the indices included in that dimension. Ignored if dim(x) is NULL or its length 1.
addrownums	deprecated - keepnums should be used instead. Taken as the value of keepnums if it is explicitly set when keepnums is not.
...	arguments to be passed to or from other methods.

Details

For vector/array based objects, head() (tail()) returns a subset of the same dimensionality as x, usually of the same class. For historical reasons, by default they select the first (last) 6 indices in the first dimension ("rows") or along the length of a non-dimensioned vector, and the full extent (all indices) in any remaining dimensions. head.matrix() and tail.matrix() are exported.

The default and array(/matrix) methods for head() and tail() are quite general. They will work as is for any class which has a dim() method, a length() method (only required if dim() returns NULL), and a [method (that accepts the drop argument and can subset in all dimensions in the dimensioned case).

For functions, the lines of the deparsed function are returned as character strings.

When x is an array(/matrix) of dimensionality two and more, tail() will add dimnames similar to how they would appear in a full printing of x for all dimensions k where n[k] is specified and non-missing and dimnames(x)[[k]] (or dimnames(x) itself) is NULL. Specifically, the form of the added dimnames will vary for different dimensions as follows:

k=1 **(rows):** "[n,]" (right justified with whitespace padding)

k=2 **(columns):** "[,n]" (with *no* whitespace padding)

k>2 **(higher dims):** "n", i.e., the indices as *character* values

Setting keepnums = FALSE suppresses this behaviour.

As data.frame subsetting ('indexing') keeps attributes, so do the head() and tail() methods for data frames.

Value

An object (usually) like `x` but generally smaller. Hence, for `array`s, the result corresponds to `x[..., drop=FALSE]`. For `ftable` objects `x`, a transformed `format(x)`.

Note

For array inputs the output of `tail` when `keepnums` is `TRUE`, any dimnames vectors added for dimensions `>2` are the original numeric indices in that dimension *as character vectors*. This means that, e.g., for 3-dimensional array `arr`, `tail(arr, c(2,2,-1))[ , , 2]` and `tail(arr, c(2,2,-1))[ , , "2"]` may both be valid but have completely different meanings.

Author(s)

Patrick Burns, improved and corrected by R-Core. Negative argument added by Vincent Goulet. Multi-dimension support added by Gabriel Becker.

Examples

```
head(letters)
head(letters, n = -6L)

head(freeny.x, n = 10L)
head(freeny.y)

head(iris3)
head(iris3, c(6L, 2L))
head(iris3, c(6L, -1L, 2L))

tail(letters)
tail(letters, n = -6L)

tail(freeny.x)
## the bottom-right "corner" :
tail(freeny.x, n = c(4, 2))
tail(freeny.y)

tail(iris3)
tail(iris3, c(6L, 2L))
tail(iris3, c(6L, -1L, 2L))

## iris with dimnames stripped
a3d <- iris3 ; dimnames(a3d) <- NULL
tail(a3d, c(6, -1, 2)) # keepnums = TRUE is default here!
tail(a3d, c(6, -1, 2), keepnums = FALSE)

## data frame w/ a (non-standard) attribute:
treeS <- structure(trees, foo = "bar")
(n <- nrow(treeS))
stopifnot(exprs = { # attribute is kept
    identical(htS <- head(treeS), treeS[1:6, ])
    identical(attr(htS, "foo") , "bar")
```

```
    identical(tlS <- tail(treeS), treeS[(n-5):n, ])
    ## BUT if I use "useAttrib(.)", this is *not* ok, when n is of length 2:
    ## --- because [i,j]-indexing of data frames *also* drops "other" attributes ..
    identical(tail(treeS, 3:2), treeS[(n-2):n, 2:3] )
})

tail(library) # last lines of function

head(stats::ftable(Titanic))

## 1d-array (with named dim) :
a1 <- array(1:7, 7); names(dim(a1)) <- "O2"
stopifnot(exprs = {
  identical( tail(a1, 10), a1)
  identical( head(a1, 10), a1)
  identical( head(a1, 1), a1 [1 , drop=FALSE] ) # was a1[1] in R <= 3.6.x
  identical( tail(a1, 2), a1[6:7])
  identical( tail(a1, 1), a1 [7 , drop=FALSE] ) # was a1[7] in R <= 3.6.x
})
```

`help` *Documentation*

Description

`help` is the primary interface to the help systems.

Usage

```
help(topic, package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     help_type = getOption("help_type"))
```

Arguments

`topic`	usually, a name or character string specifying the topic for which help is sought. A character string (enclosed in explicit single or double quotes) is always taken as naming a topic. If the value of `topic` is a length-one character vector the topic is taken to be the value of the only element. Otherwise `topic` must be a name or a reserved word (if syntactically valid) or character string. See 'Details' for what happens if this is omitted.
`package`	a name or character vector giving the packages to look into for documentation, or `NULL`. By default, all packages whose namespaces are loaded are used. To avoid a name being deparsed use e.g. `(pkg_ref)` (see the examples).

`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries. This is not used for HTML help (see 'Details').
`verbose`	logical; if `TRUE`, the file name is reported.
`try.all.packages`	logical; see `Note`.
`help_type`	character string: the type of help required. Possible values are `"text"`, `"html"` and `"pdf"`. Case is ignored, and partial matching is allowed.

Details

The following types of help are available:

- Plain text help
- HTML help pages with hyperlinks to other topics, shown in a browser by `browseURL`. (On Unix-alikes, where possible an existing browser window is re-used: the macOS GUI uses its own browser window.)

  If for some reason HTML help is unavailable (see `startDynamicHelp`), plain text help will be used instead.
- For `help` only, typeset as PDF – see the section on 'Offline help'.

On Unix-alikes: The 'factory-fresh' default is text help except from the macOS GUI, which uses HTML help displayed in its own browser window.

On Windows: The default for the type of help is selected when R is installed – the 'factory-fresh' default is HTML help.

The rendering of text help will use directional quotes in suitable locales (UTF-8 and single-byte Windows locales): sometimes the fonts used do not support these quotes so this can be turned off by setting `options(useFancyQuotes = FALSE)`.

`topic` is not optional: if it is omitted R will give

- If a package is specified, (text or, in interactive use only, HTML) information on the package, including hints/links to suitable help topics.
- If `lib.loc` only is specified, a (text) list of available packages.
- Help on `help` itself if none of the first three arguments is specified.

Some topics need to be quoted (by backticks) or given as a character string. These include those which cannot syntactically appear on their own such as unary and binary operators, `function` and control-flow reserved words (including `if`, `else` `for`, `in`, `repeat`, `while`, `break` and `next`). The other `reserved` words can be used as if they were names, for example `TRUE`, `NA` and `Inf`.

If multiple help files matching `topic` are found, in interactive use a menu is presented for the user to choose one: in batch use the first on the search path is used. (For HTML help the menu will be an HTML page, otherwise a graphical menu if possible if `getOption("menu.graphics")` is true, the default.)

Note that HTML help does not make use of `lib.loc`: it will always look first in the loaded packages and then along `.libPaths()`.

Offline help

Typeset documentation is produced by running the LaTeX version of the help page through `pdflatex`: this will produce a PDF file.

The appearance of the output can be customized through a file '`Rhelp.cfg`' somewhere in your LaTeX search path: this will be input as a LaTeX style file after `Rd.sty`. Some environment variables are consulted, notably `R_PAPERSIZE` (*via* `getOption("papersize")`) and `R_RD4PDF` (see 'Making manuals' in the 'R Installation and Administration' manual).

If there is a function `offline_help_helper` in the workspace or further down the search path it is used to do the typesetting, otherwise the function of that name in the `utils` namespace (to which the first paragraph applies). It should accept at least two arguments, the name of the LaTeX file to be typeset and the type (which is nowadays ignored). It accepts a third argument, `texinputs`, which will give the graphics path when the help document contains figures, and will otherwise not be supplied.

Note

Unless `lib.loc` is specified explicitly, the loaded packages are searched before those in the specified libraries. This ensures that if a library is loaded from a library not in the known library trees, then the help from the loaded library is used. If `lib.loc` is specified explicitly, the loaded packages are *not* searched.

If this search fails and argument `try.all.packages` is `TRUE` and neither `packages` nor `lib.loc` is specified, then all the packages in the known library trees are searched for help on `topic` and a list of (any) packages where help may be found is displayed (with hyperlinks for `help_type = "html"`). **NB:** searching all packages can be slow, especially the first time (caching of files by the OS can expedite subsequent searches dramatically).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`?` for shortcuts to help topics.

`help.search()` or `??` for finding help pages on a vague topic; `help.start()` which opens the HTML version of the R help pages; `library()` for listing available packages and the help objects they contain; `data()` for listing available data sets; `methods()`.

Use `prompt()` to get a prototype for writing `help` pages of your own package.

Examples

```
help()
help(help)              # the same

help(lapply)

help("for")             # or ?"for", but quotes/backticks are needed
```

```
try({# requires working TeX installation:
 help(dgamma, help_type = "pdf")
 ## -> nicely formatted pdf -- including math formula -- for help(dgamma):
 system2(getOption("pdfviewer"), "dgamma.pdf", wait = FALSE)
})

help(package = "splines") # get help even when package is not loaded

topi <- "women"
help(topi)

try(help("bs", try.all.packages = FALSE)) # reports not found (an error)
help("bs", try.all.packages = TRUE)       # reports can be found
                                          # in package 'splines'

## For programmatic use:
topic <- "family"; pkg_ref <- "stats"
help((topic), (pkg_ref))
```

`help.request` *Send a Post to R-help*

Description

Prompts the user to check they have done all that is expected of them before sending a post to the R-help mailing list, provides a template for the post with session information included and optionally sends the email (on Unix systems).

Usage

```
help.request(subject = "",
             address = "r-help@R-project.org",
             file = "R.help.request", ...)
```

Arguments

`subject`	subject of the email. Please do not use single quotes (') in the subject! Post separate help requests for multiple queries.
`address`	recipient's email address.
`file`	filename to use (if needed) for setting up the email.
`...`	additional named arguments such as `method` and `ccaddress` to pass to `create.post`.

Details

This function is not intended to replace the posting guide. Please read the guide before posting to R-help or using this function (see https://www.r-project.org/posting-guide.html).

The `help.request` function:

- asks whether the user has consulted relevant resources, stopping and opening the relevant URL if a negative response if given.
- checks whether the current version of R is being used and whether the add-on packages are up-to-date, giving the option of updating where necessary.
- asks whether the user has prepared appropriate (minimal, reproducible, self-contained, commented) example code ready to paste into the post.

Once this checklist has been completed a template post is prepared including current session information, and passed to `create.post`.

Value

Nothing useful.

Author(s)

Heather Turner, based on the then current code and help page of `bug.report()`.

See Also

The posting guide (https://www.r-project.org/posting-guide.html), also `sessionInfo()` from which you may add to the help request.

`create.post`.

`help.search` *Search the Help System*

Description

Allows for searching the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries (or any combination thereof), using either fuzzy matching or regular expression matching. Names and titles of the matched help entries are displayed nicely formatted.

Vignette names, titles and keywords and demo names and titles may also be searched.

Usage

```
help.search(pattern, fields = c("alias", "concept", "title"),
            apropos, keyword, whatis, ignore.case = TRUE,
            package = NULL, lib.loc = NULL,
            help.db = getOption("help.db"),
            verbose = getOption("verbose"),
            rebuild = FALSE, agrep = NULL, use_UTF8 = FALSE,
            types = getOption("help.search.types"))
??pattern
field??pattern
```

Arguments

pattern	a character string to be matched in the specified fields. If this is given, the arguments apropos, keyword, and whatis are ignored.
fields	a character vector specifying the fields of the help database to be searched. The entries must be abbreviations of "name", "title", "alias", "concept", and "keyword", corresponding to the help page's (file) name, its title, the topics and concepts it provides documentation for, and the keywords it can be classified to. See below for details and how vignettes and demos are searched.
apropos	a character string to be matched in the help page topics and title.
keyword	a character string to be matched in the help page 'keywords'. 'Keywords' are really categories: the standard categories are listed in file 'R.home("doc")/KEYWORDS' (see also the example) and some package writers have defined their own. If keyword is specified, agrep defaults to FALSE.
whatis	a character string to be matched in the help page topics.
ignore.case	a logical. If TRUE, case is ignored during matching; if FALSE, pattern matching is case sensitive.
package	a character vector with the names of packages to search through, or NULL in which case *all* available packages in the library trees specified by lib.loc are searched.
lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
help.db	a character string giving the file path to a previously built and saved help database, or NULL.
verbose	logical; if TRUE, the search process is traced. Integer values are also accepted, with TRUE being equivalent to 2, and 1 being less verbose. On Windows a progress bar is shown during rebuilding, and on Unix a heartbeat is shown for verbose = 1 and a package-by-package list for verbose >= 2.
rebuild	a logical indicating whether the help database should be rebuilt. This will be done automatically if lib.loc or the search path is changed, or if package is used and a value is not found.
agrep	if NULL (the default unless keyword is used) and the character string to be matched consists of alphanumeric characters, whitespace or a dash only, approximate (fuzzy) matching via agrep is used unless the string has fewer than 5 characters; otherwise, it is taken to contain a regular expression to be matched via grep. If FALSE, approximate matching is not used. Otherwise, one can give a numeric or a list specifying the maximal distance for the approximate match, see argument max.distance in the documentation for agrep.
use_UTF8	logical: should results be given in UTF-8 encoding? Also changes the meaning of regexps in agrep to be Perl regexps.
types	a character vector listing the types of documentation to search. The entries must be abbreviations of "vignette" "help" or "demo". Results will be presented in the order specified.
field	a single value of fields to search.

Details

Upon installation of a package, a pre-built help.search index is serialized as 'hsearch.rds' in the 'Meta' directory (provided the package has any help pages). Vignettes are also indexed in the 'Meta/vignette.rds' file. These files are used to create the help search database via hsearch_db.

The arguments apropos and whatis play a role similar to the Unix commands with the same names.

Searching with agrep = FALSE will be several times faster than the default (once the database is built). However, approximate searches should be fast enough (around a second with 5000 packages installed).

If possible, the help database is saved in memory for use by subsequent calls in the session.

Note that currently the aliases in the matching help files are not displayed.

As with ?, in ?? the pattern may be prefixed with a package name followed by :: or ::: to limit the search to that package.

For help files, '\keyword' entries which are not among the standard keywords as listed in file 'KEYWORDS' in the R documentation directory are taken as concepts. For standard keyword entries different from 'internal', the corresponding descriptions from file 'KEYWORDS' are additionally taken as concepts. All '\concept' entries used as concepts.

Vignettes are searched as follows. The "name" and "alias" are both the base of the vignette filename, and the "concept" entries are taken from the '\VignetteKeyword' entries. Vignettes are not classified using the help system "keyword" classifications. Demos are handled similarly to vignettes, without the "concept" search.

Value

The results are returned in a list object of class "hsearch", which has a print method for nicely formatting the results of the query. This mechanism is experimental, and may change in future versions of R.

In R.app on macOS, this will show up a browser with selectable items. On exiting this browser, the help pages for the selected items will be shown in separate help windows.

The internal format of the class is undocumented and subject to change.

See Also

hsearch_db for more information on the help search database employed, and for utilities to inspect available concepts and keywords.

help; help.start for starting the hypertext (currently HTML) version of R's online documentation, which offers a similar search mechanism.

RSiteSearch to access an on-line search of R resources.

apropos uses regexps and has nice examples.

Examples

```
help.search("linear models")    # In case you forgot how to fit linear
                                # models
help.search("non-existent topic")
```

```
??utils::help  # All the topics matching "help" in the utils package


## Documentation with topic/concept/title matching 'print'
## (disabling fuzzy matching to not also match 'point')
help.search("print", agrep = FALSE)
help.search(apropos = "print", agrep = FALSE)  # ignores concepts

## Help pages with documented topics starting with 'try':
help.search("^try", fields = "alias")
alias??"^try"  # the same

## Help pages documenting high-level plots:
help.search(keyword = "hplot")

RShowDoc("KEYWORDS")  # show all keywords
```

`help.start` *Hypertext Documentation*

Description

Start the hypertext (currently HTML) version of R's online documentation.

Usage

```
help.start(update = FALSE, gui = "irrelevant",
           browser = getOption("browser"), remote = NULL)
```

Arguments

`update`	logical: should this attempt to update the package index to reflect the currently available packages. (Not attempted if `remote` is non-`NULL`.)
`gui`	just for compatibility with S-PLUS.
`browser`	the name of the program to be used as hypertext browser. It should be in the `PATH`, or a full path specified. Alternatively, it can be an R function which will be called with a URL as its only argument. This option is normally unset on Windows, when the file-association mechanism will be used.
`remote`	A character string giving a valid URL for the '*R_HOME*' directory on a remote location.

Details

Unless `remote` is specified this requires the HTTP server to be available (it will be started if possible: see `startDynamicHelp`).

One of the links on the index page is the HTML package index, '*R_DOC_DIR*`/html/packages.html`', which can be remade by `make.packages.html()`.

For local operation, the HTTP server will remake a temporary version of this list when the link is first clicked, and each time thereafter check if updating is needed (if `.libPaths` has changed or any of the directories has been changed). This can be slow, and using `update = TRUE` will ensure that the packages list is updated before launching the index page.

Argument `remote` can be used to point to HTML help published by another R installation: it will typically only show packages from the main library of that installation.

See Also

`help()` for on- and off-line help in other formats.

`browseURL` for how the help file is displayed.

`RSiteSearch` to access an on-line search of R resources.

Examples

```
help.start()

## the 'remote' arg can be tested by
help.start(remote = paste0("file://", R.home()))
```

---

hsearch-utils *Help Search Utilities*

Description

Utilities for searching the help system.

Usage

```
hsearch_db(package = NULL, lib.loc = NULL,
           types = getOption("help.search.types"),
           verbose = getOption("verbose"),
           rebuild = FALSE, use_UTF8 = FALSE)
hsearch_db_concepts(db = hsearch_db())
hsearch_db_keywords(db = hsearch_db())
```

Arguments

`package`	a character vector with the names of packages to search through, or `NULL` in which case *all* available packages in the library trees specified by `lib.loc` are searched.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`types`	a character vector listing the types of documentation to search. See `help.search` for details.

verbose	a logical controlling the verbosity of building the help search database. See help.search for details.
rebuild	a logical indicating whether the help search database should be rebuilt. See help.search for details.
use_UTF8	logical: should results be given in UTF-8 encoding?
db	a help search database as obtained by calls to hsearch_db().

Details

hsearch_db() builds and caches the help search database for subsequent use by help.search. (In fact, re-builds only when forced (rebuild = TRUE) or "necessary".)

The format of the help search database is still experimental, and may change in future versions. Currently, it consists of four tables: one with base information about all documentation objects found, including their names and titles and unique ids; three more tables contain the individual aliases, concepts and keywords together with the ids of the documentation objects they belong to. Separating out the latter three tables accounts for the fact that a single documentation object may provide several of these entries, and allows for efficient searching.

See the details in help.search for how searchable entries are interpreted according to help type.

hsearch_db_concepts() and hsearch_db_keywords() extract all concepts or keywords, respectively, from a help search database, and return these in a data frame together with their total frequencies and the numbers of packages they are used in, with entries sorted in decreasing total frequency.

Examples

```
db <- hsearch_db()
## Total numbers of documentation objects, aliases, keywords and
## concepts (using the current format):
sapply(db, NROW)
## Can also be obtained from print method:
db
## 10 most frequent concepts:
head(hsearch_db_concepts(), 10)
## 10 most frequent keywords:
head(hsearch_db_keywords(), 10)
```

INSTALL *Install Add-on Packages*

Description

Utility for installing add-on packages.

Usage

```
R CMD INSTALL [options] [-l lib] pkgs
```

Arguments

pkgs	a space-separated list with the path names of the packages to be installed. See 'Details'.
lib	the path name of the R library tree to install to. Also accepted in the form '--library=lib'. Paths including spaces should be quoted, using the conventions for the shell in use.
options	a space-separated list of options through which in particular the process for building the help files can be controlled. Use R CMD INSTALL --help for the full current list of options.

Details

This will stop at the first error, so if you want all the pkgs to be tried, call this via a shell loop.

If used as R CMD INSTALL pkgs without explicitly specifying lib, packages are installed into the library tree rooted at the first directory in the library path which would be used by R run in the current environment.

To install into the library tree *lib*, use R CMD INSTALL -l *lib pkgs*. This prepends lib to the library path for duration of the install, so required packages in the installation directory will be found (and used in preference to those in other libraries).

Both lib and the elements of pkgs may be absolute or relative path names of directories. pkgs may also contain names of package archive files: these are then extracted to a temporary directory. These are tarballs containing a single directory, optionally compressed by gzip, bzip2, xz or compress. Finally, binary package archive files (as created by R CMD INSTALL --build) can be supplied.

Tarballs are by default unpackaged by the internal untar function: if needed an external tar command can be specified by the environment variable R_INSTALL_TAR: please ensure that it can handle the type of compression used on the tarball. (This is sometimes needed for tarballs containing invalid or unsupported sections, and can be faster on very large tarballs. Setting R_INSTALL_TAR to 'tar.exe' has been needed to overcome permissions issues on some Windows systems.)

The package sources can be cleaned up prior to installation by '--preclean' or after by '--clean': cleaning is essential if the sources are to be used with more than one architecture or platform.

Some package sources contain a 'configure' script that can be passed arguments or variables via the option '--configure-args' and '--configure-vars', respectively, if necessary. The latter is useful in particular if libraries or header files needed for the package are in non-system directories. In this case, one can use the configure variables LIBS and CPPFLAGS to specify these locations (and set these via '--configure-vars'), see section 'Configuration variables' in 'R Installation and Administration' for more information. (If these are used more than once on the command line they are concatenated.) The configure mechanism can be bypassed using the option '--no-configure'.

If the attempt to install the package fails, leftovers are removed. If the package was already installed, the old version is restored. This happens either if a command encounters an error or if the install is interrupted from the keyboard: after cleaning up the script terminates.

For details of the locking which is done, see the section 'Locking' in the help for install.packages.

Option '--build' can be used to tar up the installed package for distribution as a binary package (as used on macOS). This is done by utils::tar unless environment variable R_INSTALL_TAR is set.

By default a package is installed with static HTML help pages if and only if R was: use options '`--html`' and '`--no-html`' to override this.

Packages are not by default installed keeping the source formatting (see the `keep.source` argument to `source`): this can be enabled by the option '`--with-keep.source`' or by setting environment variable `R_KEEP_PKG_SOURCE` to `yes`.

Specifying the '`--install-tests`' option copies the contents of the '`tests`' directory into the package installation. If the `R_ALWAYS_INSTALL_TESTS` environment variable is set to a true value, the tests will be installed even if '`--install-tests`' is omitted.

Use `R CMD INSTALL --help` for concise usage information, including all the available options.

Sub-architectures

An R installation can support more than one sub-architecture: currently this is most commonly used for 32- and 64-bit builds on Windows.

For such installations, the default behaviour is to try to install source packages for all installed sub-architectures unless the package has a configure script or a '`src/Makefile`' (or '`src/Makefile.win`' on Windows), when only compiled code for the sub-architecture running `R CMD INSTALL` is installed.

To install a source package with compiled code only for the sub-architecture used by `R CMD INSTALL`, use '`--no-multiarch`'. To install just the compiled code for another sub-architecture, use '`--libs-only`'.

There are two ways to install for all available sub-architectures. If the configure script is known to work for both Windows architectures, use flag '`--force-biarch`' (and packages can specify this *via* a '`Biarch: yes`' field in their `DESCRIPTION` files). Second, a single tarball can be installed with

```
R CMD INSTALL --merge-multiarch mypkg_version.tar.gz
```

Staged installation

The default way to install source packages changed in R 3.6.0, so packages are first installed to a temporary location and then (if successful) moved to the destination library directory. Some older packages were written in ways that assume direct installation to the destination library.

Staged installation can currently be overridden by having a line '`StagedInstall: no`' in the package's '`DESCRIPTION`' file, *via* flag '`--no-staged-install`' or by setting environment variable `R_INSTALL_STAGED` to a false value (e.g. '`false`' or '`no`').

Staged installation requires either '`--pkglock`' or '`--lock`', one of which is used by default.

Note

The options do not have to precede '`pkgs`' on the command line, although it will be more legible if they do. All the options are processed before any packages, and where options have conflicting effects the last one will win.

Some parts of the operation of `INSTALL` depend on the R temporary directory (see `tempdir`, usually under '`/tmp`') having both write and execution access to the account running R. This is usually the case, but if '`/tmp`' has been mounted as `noexec`, environment variable `TMPDIR` may need to be set to a directory from which execution is allowed.

See Also

REMOVE; .libPaths for information on using several library trees; install.packages for R-level installation of packages; update.packages for automatic update of packages using the Internet or a local repository.

The chapter on 'Add-on packages' in 'R Installation and Administration' and the chapter on 'Creating R packages' in 'Writing R Extensions' *via* RShowDoc or in the 'doc/manual' subdirectory of the R source tree.

---

install.packages *Install Packages from Repositories or Local Files*

Description

Download and install packages from CRAN-like repositories or from local files.

Usage

```
install.packages(pkgs, lib, repos = getOption("repos"),
                 contriburl = contrib.url(repos, type),
                 method, available = NULL, destdir = NULL,
                 dependencies = NA, type = getOption("pkgType"),
                 configure.args = getOption("configure.args"),
                 configure.vars = getOption("configure.vars"),
                 clean = FALSE, Ncpus = getOption("Ncpus", 1L),
                 verbose = getOption("verbose"),
                 libs_only = FALSE, INSTALL_opts, quiet = FALSE,
                 keep_outputs = FALSE, ...)
```

Arguments

pkgs — character vector of the names of packages whose current versions should be downloaded from the repositories.

If repos = NULL, a character vector of file paths,

on Windows, file paths of '.zip' files containing binary builds of packages. ('http://' and 'file://' URLs are also accepted and the files will be downloaded and installed from local copies.) Source directories or file paths or URLs of archives may be specified with type = "source", but some packages need suitable tools installed (see the 'Details' section).

On Unix-alikes, these file paths can be source directories or archives or binary package archive files (as created by R CMD build --binary). ('http://' and 'file://' URLs are also accepted and the files will be downloaded and installed from local copies.) On a CRAN build of R for macOS these can be '.tgz' files containing binary package archives. Tilde-expansion will be done on file paths.

If this is missing, a listbox of available packages is presented where possible in an interactive R session.

`lib` character vector giving the library directories where to install the packages. Recycled as needed. If missing, defaults to the first element of `.libPaths()`.

`repos` character vector, the base URL(s) of the repositories to use, e.g., the URL of a CRAN mirror such as `"https://cloud.r-project.org"`. For more details on supported URL schemes see `url`.

Can be `NULL` to install from local files, directories or URLs: this will be inferred by extension from `pkgs` if of length one.

`contriburl` URL(s) of the contrib sections of the repositories. Use this argument if your repository mirror is incomplete, e.g., because you mirrored only the 'contrib' section, or only have binary packages. Overrides argument `repos`. Incompatible with `type = "both"`.

`method` download method, see `download.file`. Unused if a non-`NULL` `available` is supplied.

`available` a matrix as returned by `available.packages` listing packages available at the repositories, or `NULL` when the function makes an internal call to `available.packages`. Incompatible with `type = "both"`.

`destdir` directory where downloaded packages are stored. If it is `NULL` (the default) a subdirectory `downloaded_packages` of the session temporary directory will be used (and the files will be deleted at the end of the session).

`dependencies` logical indicating whether to also install uninstalled packages which these packages depend on/link to/import/suggest (and so on recursively). Not used if `repos = NULL`. Can also be a character vector, a subset of `c("Depends", "Imports", "LinkingTo", "Suggests", "Enhances")`.

Only supported if `lib` is of length one (or missing), so it is unambiguous where to install the dependent packages. If this is not the case it is ignored, with a warning.

The default, `NA`, means `c("Depends", "Imports", "LinkingTo")`.

`TRUE` means to use `c("Depends", "Imports", "LinkingTo", "Suggests")` for `pkgs` and `c("Depends", "Imports", "LinkingTo")` for added dependencies: this installs all the packages needed to run `pkgs`, their examples, tests and vignettes (if the package author specified them correctly).

In all of these, `"LinkingTo"` is omitted for binary packages.

`type` character, indicating the type of package to download and install. Will be `"source"` except on Windows and some macOS builds: see the section on 'Binary packages' for those.

`configure.args` (Used only for source installs.) A character vector or a named list. If a character vector with no names is supplied, the elements are concatenated into a single string (separated by a space) and used as the value for the '`--configure-args`' flag in the call to `R CMD INSTALL`. If the character vector has names these are assumed to identify values for '`--configure-args`' for individual packages. This allows one to specify settings for an entire collection of packages which will be used if any of those packages are to be installed. (These settings can therefore be re-used and act as default settings.)

A named list can be used also to the same effect, and that allows multi-element character strings for each package which are concatenated to a single string to be used as the value for '`--configure-args`'.

`configure.vars` (Used only for source installs.) Analogous to `configure.args` for flag '`--configure-vars`', which is used to set environment variables for the `configure` run.

`clean` a logical value indicating whether to add the '`--clean`' flag to the call to `R CMD INSTALL`. This is sometimes used to perform additional operations at the end of the package installation in addition to removing intermediate files.

`Ncpus` the number of parallel processes to use for a parallel install of more than one source package. Values greater than one are supported if the `make` command specified by `Sys.getenv("MAKE", "make")` accepts argument '`-k -j <Ncpus>`'.

`verbose` a logical indicating if some "progress report" should be given.

`libs_only` a logical value: should the '`--libs-only`' option be used to install only additional sub-architectures for source installs? (See also `INSTALL_opts`.) This can also be used on Windows to install just the DLL(s) from a binary package, e.g. to add 64-bit DLLs to a 32-bit install.

`INSTALL_opts` an optional character vector of additional option(s) to be passed to `R CMD INSTALL` for a source package install. E.g., `c("--html", "--no-multiarch", "--no-test-load")`.

Can also be a named list of character vectors to be used as additional options, with names the respective package names.

`quiet` logical: if true, reduce the amount of output. This is *not* passed to `available.packages()` in case that is called, on purpose.

`keep_outputs` a logical: if true, keep the outputs from installing source packages in the current working directory, with the names of the output files the package names with '`.out`' appended (overwriting existing files, possibly from previous installation attempts). Alternatively, a character string giving the directory in which to save the outputs. Ignored when installing from local files.

`...` further arguments to be passed to `download.file`, `available.packages`, or to the functions for binary installs on macOS and Windows (which accept an argument `"lock"`: see the section on 'Locking').

Details

This is the main function to install packages. It takes a vector of names and a destination library, downloads the packages from the repositories and installs them. (If the library is omitted it defaults to the first directory in `.libPaths()`, with a message if there is more than one.) If `lib` is omitted or is of length one and is not a (group) writable directory, in interactive use the code offers to create a personal library tree (the first element of `Sys.getenv("R_LIBS_USER")`) and install there.

Detection of a writable directory is problematic on Windows: see the 'Note' section.

For installs from a repository an attempt is made to install the packages in an order that respects their dependencies. This does assume that all the entries in `lib` are on the default library path for installs (set by environment variable `R_LIBS`).

You are advised to run `update.packages` before `install.packages` to ensure that any already installed dependencies have their latest versions.

Value

Invisible NULL.

Binary packages

This section applies only to platforms where binary packages are available: Windows and CRAN builds for macOS.

