Package 'omnibus'

May 16, 2024

Type Package

Title Helper Tools for Managing Data, Dates, Missing Values, and Text

Version 1.2.13

Date 2024-05-16

Maintainer Adam B. Smith <adam.smith@mobot.org>

Description An assortment of helper functions for managing data (e.g., rotating values in matrices by a user-defined angle, switching from row- to column-indexing), dates (e.g., intuiting year from messy date strings), handling missing values (e.g., removing elements/rows across multiple vectors or matrices if any have an NA), text (e.g., flushing reports to the console in real-time); and combining data frames with different schema (copying, filling, or concatenating columns or applying functions before combining).

Depends R (>= 3.5.0)

Imports

Suggests

License GPL (>= 3)

LazyData true

LazyLoad yes

URL https://github.com/adamlilith/omnibus

BugReports https://github.com/adamlilith/omnibus

Encoding UTF-8

RoxygenNote 7.3.1

NeedsCompilation no

Author Adam B. Smith [cre, aut] (<https://orcid.org/0000-0002-6420-1659>)

Repository CRAN

Date/Publication 2024-05-16 17:00:06 UTC

R topics documented:

`appendLists` *Append values to elements of a list from another list*

Description

This function "adds" two lists with the same or different names together. For example, if one list is as `l1 <- list(a=1, b="XYZ")` and the second is as `l2 <- list(a=3, c=FALSE)`, the output will be as `list(a = c(1, 3), b = "XYZ", c = FALSE)`. All elements in each list must have names.

Usage

```
appendLists(...)
```

Arguments

`...`	Two or more lists. All elements must have names.

Details

If two lists share the same name and these elements have the same class, then they will be merged as-is. If the classes are different, one of them will be coerced to the other (see *Examples*). The output will have elements with the names of all lists.

Value

A `list`.

See Also

`mergeLists`

Examples

```
# same data types for same named element
l1 <- list(a=1, b="XYZ")
l2 <- list(a=3, c=FALSE)
appendLists(l1, l2)

# different data types for same named element
l1 <- list(a=3, b="XYZ")
l2 <- list(a="letters", c=FALSE)
appendLists(l1, l2)
```

bracket *Identify values bracketing another value*

Description

This function takes an ordered vector of numeric or character values and finds the pair that bracket a third value, x. If x is exactly equal to one of the values in the vector, then a single value equal to x is returned. If x falls outside of the range of the vector, then the least/most extreme value of the vector is returned (depending on which side of the distribution of the vector x resides). Optionally, users can have the function return the index of the values that bracket x.

Usage

```
bracket(x, by, index = FALSE, inner = TRUE, warn = FALSE)
```

Arguments

x	A numeric or character vector.
by	A numeric or character vector. These should be sorted (from high to low or low to high... if not, an error will result).
index	Logical. If FALSE (default), then numeric values in by will be returned. If TRUE, then the index or indices of the bracketing value(s) will be returned.
inner	Logical. If TRUE (default), then if x is surrounded by at least one series of repeating values, return the values (or indices) among the repeated sequence(s) closest to the value of x. If FALSE, return the value(s) (or indices) among the repeated sequence(s) farthest from the value of x. For example, if index = TRUE, by = c(1, 2, 2, 2, 3, 3), and x = 2.5, setting inner = TRUE will return the index of the third 2 and first 3. If inner = FALSE, then the function returns the index of the first 2 and second 3.
warn	Logical. If TRUE, then warn if x is outside the range of by.

Value

If x is a single value, then the function will return a numeric vector of length 1 or 2, depending on how many values bracket x. If all values of by are the same, then the median index (or value) of by is returned. If x is a vector, then the result will be a list with one element per item in x with each element having the same format as the case when x is a single value.

Examples

```
by <- 2 * (1:5)
bracket(4.2, by)
bracket(6.8, by)

bracket(3.2, by, index=TRUE)
bracket(c(3.2, 9.8, 4), by)
```

```
bracket(2, c(0, 1, 1, 1, 3, 5), index=TRUE)
bracket(3, c(1, 2, 10))

bracket(2.5, c(1, 2, 2, 2, 3, 3), index=TRUE)
bracket(2.5, c(1, 2, 2, 2, 3, 3), index=TRUE, inner=FALSE)
bracket(2.9, c(1, 2, 2, 2, 3, 3), index=TRUE)
bracket(2.9, c(1, 2, 2, 2, 3, 3), index=TRUE, inner=FALSE)


by <- 1:10
bracket(-100, by)
bracket(100, by)
```

`capIt` *Capitalize first letter of a string*

Description

Capitalize the first letter of a string or the first letters of a list of strings.

Usage

```
capIt(x)
```

Arguments

`x`	Character or character vector.

Value

Character or character vector.

See Also

`toupper`, `tolower`,

Examples

```
x <- c('shots', 'were', 'exchanged at the ', 'hospital.')
capIt(x)
```

combineDf *Combine data frames with different fields using a crosswalk table*

Description

This function combines multiple "source" data frames, possibly with different column names, into a single "destination" data frame. Usually `merge` will be faster and easier to implement if the columns to be merged on have the same names, and `rbind` will always be faster and much easier if the column names and data types match exactly.

The key tool in this function is a "crosswalk" table (a `data.frame`) that tells the function which fields in each source data frame match to the final fields in the destination data frame. Values in source data frame fields can be used as-is, combined across fields, or have functions applied to them before they are put into the destination data frame. If a source data frame doe snot gave a field that matches the destination field, a default value (including `NA`) can be assigned to all cells for that source data frame.

The data frames to be combined can be provided in `...` or as file names in the first column of the crosswalk table. These can be either CSV files (extension "`.csv`"), TAB files (extension "`.tab`"), "Rdata" files (read using `load` and with a "`.rda`" or "`.rdata`" extension), or "RDS" files (read using `readRDS` and with a "`.rds`" extension). The file type will be intuited from the extension, and its case does not matter. Note that if an object in an Rdata file has the same name as an object in this function (i.e., any of the arguments plus any objects internal to the function), this may cause a conflict. To help obviate this issue, all internal objects are named with a period at the end (e.g., "`crossCell.`" and "`countDf.`").

All cells in each source data frame will have leading and trailing white spaces removed before combining.

Usage

```
combineDf(
  ...,
  crosswalk,
  collapse = "; ",
  useColumns = NULL,
  excludeColumns = NULL,
  useFrames = NULL,
  classes = NULL,
  verbose = FALSE
)
```

Arguments

`...`	Data frames to combine. These *must* be listed in the order that they appear in the `crosswalk` table.

crosswalk A data.frame. Column names are fields in the destination data frame. Each row corresponds to a different data frame to join. If ... is not used then the first column *must* have the paths and file names to CSV, TAB, Rdata, or RDS files representing data frames to join. All objects will be coerced to data.frames.

Other than this column, the elements of each cell contain the name of the column in each source data frame that coincides with the column name in the crosswalk table. For example, suppose the destination data frame is to have a column by the name of "species" with names of species within it. If the first source data frame has a column named "Species" and the second source data frame has a column named "scientificName", then the first value in crosswalk under its "species" column will be "Species" and the second "scientificName". This will take all the values in the "Species" field of the first source data frame and all the values in the "scientificName" field in the second source data frame and put them both into the "species" field of the destination frame.

