Package 'gdata'

May 11, 2022

Title Various R Programming Tools for Data Manipulation

Description Various R programming tools for data manipulation, including:
- medical unit conversions ('ConvertMedUnits', 'MedUnits'),
- combining objects ('bindData', 'cbindX', 'combine', 'interleave'),
- character vector operations ('centerText', 'startsWith', 'trim'),
- factor manipulation ('levels', 'reorder.factor', 'mapLevels'),
- obtaining information about R objects ('object.size', 'elem', 'env',
  'humanReadable', 'is.what', 'll', 'keep', 'ls.funs',
  'Args','nPairs', 'nobs'),
- manipulating MS-Excel formatted files ('read.xls',
  'installXLSXsupport', 'sheetCount', 'xlsFormats'),
- generating fixed-width format files ('write.fwf'),
- extricating components of date & time objects ('getYear', 'getMonth',
  'getDay', 'getHour', 'getMin', 'getSec'),
- operations on columns of data frames ('matchcols', 'rename.vars'),
- matrix operations ('unmatrix', 'upperTriangle', 'lowerTriangle'),
- operations on vectors ('case', 'unknownToNA', 'duplicated2', 'trimSum'),
- operations on data frames ('frameApply', 'wideByFactor'),
- value of last evaluated expression ('ans'), and
- wrapper for 'sample' that ensures consistent behavior for both
  scalar and vector arguments ('resample').

Depends R (>= 2.3.0)

SystemRequirements perl (>= 5.10.0)

Imports gtools, stats, methods, utils

Version 2.18.0.1

Date 2017-06-05

Author Gregory R. Warnes, Ben Bolker, Gregor Gorjanc, Gabor
Grothendieck, Ales Korosec, Thomas Lumley, Don MacQueen, Arni
Magnusson, Jim Rogers, and others

Maintainer Gregory R. Warnes <greg@warnes.net>

License GPL-2

NeedsCompilation no

Suggests RUnit

Repository CRAN

Date/Publication 2022-05-10 16:42:52

R **topics documented:**

Index **67**

---

gdata-package   *Various R programming tools for data manipulation*

Description

gdata package provides various R programming tools for data manipulation.

Details

The following are sources of information on **gdata** package:

DESCRIPTION file	`library(help="gdata")`
This file	`package?gdata`
Vignette	`vignette("unknown")`
	`vignette("mapLevels")`
Some help files	[read.xls](#)
	[write.fwf](#)
News	`file.show(system.file("NEWS", package="gdata"))`

---

ans   *Value of Last Evaluated Expression*

Description

The functon returns the value of the last evaluated *top-level* expression, which is always assigned to `.Last.value` (in `package:base`).

Usage

```
ans()
```

Details

This function retrieves `.Last.value`. For more details see `.Last.value`.

Value

.Last.value

Author(s)

Liviu Andronic

See Also

.Last.value, eval

Examples

```
2+2                 # Trivial calculation...
ans()               # See the answer again

gamma(1:15)         # Some intensive calculation...
fac14 <- ans()      # store the results into a variable

rnorm(20)           # Generate some standard normal values
ans()^2             # Convert to Chi-square(1) values...
stem(ans())         # Now show a stem-and-leaf table
```

Args   *Describe Function Arguments*

Description

Display function argument names and corresponding default values, formatted in two columns for
easy reading.

Usage

```
Args(name, sort=FALSE)
```

Arguments

name   a function or function name.

sort   whether arguments should be sorted.

Value

A data frame with named rows and a single column called value, containing the default value of
each argument.

Note

Primitive functions like sum and all have no formal arguments. See the formals help page.

Author(s)

Arni Magnusson

See Also

Args is a verbose alternative to `args`, based on `formals`.

`help` also describes function arguments.

Examples

```
Args(glm)
Args(scan)
Args(legend, sort=TRUE)
```

---

bindData	*Bind two data frames into a multivariate data frame*

Description

Usually data frames represent one set of variables and one needs to bind/join them for multivariate analysis. When `merge` is not the approriate solution, `bindData` might perform an appropriate binding for two data frames. This is especially usefull when some variables are measured once, while others are repeated.

Usage

```
bindData(x, y, common)
```

Arguments

x	data.frame
y	data.frame
common	character, list of column names that are common to both input data frames

Details

Data frames are joined in a such a way, that the new data frame has $c + (n_1 - c) + (n_2 - c)$ columns, where c is the number of common columns, and n_1 and n_2 are the number of columns in the first and in the second data frame, respectively.

