

Package ‘spatstat.utils’

June 14, 2021

Version 2.2-0

Date 2021-06-14

Title Utility Functions for 'spatstat'

Maintainer Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Depends R (>= 3.3.0), stats, graphics, grDevices, utils, methods

Suggests spatstat.core

Description Contains utility functions for the 'spatstat' family of packages which may also be useful for other purposes.

License GPL (>= 2)

URL <http://spatstat.org/>

NeedsCompilation yes

ByteCompile true

BugReports <https://github.com/spatstat/spatstat.utils/issues>

Author Adrian Baddeley [aut, cre] (<<https://orcid.org/0000-0001-9499-8382>>),
Rolf Turner [aut] (<<https://orcid.org/0000-0001-5521-5218>>),
Ege Rubak [aut] (<<https://orcid.org/0000-0002-6675-533X>>)

Repository CRAN

Date/Publication 2021-06-14 04:30:02 UTC

R topics documented:

spatstat.utils-package	2
articlebeforenumber	4
cat.factor	5
check.l.integer	6
check.anyvector	7
check.named.vector	8
check.nmatrix	9
check.nvector	11
check.range	12

commasep	13
do.call.matched	14
do.call.without	15
exactCutBreaks	16
expand.polynom	17
ifelseAB	18
methods.xypolygon	20
optimizeWithTrace	22
orderstats	23
ordinal	24
paren	25
percentage	26
primefactors	27
queueSpatstatLocator	28
RelevantNA	29
resolve.defaults	30
revcumsum	32
simplenumber	33
spatstatLocator	34
splat	35
taperoff	36
tapplysum	37
termsinformula	38
verbalogic	39

Index 41

spatstat.utils-package

The spatstat.utils Package

Description

The **spatstat.utils** package contains low-level utilities, written for the **spatstat** package, which may be useful in other packages as well.

Details

The functions in **spatstat.utils** were originally written as internal, undocumented, utility functions in the **spatstat** package.

Many of these functions could be useful to other programmers, so we have made them available in a separate package **spatstat.utils** and provided documentation.

The functionality contained in **spatstat.utils** includes:

Factorisation of integers Find prime numbers ([primesbelow](#)), factorise a composite number into its prime factors ([primefactors](#)), determine whether a number is prime ([is.prime](#)) or whether two numbers are relatively prime ([relatively.prime](#)), and find the least common multiple or greatest common divisor of two numbers ([least.common.multiple](#), [greatest.common.divisor](#)).

Faster versions of basic R tools Faster versions of some basic R tools and idioms are provided. These are only faster in particular cases, but if you know that your data have a particular form, the acceleration can be substantial. See [ifelseAB](#), [fave.order](#), [revcumsum](#), [tapplysum](#).

Grammar Use the correct word in English to refer to an ordinal number ([ordinal](#), [ordinalsuffix](#)) and the correct indefinite article ([articlebeforenumber](#)).

Tools for generating printed output The function [splat](#) is a replacement for `cat(paste(...))` which ensures that output stays inside the declared text margin ([getOption\("width"\)](#)) and can also perform automatic indentation. There are useful functions to add or remove parentheses ([paren](#), [unparen](#)) and to make comma-separated lists ([commasep](#)).

Handling intervals (ranges) of real numbers Simple functions handle an interval (range) of numerical values: [check.range](#), [intersect.ranges](#), [inside.range](#), [check.in.range](#), [prange](#).

Handling a formula Tools for handling a formula in the R language include [lhs.of.formula](#), [rhs.of.formula](#), [variablesinformula](#), [termsinformula](#), [offsetsinformula](#), [can.be.formula](#) and [identical.formulae](#).

Polynomials There are tools for creating and manipulating symbolic expressions for polynomials, as they might appear in a formula ([sympoly](#), [expand.polynom](#)).

Validating arguments There are many tools for validating an argument and generating a comprehensible error or warning message if the argument is not valid: [check.1.integer](#), [check.nvector](#), [check.named.vector](#).

Passing arguments There are many tools for calling a function while passing only some of the arguments in a supplied list of arguments: [do.call.matched](#), [do.call.without](#), [resolve.defaults](#).

Traced optimization [optimizeWithTrace](#) is a simple wrapper for the one-dimensional optimization routine [optimize](#). It stores the values of the function argument each time it is called, stores the resulting function values, and returns them along with the optimal value.

Workarounds There are workarounds for known bugs or undesirable features in other software. [spatstatLocator](#) is a replacement for [locator](#) which works around a bug in the RStudio graphics interface. [cat.factor](#) concatenates several factors, merging the levels, to produce a new factor.

Licence

This library and its documentation are usable under the terms of the “GNU General Public License”, a copy of which is distributed with R.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz> and Ege Rubak <rubak@math.aau.dk>.

`cat.factor`*Combine Several Factors*

Description

Combine (concatenate) several factor objects, to produce a factor.

Usage

```
cat.factor(...)
```

Arguments

... Any number of arguments. Each argument should be a factor, or will be converted to a factor.

Details

The arguments ... are concatenated as they would be using `c()` or `cat()`, except that factor levels are retained and merged correctly. See the Examples.

Value

A factor, whose length is the sum of the lengths of all arguments. The levels of the resulting factor are the union of the levels of the arguments.

Author(s)

Rolf Turner <r.turner@auckland.ac.nz>.

See Also

[c](#).

Examples

```
f <- factor(letters[1:3])
g <- factor(letters[3:5])
f
g
cat(f,g)
c(f,g)
cat.factor(f, g)
```

check.1.integer *Check Argument Type and Length*

Description

These utility functions check whether a given argument is a single value of the required type.

Usage

```
check.1.real(x, context = "", fatal = TRUE)
check.1.integer(x, context = "", fatal = TRUE)
check.1.string(x, context = "", fatal = TRUE)
```

Arguments

x	The argument to be checked.
context	Optional string specifying the context in which the argument is checked.
fatal	Logical value indicating what to do if x is not of the required type.

Details

These functions check whether the argument x is a single atomic value of type numeric, integer or character.

If x does have the required length and type, the result of the function is the logical value TRUE.

If x does not have the required length and type, then if fatal=TRUE (the default) an error occurs, while if fatal=FALSE a warning is issued and the function returns the value FALSE.

Value

A logical value (or an error may occur).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[check.named.vector](#)

Examples

```
x <- pi
check.1.real(x)
check.1.integer(x, fatal=FALSE)
check.1.string(x, fatal=FALSE)
```

check.anyvector	<i>Check For Vector or Factor With Correct Length</i>
-----------------	---

Description

This is a programmer's utility function to check whether the argument is a vector or factor of the correct length.

Usage

```
check.anyvector(v, npoints = NULL, fatal = TRUE, things = "data points",  
               naok = FALSE, warn = FALSE, vname, oneok = FALSE)
```

Arguments

v	The argument to be checked.
npoints	The required length of v.
fatal	Logical value indicating whether to stop with an error message if v does not satisfy all requirements.
things	Character string describing what the entries of v should correspond to.
naok	Logical value indicating whether NA values are permitted.
warn	Logical value indicating whether to issue a warning if v does not satisfy all requirements.
vname	Character string giving the name of v to be used in messages.
oneok	Logical value indicating whether v is permitted to have length 1.