More complex operations can be done using the following in cells of crosswalk:

- Filling all cells with a single value: If the value in the crosswalk cell begins with "%fill%", then the value that follows it will be repeated in destination data frame in each row. For example,

  ```
  %fill% inspected
  ```

  will repeat the string "inspected" in every row of the output corresponding to the respective source data frame (any spaces immediately after %fill% are ignored).
- Concatenating (pasting) columns together: To combine multiple fields, begin crosswalk cell with "%cat%", then list the fields to combine (with or without commas separating them). For example, if the crosswalk cell has

  ```
  %cat% field1 field2 field3
  ```

  then the resulting column in the destination data frame will have values from field1, field2, and field3 pasted together. See also argument collapse.
- Applying a function: You can manipulate values by including functions in the crosswalk cell. The crosswalk cell should begin with "%fun%", then be followed an expression to evaluate. Expressions should generally *not* use the "<-" operator (or the equals operator used in the same way). For example:

  ```
  %fun% ifelse(as.numeric(field1) >20, NA, as.numeric(field1))
  ```

  will create a column that is NA whenever values in field1 are >20, and the value in field1 otherwise. Note that for mathematical operations, it is almost always necessary to use as.numeric around column names representing numbers since fields are read in as characters.

collapse Character, specifies the string to put between fields combined with the %cat%

operator in the crosswalk table.

useColumns, excludeColumns
: Logical, character vector, integer vector, or NULL. Indicates which columns in the crosswalk table are to be used or not to be used. These can be given as a TRUE/FALSE vector, a vector of column names, or a vector of column index values (integers). You can only specify useColumns or excludeColumns at a time (one or both must be NULL). If both are NULL (default), then all columns in the crosswalk will be used.

useFrames
: Logical, character, or NULL. Indicates if a particular source data frame should be used. This should be a TRUE/FALSE vector or the name of a column in the crosswalk table that has TRUE/FALSE values. If this is the name of a column, the column will be removed from the columns in useColumns. If NULL (default). then all data frames in the crosswalk will be used.

classes
: Character or character list, specifies the classes (e.g., character, logical, numeric, integer) to be assigned to each column in the output table. If NULL, all classes will be assumed to be character. If just one value is listed, all columns will be set to this class. If a list, it must be the same length as the number of columns in crosswalk and specify the class of each column. If it has names, then the names must correspond to the columns in crosswalk and will be used to assign the data type to the corresponding columns.

verbose
: Logical, if TRUE prints extra information during execution. Useful for debugging the crosswalk table.

Value

A data frame.

See Also

merge, rbind

Examples

```
df1 <- data.frame(x1=1:5, x2=FALSE, x3=letters[1:5], x4=LETTERS[1:5],
       x5='stuff', x6=11:15)
df2 <- data.frame(y1=11:15, y2=rev(letters)[1:5], y3=runif(5))

crosswalk <- data.frame(
  a = c('x1', 'y1'),
  b = c('x2', '%fill% TRUE'),
  c = c('%cat% x3 x4', 'y2'),
  d = c('x5', '%fill% NA'),
 e = c('%fun% as.numeric(x6) > 12', '%fun% round(as.numeric(y3), 2)')
)

combined <- combineDf(df1, df2, crosswalk=crosswalk)
combined
```

compareFloat *Compare values to floating-point precision*

Description

These functions compare values while accounting for differences in floating point precision.

Usage

```
compareFloat(x, y, op, tol = .Machine$double.eps^0.5)

x %<% y

x %<=% y

x %==% y

x %>=% y

x %>% y

x %!=% y
```

Arguments

x, y	Numeric
op	Operator for comparison (must be in quotes): "<", ">", "<=", ">=", "==", or "!="
tol	Tolerance value: The largest absolute difference between x and y that is to be considered equality. The default is .Machine$double.eps^0.5.

Value

TRUE, FALSE, or NA

Examples

```
x <- 0.9 - 0.8
y <- 0.8 - 0.7

x < y
x %<% y
compareFloat(x, y, "<")

x <= y
x %<=% y
compareFloat(x, y, "<=")
```

```
x == y
x %==% y
compareFloat(x, y, "==")

y > x
y %>% x
compareFloat(y, x, ">")

y >= x
y %>=% x
compareFloat(y, x, ">=")

x != y
x %!=% y
compareFloat(x, y, "!=")
```

`conversionFactors` *Data frame or conversion factors for length or areal units*

Description

Data frame of conversion factors for length or areal units.

Usage

```
conversionFactors
```

Format

An object of class `'data.frame'`.

See Also

`convertUnits`, `expandUnits`

Examples

```
data(conversionFactors)
conversionFactors
```

convertUnits *Convert length or areal units*

Description

This function converts length and area values from one unit to another (e.g., meters to miles, or square yards to acres). Alternatively, it provides the conversion factor for changing one unit to another.

Usage

```
convertUnits(from = NULL, to = NULL, x = NULL)
```

Arguments

from, to	Character: Names of the units to convert from/to. Partial matching is used, and case is ignored. Valid values are listed below. The '*2' values represent areas (e.g., 'm2' is "meters-squared"). • 'm' or 'meters' • 'm2' or 'meters2' • 'km' or 'kilometers' • 'km2' or 'kilometers2' • 'mi' or 'miles' • 'mi2' or 'miles2' • 'ft' or 'feet' • 'ft2' or 'feet2' • 'yd' or 'yards' • 'yd2' or 'yards2' • 'ac' or 'acres' • 'ha' or 'hectares' • 'nmi' or 'nautical miles' • 'nmi2' or 'nautical miles2'
x	Numeric or NULL (default). The value(s) to convert in the unit specified by from. If left as NULL, the conversion factor is reported instead.

See Also

expandUnits, conversionFactors

Examples

```
# conversion
convertUnits(from = 'm', to = 'km', 250)
convertUnits(from = 'm', to = 'mi', 250)
```

```
convertUnits(from = 'm2', to = 'km2', 250)

# conversion factors
convertUnits(from = 'm', to = 'km')
convertUnits(from = 'm')
convertUnits(to = 'm')
```

---

corner *Corner of a matrix or data frame*

Description

Return a corner of a matrix or data frame (i.e., upper left, upper right, lower left, lower right).

Usage

```
corner(x, corner = 1, size = 5)
```

Arguments

x	Data frame or matrix.
corner	Integer in the set 1, 2, 3, 4 or character in the set 'topleft', 'topright', 'bottomleft', 'bottomright' or in the set 'tl', 'tr', 'bl', 'br'. Indicates which corner to return. Integers 1, 2, 3 and 4 correspond to top left, top right, bottom left, and bottom right corners. The default is 1, the top left corner.
size	Positive integer, number of rows and columns to return. If there are fewer columns/rows than indicated then all columns/rows are returned.

Value

A matrix or data frame.

See Also

head, tail

Examples

```
x <- matrix(1:120, ncol=12, nrow=10)
x
corner(x, 1)
corner(x, 2)
corner(x, 3)
corner(x, 4)
```

countDecDigits *Number of digits after a decimal place*

Description

Count the number of digits after a decimal place. Note that trailing zeros will likely be ignored.

Usage

```
countDecDigits(x)
```

Arguments

x	Numeric or numeric vector.

Value

Integer.

Examples

```
countDecDigits(c(1, 1.1, 1.12, 1.123, 1.1234, -1, 0, 10.0000, 10.0010))
```

cull *Force objects to have length or rows equal to the shortest*

Description

This function takes a set of vectors, data frames, or matrices and removes the last values/rows so that they all have a length/number of rows equal to the shortest among them.

Usage

```
cull(...)
```

Arguments

...	Vectors, matrices, or data frames.

Value

A list with one element per object supplied as an argument to the function.

Examples

```
a <- 1:10
b <- 1:20
c <- letters
cull(a, b, c)
x <- data.frame(x=1:10, y=letters[1:10])
y <- data.frame(x=1:26, y=letters)
cull(x, y)
```

`dirCreate` *Replacement for dir.create()*

Description

This function is a somewhat friendlier version of `dir.create` in that it automatically sets `recursive=TRUE` and `showWarnings=FALSE` arguments.