R packages are primarily distributed as *source* packages, but *binary* packages (a packaging up of the installed package) are also supported, and the type most commonly used on Windows and by the CRAN builds for macOS. This function can install either type, either by downloading a file from a repository or from a local file.

Possible values of type are (currently) "source", "mac.binary", and "win.binary": the appropriate binary type where supported can also be selected as "binary".

For a binary install from a repository, the function checks for the availability of a source package on the same repository, and reports if the source package has a later version, or is available but no binary version is. This check can be suppressed by using

```
options(install.packages.check.source = "no")
```

and should be if there is a partial repository containing only binary files.

An alternative (and the current default) is "both" which means 'use binary if available and current, otherwise try source'. The action if there are source packages which are preferred but may contain code which needs to be compiled is controlled by getOption("install.packages.compile.from.source"). type = "both" will be silently changed to "binary" if either contriburl or available is specified.

Using packages with type = "source" always works provided the package contains no C/C++/Fortran code that needs compilation. Otherwise,

on Windows, you will need to have installed the Rtools collection as described in the 'R for Windows FAQ' *and* you must have the PATH environment variable set up as required by Rtools.

For a 32/64-bit installation of R on Windows, a small minority of packages with compiled code need either INSTALL_opts = "--force-biarch" or INSTALL_opts = "--merge-multiarch" for a source installation. (It is safe to always set the latter when installing from a repository or tarballs, although it will be a little slower.)

When installing a package on Windows, install.packages will abort the install if it detects that the package is already installed and is currently in use. In some circumstances (e.g., multiple instances of R running at the same time and sharing a library) it will not detect a problem, but the installation may fail as Windows locks files in use.

On Unix-alikes, when the package contains C/C++/Fortran code that needs compilation, suitable compilers and related tools need to be installed. On macOS you need to have installed the 'Command-line tools for Xcode' (see the 'R Installation and Administration' manual) and if needed by the package a Fortran compiler, and have them in your path.

Locking

There are various options for locking: these differ between source and binary installs.

By default for a source install, the library directory is 'locked' by creating a directory '00LOCK' within it. This has two purposes: it prevents any other process installing into that library concurrently, and is used to store any previous version of the package to restore on error. A finer-grained locking is provided by the option '--pkglock' which creates a separate lock for each package: this allows enough freedom for parallel installation. Per-package locking is the default when installing a single package, and for multiple packages when Ncpus > 1L. Finally locking (and restoration on error) can be suppressed by '--no-lock'.

For a macOS binary install, no locking is done by default. Setting argument lock to TRUE (it defaults to the value of getOption("install.lock", FALSE)) will use per-directory locking as described for source installs. For Windows binary install, per-directory locking is used by default (lock defaults to the value of getOption("install.lock", TRUE)). If the value is "pkglock" per-package locking will be used.

If package locking is used on Windows with libs_only = TRUE and the installation fails, the package will be restored to its previous state.

Note that it is possible for the package installation to fail so badly that the lock directory is not removed: this inhibits any further installs to the library directory (or for '--pkglock', of the package) until the lock directory is removed manually.

Parallel installs

Parallel installs are attempted if pkgs has length greater than one and Ncpus > 1. It makes use of a parallel make, so the make specified (default make) when R was built must be capable of supporting make -j *N*: GNU make, dmake and pmake do, but Solaris make and older FreeBSD make do not: if necessary environment variable MAKE can be set for the current session to select a suitable make.

install.packages needs to be able to compute all the dependencies of pkgs from available, including if one element of pkgs depends indirectly on another. This means that if for example you are installing CRAN packages which depend on Bioconductor packages which in turn depend on CRAN packages, available needs to cover both CRAN and Bioconductor packages.

Timeouts

A limit on the elapsed time for each call to R CMD INSTALL (so for source installs) can be set *via* environment variable _R_INSTALL_PACKAGES_ELAPSED_TIMEOUT_: in seconds (or in minutes or hours with optional suffix 'm' or 'h', suffix 's' being allowed for the default seconds) with 0 meaning no limit.

For non-parallel installs this is implemented *via* the timeout argument of system2: for parallel installs *via* the OS's timeout command. (The one tested is from GNU **coreutils**, commonly available on Linux but not other Unix-alikes. If no such command is available the timeout request is ignored, with a warning. On Windows, one needs to specify a suitable timeout command via environment variable R_TIMEOUT, because 'c:/Windows/system32/timeout.exe' is not.) For parallel installs a 'Error 124' message from make indicates that timeout occurred.

Timeouts during installation might leave lock directories behind and not restore previous versions.

Version requirements on source installs

If you are not running an up-to-date version of R you may see a message like

```
package 'RODBC' is not available (for R version 3.5.3)
```

One possibility is that the package is not available in any of the selected repositories; another is that is available but only for current or recent versions of R. For CRAN packages take a look at the package's CRAN page (e.g., https://cran.r-project.org/package=RODBC). If that indicates in the 'Depends' field a dependence on a later version of R you will need to look in the 'Old sources' section and select the URL of a version of comparable age to your R. Then you can supply that URL as the first argument of install.packages(): you may need to first manually install its dependencies.

For other repositories, using available.packages(filters = "OS_type")[*pkgname*,] will show if the package is available for any R version (for your OS).

Note

On Unix-alikes: Some binary distributions of R have INSTALL in a separate bundle, e.g. an R-devel RPM. install.packages will give an error if called with type = "source" on such a system.

Some binary Linux distributions of R can be installed on a machine without the tools needed to install packages: a possible remedy is to do a complete install of R which should bring in all those tools as dependencies.

On Windows: install.packages tries to detect if you have write permission on the library directories specified, but Windows reports unreliably. If there is only one library directory (the default), R tries to find out by creating a test directory, but even this need not be the whole story: you may have permission to write in a library directory but lack permission to write binary files (such as '.dll' files) there. See the 'R for Windows FAQ' for workarounds.

See Also

update.packages, available.packages, download.packages, installed.packages, contrib.url.

See download.file for how to handle proxies and other options to monitor file transfers.

untar for manually unpacking source package tarballs.

INSTALL, REMOVE, remove.packages, library, .packages, read.dcf

The 'R Installation and Administration' manual for how to set up a repository.

Examples

```
## Not run:
## A Linux example for Fedora's layout of udunits2 headers.
install.packages(c("ncdf4", "RNetCDF"),
  configure.args = c(RNetCDF = "--with-netcdf-include=/usr/include/udunits2"))

## End(Not run)
```

`installed.packages` *Find Installed Packages*

Description

Find (or retrieve) details of all packages installed in the specified libraries.

Usage

```
installed.packages(lib.loc = NULL, priority = NULL,
                   noCache = FALSE, fields = NULL,
                   subarch = .Platform$r_arch, ...)
```

Arguments

`lib.loc`	character vector describing the location of R library trees to search through, or `NULL` for all known trees (see `.libPaths`).
`priority`	character vector or `NULL` (default). If non-null, used to select packages; `"high"` is equivalent to `c("base", "recommended")`. To select all packages without an assigned priority use `priority = NA_character_`.
`noCache`	Do not use cached information, nor cache it.
`fields`	a character vector giving the fields to extract from each package's 'DESCRIPTION' file in addition to the default ones, or `NULL` (default). Unavailable fields result in `NA` values.
`subarch`	character string or `NULL`. If non-null and non-empty, used to select packages which are installed for that sub-architecture.
`...`	allows unused arguments to be passed down from other functions.

Details

`installed.packages` scans the 'DESCRIPTION' files of each package found along `lib.loc` and returns a matrix of package names, library paths and version numbers.

The information found is cached (by library) for the R session and specified `fields` argument, and updated only if the top-level library directory has been altered, for example by installing or removing a package. If the cached information becomes confused, it can be avoided by specifying `noCache = TRUE`.

Value

A matrix with one row per package, row names the package names and column names (currently) `"Package"`, `"LibPath"`, `"Version"`, `"Priority"`, `"Depends"`, `"Imports"`, `"LinkingTo"`, `"Suggests"`, `"Enhances"`, `"OS_type"`, `"License"` and `"Built"` (the R version the package was built under). Additional columns can be specified using the `fields` argument.

Note

This needs to read several files per installed package, which will be slow on Windows and on some network-mounted file systems.

It will be slow when thousands of packages are installed, so do not use it to find out if a named package is installed (use `find.package` or `system.file`) nor to find out if a package is usable (call `requireNamespace` or `require` and check the return value) nor to find details of a small number of packages (use `packageDescription`).

See Also

`update.packages`, `install.packages`, `INSTALL`, `REMOVE`.

Examples

```
## confine search to .Library for speed
str(ip <- installed.packages(.Library, priority = "high"))
ip[, c(1,3:5)]
plic <- installed.packages(.Library, priority = "high", fields = "License")
## what licenses are there:
table( plic[, "License"] )

## Recommended setup (by many pros):
## Keep packages that come with R (priority="high") and all others separate!
## Consequently, .Library, R's "system" library, shouldn't have any
## non-"high"-priority packages :
pSys <- installed.packages(.Library, priority = NA_character_)
length(pSys) == 0 # TRUE under such a setup
```

---

isS3method *Is 'method' the Name of an S3 Method?*

Description

Checks if `method` is the name of a valid / registered S3 method. Alternatively, when `f` and `class` are specified, it is checked if `f` is the name of an S3 generic function and `paste(f, class, sep=".")` is a valid S3 method.

Usage

```
isS3method(method, f, class, envir = parent.frame())
```

Arguments

`method`	a character string, typically of the form *"fn.class"*. If omitted, `f` and `class` have to be specified instead.
`f`	optional character string, typically specifying an S3 generic function. Used, when `method` is not specified.

class	optional character string, typically specifying an S3 class name. Used, when method is not specified.
envir	the environment in which the method and its generic are searched first, as in getS3method().

Value

logical TRUE or FALSE

See Also

methods, getS3method.

Examples

```
isS3method("t")           # FALSE - it is an S3 generic
isS3method("t.default")   # TRUE
isS3method("t.ts")        # TRUE
isS3method("t.test")      # FALSE
isS3method("t.data.frame")# TRUE
isS3method("t.lm")        # FALSE - not existing
isS3method("t.foo.bar")   # FALSE - not existing

## S3 methods with "4 parts" in their name:
ff <- c("as.list", "as.matrix", "is.na", "row.names", "row.names<-")
for(m in ff) if(isS3method(m)) stop("wrongly declared an S3 method: ", m)
(m4 <- paste(ff, "data.frame", sep="."))
for(m in m4) if(!isS3method(m)) stop("not an S3 method: ", m)
```

---

isS3stdGeneric *Check if a Function Acts as an S3 Generic*

Description

Determines whether f acts as a standard S3-style generic function.

Usage

```
isS3stdGeneric(f)
```

Arguments

f	a function object

Details

A closure is considered a standard S3 generic if the first expression in its body calls `UseMethod`. Functions which perform operations before calling `UseMethod` will not be considered "standard" S3 generics.

If `f` is currently being traced, i.e., inheriting from class `"traceable"`, the definition of the original untraced version of the function is used instead.

Value

If `f` is an S3 generic, a logical containing `TRUE` with the name of the S3 generic (the string passed to `UseMethod`). Otherwise, `FALSE` (unnamed).

`LINK` *Create Executable Programs on Unix-alikes*

Description

Front-end for creating executable programs on unix-alikes, i.e., not on Windows.

Usage

```
R CMD LINK [options] linkcmd
```

Arguments

`linkcmd`	a list of commands to link together suitable object files (include library objects) to create the executable program.
`options`	further options to control the linking, or for obtaining information about usage and version.

Details

The linker front-end is useful in particular when linking against the R shared or static library: see the examples.

The actual linking command is constructed by the version of `libtool` installed at '*R_HOME*`/bin`'.

`R CMD LINK --help` gives usage information.

Note

Some binary distributions of R have `LINK` in a separate bundle, e.g. an `R-devel` RPM.

This is not available on Windows.

See Also

`COMPILE`.

Examples

```
## Not run: ## examples of front-ends linked against R.
## First a C program
CC=`R CMD config CC`
R CMD LINK $CC -o foo foo.o `R CMD config --ldflags`

## if Fortran code has been compiled into ForFoo.o
FLIBS=`R CMD config FLIBS`
R CMD LINK $CC -o foo foo.o ForFoo.o `R CMD config --ldflags` $FLIBS

## And for a C++ front-end
CXX=`R CMD config CXX`
R CMD COMPILE foo.cc
R CMD LINK $CXX -o foo foo.o `R CMD config --ldflags`

## End(Not run)
```

---

localeToCharset *Select a Suitable Encoding Name from a Locale Name*

Description

This functions aims to find a suitable coding for the locale named, by default the current locale, and if it is a UTF-8 locale a suitable single-byte encoding.

Usage

```
localeToCharset(locale = Sys.getlocale("LC_CTYPE"))
```

Arguments

`locale`	character string naming a locale.

Details

The operation differs by OS.

On Windows, a locale is specified like `"English_United Kingdom.1252"`. The final component gives the codepage, and this defines the encoding.

On Unix-alikes: Locale names are normally like `es_MX.iso88591`. If final component indicates an encoding and it is not `utf8` we just need to look up the equivalent encoding name. Otherwise, the language (here `es`) is used to choose a primary or fallback encoding.

In the `C` locale the answer will be `"ASCII"`.

Value

A character vector naming an encoding and possibly a fallback single-encoding, `NA` if unknown.

Note

The encoding names are those used by `libiconv`, and ought also to work with `glibc` but maybe not with commercial Unixen.

See Also

`Sys.getlocale`, `iconv`.

Examples

```
localeToCharset()
```

---

ls.str *List Objects and their Structure*

Description

`ls.str` and `lsf.str` are variations of `ls` applying `str()` to each matched name: see section Value.

Usage

```
ls.str(pos = -1, name, envir, all.names = FALSE,
       pattern, mode = "any")

lsf.str(pos = -1, envir, ...)

## S3 method for class 'ls_str'
print(x, max.level = 1, give.attr = FALSE, ...,
      digits = max(1, getOption("str")$digits.d))
```

Arguments

`pos`	integer indicating `search` path position, or `-1` for the current environment.
`name`	optional name indicating `search` path position, see `ls`.
`envir`	environment to use, see `ls`.
`all.names`	logical indicating if names which begin with a `.` are omitted; see `ls`.
`pattern`	a regular expression passed to `ls`. Only names matching `pattern` are considered.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default 1: Display only the first nested level.
`give.attr`	logical; if `TRUE` (default), show attributes as sub structures.
`mode`	character specifying the `mode` of objects to consider. Passed to `exists` and `get`.
`x`	an object of class `"ls_str"`.
`...`	further arguments to pass. `lsf.str` passes them to `ls.str` which passes them on to `ls`. The (non-exported) print method `print.ls_str` passes them to `str`.
`digits`	the number of significant digits to use for printing.

Value

`ls.str` and `lsf.str` return an object of class `"ls_str"`, basically the character vector of matching names (functions only for `lsf.str`), similarly to `ls`, with a `print()` method that calls `str()` on each object.

Author(s)

Martin Maechler

See Also

`str`, `summary`, `args`.

Examples

```
require(stats)

lsf.str()  #- how do the functions look like which I am using?
ls.str(mode = "list")   #- what are the structured objects I have defined?

## create a few objects
example(glm, echo = FALSE)
ll <- as.list(LETTERS)
print(ls.str(), max.level = 0)# don't show details

## which base functions have "file" in their name ?
lsf.str(pos = length(search()), pattern = "file")

## demonstrating that  ls.str() works inside functions
## ["browser/debug mode"]:
tt <- function(x, y = 1) { aa <- 7; r <- x + y; ls.str() }
(nms <- sapply(strsplit(capture.output(tt(2))," *: *"), `[`, 1))
stopifnot(nms == c("aa", "r","x","y"))
```

---

maintainer *Show Package Maintainer*

Description

Show the name and email address of the maintainer of an installed package.

Usage

```
maintainer(pkg)
```

Arguments

pkg	a character string, the name of an installed package.

Details

Accesses the package description to return the name and email address of the maintainer.

Questions about contributed packages should often be addressed to the package maintainer; questions about base packages should usually be addressed to the R-help or R-devel mailing lists. Bug reports should be submitted using the `bug.report` function.

Value

A character string giving the name and email address of the maintainer of the package, or `NA_character_` if no such package is installed.

Author(s)

David Scott <d.scott@auckland.ac.nz> from code on R-help originally due to Charlie Sharpsteen <source@sharpsteen.net>; multiple corrections by R-core.

References

https://stat.ethz.ch/pipermail/r-help/2010-February/230027.html

See Also

`packageDescription`, `bug.report`

Examples

```
maintainer("MASS")
```

---

make.packages.html *Update HTML Package List*

Description

Re-create the HTML list of packages.

Usage

```
make.packages.html(lib.loc = .libPaths(), temp = FALSE,
                   verbose = TRUE, docdir = R.home("doc"))
```

Arguments

`lib.loc`	character vector. List of libraries to be included.
`temp`	logical: should the package indices be created in a temporary location for use by the HTTP server?
`verbose`	logical. If true, print out a message.
`docdir`	If `temp` is false, directory in whose '`html`' directory the '`packages.html`' file is to be created/updated.

Details

This creates the 'packages.html' file, either a temporary copy for use by help.start, or the copy in 'R.home("doc")/html' (for which you will need write permission).

It can be very slow, as all the package 'DESCRIPTION' files in all the library trees are read.

For temp = TRUE there is some caching of information, so the file will only be re-created if lib.loc or any of the directories it lists have been changed.

Value

Invisible logical, with FALSE indicating a failure to create the file, probably due to lack of suitable permissions.

See Also

help.start

Examples

```
## Not run:
make.packages.html()
# this can be slow for large numbers of installed packages.

## End(Not run)
```

---

make.socket *Create a Socket Connection*

Description

With server = FALSE attempts to open a client socket to the specified port and host. With server = TRUE the R process listens on the specified port for a connection and then returns a server socket. It is a good idea to use on.exit to ensure that a socket is closed, as you only get 64 of them.

Usage

```
make.socket(host = "localhost", port, fail = TRUE, server = FALSE)
```

Arguments

host	name of remote host
port	port to connect to/listen on
fail	failure to connect is an error?
server	a server socket?

Value

An object of class "socket", a list with components:

socket	socket number. This is for internal use. On a Unix-alike it is a file descriptor.
port	port number of the connection.
host	name of remote computer.

Warning

I don't know if the connecting host name returned when server = TRUE can be trusted. I suspect not.

Author(s)

Thomas Lumley

References

Adapted from Luke Tierney's code for XLISP-Stat, in turn based on code from Robbins and Robbins "Practical UNIX Programming".

See Also

close.socket, read.socket.

Compiling in support for sockets was optional prior to R 3.3.0: see capabilities("sockets") to see if it is available.

Examples

```
daytime <- function(host = "localhost"){
    a <- make.socket(host, 13)
    on.exit(close.socket(a))
    read.socket(a)
}
## Official time (UTC) from US Naval Observatory
## Not run: daytime("tick.usno.navy.mil")
```

---

menu *Menu Interaction Function*

Description

menu presents the user with a menu of choices labelled from 1 to the number of choices. To exit without choosing an item one can select '0'.

Usage

```
menu(choices, graphics = FALSE, title = NULL)
```

Arguments

choices	a character vector of choices
graphics	a logical indicating whether a graphics menu should be used if available.
title	a character string to be used as the title of the menu. NULL is also accepted.

Details

If graphics = TRUE and a windowing system is available (Windows, macOS or X11 *via* Tcl/Tk) a listbox widget is used, otherwise a text menu. It is an error to use menu in a non-interactive session.

Ten or fewer items will be displayed in a single column, more in multiple columns if possible within the current display width.

No title is displayed if title is NULL or "".

Value

The number corresponding to the selected item, or 0 if no choice was made.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

select.list, which is used to implement the graphical menu, and allows multiple selections.

Examples

```
## Not run:
switch(menu(c("List letters", "List LETTERS")) + 1,
       cat("Nothing done\n"), letters, LETTERS)

## End(Not run)
```

---

methods *List Methods for S3 Generic Functions or Classes*

Description

List all available methods for a S3 and S4 generic function, or all methods for an S3 or S4 class.

Usage

```
   methods(generic.function, class, all.names = FALSE, dropPath = FALSE)
.S3methods(generic.function, class, envir = parent.frame(),
                                    all.names = FALSE, dropPath = FALSE)

## S3 method for class 'MethodsFunction'
format(x, byclass = attr(x, "byclass"), ...)
## S3 method for class 'MethodsFunction'
print(x, byclass = attr(x, "byclass"), ...)
```

Arguments

generic.function	a generic function, or a character string naming a generic function.
class	a symbol or character string naming a class: only used if generic.function is not supplied.
envir	the environment in which to look for the definition of the generic function, when the generic function is passed as a character string.
all.names	a logical indicating if all object names are returned. When FALSE as by default, names beginning with a '.' are omitted.
dropPath	a logical indicating if the search() path, apart from .GlobalEnv and package:base (i.e., baseenv()), should be skipped when searching for method definitions. The default FALSE is back compatible and typically desired for print()ing, with or without asterisk; dropPath=TRUE has been hard coded in R 4.3.0 and is faster for non-small search() paths.
x	typically the result of methods(..), an R object of S3 class "MethodsFunction", see 'Value' below.
byclass	an optional logical allowing to override the "byclass" attribute determining how the result is printed, see 'Details'.
...	potentially further arguments passed to and from methods; unused currently.

Details

methods() finds S3 and S4 methods associated with either the generic.function or class argument. Methods found are those provided by all loaded namespaces via registration, see UseMethod; normally, this includes all packages on the current search() path. .S3methods() finds only S3 methods, .S4methods() finds only S4 methods.

When invoked with the generic.function argument, the "byclass" attribute (see Details) is FALSE, and the print method by default displays the signatures (full names) of S3 and S4 methods. S3 methods are printed by pasting the generic function and class together, separated by a '.', as generic.class. The S3 method name is followed by an asterisk * if the method definition is not exported from the package namespace in which the method is defined. S4 method signatures are printed as generic,class-method; S4 allows for multiple dispatch, so there may be several classes in the signature generic,A,B-method.

When invoked with the class argument, "byclass" is TRUE, and the print method by default displays the names of the generic functions associated with the class, generic.

The source code for all functions is available. For S3 functions exported from the namespace, enter the method at the command line as `generic.class`. For S3 functions not exported from the namespace, see `getAnywhere` or `getS3method`. For S4 methods, see `getMethod`.

Help is available for each method, in addition to each generic. For interactive help, use the documentation shortcut `?` with the name of the generic and tab completion, `?"generic<tab>"` to select the method for which help is desired.

The S3 functions listed are those which *are named like methods* and may not actually be methods (known exceptions are discarded in the code).

Value

An object of class `"MethodsFunction"`, a character vector of method names with `"byclass"` and `"info"` attributes. The `"byclass"` attribute is a `logical` indicating if the results were obtained with argument `class` defined. The `"info"` attribute is a data frame with columns:

generic `character` vector of the names of the generic.

visible logical(), is the method "visible" to the user? When true, it typically is exported from the namespace of the package in which it is defined, and the package is `attach()`ed to the `search()` path.

isS4 logical(), true when the method is an S4 method.

from a `factor`, the location or package name where the method was found.

Note

The original `methods` function was written by Martin Maechler.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`S3Methods`, `class`, `getS3method`.

For S4, `getMethod`, `showMethods`, Introduction or `Methods_Details`.

Examples

```
methods(class = "MethodsFunction") # format and print

require(stats)

methods(summary)
methods(class = "aov")    # S3 class
## The same, with more details and more difficult to read:
print(methods(class = "aov"), byclass=FALSE)
methods("[[")             # uses C-internal dispatching
methods("$")
methods("$<-")            # replacement function
```

```
methods("+")              # binary operator
methods("Math")           # group generic
require(graphics)
methods(axis)             # looks like a generic, but is not

mf <- methods(format)     # quite a few; ... the last few :
tail(cbind(meth = format(mf)))

if(require(Matrix, quietly = TRUE)) {
print(methods(class = "Matrix"))  # S4 class
m <- methods(dim)         # S3 and S4 methods
print(m)
print(attr(m, "info"))    # more extensive information

## --> help(showMethods) for related examples
}
```

`mirrorAdmin` *Managing Repository Mirrors*

Description

Functions helping to maintain CRAN, some of them may also be useful for administrators of other repository networks.

Usage

```
mirror2html(mirrors = NULL, file = "mirrors.html",
  head = "mirrors-head.html", foot = "mirrors-foot.html")
checkCRAN(method)
```

Arguments

`mirrors`	A data frame, by default the CRAN list of mirrors is used.
`file`	A connection or a character string.
`head`	Name of optional header file.
`foot`	Name of optional footer file.
`method`	Download method, see `download.file`.

Details

`mirror2html` creates the HTML file for the CRAN list of mirrors and invisibly returns the HTML text.

`checkCRAN` performs a sanity checks on all CRAN mirrors.

`modifyList` *Recursively Modify Elements of a List*

Description

Modifies a possibly nested list recursively by changing a subset of elements at each level to match a second list.

Usage

```
modifyList(x, val, keep.null = FALSE)
```

Arguments

`x`	A named `list`, possibly empty.
`val`	A named list with components to replace corresponding components in `x` or add new components.
`keep.null`	If `TRUE`, `NULL` elements in `val` become `NULL` elements in `x`. Otherwise, the corresponding element, if present, is deleted from `x`.

Value

A modified version of `x`, with the modifications determined as follows (here, list elements are identified by their names). Elements in `val` which are missing from `x` are added to `x`. For elements that are common to both but are not both lists themselves, the component in `x` is replaced (or possibly deleted, depending on the value of `keep.null`) by the one in `val`. For common elements that are in both lists, `x[[name]]` is replaced by `modifyList(x[[name]], val[[name]])`.

Author(s)

Deepayan Sarkar <Deepayan.Sarkar@R-project.org>

Examples

```
foo <- list(a = 1, b = list(c = "a", d = FALSE))
bar <- modifyList(foo, list(e = 2, b = list(d = TRUE)))
str(foo)
str(bar)
```

news *Build and Query R or Package News Information*

Description

Build and query the news data base for R or add-on packages.

Usage

```
news(query, package = "R", lib.loc = NULL, format = NULL,
     reader = NULL, db = NULL)

## S3 method for class 'news_db'
print(x, doBrowse = interactive(),
      browser = getOption("browser"), ...)
```

Arguments

query	an optional expression for selecting news entries.
package	a character string giving the name of an installed add-on package, or "R" or "R-3" or "R-2".
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known.
format	Not yet used.
reader	Not yet used.
db, x	a news db obtained from news().
doBrowse	logical specifying that the news should be opened in the browser (by browseURL, accessible as via help.start) instead of printed to the console.
browser	the browser to be used, see browseURL.
...	potentially further arguments passed to print().

Details

If package is "R" (default), a news db is built with the news since the 4.0.0 release of R, corresponding to the 'NEWS' file in the R.home("doc") directory. "R-3" or "R-2" give the news for R 3.x.y or R 2.x.y respectively. Otherwise, if the given add-on package can be found in the given libraries, it is attempted to read its news in structured form from files 'inst/NEWS.Rd', 'NEWS.md' (since R version 3.6.0, needs packages **commonmark** and **xml2** to be available), 'NEWS' or 'inst/NEWS' (in that order). See section 'NEWS Formats' for the file specifications.

Using query, one can select news entries from the db. If missing or NULL, the complete db is returned. Otherwise, query should be an expression involving (a subset of) the variables Version, Category, Date and Text, and when evaluated within the db returning a logical vector with length the number of entries in the db. The entries for which evaluation gave TRUE are selected. When evaluating, Version and Date are coerced to numeric_version and Date objects, respectively, so that the comparison operators for these classes can be employed.

Value

A data frame inheriting from class "news_db", with character variables Version, Category, Date, Text and HTML, where the last two each contain the entry texts read (in plain-text and HTML format, respectively), and the other variables may be NA if they were missing or could not be determined. The data frame has attributes "package" (and "subset" if the query lead to proper subsetting).

NEWS Formats

'inst/NEWS.Rd': File 'inst/NEWS.Rd' should be an Rd file given the entries as Rd \itemize lists, grouped according to version using \section elements. Section titles start with a suitable prefix followed by a space and the version number, and optionally end with a (parenthesized) ISO 8601 (%Y-%m-%d, see strptime) format date (optionally including a note), for example:

```
\section{Changes in version 2.0 (2020-02-02, <note>)}{
  \itemize{
    \item ....
  }
}
```

The entries can be further grouped according to categories using \subsection elements named as the categories. The 'NEWS.Rd' file is assumed to be UTF-8-encoded (but an included \encoding specification takes precedence).

'NEWS.md': File 'NEWS.md' should contain the news in Markdown (following the CommonMark (https://commonmark.org/) specification), with the primary heading level giving the version number after a prefix followed by a space, and optionally followed by a space and a parenthesized ISO 8601 format date. Where available, secondary headings are taken to indicate categories. To accommodate for common practice, news entries are only split down to the category level.

'NEWS': The plain text 'NEWS' files in add-on packages use a variety of different formats; the default news reader should be capable to extract individual news entries from a majority of packages from the standard repositories, which use (slight variations of) the following format:

- Entries are grouped according to version, with version header "Changes in version" at the beginning of a line, followed by a version number, optionally followed by an ISO 8601 format date, possibly parenthesized.
- Entries may be grouped according to category, with a category header (different from a version header) starting at the beginning of a line.
- Entries are written as itemize-type lists, using one of 'o', '*', '-' or '+' as item tag. Entries must be indented, and ideally use a common indentation for the item texts.

Package **tools** provides an (internal) utility function news2Rd to convert plain text 'NEWS' files to Rd. For 'NEWS' files in a format which can successfully be handled by the default reader, package maintainers can use tools:::news2Rd(dir, "NEWS.Rd"), possibly with additional argument codify = TRUE, with dir a character string specifying the path to a package's root directory. Upon success, the 'NEWS.Rd' file can further be improved and then be moved to the 'inst' subdirectory of the package source directory.

Additional formats and readers may be supported in the future.

Examples

```
## Build a db of all R news entries.
db <- news()

## Bug fixes with PR number in 4.0.0.
db4 <- news(Version == "4.0.0" & grepl("^BUG", Category) & grepl("PR#", Text),
            db = db)
nrow(db4)

## print db4 to show in an HTML browser.

## News from a date range ('Matrix' is there in a regular R installation):
if(length(iM <- find.package("Matrix", quiet = TRUE)) && nzchar(iM)) {
   dM <- news(package="Matrix")
   stopifnot(identical(dM, news(db=dM)))
   dM2014 <- news("2014-01-01" <= Date & Date <= "2014-12-31", db = dM)
   stopifnot(paste0("1.1-", 2:4) %in% dM2014[,"Version"])
}

## Which categories have been in use? % R-core maybe should standardize a bit more
sort(table(db[, "Category"]), decreasing = TRUE)
## Entries with version >= 4.0.0
table(news(Version >= "4.0.0", db = db)$Version)


## do the same for R 3.x.y, more slowly
db3 <- news(package = "R-3")
sort(table(db3[, "Category"]), decreasing = TRUE)
## Entries with version >= 3.6.0
table(news(Version >= "3.6.0", db = db3)$Version)
```

---

nsl *Look up the IP Address by Hostname (on Unix-alikes)*

Description

Interface to the system `gethostbyname`, currently available only on unix-alikes, i.e., not on Windows.

Usage

```
nsl(hostname)
```

Arguments

`hostname`	the name of the host.

Details

This was included as a test of internet connectivity, to fail if the node running R is not connected. It will also return NULL if BSD networking is not supported, including the header file 'arpa/inet.h'.