Details

This function checks whether v is a vector or factor with length equal to npoints (or length equal to 1 if oneok=TRUE), not containing any NA values (unless naok=TRUE).

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if fatal=TRUE (the default), an error occurs; if fatal=FALSE, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied. If FALSE, then this value has an attribute "whinge", a character string describing the requirements that were not satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[check.nvector](#), [check.named.vector](#).

Examples

```
z <- factor(1:10)
check.anyvector(z, 5, fatal=FALSE)
y <- z[1]
check.anyvector(y, 5, oneok=TRUE)
```

check.named.vector *Check Whether Object Has Required Components*

Description

These functions check whether the object `x` has components with the required names, and does not have any unexpected components.

Usage

```
check.named.vector(x, nam, context = "", namopt = character(0),
                  onError = c("fatal", "null"))

check.named.list(x, nam, context = "", namopt = character(0),
                onError = c("fatal", "null"))

check.named.thing(x, nam, namopt = character(0),
                 xtitle = NULL, valid = TRUE, type = "object",
                 context = "", fatal = TRUE)
```

Arguments

<code>x</code>	The object to be checked.
<code>nam</code>	Vector of character strings giving the names of all the components which must be present.
<code>namopt</code>	Vector of character strings giving the names of components which may optionally be present.
<code>context</code>	Character string describing the context in which <code>x</code> is being checked.
<code>xtitle</code>	Optional character string to be used when referring to <code>x</code> .
<code>valid</code>	Logical value indicating whether <code>x</code> belongs to the required class of objects.
<code>type</code>	Character string describing the required class of objects.
<code>onError</code>	Character string indicating what to do if <code>x</code> fails the checks.
<code>fatal</code>	Logical value indicating what to do if <code>x</code> fails the checks. If <code>fatal=TRUE</code> (the default), an error occurs.

Details

check.named.thing checks whether `x` has all the required components, in the sense that `names(x)` includes all the names in `nam`, and that every entry in `names(x)` belongs to either `nam` or `namopt`. If all these checks are true, the result is a zero-length character vector. Otherwise, if `fatal=TRUE` (the default), an error occurs; otherwise the result is a character vector describing the checks which failed.

check.named.vector checks whether `x` is a numeric vector and check.named.list checks whether `x` is a list. They then call check.named.thing to check whether all the required components are present. If all these checks are true, the result is a reordered version of `x` in which all the compulsory entries appear first. Otherwise, if `onError="fatal"` (the default) an error occurs; otherwise the result is `NULL`.

Value

check.named.vector returns a numeric vector or `NULL`.

check.named.list returns a list or `NULL`.

check.named.thing returns a character vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[check.1.integer](#)

Examples

```
z <- list(a=1, b=2, e=42)
check.named.list(z, c("a", "b"), namopt=c("c", "d", "e"))
check.named.thing(z, c("a", "b"), namopt=c("c", "d", "e"))
zz <- unlist(z)
check.named.vector(zz, c("a", "b"), namopt=c("c", "d", "e"))
check.named.thing(z, c("b", "c"), namopt=c("d", "e"), fatal=FALSE)
```

check.nmatrix

Check for Numeric Matrix with Correct Dimensions

Description

This is a programmer's utility function to check whether the argument is a numeric vector of the correct length.

Usage

```
check.nmatrix(m, npoints = NULL, fatal = TRUE, things = "data points",
             naok = FALSE, squarematrix = TRUE, matchto = "nrow", warn = FALSE)
```

Arguments

<code>m</code>	The argument to be checked.
<code>npoints</code>	The required number of rows and/or columns for the matrix <code>m</code> .
<code>fatal</code>	Logical value indicating whether to stop with an error message if <code>m</code> does not satisfy all requirements.
<code>things</code>	Character string describing what the rows/columns of <code>m</code> should correspond to.
<code>naok</code>	Logical value indicating whether NA values are permitted.
<code>squarematrix</code>	Logical value indicating whether <code>m</code> must be a square matrix.
<code>matchto</code>	Character string (either "nrow" or "ncol") indicating whether it is the rows or the columns of <code>m</code> which must correspond to <code>npoints</code> .
<code>warn</code>	Logical value indicating whether to issue a warning if <code>v</code> does not satisfy all requirements.

Details

This programmer's utility function checks whether `m` is a numeric matrix of the correct dimensions, and checks for NA values. If `matchto="nrow"` (the default) then the number of rows of `m` must be equal to `npoints`. If `matchto="ncol"` then the number of columns of `m` must be equal to `npoints`. If `squarematrix=TRUE` (the default) then the numbers of rows and columns must be equal. If `naok = FALSE` (the default) then the entries of `m` must not include NA.

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if `fatal=TRUE` (the default), an error occurs; if `fatal=FALSE`, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[check.nvector](#)

Examples

```
z <- matrix(1:16, 4, 4)
check.nmatrix(z, 4)
```

check.nvector	<i>Check For Numeric Vector With Correct Length</i>
---------------	---

Description

This is a programmer's utility function to check whether the argument is a numeric vector of the correct length.

Usage

```
check.nvector(v, npoints = NULL, fatal = TRUE, things = "data points",
             naok = FALSE, warn = FALSE, vname, oneok = FALSE)
```

Arguments

v	The argument to be checked.
npoints	The required length of v.
fatal	Logical value indicating whether to stop with an error message if v does not satisfy all requirements.
things	Character string describing what the entries of v should correspond to.
naok	Logical value indicating whether NA values are permitted.
warn	Logical value indicating whether to issue a warning if v does not satisfy all requirements.
vname	Character string giving the name of v to be used in messages.
oneok	Logical value indicating whether v is permitted to have length 1.

Details

This function checks whether v is a numeric vector with length equal to npoints (or length equal to 1 if oneok=TRUE), not containing any NA values (unless naok=TRUE).

If these requirements are all satisfied, the result is the logical value TRUE.

If not, then if fatal=TRUE (the default), an error occurs; if fatal=FALSE, the result is the logical value FALSE with an attribute describing the requirement that was not satisfied.

Value

A logical value indicating whether all the requirements were satisfied. If FALSE, then this value has an attribute "whinge", a character string describing the requirements that were not satisfied.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[check.anyvector](#), [check.nmatrix](#), [check.l.real](#), [check.named.vector](#).

Examples

```
z <- 1:10
check.nvector(z, 5, fatal=FALSE)
y <- 42
check.nvector(y, 5, fatal=FALSE, oneok=TRUE)
```

 check.range

Utilities for Ranges of Values

Description

These simple functions handle an interval or range of numerical values. `check.range(r)` checks whether `r` specifies a range of values, that is, whether `r` is a vector of length 2 with `r[1] <= r[2]`. `intersect.ranges(r, s)` finds the intersection of two ranges `r` and `s`. `inside.range(x, r)` returns a logical vector containing TRUE if the corresponding entry of `x` falls inside the range `r`, and FALSE if it does not. `check.in.range(x, r)` checks whether a single number `x` falls inside the specified range `r`. Finally `prange(r)` produces a character string that represents the range `r`.