Value

A data frame.

Author(s)

Gregor Grojanc

See Also

merge, wideByFactor

Examples

```
n1 <- 6
n2 <- 12
n3 <- 4
## Single trait 1
num <- c(5:n1, 10:13)
(tmp1 <- data.frame(y1=rnorm(n=n1),
                    f1=factor(rep(c("A", "B"), n1/2)),
                    ch=letters[num],
                    fa=factor(letters[num]),
                    nu=(num) + 0.5,
                    id=factor(num), stringsAsFactors=FALSE))

## Single trait 2 with repeated records, some subjects also in tmp1
num <- 4:9
(tmp2 <- data.frame(y2=rnorm(n=n2),
                    f2=factor(rep(c("C", "D"), n2/2)),
                    ch=letters[rep(num, times=2)],
                    fa=factor(letters[rep(c(num), times=2)]),
                    nu=c((num) + 0.5, (num) + 0.25),
                    id=factor(rep(num, times=2)), stringsAsFactors=FALSE))

## Single trait 3 with completely distinct set of subjects
num <- 1:4
(tmp3 <- data.frame(y3=rnorm(n=n3),
                    f3=factor(rep(c("E", "F"), n3/2)),
                    ch=letters[num],
                    fa=factor(letters[num]),
                    nu=(num) + 0.5,
                    id=factor(num), stringsAsFactors=FALSE))

## Combine all datasets
(tmp12 <- bindData(x=tmp1, y=tmp2, common=c("id", "nu", "ch", "fa")))
(tmp123 <- bindData(x=tmp12, y=tmp3, common=c("id", "nu", "ch", "fa")))

## Sort by subject
tmp123[order(tmp123$ch), ]
```

---

case   *Map elements of a vector according to the provided 'cases'*

Description

Map elements of a vector according to the provided 'cases'. This function is useful for mapping
discrete values to factor labels and is the vector equivalent to the switch function.

Usage

```
case(x, ..., default = NA)
```

Arguments

x	Vector to be converted
...	Map of alternatives, specified as "name"=value
default	Value to be assigned to elements of x not matching any of the alternatives. Defaults to NA.

Details

This function is to switch what ifelse is to if, and is a convenience wrapper for factor.

Value

A factor variables with each element of x mapped into the corresponding level of specified in the mapping.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

factor, switch, ifelse

Examples

```
## default = NA
case( c(1,1,4,3), "a"=1, "b"=2, "c"=3)

## default = "foo"
case( c(1,1,4,3), "a"=1, "b"=2, "c"=3, default="foo" )
```

---

cbindX   *Column-bind objects with different number of rows*

Description

cbindX column-binds objects with different number of rows.

Usage

```
cbindX(...)
```

Arguments

`...`	matrix and data.frame objects

Details

First the object with maximal number of rows is found. Other objects that have less rows get (via [rbind](#)) additional rows with NA values. Finally, all objects are column-binded (via [cbind](#)).

Value

See details

Author(s)

Gregor Gorjanc

See Also

Regular [cbind](#) and [rbind](#)

Examples

```
df1 <- data.frame(a=1:3, b=c("A", "B", "C"))
df2 <- data.frame(c=as.character(1:5), a=5:1)

ma1 <- matrix(as.character(1:4), nrow=2, ncol=2)
ma2 <- matrix(1:6, nrow=3, ncol=2)

cbindX(df1, df2)
cbindX(ma1, ma2)
cbindX(df1, ma1)
cbindX(df1, df2, ma1, ma2)
cbindX(ma1, ma2, df1, df2)
```

`centerText`	*Center Text Strings*

Description

Function to center text strings for display on the text console by prepending the necessary number of spaces to each element.

Usage

```
centerText(x, width = getOption("width"))
```

Arguments

x	Character vector containing text strings to be centered.
width	Desired display width. Defaults to the R display width given by getOption("width").

Details

Each element will be centered individually by prepending the necessary number of spaces to center the text in the specified display width assuming a fixed width font.

Value

Vector of character strings.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

[strwrap](#)

Examples

```
cat(centerText("One Line Test"), "\n\n")

mText <-c("This", "is an example",
          " of a multiline text    ",
          "with ",
          "          leading",
          " and trailing          ",
          "spaces.")
cat("\n", centerText(mText), "\n", sep="\n")
```

---

combine   *Combine R Objects With a Column Labeling the Source*

Description

Take a sequence of vector, matrix or data frames and combine into rows of a common data frame with an additional column source indicating the source object.