This function is not available on Windows.

Value

The IP address, as a character string, or NULL if the call fails.

Examples

```
if(.Platform$OS.type == "unix") # includes Mac
  print( nsl("www.r-project.org") )
```

---

object.size *Report the Space Allocated for an Object*

Description

Provides an estimate of the memory that is being used to store an R object.

Usage

```
object.size(x)

## S3 method for class 'object_size'
format(x, units = "b", standard = "auto", digits = 1L, ...)
## S3 method for class 'object_size'
print(x, quote = FALSE, units = "b", standard = "auto",
      digits = 1L, ...)
```

Arguments

x	an R object.
quote	logical, indicating whether or not the result should be printed with surrounding quotes.
units	the units to be used in formatting and printing the size. Allowed values for the different standards are standard = "legacy": "b", "Kb", "Mb", "Gb", "Tb", "Pb", "B", "KB", "MB", "GB", "TB" and "PB". standard = "IEC": "B", "KiB", "MiB", "GiB", "TiB", "PiB", "EiB", "ZiB" and "YiB". standard = "SI": "B", "kB", "MB", "GB", "TB", "PB", "EB", "ZB", "YB", "RB", and "QB".

	For all standards, `units = "auto"` is also allowed. If `standard = "auto"`, any of the "legacy" and IEC units are allowed. See 'Formatting and printing object sizes' for details.
`standard`	the byte-size unit standard to be used. A character string, possibly abbreviated from `"legacy"`, `"IEC"`, `"SI"` and `"auto"`. See 'Formatting and printing object sizes' for details.
`digits`	the number of digits after the decimal point, passed to `round`.
`...`	arguments to be passed to or from other methods.

Details

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (Sharing amongst elements of a character vector is taken into account, but not that between character vectors in a single object.)

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Associated space (e.g., the environment of a function and what the pointer in a `EXTPTRSXP` points to) is not included in the calculation.

Object sizes are larger on 64-bit builds than 32-bit ones, but will very likely be the same on different platforms with the same word length and pointer size.

Sizes of objects using a compact internal representation may be over-estimated.

Value

An object of class `"object_size"` with a length-one double value, an estimate of the memory allocation attributable to the object in bytes.

Formatting and printing object sizes

Object sizes can be formatted using byte-size units from R's legacy standard, the IEC standard, or the SI standard. As illustrated by below tables, the legacy and IEC standards use *binary* units (multiples of 1024), whereas the SI standard uses *decimal* units (multiples of 1000).

For methods `format` and `print`, argument `standard` specifies which standard to use and argument `units` specifies which byte-size unit to use. `units = "auto"` chooses the largest units in which the result is one or more (before rounding). Byte sizes are rounded to `digits` decimal places. `standard = "auto"` chooses the standard based on `units`, if possible, otherwise, the legacy standard is used.

Summary of R's legacy and IEC units:

object size	legacy	IEC
1	1 bytes	1 B
1024	1 Kb	1 KiB
1024^2	1 Mb	1 MiB
1024^3	1 Gb	1 GiB
1024^4	1 Tb	1 TiB
1024^5	1 Pb	1 PiB

1024^6	1 EiB
1024^7	1 ZiB
1024^8	1 YiB

Summary of SI units:

object size	SI
1	1 B
1000	1 kB
1000^2	1 MB
1000^3	1 GB
1000^4	1 TB
1000^5	1 PB
1000^6	1 EB
1000^7	1 ZB
1000^8	1 YB
1000^9	1 RB
1000^10	1 QB

Author(s)

R Core; Henrik Bengtsson for the non-legacy `standards`.

References

The wikipedia page, https://en.wikipedia.org/wiki/Binary_prefix, is extensive on the different standards, usages and their history.

See Also

`Memory-limits` for the design limitations on object size.

Examples

```
object.size(letters)
object.size(ls)
format(object.size(library), units = "auto")

sl <- object.size(rep(letters, 1000))

print(sl)                                    ## 209288 bytes
print(sl, units = "auto")                    ## 204.4 Kb
print(sl, units = "auto", standard = "IEC")  ## 204.4 KiB
print(sl, units = "auto", standard = "SI")   ## 209.3 kB

(fsl <- sapply(c("Kb", "KB", "KiB"),
               function(u) format(sl, units = u)))
stopifnot(identical( ## assert that all three are the same :
```

```
             unique(substr(as.vector(fsl), 1,5)),
             format(round(as.vector(sl)/1024, 1))))

## find the 10 largest objects in the base package
z <- sapply(ls("package:base"), function(x)
           object.size(get(x, envir = baseenv())))
if(interactive()) {
as.matrix(rev(sort(z))[1:10])
} else # (more constant over time):
    names(rev(sort(z))[1:10])
```

---

package.skeleton *Create a Skeleton for a New Source Package*

Description

package.skeleton automates some of the setup for a new source package. It creates directories, saves functions, data, and R code files to appropriate places, and creates skeleton help files and a 'Read-and-delete-me' file describing further steps in packaging.

Usage

```
package.skeleton(name = "anRpackage", list,
                 environment = .GlobalEnv,
                 path = ".", force = FALSE,
                 code_files = character(), encoding = "unknown")
```

Arguments

name	character string: the package name and directory name for your package. Must be a valid package name.
list	character vector naming the R objects to put in the package. Usually, at most one of list, environment, or code_files will be supplied. See 'Details'.
environment	an environment where objects are looked for. See 'Details'.
path	path to put the package directory in.
force	If FALSE will not overwrite an existing directory.
code_files	a character vector with the paths to R code files to build the package around. See 'Details'.
encoding	optionally a character string with an encoding for an optional 'Encoding:' line in 'DESCRIPTION' when non-ASCII characters will be used; typically one of "latin1", "latin2", or "UTF-8"; see the WRE manual.

Details

The arguments `list`, `environment`, and `code_files` provide alternative ways to initialize the package. If `code_files` is supplied, the files so named will be sourced to form the environment, then used to generate the package skeleton. Otherwise `list` defaults to the objects in `environment` (including those whose names start with `.`), but can be supplied to select a subset of the objects in that environment.

Stubs of help files are generated for functions, data objects, and S4 classes and methods, using the `prompt`, `promptClass`, and `promptMethods` functions. If an object from another package is intended to be imported and re-exported without changes, the `promptImport` function should be used after `package.skeleton` to generate a simple help file linking to the original one.

The package sources are placed in subdirectory `name` of `path`. If `code_files` is supplied, these files are copied; otherwise, objects will be dumped into individual source files. The file names in `code_files` should have suffix `".R"` and be in the current working directory.

The filenames created for source and documentation try to be valid for all OSes known to run R. Invalid characters are replaced by '_', invalid names are preceded by 'zz', names are converted to lower case (to avoid case collisions on case-insensitive file systems) and finally the converted names are made unique by `make.unique(sep = "_")`. This can be done for code and help files but not data files (which are looked for by name). Also, the code and help files should have names starting with an ASCII letter or digit, and this is checked and if necessary `z` prepended.

Functions with names starting with a dot are placed in file '`R/`*name*`-internal.R`'.

When you are done, delete the '`Read-and-delete-me`' file, as it should not be distributed.

Value

Used for its side-effects.

References

Read the 'Writing R Extensions' manual for more details.

Once you have created a *source* package you need to install it: see the 'R Installation and Administration' manual, `INSTALL` and `install.packages`.

See Also

`prompt`, `promptClass`, and `promptMethods`.

`package_native_routine_registration_skeleton` for helping in preparing packages with compiled code.

Examples

```
require(stats)
## two functions and two "data sets" :
f <- function(x, y) x+y
g <- function(x, y) x-y
d <- data.frame(a = 1, b = 2)
e <- rnorm(1000)
```

```
package.skeleton(list = c("f","g","d","e"), name = "mypkg")
```

---

packageDescription *Package Description*

Description

Parses and returns the 'DESCRIPTION' file of a package as a "packageDescription".

Utility functions return (transformed) parts of that.

Usage

```
packageDescription(pkg, lib.loc = NULL, fields = NULL,
                   drop = TRUE, encoding = "")
packageVersion(pkg, lib.loc = NULL)
packageDate(pkg, lib.loc = NULL,
            date.fields = c("Date", "Packaged", "Date/Publication", "Built"),
            tryFormats = c("%Y-%m-%d", "%Y/%m/%d", "%D", "%m/%d/%y"),
           desc = packageDescription(pkg, lib.loc=lib.loc, fields=date.fields))
asDateBuilt(built)
```

Arguments

pkg	a character string with the package name.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages and namespaces are searched before the libraries.
fields	a character vector giving the tags of fields to return (if other fields occur in the file they are ignored).
drop	If TRUE and the length of fields is 1, then a single character string with the value of the respective field is returned instead of an object of class "packageDescription".
encoding	If there is an Encoding field, to what encoding should re-encoding be attempted? If NA, no re-encoding. The other values are as used by iconv, so the default "" indicates the encoding of the current locale.
date.fields	character vector of field tags to be tried. The first for which as.Date(.) is not NA will be returned. (Partly experimental, see *Note*.)
tryFormats	date formats to try, see as.Date.character().
desc	optionally, a named list with components named from date.fields; where the default is fine, a complete packageDescription() maybe specified as well.
built	for asDateBuilt(), a character string as from packageDescription(*, fields="Built").

Details

A package will not be 'found' unless it has a 'DESCRIPTION' file which contains a valid Version field. Different warnings are given when no package directory is found and when there is a suitable directory but no valid 'DESCRIPTION' file.

An attached environment named to look like a package (e.g., package:utils2) will be ignored.

packageVersion() is a convenience shortcut, allowing things like if (packageVersion("MASS") < "7.3") { do.things } .

For packageDate(), if desc is valid, both pkg and lib.loc are not made use of.

Value

If a 'DESCRIPTION' file for the given package is found and can successfully be read, packageDescription returns an object of class "packageDescription", which is a named list with the values of the (given) fields as elements and the tags as names, unless drop = TRUE.

If parsing the 'DESCRIPTION' file was not successful, it returns a named list of NAs with the field tags as names if fields is not null, and NA otherwise.

packageVersion() returns a (length-one) object of class "package_version".

packageDate() will return a "Date" object from as.Date() or NA.

asDateBuilt(built) returns a "Date" object or signals an error if built is invalid.

Note

The default behavior of packageDate(), notably for date.fields, is somewhat experimental and may change.

See Also

read.dcf

Examples

```
packageDescription("stats")
packageDescription("stats", fields = c("Package", "Version"))

packageDescription("stats", fields = "Version")
packageDescription("stats", fields = "Version", drop = FALSE)

if(requireNamespace("MASS") && packageVersion("MASS") < "7.3.29")
  message("you need to update 'MASS'")

pu <- packageDate("utils")
str(pu)
stopifnot(identical(pu, packageDate(desc = packageDescription("utils"))),
          identical(pu, packageDate("stats"))) # as "utils" and "stats" are
                                # both 'base R' and "Built" at same time
```

packageName *Find Package Associated with an Environment*

Description

Many environments are associated with a package; this function attempts to determine that package.

Usage

```
packageName(env = parent.frame())
```

Arguments

env	The environment whose name we seek.

Details

Environment env would be associated with a package if topenv(env) is the namespace environment for that package. Thus when env is the environment associated with functions inside a package, or local functions defined within them, packageName will normally return the package name.

Not all environments are associated with a package: for example, the global environment, or the evaluation frames of functions defined there. packageName will return NULL in these cases.

Value

A length one character vector containing the name of the package, or NULL if there is no name.

See Also

getPackageName is a more elaborate function that can construct a name if none is found.

Examples

```
packageName()
packageName(environment(mean))
```

packageStatus *Package Management Tools*

Description

Summarize information about installed packages and packages available at various repositories, and automatically upgrade outdated packages.

Usage

```
packageStatus(lib.loc = NULL, repositories = NULL, method,
              type = getOption("pkgType"), ...)

## S3 method for class 'packageStatus'
summary(object, ...)

## S3 method for class 'packageStatus'
update(object, lib.loc = levels(object$inst$LibPath),
       repositories = levels(object$avail$Repository), ...)

## S3 method for class 'packageStatus'
upgrade(object, ask = TRUE, ...)
```

Arguments

lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
repositories	a character vector of URLs describing the location of R package repositories on the Internet or on the local machine. These should be full paths to the appropriate 'contrib' sections of the repositories. The default (NULL) derives URLs from option "repos" and type.
method	download method, see download.file.
type	type of package distribution: see install.packages.
object	an object of class "packageStatus" as returned by packageStatus.
ask	if TRUE, the user is prompted which packages should be upgraded and which not.
...	for packageStatus: arguments to be passed to available.packages and installed.packages. for the upgrade method, arguments to be passed to install.packages for other methods: currently not used.

Details

There are print and summary methods for the "packageStatus" objects: the print method gives a brief tabular summary and the summary method prints the results.

The update method updates the "packageStatus" object. The upgrade method is similar to update.packages: it offers to install the current versions of those packages which are not currently up-to-date.

Value

An object of class "packageStatus". This is a list with two components

inst	a data frame with columns as the *matrix* returned by installed.packages plus "Status", a factor with levels c("ok", "upgrade", "unavailable"). Only the newest version of each package is reported, in the first repository in which it appears.
avail	a data frame with columns as the *matrix* returned by available.packages plus "Status", a factor with levels c("installed", "not installed").

For the summary method the result is also of class "summary.packageStatus" with additional components

Libs	a list with one element for each library
Repos	a list with one element for each repository

with the elements being lists of character vectors of package name for each status.

See Also

installed.packages, available.packages

Examples

```
x <- packageStatus(repositories = contrib.url(findCRANmirror("web")))
print(x)
summary(x)

## Not run:
upgrade(x)
x <- update(x)
print(x)

## End(Not run)
```

---

page *Invoke a Pager on an R Object*

Description

Displays a representation of the object named by x in a pager *via* file.show.

Usage

```
page(x, method = c("dput", "print"), ...)
```

Arguments

x	An R object, or a character string naming an object.
method	The default method is to dump the object *via* dput. An alternative is to use print and capture the output to be shown in the pager. Can be abbreviated.
...	additional arguments for dput, print or file.show (such as title).

Details

If x is a length-one character vector, it is used as the name of an object to look up in the environment from which page is called. All other objects are displayed directly.

A default value of title is passed to file.show if one is not supplied in ....

See Also

file.show, edit, fix.

To go to a new page when graphing, see frame.

Examples

```
## Not run: ## four ways to look at the code of 'page'
page(page)             # as an object
page("page")           # a character string
v <- "page"; page(v)   # a length-one character vector
page(utils::page)      # a call

## End(Not run)
```

---

person *Persons*

Description

A class and utility methods for holding information about persons like name and email address.

Usage

```
person(given = NULL, family = NULL, middle = NULL,
       email = NULL, role = NULL, comment = NULL,
       first = NULL, last = NULL)

as.person(x)
## Default S3 method:
as.person(x)
```

```
## S3 method for class 'person'
format(x,
       include = c("given", "family", "email", "role", "comment"),
       braces = list(given = "", family = "", email = c("<", ">"),
                     role = c("[", "]"), comment = c("(", ")")),
       collapse = list(given = " ", family = " ", email = ", ",
                       role = ", ", comment = ", "),
       ...,
       style = c("text", "R")
)

## S3 method for class 'person'
toBibtex(object, escape = FALSE, ...)
```

Arguments

given	a character vector with the *given* names, or a list thereof.
family	a character string with the *family* name, or a list thereof.
middle	a character string with the collapsed middle name(s). Deprecated, see **Details**.
email	a character string (or vector) giving an e-mail address (each), or a list thereof.
role	a character vector specifying the role(s) of the person (see **Details**), or a list thereof.
comment	a character string (or vector) providing comments, or a list thereof.
first	a character string giving the first name. Deprecated, see **Details**.
last	a character string giving the last name. Deprecated, see **Details**.
x	an object for the `as.person` generic; a character string for the `as.person` default method; an object of class `"person"` otherwise.
include	a character vector giving the fields to be included when formatting.
braces	a list of characters (see **Details**).
collapse	a list of characters (see **Details**).
...	currently not used.
style	a character string specifying the print style, with `"R"` yielding formatting as R code.
object	an R object inhering from class `"person"`.
escape	a logical indicating whether non-ASCII characters should be translated to LaTeX escape sequences.

Details

Objects of class `"person"` can hold information about an arbitrary positive number of persons. These can be obtained by one call to `person()` with list arguments, or by first creating objects representing single persons and combining these via `c()`.

The format() method collapses information about persons into character vectors (one string for each person): the fields in include are selected, each collapsed to a string using the respective element of collapse and subsequently "embraced" using the respective element of braces, and finally collapsed into one string separated by white space. If braces and/or collapse do not specify characters for all fields, the defaults shown in the usage are imputed. If collapse is FALSE or NA the corresponding field is not collapsed but only the first element is used. The print() method calls the format() method and prints the result, the toBibtex() method creates a suitable BibTeX representation.

Person objects can be subscripted by fields (using $) or by position (using []).

as.person() is a generic function. Its default method tries to reverse the default person formatting, and can also handle formatted person entries collapsed by comma or "and" (with appropriate white space).

Personal names are rather tricky, e.g., https://en.wikipedia.org/wiki/Personal_name.

The current implementation (starting from R 2.12.0) of the "person" class uses the notions of *given* (including middle names) and *family* names, as specified by given and family respectively. Earlier versions used a scheme based on first, middle and last names, as appropriate for most of Western culture where the given name precedes the family name, but not universal, as some other cultures place it after the family name, or use no family name. To smooth the transition to the new scheme, arguments first, middle and last are still supported, but their use is deprecated and they must not be given in combination with the corresponding new style arguments. For persons which are not natural persons (e.g., institutions, companies, etc.) it is appropriate to use given (but not family) for the name, e.g., person("R Core Team", role = "aut").

The new scheme also adds the possibility of specifying *roles* based on a subset of the MARC Code List for Relators (https://www.loc.gov/marc/relators/relaterm.html). When giving the roles of persons in the context of authoring R packages, the following usage is suggested.

"aut" (Author) Use for full authors who have made substantial contributions to the package and should show up in the package citation.

"com" (Compiler) Use for persons who collected code (potentially in other languages) but did not make further substantial contributions to the package.

"cph" (Copyright holder) Use for all copyright holders. This is a legal concept so should use the legal name of an institution or corporate body.

"cre" (Creator) Use for the package maintainer.

"ctb" (Contributor) Use for authors who have made smaller contributions (such as code patches etc.) but should not show up in the package citation.

"ctr" (Contractor) Use for authors who have been contracted to write (parts of) the package and hence do not own intellectual property.

"dtc" (Data contributor) Use for persons who contributed data sets for the package.

"fnd" (Funder) Use for persons or organizations that furnished financial support for the development of the package.

"rev" (Reviewer) Use for persons or organizations responsible for reviewing (parts of) the package.

"ths" (Thesis advisor) If the package is part of a thesis, use for the thesis advisor.

"trl" (Translator) If the R code is a translation from another language (typically S), use for the translator to R.

In the old scheme, person objects were used for single persons, and a separate "personList" class with corresponding creator personList() for collections of these. The new scheme employs a single class for information about an arbitrary positive number of persons, eliminating the need for the personList mechanism.

The comment field can be used for "arbitrary" additional information about persons. Elements named "ORCID" will be taken to give ORCID identifiers (see https://orcid.org/ for more information), and be displayed as the corresponding URIs by the print() and format() methods (see **Examples** below).

Value

person() and as.person() return objects of class "person".

See Also

citation

Examples

```
## Create a person object directly ...
p1 <- person("Karl", "Pearson", email = "pearson@stats.heaven")

## ... or convert a string.
p2 <- as.person("Ronald Aylmer Fisher")

## Combining and subsetting.
p <- c(p1, p2)
p[1]
p[-1]

## Extracting fields.
p$family
p$email
p[1]$email

## Specifying package authors, example from "boot":
## AC is the first author [aut] who wrote the S original.
## BR is the second author [aut], who translated the code to R [trl],
## and maintains the package [cre].
b <- c(person("Angelo", "Canty", role = "aut", comment =
         "S original, <http://statwww.epfl.ch/davison/BMA/library.html>"),
       person(c("Brian", "D."), "Ripley", role = c("aut", "trl", "cre"),
              comment = "R port", email = "ripley@stats.ox.ac.uk")
     )
b

## Formatting.
format(b)
format(b, include = c("family", "given", "role"),
   braces = list(family = c("", ","), role = c("(Role(s): ", ")")))

## Conversion to BibTeX author field.
```

```
paste(format(b, include = c("given", "family")), collapse = " and ")
toBibtex(b)

## ORCID identifiers.
(p3 <- person("Achim", "Zeileis",
              comment = c(ORCID = "0000-0003-0918-3766")))
```

---

personList *Collections of Persons (Older Interface)*

Description

Old interface providing functionality for information about collections of persons. Since R 2.14.0 person objects can be combined with the corresponding c method which supersedes the personList function.

Usage

```
personList(...)
as.personList(x)
```

Arguments

...	person objects (inheriting from class "person")
x	an object the elements of which are coercible via as.person

Value

a person object (inheriting from class "person")

See Also

person for the new functionality for representing and manipulating information about persons.

PkgUtils *Utilities for Building and Checking Add-on Packages*

Description

Utilities for checking whether the sources of an R add-on package work correctly, and for building a source package from them.

Usage

```
R CMD check [options] pkgdirs
R CMD build [options] pkgdirs
```

Arguments

pkgdirs	a list of names of directories with sources of R add-on packages. For check these can also be the filenames of compressed tar archives with extension '.tar.gz', '.tgz', '.tar.bz2' or '.tar.xz'.
options	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

R CMD check checks R add-on packages from their sources, performing a wide variety of diagnostic checks.

R CMD build builds R source tarballs. The name(s) of the packages are taken from the 'DESCRIPTION' files and not from the directory names. This works entirely on a copy of the supplied source directories.

Use R CMD *foo* --help to obtain usage information on utility *foo*, notably the possible options.

The defaults for some of the options to R CMD build can be set by environment variables _R_BUILD_RESAVE_DATA_ and _R_BUILD_COMPACT_VIGNETTES_: see 'Writing R Extensions'. Many of the checks in R CMD check can be turned off or on by environment variables: see Chapter 'Tools' of the 'R Internals' manual.

By default R CMD build uses the "internal" option to tar to prepare the tarball. An external tar program can be specified by the R_BUILD_TAR environment variable. This may be substantially faster for very large packages, and can be needed for packages with long path names (over 100 bytes) or very large files (over 8GB): however, the resulting tarball may not be portable.

R CMD check by default unpacks tarballs by the internal untar function: if needed an external tar command can be specified by the environment variable R_INSTALL_TAR: please ensure that it can handle the type of compression used on the tarball. (This is sometimes needed for tarballs containing invalid or unsupported sections, and can be faster on very large tarballs. Setting R_INSTALL_TAR to 'tar.exe' has been needed to overcome permissions issues on some Windows systems.)

Note

Only on Windows: They make use of a temporary directory specified by the environment variable TMPDIR and defaulting to 'c:/TEMP'. Do ensure that if set forward slashes are used.

See Also

The sections on 'Checking and building packages' and 'Processing documentation files' in 'Writing R Extensions': RShowDoc("R-exts").

process.events *Trigger Event Handling*

Description

R front ends like the Windows GUI handle key presses and mouse clicks through "events" generated by the OS. These are processed automatically by R at intervals during computations, but in some cases it may be desirable to trigger immediate event handling. The `process.events` function does that.

Usage

```
process.events()
```

Details

This is a simple wrapper for the C API function `R_ProcessEvents`. As such, it is possible that it will not return if the user has signalled to interrupt the calculation.

Value

`NULL` is returned invisibly.

Author(s)

Duncan Murdoch

See Also

See 'Writing R Extensions' and the 'R for Windows FAQ' for more discussion of the `R_ProcessEvents` function.

prompt *Produce Prototype of an R Documentation File*

Description

Facilitate the constructing of files documenting R objects.

Usage

```
prompt(object, filename = NULL, name = NULL, ...)

## Default S3 method:
prompt(object, filename = NULL, name = NULL,
       force.function = FALSE, ...)

## S3 method for class 'data.frame'
prompt(object, filename = NULL, name = NULL, ...)

promptImport(object, filename = NULL, name = NULL,
 importedFrom = NULL, importPage = name, ...)
```

Arguments

object	an R object, typically a function for the default method. Can be missing when name is specified.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the object.
force.function	a logical. If TRUE, treat object as function in any case.
...	further arguments passed to or from other methods.
importedFrom	a character string naming the package from which object was imported. Defaults to the environment of object if object is a function.
importPage	a character string naming the help page in the package from which object was imported.

Details

Unless filename is NA, a documentation shell for object is written to the file specified by filename, and a message about this is given. For function objects, this shell contains the proper function and argument names. R documentation files thus created still need to be edited and moved into the 'man' subdirectory of the package containing the object to be documented.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

When prompt is used in for loops or scripts, the explicit name specification will be useful.

The importPage argument for promptImport needs to give the base of the name of the help file of the original help page. For example, the approx function is documented in 'approxfun.Rd' in the **stats** package, so if it were imported and re-exported it should have importPage = "approxfun". Objects that are imported from other packages are not normally documented unless re-exported.

Value

If filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Warning

The default filename may not be a valid filename under limited file systems (e.g., those on Windows).

Currently, calling `prompt` on a non-function object assumes that the object is in fact a data set and hence documents it as such. This may change in future versions of R. Use `promptData` to create documentation skeletons for data sets.

Note

The documentation file produced by `prompt.data.frame` does not have the same format as many of the data frame documentation files in the **base** package. We are trying to settle on a preferred format for the documentation.

Author(s)

Douglas Bates for `prompt.data.frame`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`promptData`, `help` and the chapter on 'Writing R documentation files' in the 'Writing R Extensions' manual: `RShowDoc("R-exts")`.

For creation of many help pages (for a package), see `package.skeleton`.

To prompt the user for input, see `readline`.

Examples

```
require(graphics)

prompt(plot.default)
prompt(interactive, force.function = TRUE)
unlink("plot.default.Rd")
unlink("interactive.Rd")

prompt(women) # data.frame
unlink("women.Rd")

prompt(sunspots) # non-data.frame data
unlink("sunspots.Rd")


## Not run:
## Create a help file for each function in the .GlobalEnv:
for(f in ls()) if(is.function(get(f))) prompt(name = f)

## End(Not run)
```

promptData *Generate Outline Documentation for a Data Set*

Description

Generates a shell of documentation for a data set.

Usage

```
promptData(object, filename = NULL, name = NULL)
```

Arguments

object	an R object to be documented as a data set.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the object.

Details

Unless filename is NA, a documentation shell for object is written to the file specified by filename, and a message about this is given.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

Currently, only data frames are handled explicitly by the code.

Value

If filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

See Also

prompt

Examples

```
promptData(sunspots)
unlink("sunspots.Rd")
```

promptPackage *Generate a Shell for Documentation of a Package*

Description

Generates a prototype of a package overview help page using Rd macros that dynamically extract information from package metadata when building the package.

Usage

```
promptPackage(package, lib.loc = NULL, filename = NULL,
              name = NULL, final = FALSE)
```

Arguments

package	a character string with the name of the package to be documented.
lib.loc	ignored.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the help topic; defaults to "*pkgname*-package", which is the required \alias for the overview help page.
final	a logical value indicating whether to attempt to create a usable version of the help topic, rather than just a shell.

Details

Unless filename is NA, a documentation shell for package is written to the file specified by filename, and a message about this is given.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

If final is TRUE, the generated documentation will not include the place-holder slots for manual editing, it will be usable as-is. In most cases a manually edited file is preferable (but final = TRUE is certainly less work).

Value

If filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

See Also

prompt, package.skeleton

Examples

```
filename <- tempfile()
promptPackage("utils", filename = filename)
file.show(filename)
unlink(filename)
```

`Question` *Documentation Shortcuts*

Description

These functions provide access to documentation. Documentation on a topic with name `name` (typically, an R object or a data set) can be displayed by either `help("name")` or `?name`.

Usage

```
?topic

type?topic
```

Arguments

`topic`	Usually, a name or character string specifying the topic for which help is sought. Alternatively, a function call to ask for documentation on a corresponding S4 method: see the section on S4 method documentation. The calls *pkg*`::`*topic* and *pkg*`:::`*topic* are treated specially, and look for help on `topic` in package *pkg*.
`type`	the special type of documentation to use for this topic; for example, type `package` will look for the overview help page of a package named `topic`, and if the type is `class`, documentation is provided for the S4 class with name `topic`. See the Section 'S4 Method Documentation' for the uses of `type` to get help on formal methods, including `methods?`*function* and `method?`*call*.

Details

This is a shortcut to `help` and uses its default type of help.

Some topics need to be quoted (by backticks) or given as a character string. There include those which cannot syntactically appear on their own such as unary and binary operators, `function` and control-flow reserved words (including `if`, `else` `for`, `in`, `repeat`, `while`, `break` and `next`). The other `reserved` words can be used as if they were names, for example `TRUE`, `NA` and `Inf`.

S4 Method Documentation

Authors of formal ('S4') methods can provide documentation on specific methods, as well as overall documentation on the methods of a particular function. The `"?"` operator allows access to this documentation in three ways.

The expression `methods?`*f* will look for the overall documentation methods for the function *f*. Currently, this means the documentation file containing the alias *f*`-methods`.

There are two different ways to look for documentation on a particular method. The first is to supply the `topic` argument in the form of a function call, omitting the `type` argument. The effect is to look for documentation on the method that would be used if this function call were actually evaluated. See the examples below. If the function is not a generic (no S4 methods are defined for it), the help reverts to documentation on the function name.

The `"?"` operator can also be called with `type` supplied as `method`; in this case also, the `topic` argument is a function call, but the arguments are now interpreted as specifying the class of the argument, not the actual expression that will appear in a real call to the function. See the examples below.

The first approach will be tedious if the actual call involves complicated expressions, and may be slow if the arguments take a long time to evaluate. The second approach avoids these issues, but you do have to know what the classes of the actual arguments will be when they are evaluated.

Both approaches make use of any inherited methods; the signature of the method to be looked up is found by using `selectMethod` (see the documentation for `getMethod`). A limitation is that methods in packages (as opposed to regular functions) will only be found if the package exporting them is on the search list, even if it is specified explicitly using the `?package::generic()` notation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`help`

`??` for finding help pages on a vague topic.

Examples

```
?lapply

?"for"                  # but quotes/backticks are needed
?`+`

?women                  # information about data set "women"

package?parallel        # overview help page of package 'parallel'

## Not run:
require(methods)
## define a S4 generic function and some methods
combo <- function(x, y) c(x, y)
```

```
setGeneric("combo")
setMethod("combo", c("numeric", "numeric"), function(x, y) x+y)

## assume we have written some documentation
## for combo, and its methods ....

?combo  # produces the function documentation

methods?combo  # looks for the overall methods documentation

method?combo("numeric", "numeric")  # documentation for the method above

?combo(1:10, rnorm(10))  # ... the same method, selected according to
                         # the arguments (one integer, the other numeric)

?combo(1:10, letters)    # documentation for the default method

## End(Not run)
```

---

rcompgen *A Completion Generator for R*

Description

This page documents a mechanism to generate relevant completions from a partially completed command line. It is not intended to be useful by itself, but rather in conjunction with other mechanisms that use it as a backend. The functions listed in the usage section provide a simple control and query mechanism. The actual interface consists of a few unexported functions described further down.