Usage

```
check.range(r, fatal = TRUE)

check.in.range(x, r, fatal = TRUE)

inside.range(x, r)

intersect.ranges(r, s, fatal = TRUE)

prange(r)
```

Arguments

<code>r</code>	A numeric vector of length 2 specifying the endpoints of a range of values.
<code>x</code>	Numeric vector of data.
<code>s</code>	A numeric vector of length 2 specifying the endpoints of a range of values.
<code>fatal</code>	Logical value indicating whether to stop with an error message if the data do not pass the check.

Details

`check.range` checks whether `r` specifies a range of values, that is, whether `r` is a vector of length 2 with `r[1] <= r[2]`. If so, the result is TRUE. If not, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is FALSE.

`intersect.ranges(r, s)` finds the intersection of two ranges `r` and `s`. If the intersection is non-empty, the result is a numeric vector of length 2. If the intersection is empty, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is NULL.

`inside.range(x,r)` returns a logical vector containing TRUE if the corresponding entry of `x` falls inside the range `r`, and FALSE if it does not.

`check.in.range(x,r)` checks whether a single number `x` falls inside the specified range `r`. If so, the result is TRUE. If not, then if `fatal=TRUE`, an error occurs, while if `fatal=FALSE` the result is FALSE.

Finally `prange(r)` produces a character string that represents the range `r`.

Value

The result of `check.range`, `check.in.range` and `inside.range`, is a logical value or logical vector. The result of `intersect.ranges` is a numerical vector of length 2, or NULL. The result of `prange` is a character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Examples

```
rr <- c(0, 2)
ss <- c(1, 3)
x <- seq(0.5, 3.5, by=1)
check.range(rr)
check.range(42, fatal=FALSE)
inside.range(x, rr)
intersect.ranges(rr, ss)
prange(rr)
```

commasep

List of Items Separated By Commas

Description

Convert the elements of a vector into character strings and paste them together, separated by commas.

Usage

```
commasep(x, join = " and ", flatten = TRUE)
```

Arguments

<code>x</code>	Vector of items in the list.
<code>join</code>	The string to be used to separate the last two items in the list.
<code>flatten</code>	Logical value indicating whether to return a single character string (<code>flatten=TRUE</code> , the default) or a list (<code>flatten=FALSE</code>).

Value

A character string (if `flatten=TRUE`, the default) or a list of character strings.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
commasep(letters[1:4])
y <- commasep(sQuote(letters[1:4]))
cat(y, fill=TRUE)
```

do.call.matched

Call a Function, Passing Only Recognised Arguments

Description

Call a specified function, using only those arguments which are known to be acceptable to the function.

Usage

```
do.call.matched(fun, arglist, funargs, extrargs = NULL,
               matchfirst = FALSE, sieve = FALSE, skipargs = NULL,
               envir=parent.frame())
```

Arguments

<code>fun</code>	A function, or a character string giving the name of a function, to be called.
<code>arglist</code>	A named list of arguments.
<code>funargs</code>	Character vector giving the names of arguments that are recognised by <code>fun</code> . Defaults to the names of the formal arguments of <code>fun</code> .
<code>extrargs</code>	Optional. Character vector giving the names of additional arguments that can be handled by <code>fun</code> .
<code>skipargs</code>	Optional. Character vector giving the names of arguments which should not be passed to <code>fun</code> .
<code>matchfirst</code>	Logical value indicating whether the first entry of <code>arglist</code> is permitted to have an empty name and should be matched to the first argument of <code>fun</code> .
<code>sieve</code>	Logical value indicating whether to return the un-used arguments as well as the result of the function call. See Details.
<code>envir</code>	An environment within which to evaluate the call, if any entries of <code>arglist</code> are quoted expressions.

Details

This function is a wrapper for `do.call` which avoids passing arguments that are unrecognised by `fun`.

In the simplest case `do.call.matched(fun, arglist)` is like `do.call(fun, arglist)`, except that entries of `arglist` which do not match any formal argument of `fun` are removed. Extra argument names can be permitted using `extrargs`, and argument names can be forbidden using `skipargs`.

Value

If `sieve=FALSE` (the default), the result is the return value from `fun`.

If `sieve=TRUE`, the result is a list with entries `result` (the return value from `fun`) and `otherargs` (a list of the arguments that were not passed to `fun`).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

[resolve.defaults](#), [do.call.without](#).

[do.call](#)

Examples

```
f <- function(x=0,y=0, ...) { paste(x, y, ..., sep=", ") }
f()
do.call.matched(f, list(y=2))
do.call.matched(f, list(y=2, z=5), extrargs="z")
do.call.matched(f, list(y=2, z=5), extrargs="z", skipargs="y")
```

do.call.without

Call a Function, Omitting Certain Arguments

Description

Call a specified function, omitting some arguments which are inappropriate to the function.

Usage

```
do.call.without(fun, ..., avoid, envir=parent.frame())
```

Arguments

fun	The function to be called. A function name, a character string giving the name of the function, or an expression that yields a function.
...	Any number of arguments.
avoid	Vector of character strings, giving the names of arguments that should <i>not</i> be passed to fun.
envir	An environment within which to evaluate the call, if any entries of <code>arglist</code> are quoted expressions.

Details

This is a simple mechanism for preventing some arguments from being passed in a function call. The arguments ... are collected in a list. A argument is omitted if its name exactly matches one of the strings in `avoid`.

Value

The return value of fun.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[do.call.matched](#) for a more complicated and flexible call.

Examples

```
do.call.without(paste, 1, 2, z=3, w=4, avoid="z")
```

exactCutBreaks

Determine Breakpoints for Cut

Description

Computes the numerical breakpoints used by `cut.default`.

Usage

```
exactCutBreaks(x, breaks)
```

Arguments

x	Numeric vector which would be converted to a factor.
breaks	Either a numeric vector of breakpoints, or a single integer giving the number of intervals into which x will be cut.

Details

This function contains a copy of the code in `cut.default` which determines the numerical breakpoints used to convert `x` to a factor. It returns the breakpoints only.

The arguments `x` and `breaks` have the same interpretation as in `cut.default`. Only the range of `x` is used in the computation, so `x` could be replaced by `range(x)`.

This function would normally be used when `breaks` is a single integer specifying the number of intervals for the cut operation. It returns the exact numerical values of the breakpoints which are determined, but not returned, by `cut.default`.

Value

Numeric vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[cut.default](#)

Examples

```
exactCutBreaks(c(0,1), 4)
```

expand.polynom

Expand Symbolic Polynomials in a Formula

Description

Create a formula representing a polynomial, or expand polynomials in an existing formula.

Usage

```
expand.polynom(f)  
sympoly(x, y, n)
```

Arguments

<code>f</code>	A formula.
<code>x,y</code>	Variable names.
<code>n</code>	Integer specifying the degree of the polynomial. (If <code>n</code> is missing, <code>y</code> will be interpreted as the degree.)

Details

These functions expand a polynomial into its homogeneous terms and return a model formula.

`sympoly(x,n)` creates a formula whose right-hand side represents the polynomial of degree n in the variable x . Each homogeneous term x^k is a separate term in the formula.

`sympoly(x,y,n)` creates a formula representing the polynomial of degree n in the two variables x and y .

If f is a formula containing a term of the form `polynom(...)` then `expand.polynom(f)` replaces this term by its expansion as a sum of homogeneous terms, as defined in the help for [polynom](#).

Value

A formula.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz> and Ege Rubak <rubak@math.aau.dk>.

See Also

[polynom](#)

Examples