Usage

```
combine(..., names=NULL)
```

Arguments

...	vectors or matrices to combine.
names	character vector of names to use when creating source column.

Details

If there are several matrix arguments, they must all have the same number of columns. The number of columns in the result will be one larger than the number of columns in the component matrixes. If all of the arguments are vectors, these are treated as single column matrixes. In this case, the column containing the combineinated vector data is labeled `data`.

When the arguments consist of a mix of matrices and vectors the number of columns of the result is determined by the number of columns of the matrix arguments. Vectors are considered row vectors and have their values recycled or subsetted (if necessary) to achieve this length.

The `source` column is created as a factor with levels corresponding to the name of the object from which the each row was obtained. When the `names` argument is ommitted, the name of each object is obtained from the specified argument name in the call (if present) or from the name of the object. See below for examples.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

[rbind](), [merge]()

Examples

```
a  <-  matrix(rnorm(12),ncol=4,nrow=3)
b  <-  1:4
combine(a,b)

combine(x=a,b)
combine(x=a,y=b)
combine(a,b,names=c("one","two"))

c <- 1:6
combine(b,c)
```

ConvertMedUnits	*Convert medical measurements between International Standard (SI) and US 'Conventional' Units.*

Description

Convert Medical measurements between International Standard (SI) and US 'Conventional' Units.

Usage

```
ConvertMedUnits(x, measurement, abbreviation,
                to = c("Conventional", "SI", "US"),
                exact = !missing(abbreviation))
```

Arguments

`x`	Vector of measurement values
`measurement`	Name of the measurement
`abbreviation`	Measurement abbreviation
`to`	Target units
`exact`	Logicial indicating whether matching should be exact

Details

Medical laboratories and practitioners in the United States use one set of units (the so-called 'Conventional' units) for reporting the results of clinical laboratory measurements, while the rest of the world uses the International Standard (SI) units. It often becomes necessary to translate between these units when participating in international collaborations.

This function converts between SI and US 'Conventional' units.

If `exact=FALSE`, `grep` will be used to do a case-insensitive sub-string search for matching measurement names. If more than one match is found, an error will be generated, along with a list of the matching entries.

Value

Returns a vector of converted values. The attribute 'units' will contain the target units converted.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

<https://globalrph.com/medical/conventional-units-international-units/>

See Also

The data set MedUnits provides the conversion factors.

Examples

```
data(MedUnits)

# show available conversions
MedUnits$Measurement

# Convert SI Glucose measurement to 'Conventional' units
GlucoseSI = c(5, 5.4, 5, 5.1, 5.6, 5.1, 4.9, 5.2, 5.5) # in SI Units
GlucoseUS = ConvertMedUnits( GlucoseSI, "Glucose", to="US" )
cbind(GlucoseSI,GlucoseUS)

## Not run:
# See what happens when there is more than one match
```

```
ConvertMedUnits( 27.5, "Creatin", to="US")

## End(Not run)

# To solve the problem do:
ConvertMedUnits( 27.5, "Creatinine", to="US", exact=TRUE)
```

---

drop.levels   *Drop unused factor levels*

Description

Drop unused levels in a factor

Usage

```
drop.levels(x, reorder=TRUE, ...)
```

Arguments

x	object to be processed
reorder	should factor levels be reordered using reorder.factor?
...	additional arguments to reorder.factor

Details

drop.levels is a generic function, where default method does nothing, while method for factor s drops all unused levels. Drop is done with x[, drop=TRUE].

There are also convenient methods for list and data.frame, where all unused levels are dropped in all factors (one by one) in a list or a data.frame.

Value

Input object without unused levels.

Author(s)

Jim Rogers <james.a.rogers@pfizer.com> and Gregor Gorjanc

Examples

```
f <- factor(c("A", "B", "C", "D"))[1:3]
drop.levels(f)

l <- list(f=f, i=1:3, c=c("A", "B", "D"))
drop.levels(l)
```

```
df <- as.data.frame(l)
str(df)
str(drop.levels(df))
```

---

duplicated2 *Determine Duplicate Elements*

Description

duplicated2() determines which elements of a vector or data frame are duplicates, and returns a logical vector indicating which elements (rows) are duplicates.