Usage

```
rc.settings(ops, ns, args, dots, func, ipck, S3, data, help,
            argdb, fuzzy, quotes, files, backtick)

rc.status()
rc.getOption(name)
rc.options(...)

.DollarNames(x, pattern)
.AtNames(x, pattern)

## Default S3 method:
.DollarNames(x, pattern = "")
## S3 method for class 'list'
.DollarNames(x, pattern = "")
## S3 method for class 'environment'
.DollarNames(x, pattern = "")
```

```
## Default S3 method:
.AtNames(x, pattern = "")

findMatches(pattern, values, fuzzy, backtick)
```

Arguments

ops	Logical flag. Activates completion after the $ and @ operators.
ns	Logical flag. Controls namespace related completions.
args	Logical flag. Enables completion of function arguments.
dots	Logical flag. If disabled, drops ... from list of function arguments. Relevant only if args is enabled.
func	Logical flag. Enables detection of functions. If enabled, a customizable extension ("(" by default) is appended to function names. The process of determining whether a potential completion is a function requires evaluation, including for lazy loaded symbols. This is undesirable for large objects, because of potentially wasteful use of memory in addition to the time overhead associated with loading. For this reason, this feature is disabled by default.
S3	Logical flag. When args = TRUE, activates completion on arguments of all S3 methods (otherwise just the generic, which usually has very few arguments).
ipck	Logical flag. Enables completion of installed package names inside library and require.
data	Logical flag. Enables completion of data sets (including those already visible) inside data.
help	Logical flag. Enables completion of help requests starting with a question mark, by looking inside help index files.
argdb	Logical flag. When args = TRUE, completion is attempted on function arguments. Generally, the list of valid arguments is determined by dynamic calls to args. While this gives results that are technically correct, the use of the ... argument often hides some useful arguments. To give more flexibility in this regard, an optional table of valid arguments names for specific functions is retained internally. Setting argdb = TRUE enables preferential lookup in this internal data base for functions with an entry in it. Of course, this is useful only when the data base contains information about the function of interest. Some functions are already included, and more can be added by the user through the unexported function .addFunctionInfo (see below).
fuzzy	Logical flag. Enables fuzzy matching, where close but non-exact matches (e.g., with different case) are considered if no exact matches are found. This feature is experimental and the details can change. In findMatches, this argument defaults to the current setting.
backtick	Logical flag. If enabled, non-syntactic completions are wrapped in backticks to make them syntactically valid. This is useful only if the backend can handle such completions. In findMatches, this argument defaults to the current setting.

quotes	Logical flag. Enables completion in R code when inside quotes. This normally leads to filename completion, but can be otherwise depending on context (for example, when the open quote is preceded by ?), help completion is invoked. Setting this to FALSE relegates completion to the underlying completion front-end, which may do its own processing (for example, readline on Unix-alikes will do filename completion).
files	Logical flag. Deprecated. Use quotes instead.
name, ...	user-settable options. Currently valid names are function.suffix: default "(" funarg.suffix: default "=" package.suffix default "::" Usage is similar to that of options.
x	An R object for which valid names after "$" are computed and returned.
pattern	A regular expression. Only matching names are returned.
values	character string giving set of candidate values in which matches are to be found.

Details

There are several types of completion, some of which can be disabled using rc.settings. The arguments of rc.settings are all logical flags, turning specific optional completion features on and off. All settings are on by default except ipck, func, and fuzzy. Turn more off if your CPU cycles are valuable; you will still retain basic completion.

The most basic level, which can not be turned off once the completion functionality is activated, provides completion on names visible on the search path, along with a few special keywords (e.g., TRUE). This type of completion is not attempted if the partial 'word' (a.k.a. token) being completed is empty (since there would be too many completions). The more advanced types of completion are described below.

Completion after extractors $ and @: When the ops setting is turned on, completion after $ and @ is attempted. This requires the prefix to be evaluated, which is attempted unless it involves an explicit function call (implicit function calls involving the use of [, $, etc *do not* inhibit evaluation).

Valid completions after the $ and @ extractors are determined by the generic functions .DollarNames and .AtNames respectively. A few basic methods are provided, and more can be written for custom classes. The findMatches function can be useful for this purpose.

Completion inside namespaces: When the ns setting is turned on, completion inside namespaces is attempted when a token is preceded by the :: or ::: operators. Additionally, the basic completion mechanism is extended to include all loaded namespaces, i.e., foopkg:: becomes a valid completion of foo if "foopkg" is a loaded namespace.

The completion of package namespaces applies only to already loaded namespaces, i.e. if MASS is not loaded, MAS will not complete to MASS::. However, attempted completion *inside* an apparent namespace will attempt to load the namespace if it is not already loaded, e.g. trying to complete on MASS::fr will load MASS if it is not already loaded.

Completion for help items: When the help setting is turned on, completion on help topics is attempted when a token is preceded by ?. Prefixes (such as class, method) are supported, as well as quoted help topics containing special characters.

Completion of function arguments: When the `args` setting is turned on, completion on function arguments is attempted whenever deemed appropriate. The mechanism used will currently fail if the relevant function (at the point where completion is requested) was entered on a previous prompt (which implies in particular that the current line is being typed in response to a continuation prompt, usually `+`). Note that separation by newlines is fine.

The list of possible argument completions that is generated can be misleading. There is no problem for non-generic functions (except that `...` is listed as a completion; this is intentional as it signals the fact that the function can accept further arguments). However, for generic functions, it is practically impossible to give a reliable argument list without evaluating arguments (and not even then, in some cases), which is risky (in addition to being difficult to code, which is the real reason it hasn't even been tried), especially when that argument is itself an inline function call. Our compromise is to consider arguments of *all* currently available methods of that generic. This has two drawbacks. First, not all listed completions may be appropriate in the call currently being constructed. Second, for generics with many methods (like `print` and `plot`), many matches will need to be considered, which may take a noticeable amount of time. Despite these drawbacks, we believe this behaviour to be more useful than the only other practical alternative, which is to list arguments of the generic only.

Only S3 methods are currently supported in this fashion, and that can be turned off using the `S3` setting.

Since arguments can be unnamed in R function calls, other types of completion are also appropriate whenever argument completion is. Since there are usually many many more visible objects than formal arguments of any particular function, possible argument completions are often buried in a bunch of other possibilities. However, recall that basic completion is suppressed for blank tokens. This can be useful to list possible arguments of a function. For example, trying to complete `seq([TAB]` and `seq(from = 1, [TAB])` will both list only the arguments of `seq` (or any of its methods), whereas trying to complete `seq(length[TAB]` will list both the `length.out` argument and the `length(` function as possible completions. Note that no attempt is made to remove arguments already supplied, as that would incur a further speed penalty.

Special functions: For a few special functions (`library`, `data`, etc), the first argument is treated specially, in the sense that normal completion is suppressed, and some function specific completions are enabled if so requested by the settings. The `ipck` setting, which controls whether `library` and `require` will complete on *installed packages*, is disabled by default because the first call to `installed.packages` is potentially time consuming (e.g., when packages are installed on a remote network file server). Note, however, that the results of a call to `installed.packages` is cached, so subsequent calls are usually fast, so turning this option on is not particularly onerous even in such situations.

`findMatches` is an utility function that is used internally to determine matches. It can be used for writing methods for `.DollarNames` or `.AtNames`, the main benefit being that it will take the current `fuzzy` setting into account.

Value

If `rc.settings` is called without any arguments, it returns the current settings as a named logical vector. Otherwise, it returns `NULL` invisibly.

`rc.status` returns, as a list, the contents of an internal (unexported) environment that is used to record the results of the last completion attempt. This can be useful for debugging. For such use,

one must resist the temptation to use completion when typing the call to `rc.status` itself, as that then becomes the last attempt by the time the call is executed.

The items of primary interest in the returned list are:

`comps`	The possible completions generated by the last call to `.completeToken`, as a character vector.
`token`	The token that was (or, is to be) completed, as set by the last call to `.assignToken` (possibly inside a call to `.guessTokenFromLine`).
`linebuffer`	The full line, as set by the last call to `.assignLinebuffer`.
`start`	The start position of the token in the line buffer, as set by the last call to `.assignStart`.
`end`	The end position of the token in the line buffer, as set by the last call to `.assignEnd`.
`fileName`	Logical, indicating whether the cursor is currently inside quotes.
`fguess`	The name of the function the cursor is currently inside.
`isFirstArg`	Logical. If cursor is inside a function, is it the first argument?

In addition, the components `settings` and `options` give the current values of settings and options respectively.

`rc.getOption` and `rc.options` behave much like `getOption` and `options` respectively.

`findMatches` returns values that match the input pattern, taking the `fuzzy` flag into account.

Unexported API

There are several unexported functions in the package. Of these, a few are special because they provide the API through which other mechanisms can make use of the facilities provided by this package (they are unexported because they are not meant to be called directly by users). The usage of these functions are:

```
.assignToken(text)
.assignLinebuffer(line)
.assignStart(start)
.assignEnd(end)

.completeToken(custom = TRUE)
.retrieveCompletions()
.getFileComp()

.guessTokenFromLine()
.win32consoleCompletion(linebuffer, cursorPosition,
                        check.repeat = TRUE,
                        minlength = -1)

.addFunctionInfo(...)
```

The first four functions set up a completion attempt by specifying the token to be completed (`text`), and indicating where (`start` and `end`, which should be integers) the token is placed within the complete line typed so far (`line`).

Potential completions of the token are generated by `.completeToken`, and the completions can be retrieved as an R character vector using `.retrieveCompletions`. It is possible for the user to specify a replacement for this function by setting `rc.options("custom.completer")`; if not `NULL`, this function is called to compute potential completions. This facility is meant to help in situations where completing as R code is not appropriate. See source code for more details. Custom completion can be disabled by setting `custom = FALSE` when calling `.completeToken`.

If the cursor is inside quotes, completion may be suppressed. The function `.getFileComp` can be used after a call to `.completeToken` to determine if this is the case (returns `TRUE`), and alternative completions generated as deemed useful. In most cases, filename completion is a reasonable fallback.

The `.guessTokenFromLine` function is provided for use with backends that do not already break a line into tokens. It requires the linebuffer and endpoint (cursor position) to be already set, and itself sets the token and the start position. It returns the token as a character string.

The `.win32consoleCompletion` is similar in spirit, but is more geared towards the Windows GUI (or rather, any front-end that has no completion facilities of its own). It requires the linebuffer and cursor position as arguments, and returns a list with three components, `addition`, `possible` and `comps`. If there is an unambiguous extension at the current position, `addition` contains the additional text that should be inserted at the cursor. If there is more than one possibility, these are available either as a character vector of preformatted strings in `possible`, or as a single string in `comps`. `possible` consists of lines formatted using the current `width` option, so that printing them on the console one line at a time will be a reasonable way to list them. `comps` is a space separated (collapsed) list of the same completions, in case the front-end wishes to display it in some other fashion.

The `minlength` argument can be used to suppress completion when the token is too short (which can be useful if the front-end is set up to try completion on every keypress). If `check.repeat` is `TRUE`, it is detected if the same completion is being requested more than once in a row, and ambiguous completions are returned only in that case. This is an attempt to emulate GNU Readline behaviour, where a single TAB completes up to any unambiguous part, and multiple possibilities are reported only on two consecutive TABs.

As the various front-end interfaces evolve, the details of these functions are likely to change as well.

The function `.addFunctionInfo` can be used to add information about the permitted argument names for specific functions. Multiple named arguments are allowed in calls to it, where the tags are names of functions and values are character vectors representing valid arguments. When the `argdb` setting is `TRUE`, these are used as a source of valid argument names for the relevant functions.

Note

If you are uncomfortable with unsolicited evaluation of pieces of code, you should set `ops = FALSE`. Otherwise, trying to complete `foo@ba` will evaluate `foo`, trying to complete `foo[i, 1:10]$ba` will evaluate `foo[i, 1:10]`, etc. This should not be too bad, as explicit function calls (involving parentheses) are not evaluated in this manner. However, this *will* affect promises and lazy loaded symbols.

Author(s)

Deepayan Sarkar, <deepayan.sarkar@r-project.org>

`read.DIF` *Data Input from Spreadsheet*

Description

Reads a file in Data Interchange Format (DIF) and creates a data frame from it. DIF is a format for data matrices such as single spreadsheets.

Usage

```
read.DIF(file, header = FALSE,
         dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
         row.names, col.names, as.is = !stringsAsFactors,
         na.strings = "NA", colClasses = NA, nrows = -1,
         skip = 0, check.names = TRUE, blank.lines.skip = TRUE,
         stringsAsFactors = FALSE,
         transpose = FALSE, fileEncoding = "")
```

Arguments

`file`	the name of the file which the data are to be read from, or a connection, or a complete URL. The name `"clipboard"` may also be used on Windows, in which case `read.DIF("clipboard")` will look for a DIF format entry in the Windows clipboard.
`header`	a logical value indicating whether the spreadsheet contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains only character values and the top left cell is empty.
`dec`	the character used in the file for decimal points.
`numerals`	string indicating how to convert numbers whose conversion to double precision would lose accuracy, see `type.convert`.
`row.names`	a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names. If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered. Using `row.names = NULL` forces row numbering.
`col.names`	a vector of optional names for the variables. The default is to use `"V"` followed by the column number.

`as.is` controls conversion of character variables (insofar as they are not converted to logical, numeric or complex) to factors, if not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: In releases prior to R 2.12.1, cells marked as being of character type were converted to logical, numeric or complex using `type.convert` as in `read.table`.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`.

Possible values are `NA` (when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` the maximum number of rows to read in. Negative values are ignored.

`skip` the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`blank.lines.skip` logical: if `TRUE` blank lines in the input are ignored.

`stringsAsFactors` logical: should character vectors be converted to factors?

`transpose` logical, indicating if the row and column interpretation should be transposed. Microsoft's Excel has been known to produce (non-standard conforming) DIF files which would need `transpose = TRUE` to be read correctly.

`fileEncoding` character string: if non-empty declares the encoding used on a file (not a connection or clipboard) so the character data can be re-encoded. See the 'Encoding' section of the help for `file`, the 'R Data Import/Export' manual and 'Note'.

Value

A data frame (`data.frame`) containing a representation of the data in the file. Empty input is an error unless `col.names` is specified, when a 0-row data frame is returned: similarly giving just a header line if `header = TRUE` results in a 0-row data frame.

Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes.

Author(s)

R Core; `transpose` option by Christoph Buser, ETH Zurich

References

The DIF format specification can be found by searching on http://www.wotsit.org/; the optional header fields are ignored. See also https://en.wikipedia.org/wiki/Data_Interchange_Format.

The term is likely to lead to confusion: Windows will have a 'Windows Data Interchange Format (DIF) data format' as part of its WinFX system, which may or may not be compatible.

See Also

The *R Data Import/Export* manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `read.table`; `data.frame`.

Examples

```
## read.DIF() may need transpose = TRUE for a file exported from Excel
udir <- system.file("misc", package = "utils")
dd <- read.DIF(file.path(udir, "exDIF.dif"), header = TRUE, transpose = TRUE)
dc <- read.csv(file.path(udir, "exDIF.csv"), header = TRUE)
stopifnot(identical(dd, dc), dim(dd) == c(4,2))
```

`read.fortran` *Read Fixed-Format Data in a Fortran-like Style*

Description

Read fixed-format data files using Fortran-style format specifications.

Usage

```
read.fortran(file, format, ..., as.is = TRUE, colClasses = NA)
```

Arguments

`file`	File or connection to read from.
`format`	Character vector or list of vectors. See 'Details' below.
`...`	Other arguments for `read.fwf`.
`as.is`	Keep characters as characters?
`colClasses`	Variable classes to override defaults. See `read.table` for details.

Details

The format for a field is of one of the following forms: `rFl.d`, `rDl.d`, `rXl`, `rAl`, `rIl`, where `l` is the number of columns, `d` is the number of decimal places, and `r` is the number of repeats. `F` and `D` are numeric formats, `A` is character, `I` is integer, and `X` indicates columns to be skipped. The repeat code `r` and decimal place code `d` are always optional. The length code `l` is required except for `X` formats when `r` is present.

For a single-line record, `format` should be a character vector. For a multiline record it should be a list with a character vector for each line.

Skipped (`X`) columns are not passed to `read.fwf`, so `colClasses`, `col.names`, and similar arguments passed to `read.fwf` should not reference these columns.

Value

A data frame

Note

`read.fortran` does not use actual Fortran input routines, so the formats are at best rough approximations to the Fortran ones. In particular, specifying `d > 0` in the `F` or `D` format will shift the decimal `d` places to the left, even if it is explicitly specified in the input file.

See Also

`read.fwf`, `read.table`, `read.csv`

Examples

```
ff <- tempfile()
cat(file = ff, "123456", "987654", sep = "\n")
read.fortran(ff, c("F2.1","F2.0","I2"))
read.fortran(ff, c("2F1.0","2X","2A1"))
unlink(ff)
cat(file = ff, "123456AB", "987654CD", sep = "\n")
read.fortran(ff, list(c("2F3.1","A2"), c("3I2","2X")))
unlink(ff)
# Note that the first number is read differently than Fortran would
# read it:
cat(file = ff, "12.3456", "1234567", sep = "\n")
read.fortran(ff, "F7.4")
unlink(ff)
```

---

read.fwf *Read Fixed Width Format Files*

Description

Read a table of **f**ixed **w**idth **f**ormatted data into a `data.frame`.

Usage

```
read.fwf(file, widths, header = FALSE, sep = "\t",
         skip = 0, row.names, col.names, n = -1,
         buffersize = 2000, fileEncoding = "", ...)
```

Arguments

file	the name of the file which the data are to be read from. Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call.
widths	integer vector, giving the widths of the fixed-width fields (of one line), or list of integer vectors giving widths for multiline records.
header	a logical value indicating whether the file contains the names of the variables as its first line. If present, the names must be delimited by sep.
sep	character; the separator used internally; should be a character that does not occur in the file (except in the header).
skip	number of initial lines to skip; see read.table.
row.names	see read.table.
col.names	see read.table.
n	the maximum number of records (lines) to be read, defaulting to no limit.
buffersize	Maximum number of lines to read at one time
fileEncoding	character string: if non-empty declares the encoding used on a file (not a connection) so the character data can be re-encoded. See the 'Encoding' section of the help for file, the 'R Data Import/Export' manual and 'Note'.
...	further arguments to be passed to read.table. Useful such arguments include as.is, na.strings, colClasses and strip.white.

Details

Multiline records are concatenated to a single line before processing. Fields that are of zero-width or are wholly beyond the end of the line in file are replaced by NA.

Negative-width fields are used to indicate columns to be skipped, e.g., -5 to skip 5 columns. These fields are not seen by read.table and so should not be included in a col.names or colClasses argument (nor in the header line, if present).

Reducing the buffersize argument may reduce memory use when reading large files with long lines. Increasing buffersize may result in faster processing when enough memory is available.

Note that read.fwf (not read.table) reads the supplied file, so the latter's argument encoding will not be useful.

Value

A data.frame as produced by read.table which is called internally.

Author(s)

Brian Ripley for R version: originally in Perl by Kurt Hornik.

See Also

scan and read.table.

read.fortran for another style of fixed-format files.

Examples

```
ff <- tempfile()
cat(file = ff, "123456", "987654", sep = "\n")
read.fwf(ff, widths = c(1,2,3))    #> 1 23 456 \ 9 87 654
read.fwf(ff, widths = c(1,-2,3))   #> 1 456 \ 9 654
unlink(ff)
cat(file = ff, "123", "987654", sep = "\n")
read.fwf(ff, widths = c(1,0, 2,3))    #> 1 NA 23 NA \ 9 NA 87 654
unlink(ff)
cat(file = ff, "123456", "987654", sep = "\n")
read.fwf(ff, widths = list(c(1,0, 2,3), c(2,2,2))) #> 1 NA 23 456 98 76 54
unlink(ff)
```

---

read.socket *Read from or Write to a Socket*

Description

read.socket reads a string from the specified socket, write.socket writes to the specified socket. There is very little error checking done by either.

Usage

```
read.socket(socket, maxlen = 256L, loop = FALSE)
write.socket(socket, string)
```

Arguments

socket	a socket object.
maxlen	maximum length (in bytes) of string to read.
loop	wait for ever if there is nothing to read?
string	string to write to socket.

Value

read.socket returns the string read as a length-one character vector.

write.socket returns the number of bytes written.

Author(s)

Thomas Lumley

See Also

close.socket, make.socket

Examples

```
finger <- function(user, host = "localhost", port = 79, print = TRUE)
{
    if (!is.character(user))
        stop("user name must be a string")
    user <- paste(user,"\r\n")
    socket <- make.socket(host, port)
    on.exit(close.socket(socket))
    write.socket(socket, user)
    output <- character(0)
    repeat{
        ss <- read.socket(socket)
        if (ss == "") break
        output <- paste(output, ss)
    }
    close.socket(socket)
    if (print) cat(output)
    invisible(output)
}
## Not run:
finger("root")  ## only works if your site provides a finger daemon
## End(Not run)
```

---

read.table *Data Input*

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors, tryLogical = TRUE,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
```

```
            stringsAsFactors = FALSE,
            fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)

read.csv2(file, header = TRUE, sep = ";", quote = "\"",
          dec = ",", fill = TRUE, comment.char = "", ...)

read.delim(file, header = TRUE, sep = "\t", quote = "\"",
           dec = ".", fill = TRUE, comment.char = "", ...)

read.delim2(file, header = TRUE, sep = "\t", quote = "\"",
            dec = ",", fill = TRUE, comment.char = "", ...)
```

Arguments

`file`	the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an *absolute* path, the file name is *relative* to the current working directory, `getwd()`. Tilde-expansion is performed where supported. This can be a compressed file (see `file`). Alternatively, `file` can be a readable text-mode connection (which will be opened for reading if necessary, and if so `close`d (and hence destroyed) at the end of the function call). (If `stdin()` is used, the prompts for lines may be somewhat confusing. Terminate input with a blank line or an EOF signal, `Ctrl-D` on Unix and `Ctrl-Z` on Windows. Any pushback on `stdin()` will be cleared before return.) `file` can also be a complete URL. (For the supported URL schemes, see the 'URLs' section of the help for `url`.)
`header`	a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.
`sep`	the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default for `read.table`) the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.
`quote`	the set of quoting characters. To disable quoting altogether, use `quote = ""`. See `scan` for the behaviour on quotes embedded in quotes. Quoting is only considered for columns read as character, which is all of them unless `colClasses` is specified.
`dec`	the character used in the file for decimal points.
`numerals`	string indicating how to convert numbers whose conversion to double precision would lose accuracy, see `type.convert`. Can be abbreviated. (Applies also to complex-number inputs.)
`row.names`	a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered.

Using `row.names = NULL` forces row numbering. Missing or `NULL` `row.names` generate row names that are considered to be 'automatic' (and not preserved by `as.matrix`).

`col.names` a vector of optional names for the variables. The default is to use `"V"` followed by the column number.

`as.is` controls conversion of character variables (insofar as they are not converted to logical, numeric or complex) to factors, if not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`tryLogical` a `logical` determining if columns consisting entirely of `"F"`, `"T"`, `"FALSE"`, and `"TRUE"` should be converted to `logical`; passed to `type.convert`, true by default.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields. Note that the test happens *after* white space is stripped from the input (if enabled), so `na.strings` values may need their own white space stripped in advance.

`colClasses` character. A vector of classes to be assumed for the columns. If unnamed, recycled as necessary. If named, names are matched with unspecified values being taken to be `NA`.

Possible values are `NA` (the default, when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` integer: the maximum number of rows to read in. Negative and other invalid values are ignored.

`skip` integer: the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`fill` logical. If `TRUE` then in case the rows have unequal length, blank fields are implicitly added. See 'Details'.

`strip.white` logical. Used only when `sep` has been specified, and allows the stripping of leading and trailing white space from unquoted `character` fields (`numeric` fields are always stripped). See `scan` for further details (including the exact meaning of 'white space'), remembering that the columns may include the row names.

`blank.lines.skip` logical: if `TRUE` blank lines in the input are ignored.

`comment.char` character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether.

`allowEscapes` logical. Should C-style escapes such as '`\n`' be processed or read verbatim (the default)? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character). For more details see `scan`.

`flush` logical: if `TRUE`, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field.

`stringsAsFactors` logical: should character vectors be converted to factors? Note that this is overridden by `as.is` and `colClasses`, both of which allow finer control.

`fileEncoding` character string: if non-empty declares the encoding used on a file when given as a character string (not on an existing connection) so the character data can be re-encoded. See the 'Encoding' section of the help for `file`, the 'R Data Import/Export' manual and 'Note'.

`encoding` encoding to be assumed for input strings. It is used to mark character strings as known to be in Latin-1 or UTF-8 (see `Encoding`): it is not used to re-encode the input, but allows R to handle encoded strings in their native encoding (if one of those two). See 'Value' and 'Note'.

`text` character string: if `file` is not supplied and this is, then data are read from the value of `text` via a text connection. Notice that a literal string can be used to include (small) data sets within R code.

`skipNul` logical: should NULs be skipped?

`...` Further arguments to be passed to `read.table`.

Details

This function is the principal means of reading tabular data into R.

Unless `colClasses` is specified, all columns are read as character columns and then converted using `type.convert` to logical, integer, numeric, complex or (depending on `as.is`) factor as appropriate. Quotes are (by default) interpreted in all fields, so a column of values like `"42"` will result in an integer column.

A field or line is 'blank' if it contains nothing (except whitespace if no separator is specified) before a comment character or the end of the field or line.

If `row.names` is not specified and the header line has one less entry than the number of columns, the first column is taken to be the row names. This allows data frames to be read in from the format in which they are printed. If `row.names` is specified and does not refer to the first column, that column is discarded from such files.

The number of data columns is determined by looking at the first five lines of input (or the whole input if it has less than five lines), or from the length of `col.names` if it is specified and is longer.

This could conceivably be wrong if `fill` or `blank.lines.skip` are true, so specify `col.names` if necessary (as in the 'Examples').

`read.csv` and `read.csv2` are identical to `read.table` except for the defaults. They are intended for reading 'comma separated value' files ('`.csv`') or (`read.csv2`) the variant used in countries that use a comma as decimal point and a semicolon as field separator. Similarly, `read.delim` and `read.delim2` are for reading delimited files, defaulting to the TAB character for the delimiter. Notice that `header = TRUE` and `fill = TRUE` in these variants, and that the comment character is disabled.

The rest of the line after a comment character is skipped; quotes are not processed in comments. Complete comment lines are allowed provided `blank.lines.skip = TRUE`; however, comment lines prior to the header must have the comment character in the first non-blank column.

Quoted fields with embedded newlines are supported except after a comment character. Embedded NULs are unsupported: skipping them (with `skipNul = TRUE`) may work.

Value

A data frame (`data.frame`) containing a representation of the data in the file.

Empty input is an error unless `col.names` is specified, when a 0-row data frame is returned: similarly giving just a header line if `header = TRUE` results in a 0-row data frame. Note that in either case the columns will be logical unless `colClasses` was supplied.

Character strings in the result (including factor levels) will have a declared encoding if `encoding` is `"latin1"` or `"UTF-8"`.

CSV files

See the help on `write.csv` for the various conventions for `.csv` files. The commonest form of CSV file with row names needs to be read with `read.csv(..., row.names = 1)` to use the names in the first column of the file as row names.

Memory usage

These functions can use a surprising amount of memory when reading large files. There is extensive discussion in the 'R Data Import/Export' manual, supplementing the notes here.

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes. This can be particularly so when reading a column that takes many distinct numeric values, as storing each distinct value as a character string can take up to 14 times as much memory as storing it as an integer.

Using `nrows`, even as a mild over-estimate, will help memory usage.

Using `comment.char = ""` will be appreciably faster than the `read.table` default.

`read.table` is not the right tool for reading large matrices, especially those with many columns: it is designed to read *data frames* which may have columns of very different classes. Use `scan` instead for matrices.

Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

There are two approaches for reading input that is not in the local encoding. If the input is known to be UTF-8 or Latin1, use the `encoding` argument to declare that. If the input is in some other encoding, then it may be translated on input. The `fileEncoding` argument achieves this by setting up a connection to do the re-encoding into the current locale. Note that on Windows or other systems not running in a UTF-8 locale, this may not be possible.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The 'R Data Import/Export' manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `write.table`; `data.frame`.

`count.fields` can be useful to determine problems with reading files which result in reports of incorrect record lengths (see the 'Examples' below).

https://www.rfc-editor.org/rfc/rfc4180 for the IANA definition of CSV files (which requires comma as separator and CRLF line endings).

Examples

```
## using count.fields to handle unknown maximum number of fields
## when fill = TRUE
test1 <- c(1:5, "6,7", "8,9,10")
tf <- tempfile()
writeLines(test1, tf)

read.csv(tf, fill = TRUE) # 1 column
ncol <- max(count.fields(tf, sep = ","))
read.csv(tf, fill = TRUE, header = FALSE,
         col.names = paste0("V", seq_len(ncol)))
unlink(tf)

## "Inline" data set, using text=
## Notice that leading and trailing empty lines are auto-trimmed

read.table(header = TRUE, text = "
a b
1 2
3 4
")
```

readRegistry *Read a Windows Registry Hive*

Description

On Windows, read values of keys in the Windows Registry, and optionally whole hives.

Usage

```
readRegistry(key, hive = c("HLM", "HCR", "HCU", "HU", "HCC", "HPD"),
             maxdepth = 1, view = c("default", "32-bit", "64-bit"))
```

Arguments

key	character string, the path to the key in the Windows Registry.
hive	The 'hive' containing the key. The abbreviations are for HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT. HKEY_CURRENT_USER, HKEY_USERS, HKEY_CURRENT_CONFIG and HKEY_PERFORMANCE_DATA
maxdepth	How far to recurse into the subkeys of the key. By default only the values of the key and the names of subkeys are returned.
view	On 64-bit Windows, the view of the Registry to be used: see 'Details'.

Details

Registry access is done using the security settings of the current R session: this means that some Registry keys may not be accessible even if they exist. This may result in NULL values in the object returned, and, possibly, empty element names.

On 64-bit Windows, this will by default read the 32-bit view of the Registry when run from 32-bit R, and the 64-bit view when run from 64-bit R: see https://learn.microsoft.com/en-us/windows/win32/winprog64/registry-redirector.

Value

A named list of values and subkeys (which may themselves be named lists). The default value (if any) precedes named values which precede subkeys, and both the latter sets are sorted alphabetically.

Note

This is only available on Windows.

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
  ## only in HLM if set in an admin-mode install.
  try(readRegistry("SOFTWARE\\R-core", maxdepth = 3))

  gmt <- file.path("SOFTWARE", "Microsoft", "Windows NT",
                   "CurrentVersion", "Time Zones",
     "GMT Standard Time", fsep = "\\")
  readRegistry(gmt, "HLM")
})
## Not run: ## on a 64-bit R need this to find 32-bit JAGS
readRegistry("SOFTWARE\\JAGS", maxdepth = 3, view = "32")

## See if there is a 64-bit user install
readRegistry("SOFTWARE\\R-core\\R64", "HCU", maxdepth = 2)

## End(Not run)
```

---

recover *Browsing after an Error*

Description

This function allows the user to browse directly on any of the currently active function calls, and is suitable as an error option. The expression `options(error = recover)` will make this the error option.

Usage

```
recover()
```

Details

When called, `recover` prints the list of current calls, and prompts the user to select one of them. The standard R `browser` is then invoked from the corresponding environment; the user can type ordinary R language expressions to be evaluated in that environment.

When finished browsing in this call, type `c` to return to `recover` from the browser. Type another frame number to browse some more, or type `0` to exit `recover`.