```
sympoly(A, 4)
sympoly(A, B, 3)
expand.polynom(U ~ A + polynom(B, 2))
```

ifelseAB

Conditional Selection

Description

These low-level functions provide faster alternatives to some uses of `ifelse`.

Usage

```
ifelseAB(test, a, b)
ifelseAX(test, a, x)
ifelseXB(test, x, b)
ifelseXY(test, x, y)
ifelseNegPos(test, x)
ifelse0NA(test)
ifelse1NA(test)
```

Arguments

test	A logical vector.
a	A single atomic value.
b	A single atomic value.
x	A vector of values, of the same length as test.
y	A vector of values, of the same length as test.

Details

These low-level functions provide faster alternatives to some uses of `ifelse`. They were developed by trial-and-error comparison of computation times of different expressions.

`ifelse0NA(test)` is equivalent to `ifelse(test, 0, NA)`.

`ifelse1NA(test)` is equivalent to `ifelse(test, 1, NA)`.

`ifelseAB(test, a, b)` is equivalent to `ifelse(test, a, b)` where `a` and `b` must be single values.

`ifelseAX(test, a, x)` is equivalent to `ifelse(test, a, x)` where `a` must be a single value, and `x` a vector of the same length as `test`.

`ifelseXB(test, x, b)` is equivalent to `ifelse(test, x, b)` where `b` must be a single value, and `x` a vector of the same length as `test`.

`ifelseXY(test, x, y)` is equivalent to `ifelse(test, x, y)` where `x` and `y` must be vectors of the same length as `test`.

`ifelseNegPos(test, x)` is equivalent to `ifelse(test, x, -x)` where `x` must be a vector of the same length as `test`.

Value

A vector of the same length as `test` containing values of the same type as `a, b, x, y`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz>
and Ege Rubak <rubak@math.aau.dk>.

See Also

[ifelse](#)

Examples

```
x <- runif(4e5)
u <- (x < 0.5)
system.time(ifelse(u, 2, x))
system.time(ifelseAX(u, 2, x))
```

methods.xypolygon *Calculations for Polygons in the Plane*

Description

Compute the area or boundary length of a polygon, determine whether a point falls inside a polygon, compute the area of overlap between two polygons, and related tasks.

Usage

```
verify.xypolygon(p, fatal = TRUE)
is.hole.xypolygon(polly)
Area.xypolygon(polly)
bdrylength.xypolygon(polly)
reverse.xypolygon(p, adjust=FALSE)
overlap.xypolygon(P, Q)
simplify.xypolygon(p, dmin)
inside.xypolygon(pts, polly, test01, method)
```

Arguments

<code>p, polly, P, Q</code>	Data representing a polygon. See Details.
<code>dmin</code>	Single numeric value giving the minimum permissible length of an edge in the simplified polygon.
<code>fatal</code>	Logical value indicating whether failure is a fatal error.
<code>pts</code>	Coordinates of points to be tested. A named list with entries <code>x, y</code> which are numeric vectors of coordinates.
<code>adjust</code>	Logical value indicating whether internal data should be adjusted. See Details.
<code>test01, method</code>	For developer use only.

Details

In the **spatstat** family of packages, a polygon in the Euclidean plane is represented as a named list with the following entries:

x, y Numeric vectors giving the coordinates of the vertices. The vertices should be traversed in anti-clockwise order (unless the polygon is a hole, when they should be traversed in clockwise order) and the last vertex should **not** repeat the first vertex.

hole Optional. A logical value indicating whether the polygon is a hole.

area Optional. Single numeric value giving the area of the polygon (negative if it is a hole).

The function `verify.xypolygon` checks whether its argument satisfies this format. If so, it returns TRUE; if not, it returns FALSE or (if `fatal=TRUE`) generates a fatal error.

The other functions listed here perform basic calculations for polygons using elementary Cartesian analytic geometry in R.

`is.hole.xypolygon` determines whether a polygon is a hole or not.

`Area.xypolygon` computes the area of the polygon using the discrete Green's formula.

`bdrylength.xypolygon` calculates the total length of edges of the polygon.

`reverse.xypolygon` reverses the order of the coordinate vectors `x` and `y`. If `adjust=TRUE`, the other entries `hole` and `area` will be adjusted as well.

`overlap.xypolygon` computes the area of overlap between two polygons using the discrete Green's formula. It is slow compared to the code in the **polyclip** package.

`simplify.xypolygon` removes vertices of the polygon until every edge is longer than `dmin`.

`inside.xypolygon(pts,polly)` determines whether each point in `pts` lies inside the polygon `polly` and returns a logical vector.

Value

`verify.xypolygon` and `is.hole.xypolygon` return a single logical value.

`inside.xypolygon` returns a logical vector.

`Area.xypolygon`, `bdrylength.xypolygon` and `overlap.xypolygon` return a single numeric value.

`reverse.xypolygon` and `simplify.xypolygon` return another polygon object.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
p <- list(x=c(0,1,4,2), y=c(0,0,2,3))
is.hole.xypolygon(p)
Area.xypolygon(p)
bdrylength.xypolygon(p)
overlap.xypolygon(p, list(x=p$x+1, y=p$y+1))
reverse.xypolygon(p)

plot(c(0,5),c(0,3),type="n",xlab="x", ylab="y")
polygon(p)
polygon(simplify.xypolygon(p, 1.1), lty=3)

plot(c(0,5),c(0,3),type="n",xlab="x", ylab="y")
polygon(p)
xx <- runif(10, max=5)
yy <- runif(10, max=3)
points(xx, yy)
ok <- as.logical(inside.xypolygon(list(x=xx, y=yy), p))
points(xx[ok], yy[ok], pch=16)
```

optimizeWithTrace *One Dimensional Optimization with Tracing*

Description

Find the minimum or maximum of a function over an interval of real numbers, keeping track of the function arguments and function values that were evaluated.

Usage

```
optimizeWithTrace(f, interval, ...,
                  lower = min(interval), upper = max(interval))
```

Arguments

f	The function to be minimized or maximized.
interval	Numeric vector of length 2 containing the end-points of the interval to be searched.
lower, upper	The lower and upper endpoints of the interval to be searched.
...	Other arguments passed to <code>optimize</code> , including arguments to the function f.

Details

This is a simple wrapper for the optimization routine `optimize`. The function f will be optimized by computing its value at several locations in the interval, as described in the help for `optimize`. This wrapper function stores the locations and resulting function values, and returns them along with the result of the optimization.

Value

A list with components

- minimum (or maximum), the location in the search interval which yielded the optimum value;
- objective, the value of the function at this location;
- x, the sequence of locations in the interval that were considered (in the order considered);
- y, the function values corresponding to x.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz>
and Ege Rubak <rubak@math.aau.dk>.