Usage

```
duplicated2(x, bothWays=TRUE, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
bothWays	if TRUE (the default), duplication should be considered from both sides. For more information see the argument fromLast to the function [duplicated]().
...	further arguments passed down to duplicated() and its methods.

Details

The standard [duplicated]() function (in package:base) only returns TRUE for the second and following copies of each duplicated value (second-to-last and earlier when fromLast=TRUE). This function returns all duplicated elementes, including the first (last) value.

When bothWays is FALSE, duplicated2() defaults to a [duplicated]() call. When bothWays is TRUE, the following call is being executed: duplicated(x, ...) | duplicated(x, fromLast=TRUE, ...)

Value

For a vector input, a logical vector of the same length as x. For a data frame, a logical vector with one element for each row. For a matrix or array, and when MARGIN = 0, a logical array with the same dimensions and dimnames.

For more details see [duplicated]().

Author(s)

Liviu Andronic

See Also

[duplicated](), [unique]()

Examples

```
    data(iris)
    iris[duplicated(iris), ]              # 2nd duplicated value
    iris[duplicated(iris, fromLast=TRUE), ] # 1st duplicated value
    iris[duplicated2(iris), ]             # both duplicated values
```

---

elem   *Display Information about Elements in a Given Object*

Description

This function is depreciated. Please use [ll](#) *instead.*

Display name, class, size, and dimensions of each element in a given object.

Usage

```
elem(object=1, unit=c("KB","MB","bytes"), digits=0,
                 dimensions=FALSE)
```

Arguments

object	object containing named elements, perhaps a model or data frame.
unit	required unit for displaying element size: "KB", "MB", "bytes", or first letter.
digits	number of decimals to display when rounding element size.
dimensions	whether element dimensions should be returned.

Details

A verbose alternative to names().

Value

A data frame with named rows and the following columns:

Class	element class.
KB	element size *(see notes)*.
Dim	element dimensions *(optional)*.

Note

The name of the element size column is the same as the unit used.

Elements of class classRepresentation, ClassUnionRepresentation, and grob do not have a defined size, so 0 bytes are assumed for those.

Author(s)

Arni Magnusson <arnima@u.washington.edu>

See Also

names, str, and summary display different information about object elements.

ll and env are related to elem.

Examples

```
## Not run:
data(infert)
elem(infert)
model <- glm(case~spontaneous+induced, family=binomial, data=infert)
elem(model, dim=TRUE)
elem(model$family)

## End(Not run)
```

---

env *Describe All Loaded Environments*

Description

Display name, number of objects, and size of all loaded environments.

Usage

```
env(unit="KB", digits=0)
```

Arguments

unit	unit for displaying environment size: "bytes", "KB", "MB", or first letter.
digits	number of decimals to display when rounding environment size.

Value

A data frame with the following columns:

Environment	environment name.
Objects	number of objects in environment.
KB	environment size *(see notes)*.

Note

The name of the environment size column is the same as the unit used.

Author(s)

Arni Magnusson

See Also

env is a verbose alternative to search.

[ll](ll) is a related function that describes objects in an environment.

Examples

```
## Not run:
env()

## End(Not run)
```

---

first   *Return first or last element of an object*

Description

Return first or last element of an object. These functions are convenience wrappers for head(x, n=1, ...) and tail(x, n=1, ...).

Usage

```
first(x, n=1, ...)
last(x, n=1, ...)
first(x, n=1, ...) <- value
last(x, n=1, ...) <- value
```

Arguments

x	data object
n	a single integer. If positive, size for the resulting object: number of elements for a vector (including lists), rows for a matrix or data frame or lines for a function. If negative, all but the 'n' last/first number of elements of 'x'.
...	arguments to be passed to or from other methods.
value	a vector of values to be assigned (should be of length n)

Value

An object (usually) like 'x' but generally smaller.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

head, tail, left, right

Examples

```
## works for vectors..
v <- 1:10
first(v)
last(v)

first(v) <- 9
v

last(v) <- 20
v

## and for lists
l <- list(a=1, b=2, c=3)
first(l)
last(l)

first(l) <- "apple"
last(l)  <- "bannana"
l

## and data.frames
df <- data.frame(a=1:2, b=3:4, c=5:6)
first(df)
last(df)

first(df) <- factor(c("red","green"))
last(df) <- list(c(20,30)) # note the enclosing list!
df

## and matrixes
m <- as.matrix(df)
first(m)
last(m)

first(m) <- "z"
last(m)  <- "q"
m
```

---

frameApply   *Subset analysis on data frames*

Description

Apply a function to row subsets of a data frame.

Usage