The use of `recover` largely supersedes `dump.frames` as an error option, unless you really want to wait to look at the error. If `recover` is called in non-interactive mode, it behaves like `dump.frames`. For computations involving large amounts of data, `recover` has the advantage that it does not need to copy out all the environments in order to browse in them. If you do decide to quit interactive debugging, call `dump.frames` directly while browsing in any frame (see the examples).

Value

Nothing useful is returned. However, you *can* invoke `recover` directly from a function, rather than through the error option shown in the examples. In this case, execution continues after you type `0` to exit `recover`.

Compatibility Note

The R `recover` function can be used in the same way as the S function of the same name; therefore, the error option shown is a compatible way to specify the error action. However, the actual functions are essentially unrelated and interact quite differently with the user. The navigating commands up and down do not exist in the R version; instead, exit the browser and select another frame.

References

John M. Chambers (1998). *Programming with Data*; Springer.
See the compatibility note above, however.

See Also

`browser` for details about the interactive computations; `options` for setting the error option; `dump.frames` to save the current environments for later debugging.

Examples

```
## Not run:

options(error = recover) # setting the error option

### Example of interaction

> myFit <- lm(y ~ x, data = xy, weights = w)
Error in lm.wfit(x, y, w, offset = offset, ...) :
        missing or negative weights not allowed

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> objects() # all the objects in this frame
[1] "method" "n"      "ny"     "offset" "tol"    "w"
[7] "x"      "y"
Browse[1]> w
[1] -0.5013844  1.3112515  0.2939348 -0.8983705 -0.1538642
[6] -0.9772989  0.7888790 -0.1919154 -0.3026882
Browse[1]> dump.frames() # save for offline debugging
Browse[1]> c # exit the browser

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 0 # exit recover
>


## End(Not run)
```

relist *Allow Re-Listing an unlist()ed Object*

Description

relist() is an S3 generic function with a few methods in order to allow easy inversion of unlist(obj) when that is used with an object obj of (S3) class "relistable".

Usage

```
relist(flesh, skeleton)
## Default S3 method:
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'factor'
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'list'
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'matrix'
relist(flesh, skeleton = attr(flesh, "skeleton"))

as.relistable(x)
is.relistable(x)

## S3 method for class 'relistable'
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

flesh	a vector to be relisted
skeleton	a list, the structure of which determines the structure of the result
x	an R object, typically a list (or vector).
recursive	logical. Should unlisting be applied to list components of x?
use.names	logical. Should names be preserved?

Details

Some functions need many parameters, which are most easily represented in complex structures, e.g., nested lists. Unfortunately, many mathematical functions in R, including optim and nlm can only operate on functions whose domain is a vector. R has unlist() to convert nested list objects into a vector representation. relist(), its methods and the functionality mentioned here provide the inverse operation to convert vectors back to the convenient structural representation. This allows structured functions (such as optim()) to have simple mathematical interfaces.

For example, a likelihood function for a multivariate normal model needs a variance-covariance matrix and a mean vector. It would be most convenient to represent it as a list containing a vector and a matrix. A typical parameter might look like

```
list(mean = c(0, 1), vcov = cbind(c(1, 1), c(1, 0))).
```

However, optim cannot operate on functions that take lists as input; it only likes numeric vectors. The solution is conversion. Given a function mvdnorm(x, mean, vcov, log = FALSE) which computes the required probability density, then

```
ipar <- list(mean = c(0, 1), vcov = c bind(c(1, 1), c(1, 0)))
initial.param <- as.relistable(ipar)

ll <- function(param.vector)
{
   param <- relist(param.vector, skeleton = ipar)
   -sum(mvdnorm(x, mean = param$mean, vcov = param$vcov,
                log = TRUE))
}

optim(unlist(initial.param), ll)
```

relist takes two parameters: skeleton and flesh. Skeleton is a sample object that has the right shape but the wrong content. flesh is a vector with the right content but the wrong shape. Invoking

```
relist(flesh, skeleton)
```

will put the content of flesh on the skeleton. You don't need to specify skeleton explicitly if the skeleton is stored as an attribute inside flesh. In particular, if flesh was created from some object obj with unlist(as.relistable(obj)) then the skeleton attribute is automatically set. (Note that this does not apply to the example here, as optim is creating a new vector to pass to ll and not its par argument.)

As long as skeleton has the right shape, it should be an inverse of unlist. These equalities hold:

```
relist(unlist(x), x) == x
unlist(relist(y, skeleton)) == y

x <- as.relistable(x)
relist(unlist(x)) == x
```

Note however that the relisted object might not be *identical* to the skeleton because of implicit coercions performed during the unlisting step. All elements of the relisted objects have the same type as the unlisted object. NULL values are coerced to empty vectors of that type.

Value

an object of (S3) class "relistable" (and "list").

Author(s)

R Core, based on a code proposal by Andrew Clausen.

See Also

unlist

Examples

```
ipar <- list(mean = c(0, 1), vcov = cbind(c(1, 1), c(1, 0)))
initial.param <- as.relistable(ipar)
ul <- unlist(initial.param)
relist(ul)
stopifnot(identical(relist(ul), initial.param))
```

REMOVE *Remove Add-on Packages*

Description

Utility for removing add-on packages.

Usage

```
R CMD REMOVE [options] [-l lib] pkgs
```

Arguments

pkgs	a space-separated list with the names of the packages to be removed.
lib	the path name of the R library tree to remove from. May be absolute or relative. Also accepted in the form '--library=lib'.
options	further options for help or version.

Details

If used as R CMD REMOVE pkgs without explicitly specifying lib, packages are removed from the library tree rooted at the first directory in the library path which would be used by R run in the current environment.

To remove from the library tree *lib* instead of the default one, use R CMD REMOVE -l lib *pkgs*.

Use R CMD REMOVE --help for more usage information.

Note

Some binary distributions of R have REMOVE in a separate bundle, e.g. an R-devel RPM.

See Also

INSTALL, remove.packages

remove.packages *Remove Installed Packages*

Description

Removes installed packages/bundles and updates index information as necessary.

Usage

```
remove.packages(pkgs, lib)
```

Arguments

pkgs	a character vector with the names of the packages to be removed.
lib	a character vector giving the library directories to remove the packages from. If missing, defaults to the first element in `.libPaths()`.

See Also

On Unix-alikes, `REMOVE` for a command line version;

`install.packages` for installing packages.

removeSource *Remove Stored Source from a Function or Language Object*

Description

When `options("keep.source")` is `TRUE`, a copy of the original source code to a function is stored with it. Similarly, `parse()` may keep formatted source for an expression. Such source reference attributes are removed from the object by `removeSource()`.

Usage

```
removeSource(fn)
```

Arguments

fn	a `function` or another language object (fulfilling `is.language`) from which to remove the source.

Details

This removes the `"srcref"` and related attributes, via *recursive* cleaning of `body(fn)` in the case of a function or the recursive language parts, otherwise.

Value

A copy of the `fn` object with the source removed.

See Also

`is.language` about language objects.

`srcref` for a description of source reference records, `deparse` for a description of how functions are deparsed.

Examples

```
## to make this act independently of the global 'options()' setting:
op <- options(keep.source = TRUE)
fn <- function(x) {
  x + 1 # A comment, kept as part of the source
}
fn
names(attributes(fn))       # "srcref" (only)
names(attributes(body(fn))) # "srcref" "srcfile" "wholeSrcref"
f2 <- removeSource(fn)
f2
stopifnot(length(attributes(fn)) > 0,
          is.null(attributes(f2)),
          is.null(attributes(body(f2))))

## Source attribute of parse()d expressions,
##   have {"srcref", "srcfile", "wholeSrcref"} :
E  <- parse(text ="a <- x^y  # power")  ; names(attributes(E ))
E. <- removeSource(E)                  ; names(attributes(E.))
stopifnot(length(attributes(E ))  > 0,
          is.null(attributes(E.)))
options(op) # reset to previous state
```

RHOME *R Home Directory*

Description

Returns the location of the R home directory, which is the root of the installed R tree.

Usage

```
R RHOME
```

roman *Roman Numerals*

Description

Simple manipulation of (a small set of) integer numbers as roman numerals.

Usage

```
as.roman(x)
.romans

r1 + r2
r1 <= r2
max(r1)
sum(r2)
```

Arguments

x	a numeric or character vector of arabic or roman numerals.
r1, r2	a roman number vector, i.e., of class "roman".

Details

as.roman creates objects of class "roman" which are internally represented as integers, and have suitable methods for printing, formatting, subsetting, coercion, etc, see methods(class = "roman").

Arithmetic ("Arith"), Comparison ("Compare") and ("Logic"), i.e., all "Ops" group operations work as for regular numbers via R's integer functionality.

Only numbers between 1 and 3999 have a unique representation as roman numbers, and hence others result in as.roman(NA).

.romans is the basic dictionary, a named character vector.

References

Wikipedia contributors (2024). Roman numerals. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Roman_numerals&oldid=1188781837. Accessed February 22, 2024.

Examples

```
## First five roman 'numbers'.
(y <- as.roman(1 : 5))
## Middle one.
y[3]
## Current year as a roman number.
```

```
(y <- as.roman(format(Sys.Date(), "%Y")))
## Today, and  10, 20, 30, and 100 years ago ...
y - 10*c(0:3,10)

## mixture of arabic and roman numbers :
as.roman(c(NA, 1:3, "", strrep("I", 1:6))) # + NA with a warning for "IIIIII"
cc <- c(NA, 1:3, strrep("I", 0:5))
(rc <- as.roman(cc)) # two NAs: 0 is not "roman"
(ic <- as.integer(rc)) # works automatically [without an explicit method]
rNA <- as.roman(NA)
## simple consistency checks -- arithmetic when result is in  {1,2,..,3999} :
stopifnot(identical(rc, as.roman(rc)), # as.roman(.) is "idempotent"
          identical(rc + rc + (3*rc), rc*5),
          identical(ic, c(NA, 1:3, NA, 1:5)),
          identical(as.integer(5*rc), 5L*ic),
          identical(as.numeric(rc), as.numeric(ic)),
          identical(rc[1], rNA),
          identical(as.roman(0), rNA),
          identical(as.roman(NA_character_), rNA),
          identical(as.list(rc), as.list(ic)))
## Non-Arithmetic 'Ops' :
stopifnot(exprs = {
        # Comparisons :
        identical(ic < 1:5, rc < 1:5)
        identical(ic < 1:5, rc < as.roman(1:5))
        # Logic  [integers |-> logical] :
        identical(rc & TRUE , ic & TRUE)
        identical(rc & FALSE, ic & FALSE)
        identical(rc | FALSE, ic | FALSE)
        identical(rc | NA   , ic | NA)
})
## 'Summary' group functions (and comparison):
(rc. <- rc[!is.na(rc)])
stopifnot(exprs = {
        identical(min(rc), as.roman(NA))
        identical(min(rc, na.rm=TRUE),
         as.roman(min(ic, na.rm=TRUE)))
        identical(range(rc.),
         as.roman(range(as.integer(rc.))))
        identical(sum (rc, na.rm=TRUE), as.roman("XXI"))
        identical(format(prod(rc, na.rm=TRUE)), "DCCXX")
        format(prod(rc.)) == "DCCXX"
})
```

Rprof *Enable Profiling of R's Execution*

Description

Enable or disable profiling of the execution of R expressions.

Usage

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
      memory.profiling = FALSE, gc.profiling = FALSE,
      line.profiling = FALSE, filter.callframes = FALSE,
      numfiles = 100L, bufsize = 10000L,
      event = c("default", "cpu", "elapsed"))
```

Arguments

`filename`	The file to be used for recording the profiling results. Set to `NULL` or `""` to disable profiling.
`append`	logical: should the file be over-written or appended to?
`interval`	real: distance (time interval) between samples in seconds.
`memory.profiling`	logical: write memory use information to the file?
`gc.profiling`	logical: record whether GC is running?
`line.profiling`	logical: write line locations to the file?
`filter.callframes`	logical: filter out intervening call frames of the call tree. See the filtering out call frames section.
`numfiles`, `bufsize`	integers: line profiling memory allocation
`event`	character: profiling event, character vector of length one, `"elapsed"` for elapsed (real, wall-clock) time and `"cpu"` for CPU time, both measured in seconds. `"default"` is the default event on the platform, one of the two. See the 'Details'.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling works by writing out the call stack every `interval` seconds (units of the profiling event), to the file specified. Either the `summaryRprof` function or the wrapper script `R CMD Rprof` can be used to process the output file to produce a summary of the usage; use `R CMD Rprof --help` for usage information.

Exactly what is measured is subtle and depends on the profiling event.

With `"elapsed"` (the default and only supported event on Windows): it is time that the R process is running and executing an R command. It is not however just CPU time, for if `readline()` is waiting for input, that counts as well. It is also known as 'elapsed' time.

With `"cpu"` (the default on Unix and typically the preferred event for identifying performance bottlenecks), it is CPU time of the R process, so for example excludes time when R is waiting for input or for processes run by `system` to return. It may go slower than `"elapsed"` when the process is often waiting for I/O to finish, but it may go faster with actively computing concurrent threads (say via OpenMP) on a multi-core system.

Note that the (timing) interval cannot be too small. With `"cpu"`, the time spent in each profiling step is currently added to the interval. With all profiling events, the computation in each profiling

step causes perturbation to the observed system and biases the results. What is feasible is machine-dependent. On Linux, R requires the interval to be at least 10ms, on all other platforms at least 1ms. Shorter intervals will be rounded up with a warning.

The "default" profiling event is "elapsed" on Windows and "cpu" on Unix.

Support for "elapsed" event on Unix is new and considered experimental. To reduce the risk of missing a sample, R tries to use the (real-time) FIFO scheduling policy with the maximum scheduling priority for an internal thread which initiates collection of each sample. If setting that priority fails, it tries to use the maximum scheduling priority of the current scheduling policy, falling back to the current scheduling parameters. On Linux, regular users are typically not allowed to use the real-time scheduling priorities. This can be usually allowed via PAM (e.g. '/etc/security/limits.conf'), see the OS documentation for details. The priorities only matter when profiling a system under high load.

Functions will only be recorded in the profile log if they put a context on the call stack (see sys.calls). Some primitive functions do not do so: specifically those which are of type "special" (see the 'R Internals' manual for more details).

Individual statements will be recorded in the profile log if line.profiling is TRUE, and if the code being executed was parsed with source references. See parse for a discussion of source references. By default the statement locations are not shown in summaryRprof, but see that help page for options to enable the display.

Filtering Out Call Frames

Lazy evaluation makes the call stack more complex because intervening call frames are created between the time arguments are applied to a function, and the time they are effectively evaluated. When the call stack is represented as a tree, these intervening frames appear as sibling nodes. For instance, evaluating try(EXPR) produces the following call tree, at the time EXPR gets evaluated:

```
1. +-base::try(EXPR)
2. | \-base::tryCatch(...)
3. |   \-base:::tryCatchList(expr, classes, parentenv, handlers)
4. |     \-base:::tryCatchOne(expr, names, parentenv, handlers[[1L]])
5. |       \-base:::doTryCatch(return(expr), name, parentenv, handler)
6. \-EXPR
```

Lines 2 to 5 are intervening call frames, the last of which finally triggered evaluation of EXPR. Setting filter.callframes to TRUE simplifies the profiler output by removing all sibling nodes of intervening frames.

The same kind of call frame filtering is applied with eval() frames. When you call eval(), two frames are pushed on the stack to ensure a continuity between frames. Say we have these definitions:

```
calling <- function() evaluator(quote(called()), environment())
evaluator <- function(expr, env) eval(expr, env)
called <- function() EXPR()
```

calling() calls called() in its own environment, via eval(). The latter is called indirectly through evaluator(). The net effect of this code is identical to just calling called() directly, without the intermediaries. However, the full call stack looks like this:

```
1. calling()
2. \-evaluator(quote(called()), environment())
3.   \-base::eval(expr, env)
4.     \-base::eval(expr, env)
5.       \-called()
6.         \-EXPR()
```

When call frame filtering is turned on, the true calling environment of `called()` is looked up, and the filtered call stack looks like this:

```
1. calling()
5. \-called()
6.   \-EXPR()
```

If the calling environment is not on the stack, the function called by `eval()` becomes a root node. Say we have:

```
calling <- function() evaluator(quote(called()), new.env())
```

With call frame filtering we then get the following filtered call stack:

```
5. called()
6. \-EXPR()
```

Note

On Unix-alikes: Profiling is not available on all platforms. By default, support for profiling is compiled in if possible – configure R with '`--disable-R-profiling`' to change this.

As R CPU profiling uses the same mechanisms as C profiling, the two cannot be used together, so do not use `Rprof(event = "cpu")` (the default) in an executable built for C-level profiling (such as using the GCC option '`-p`' or '`-pg`').

On Windows: `filename` can be a UTF-8-encoded filepath that cannot be translated to the current locale.

The profiler interrupts R asynchronously, and it cannot allocate memory to store results as it runs. This affects line profiling, which needs to store an unknown number of file pathnames. The `numfiles` and `bufsize` arguments control the size of pre-allocated buffers to hold these results: the former counts the maximum number of paths, the latter counts the numbers of bytes in them. If the profiler runs out of space it will skip recording the line information for new files, and issue a warning when `Rprof(NULL)` is called to finish profiling.

See Also

The chapter on "Tidying and profiling R code" in 'Writing R Extensions': `RShowDoc("R-exts")`.

`summaryRprof` to analyse the output file.

`tracemem`, `Rprofmem` for other ways to track memory use.

Examples

```
## Not run: Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append = TRUE)
## some code to be profiled
Rprof(NULL)
## ...
## Now post-process the output as described in Details

## End(Not run)
```

Rprofmem *Enable Profiling of R's Memory Use*

Description

Enable or disable reporting of memory allocation in R.

Usage

```
Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)
```

Arguments

filename	The file to be used for recording the memory allocations. Set to NULL or "" to disable reporting.
append	logical: should the file be over-written or appended to?
threshold	numeric: allocations on R's "large vector" heap larger than this number of bytes will be reported.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling writes the call stack to the specified file every time malloc is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which malloc is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively.

The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Value

None

Note

The memory profiler slows down R even when not in use, and so is a compile-time option. (It is enabled in a standard Windows build of R.)

The memory profiler can be used at the same time as other R and C profilers.

See Also

The R sampling profiler, `Rprof` also collects memory information.

`tracemem` traces duplications of specific objects.

The chapter on 'Tidying and profiling R code' in the 'Writing R Extensions' manual.

Examples

```
## Not run:
## not supported unless R is compiled to support it.
Rprofmem("Rprofmem.out", threshold = 1000)
example(glm)
Rprofmem(NULL)
noquote(readLines("Rprofmem.out", n = 5))

## End(Not run)
```

Rscript *Scripting Front-End for R*

Description

This is an alternative front end for use in '`#!`' scripts and other scripting applications.

Usage

```
Rscript [options] file [args]
Rscript [options] -e expr [-e expr2 ...] [args]
```

Arguments

`options`	a list of options, all beginning with '`--`'. These can be any of the options of the standard R front-end, and also those described in the details.
`expr`, `expr2`	R expression(s), properly quoted.
`file`	the name of a file containing R commands. '`-`' indicates '`stdin`'.
`args`	arguments to be passed to the script in `file` or expressions supplied via '`-e`'.

Details

`Rscript --help` gives details of usage, and `Rscript --version` gives the version of `Rscript`.

Other invocations invoke the R front-end with selected options. This front-end is convenient for writing '`#!`' scripts since it is an executable and takes `file` directly as an argument. Options '`--no-echo --no-restore`' are always supplied: these imply '`--no-save`'. Arguments that contain spaces cannot be specified directly on the '`#!`' line, because spaces and tabs are interpreted as delimiters and there is no way to protect them from this interpretation on the '`#!`' line. (The standard Windows command line has no concept of '`#!`' scripts, but Cygwin shells do.)

Either one or more '`-e`' options or `file` should be supplied. When using '`-e`' options be aware of the quoting rules in the shell used: see the examples.

The prescribed order of arguments is important: e.g. '`--verbose`' specified after '`-e`' will be part of `args` and passed to the expression; the same will happen to '`-e`' specified after `file`.

Additional options accepted as part of `options` (before `file` or '`-e`') are

'`-verbose`' gives details of what `Rscript` is doing.

'`-default-packages=list`' where `list` is a comma-separated list of package names or `NULL`. Sets the environment variable `R_DEFAULT_PACKAGES` which determines the packages loaded on startup.

Spaces are allowed in `expr` and `file` (but will need to be protected from the shell in use, if any, for example by enclosing the argument in quotes).

If '`--default-packages`' is not used, then `Rscript` checks the environment variable `R_SCRIPT_DEFAULT_PACKAGES`. If this is set, then it takes precedence over `R_DEFAULT_PACKAGES`.

Normally the version of R is determined at installation, but this can be overridden by setting the environment variable `RHOME`.

`stdin()` refers to the input file, and `file("stdin")` to the `stdin` file stream of the process.

Note

`Rscript` is only supported on systems with the `execv` system call.

Examples

```
## Not run:
Rscript -e 'date()' -e 'format(Sys.time(), "%a %b %d %X %Y")'

# Get the same initial packages in the same order as default R:
Rscript --default-packages=methods,datasets,utils,grDevices,graphics,stats -e 'sessionInfo()'

## example #! script for a Unix-alike
## (arguments given on the #! line end up as [options] to Rscript, while
## arguments passed to the #! script end up as [args], so available to
## commandArgs())
#! /path/to/Rscript --vanilla --default-packages=utils
args <- commandArgs(TRUE)
res <- try(install.packages(args))
if(inherits(res, "try-error")) q(status=1) else q()
```

```
## End(Not run)
```

RShowDoc *Show R Manuals and Other Documentation*

Description

Utility function to find and display R documentation.

Usage

```
RShowDoc(what, type = c("pdf", "html", "txt"), package)
```

Arguments

what	a character string: see 'Details'.
type	an optional character string giving the preferred format. Can be abbreviated.
package	an optional character string specifying the name of a package within which to look for documentation.

Details

what can specify one of several different sources of documentation, including the R manuals (R-admin, R-data, R-exts, R-intro, R-ints, R-lang), NEWS, COPYING (the GPL licence), any of the licenses in 'share/licenses', FAQ (also available as R-FAQ), and the files in '*R_HOME*/doc'.

Only on Windows, the R for Windows FAQ is specified by rw-FAQ.

If package is supplied, documentation is looked for in the 'doc' and top-level directories of an installed package of that name.

If what is missing a brief usage message is printed.

The documentation types are tried in turn starting with the first specified in type (or "pdf" if none is specified).

Value

A invisible character string given the path to the file found.

See Also

For displaying regular help files, help (or ?) and help.start.

For type = "txt", file.show is used. vignettes are nicely viewed via RShowDoc(*, package= .).

Examples

```
RShowDoc("R-lang")
RShowDoc("FAQ", type = "html")
RShowDoc("frame", package = "grid")
RShowDoc("changes.txt", package = "grid")
RShowDoc("NEWS", package = "MASS")
```

RSiteSearch *Search for Key Words or Phrases in Documentation*

Description

Search for key words or phrases in various documentation, such as R manuals, help pages of base and CRAN packages, vignettes, task views and others, using the search engine at https://search.r-project.org and view them in a web browser.

Usage

```
RSiteSearch(string,
            restrict = c("functions", "descriptions", "news", "Rfunctions",
                         "Rmanuals", "READMEs", "views", "vignettes"),
            format,
            sortby = c("score", "date:late", "date:early", "subject",
                       "subject:descending", "size", "size:descending"),
            matchesPerPage = 20,
            words = c("all", "any"))
```

Arguments

string	A character string specifying word(s) or phrase(s) to search. If the words are to be searched as one entity, enclose them either in (escaped) quotes or in braces.
restrict	A character vector, typically of length greater than one. Values can be abbreviated. Possible areas to search in: functions for help pages of CRAN packages, descriptions for extended descriptions of CRAN packages, news for package NEWS, Rfunctions for help pages of R base packages, Rmanuals for R manuals, READMEs for 'README' files of CRAN packages, views for task views, vignettes for package vignettes.
format	deprecated.
sortby	character string (can be abbreviated) indicating how to sort the search results: (score, date:late for sorting by date with latest results first, date:early for earliest first, subject for captions in alphabetical order, subject:descending for reverse alphabetical order, size or size:descending for size.)
matchesPerPage	How many items to show per page.
words	Show results matching all words/phrases (default) or any of them.

Details

This function is designed to work with the search site at https://search.r-project.org.

Unique partial matches will work for all arguments. Each new browser window will stay open unless you close it.

Value

(Invisibly) the complete URL passed to the browser, including the query string.

Author(s)

Andy Liaw and Jonathan Baron and Gennadiy Starostin

See Also

help.search, help.start for local searches.

browseURL for how the help file is displayed.

Examples

```
# need Internet connection
## for phrase searching you may use (escaped) double quotes or brackets
RSiteSearch("{logistic regression} \"glm object\"")
RSiteSearch('"logistic regression"')

## Search in vignettes and help files of R base packages
## store the query string:
fullquery <- RSiteSearch("lattice", restrict = c("vignettes","Rfunctions"))
fullquery # a string of 112 characters
```

---

rtags *An Etags-like Tagging Utility for R*

Description

rtags provides etags-like indexing capabilities for R code, using R's own parser.

Usage

```
rtags(path = ".", pattern = "\\.[RrSs]$",
      recursive = FALSE,
      src = list.files(path = path, pattern = pattern,
                       full.names = TRUE,
                       recursive = recursive),
      keep.re = NULL,
      ofile = "", append = FALSE,
      verbose = getOption("verbose"),
      type = c("etags", "ctags"))
```

Arguments

`path`, `pattern`, `recursive`
: Arguments passed on to `list.files` to determine the files to be tagged. By default, these are all files with extension '`.R`', '`.r`', '`.S`', and '`.s`' in the current directory. These arguments are ignored if `src` is specified.

`src`
: A vector of file names to be indexed.

`keep.re`
: A regular expression further restricting `src` (the files to be indexed). For example, specifying `keep.re = "/R/[^/]*\\.R$"` will only retain files with extension '`.R`' inside a directory named '`R`'.

`ofile`
: Passed on to `cat` as the `file` argument; typically the output file where the tags will be written (`"TAGS"` or `"tags"` by convention). By default, the output is written to the R console (unless redirected).

`append`
: Logical, indicating whether the output should overwrite an existing file, or append to it.

`verbose`
: Logical. If `TRUE`, file names are echoed to the R console as they are processed.

`type`
: Character string specifying whether emacs style (`"etags"`) or vi style (`"ctags"`) tags are to be generated.

Details

Many text editors allow definitions of functions and other language objects to be quickly and easily located in source files through a tagging utility. This functionality requires the relevant source files to be preprocessed, producing an index (or tag) file containing the names and their corresponding locations. There are multiple tag file formats, the most popular being the vi-style ctags format and the and emacs-style etags format. Tag files in these formats are usually generated by the `ctags` and `etags` utilities respectively. Unfortunately, these programs do not recognize R code syntax. They do allow tagging of arbitrary language files through regular expressions, but this too is insufficient.

The `rtags` function is intended to be a tagging utility for R code. It parses R code files (using R's parser) and produces tags in both etags and ctags formats. The support for vi-style ctags is rudimentary, and was adapted from a patch by Neal Fultz; see PR#17214.

It may be more convenient to use the command-line wrapper script `R CMD rtags`.

Author(s)

Deepayan Sarkar

References

https://en.wikipedia.org/wiki/Ctags, https://www.gnu.org/software/emacs/manual/html_node/emacs/Tags-Tables.html

See Also

`list.files`, `cat`

Examples

```
## Not run:
rtags("/path/to/src/repository",
      pattern = "[.]*\\.[RrSs]$",
      keep.re = "/R/",
      verbose = TRUE,
      ofile = "TAGS",
      append = FALSE,
      recursive = TRUE)

## End(Not run)
```

Rtangle *R Driver for* Stangle

Description

A driver for Stangle that extracts R code chunks. Notably all RtangleSetup() arguments may be used as arguments in the Stangle() call.

Usage

```
Rtangle()
RtangleSetup(file, syntax, output = NULL, annotate = TRUE,
             split = FALSE, quiet = FALSE, drop.evalFALSE = FALSE, ...)
```

Arguments

file	name of Sweave source file. See the description of the corresponding argument of Sweave.
syntax	an object of class SweaveSyntax.
output	name of output file used unless split = TRUE: see 'Details'.
annotate	a logical or function. When true, as by default, code chunks are separated by comment lines specifying the names and line numbers of the code chunks. If FALSE the decorating comments are omitted. Alternatively, annotate may be a function, see section 'Chunk annotation'.
split	split output into a file for each code chunk?
quiet	logical to suppress all progress messages.
drop.evalFALSE	logical; When false, as by default, all chunks with option eval = FALSE are *commented out* in the output; otherwise (drop.evalFALSE = TRUE) they are omitted entirely.
...	additional named arguments setting defaults for further options listed in 'Supported Options'.

Details

Unless `split = TRUE`, the default name of the output file is `basename(file)` with an extension corresponding to the Sweave syntax (e.g., '`Rnw`', '`Stex`') replaced by '`R`'. File names `"stdout"` and `"stderr"` are interpreted as the output and message connection respectively.

If splitting is selected (including by the options in the file), each chunk is written to a separate file with extension the name of the 'engine' (default '`.R`').

Note that this driver does more than simply extract the code chunks verbatim, because chunks may re-use earlier chunks.

Chunk annotation (`annotate`)

By default `annotate = TRUE`, the annotation is of one of the forms

```
##################################################
### code chunk number 3: viewport
##################################################

##################################################
### code chunk number 18: grid.Rnw:647-648
##################################################

##################################################
### code chunk number 19: trellisdata (eval = FALSE)
##################################################
```

using either the chunk label (if present, i.e., when specified in the source) or the file name and line numbers.

`annotate` may be a function with formal arguments `(options, chunk, output)`, e.g. to produce less dominant chunk annotations; see `Rtangle()$runcode` how it is called instead of the default.

Supported Options

`Rtangle` supports the following options for code chunks (the values in parentheses show the default values):

engine: character string (`"R"`). Only chunks with `engine` equal to `"R"` or `"S"` are processed.

keep.source: logical (`TRUE`). If `keep.source == TRUE` the original source is copied to the file. Otherwise, deparsed source is output.

eval: logical (`TRUE`). If `FALSE`, the code chunk is copied across but commented out.

prefix Used if `split = TRUE`. See `prefix.string`.

prefix.string: a character string, default is the name of the source file (without extension). Used if `split = TRUE` as the prefix for the filename if the chunk has no label, or if it has a label and `prefix = TRUE`. Note that this is used as part of filenames, so needs to be portable.

show.line.nos logical (`FALSE`). Should the output be annotated with comments showing the line number of the first code line of the chunk?

Author(s)

Friedrich Leisch and R-core.

See Also

'Sweave User Manual', a vignette in the **utils** package.

Sweave, RweaveLatex

Examples

```
nmRnw <- "example-1.Rnw"
exfile <- system.file("Sweave", nmRnw, package = "utils")
## Create R source file
Stangle(exfile)
nmR <- sub("Rnw$", "R", nmRnw) # the (default) R output file name
if(interactive()) file.show("example-1.R")

## Smaller R source file with custom annotation:
my.Ann <- function(options, chunk, output) {
  cat("### chunk #", options$chunknr, ": ",
      if(!is.null(ol <- options$label)) ol else .RtangleCodeLabel(chunk),
      if(!options$eval) " (eval = FALSE)", "\n",
      file = output, sep = "")
}
Stangle(exfile, annotate = my.Ann)
if(interactive()) file.show("example-1.R")

Stangle(exfile, annotate = my.Ann, drop.evalFALSE=TRUE)
if(interactive()) file.show("example-1.R")
```

RweaveLatex *R/LaTeX Driver for Sweave*

Description

A driver for Sweave that translates R code chunks in LaTeX files by "running them", i.e., parse() and eval() each.