See Also

`optimize`

Examples

```
f <- function (x, a) (x - a)^2
result <- optimizeWithTrace(f, c(0, 1), tol = 0.0001, a = 1/3)
result
curve(f(x, 1/3))
with(result, points(x, y, pch=16))
```

orderstats

*Compute Order Statistics***Description**

Compute the k -th smallest value in a dataset, or find which entry in a dataset is the k -th smallest.

Usage

```
orderstats(x, k, decreasing = FALSE)
orderwhich(x, k, decreasing = FALSE)
```

Arguments

<code>x</code>	Data whose order statistics will be computed. A numeric vector.
<code>k</code>	Rank. An integer, or vector of integers.
<code>decreasing</code>	Logical value specifying whether a rank of 1 is assigned to the highest value (decreasing=TRUE) or the lowest value (decreasing=FALSE, the default).

Details

These are low-level functions for efficiently computing order statistics: `orderstats(x, k)` returns the k -th smallest value in x , and `orderwhich(x, k)` returns the *position* of the k -th smallest value in x .

Given a dataset of values x_1, \dots, x_n , the *order statistic* of rank k is the k -th smallest value in the dataset. The order statistic of rank 1 is the smallest value, and the order statistic of rank n is the largest value. The order statistic of rank k is denoted $x_{[k]}$.

The full sequence of order statistics

$$x_{[1]} \leq x_{[2]} \leq \dots \leq x_{[n]}$$

can simply be obtained by sorting the original values into increasing order.

The command `orderstats(x, k)` is equivalent to `sort(x)[k]`; it calculates the k -th smallest value in x .

The command `orderwhich(x, k)` is equivalent to `order(x)[k]`. It identifies the *position* of the k -th smallest value in x , that is, it returns the index j such that $x[j]$ is the k -th smallest value in x .

The functions `orderstats` and `orderwhich` are more efficient than using `sort` and `order` when it is only desired to calculate a few of the order statistics (for example, only the smallest and second-smallest values in the dataset).

Value

orderstats returns a vector of the same kind as x, with the same length as k. orderwhich returns an integer vector with the same length as k.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[sort](#), [order](#).

Examples

```
x <- runif(10)
orderstats(x, 2)
sort(x)[2]
orderwhich(x, 2:3)
order(x)[2:3]
```

ordinal

Ordinal Numbers

Description

Returns the appropriate abbreviation in English for an ordinal number (for example ordinal(5) is "5th").

Usage

```
ordinal(k)
ordinalsuffix(k)
```

Arguments

k An integer or vector of integers.

Details

ordinal(k) returns a character string representing the kth ordinal number. ordinalsuffix(k) determines the appropriate suffix.

The suffix can be either "st" (abbreviating *first*), "nd" (abbreviating *second*), "rd" (abbreviating *third*) or "th" (for all other ordinal numbers fourth, fifth, etc).

Value

A character string or character vector of the same length as k.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[articlebeforenumber](#)

Examples

```
ordinal(1:7)
cat(paste("Happy", ordinal(21), "Birthday"), fill=TRUE)
```

paren

Add or Remove Parentheses

Description

Add or remove enclosing parentheses around a string.

Usage

```
paren(x, type = "(")
unparen(x)
```

Arguments

x	A character string, or vector of character strings.
type	Type of parentheses: either "(", "[" or "{".

Details

paren(x) adds enclosing parentheses to the beginning and end of the string x.
unparen(x) removes enclosing parentheses if they are present.

Value

A character string, or vector of character strings of the same length as x.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[commasep](#)

Examples

```
paren("Hello world")
paren(42, "[")
paren(letters[1:10])
unparen(c("yes"), "[no]", "{42}")
```

percentage

Convert Fraction to Percentage

Description

This is a programmer's utility which converts a fraction to a percentage and encodes the percentage as a character string.

Usage

```
percentage(x, digits = 3)
```

Arguments

<code>x</code>	Either a single number, or a logical vector.
<code>digits</code>	Number of digits accuracy.

Details

If `x` is a single number, it should be a fraction between 0 and 1. It will be converted to a percentage and then converted to a character string followed by the percentage symbol.

If `x` is a logical vector, the fraction of values which are TRUE will be computed, and used to determine the percentage.

Value

A character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Examples

```
percentage(1/3)
percentage(runif(20) > 0.2)
```

primefactors

Primes, Prime Factorization, Common Divisor

Description

These functions find prime numbers, factorise a composite number into its prime factors, determine whether a number is prime, and find the least common multiple or greatest common divisor of two numbers.

Usage

```
primefactors(n, method=c("C", "interpreted"))
divisors(n)
is.prime(n)
relatively.prime(n, m)
least.common.multiple(n,m)
greatest.common.divisor(n,m)
primesbelow(nmax)
```

Arguments

n,m	Integers to be factorized.
nmax	Integer. Upper limit on prime numbers to be found.
method	Character string indicating the choice of algorithm. (Developer use only.)

Details

`is.prime(n)` returns TRUE if `n` is a prime number, and FALSE otherwise.

`primefactors(n)` factorises the integer `n` into its prime number factors, and returns an integer vector containing these factors. Some factors may be repeated.

`divisors(n)` finds all the integers which divide the integer `n`, and returns them as a sorted vector of integers (beginning with 1 and ending with `n`).

`relatively.prime(n,m)` returns TRUE if the integers `n` and `m` are relatively prime, that is, if they have no common factors.

`least.common.multiple` and `greatest.common.divisor` return the least common multiple or greatest common divisor of two integers `n` and `m`.

`primesbelow(nmax)` returns an integer vector containing all the prime numbers less than or equal to `nmax`.

Value

`is.prime` and `relatively.prime` return a logical value.

`least.common.multiple` and `greatest.common.divisor` return a single integer.

`primefactors` and `primesbelow` return an integer vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
is.prime(17)

relatively.prime(2, 3)

primefactors(24) ## Note repeated factors

divisors(24)

greatest.common.divisor(60, 100)

primesbelow(20)
```

queueSpatstatLocator *Add Coordinates to a Queue for Use by Locator Function*

Description

Add the coordinates of a spatial location to a queue. The queue can be accessed by the `spatstatLocator` function in a non-interactive session.

Usage

```
queueSpatstatLocator(x, y)
```

Arguments

`x, y` Numeric values, or vectors of the same length, containing spatial coordinates. Any data acceptable to `xy.coords`.

Details

The `spatstatLocator` function is a replacement for the `locator` function that can be used to test software which depends on user input.

When `queueSpatstatLocator(x, y)` is called, the coordinate data `x, y` are saved in a queue. The first-listed coordinate pair `x[1], y[1]` is at the front of the queue. Subsequently, when `spatstatLocator` is called, the coordinates are taken from the front of the queue and returned as if they had been clicked by the user.

This only works in a **non**-interactive session, that is, when `interactive()` returns `FALSE`.

Value

Integer (invisible). The length of the queue, after inclusion of the new points.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz>
and Ege Rubak <rubak@math.aau.dk>.

See Also

[spatstatLocator](#)

Examples