Usage

```
dirCreate(...)
```

Arguments

`...`	Character string(s). The path and name of the directory to create. Multiple strings will be pasted together into one path, although slashes will not be pasted between them.

Value

Nothing (creates a directory on the storage system).

See Also

`dir.create`

`domLeap` *Day of month for leap years*

Description

Data frame of day of month for each month in a leap year.

Usage

```
domLeap
```

Format

An object of class 'data.frame'.

Examples

```
data(domLeap)
domLeap
```

---

domNonLeap *Day of month for non-leap years*

Description

Data frame of day of month for each month in a non-leap year.

Usage

```
domNonLeap
```

Format

An object of class 'data.frame'.

Examples

```
data(domNonLeap)
domNonLeap
```

---

doyLeap *Day of year for leap years*

Description

Data frame of day of year for each month in a leap year.

Usage

```
doyLeap
```

Format

An object of class 'data.frame'.

Examples

```
data(doyLeap)
doyLeap
```

doyNonLeap *Days of year for non-leap years*

Description

Data frame of days of year for each month in a non-leap year

Usage

```
doyNonLeap
```

Format

An object of class `'data.frame'`.

Examples

```
data(doyNonLeap)
doyNonLeap
```

ellipseNames *Get names of objects input as arguments in ellipse (. . .) form*

Description

This function returns the names of objects input into a function as ellipses. It is only useful if called inside a function.

Usage

```
ellipseNames(...)
```

Arguments

...	Objects.

Value

Character list.

eps *The smallest machine-readable number*

Description

This function returns the smallest machine-readable number (equal to `.Machine$double.eps`).

Usage

```
eps()
```

Value

Numeric value.

Examples

```
eps()
```

expandUnits *Convert unit abbreviations to proper unit names*

Description

This function converts abbreviations of length and area units (e.g., "m", "km", and "ha") to their proper names (e.g., "meters", "kilometers", "hectares"). Square areal units are specified using an appended "2", where appropriate (e.g., "m2" means "meters-squared" and will be converted to "meters2").

Usage

```
expandUnits(x)
```

Arguments

x	Character: Abbreviations to convert. Case is ignored. • `'m'` will be converted to `'meters'` • `'m2'` will be converted to `'meters2'` • `'km'` will be converted to `'kilometers'` • `'km2'` will be converted to `'kilometers2'` • `'mi'` will be converted to `'miles'` • `'mi2'` will be converted to `'miles2'` • `'ft'` will be converted to `'feet'` • `'ft2'` will be converted to `'feet2'`

- 'yd' will be converted to 'yards'
- 'yd2' will be converted to 'yards2'
- 'ac' will be converted to 'acres'
- 'ha' will be converted to 'hectares'
- 'nmi' will be converted to 'nautical miles'
- 'nmi2' will be converted to 'nautical miles2'

See Also

convertUnits, conversionFactors

Examples

```
expandUnits(c('m', 'm2', 'ac', 'nm2'))
```

---

forwardSlash *Replace backslash with forward slash*

Description

This function is helpful for Windows systems, where paths are usually expressed with left slashes, whereas R requires right slashes.

Usage

```
forwardSlash(x)
```

Arguments

x	A string.

Value

Character.

Examples

```
forwardSlash("C:\\ecology\\main project")
```

`insert` *Insert values into a vector*

Description

This function inserts values into a vector, lengthening the overall vector. It is different from, say, `x[1:3] <- c('a', 'b', 'c')` which simply replaces the values at indices 1 through 3.

Usage

```
insert(x, into, at, warn = TRUE)
```

Arguments

`x`	Vector of numeric, integer, character, or other values of the class of `x` to be inserted.
`into`	Vector of values into which to insert `x`.
`at`	Vector of positions (indices) where `x` should be inserted. If the length of `x` is shorter than the length of `at`, then values in `x` will be recycled and a warning produced.
`warn`	If `TRUE`, provide warnings.

Value

Vector.

See Also

`insertCol`, `insertRow`

Examples

```
x <- -1:-3
into <- 10:20
at <- c(1, 3, 14)
insert(x, into, at)

insert(-1, into, at)
```

insertCol *Insert a column or row into a data frame or matrix*

Description

This function inserts one or more columns or rows before or after another column or row in a data frame or matrix. It is similar to cbind except that the inserted column(s)/row(s) can be placed anywhere.

Usage

```
insertCol(x, into, at = NULL, before = TRUE)

insertRow(x, into, at = NULL, before = TRUE)
```

Arguments

x	Data frame, matrix, or vector with same number of columns or rows or elements as into.
into	Data frame or matrix into which x is to be inserted.
at	Character, integer, or NULL. Name of column or column number or name of row or row number at which to do insertion. If NULL (default), the result is exactly the same as cbind(into, x except that it retains row numbers or column names from into.
before	Logical, if TRUE (default) then the insertion will occur in front of the column or row named in at, if FALSE then after. Ignored if at is NULL.

Value

A data frame.

Functions

- insertRow(): Insert a column or row into a data frame or matrix

See Also

merge, cbind, insert

Examples

```
x <- data.frame(y1=11:15, y2=rev(letters)[1:5])
into <- data.frame(x1=1:5, x2='valid', x3=letters[1:5], x4=LETTERS[1:5], x5='stuff')

insertCol(x, into=into, at='x3')
insertCol(x, into=into, at='x3', before=FALSE)
```

```
insertCol(x, into)

x <- data.frame(x1=1:3, x2=LETTERS[1:3])
into <- data.frame(x1=11:15, x2='valid')
row.names(into) <- letters[1:5]

insertRow(x, into=into, at='b')
insertRow(x, into=into, at='b', before=FALSE)
insertRow(x, into)
```

`is.wholeNumber` *Test if a numeric value is a whole number*

Description

Sometimes numeric values can appear to be whole numbers but are actually represented in the computer as floating-point values. In these cases, simple inspection of a value will not tell you if it is a whole number or not. This function tests if a number is "close enough" to an integer to be a whole number. Note that `is.integer` will indicate if a value is of *class* integer (which if it is, will always be a whole number), but objects of class `numeric` will not evaluate to `TRUE` even if they are "supposed" to represent integers.

Usage

```
is.wholeNumber(x, tol = .Machine$double.eps^0.5)
```

Arguments

`x`	A numeric or integer vector.
`tol`	Largest absolute difference between a value and its integer representation for it to be considered a whole number.

Value

A logical vector.

See Also

`is.integer`

Examples

```
x <- c(4, 12 / 3, 21, 21.1)
is.wholeNumber(x)
```

isLeapYear *Is a year a leap year?*

Description

Returns TRUE if the year is a leap year. You can use "negative" years for BCE.

Usage

```
isLeapYear(x)
```

Arguments

x	Integer or vector of integers representing years.

Value

Vector of logical values.

Examples

```
isLeapYear(1990:2004) # note 2000 *was* not a leap year
isLeapYear(1896:1904) # 1900 was *not* a leap year
```

isTRUENA *Vectorized test for truth robust to NA*

Description

These functions work exactly the same as x == TRUE and x == FALSE but by default return FALSE for cases that are NA.

Usage

```
isTRUENA(x, ifNA = FALSE)

isFALSENA(x, ifNA = FALSE)
```

Arguments

x	Logical, or a condition that evaluates to logical, or a vector of logical values or conditions to evaluate.
ifNA	Logical, value to return if the result of evaluating x is NA. Note that this can be anything (i.e., TRUE, FALSE, a number, etc.).