Usage

```
RweaveLatex()

RweaveLatexSetup(file, syntax, output = NULL, quiet = FALSE,
                 debug = FALSE, stylepath, ...)
```

Arguments

file	Name of Sweave source file. See the description of the corresponding argument of Sweave.
syntax	An object of class SweaveSyntax.
output	Name of output file. The default is to remove extension '.nw', '.Rnw' or '.Snw' and to add extension '.tex'. Any directory paths in file are also removed such that the output is created in the current working directory.
quiet	If TRUE all progress messages are suppressed.
debug	If TRUE, input and output of all code chunks is copied to the console.
stylepath	See 'Details'.
...	named values for the options listed in 'Supported Options'.

Details

The LaTeX file generated needs to contain the line '\usepackage{Sweave}', and if this is not present in the Sweave source file (possibly in a comment), it is inserted by the RweaveLatex driver as last line before the '\begin{document}' statement. If stylepath = TRUE, a hard-coded path to the file 'Sweave.sty' in the R installation is set in place of Sweave. The hard-coded path makes the LaTeX file less portable, but avoids the problem of installing the current version of 'Sweave.sty' to some place in your TeX input path. However, TeX may not be able to process the hard-coded path if it contains spaces (as it often will under Windows) or TeX special characters.

The default for stylepath is now taken from the environment variable SWEAVE_STYLEPATH_DEFAULT, or is FALSE it that is unset or empty. If set, it should be exactly TRUE or FALSE: any other values are taken as FALSE.

The simplest way for frequent Sweave users to ensure that 'Sweave.sty' is in the TeX input path is to add '*R_HOME*/share/texmf' as a 'texmf tree' ('root directory' in the parlance of the 'MiKTeX settings' utility).

By default, 'Sweave.sty' loads the 'graphicx' LaTeX package and sets the width of all included graphics to:
'\setkeys{Gin}{width=0.8\textwidth}'.

This setting (defined in the 'graphicx' package) affects the width size option passed to the '\includegraphics{}' directive for each plot file and in turn impacts the scaling of your plot files as they will appear in your final document.

Thus, for example, you may set width=3 in your figure chunk and the generated graphics files will be set to 3 inches in width. However, the width of your graphic in your final document will be set to '0.8\textwidth' and the height dimension will be scaled accordingly. Fonts and symbols will be similarly scaled in the final document.

You can adjust the default value by including the '\setkeys{Gin}{width=...}' directive in your '.Rnw' file after the '\begin{document}' directive and changing the width option value as you prefer, using standard LaTeX measurement values.

If you wish to override this default behavior entirely, you can add a '\usepackage[nogin]{Sweave}' directive in your preamble. In this case, no size/scaling options will be passed to the '\includegraphics{}' directive and the height and width options

will determine both the runtime generated graphic file sizes and the size of the graphics in your final document.

'Sweave.sty' also supports the '[nofontenc]' option, which skips the default inclusion of '\usepackage[T1]{fontenc}' for pdfTeX processing.

It also supports the '[inconsolata]' option, to render monospaced text in inconsolata, the font used by default for R help pages.

The use of fancy quotes (see sQuote) can cause problems when setting R output in non-UTF-8 locales (note that pdfTeX assumes UTF-8 by default since 2018). Either set options(useFancyQuotes = FALSE) or arrange that LaTeX is aware of the encoding used and ensure that typewriter fonts containing directional quotes are used.

Some LaTeX graphics drivers do not include '.png' or '.jpg' in the list of known extensions. To enable them, add something like '\DeclareGraphicsExtensions{.png,.pdf,.jpg}' to the preamble of your document or check the behavior of your graphics driver. When both pdf and png are TRUE both files will be produced by Sweave, and their order in the 'DeclareGraphicsExtensions' list determines which will be used by pdflatex.

Supported Options

RweaveLatex supports the following options for code chunks (the values in parentheses show the default values). Character string values should be quoted when passed from Sweave through ... but not when used in the header of a code chunk.

engine: character string ("R"). Only chunks with engine equal to "R" or "S" are processed.

echo: logical (TRUE). Include R code in the output file?

keep.source: logical (TRUE). When echoing, if TRUE the original source is copied to the file. Otherwise, deparsed source is echoed.

eval: logical (TRUE). If FALSE, the code chunk is not evaluated, and hence no text nor graphical output produced.

results: character string ("verbatim"). If "verbatim", the output of R commands is included in the verbatim-like 'Soutput' environment. If "tex", the output is taken to be already proper LaTeX markup and included as is. If "hide" then all output is completely suppressed (but the code executed during the weave). Values can be abbreviated.

print: logical (FALSE). If TRUE, this forces auto-printing of all expressions.

term: logical (TRUE). If TRUE, visibility of values emulates an interactive R session: values of assignments are not printed, values of single objects are printed. If FALSE, output comes only from explicit print or similar statements.

split: logical (FALSE). If TRUE, text output is written to separate files for each code chunk.

strip.white: character string ("true"). If "true", blank lines at the beginning and end of output are removed. If "all", then all blank lines are removed from the output. If "false" then blank lines are retained.

A 'blank line' is one that is empty or includes only whitespace (spaces and tabs).

Note that blank lines in a code chunk will usually produce a prompt string rather than a blank line on output.

prefix: logical (TRUE). If TRUE generated filenames of figures and output all have the common prefix given by the prefix.string option: otherwise only unlabelled chunks use the prefix.

prefix.string: a character string, default is the name of the source file (without extension). Note that this is used as part of filenames, so needs to be portable.

include: logical (TRUE), indicating whether input statements for text output (if split = TRUE) and '\includegraphics' statements for figures should be auto-generated. Use include = FALSE if the output should appear in a different place than the code chunk (by placing the input line manually).

fig: logical (FALSE), indicating whether the code chunk produces graphical output. Note that only one figure per code chunk can be processed this way. The labels for figure chunks are used as part of the file names, so should preferably be alphanumeric.

eps: logical (FALSE), indicating whether EPS figures should be generated. Ignored if fig = FALSE.

pdf: logical (TRUE), indicating whether PDF figures should be generated. Ignored if fig = FALSE.

pdf.version, pdf.encoding, pdf.compress: passed to pdf to set the version, encoding and compression (or not). Defaults taken from pdf.options().

png: logical (FALSE), indicating whether PNG figures should be generated. Ignored if fig = FALSE. Only available in R $\geq$ 2.13.0.

jpeg: logical (FALSE), indicating whether JPEG figures should be generated. Ignored if fig = FALSE. Only available in R $\geq$ 2.13.0.

grdevice: character (NULL): see section 'Custom Graphics Devices'. Ignored if fig = FALSE. Only available in R $\geq$ 2.13.0.

width: numeric (6), width of figures in inches. See 'Details'.

height: numeric (6), height of figures in inches. See 'Details'.

resolution: numeric (300), resolution in pixels per inch: used for PNG and JPEG graphics. Note that the default is a fairly high value, appropriate for high-quality plots. Something like 100 is a better choice for package vignettes.

concordance: logical (FALSE). Write a concordance file to link the input line numbers to the output line numbers.

figs.only: logical (FALSE). By default each figure chunk is run once, then re-run for each selected type of graphics. That will open a default graphics device for the first figure chunk and use that device for the first evaluation of all subsequent chunks. If this option is true, the figure chunk is run only for each selected type of graphics, for which a new graphics device is opened and then closed.

In addition, users can specify further options, either in the header of an individual code section or in a '\SweaveOpts{}' line in the document. For unknown options, their type is set at first use.

Custom Graphics Devices

If option grdevice is supplied for a code chunk with both fig and eval true, the following call is made

```
get(options$grdevice, envir = .GlobalEnv)(name=, width=,
                                          height=, options)
```

which should open a graphics device. The chunk's code is then evaluated and dev.off is called. Normally a function of the name given will have been defined earlier in the Sweave document, e.g.

```
<<results=hide>>=
my.Swd <- function(name, width, height, ...)
  grDevices::png(filename = paste(name, "png", sep = "."),
                 width = width, height = height, res = 100,
                 units = "in", type = "quartz", bg = "transparent")
@
```

Alternatively for R >= 3.4.0, if the function exists in a package (rather than the `.GlobalEnv`) it can be used by setting `grdevice = "pkg::my.Swd"` (or with ':::' instead of '::' if the function is not exported).

Currently only one custom device can be used for each chunk, but different devices can be used for different chunks.

A replacement for `dev.off` can be provided as a function with suffix `.off`, e.g. `my.Swd.off()` or `pkg::my.Swd.off()`, respectively.

Hook Functions

Before each code chunk is evaluated, zero or more hook functions can be executed. If `getOption("SweaveHooks")` is set, it is taken to be a named list of hook functions. For each *logical* option of a code chunk (`echo`, `print`, ...) a hook can be specified, which is executed if and only if the respective option is `TRUE`. Hooks must be named elements of the list returned by `getOption("SweaveHooks")` and be functions taking no arguments. E.g., if option `"SweaveHooks"` is defined as `list(fig = foo)`, and `foo` is a function, then it would be executed before the code in each figure chunk. This is especially useful to set defaults for the graphical parameters in a series of figure chunks.

Note that the user is free to define new Sweave logical options and associate arbitrary hooks with them. E.g., one could define a hook function for a new option called `clean` that removes all objects in the workspace. Then all code chunks specified with `clean = TRUE` would start operating on an empty workspace.

Author(s)

Friedrich Leisch and R-core

See Also

'Sweave User Manual', a vignette in the **utils** package.

`Sweave`, `Rtangle`

`Rwin configuration` *R for Windows Configuration*

Description

The file 'Rconsole' configures the R GUI (Rgui) console under MS Windows and loadRconsole(*) loads a new configuration.

The file 'Rdevga' configures the graphics devices windows, win.graph, win.metafile and win.print, as well as the bitmap devices bmp, jpeg, png and tiff (which use for type = "windows" use windows internally).

Usage

```
loadRconsole(file)
```

Arguments

file	The file from which to load a new 'Rconsole' configuration. By default a file dialog is used to select a file.

Details

There are system copies of these files in '*R_HOME*\etc'. Users can have personal copies of the files: these are looked for in the location given by the environment variable R_USER. The system files are read only if a corresponding personal file is not found.

If the environment variable R_USER is not set, the R system sets it to HOME if that is set (stripping any trailing slash), otherwise to the Windows 'personal' directory, otherwise to {HOMEDRIVE}{HOMEPATH} if HOMEDRIVE and HOMEDRIVE are both set otherwise to the working directory. This is as described in the file 'rw-FAQ'.

Value

Each of the files contains details in its comments of how to set the values.

At the time of writing 'Rdevga' configured the mapping of font numbers to fonts, and 'Rconsole' configured the appearance (single or multiple document interface, toolbar, status bar on MDI), size, font and colours of the GUI console, and whether resizing the console sets options("width").

The file 'Rconsole' also configures the internal pager. This shares the font and colours of the console, but can be sized separately.

'Rconsole' can also set the initial positions of the console and the graphics device, as well as the size and position of the MDI workspace in MDI mode.

loadRconsole is called for its side effect of loading new defaults. It returns no useful value.

Chinese/Japanese/Korean

Users of these languages will need to select a suitable font for the console (perhaps MS Mincho) and for the graphics device (although the default Arial has many East Asian characters). It is essential that the font selected for the console has double-width East Asian characters – many monospaced fonts do not.

Note

The GUI preferences item on the Edit menu brings up an dialog box which can be used to edit the console settings, and to save them to a file.

This is only available on Windows.

Author(s)

Guido Masarotto and R-core members

See Also

windows

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
  ruser <- Sys.getenv("R_USER")
  cat("\n\nLocation for personal configuration files is\n   R_USER = ",
      ruser, "\n\n", sep = "")
  ## see if there are personal configuration files
  file.exists(file.path(ruser, c("Rconsole", "Rdevga")))

  ## show the configuration files used
  showConfig <- function(file)
  {
      ruser <- Sys.getenv("R_USER")
      path <- file.path(ruser, file)
      if(!file.exists(path)) path <- file.path(R.home(), "etc", file)
      file.show(path, header = path)
  }
  showConfig("Rconsole")
})
```

---

savehistory *Load or Save or Display the Commands History*

Description

Load or save or display the commands history.

Usage

```
loadhistory(file = ".Rhistory")
savehistory(file = ".Rhistory")

history(max.show = 25, reverse = FALSE, pattern, ...)

timestamp(stamp = date(),
          prefix = "##------ ", suffix = " ------##",
          quiet = FALSE)
```

Arguments

file	The name of the file in which to save the history, or from which to load it. The path is relative to the current working directory.
max.show	The maximum number of lines to show. Inf will give all of the currently available history.
reverse	logical. If true, the lines are shown in reverse order. Note: this is not useful when there are continuation lines.
pattern	A character string to be matched against the lines of the history. When supplied, only *unique* matching lines are shown.
...	Arguments to be passed to grep when doing the matching.
stamp	A value or vector of values to be written into the history.
prefix	A prefix to apply to each line.
suffix	A suffix to apply to each line.
quiet	If TRUE, suppress printing timestamp to the console.

Details

There are several history mechanisms available for the different R consoles, which work in similar but not identical ways. Notably, there are different implementations for Unix and Windows.

Windows: The functions described here work in Rgui and interactive Rterm but not in batch use of Rterm nor in embedded/DCOM versions.

Unix-alikes: The functions described here work under the readline command-line interface but may not otherwise (for example, in batch use or in an embedded application). Note that R can be built without readline.

R.app, the console on macOS, has a separate and largely incompatible history mechanism, which by default uses a file '.Rapp.history' and saves up to 250 entries. These functions are not currently implemented there.

The (readline on Unix-alikes) history mechanism is controlled by two environment variables: R_HISTSIZE controls the number of lines that are saved (default 512), and R_HISTFILE (default '.Rhistory') sets the filename used for the loading/saving of history if requested at the beginning/end of a session (but not the default for loadhistory/savehistory). There is no limit on the number of lines of history retained during a session, so setting R_HISTSIZE to a large value has no penalty unless a large file is actually generated.

These environment variables are read at the time of saving, so can be altered within a session by the use of Sys.setenv.

On Unix-alikes: Note that readline history library saves files with permission 0600, that is with read/write permission for the user and not even read permission for any other account.

The timestamp function writes a timestamp (or other message) into the history and echos it to the console. On platforms that do not support a history mechanism only the console message is printed.

Note

If you want to save the history at the end of (almost) every interactive session (even those in which you do not save the workspace), you can put a call to savehistory() in .Last. See the examples.

Examples

```
## Not run:
## Save the history in the home directory: note that it is not
## (by default) read from there but from the current directory
.Last <- function()
    if(interactive()) try(savehistory("~/.Rhistory"))

## End(Not run)
```

`select.list` *Select Items from a List*

Description

Select item(s) from a character vector.

Usage

```
select.list(choices, preselect = NULL, multiple = FALSE,
            title = NULL, graphics = getOption("menu.graphics"))
```

Arguments

`choices`	a character vector of items.
`preselect`	a character vector, or `NULL`. If non-null and if the string(s) appear in the list, the item(s) are selected initially.
`multiple`	logical: can more than one item be selected?
`title`	optional character string for window title, or `NULL` for no title.
`graphics`	logical: should a graphical widget be used?

Details

The normal default is `graphics = TRUE`.

On Windows, this brings up a modal dialog box with a (scrollable) list of items, which can be selected by the mouse. If `multiple` is true, further items can be selected or deselected by holding the control key down whilst selecting, and shift-clicking can be used to select ranges. Normal termination is via the 'OK' button or by hitting Enter or double-clicking an item. Selection can be aborted via the 'Cancel' button or pressing Escape.

Under the macOS GUI, this brings up a modal dialog box with a (scrollable) list of items, which can be selected by the mouse.

On other Unix-like platforms it will use a Tcl/Tk listbox widget if possible.

If `graphics` is FALSE or no graphical widget is available it displays a text list from which the user can choose by number(s). The `multiple = FALSE` case uses `menu`. Preselection is only supported for `multiple = TRUE`, where it is indicated by a `"+"` preceding the item.

It is an error to use `select.list` in a non-interactive session.

Value

A character vector of selected items. If `multiple` is false and no item was selected (or `Cancel` was used), `""` is returned. If `multiple` is true and no item was selected (or `Cancel` was used) then a character vector of length 0 is returned.

See Also

`menu`, `tk_select.list` for a graphical version using Tcl/Tk.

Examples

```
## Not run:
select.list(sort(.packages(all.available = TRUE)))

## End(Not run)
```

`sessionInfo` *Collect Information About the Current R Session*

Description

Get and report version information about R, the OS and attached or loaded packages.

The `print()` and `toLatex()` methods (for a `"sessionInfo"` object) show the locale and timezone information by default, when `locale` or `tzone` are true. The `system.codepage` is only shown when it is not empty, i.e., only on Windows, *and* if it differs from `code.page`, see below or `l10n_info()`.

Usage

```
sessionInfo(package = NULL)
## S3 method for class 'sessionInfo'
print(x, locale = TRUE, tzone = locale,
      RNG = !identical(x$RNGkind, .RNGdefaults), ...)
## S3 method for class 'sessionInfo'
toLatex(object, locale = TRUE, tzone = locale,
        RNG = !identical(object$RNGkind, .RNGdefaults), ...)
osVersion
```

Arguments

`package`	a character vector naming installed packages, or `NULL` (the default) meaning all attached packages.
`x`	an object of class `"sessionInfo"`.
`object`	an object of class `"sessionInfo"`.
`locale`	show locale, by default `tzone`, and (on Windows) code page information?
`tzone`	show time zone information?
`RNG`	show information on `RNGkind()`? Defaults to true iff it differs from the R version's default, i.e., `RNGversion(*)`.
`...`	currently not used.

Value

`sessionInfo()` returns an object of class `"sessionInfo"` which has `print` and `toLatex` methods. This is a list with components

`R.version`	a list, the result of calling `R.Version()`.
`platform`	a character string describing the platform R was built under. Where sub-architectures are in use this is of the form '`platform/sub-arch`': 32-bit builds have `(32-bit)` appended
`running`	a character string (or possibly `NULL`), the same as `osVersion`, see below.
`RNGkind`	a character vector, the result of calling `RNGkind()`.
`matprod`	a character string, the result of calling `getOption("matprod")`.
`BLAS`	a character string, the result of calling `extSoftVersion()["BLAS"]`.
`LAPACK`	a character string, the result of calling `La_library()`.
`LA_version`	a character string, the result of calling `La_version()`.
`locale`	a character string, the result of calling `Sys.getlocale()`.
`tzone`	a character string, the result of calling `Sys.timezone()`.
`tzcode_type`	a character string indicating source (system/internal) of the date-time conversion and printing functions.
`basePkgs`	a character vector of base packages which are attached.
`otherPkgs`	(not always present): a named list of the results of calling `packageDescription` on other attached packages.
`loadedOnly`	(not always present): a named list of the results of calling `packageDescription` on packages whose namespaces are loaded but are not attached.

`osVersion`

`osVersion` is a character string (or possibly `NULL` on bizarre platforms) describing the OS and version which it is running under (as distinct from built under). This attempts to name a Linux distribution and give the OS name on an Apple Mac.

It is the same as `sessionInfo()$running` and created when loading the **utils** package.

Windows may report unexpected versions: see the help for `win.version`.

How OSes identify themselves and their versions can be arcane: where possible `osVersion` (and hence `sessionInfo()$running`) uses a human-readable form.

Where R was compiled under macOS 10.x (as the CRAN Intel distributions were prior to R 4.3.0) but running under 'Big Sur' or later, macOS reports itself as '`10.16`' (which R recognizes as 'Big Sur ...') and not '11', '12', ...

Note

The information on 'loaded' packages and namespaces is the *current* version installed at the location the package was loaded from: it can be wrong if another process has been changing packages during the session.

See Also

R.version, R_compiled_by

Examples

```
sI <- sessionInfo()
sI
# The same, showing the RNGkind, but not the locale :
  print(sI, RNG = TRUE, locale = FALSE)
toLatex(sI, locale = FALSE) # shortest; possibly desirable at end of report
```

---

setRepositories *Select Package Repositories*

Description

Interact with the user to choose the package repositories to be used.

Usage

```
setRepositories(graphics = getOption("menu.graphics"),
                ind = NULL, addURLs = character(), name = NULL)
```

Arguments

graphics	Logical. If true, use a graphical list: on Windows or macOS GUI use a list box, and on a Unix-alike if **tcltk** and an X server are available, use Tk widget. Otherwise use a text menu.
ind	NULL or a vector of integer indices, which have the same effect as if they were entered at the prompt for graphics = FALSE.
name	NULL or character vector of names of the repositories in the repository table which has the same effect as passing the corresponding indices to ind.
addURLs	A character vector of additional URLs: it is often helpful to use a named vector.

Details

The default list of known repositories is stored in the file '*R_HOME*/etc/repositories'. That file can be edited for a site, or a user can have a personal copy in the file pointed to by the environment variable R_REPOSITORIES, or if this is unset, NULL or does not exist, in '*HOME*/.R/repositories', which will take precedence.

A Bioconductor mirror can be selected by setting options("BioC_mirror"), e.g. via chooseBioCmirror — the default value is '"https://bioconductor.org"'. This version of R chooses Bioconductor version 3.19 by default, but that can be changed via the environment variable R_BIOC_VERSION.

The items that are preselected are those that are currently in options("repos") plus those marked as default in the list of known repositories.

The list of repositories offered depends on the setting of option "pkgType" as some repositories only offer a subset of types (e.g., only source packages or not macOS binary packages). Further, for binary packages some repositories (notably R-Forge) only offer packages for the current or recent versions of R. (Type "both" is equivalent to "source".)

Repository 'CRAN' is treated specially: the value is taken from the current setting of getOption("repos") if this has an element "CRAN": this ensures mirror selection is sticky.

This function requires the R session to be interactive unless ind or name is supplied. The latter overrides the former if both are supplied and values are not case-sensitive. If any of the supplied names does not match, an error is raised.

Value

This function is invoked mainly for its side effect of updating options("repos"). It returns (invisibly) the previous repos options setting (as a list with component repos) or NULL if no changes were applied.

Note

This does **not** set the list of repositories at startup: to do so set options(repos =) in a start up file (see help topic Startup) or via a customized 'repositories' file.

See Also

chooseCRANmirror, chooseBioCmirror, install.packages.

Examples

```
## Not run:
setRepositories(addURLs =
                c(CRANxtras = "https://www.stats.ox.ac.uk/pub/RWin"))

## End(Not run)
oldrepos <- setRepositories(name = c("CRAN", "R-Forge"))
getOption("repos")
options(oldrepos) # restore
```

---

setWindowTitle *Set the Window Title or the Status Bar of the RGui in Windows*

Description

Set or get the title of the R (i.e. RGui) window which will appear in the task bar, or set the status bar (if in use).

Usage

```
setWindowTitle(suffix, title = paste(getIdentification(), suffix))

getWindowTitle()

getIdentification()

setStatusBar(text)
```

Arguments

suffix	a character string to form part of the title
title	a character string forming the complete new title
text	a character string of up to 255 characters, to be displayed in the status bar.

Details

setWindowTitle appends suffix to the normal window identification (RGui, R Console or Rterm). Use suffix = "" to reset the title.

getWindowTitle gets the current title.

This sets the title of the frame in MDI mode, the title of the console for RGui --sdi, and the title of the window from which it was launched for Rterm. It has no effect in embedded uses of R.

getIdentification returns the normal window identification.

setStatusBar sets the text in the status bar of an MDI frame: if this is not currently shown it is selected and shown.

Value

The first three functions return a length 1 character vector.

setWindowTitle returns the previous window title (invisibly).

getWindowTitle and getIdentification return the current window title and the normal window identification, respectively.

Note

These functions are only available on Windows and only make sense when using the Rgui. E.g., in Rterm (and hence in ESS) the title is not visible (but can be set and gotten), and in a version of RStudio it has been "", invariably.

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({
## show the current working directory in the title, saving the old one
oldtitle <- setWindowTitle(getwd())
Sys.sleep(0.5)
## reset the title
setWindowTitle("")
```

```
Sys.sleep(0.5)
## restore the original title
setWindowTitle(title = oldtitle)
})
```

SHLIB *Build Shared Object/DLL for Dynamic Loading*

Description

Compile the given source files and then link all specified object files into a shared object aka DLL which can be loaded into R using `dyn.load` or `library.dynam`.

Usage

```
R CMD SHLIB [options] [-o dllname] files
```

Arguments

`files`	a list specifying the object files to be included in the shared object/DLL. You can also include the name of source files (for which the object files are automagically made from their sources) and library linking commands.
`dllname`	the full name of the shared object/DLL to be built, including the extension (typically '`.so`' on Unix systems, and '`.dll`' on Windows). If not given, the basename of the object/DLL is taken from the basename of the first file.
`options`	Further options to control the processing. Use `R CMD SHLIB --help` for a current list.

Details

`R CMD SHLIB` is the mechanism used by `INSTALL` to compile source code in packages. It will generate suitable compilation commands for C, C++, Objective C(++) and Fortran sources: Fortran 90/95 sources can also be used but it may not be possible to mix these with other languages (on most platforms it is possible to mix with C, but mixing with C++ rarely works).

Please consult section 'Creating shared objects' in the manual 'Writing R Extensions' for how to customize it (for example to add `cpp` flags and to add libraries to the link step) and for details of some of its quirks.

Items in `files` with extensions '`.c`', '`.cpp`', '`.cc`', '`.C`', '`.f`', '`.f90`', '`.f95`', '`.m`' (Objective-C), '`.M`' and '`.mm`' (Objective-C++) are regarded as source files, and those with extension '`.o`' as object files. All other items are passed to the linker.

Objective C(++) support is optional when R was configured: their main usage is on macOS.

Note that the appropriate run-time libraries will be used when linking if C++, Fortran or Objective C(++) sources are supplied, but not for compiled object files from these languages.

Option '`-n`' (also known as '`--dry-run`') will show the commands that would be run without actually executing them.

Note

Some binary distributions of R have SHLIB in a separate bundle, e.g., an R-devel RPM.

See Also

COMPILE, dyn.load, library.dynam.

The 'R Installation and Administration' and 'Writing R Extensions' manuals, including the section on 'Customizing package compilation' in the former.

Examples

```
## Not run:
# To link against a library not on the system library paths:
R CMD SHLIB -o mylib.so a.f b.f -L/opt/acml3.5.0/gnu64/lib -lacml

## End(Not run)
```

---

shortPathName *Express File Paths in Short Form on Windows*

Description

Convert file paths to the short form. This is an interface to the Windows API call GetShortPathNameW.

Usage

```
shortPathName(path)
```

Arguments

path	character vector of file paths.

Details

For most file systems, the short form is the 'DOS' form with 8+3 path components and no spaces, and this used to be guaranteed. But some file systems on recent versions of Windows do not have short path names when the long-name path will be returned instead.

Value

A character vector. The path separator will be '\'. If a file path does not exist, the supplied path will be returned with slashes replaced by backslashes.

Note

This is only available on Windows.

See Also

normalizePath.

Examples

```
if(.Platform$OS.type == "windows") withAutoprint({

  cat(shortPathName(c(R.home(), tempdir())), sep = "\n")

})
```

---

sourceutils *Source Reference Utilities*

Description

These functions extract information from source references.

Usage

```
getSrcFilename(x, full.names = FALSE, unique = TRUE)
getSrcDirectory(x, unique = TRUE)
getSrcref(x)
getSrcLocation(x, which = c("line", "column", "byte", "parse"),
               first = TRUE)
```

Arguments

x	An object (typically a function) containing source references.
full.names	Whether to include the full path in the filename result.
unique	Whether to list only unique filenames/directories.
which	Which part of a source reference to extract. Can be abbreviated.
first	Whether to show the first (or last) location of the object.

Details

Each statement of a function will have its own source reference if the "keep.source" option is TRUE. These functions retrieve all of them.

The components are as follows:

line The line number where the object starts or ends.

column The column number where the object starts or ends. Horizontal tabs are converted to spaces.

byte As for "column", but counting bytes, which may differ in case of multibyte characters (and horizontal tabs).

parse As for "line", but this ignores #line directives.

Value

`getSrcFilename` and `getSrcDirectory` return character vectors holding the filename/directory.

`getSrcref` returns a list of `"srcref"` objects or `NULL` if there are none.

`getSrcLocation` returns an integer vector of the requested type of locations.

See Also

`srcref`, `getParseData`

Examples

```
fn <- function(x) {
  x + 1 # A comment, kept as part of the source
}

# Show the temporary file directory
# where the example was saved

getSrcDirectory(fn)
getSrcLocation(fn, "line")
```

---

stack *Stack or Unstack Vectors from a Data Frame or List*

Description

Stacking vectors concatenates multiple vectors into a single vector along with a factor indicating where each observation originated. Unstacking reverses this operation.

Usage

```
stack(x, ...)
## Default S3 method:
stack(x, drop=FALSE, ...)
## S3 method for class 'data.frame'
stack(x, select, drop=FALSE, ...)

unstack(x, ...)
## Default S3 method:
unstack(x, form, ...)
## S3 method for class 'data.frame'
unstack(x, form, ...)
```

Arguments

x	a list or data frame to be stacked or unstacked.
select	an expression, indicating which variable(s) to select from a data frame.
form	a two-sided formula whose left side evaluates to the vector to be unstacked and whose right side evaluates to the indicator of the groups to create. Defaults to formula(x) in the data frame method for unstack.
drop	Whether to drop the unused levels from the "ind" column of the return value.
...	further arguments passed to or from other methods.

Details

The stack function is used to transform data available as separate columns in a data frame or list into a single column that can be used in an analysis of variance model or other linear model. The unstack function reverses this operation.

Note that stack applies to *vectors* (as determined by is.vector): non-vector columns (e.g., factors) will be ignored with a warning. Where vectors of different types are selected they are concatenated by unlist whose help page explains how the type of the result is chosen.

These functions are generic: the supplied methods handle data frames and objects coercible to lists by as.list.

Value

unstack produces a list of columns according to the formula form. If all the columns have the same length, the resulting list is coerced to a data frame.

stack produces a data frame with two columns:

values	the result of concatenating the selected vectors in x.
ind	a factor indicating from which vector in x the observation originated.

Author(s)

Douglas Bates

See Also

lm, reshape

Examples

```
require(stats)
formula(PlantGrowth)         # check the default formula
pg <- unstack(PlantGrowth)   # unstack according to this formula
pg
stack(pg)                    # now put it back together
stack(pg, select = -ctrl)    # omitting one vector
```

str *Compactly Display the Structure of an Arbitrary R Object*

Description

Compactly display the internal **str**ucture of an R object, a diagnostic function and an alternative to `summary` (and to some extent, `dput`). Ideally, only one line for each 'basic' structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls `args` for (non-primitive) function objects.

`strOptions()` is a convenience function for setting `options(str = .)`, see the examples.