```
queueSpatstatLocator(0.5, 0.7)
queueSpatstatLocator(c(0.3, 0.4), c(0.2, 0.9))
if(!interactive()) {
  spatstatLocator(2)
  spatstatLocator(1)
}
```

RelevantNA	<i>Missing Value, Zero-length Vector, or Zero Value of the Appropriate Type</i>
------------	---

Description

Given any data x , these functions return the missing value NA, the empty vector, or the equivalent of the number 0, with the same type as x .

Usage

```
RelevantZero(x)
RelevantNA(x)
RelevantEmpty(x)

isRelevantZero(x)
```

Arguments

x Data of any type.

Details

In the R system, missing values may have different types. For example, if an entry is missing from a numeric vector, it is a missing numeric value, not a missing logical value, and R distinguishes between these two types of missing values.

The function RelevantNA returns a missing value of the same type as the input x (as defined by [typeof](#)). Thus, RelevantNA(3.2) returns a missing numeric value and RelevantNA(TRUE) returns a missing logical value.

RelevantEmpty(x) returns a vector of length zero which has the same type as x. Thus, RelevantEmpty(TRUE) is equivalent to logical(0).

RelevantZero(x) returns a single value, of the same type as x, that is equivalent to the number zero. For example, RelevantZero(TRUE) returns FALSE.

The function isRelevantZero tests whether x is a single zero value, by testing whether x is identical to RelevantZero(x).

Value

RelevantZero and RelevantNA return a single value of the same type as x.

RelevantEmpty returns a zero-length vector of the same type as x.

isRelevantZero returns a single logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[typeof](#)

Examples

```
RelevantZero(42)
RelevantZero(TRUE)
RelevantZero("hello world")

RelevantNA(1:3)
typeof(RelevantNA(1:3))
typeof(RelevantNA("hello world"))
```

resolve.defaults

Determine Values of Variables Using Several Default Rules

Description

Determine the values of variables by applying several different default rules in a given order.

Usage

```
resolve.defaults(..., .MatchNull = TRUE, .StripNull = FALSE)

resolve.1.default(.A, ...)
```

Arguments

...	Several lists of name=value pairs.
.MatchNull	Logical value. If TRUE (the default), an entry of the form name=NULL will be treated as assigning the value NULL to the variable name. If FALSE, such entries will be ignored.
.StripNull	Logical value indicating whether entries of the form name=NULL should be removed from the result.
.A	Either a character string giving the name of the variable to be extracted, or a list consisting of one name=value pair giving the variable name and its fallback default value.

Details

These functions determine the values of variables by applying a series of default rules, in the order specified.

Each of the arguments ... should be a list of name=value pairs giving a value for a variable name. Each list could represent a set of arguments given by the user, or a rule assigning default values to some variables. Lists that appear earlier in the sequence of arguments ... take precedence.

The arguments ... will be concatenated into a single list. The earliest occurrence of each name is then used to determine the final value of the variable name.

The function resolve.defaults returns a list of name=value pairs for all variables encountered. It is commonly used to decide the values of arguments to be passed to another function using [do.call](#).

The function resolve.1.default returns the value of the specified variable as determined by resolve.defaults. It is commonly used inside a function to determine the value of an argument.

Value

The result of resolve.defaults is a list of name=value pairs.

The result of resolve.1.default can be any kind of value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

[do.call](#)

Examples

```
user <- list(day="Friday")
ruleA <- list(month="Jan", gravity=NULL)
ruleB <- list(day="Tuesday", month="May", gravity=42)
resolve.defaults(user, ruleA, ruleB)
resolve.defaults(user, ruleA, ruleB, .StripNull=TRUE)
resolve.defaults(user, ruleA, ruleB, .MatchNull=FALSE)

resolve.1.default("month", user, ruleA, ruleB)
```

`revcumsum`*Reverse Cumulative Sum*

Description

Returns a vector of cumulative sums of the input values, running in reverse order. That is, the i th entry in the output is the sum of entries i to n in the input, where n is the length of the input.

Usage

```
revcumsum(x)
```

Arguments

`x` A numeric or complex vector.

Details

This low-level utility function is a faster alternative to `rev(cumsum(rev(x)))` under certain conditions. It computes the reverse cumulative sum of the entries of `x`. If `y <- revcumsum(x)`, then `y[i] = sum(x[i:n])` where `n = length(x)`.

This function should not be used if `x` could contain NA values: this would lead to an error.

Value

A vector of the same length and type as `x`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

[cumsum](#).

Examples

```
revcumsum(1:5)
rev(cumsum(rev(1:5)))
x <- runif(1e6)
system.time(rev(cumsum(rev(x))))
system.time(revcumsum(x))
```

simplenumber	<i>Simple Rational Number</i>
--------------	-------------------------------

Description

Given a numeric value, try to express it as a simple rational number.

Usage

```
simplenumber(x, unit = "", multiply = "*", tol = .Machine$double.eps)
```

Arguments

x	A single numeric value.
unit	Optional. Character string giving the name of the unit in which x is expressed. Typically an irrational number such as pi. See Examples.
multiply	Optional. Character string representing multiplication.
tol	Numerical tolerance.

Details

The code tries to express x as an integer $x=n$, or as the reciprocal of an integer $x=1/n$, or as a simple rational number $x = m/n$, where m, n are small integers.

Value

A character string representing the simple number, or NULL if not successful.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
simplenumber(0.3)
simplenumber(0.33333333333333333333333333333333)
x <- pi * 2/3
simplenumber(x/pi, "pi")
```

spatstatLocator *Graphical Input*

Description

This is an alternative to the [locator](#) function. It contains a workaround for a bug that occurs in RStudio.

Usage

```
spatstatLocator(n, type = c("p", "l", "o", "n"), ...)
```

Arguments

n	Optional. Maximum number of points to locate.
type	Character specifying how to plot the locations. If "p" or "o" the points are plotted; if "l" or "o" they are joined by lines.
...	Additional graphics parameters used to plot the locations.

Details

This is a replacement/workaround for the [locator](#) function in some versions of **RStudio** which do not seem to recognise the option `type="p"`.

See [locator](#) for a description of the behaviour.

Value

A list containing components `x` and `y` which are vectors giving the coordinates of the identified points in the user coordinate system, i.e., the one specified by `par("usr")`.

Software Testing

Programmers may like to know that code which depends on `spatstatLocator` can be tested in a non-interactive session, if the coordinates are previously queued using [queueSpatstatLocator](#).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz> and Ege Rubak <rubak@math.aau.dk>.

See Also

[locator](#).
[queueSpatstatLocator](#)

Examples

```
if(interactive()) locator(1, type="p")
```

splat	<i>Print Text Within Margins</i>
-------	----------------------------------

Description

Prints a given character string or strings inside the text margin specified by `options("width")`. Indents the text if required.

Usage

```
splat(..., indent = 0)
```

Arguments

`...` Character strings, or other arguments acceptable to [paste](#).

`indent` Optional. Indentation of the text. Either an integer specifying the number of character positions by which the text should be indented, or a character string whose length determines the indentation.

Details

`splat` stands for ‘split cat’.

The command `splat(...)` is like `cat(paste(...))` except that the output will be split into lines that can be printed within the current text margin specified by `getOption("width")`.