Value

Logical or value specified in `ifNA`.

Functions

- `isFALSENA()`: Vectorized test for truth robust to NA

See Also

`isTRUE`, `isFALSE`, `TRUE`, `logical`

Examples

```
x <- c(TRUE, TRUE, FALSE, NA)
x == TRUE
isTRUENA(x)
x == FALSE
isFALSENA(x)
isTRUENA(x, ifNA = Inf)
# note that isTRUE and isFALSE are not vectorized
isTRUE(x)
isFALSE(x)
```

---

listFiles *Replacement for* `list.files()`

Description

This function is a slightly friendlier version of `list.files` in that it automatically includes the `full.names=TRUE` argument.

Usage

```
listFiles(x, ...)
```

Arguments

`x`	Path name of folder containing files to list.
`...`	Arguments to pass to `list.files` (other than `full.names`).

Value

Character.

See Also

`list.files`

Examples

```
# list files in location where R is installed
listFiles(R.home())
listFiles(R.home(), pattern='README')
```

`longRun` *Length of the longest run of a particular value in a numeric vector*

Description

This function returns the lengh of the longest run of a particular numeric value in a numeric vector. A "run" is an uninterrupted sequence of the same number. Runs can be "wrapped" so that if the sequence starts and ends with the target value then it is considered as a consecutive run.

Usage

```
longRun(x, val, wrap = FALSE, na.rm = FALSE)
```

Arguments

`x`	Numeric vector.
`val`	Numeric. Value of the elements of `x` of which to calculate length of longest run.
`wrap`	Logical. If `TRUE` then runs can "wrap" from the end of `x` to the start of `x` if the first and last elements of `x` are equal to `val`.
`na.rm`	Logical. If `TRUE` then remove `NA`s first.

Value

Integer.

See Also

[base::rle()]

Examples

```
x <- c(1, 1, 1, 2, 2, 3, 4, 5, 6, 1, 1, 1, 1, 1)
longRun(x, 2)
longRun(x, 1)
longRun(x, 1, wrap=TRUE)
```

maxRuns *Maximum number of continuous "runs" of values meeting a particular condition*

Description

Consider an ordered set of values, say 0, 4, 0, 0, 0, 2, 0, 10. We can ask, "What is the number of times in which zeros appear successively?" This function can answer this question and similar ones. What is considered a "run" is defined by a user-supplied function that must have a TRUE/FALSE output. For example, a "run" could be any succession of values less than two, in which case the criterion function would be `function(x) < 2`, or any succession of values not equal to 0, in which case the function would be `function(x) x != 0`.

Usage

```
maxRuns(x, fx, args = NULL, failIfAllNA = FALSE)
```

Arguments

x	Vector of numeric, character, or other values.
fx	A function that returns TRUE, FALSE, or (optionally) NA. The function must use x as its first argument. For example, `function(x) x == 0` is allowable, but `function(y) y == 0` is not. Values that count as TRUE will be counted toward a run.
args	A *list* object with additional arguments to supply to the function fx.
failIfAllNA	If TRUE, fail if all values are NA after being evaluated by fx.

Value

Lengths of successive runs of elements that meet the criterion. A single value of 0 indicates no conditions meet the criterion.

Examples

```
x <- c(1, 4, 0, 0, 0, 2, 0, 10)
fx <- function(x) x == 0
maxRuns(x, fx)

fx <- function(x) x > 0
maxRuns(x, fx)

fx <- function(x) x > 0 & x < 5
maxRuns(x, fx)

x <- c(1, 4, 0, 0, 0, 2, 0, 10)
fx <- function(x, th) x == th
maxRuns(x, fx, args=list(th=0))
```

```
# "count" NA as an observation
x <- c(1, 4, 0, 0, 0, NA, 0, 10)
fx <- function(x, th) ifelse(is.na(x), FALSE, x == th)
maxRuns(x, fx, args=list(th=0))

# include NAs as part of a run
x <- c(1, 4, 0, 0, 0, NA, 0, 10)
fx <- function(x, th) ifelse(is.na(x), TRUE, x == th)
maxRuns(x, fx, args=list(th=0))
```

`memUse` *Size of objects taking most memory use*

Description

Displays the largest objects in memUse.

Usage

```
memUse(
  n = 10,
  orderBy = "size",
  decreasing = TRUE,
  pos = 1,
  display = TRUE,
  ...
)
```

Arguments

`n`	Positive integer: Maximum number of objects to display.
`orderBy`	Either `'size'` (default) or `'name'`.
`decreasing`	Logical, if `TRUE` (default), objects are displayed from largest to smallest.
`pos`	Environment from which to obtain size of objects. Default is 1. See `ls`.#"
`display`	If `TRUE` (default), print a table with memUse used.
`...`	Other arguments to pass to `ls`.

Value

Data frame (invisible).

Examples

```
memUse()
memUse(3)
```

`mergeLists` *Merge two lists with precedence*

Description

This function merges two or more lists to create a single, combined list. If two elements in different lists have the same name, items in the later list gain preference (e.g., if there are three lists, then values in the third list gain precedence over items with the same name in the second, and the second has precedence over items in the first).

Usage

```
mergeLists(...)
```

Arguments

...	Two or more lists.

Value

A list.

See Also

`appendLists`

Examples

```
list1 <- list(a=1:3, b='Hello world!', c=LETTERS[1:3])
list2 <- list(x=4, b='Goodbye world!', z=letters[1:2])
list3 <- list(x=44, b='What up, world?', z=c('_A_', '_Z_'), w = TRUE)

mergeLists(list1, list2)
mergeLists(list2, list1)

mergeLists(list1, list2, list3)
mergeLists(list3, list2, list1)
```

`mirror` *Flip an object*

Description

This function creates a "mirror" image of a character string, a number, a matrix, or a data frame. For example "Shots were exchanged at the hospital" becomes "latipsoh eht ta degnahcxe erew stohS' and 3.14159 becomes 95141.3. Data frames and matrices will be returned with the order of columns or order of rows reversed.

Usage

```
mirror(x, direction = "lr")
```

Arguments

`x`	A vector of numeric or character values, or a matrix or data frame.
`direction`	Only used if `x` is a matrix or data frame. Accepted values are `'lr'` (left-right mirror) or `'ud'` (up-down mirror).

Value

Object with same class as `x`.

Examples

```
x <- 'Shots were exchanged at the hospital'
mirror(x)

x <- c('Water', 'water', 'everywhere')
mirror(x)

# last value will return NA because the exponentiation does not
# make sense when written backwards
x <- c(3.14159, 2.71828, 6.02214076e+23)

mirror(x)
x <- data.frame(x=1:5, y=6:10)
mirror(x)

x <- matrix(1:10, nrow=2)
mirror(x)
```

mmode *Modal value(s)*

Description

Modal value. If there is more than one unique mode, all modal values are returned.

Usage

```
mmode(x)
```

Arguments

x	Numeric or character vector.

Value

Numeric or character vector.

Examples

```
x <- c(1, 2, 3, 3, 4, 5, 3, 1, 2)
mmode(x)

x <- c(1, 2, 3)
mmode(x)
```

naCompare *Compare values using <, <=, >, >=, !=, and == (robust to NAs)*

Description

This function and set of operators perform simple (vectorized) comparisons using <, <=, >, >=, !=, or == between values and *always* returns TRUE or FALSE. TRUE only occurs if the condition can be evaluated and it is TRUE. FALSE is returned if the condition is FALSE *or* it cannot be evaluated.