Usage

```
str(object, ...)

## S3 method for class 'data.frame'
str(object, ...)

## Default S3 method:
str(object, max.level = NA,
    vec.len  = strO$vec.len, digits.d = strO$digits.d,
    nchar.max = 128, give.attr = TRUE,
    drop.deparse.attr = strO$drop.deparse.attr,
    give.head = TRUE, give.length = give.head,
    width = getOption("width"), nest.lev = 0,
    indent.str = paste(rep.int(" ", max(0, nest.lev + 1)),
                       collapse = ".."),
    comp.str = "$ ", no.list = FALSE, envir = baseenv(),
    strict.width = strO$strict.width,
    formatNum = strO$formatNum, list.len = strO$list.len,
    deparse.lines = strO$deparse.lines, ...)

strOptions(strict.width = "no", digits.d = 3, vec.len = 4,
           list.len = 99, deparse.lines = NULL,
           drop.deparse.attr = TRUE,
           formatNum = function(x, ...)
                       format(x, trim = TRUE, drop0trailing = TRUE, ...))
```

Arguments

`object`	any R object about which you want to have some information.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default NA: Display all nesting levels.
`vec.len`	numeric (>= 0) indicating how many 'first few' elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending

on the kind of vector. Defaults to the `vec.len` component of option `"str"` (see `options`) which defaults to 4.

`digits.d` number of digits for numerical components (as for `print`). Defaults to the `digits.d` component of option `"str"` which defaults to 3.

`nchar.max` maximal number of characters to show for `character` strings. Longer strings are truncated, see `longch` example below.

`give.attr` logical; if `TRUE` (default), show attributes as sub structures.

`drop.deparse.attr` logical; if `TRUE` (default), `deparse(control = control)` will not have `"showAttributes"` in *control*. Used to be hard coded to `FALSE` and hence can be set via `strOptions()` for back compatibility.

`give.length` logical; if `TRUE` (default), indicate length (as `[1:...]`).

`give.head` logical; if `TRUE` (default), give (possibly abbreviated) mode/class and length (as *type*`[1:...]`).

`width` the page width to be used. The default is the currently active `options("width")`; note that this has only a weak effect, unless `strict.width` is not `"no"`.

`nest.lev` current nesting level in the recursive calls to `str`.

`indent.str` the indentation string to use.

`comp.str` string to be used for separating list components.

`no.list` logical; if true, no 'list of ...' nor the class are printed.

`envir` the environment to be used for *promise* (see `delayedAssign`) objects only.

`strict.width` string indicating if the `width` argument's specification should be followed strictly, one of the values `c("no", "cut", "wrap")`, which can be abbreviated. Defaults to the `strict.width` component of option `"str"` (see `options`) which defaults to `"no"` for back compatibility reasons; `"wrap"` uses `strwrap(*, width = width)` whereas `"cut"` cuts directly to `width`. Note that a small `vec.length` may be better than setting `strict.width = "wrap"`.

`formatNum` a function such as `format` for formatting numeric vectors. It defaults to the `formatNum` component of option `"str"`, see "Usage" of `strOptions()` above, which is almost back compatible to R <= 2.7.x, however, using `formatC` may be slightly better.

`list.len` numeric; maximum number of list elements to display within a level.

`deparse.lines` numeric or `NULL` as by default, determining the `nlines` argument to `deparse()` when `object` is a `call`. When `NULL`, the `nchar.max` and `width` arguments are used to determine a smart default.

`...` potential further arguments (required for Method/Generic reasons).

Value

`str` does not return anything, for efficiency reasons. The obvious side effect is output to the terminal.

Note

See the extensive annotated 'Examples' below.

The default method tries to "work always", but needs to make some assumptions for the case when `object` has a `class` but no own `str()` method which is *the typical* case: There it relies on `"["` and `"[["` subsetting methods to be compatible with `length()`. When this is not the case, or when `is.list(object)` is `TRUE`, but `length(object)` differs from `length(unclass(object))` it treats it as "irregular" and reports the contents of `unclass(object)` as "hidden list".

Author(s)

Martin Maechler <maechler@stat.math.ethz.ch> since 1990.

See Also

`ls.str` for *listing* objects with their structure; `summary`, `args`.

Examples

```
require(stats); require(grDevices); require(graphics)
## The following examples show some of 'str' capabilities
str(1:12)
str(ls)
str(args) #- more useful than  args(args) !
str(freeny)
str(str)
str(.Machine, digits.d = 20) # extra digits for identification of binary numbers
str( lsfit(1:9, 1:9))
str( lsfit(1:9, 1:9), max.level = 1)
str( lsfit(1:9, 1:9), width = 60, strict.width = "cut")
str( lsfit(1:9, 1:9), width = 60, strict.width = "wrap")
op <- options(); str(op)   # save first;
                           # otherwise internal options() is used.
need.dev <-
  !exists(".Device") || is.null(.Device) || .Device == "null device"
{ if(need.dev) pdf()
  str(par())
  if(need.dev) graphics.off()
}
ch <- letters[1:12]; is.na(ch) <- 3:5
str(ch) # character NA's

str(list(a = "A", L = as.list(1:100)), list.len = 9)
##                                       ------------
## " .. [list output truncated] "

## Long strings,   'nchar.max'; 'strict.width' :
nchar(longch <- paste(rep(letters,100), collapse = ""))
str(longch)
str(longch, nchar.max = 52)
str(longch, strict.width = "wrap")
```

```
## Multibyte characters in strings:
## Truncation behavior (<-> correct width measurement) for "long" non-ASCII:
idx <- c(65313:65338, 65345:65350)
fwch <- intToUtf8(idx) # full width character string: each has width 2
ch <- strtrim(paste(LETTERS, collapse="._"), 64)
(ncc <- c(c.ch = nchar(ch),   w.ch = nchar(ch,   "w"),
          c.fw = nchar(fwch), w.fw = nchar(fwch, "w")))
stopifnot(unname(ncc) == c(64,64, 32, 64))
## nchar.max: 1st line needs an increase of  2  in order to see  1  (in UTF-8!):
invisible(lapply(60:66, function(N) str(fwch, nchar.max = N)))
invisible(lapply(60:66, function(N) str( ch , nchar.max = N))) # "1 is 1" here


## Settings for narrow transcript :
op <- options(width = 60,
              str = strOptions(strict.width = "wrap"))
str(lsfit(1:9,1:9))
str(options())
## reset to previous:
options(op)



str(quote( { A+B; list(C, D) } ))


## S4 classes :
require(stats4)
x <- 0:10; y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax = 15, xh = 6)
      -sum(dpois(y, lambda=ymax/(1+x/xh), log=TRUE))
fit <- mle(ll)
str(fit)
```

`strcapture` *Capture String Tokens into a data.frame*

Description

Given a character vector and a regular expression containing capture expressions, `strcapture` will extract the captured tokens into a tabular data structure, such as a data.frame, the type and structure of which is specified by a prototype object. The assumption is that the same number of tokens are captured from every input string.

Usage

```
strcapture(pattern, x, proto, perl = FALSE, useBytes = FALSE)
```

Arguments

pattern	The regular expression with the capture expressions.
x	A character vector in which to capture the tokens.
proto	A data.frame or S4 object that behaves like one. See details.
perl, useBytes	Arguments passed to regexec.

Details

The proto argument is typically a data.frame, with a column corresponding to each capture expression, in order. The captured character vector is coerced to the type of the column, and the column names are carried over to the return value. Any data in the prototype are ignored. See the examples.

Value

A tabular data structure of the same type as proto, so typically a data.frame, containing a column for each capture expression. The column types and names are inherited from proto. Cases in x that do not match pattern have NA in every column.

See Also

regexec and regmatches for related low-level utilities.

Examples

```
x <- "chr1:1-1000"
pattern <- "(.*?):([[:digit:]]+)-([[:digit:]]+)"
proto <- data.frame(chr=character(), start=integer(), end=integer())
strcapture(pattern, x, proto)
```

---

summaryRprof *Summarise Output of R Sampling Profiler*

Description

Summarise the output of the Rprof function to show the amount of time used by different R functions.

Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000,
             memory = c("none", "both", "tseries", "stats"),
             lines = c("hide", "show", "both"),
             index = 2, diff = TRUE, exclude = NULL,
             basenames = 1)
```

Arguments

filename	Name of a file produced by Rprof().
chunksize	Number of lines to read at a time.
memory	Summaries for memory information. See 'Memory profiling' below. Can be abbreviated.
lines	Summaries for line information. See 'Line profiling' below. Can be abbreviated.
index	How to summarize the stack trace for memory information. See 'Details' below.
diff	If TRUE memory summaries use change in memory rather than current memory.
exclude	Functions to exclude when summarizing the stack trace for memory summaries.
basenames	Number of components of the path to filenames to display.

Details

This function provides the analysis code for Rprof files used by R CMD Rprof.

As the profiling output file could be larger than available memory, it is read in blocks of chunksize lines. Increasing chunksize will make the function run faster if sufficient memory is available.

Value

If memory = "none" and lines = "hide", a list with components

by.self	A data frame of timings sorted by 'self' time.
by.total	A data frame of timings sorted by 'total' time.
sample.interval	The sampling interval.
sampling.time	Total time of profiling run.

The first two components have columns 'self.time', 'self.pct', 'total.time' and 'total.pct', the times in seconds and percentages of the total time spent executing code in that function and code in that function or called from that function, respectively.

If lines = "show", an additional component is added to the list:

by.line	A data frame of timings sorted by source location.

If memory = "both" the same list but with memory consumption in Mb in addition to the timings.

If memory = "tseries" a data frame giving memory statistics over time. Memory usage is in bytes.

If memory = "stats" a by object giving memory statistics by function. Memory usage is in bytes.

If no events were recorded, a zero-row data frame is returned.

Memory profiling

Options other than `memory = "none"` apply only to files produced by `Rprof(memory.profiling = TRUE)`.

When called with `memory.profiling = TRUE`, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from `malloc`), memory in nodes on the R heap. It also records the number of calls to the internal function `duplicate` in the time interval. `duplicate` is called by C code when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.

With `memory = "both"` the change in total memory (truncated at zero) is reported in addition to timing data.

With `memory = "tseries"` or `memory = "stats"` the `index` argument specifies how to summarize the stack trace. A positive number specifies that many calls from the bottom of the stack; a negative number specifies the number of calls from the top of the stack. With `memory = "tseries"` the index is used to construct labels and may be a vector to give multiple sets of labels. With `memory = "stats"` the index must be a single number and specifies how to aggregate the data to the maximum and average of the memory statistics. With both `memory = "tseries"` and `memory = "stats"` the argument `diff = TRUE` asks for summaries of the increase in memory use over the sampling interval and `diff = FALSE` asks for the memory use at the end of the interval.

Line profiling

If the code being run has source reference information retained (via `keep.source = TRUE` in `source` or `KeepSource = TRUE` in a package 'DESCRIPTION' file or some other way), then information about the origin of lines is recorded during profiling. By default this is not displayed, but the `lines` parameter can enable the display.

If `lines = "show"`, line locations will be used in preference to the usual function name information, and the results will be displayed ordered by location in addition to the other orderings.

If `lines = "both"`, line locations will be mixed with function names in a combined display.

See Also

The chapter on 'Tidying and profiling R code' in 'Writing R Extensions': `RShowDoc("R-exts")`.

`Rprof`

`tracemem` traces copying of an object via the C function `duplicate`.

`Rprofmem` is a non-sampling memory-use profiler.

https://developer.r-project.org/memory-profiling.html

Examples

```
## Not run:
## Rprof() is not available on all platforms
Rprof(tmp <- tempfile())
example(glm)
Rprof()
summaryRprof(tmp)
unlink(tmp)
```

```
## End(Not run)
```

`Sweave` *Automatic Generation of Reports*

Description

`Sweave` provides a flexible framework for mixing text and R/S code for automatic report generation. The basic idea is to replace the code with its output, such that the final document only contains the text and the output of the statistical analysis: however, the source code can also be included.

Usage

```
Sweave(file, driver = RweaveLatex(),
       syntax = getOption("SweaveSyntax"), encoding = "", ...)

Stangle(file, driver = Rtangle(),
        syntax = getOption("SweaveSyntax"), encoding = "", ...)
```

Arguments

`file`	Path to Sweave source file. Note that this can be supplied without the extension, but the function will only proceed if there is exactly one Sweave file in the directory whose basename matches `file`.
`driver`	the actual workhorse, (a function returning) a named `list` of five functions; for details, see Section 5 of the 'Sweave User Manual' available as `vignette("Sweave")`.
`syntax`	`NULL` or an object of class `"SweaveSyntax"` or a character string with its name. See the section 'Syntax Definition'.
`encoding`	The default encoding to assume for `file`.
`...`	further arguments passed to the driver's setup function. See `RweaveLatexSetup` and `RtangleSetup`, respectively, for the arguments of the default drivers.

Details

An Sweave source file contains both text in a markup language (like LaTeX) and R (or S) code. The code gets replaced by its output (text or graphs) in the final markup file. This allows a report to be re-generated if the input data change and documents the code to reproduce the analysis in the same file that also produces the report.

`Sweave` combines the documentation and code chunks (or their output) into a single document. `Stangle` extracts only the code from the Sweave file creating an R source file that can be run using `source`. (Code inside `\Sexpr{}` statements is ignored by `Stangle`.)

`Stangle` is just a wrapper to `Sweave` specifying a different default driver. Alternative drivers can be used and are provided by various contributed packages.

Environment variable `SWEAVE_OPTIONS` can be used to override the initial options set by the driver: it should be a comma-separated set of `key=value` items, as would be used in a '`\SweaveOpts`' statement in a document.

If the `encoding` is unspecified (the default), *non-ASCII* source files must contain a line of the form

```
\usepackage[foo]{inputenc}
```

(where '`foo`' is typically '`latin1`', '`latin2`', '`utf8`' or '`cp1252`' or '`cp1250`') or a comment line

```
%\SweaveUTF8
```

to declare UTF-8 input (the default encoding assumed by pdfTeX since 2018), or they will give an error. Re-encoding can be turned off completely with argument `encoding = "bytes"`.

Syntax Definition

Sweave allows a flexible syntax framework for marking documentation and text chunks. The default is a noweb-style syntax, as alternative a LaTeX-style syntax can be used. (See the user manual for further details.)

If `syntax = NULL` (the default) then the available syntax objects are consulted in turn, and selected if their `extension` component matches (as a regexp) the file name. Objects `SweaveSyntaxNoweb` (with `extension = "[.][rsRS]nw$"`) and `SweaveSyntaxLatex` (with `extension = "[.][rsRS]tex$"`) are supplied, but users or packages can supply others with names matching the pattern `SweaveSyntax.*`.

Author(s)

Friedrich Leisch and R-core.

References

Friedrich Leisch (2002) Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, *Compstat 2002 - Proceedings in Computational Statistics*, pages 575–580. Physika Verlag, Heidelberg, Germany, ISBN 3-7908-1517-9.

See Also

'Sweave User Manual', a vignette in the **utils** package.

`RweaveLatex`, `Rtangle`. Alternative Sweave drivers are in, for example, packages **weaver** (Bioconductor), **R2HTML**, and **ascii**.

`tools::buildVignette` to process source files using Sweave or alternative vignette processing engines.

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")


## enforce par(ask = FALSE)
options(device.ask.default = FALSE)

## create a LaTeX file - in the current working directory, getwd():
Sweave(testfile)

## This can be compiled to PDF by
## tools::texi2pdf("Sweave-test-1.tex")

## or outside R by
##
##  R CMD texi2pdf Sweave-test-1.tex
## on Unix-alikes which sets the appropriate TEXINPUTS path.
##
## On Windows,
##      Rcmd texify --pdf Sweave-test-1.tex
## if MiKTeX is available.

## create an R source file from the code chunks
Stangle(testfile)
## which can be sourced, e.g.
source("Sweave-test-1.R")
```

SweaveSyntConv *Convert Sweave Syntax*

Description

This function converts the syntax of files in Sweave format to another Sweave syntax definition.

Usage

```
SweaveSyntConv(file, syntax, output = NULL)
```

Arguments

file	Name of Sweave source file.
syntax	An object of class SweaveSyntax or a character string with its name giving the target syntax to which the file is converted.
output	Name of output file, default is to remove the extension from the input file and to add the default extension of the target syntax. Any directory names in file are also removed such that the output is created in the current working directory.

Author(s)

Friedrich Leisch

See Also

'Sweave User Manual', a vignette in the **utils** package.

RweaveLatex, Rtangle

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")


## convert the file to latex syntax
SweaveSyntConv(testfile, SweaveSyntaxLatex)

## and run it through Sweave
Sweave("Sweave-test-1.Stex")
```

---

tar *Create a Tar Archive*

Description

Create a tar archive.

Usage

```
tar(tarfile, files = NULL,
    compression = c("none", "gzip", "bzip2", "xz"),
    compression_level = 6, tar = Sys.getenv("tar"),
    extra_flags = "")
```

Arguments

tarfile	The pathname of the tar file: tilde expansion (see path.expand) will be performed. Alternatively, a connection that can be used for binary writes.
files	A character vector of filepaths to be archived: the default is to archive all files under the current directory.
compression	character string giving the type of compression to be used (default none). Can be abbreviated.
compression_level	integer: the level of compression. Only used for the internal method.
tar	character string: the path to the command to be used. If the command itself contains spaces it needs to be quoted (e.g., by shQuote) – but argument tar may also contain flags separated from the command by spaces.
extra_flags	Any extra flags for an external tar.

Details

This is either a wrapper for a tar command or uses an internal implementation in R. The latter is used if tarfile is a connection or if the argument tar is "internal" or "" (the 'factory-fresh' default). Note that whereas Unix-alike versions of R set the environment variable TAR, its value is not the default for this function.

Argument extra_flags is passed to an external tar and so is platform-dependent. Possibly useful values include '-h' (follow symbolic links, also '-L' on some platforms), '--acls', '--exclude-backups', '--exclude-vcs' (and similar) and on Windows '--force-local' (so drives can be included in filepaths). Rtools 4 and earlier included a tar which used '--force-local', but Rtools 4.2 includes original GNU tar, which does not use it by default.

A convenient and robust way to set options for GNU tar is via environment variable TAR_OPTIONS. Appending '--force-local' to TAR does not work with GNU tar due to restrictions on how some options can be mixed. The tar available on Windows 10 (libarchive's bsdtar) supports drive letters by default. It does not support the '--force-local', but ignores TAR_OPTIONS.

For GNU tar, '--format=ustar' forces a more portable format. (The default is set at compilation and will be shown at the end of the output from tar --help: for version 1.30 'out-of-the-box' it is '--format=gnu', but the manual says the intention is to change to '--format=posix' which is the same as pax – it was never part of the POSIX standard for tar and should not be used.) For libarchive's bsdtar, '--format=ustar' is more portable than the default.

One issue which can cause an external command to fail is a command line too long for the system shell: as from R 3.5.0 this is worked around if the external command is detected to be GNU tar or libarchive tar (aka bsdtar).

Note that files = '.' will usually not work with an external tar as that would expand the list of files after tarfile is created. (It does work with the default internal method.)

Value

The return code from system or 0 for the internal version, invisibly.

Portability

The 'tar' format no longer has an agreed standard! 'Unix Standard Tar' was part of POSIX 1003.1:1998 but has been removed in favour of pax, and in any case many common implementations diverged from the former standard.

Many R platforms use a version of GNU tar, but the behaviour seems to be changed with each version. macOS >= 10.6, FreeBSD and Windows 10 use bsdtar from the libarchive project (but for macOS often a quite-old version), and commercial Unixes will have their own versions. bsdtar is available for many other platforms: macOS up to at least 10.9 had GNU tar as gnutar and other platforms, e.g. Solaris, have it as gtar: on a Unix-alike configure will try gnutar and gtar before tar.

Known problems arise from

- The handling of file paths of more than 100 bytes. These were unsupported in early versions of tar, and supported in one way by POSIX tar and in another by GNU tar and yet another by the POSIX pax command which recent tar programs often support. The internal implementation warns on paths of more than 100 bytes, uses the 'ustar' way from the 1998 POSIX

standard which supports up to 256 bytes (depending on the path: in particular the final component is limited to 100 bytes) if possible, otherwise the GNU way (which is widely supported, including by untar).

Most formats do not record the encoding of file paths.

- (File) links. tar was developed on an OS that used hard links, and physical files that were referred to more than once in the list of files to be included were included only once, the remaining instances being added as links. Later a means to include symbolic links was added. The internal implementation supports symbolic links (on OSes that support them), only. Of course, the question arises as to how links should be unpacked on OSes that do not support them: for regular files file copies can be used.

  Names of links in the 'ustar' format are restricted to 100 bytes. There is an GNU extension for arbitrarily long link names, but bsdtar ignores it. The internal method uses the GNU extension, with a warning.
- Header fields, in particular the padding to be used when fields are not full or not used. POSIX did define the correct behaviour but commonly used implementations did (and still do) not comply.
- File sizes. The 'ustar' format is restricted to 8GB per (uncompressed) file.

For portability, avoid file paths of more than 100 bytes and all links (especially hard links and symbolic links to directories).

The internal implementation writes only the blocks of 512 bytes required (including trailing blocks of NULs), unlike GNU tar which by default pads with 'nul' to a multiple of 20 blocks (10KB). Implementations which pad differ on whether the block padding should occur before or after compression (or both): padding was designed for improved performance on physical tape drives.

The 'ustar' format records file modification times to a resolution of 1 second: on file systems with higher resolution it is conventional to discard fractional seconds.

Compression

When an external tar command is used, compressing the tar archive requires that tar supports the '-z', '-j' or '-J' flag, and may require the appropriate command (gzip, bzip2 or xz) to be available. For GNU tar, further compression programs can be specified by e.g. extra_flags = "-I lz4". Some versions of bsdtar accept options such as '--lz4', '--lzop' and '--lrzip' or an external compressor *via* '--use-compress-program lz4': these could be supplied in extra_flags.

NetBSD prior to 8.0 used flag '--xz' rather than '-J', so this should be used *via* extra_flags = "--xz" rather than compression = "xz". The commands from OpenBSD and the Heirloom Toolchest are not documented to support xz.

The tar programs in commercial Unixen such as AIX and Solaris do not support compression.

Note

For users of macOS. Apple's file systems have a legacy concept of 'resource forks' dating from classic Mac OS and rarely used nowadays. Apple's version of tar stores these as separate files in the tarball with names prefixed by '._', and unpacks such files into resource forks (if possible): other ways of unpacking (including untar in R) unpack them as separate files.

When argument tar is set to the command tar on macOS, environment variable COPYFILE_DISABLE=1 is set, which for the system version of tar prevents these separate files being included in the tarball.

See Also

https://en.wikipedia.org/wiki/Tar_(file_format), https://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html#tag_20_92_13_06 for the way the POSIX utility pax handles tar formats.

https://github.com/libarchive/libarchive/wiki/FormatTar.

untar.

---

toLatex *Converting R Objects to BibTeX or LaTeX*

Description

These methods convert R objects to character vectors with BibTeX or LaTeX markup.

Usage

```
toBibtex(object, ...)
toLatex(object, ...)
## S3 method for class 'Bibtex'
print(x, prefix = "", ...)
## S3 method for class 'Latex'
print(x, prefix = "", ...)
```

Arguments

object	object of a class for which a toBibtex or toLatex method exists.
x	object of class "Bibtex" or "Latex".
prefix	a character string which is printed at the beginning of each line, mostly used to insert whitespace for indentation.
...	in the print methods, passed to writeLines.

Details

Objects of class "Bibtex" or "Latex" are simply character vectors where each element holds one line of the corresponding BibTeX or LaTeX file.

See Also

citEntry and sessionInfo for examples

`txtProgressBar` *Text Progress Bar*

Description

Text progress bar in the R console.

Usage

```
txtProgressBar(min = 0, max = 1, initial = 0, char = "=",
               width = NA, title, label, style = 1, file = "")

getTxtProgressBar(pb)
setTxtProgressBar(pb, value, title = NULL, label = NULL)
## S3 method for class 'txtProgressBar'
close(con, ...)
```

Arguments

`min`, `max`	(finite) numeric values for the extremes of the progress bar. Must have `min` < `max`.
`initial`, `value`	initial or new value for the progress bar. See 'Details' for what happens with invalid values.
`char`	the character (or character string) to form the progress bar. Must have non-zero display width.
`width`	the width of the progress bar, as a multiple of the width of `char`. If `NA`, the default, the number of characters is that which fits into `getOption("width")`.
`style`	the 'style' of the bar – see 'Details'.
`file`	an open connection object or `""` which indicates the console: `stderr()` might be useful here.
`pb`, `con`	an object of class `"txtProgressBar"`.
`title`, `label`	ignored, for compatibility with other progress bars.
`...`	for consistency with the generic.

Details

`txtProgressBar` will display a progress bar on the R console (or a connection) via a text representation.

`setTxtProgessBar` will update the value. Missing (`NA`) and out-of-range values of `value` will be (silently) ignored. (Such values of `initial` cause the progress bar not to be displayed until a valid value is set.)

The progress bar should be `close`d when finished with: this outputs the final newline character.

`style = 1` and `style = 2` just shows a line of `char`. They differ in that `style = 2` redraws the line each time, which is useful if other code might be writing to the R console. `style = 3` marks the end of the range by '`|`' and gives a percentage to the right of the bar.

Value

For `txtProgressBar` an object of class `"txtProgressBar"`.

For `getTxtProgressBar` and `setTxtProgressBar`, a length-one numeric vector giving the previous value (invisibly for `setTxtProgressBar`).

Note

Using `style` 2 or 3 or reducing the value with `style` = 1 uses '`\r`' to return to the left margin – the interpretation of carriage return is up to the terminal or console in which R is running, and this is liable to produce ugly output on a connection other than a terminal, including when `stdout()` is redirected to a file.

See Also

`winProgressBar` (Windows only), `tkProgressBar` (Unix-alike platforms).

Examples

```
 # slow
testit <- function(x = sort(runif(20)), ...)
{
    pb <- txtProgressBar(...)
    for(i in c(0, x, 1)) {Sys.sleep(0.5); setTxtProgressBar(pb, i)}
    Sys.sleep(1)
    close(pb)
}
testit()
testit(runif(10))
testit(style = 3)
testit(char=' \u27a4')
```

---

type.convert *Convert Data to Appropriate Type*

Description

Convert a data object to logical, integer, numeric, complex, character or factor as appropriate.

Usage

```
type.convert(x, ...)
## Default S3 method:
type.convert(x, na.strings = "NA", as.is, dec = ".",
             numerals = c("allow.loss", "warn.loss", "no.loss"),
             tryLogical = TRUE, ...)
## S3 method for class 'data.frame'
type.convert(x, ...)
## S3 method for class 'list'
type.convert(x, ...)
```

Arguments

x	a vector, matrix, array, data frame, or list.
na.strings	a vector of strings which are to be interpreted as NA values. Blank fields are also considered to be missing values in logical, integer, numeric or complex vectors.
as.is	whether to store strings as plain character. When false, convert character vectors to factors. See 'Details'.
dec	the character to be assumed for decimal points.
numerals	string indicating how to convert numbers whose conversion to double precision would lose accuracy, typically when x has more digits than can be stored in a double. Can be abbreviated. Possible values are numerals = "allow.loss", **default:** the conversion happens with some accuracy loss. numerals = "warn.loss": a warning about accuracy loss is signalled and the conversion happens as with numerals = "allow.loss". numerals = "no.loss": x is *not* converted to a number, but to a factor or character, depending on as.is.
tryLogical	a logical determining if vectors consisting entirely of F, T, FALSE, TRUE and na.strings should be converted to logical; true, by default.
...	arguments to be passed to or from methods.

Details

This helper function is used by read.table. When the data object x is a data frame or list, the function is called recursively for each column or list element.

Given a vector, the function attempts to convert it to logical, integer, numeric or complex, and when additionally as.is = FALSE (no longer the default!), converts a character vector to factor. The first type that can accept all the non-missing values is chosen.

Vectors which are entirely missing values are converted to logical, since NA is primarily logical.

If tryLogical is true as by default, vectors containing just F, T, FALSE, TRUE and values from na.strings are converted to logical. This may be surprising in a context where you have many character columns with e.g., 1-letter strings and you happen to get one with only "F". In such cases tryLogical = FALSE is recommended. Vectors containing optional whitespace followed by decimal constants representable as R integers or values from na.strings are converted to integer. Other vectors containing optional whitespace followed by other decimal or hexadecimal constants (see NumericConstants), or NaN, Inf or infinity (ignoring case) or values from na.strings are converted to numeric. Where converting inputs to numeric or complex would result in loss of accuracy they can optionally be returned as strings or (for as.is = FALSE) factors.

Since this is a helper function, the caller should always pass an appropriate value of as.is.

Value

An object like x but using another storage mode when appropriate.

Author(s)

R Core, with a contribution by Arni Magnusson

See Also

read.table, class, storage.mode.

Examples

```
## Numeric to integer
class(rivers)
x <- type.convert(rivers, as.is = TRUE)
class(x)

## Convert many columns
auto <- type.convert(mtcars, as.is = TRUE)
str(mtcars)
str(auto)

## Convert matrix
phones <- type.convert(WorldPhones, as.is = TRUE)
storage.mode(WorldPhones)
storage.mode(phones)

## Factor or character
chr <- c("A", "B", "B", "A")
ch2 <- c("F", "F", "NA", "F")
(fac <- factor(chr))
type.convert(chr, as.is = FALSE) # -> factor
type.convert(fac, as.is = FALSE) # -> factor
type.convert(chr, as.is = TRUE)  # -> character
type.convert(fac, as.is = TRUE)  # -> character
type.convert(ch2, as.is = TRUE)                   #-> logical
type.convert(ch2, as.is = TRUE, tryLogical=FALSE) #-> character
```

---

untar *Extract or List Tar Archives*

Description

Extract files from or list the contents of a tar archive.

Usage