The arguments `...` are first combined into a character vector using [paste](#). Then they are split into words separated by white space. A newline will be inserted whenever the next word does not fit in the available text area. (Words will not be broken, so the text margin could be exceeded if any word is longer than `getOption("width")`).

If any argument is a vector, each element of the vector is treated as a separate line. Existing newline characters in `...` are also respected.

Value

Null.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz> and Ege Rubak <rubak@math.aau.dk>.

Examples

```
op <- options(width=20)
splat("There is more than one way to skin a cat.")
splat("There is more than one", "way to skin a cat.", indent=5)

options(width=10)
```

```
splat("The value of pi is", pi)
splat("The value of pi is", signif(pi))
options(op)
```

taperoff

Taper Functions

Description

Computes a function that tapers smoothly from 0 to 1.

Usage

```
taperoff(x, zeropoint = 0, onepoint = 1,
         type = c("smooth", "cosine", "Gaussian"))
```

Arguments

x	Function argument. A number or a numeric vector.
zeropoint	Value of x that should return a function value of 0.
onepoint	Value of x that should return a function value of 1.
type	Character string (partially matched) specifying which taper function to use.

Details

A taper is a mathematical function that exhibits a gradual transition between the values 0 and 1.

By default, the function value $f(x)$ is equal to 0 if $x \leq 0$, is equal to 1 if $x \geq 1$, and lies between 0 and 1 when $0 < x < 1$.

If type="cosine", the function is the cosine taper $f(x) = (1 - \cos(\pi x))/2$.

If type="smooth" the function is the smooth partition of unity $f(x) = \theta(x)/(\theta(x) + \theta(1 - x))$ where $\theta(x) = \exp(-1/x)$.

If type="Gaussian" the function is the cumulative distribution function of the Gaussian (normal) distribution with mean 1/2 and standard deviation 1/6.

If zeropoint and onepoint are specified, then the function value is equal to 0 when $x=\text{zeropoint}$, equal to 1 when $x=\text{onepoint}$, and lies between 0 and 1 when x lies between zeropoint and onepoint.

Value

A numeric vector of the same length as x.

Author(s)

Adrian Baddeley

Examples

```
curve(taperoff(x, type="smooth"))
curve(taperoff(x, type="cosine"), add=TRUE, col="green")
curve(taperoff(x, type="Gaussian"), add=TRUE, col="blue")
```

tapplysum

Sum By Factor Level

Description

A faster equivalent of `tapply(FUN=sum)`.

Usage

```
tapplysum(x, flist, do.names = FALSE, na.rm = TRUE)
```

Arguments

<code>x</code>	Vector of numeric or complex values.
<code>flist</code>	A list of factors of the same length as <code>x</code> .
<code>do.names</code>	Logical value indicating whether to attach names to the result.
<code>na.rm</code>	Logical value indicating whether to remove NA values before computing the sums.

Details

This function is designed to be a faster alternative to the idiom `y <-tapply(x, flist, sum); y[is.na(y)] <-0`. The result `y` is a vector, matrix or array of dimension equal to the number of factors in `flist`. Each position in `y` represents one of the possible combinations of the factor levels. The resulting value in this position is the sum of all entries of `x` where the factors in `flist` take this particular combination of values. The sum is zero if this combination does not occur.

Currently this is implemented for the cases where `flist` has length 1, 2 or 3 (resulting in a vector, matrix or 3D array, respectively). For other cases we fall back on `tapply`.

Value

A numeric vector, matrix or array.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Tilman Davies.

See Also

[tapply](#), [table](#)

Examples

```
x <- 1:12
a <- factor(rep(LETTERS[1:2], each=6))
b <- factor(rep(letters[1:4], times=3))
ff <- list(a, b)
tapply(x, ff, sum)
tapplysum(x, ff, do.names=TRUE)
tapplysum(x + 2i, ff, do.names=TRUE)
```

termsinformula

Manipulate Formulae

Description

Operations for manipulating formulae.

Usage

```
termsinformula(x)
variablesinformula(x)
offsetsinformula(x)
lhs.of.formula(x)
rhs.of.formula(x, tilde=TRUE)
lhs.of.formula(x) <- value
rhs.of.formula(x) <- value
can.be.formula(x)
identical.formulae(x,y)
```

Arguments

<code>x,y</code>	Formulae, or character strings representing formulae.
<code>tilde</code>	Logical value indicating whether to retain the tilde.
<code>value</code>	Symbol or expression in the R language. See Examples.

Details

`variablesinformula(x)` returns a character vector of the names of all variables which appear in the formula `x`.

`termsinformula(x)` returns a character vector of all terms in the formula `x` (after expansion of interaction terms).

`offsetsinformula(x)` returns a character vector of all offset terms in the formula.

`rhs.of.formula(x)` returns the right-hand side of the formula as another formula (that is, it removes the left-hand side) provided `tilde=TRUE` (the default). If `tilde=FALSE`, then the right-hand side is returned as a language object.

`lhs.of.formula(x)` returns the left-hand side of the formula as a symbol or language object, or `NULL` if the formula has no left-hand side.

`lhs.of.formula(x) <-value` and `rhs.of.formula(x) <-value` change the formula `x` by replacing the left or right hand side of the formula by `value`.

`can.be.formula(x)` returns TRUE if `x` is a formula or a character string that can be parsed as a formula, and returns FALSE otherwise.

`identical.formulae(x,y)` returns TRUE if `x` and `y` are identical formulae (ignoring their environments).

Value

`variablesinformula`, `termsinformula` and `offsetsinformula` return a character vector.

`rhs.of.formula` returns a formula. `lhs.of.formula` returns a symbol or language object, or NULL.

`can.be.formula` and `identical.formulae` return a logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <r.turner@auckland.ac.nz> and Ege Rubak <rubak@math.aau.dk>.

Examples

```
f <- (y ~ x + z*w + offset(h))
lhs.of.formula(f)
rhs.of.formula(f)
variablesinformula(f)
termsinformula(f)
offsetsinformula(f)
g <- f
environment(g) <- new.env()
identical(f,g)
identical.formulae(f,g)
lhs.of.formula(f) <- quote(mork) # or as.name("mork")
f
rhs.of.formula(f) <- quote(x+y+z) # or parse(text="x+y+z")[[1]]
f
```

verbalogic

Verbal Logic

Description

Perform the specified logical operation on the character vector `x`, recognising the special strings "TRUE" and "FALSE" and treating other strings as logical variables.

Usage

```
verbalogic(x, op = "and")
```

Arguments

x	Character vector.
op	Logical operation: one of the character strings "and", "or" or "not".

Details

This function performs simple logical operations on character strings that represent human-readable statements.

The character vector `x` may contain any strings: the special strings "TRUE" and "FALSE" are treated as the logical values TRUE and FALSE, while all other strings are treated as if they were logical variables.

If `op="and"`, the result is a single string, logically equivalent to `x[1] && x[2] && ... && x[n]`. First, any entries of `x` equal to "TRUE" are removed. The result is "FALSE" if any of the entries of `x` is "FALSE"; otherwise it is equivalent to `paste(x, collapse=" and ")`.