Usage

```
naCompare(op, x, y)

x %<na% y

x %<=na% y

x %==na% y
```

```
x %!=na% y

x %>na% y

x %>=na% y
```

Arguments

op	Character, the operation to perform: '<', '<=', '>', '>=', '!=', or '==' . Note this must be a character (i.e., put it in quotes).
x, y	Vectors of numeric, character, NA, and/or NaN values. This is the first value in the operation x XXX y where XXX is the operator in op. If x is shorter than y then x is recycled.

Value

Vector of logical values.

Examples

```
naCompare('<', c(1, 2, NA), c(10, 1, 0))
naCompare('<', c(1, 2, NA), 10)
naCompare('<', c(1, 2, NA), NA)
# compare to:
NA < 5
NA < NA

# same operations with operators:
1 %<na% 2
1 %<na% NA
3 %==na% 3
NA %==na% 3
4 %!=na% 4
4 %!=na% NA
5 %>=na% 3
5 %>=na% NA
3 %==na% c(NA, 1, 2, 3, 4)

# compare to:
1 < 2
1 < NA
3 == 3
NA == 3
4 != 4
4 != NA
5 >= 3
5 >= NA
3 == c(NA, 1, 2, 3, 4)
```

naOmitMulti *Remove NAs from one or more equal-length vectors*

Description

This function removes elements in one or more equal-length vectors in which there is one NA at that position. For example, if there are three vectors A, B, and C, and A has an NA in the first position and C has an NA in the third position, then A, B, and C will each have the elements at positions 1 and 3 removed.

Usage

```
naOmitMulti(...)
```

Arguments

...	Numeric or character vectors.

Value

List of objects of class ....

See Also

na.omit

Examples

```
a <- c(NA, 'b', 'c', 'd', 'e', NA)
b <- c(1, 2, 3, NA, 5, NA)
c <- c(6, 7, 8, 9, 10, NA)
naOmitMulti(a, b, c)
```

naRows *Index of rows in a data frame or matrix that contain at least one* NA

Description

This function returns the row number of any row in a data frame or matrix that has at least one NA. This is the same as which(!complete.cases(x)).

Usage

```
naRows(x, inf = FALSE, inverse = FALSE)
```

Arguments

x	Data frame or matrix.
inf	Logical, if TRUE then also return row numbers of rows in which at least one element is Inf or -Inf. The default is FALSE.
inverse	Logical, if TRUE then return row numbers of rows that *do not* have NAs (and possibly Inf or -Inf). The default is FALSE.

Value

Integer vector.

Examples

```
x <- data.frame(a=1:5, b=c(1, 2, NA, 4, 5), c=c('a', 'b', 'c', 'd', NA))
naRows(x)
```

---

notIn *Opposite of '%in%'*

Description

Indicate if elements of a vector are not in another vector.

Usage

```
notIn(x, table)

x %notin% table
```

Arguments

x, table	Vectors.

Value

A logical vector.

Examples

```
x <- c('a', 'v', 'o', 'C', 'a', 'd', 'O')
y <- letters

y %notin% x
x %notin% y
```

`pairDist` *Calculate pairwise distances between two matrices or data frames.*

Description

This function takes two data frames or matrices and returns a matrix of pairwise Euclidean distances between the two.

Usage

```
pairDist(x1, x2 = NULL, na.rm = FALSE)
```

Arguments

`x1`	Data frame or matrix one or more columns wide.
`x2`	Data frame or matrix one or more columns wide. If `NULL`, then pairwise distances between all points in `x1` are calculated.
`na.rm`	Logical, if `TRUE` then any rows in `x1` or `x2` with at least one `NA` are removed first.

Value

Matrix with `nrow(x1)` rows and `nrow(x2)` columns. Values are the distance between each row of `x1` and row of `x2`.

See Also

`dist`

Examples

```
x1 <- data.frame(x=sample(1:30, 30), y=sort(round(100 * rnorm(30))), z=sample(1:30, 30))
x2 <- data.frame(x=1:20, y=round(100 * rnorm(20)), z=sample(1:20, 20))
pairDist(x1, x2)
pairDist(x1)
```

pmatchSafe *Partial matching of strings with error checking*

Description

This function is the same as pmatch, but it can throw an error instead of NA if not match is found, and can be forced to throw the error if more than the desired number of matches is found.

Usage

```
pmatchSafe(
  x,
  table,
  useFirst = FALSE,
  error = TRUE,
  ignoreCase = TRUE,
  nmax = length(x),
  ...
)
```

Arguments

x	Character: String to match.
table	Character vector: Values to which to match.
useFirst	Logical: If TRUE, and there is more than one match for a given x, then the first value in table that matches x will be returned (without an error or warning).
error	Logical: If no match is found, return an error?
ignoreCase	Logical: If TRUE (default), ignore the case of values in x and table when checking for matches.
nmax	Positive numeric integer: Maximum allowable number of matches. If more than this number of matches is found, an error will be thrown (regardless of the value of error).
...	Arguments to pass to pmatch.

Value

One or more of the values in table.

Examples

```
pmatchSafe('ap', c('apples', 'oranges', 'bananas'))

pmatchSafe('AP', c('apples', 'oranges', 'bananas'))

pmatchSafe('AP', c('apples', 'oranges', 'bananas'),
```

```
    ignoreCase = FALSE, error = FALSE)

pmatchSafe(c('ba', 'ap'), c('apples', 'oranges', 'bananas'))

# No match:
tryCatch(
    pmatchSafe('kumquats', c('apples', 'oranges', 'bananas')),
error = function(cond) FALSE
)

pmatchSafe('kumquats', c('apples', 'oranges', 'bananas'), error = FALSE)

pmatchSafe(c('ap', 'corn'), c('apples', 'oranges', 'bananas'), error = FALSE)

# Too many matches:
tryCatch(
    pmatchSafe(c('ap', 'ba'), c('apples', 'oranges', 'bananas'), nmax = 1),
error=function(cond) FALSE
)
```

`prefix` *Add leading characters to a string*

Description

Add leading characters to a string. This function is useful for ensuring, say, files get sorted in a particular order. For example, on some operating systems a file name "file 1" would come first, then "file 10", then "file 11", "file 12", etc., then "file 2", "file 21", and so on. Using `prefix`, you can add one or more leading zeros so that file names are as "file 01", "file 02", "file 03", and so on... and they will sort that way.

Usage

```
prefix(x, len, pad = "0")
```

Arguments

`x`	Character or character vector to which to add a prefix.
`len`	The total number of characters desired for each string. If a string is already this length or longer then nothing will be prefixed to that string.
`pad`	Character. Symbol to prefix to each string.

Value

Character or character vector.

Examples

```
prefix(1:5, len=2)
prefix(1:5, len=5)
prefix(1:5, len=3, pad='!')
```

---

quadArea *Area of a quadrilateral*

Description

Calculates the area of a quadrilateral by dividing it into two triangles and applying Heron's formula.

Usage

```
quadArea(x, y)
```

Arguments

x	Numeric vector. x coordinates of quadrilateral.
y	Numeric vector. y coordinates of quadrilateral.

Value

Numeric (area of a quadrilateral in same units as x and y.

Examples

```
x <- c(0, 6, 4, 1)
y <- c(0, 1, 7, 4)
quadArea(x, y)
plot(1, type='n', xlim=c(0, 7), ylim=c(0, 7), xlab='x', ylab='y')
polygon(x, y)
text(x, y, LETTERS[1:4], pos=4)
lines(x[c(1, 3)], y[c(1, 3)], lty='dashed', col='red')
```

---

renumSeq *Renumber a sequence of numbers*

Description

This function renumbers a sequence, which is helpful if "gaps" appear in the sequence. For example, consider the sequence {1, 1, 3, 1, 8, 8, 8}. This function will renumber the sequence {1, 1, 2, 1, 3, 3, 3}. NAs are ignored.