```
untar(tarfile, files = NULL, list = FALSE, exdir = ".",
      compressed = NA, extras = NULL, verbose = FALSE,
      restore_times =  TRUE,
      support_old_tars = Sys.getenv("R_SUPPORT_OLD_TARS", FALSE),
      tar = Sys.getenv("TAR"))
```

Arguments

tarfile	The pathname of the tar file: tilde expansion (see path.expand) will be performed. Alternatively, a connection that can be used for binary reads. For a *compressed* tarfile, and if a connection is to be used, that should be created by gzfile(.) (or gzcon(.) which currently only works for "gzip", whereas gzfile() works for all compressions available in tar()).
files	A character vector of recorded filepaths to be extracted: the default is to extract all files.
list	If TRUE, list the files (the equivalent of tar -tf). Otherwise extract the files (the equivalent of tar -xf).
exdir	The directory to extract files to (the equivalent of tar -C). It will be created if necessary.
compressed	(Deprecated in favour of auto-detection, used only for an external tar command.) Logical or character string. Values "gzip", "bzip2" and "xz" select that form of compression (and may be abbreviated to the first letter). TRUE indicates gzip compression, FALSE no known compression, and NA (the default) indicates that the type is to be inferred from the file header. The external command may ignore the selected compression type but detect a type automagically.
extras	NULL or a character string: further command-line flags such as '-p' to be passed to an external tar program.
verbose	logical: if true echo the command used for an external tar program.
restore_times	logical. If true (default) restore file modification times. If false, the equivalent of the '-m' flag. Times in tarballs are supposed to be in UTC, but tarballs have been submitted to CRAN with times in the future or far past: this argument allows such times to be discarded. Note that file times in a tarball are stored with a resolution of 1 second, and can only be restored to the resolution supported by the file system (which on a FAT system is 2 seconds).
support_old_tars	logical. If false (the default), the external tar command is assumed to be able handle compressed tarfiles and if compressed does not specify it, to automagically detect the type of compression. (The major implementations have done so since 2009; for GNU tar since version 1.22.) If true, the R code calls an appropriate decompressor and pipes the output to tar, for compressed = NA examining the tarfile header to determine the type of compression.
tar	character string: the path to the command to be used or "internal". If the command itself contains spaces it needs to be quoted – but tar can also contain flags separated from the command by spaces.

Details

This is either a wrapper for a tar command or for an internal implementation written in R. The latter is used if tarfile is a connection or if the argument tar is "internal" or "" (except on Windows, when tar.exe is tried first).

Unless otherwise stated three types of compression of the tar file are supported: `gzip`, `bzip2` and `xz`.

What options are supported will depend on the `tar` implementation used: the `"internal"` one is intended to provide support for most in a platform-independent way.

GNU tar: Modern GNU `tar` versions support compressed archives and since 1.15 are able to detect the type of compression automatically: version 1.22 added support for `xz` compression.

On a Unix-alike, `configure` will set environment variable `TAR`, preferring GNU tar if found.

`bsdtar`: macOS 10.6 and later (and FreeBSD and some other OSes) have a `tar` from the libarchive project which detects all three forms of compression automagically (even if undocumented in macOS).

NetBSD: It is undocumented if NetBSD's `tar` can detect compression automagically: for versions before 8 the flag for `xz` compression was '`--xz`' not '`-J`'. So `support_old_tars = TRUE` is recommended (or use `bsdtar` if installed).

OpenBSD: OpenBSD's `tar` does not detect compression automagically. It has no support for `xz` beyond reporting that the file is `xz`-compressed. So `support_old_tars = TRUE` is recommended.

Heirloom Toolchest: This `tar` does automagically detect `gzip` and `bzip2` compression (undocumented) but has no support for `xz` compression.

Older support: Environment variable `R_GZIPCMD` gives the command to decompress `gzip` files, and `R_BZIPCMD` for `bzip2` files. (On Unix-alikes these are set at installation if found.) `xz` is used if available: if not decompression is expected to fail.

Arguments `compressed`, `extras` and `verbose` are only used when an external `tar` is used.

Some external `tar` commands will detect some of `lrzip`, `lzma`, `lz4`, `lzop` and `zstd` compression in addition to `gzip`, `bzip2` and `xz`. (For some external `tar` commands, compressed tarfiles can only be read if the appropriate utility program is available.) For GNU `tar`, further (de)compression programs can be specified by e.g. `extras = "-I lz4"`. For `bsdtar` this could be `extras = "--use-compress-program lz4"`. Most commands will detect (the nowadays rarely seen) '`.tar.Z`' archives compressed by `compress`.

The internal implementation restores symbolic links as links on a Unix-alike, and as file copies on Windows (which works only for existing files, not for directories), and hard links as links. If the linking operation fails (as it may on a FAT file system), a file copy is tried. Since it uses `gzfile` to read a file it can handle files compressed by any of the methods that function can handle: at least `compress`, `gzip`, `bzip2` and `xz` compression, and some types of `lzma` compression. It does not guard against restoring absolute file paths, as some `tar` implementations do. It will create the parent directories for directories or files in the archive if necessary. It handles the USTAR/POSIX, GNU and `pax` ways of handling file paths of more than 100 bytes, and the GNU way of handling link targets of more than 100 bytes.

You may see warnings from the internal implementation such as

```
unsupported entry type 'x'
```

This often indicates an invalid archive: entry types `"A-Z"` are allowed as extensions, but other types are reserved. The only thing you can do with such an archive is to find a `tar` program that handles it, and look carefully at the resulting files. There may also be the warning

```
using pax extended headers
```

This indicates that additional information may have been discarded, such as ACLs, encodings ....

The former standards only supported ASCII filenames (indeed, only alphanumeric plus period, underscore and hyphen). `untar` makes no attempt to map filenames to those acceptable on the current system, and treats the filenames in the archive as applicable without any re-encoding in the current locale.

The internal implementation does not special-case 'resource forks' in macOS: that system's `tar` command does. This may lead to unexpected files with names with prefix '`._`'.

Value

If `list = TRUE`, a character vector of (relative or absolute) paths of files contained in the tar archive.

Otherwise the return code from `system` with an external `tar` or `0L`, invisibly.

See Also

`tar`, `unzip`.

`unzip` *Extract or List Zip Archives*

Description

Extract files from or list a zip archive.

Usage

```
unzip(zipfile, files = NULL, list = FALSE, overwrite = TRUE,
      junkpaths = FALSE, exdir = ".", unzip = "internal",
      setTimes = FALSE)
```

Arguments

`zipfile`	The pathname of the zip file: tilde expansion (see `path.expand`) will be performed.
`files`	A character vector of recorded filepaths to be extracted: the default is to extract all files.
`list`	If `TRUE`, list the files and extract none. The equivalent of `unzip -l`.
`overwrite`	If `TRUE`, overwrite existing files (the equivalent of `unzip -o`), otherwise ignore such files (the equivalent of `unzip -n`).
`junkpaths`	If `TRUE`, use only the basename of the stored filepath when extracting. The equivalent of `unzip -j`.
`exdir`	The directory to extract files to (the equivalent of `unzip -d`). It will be created if necessary.

unzip	The method to be used. An alternative is to use getOption("unzip"), which on a Unix-alike may be set to the path to a unzip program.
setTimes	logical. For the internal method only, should the file times be set based on the times in the zip file? (NB: this applies to included files, not to directories.)

Value

If list = TRUE, a data frame with columns Name (character) Length (the size of the uncompressed file, numeric) and Date (of class "POSIXct").

Otherwise for the "internal" method, a character vector of the filepaths extracted to, invisibly.

Note

The default internal method is a minimal implementation, principally designed for Windows' users to be able to unpack Windows binary packages without external software. It does not (for example) support Unicode filenames as introduced in zip 3.0: for that use unzip = "unzip" with unzip 6.00 or later. It does have some support for bzip2 compression and > 2GB zip files (but not >= 4GB files pre-compression contained in a zip file: like many builds of unzip it may truncate these, in R's case with a warning if possible).

If unzip specifies a program, the format of the dates listed with list = TRUE is unknown (on Windows it can even depend on the current locale) and the return values could be NA or expressed in the wrong time zone or misinterpreted (the latter being far less likely as from unzip 6.00).

File times in zip files are stored in the style of MS-DOS, as local times to an accuracy of 2 seconds. This is not very useful when transferring zip files between machines (even across continents), so we chose not to restore them by default.

Source

The internal C code uses zlib and is in particular based on the contributed 'minizip' application in the zlib sources (from https://zlib.net/) by Gilles Vollant.

See Also

unz to read a single component from a zip file.

zip for packing, i.e., the "inverse" of unzip(); further untar and tar, the corresponding pair for (un)packing tar archives ("tarballs") such as R source packages.

---

update.packages *Compare Installed Packages with CRAN-like Repositories*

Description

old.packages indicates packages which have a (suitable) later version on the repositories whereas update.packages offers to download and install such packages.

new.packages looks for (suitable) packages on the repositories that are not already installed, and optionally offers them for installation.

Usage

```
update.packages(lib.loc = NULL, repos = getOption("repos"),
                contriburl = contrib.url(repos, type),
                method, instlib = NULL,
                ask = TRUE, available = NULL,
                oldPkgs = NULL, ..., checkBuilt = FALSE,
                type = getOption("pkgType"))

old.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos, type),
             instPkgs = installed.packages(lib.loc = lib.loc, ...),
             method, available = NULL, checkBuilt = FALSE, ...,
             type = getOption("pkgType"))

new.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos, type),
             instPkgs = installed.packages(lib.loc = lib.loc, ...),
             method, available = NULL, ask = FALSE, ...,
             type = getOption("pkgType"))
```

Arguments

lib.loc	character vector describing the location of R library trees to search through (and update packages therein), or NULL for all known trees (see .libPaths).
repos	character vector, the base URL(s) of the repositories to use, e.g., the URL of a CRAN mirror such as "https://cloud.r-project.org".
contriburl	URL(s) of the contrib sections of the repositories. Use this argument if your repository is incomplete. Overrides argument repos. Incompatible with type = "both".
method	Download method, see download.file. Unused if a non-NULL available is supplied.
instlib	character string giving the library directory where to install the packages.
ask	logical indicating whether to ask the user to select packages before they are downloaded and installed, or the character string "graphics", which brings up a widget to allow the user to (de)select from the list of packages which could be updated. The latter value only works on systems with a GUI version of select.list, and is otherwise equivalent to ask = TRUE. ask does not control questions asked before installing packages from source via type = "both" (see option "install.packages.compile.from.source").
available	an object as returned by available.packages listing packages available at the repositories, or NULL which makes an internal call to available.packages. Incompatible with type = "both".
checkBuilt	If TRUE, a package built under an earlier major.minor version of R (e.g., 3.4) is considered to be 'old'.
oldPkgs	if specified as non-NULL, update.packages() only considers these packages for updating. This may be a character vector of package names or a matrix as returned by old.packages.

instPkgs	by default all installed packages, installed.packages(lib.loc = lib.loc). A subset can be specified; currently this must be in the same (character matrix) format as returned by installed.packages().
...	Arguments such as destdir and dependencies to be passed to install.packages and ignore_repo_cache, max_repo_cache_age and noCache to available.packages or installed.packages.
type	character, indicating the type of package to download and install. See install.packages.

Details

old.packages compares the information from available.packages with that from instPkgs (computed by installed.packages by default) and reports installed packages that have newer versions on the repositories or, if checkBuilt = TRUE, that were built under an earlier minor version of R (for example built under 3.3.x when running R 3.4.0). (For binary package types there is no check that the version on the repository was built under the current minor version of R, but it is advertised as being suitable for this version.)

new.packages does the same comparison but reports uninstalled packages that are available at the repositories. If ask != FALSE it asks which packages should be installed in the first element of lib.loc.

The main function of the set is update.packages. First a list of all packages found in lib.loc is created and compared with those available at the repositories. If ask = TRUE (the default) packages with a newer version are reported and for each one the user can specify if it should be updated. If so the packages are downloaded from the repositories and installed in the respective library path (or instlib if specified).

For how the list of suitable available packages is determined see available.packages. available = NULL make a call to available.packages(contriburl = contriburl, method = method) and hence by default filters on R version, OS type and removes duplicates.

Value

update.packages returns NULL invisibly.

For old.packages, NULL or a matrix with one row per package, row names the package names and column names "Package", "LibPath", "Installed" (the version), "Built" (the version built under), "ReposVer" and "Repository".

For new.packages a character vector of package names, *after* any selected *via* ask have been installed.

Warning

Take care when using dependencies (passed to install.packages) with update.packages, for it is unclear where new dependencies should be installed. The current implementation will only allow it if all the packages to be updated are in a single library, when that library will be used.

See Also

install.packages, available.packages, download.packages, installed.packages, contrib.url.

The options listed for install.packages under options.

See download.file for how to handle proxies and other options to monitor file transfers.

INSTALL, REMOVE, remove.packages, library, .packages, read.dcf

The 'R Installation and Administration' manual for how to set up a repository.

---

upgrade *Upgrade*

Description

Upgrade objects.

Usage

```
upgrade(object, ...)
```

Arguments

object	an R object.
...	further arguments passed to or from other methods.

Details

This is a generic function, with a method for "packageStatus" objects.

---

url.show *Display a Text URL*

Description

Extension of file.show to display text files from a remote server.

Usage

```
url.show(url, title = url, file = tempfile(),
         delete.file = TRUE, method, ...)
```

Arguments

url	The URL to read from.
title	Title for the browser.
file	File to copy to.
delete.file	Delete the file afterwards?
method	File transfer method: see download.file
...	Arguments to pass to file.show.

Note

Since this is for text files, it will convert to CRLF line endings on Windows.

See Also

url, file.show, download.file

Examples

```
## Not run: url.show("https://www.stats.ox.ac.uk/pub/datasets/csb/ch3a.txt")
```

URLencode *Encode or Decode (partial) URLs*

Description

Functions to percent-encode or decode characters in URLs.

Usage

```
URLencode(URL, reserved = FALSE, repeated = FALSE)
URLdecode(URL)
```

Arguments

URL	a character vector.
reserved	logical: should 'reserved' characters be encoded? See 'Details'.
repeated	logical: should apparently already-encoded URLs be encoded again?

Details

Characters in a URL other than the English alphanumeric characters and '- _ . ~' should be encoded as % plus a two-digit hexadecimal representation, and any single-byte character can be so encoded. (Multi-byte characters are encoded byte-by-byte.) The standard refers to this as 'percent-encoding'.

In addition, '! $ & ' ( ) * + , ; = : / ? @ # [ ]' are reserved characters, and should be encoded unless used in their reserved sense, which is scheme specific. The default in URLencode is to leave them alone, which is appropriate for 'file://' URLs, but probably not for 'http://' ones.

An 'apparently already-encoded URL' is one containing %xx for two hexadecimal digits.

Value

A character vector.

References

Internet STD 66 (formerly RFC 3986), https://www.rfc-editor.org/info/std66

Examples

```
(y <- URLencode("a url with spaces and / and @"))
URLdecode(y)
(y <- URLencode("a url with spaces and / and @", reserved = TRUE))
URLdecode(y)

URLdecode(z <- "ab%20cd")
c(URLencode(z), URLencode(z, repeated = TRUE)) # first is usually wanted

## both functions support character vectors of length > 1
y <- URLdecode(URLencode(c("url with space", "another one")))
```

---

utils-deprecated *Deprecated Functions in Package* **utils**

Description

(Currently none)

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

See Also

Deprecated, Defunct

View *Invoke a Data Viewer*

Description

Invoke a spreadsheet-style data viewer on a matrix-like R object.

Usage

```
View(x, title)
```

Arguments

x	an R object which can be coerced to a data frame with non-zero numbers of rows and columns.
title	title for viewer window. Defaults to name of x prefixed by Data:.

Details

Object x is coerced (if possible) to a data frame, then columns are converted to character using format.data.frame. The object is then viewed in a spreadsheet-like data viewer, a read-only version of data.entry.

If there are row names on the data frame that are not 1:nrow, they are displayed in a separate first column called row.names.

Objects with zero columns or zero rows are not accepted.

On Unix-alikes, the array of cells can be navigated by the cursor keys and Home, End, Page Up and Page Down (where supported by X11) as well as Enter and Tab.

On Windows, the array of cells can be navigated *via* the scrollbars and by the cursor keys, Home, End, Page Up and Page Down.

On Windows, the initial size of the data viewer window is taken from the default dimensions of a pager (see Rconsole), but adjusted downwards to show a whole number of rows and columns.

Value

Invisible NULL. The functions puts up a window and returns immediately: the window can be closed via its controls or menus.

See Also

edit.data.frame, data.entry.

---

vignette *View, List or Get R Source of Package Vignettes*

Description

View a specified package vignette, or list the available ones; display it rendered in a viewer, and get or edit its R source file.

Usage

```
vignette(topic, package = NULL, lib.loc = NULL, all = TRUE)

## S3 method for class 'vignette'
print(x, ...)
## S3 method for class 'vignette'
edit(name, ...)
```

Arguments

topic	a character string giving the (base) name of the vignette to view. If omitted, all vignettes from all installed packages are listed.
package	a character vector with the names of packages to search through, or NULL in which 'all' packages (as defined by argument all) are searched.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known.
all	logical; if TRUE search all available packages in the library trees specified by lib.loc, and if FALSE, search only attached packages.
x, name	object of class vignette.
...	ignored by the print method, passed on to file.edit by the edit method.

Details

Function vignette returns an object of the same class, the print method opens a viewer for it.

On Unix-alikes, the program specified by the pdfviewer option is used for viewing PDF versions of vignettes.

If several vignettes have PDF/HTML versions with base name identical to topic, the first one found is used.

If no topics are given, all available vignettes are listed. The corresponding information is returned in an object of class "packageIQR".

See Also

browseVignettes for an HTML-based vignette browser; RShowDoc("*basename*", package = "*pkgname*") displays a "rendered" vignette (pdf or html).

Examples

```
## List vignettes from all *attached* packages
vignette(all = FALSE)

## List vignettes from all *installed* packages (can take a long time!):
vignette(all = TRUE)

## The grid intro vignette -- open it
## Not run: vignette("grid") # calling print()
## The same (conditional on existence of the vignettte).
## Note that 'package = *' is much faster in the case of many installed packages:
if(!is.null(v1 <- vignette("grid", package="grid"))) {
## Not run: v1 # calling print(.)
  str(v1)
  ## Now let us have a closer look at the code



## Not run: edit(v1) # e.g., to send lines ...
```

```
}# if( has vignette "installed")
## A package can have more than one vignette (package grid has several):
vignette(package = "grid")
if(interactive()) {
   ## vignette("rotated")
   ## The same, but without searching for it:
   vignette("rotated", package = "grid")
}
```

---

warnErrList *Collect and Summarize Errors From List*

Description

Collect errors (class "error", typically from tryCatch) from a list x into a "summary warning", by default produce a warning and keep that message as "warningMsg" attribute.

Usage

```
warnErrList(x, warn = TRUE, errValue = NULL)
```

Arguments

x	a list, typically from applying models to a list of data (sub)sets, e.g., using tryCatch(*, error = identity).
warn	logical indicating if warning() should be called.
errValue	the value with which errors should be replaced.

Value

a list of the same length and names as the x argument, with the error components replaced by errValue, NULL by default, and summarized in the "warningMsg" attribute.

See Also

The warnErrList() utility has been used in lmList() and nlsList() in recommended package **nlme** forever.

Examples

```
## Regression for each Chick:
ChWtgrps <- split(ChickWeight, ChickWeight[,"Chick"])
sapply(ChWtgrps, nrow)# typically 12 obs.
nlis1 <- lapply(ChWtgrps, function(DAT) tryCatch(error = identity,
                      lm(weight ~ (Time + I(Time^2)) * Diet, data = DAT)))
nl1 <- warnErrList(nlis1) #-> warning :
## 50 times the same error (as Diet has only one level in each group)
stopifnot(sapply(nl1, is.null)) ## all errors --> all replaced by NULL
nlis2 <- lapply(ChWtgrps, function(DAT) tryCatch(error = identity,
```

```
                      lm(weight ~ Time + I(Time^2), data = DAT)))
nl2 <- warnErrList(nlis2)
stopifnot(identical(nl2, nlis2)) # because there was *no* error at all
nlis3 <- lapply(ChWtgrps, function(DAT) tryCatch(error = identity,
                      lm(weight ~ poly(Time, 3), data = DAT)))
nl3 <- warnErrList(nlis3) # 1 error caught:
stopifnot(inherits(nlis3[[1]], "error")
        , identical(nl3[-1], nlis3[-1])
        , is.null(nl3[[1]])
)

## With different error messages
if(requireNamespace("nlme")) { # almost always, as it is recommended
 data(Soybean, package="nlme")
 attr(Soybean, "formula") #-> weight ~ Time | Plot  => split by "Plot":
 L <- lapply(split(Soybean, Soybean[,"Plot"]),
             function(DD) tryCatch(error = identity,
                 nls(weight ~ SSlogis(Time, Asym, xmid, scal), data = DD)))
 Lw <- warnErrList(L)
} # if <nlme>
```

---

winDialog *Dialog Boxes under Windows*

Description

On MS Windows only, put up a dialog box to communicate with the user. There are various types, either for the user to select from a set of buttons or to edit a string.

Usage

```
winDialog(type = c("ok", "okcancel", "yesno", "yesnocancel"),
          message)

winDialogString(message, default)
```

Arguments

type	character. The type of dialog box. It will have the buttons implied by its name.
message	character. The information field of the dialog box. Limited to 255 chars (by Windows, checked by R).
default	character. The default string.

Value

For winDialog a character string giving the name of the button pressed (in capitals) or NULL (invisibly) if the user had no choice.

For winDialogString a string giving the contents of the text box when Ok was pressed, or NULL if Cancel was pressed.

Note

The standard keyboard accelerators work with these dialog boxes: where appropriate `Return` accepts the default action, `Esc` cancels and the underlined initial letter (`Y` or `N`) can be used.

These functions are only available on Windows.

See Also

`winMenuAdd`
`file.choose` to select a file
package `windlgs` in the package source distribution for ways to program dialogs in C in the `GraphApp` toolkit.

Examples

```
## Not run: winDialog("yesno", "Is it OK to delete file blah")
```

---

winextras *Get Windows Version*

Description

Get the self-reported Microsoft Windows version number.

Usage

```
win.version()
```

Details

`win.version` is an auxiliary function for `sessionInfo` and `bug.report`.

Value

A character string describing the version of Windows reported to be in use.

Note

This function is only available on Microsoft Windows.

The result is based on the Windows `GetVersionEx` API function. It is not known how to detect a version of Windows before it is released, and hence the textual information returned by R may identify an older version than installed. The build number is more reliable. When running R in compatibility mode, the reported version including the build number is the compatibility version, not the installed version.

Examples

```
if(.Platform$OS.type == "windows")
   print(win.version())
```

winMenus *User Menus under MS Windows (RGui)*

Description

Enables users to add, delete and program menus for the Rgui in MS Windows.

Usage

```
winMenuAdd(menuname)
winMenuAddItem(menuname, itemname, action)
winMenuDel(menuname)
winMenuDelItem(menuname, itemname)
winMenuNames()
winMenuItems(menuname)
```

Arguments

menuname	a character string naming a menu.
itemname	a character string naming a menu item on an existing menu.
action	a character string describing the action when that menu is selected, or "enable" or "disable".

Details

User menus are added to the right of existing menus, and items are added at the bottom of the menu.

By default the action character string is treated as R input, being echoed on the command line and parsed and executed as usual.

If the menuname parameter of winMenuAddItem does not already exist, it will be created automatically.

Normally new submenus and menu items are added to the main console menu. They may be added elsewhere using the following special names:

$ConsoleMain The console menu (the default)

$ConsolePopup The console popup menu

$Graph<n>Main The menu for graphics window <n>

$Graph<n>Popup The popup menu for graphics window <n>

Specifying an existing item in winMenuAddItem enables the action to be changed.

Submenus can be specified by separating the elements in menuname by slashes: as a consequence menu names may not contain slashes.

If the action is specified as "none" no action is taken: this can be useful to reserve items for future expansion.

The function winMenuNames can be used to find out what menus have been created by the user and returns a vector of the existing menu names.

The `winMenuItems` function will take the name of a menu and return the items that exist in that menu. The return value is a named vector where the names correspond to the names of the items and the values of the vector are the corresponding actions.

The `winMenuDel` function will delete a menu and all of its items and submenus. `winMenuDelItem` just deletes one menu item.

The total path to an item (menu string plus item string) cannot exceed 1000 bytes, and the menu string cannot exceed 500 bytes.

Value

`NULL`, invisibly. If an error occurs, an informative error message will be given.

Note

These functions are only available on Windows and only when using the `Rgui`, hence not in `ESS` nor `RStudio`.

See Also

`winDialog`

Examples

```
## Not run:
winMenuAdd("Testit")
winMenuAddItem("Testit", "one", "aaaa")
winMenuAddItem("Testit", "two", "bbbb")
winMenuAdd("Testit/extras")
winMenuAddItem("Testit", "-", "")
winMenuAddItem("Testit", "two", "disable")
winMenuAddItem("Testit", "three", "cccc")
winMenuAddItem("Testit/extras", "one more", "ddd")
winMenuAddItem("Testit/extras", "and another", "eee")
winMenuAdd("$ConsolePopup/Testit")
winMenuAddItem("$ConsolePopup/Testit", "six", "fff")
winMenuNames()
winMenuItems("Testit")

## End(Not run)
```

---

winProgressBar *Progress Bars under MS Windows*

Description

Put up a Windows progress bar widget, update and access it.

Usage

```
winProgressBar(title = "R progress bar", label = "",
               min = 0, max = 1, initial = 0, width = 300)

getWinProgressBar(pb)
setWinProgressBar(pb, value, title = NULL, label = NULL)
## S3 method for class 'winProgressBar'
close(con, ...)
```

Arguments

`title`, `label`	character strings, giving the window title and the label on the dialog box respectively.
`min`, `max`	(finite) numeric values for the extremes of the progress bar.
`initial`, `value`	initial or new value for the progress bar.
`width`	the width of the progress bar in pixels: the dialog box will be 40 pixels wider (plus frame).
`pb`, `con`	an object of class `"winProgressBar"`.
`...`	for consistency with the generic.

Details

`winProgressBar` will display a progress bar centred on the screen. Space will be allocated for the label only if it is non-empty.

`setWinProgessBar` will update the value and for non-`NULL` values, the title and label (provided there was one when the widget was created). Missing (`NA`) and out-of-range values of `value` will be (silently) ignored.

The progress bar should be `closed` when finished with, but it will be garbage-collected once no R object refers to it.

Value

For `winProgressBar` an object of class `"winProgressBar"`.

For `getWinProgressBar` and `setWinProgressBar`, a length-one numeric vector giving the previous value (invisibly for `setWinProgressBar`).

Note

These functions are only available on Windows.

See Also

On all platforms, `txtProgressBar`, `tkProgressBar`

write.table *Data Output*

Description

write.table prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")

write.csv(...)
write.csv2(...)
```

Arguments

x	the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce x to a data frame.
file	either a character string naming a file or a connection open for writing. "" indicates output to the console.
append	logical. Only relevant if file is a character string. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
quote	a logical value (TRUE or FALSE) or a numeric vector. If TRUE, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If FALSE, nothing is quoted.
sep	the field separator string. Values within each row of x are separated by this string.
eol	the character(s) to print at the end of each line (row). For example, eol = "\r\n" will produce Windows' line endings on a Unix-alike OS, and eol = "\r" will produce files as expected by Excel:mac 2004.
na	the string to use for missing values in the data.
dec	the string to use for decimal points in numeric or complex columns: must be a single character.
row.names	either a logical value indicating whether the row names of x are to be written along with x, or a character vector of row names to be written.
col.names	either a logical value indicating whether the column names of x are to be written along with x, or a character vector of column names to be written. See the section on 'CSV files' for the meaning of col.names = NA.

qmethod	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of "escape" (default for write.table), in which case the quote character is escaped in C style by a backslash, or "double" (default for write.csv and write.csv2), in which case it is doubled. You can specify just the initial letter.
fileEncoding	character string: if non-empty declares the encoding to be used on a file (not a connection) so the character data can be re-encoded as they are written. See file.
...	arguments to write.table: append, col.names, sep, dec and qmethod cannot be altered.

Details

If the table has no columns the rownames will be written only if row.names = TRUE, and *vice versa*.

Real and complex numbers are written to the maximal possible precision.

If a data frame has matrix-like columns these will be converted to multiple columns in the result (*via* as.matrix) and so a character col.names or a numeric quote should refer to the columns in the result, not the input. Such matrix-like columns are unquoted by default.

Any columns in a data frame which are lists or have a class (e.g., dates) will be converted by the appropriate as.character method: such columns are unquoted by default. On the other hand, any class information for a matrix is discarded and non-atomic (e.g., list) matrices are coerced to character.

Only columns which have been converted to character will be quoted if specified by quote.

The dec argument only applies to columns that are not subject to conversion to character because they have a class or are part of a matrix-like column (or matrix), in particular to columns protected by I(). Use options("OutDec") to control such conversions.

In almost all cases the conversion of numeric quantities is governed by the option "scipen" (see options), but with the internal equivalent of digits = 15. For finer control, use format to make a character matrix/data frame, and call write.table on that.

These functions check for a user interrupt every 1000 lines of output.

If file is a non-open connection, an attempt is made to open it and then close it after use.

To write a Unix-style file on Windows, use a binary connection e.g. file = file("filename", "wb").

CSV files

By default there is no column name for a column of row names. If col.names = NA and row.names = TRUE a blank column name is added, which is the convention used for CSV files to be read by spreadsheets. Note that such CSV files can be read in R by

```
read.csv(file = "<filename>", row.names = 1)
```

write.csv and write.csv2 provide convenience wrappers for writing CSV files. They set sep and dec (see below), qmethod = "double", and col.names to NA if row.names = TRUE (the default) and to TRUE otherwise.

`write.csv` uses `"."` for the decimal point and a comma for the separator.

`write.csv2` uses a comma for the decimal point and a semicolon for the separator, the Excel convention for CSV files in some Western European locales.

These wrappers are deliberately inflexible: they are designed to ensure that the correct conventions are used to write a valid file. Attempts to change `append`, `col.names`, `sep`, `dec` or `qmethod` are ignored, with a warning.

CSV files do not record an encoding, and this causes problems if they are not ASCII for many other applications. Windows Excel 2007/10 will open files (e.g., by the file association mechanism) correctly if they are ASCII or UTF-16 (use `fileEncoding = "UTF-16LE"`) or perhaps in the current Windows codepage (e.g., `"CP1252"`), but the 'Text Import Wizard' (from the 'Data' tab) allows far more choice of encodings. Excel:mac 2004/8 can *import* only 'Macintosh' (which seems to mean Mac Roman), 'Windows' (perhaps Latin-1) and 'PC-8' files. OpenOffice 3.x asks for the character set when opening the file.

There is an IETF RFC4180 (https://www.rfc-editor.org/rfc/rfc4180) for CSV files, which mandates comma as the separator and CRLF line endings. `write.csv` writes compliant files on Windows: use `eol = "\r\n"` on other platforms.

Note

`write.table` can be slow for data frames with large numbers (hundreds or more) of columns: this is inevitable as each column could be of a different class and so must be handled separately. If they are all of the same class, consider using a matrix instead.

See Also

The 'R Data Import/Export' manual.

`read.table`, `write`.

`write.matrix` in package **MASS**.

Examples

```
x <- data.frame(a = I("a \" quote"), b = pi)
tf <- tempfile(fileext = ".csv")

## To write a CSV file for input to Excel one might use
write.table(x, file = tf, sep = ",", col.names = NA,
            qmethod = "double")
file.show(tf)
## and to read this file back into R one needs
read.table(tf, header = TRUE, sep = ",", row.names = 1)
## NB: you do need to specify a separator if qmethod = "double".

### Alternatively
write.csv(x, file = tf)
read.csv(tf, row.names = 1)
## or without row names
write.csv(x, file = tf, row.names = FALSE)
read.csv(tf)
```

```
## Not run:
## To write a file in Mac Roman for simple use in Mac Excel 2004/8
write.csv(x, file = "foo.csv", fileEncoding = "macroman")
## or for Windows Excel 2007/10
write.csv(x, file = "foo.csv", fileEncoding = "UTF-16LE")

## End(Not run)
```

`zip` *Create Zip Archives*

Description

A wrapper for an external `zip` command to create zip archives.

Usage

```
zip(zipfile, files, flags = "-r9X", extras = "",
    zip = Sys.getenv("R_ZIPCMD", "zip"))
```

Arguments

`zipfile`	The pathname of the zip file: tilde expansion (see `path.expand`) will be performed.
`files`	A character vector of recorded filepaths to be included.
`flags`	A character string of flags to be passed to the command: see 'Details'.
`extras`	An optional character vector: see 'Details'.
`zip`	A character string specifying the external command to be used.

Details

On a Unix-alike, the default for `zip` will use the value of `R_ZIPCMD`, whose default is set in '`etc/Renviron`' to the `zip` command found during configuration. On Windows, the default relies on a `zip` program (for example that from Rtools) being in the path.

The default for `flags` is that appropriate for zipping up a directory tree in a portable way: see the system-specific help for the `zip` command for other possibilities.

Argument `extras` can be used to specify `-x` or `-i` followed by a list of filepaths to exclude or include. Since `extras` will be treated as if passed to `system`, if the filepaths contain spaces they must be quoted e.g. by `shQuote`.

Value

The status value returned by the external command, invisibly.

See Also

`unzip`, `unz`; further, `tar` and `untar` for (un)packing tar archives.

Index