If `op="or"`, the result is a single string, logically equivalent to `x[1] || x[2] || ... || x[n]`. First, any entries of `x` equal to "FALSE" are removed. The result is "TRUE" if any of the entries of `x` is "TRUE"; otherwise it is equivalent to `paste(x, collapse=" or ")`.

If `op="not"`, the result is a character vector `y` such that `y[i]` is the logical negation of `x[i]`.

The code does not understand English grammar and cannot expand logical expressions.

Value

A character string.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

Examples

```
x <- c("The sky is blue", "my name is not Einstein", "FALSE")
verbalogic(x, "and")
verbalogic(x, "or")
verbalogic(x, "not")
```


Index

- * **arith**
 - revcumsum, 32
 - tapplysum, 37
- * **classes**
 - check.1.integer, 6
- * **error**
 - check.1.integer, 6
 - check.anyvector, 7
 - check.named.vector, 8
 - check.nmatrix, 9
 - check.nvector, 11
- * **iplot**
 - queueSpatstatLocator, 28
 - spatstatLocator, 34
- * **logic**
 - verbalogic, 39
- * **manip**
 - articlebeforenumber, 4
 - cat.factor, 5
 - commasep, 13
 - exactCutBreaks, 16
 - ifelseAB, 18
 - ordinal, 24
 - paren, 25
 - percentage, 26
 - RelevantNA, 29
 - verbalogic, 39
- * **math**
 - methods.xypolygon, 20
 - orderstats, 23
 - primefactors, 27
 - taperoff, 36
- * **optimize**
 - optimizeWithTrace, 22
- * **package**
 - spatstat.utils-package, 2
- * **print**
 - splat, 35
- * **programming**
 - check.range, 12
 - do.call.matched, 14
 - do.call.without, 15
 - resolve.defaults, 30
- * **spatial**
 - spatstat.utils-package, 2
- * **symbolmath**
 - simplenumber, 33
- * **utilities**
 - articlebeforenumber, 4
 - check.anyvector, 7
 - check.nmatrix, 9
 - check.nvector, 11
 - check.range, 12
 - commasep, 13
 - do.call.matched, 14
 - do.call.without, 15
 - expand.polynom, 17
 - ifelseAB, 18
 - ordinal, 24
 - paren, 25
 - percentage, 26
 - resolve.defaults, 30
 - revcumsum, 32
 - splat, 35
 - tapplysum, 37
 - termsinformula, 38
- Area.xypolygon (methods.xypolygon), 20
- articlebeforenumber, 3, 4, 25
- bdrylength.xypolygon (methods.xypolygon), 20
- c, 5
- can.be.formula, 3
- can.be.formula (termsinformula), 38
- cat, 5
- cat.factor, 3, 5
- check.1.integer, 3, 6, 9

- check.1.real, [11](#)
- check.1.real (check.1.integer), [6](#)
- check.1.string (check.1.integer), [6](#)
- check.anyvector, [7](#), [11](#)
- check.in.range, [3](#)
- check.in.range (check.range), [12](#)
- check.named.list (check.named.vector), [8](#)
- check.named.thing (check.named.vector), [8](#)
- check.named.vector, [3](#), [6](#), [7](#), [8](#), [11](#)
- check.nmatrix, [9](#), [11](#)
- check.nvector, [3](#), [7](#), [10](#), [11](#)
- check.range, [3](#), [12](#)
- commasep, [3](#), [13](#), [25](#)
- cumsum, [32](#)
- cut.default, [17](#)
- divisors (primefactors), [27](#)
- do.call, [15](#), [31](#)
- do.call.matched, [3](#), [14](#), [16](#)
- do.call.without, [3](#), [15](#), [15](#)
- exactCutBreaks, [16](#)
- expand.polynom, [3](#), [17](#)
- fave.order, [3](#)
- getOption, [3](#), [35](#)
- greatest.common.divisor, [2](#)
- greatest.common.divisor (primefactors), [27](#)
- identical.formulae, [3](#)
- identical.formulae (termsinformula), [38](#)
- ifelse, [19](#)
- ifelse0NA (ifelseAB), [18](#)
- ifelse1NA (ifelseAB), [18](#)
- ifelseAB, [3](#), [18](#)
- ifelseAX (ifelseAB), [18](#)
- ifelseNegPos (ifelseAB), [18](#)
- ifelseXB (ifelseAB), [18](#)
- ifelseXY (ifelseAB), [18](#)
- inside.range, [3](#)
- inside.range (check.range), [12](#)
- inside.xypolygon (methods.xypolygon), [20](#)
- interactive, [28](#)
- intersect.ranges, [3](#)
- intersect.ranges (check.range), [12](#)
- is.hole.xypolygon (methods.xypolygon), [20](#)
- is.prime, [2](#)
- is.prime (primefactors), [27](#)
- isRelevantZero (RelevantNA), [29](#)
- least.common.multiple, [2](#)
- least.common.multiple (primefactors), [27](#)
- lhs.of.formula, [3](#)
- lhs.of.formula (termsinformula), [38](#)
- lhs.of.formula<- (termsinformula), [38](#)
- locator, [3](#), [28](#), [34](#)
- methods.xypolygon, [20](#)
- offsetsinformula, [3](#)
- offsetsinformula (termsinformula), [38](#)
- optimize, [3](#), [22](#)
- optimizeWithTrace, [3](#), [22](#)
- order, [23](#), [24](#)
- orderstats, [23](#)
- orderwhich (orderstats), [23](#)
- ordinal, [3](#), [4](#), [24](#)
- ordinalsuffix, [3](#)
- ordinalsuffix (ordinal), [24](#)
- overlap.xypolygon (methods.xypolygon), [20](#)
- paren, [3](#), [25](#)
- paste, [35](#)
- percentage, [26](#)
- polynom, [18](#)
- prange, [3](#)
- prange (check.range), [12](#)
- primefactors, [2](#), [27](#)
- primesbelow, [2](#)
- primesbelow (primefactors), [27](#)
- queueSpatstatLocator, [28](#), [34](#)
- relatively.prime, [2](#)
- relatively.prime (primefactors), [27](#)
- RelevantEmpty (RelevantNA), [29](#)
- RelevantNA, [29](#)
- RelevantZero (RelevantNA), [29](#)
- resolve.1.default (resolve.defaults), [30](#)
- resolve.defaults, [3](#), [15](#), [30](#)
- rev, [32](#)
- revcumsum, [3](#), [32](#)
- reverse.xypolygon (methods.xypolygon), [20](#)
- rhs.of.formula, [3](#)

rhs.of.formula (termsinformula), 38
rhs.of.formula<- (termsinformula), 38

simplenumber, 33
simplify.xypolygon (methods.xypolygon),
20
sort, 23, 24
spatstat.utils
 (spatstat.utils-package), 2
spatstat.utils-package, 2
spatstatLocator, 3, 28, 29, 34
splat, 3, 35
sympoly, 3
sympoly (expand.polynom), 17

table, 37
taperoff, 36
tapply, 37
tapplysum, 3, 37
termsinformula, 3, 38
typeof, 29, 30

unparen, 3
unparen (paren), 25

variablesinformula, 3
variablesinformula (termsinformula), 38
verbalogic, 39
verify.xypolygon (methods.xypolygon), 20

xy.coords, 28