Usage

```
renumSeq(x)
```

Arguments

x	Numerical or character vector.

Value

A vector.

See Also

order, rank

Examples

```
x <- c(1, 1, 3, 1, 8, 8, 8)
renumSeq(x)

y <- c(1, 1, 3, 1, 8, NA, 8, 8)
renumSeq(y)

z <- c('c', 'c', 'b', 'a', 'w', 'a')
renumSeq(z)
```

---

rotateMatrix *Rotate values in a matrix*

Description

This function rotates the values in a matrix by a user-specified number of degrees. In almost all cases some values will fall outside the matrix so they will be discarded. Cells that have no rotated values will become NA. Only square matrices can be accommodated. In some cases a rotation will cause cells to have no assigned value because no original values fall within them. In these instances the mean value of surrounding cells is assigned to the cells with missing values. If the angle of rotation is too small then no rotation will occur.

Usage

```
rotateMatrix(x, rot)
```

Arguments

x	A matrix.
rot	Numeric. Number of degrees to rotate matrix. Values represent difference in degrees between "north" (up) and the clockwise direction.

Value

A matrix.

See Also

[base::t()]

Examples

```
x <- matrix(1:100, nrow=10)
x
rotateMatrix(x, 90) # 90 degrees to the right
rotateMatrix(x, 180) # 180 degrees to the right
rotateMatrix(x, 45) # 45 degrees to the right
rotateMatrix(x, 7) # slight rotation
rotateMatrix(x, 5) # no rotation because angle is too small
```

---

roundedSigDigits *Number of significant digits in rounded numbers*

Description

This function examines a numeric value (typically with numbers after the decimal place) and estimates either:

- The number of significant digits of the numerator and denominator of a fraction that would (approximately) result in the given value.
- The number of digits to which an integer may have been rounded, depending on whether the input has values after the decimal place or is an integer. Negative values are treated as positive so the negative of a number will returns the same value as its positive version. See *Details* for more details. *Obviously, values can appear to be rounded or repeating even when they are not!*

Usage

```
roundedSigDigits(x, minReps = 3)
```

Arguments

x	Numeric or numeric vector.
minReps	Integer. Number of times a digit or sequence of digits that occur after a decimal place needs to be repeated to assume it represents a repeating series and thus is assumed to arise from using decimal places to represent a fraction. Default is 3. For example, if minReps is 3 then 0.111 would be assumed to represent a repeating value because 1 occurs three times, so -1 would be returned since this decimal can be represented by 1/9 (i.e., division of 1 by a single-digit number). However, if minReps is 4 then the function would assume that if the value had had four digits, the next digit would not have been a 1, so returns -3 because there are three significant values after the decimal place. When the penultimate digit is >5 and the last digit is equal to the penultimate digit plus 1, then the last digit counts as a repeat of the penultimate digit. So 0.067 is assumed to have two repeating 6s. If minReps is 0 or 1 then the function will (usually) return the negative of the total number of decimal places in the value.

Details

For values with at least one non-zero digit after a decimal place with no repeated series of digits detected, the function simply returns the total number of digits (ignoring trailing zeros) times -1. For example:

- 0.3 returns -1 because there is just one value after the decimal.
- 0.34567 returns -5 because there are no repeats up to the 5th decimal place.
- 0.1212125 returns -7 because there are no repeats (starting from the right) up to the 7th decimal place.
- 0.111117 returns -6 because there are no repeats (starting from the right) up to the 7th decimal place.

The function takes account of rounding up:

- 0.666 might be a truncated version of 2/3. Two and three each have 1 significant digit, so the function returns -1 (1 value after the decimal place).
- 0.667 also returns -1 because this might represent a rounding of 2/3 and it is customary to round digits up if the next digit would have been >5.
- 0.3334 returns -4 because it is inappropriate to round 3 up to 4 if the next digit would have been 5 or less.

Repeating series are accounted for. For example:

- 0.121212 returns -2 because "12" starts repeating after the second decimal place.
- 0.000678678678 returns -6 because "678" starts repeating after the 6th place.
- 0.678678678 returns -3.
- 0.678678679 also returns -3 because 678 could be rounded to 679 if the next digit were 6.

Note that you can set the minimum number of times a digit or series needs to be repeated to count as being repeated using the argument `minReps`. The default is 3, so digits or series of digits need to be repeated at least 3 times to count a repetition, but this can be changed:

- 0.1111 returns -1 using the default requirement for 3 repetitions but -4 if the number of minimum repetitions is 5 or more.
- 0.121212 returns -2 using the default requirement for 3 repetitions but -6 if the number of minimum repetitions is 4 or more.

Trailing zeros are ignored, so 0.12300 returns -3. When values do not have digits after a decimal place the location of the first non-zero digit from the right is returned as a positive integer. For example:

- 234 returns 1 because the first non-zero digit from the right is in the 1s place.
- 100 return 3 because the first non-zero digit from the right is in the 100s place.
- 70001 returns 1 because the first non-zero digit from the right is in the 1s place.

However, note a few oddities:

- 4E5 returns 6 but 4E50 probably will not return 51 because many computers have a hard time internally representing numbers that large.

- 4E-5 returns -5 but probably will not return -50 because many computers have a hard time internally representing numbers that small.
- -100 and 100 return 3 and -0.12 and 0.12 return -2 because the negative sign is ignored.
- 0 returns 0.
- NA and NaN returns NA.

Value

Integer (number of digits) or NA (does not appear to be rounded).

Examples

```
roundedSigDigits(0.3)
roundedSigDigits(0.34567)
roundedSigDigits(0.1212125)
roundedSigDigits(0.111117)
roundedSigDigits(0.666)
roundedSigDigits(0.667)
roundedSigDigits(0.3334)
roundedSigDigits(0.121212)
roundedSigDigits(0.000678678678)
roundedSigDigits(0.678678678)
roundedSigDigits(0.678678679)
roundedSigDigits(0.1111)
roundedSigDigits(0.1111, minReps=5)
roundedSigDigits(0.121212)
roundedSigDigits(0.121212, minReps=4)
roundedSigDigits(234)
roundedSigDigits(100)
roundedSigDigits(70001)
roundedSigDigits(4E5)
roundedSigDigits(4E50)
roundedSigDigits(4E-5)
roundedSigDigits(4E-50)
roundedSigDigits(0)
roundedSigDigits(NA)

x <- c(0.0667, 0.0667, 0.067)
roundedSigDigits(x)
```

---

roundTo *Round to nearest target value*

Description

This function rounds a value to a nearest "target" value (e.g., you could round 0.72 to the nearest 0.25, or 0.75).

Usage

```
roundTo(x, target, roundFx = round)
```

Arguments

x	Numeric.
target	Numeric.
roundFx	Any of round, floor, or ceiling.

Value

Numeric.

Examples

```
roundTo(0.73, 0.05)
roundTo(0.73, 0.1)
roundTo(0.73, 0.25)
roundTo(0.73, 0.25, floor)
roundTo(0.73, 1)
roundTo(0.73, 10)
roundTo(0.73, 10, ceiling)
```

---

rowColIndexing *Convert between row- and column-style indexing of matrices*

Description

These functions converts index values of cells between row- and column-style indexing of cells in matrices. Column indexing (the default for matrices) has the cell "1" in the upper left corner of the matrix. The cell "2" is below it, and so on. The numbering then wraps around to the top of the next column. Row indexing (the default for rasters, for example), also has cell "1" in the upper left, but cell "2" is to its right, and so on. Numbering then wraps around to the next row.

Usage

```
rowColIndexing(x, cell, dir)
```

Arguments

x	Either a matrix, *or* a vector with two values, one for the number of rows and one for the number of columns in a matrix.
cell	One or more cell indices (positive integers).
dir	The "direction" in which to convert. If 'row', it is assumed that cell is a column-style index and so should be converted to a row-style index. If 'col', it is assumed that cell is a row-style index and so should be converted to a column-style index.

Value

One or more positive integers.

Examples

```
# column versus row indexing
colIndex <- matrix(1:40, nrow=5, ncol=8)
rowIndex <- matrix(1:40, nrow=5, ncol=8, byrow=TRUE)
colIndex
rowIndex

# examples
x <- matrix('a', nrow=5, ncol=8, byrow=TRUE)
rowColIndexing(x, cell=c(1, 6, 20), 'row')
rowColIndexing(x, cell=c(1, 6, 20), 'col')

rowColIndexing(c(5, 8), cell=c(1, 6, 20), 'row')
rowColIndexing(c(5, 8), cell=c(1, 6, 20), 'col')
```

---

rstring *Make a nearly-guaranteed unique string*

Description

`rstring()` makes a string that is statically extremely likely to be unique (using default options).

Usage

```
rstring(n, x = 12, filesafe = TRUE)
```

Arguments

`n`	Numeric integer: How many strings to make (default is 1).
`x`	Numeric integer: Number of letters and digits to use to make the string. Default is 12, leading to a probability of two matching random strings of <3.7E-18 if `filesafe` = `TRUE` and <3.1E-22 if `FALSE`.
`filesafe`	Logical: If `TRUE` (default), make file-safe names (leading character is a letter).

Value

Character.

Examples

```
rstring(1)
rstring(5)
rstring(5, 3)
```

---

say *Nicer version of* `print()` *or* `cat()` *function*

Description

This function is a nicer version of `print()` or `cat()`, especially when used inline for functions because it displays immediately and pastes all strings together. It also does some rudimentary but optional word wrapping.

Usage

```
say(
  ...,
  pre = 0,
  post = 1,
  breaks = NULL,
  wiggle = 10,
  preBreak = 1,
  level = NULL,
  deco = "#"
)
```

Arguments

`...`	Character strings to print
`pre`	Integer >= 0. Number of blank lines to print before strings
`post`	Integer >= 0. Number of blank lines to print after strings
`breaks`	Either `NULL`, which causes all strings to be printed on the same line (no wrap overflow) or a positive integer which wraps lines at this character length (e.g., `breaks=80` inserts line breaks every 80 characters).
`wiggle`	Integer >- 0. Allows line to overrun `breaks` length in characters before inserting line breaks.
`preBreak`	If wrapping long lines indicates how subsequent lines are indented. NULL causes lines to be printed starting at column 1 on the display device. A positive integer inserts `preBreak` number of spaces before printing each line. A string causes each line to start with this string.
`level`	Integer or `NULL`. If `NULL`, then the items in ... are displayed as-is. Otherwise, a value of 1, 2, or 3 indicates teh heading level, with lower numbers causing more decoration and spacing to be used.
`deco`	Character. Character to decorate text with if `level` is not `NULL`.

Value

Nothing (side effect is output on the display device).

Examples

```
say('The quick brown fox ', 'jumps over the lazy ', 'Susan.')
say('The quick brown fox ', 'jumps over the lazy ', 'Susan.', breaks=10)
say('The quick brown fox ', 'jumps over the lazy ', 'Susan.', level=1)
say('The quick brown fox ', 'jumps over the lazy ', 'Susan.', level=2)
say('The quick brown fox ', 'jumps over the lazy ', 'Susan.', level=3)
```

---

stretchMinMax *Rescale values*

Description

This function rescales a vector of numeric values to an arbitrary range. Optionally, after the stretch values equal to the lowest value can be "nudged" slightly higher to half the minimum value across the rescaled vector of values > 0.

Usage

```
stretchMinMax(
  x,
  lower = 0,
  upper = 1,
  nudgeUp = FALSE,
  nudgeDown = FALSE,
  na.rm = FALSE
)
```

Arguments

x	Numeric list.
lower	Numeric, low end of range to which to stretch.
upper	Numeric, high end of range to which to stretch.
nudgeUp, nudgeDown	Logical, if FALSE (default) then do nothing. If TRUE then *after* rescaling to [0, 1], a small value will be added to all values of x equal to 0. This value is equal to 0.5 * min(x[x > 0]).
na.rm	Logical, if FALSE (default) then if any values of x are NA then the returned value will be NA. If TRUE then NA's are ignored in calculation.

Value

Numeric value.

See Also

scale

Examples

```
x <- 1:10
stretchMinMax(x)
stretchMinMax(x, lower=2, upper=5)
stretchMinMax(x, nudgeUp=TRUE)
stretchMinMax(x, lower=2, upper=5, nudgeUp=TRUE)
stretchMinMax(x, nudgeDown=TRUE)
stretchMinMax(x, lower=2, upper=5, nudgeUp=TRUE, nudgeDown=TRUE)
x <- c(1:5, NA)
stretchMinMax(x)
stretchMinMax(x, na.rm=TRUE)
```

---

unlistRecursive *For any object in a list that is also a list, unlist it*

Description

This function takes as an argument a `list`. If any of its elements are also lists, it unlists them. The output is the same as the input, except that there will be one new element per element in each sublist, and the sublists will be removed.

Usage

```
unlistRecursive(x)
```

Arguments

x	A `list`.

Value

A `list`.

See Also

unlist

Examples

```
x <- list(
   a = 1:3,
   b = list(
      b1 = c("The", "quick", "brown", "function"),
     b2 = 4:1,
      b3 = list(
         b3_1 = 5:7
      )
   ),
   c = "end"
)

unlistRecursive(x)
```

`which.pmax` *Which vector has maximum value at each element*

Description

These functions are vectorized versions of `which.max` and `which.min`, which return the index of the value that is maximum or minimum (or the first maximum/minimum value, if there is a tie). In this case, the function is supplied two or more vectors of the same length. For each element at the same position (e.g., the first element in each vector, then the second element, etc.) the function returns an integer indicating which vector has the highest or lowest value (or the index of the first vector with the highest or lowest value in case of ties).

Usage

```
which.pmax(..., na.rm = TRUE)

which.pmin(..., na.rm = TRUE)
```

Arguments

`...`	Two or more vectors. If lengths do not match, the results will likely be be unanticipated.
`na.rm`	Logical, if `FALSE` and any of the vectors contains an `NA` or `NaN`, the function will return an `NA`. If `TRUE` (default), then `NA` will only be returned if all elements at that position are `NA` or `NaN`.

Value

Vector the same length as the input, with numeric values indicating which vector has the highest value at that position. In case of ties, the index of the first vector is returned.

Functions

- `which.pmin()`: Which vector has minimum value at each element

See Also

`which.max`, `which.min`, `pmax`, `pmin`

Examples

```
set.seed(123)
a <- sample(9, 5)
b <- sample(9, 5)
c <- sample(9, 5)
a[2:3] <- NA
b[3] <- NA
a[6] <- NA
b[6] <- NA
c[6] <- NA
which.pmax(a, b, c)
which.pmin(a, b, c)
which.pmax(a, b, c, na.rm=FALSE)
which.pmin(a, b, c, na.rm=FALSE)
```

---

yearFromDate *Year from date formats that are possibly ambiguous*

Description

This function attempts to return the year from characters representing dates formats. The formats can be ambigous and varied within the same set. For example, it returns "1982" (or 9982 if century is ambiguous) from "11/20/82", "1982-11-20", "Nov. 20, 1982", "20 Nov 1982", "20-Nov-1982", "20/Nov/1982", "20 Nov. 82", "20 Nov 82". The function handles ambiguous centuries (e.g., 1813, 1913, 2013) by including a dummy place holder for the century place (i.e., 9913). Note that it may return warnings like "NAs introduced by coercion".

Usage

```
yearFromDate(x, yearLast = TRUE)
```

Arguments

`x`	Character or character vector, one or more dates.
`yearLast`	Logical, if `TRUE` assume that dates like "XX/YY/ZZ" list the year last (=ZZ). If `FALSE`, assume they're first (=XX).

Value

Numeric.

Examples

```
yearFromDate(1969, yearLast=TRUE)
yearFromDate('10-Jul-71', yearLast=TRUE) # --> 9971
yearFromDate('10-Jul-1971', yearLast=TRUE) # --> 1971
yearFromDate('10-19-71', yearLast=TRUE) # --> 9971
yearFromDate('10-19-1969', yearLast=TRUE) # --> 1969
yearFromDate('10-1-71', yearLast=TRUE) # --> 9971
yearFromDate('3-22-71', yearLast=TRUE) # --> 9971
yearFromDate('3-2-71', yearLast=TRUE) # --> 9971
yearFromDate('10-1-1969', yearLast=TRUE) # --> 1969
yearFromDate('3-22-1969', yearLast=TRUE) # --> 1969
yearFromDate('3-2-1969', yearLast=TRUE) # --> 1969
yearFromDate('10/Jul/71', yearLast=TRUE) # --> 9971
yearFromDate('10/Jul/1971', yearLast=TRUE) # --> 1971
yearFromDate('10/19/71', yearLast=TRUE) # --> 9971
yearFromDate('10/19/1969', yearLast=TRUE) # --> 1969
yearFromDate('10/1/71', yearLast=TRUE) # --> 9971
yearFromDate('3/22/71', yearLast=TRUE) # --> 9971
yearFromDate('3/2/71', yearLast=TRUE) # --> 9971
yearFromDate('10/1/1969', yearLast=TRUE) # --> 1969
yearFromDate('3/22/1969', yearLast=TRUE) # --> 1969
yearFromDate('3/2/1969', yearLast=TRUE) # --> 1969
yearFromDate('10 mmm 71', yearLast=TRUE) # "mmm" is month abbreviation--> 9971
yearFromDate('5 mmm 71', yearLast=TRUE) # "mmm" is month abbreviation--> 9971
yearFromDate('10 19 71', yearLast=TRUE) # --> 9971
yearFromDate('10 19 1969', yearLast=TRUE) # --> 1969
yearFromDate('10 1 71', yearLast=TRUE) # --> 9971
yearFromDate('3 22 71', yearLast=TRUE) # --> 9971
yearFromDate('3 2 71', yearLast=TRUE) # --> 9971
yearFromDate('10 1 1969', yearLast=TRUE) # --> 1969
yearFromDate('3 22 1969', yearLast=TRUE) # --> 1969
yearFromDate('3 2 1969', yearLast=TRUE) # --> 1969
yearFromDate('Oct. 19, 1969', yearLast=TRUE) # --> 1969
yearFromDate('19 October 1969', yearLast=TRUE) # --> 1969
yearFromDate('How you do dat?', yearLast=TRUE) # --> NA
yearFromDate('2014-07-03', yearLast=TRUE) # --> 2014
yearFromDate('2014-7-03', yearLast=TRUE) # --> 2014
yearFromDate('2014-07-3', yearLast=TRUE) # --> 2014
yearFromDate('2014-7-3', yearLast=TRUE) # --> 2014
yearFromDate('2014/07/03', yearLast=TRUE) # --> 2014
yearFromDate('2014/7/03', yearLast=TRUE) # --> 2014
yearFromDate('2014/07/3', yearLast=TRUE) # --> 2014
yearFromDate('2014/7/3', yearLast=TRUE) # --> 2014
yearFromDate('2014 07 03', yearLast=TRUE) # --> 2014
yearFromDate('2014 7 03', yearLast=TRUE) # --> 2014
yearFromDate('2014 07 3', yearLast=TRUE) # --> 2014
yearFromDate('2014 7 3', yearLast=TRUE) # --> 2014

yearFromDate(1969, yearLast=FALSE)
yearFromDate('10-Jul-71', yearLast=FALSE) # --> 9971
yearFromDate('10-Jul-1971', yearLast=FALSE) # --> 1971
yearFromDate('10-19-71', yearLast=FALSE) # --> 9910
```

```
yearFromDate('10-19-1969', yearLast=FALSE) # --> 1969
yearFromDate('10-1-71', yearLast=FALSE) # --> 9910
yearFromDate('3-22-71', yearLast=FALSE) # --> 9971
yearFromDate('3-2-71', yearLast=FALSE) # --> 9971
yearFromDate('10-1-1969', yearLast=FALSE) # --> 1969
yearFromDate('3-22-1969', yearLast=FALSE) # --> 1969
yearFromDate('3-2-1969', yearLast=FALSE) # --> 1969
yearFromDate('10/19/71', yearLast=FALSE) # --> 9910
yearFromDate('10/19/1969', yearLast=FALSE) # --> 1969
yearFromDate('10/1/71', yearLast=FALSE) # --> 9910
yearFromDate('3/22/71', yearLast=FALSE) # --> 9971
yearFromDate('3/2/71', yearLast=FALSE) # --> 9971
yearFromDate('10/1/1969', yearLast=FALSE) # --> 1969
yearFromDate('3/22/1969', yearLast=FALSE) # --> 1969
yearFromDate('3/2/1969', yearLast=FALSE) # --> 1969
yearFromDate('10 mmm 71', yearLast=FALSE) # "mmm" is month abbreviation--> 9971
yearFromDate('5 mmm 71', yearLast=FALSE) # "mmm" is month abbreviation--> 9971
yearFromDate('10 19 71', yearLast=FALSE) # --> 9910
yearFromDate('10 19 1969', yearLast=FALSE) # --> 1969
yearFromDate('10 1 71', yearLast=FALSE) # --> 9910
yearFromDate('3 22 71', yearLast=FALSE) # --> 9971
yearFromDate('3 2 71', yearLast=FALSE) # --> 9971
yearFromDate('10 1 1969', yearLast=FALSE) # --> 1969
yearFromDate('3 22 1969', yearLast=FALSE) # --> 1969
yearFromDate('3 2 1969', yearLast=FALSE) # --> 1969
yearFromDate('Oct. 19, 1969', yearLast=FALSE) # --> 1969
yearFromDate('19 October 1969', yearLast=FALSE) # --> 1969
yearFromDate('How you do dat?', yearLast=FALSE) # --> NA
yearFromDate('2014-07-03', yearLast=FALSE) # --> 2014
yearFromDate('2014-7-03', yearLast=FALSE) # --> 2014
yearFromDate('2014-07-3', yearLast=FALSE) # --> 2014
yearFromDate('2014-7-3', yearLast=FALSE) # --> 2014
yearFromDate('2014/07/03', yearLast=FALSE) # --> 2014
yearFromDate('2014/7/03', yearLast=FALSE) # --> 2014
yearFromDate('2014/07/3', yearLast=FALSE) # --> 2014
yearFromDate('2014/7/3', yearLast=FALSE) # --> 2014
yearFromDate('2014 07 03', yearLast=FALSE) # --> 2014
yearFromDate('2014 7 03', yearLast=FALSE) # --> 2014
yearFromDate('2014 07 3', yearLast=FALSE) # --> 2014
yearFromDate('2014 7 3', yearLast=FALSE) # --> 2014
```

Index