```
frameApply(x, by=NULL, on=by[1], fun=function(xi) c(Count=nrow(xi)),
          subset=TRUE, simplify=TRUE, byvar.sep="\\$\\@\\$", ...)
```

Arguments

x   a data frame

by   names of columns in x specifying the variables to use to form the subgroups.
                None of the by variables should have the name "sep" (you will get an error if
                one of them does; a bit of laziness in the code). Unused levels of the by variables
                will be dropped. Use by = NULL (the default) to indicate that all of the data is to
                be treated as a single (trivial) subgroup.

on   names of columns in x specifying columns over which fun is to be applied.
                These can include columns specified in by, (as with the default) although that is
                not usually the case.

fun   a function that can operate on data frames that are row subsets of x[on]. If
                simplify = TRUE, the return value of the function should always be either a try-
                error (see [try](#)), or a vector of fixed length (i.e. same length for every subset),
                preferably with named elements.

subset   logical vector (can be specified in terms of variables in data). This row subset of
                x is taken before doing anything else.

simplify   logical. If TRUE (the default), return value will be a data frame including the by
                columns and a column for each element of the return vector of fun. If FALSE,
                the return value will be a list, sometimes necessary for less structured output
                (see description of return value below).

byvar.sep   character. This can be any character string not found anywhere in the values
                of the by variables. The by variables will be pasted together using this as the
                separator, and the result will be used as the index to form the subgroups.

...   additional arguments to fun.

Details

This function accomplishes something similar to [by](#). The main difference is that frameApply is
designed to return data frames and lists instead of objects of class 'by'. Also, frameApply works
only on the unique combinations of the by that are actually present in the data, not on the entire
cartesian product of the by variables. In some cases this results in great gains in efficiency, although
frameApply is hardly an efficient function.

Value

a data frame if simplify = TRUE (the default), assuming there is sufficiently structured output from
fun. If simplify = FALSE and by is not NULL, the return value will be a list with two elements.
The first element, named "by", will be a data frame with the unique rows of x[by], and the second
element, named "result" will be a list where the ith component gives the result for the ith row of the
"by" element.

Author(s)

Jim Rogers <james.a.rogers@pfizer.com>

Examples

```
data(ELISA, package="gtools")

# Default is slightly unintuitive, but commonly useful:
frameApply(ELISA, by = c("PlateDay", "Read"))

# Wouldn't actually recommend this model! Just a demo:
frameApply(ELISA, on = c("Signal", "Concentration"), by = c("PlateDay", "Read"),
           fun = function(dat) coef(lm(Signal ~ Concentration, data =
dat)))

frameApply(ELISA, on = "Signal", by = "Concentration",
           fun = function(dat, ...) {
                    x <- dat[[1]]
                    out <- c(Mean = mean(x, ...),
                             SD = sd(x, ...),
                             N = sum(!is.na(x)))
                 },
           na.rm = TRUE,
           subset = !is.na(Concentration))
```

---

gdata-defunct   *Defunct Functions in Package 'gdata'*

Description

The functions or variables listed here are no longer part of 'gdata'.

Usage

```
aggregate.table(x, by1, by2, FUN=mean, ...)
```

Arguments

x	data to be summarized
by1	first grouping factor.
by2	second grouping factor.
FUN	a scalar function to compute the summary statistics which can be applied to all data subsets. Defaults to mean.
...	Optional arguments for FUN.

Details

aggregate.table(x, by1, by2, FUN=mean, ...) should be replacede by tapply(X=x, INDEX=list(by1, by2), FUN=FUN, ...).

getYear   *Get date/time parts from date and time objects*

Description

get* functions provide an *experimental* approach for extracting the date/time parts from objects of a date/time class. They are designed to be intiutive and thus lowering the learning curve for work with date and time classes in R.

Usage

```
getYear(x, format, ...)
getMonth(x, format, ...)
getDay(x, format, ...)
getHour(x, format, ...)
getMin(x, format, ...)
getSec(x, format, ...)
```

Arguments

x	generic, date/time object
format	character, format
...	arguments pased to other methods

Value

Character

Author(s)

Gregor Gorjanc

See Also

[Date](), [DateTimeClasses](), [strptime]()

Examples

```
## --- Date class ---

tmp <- Sys.Date()
tmp

getYear(tmp)
getMonth(tmp)
```

```
getDay(tmp)

## --- POSIXct class ---

tmp <- as.POSIXct(tmp)

getYear(tmp)
getMonth(tmp)
getDay(tmp)

## --- POSIXlt class ---

tmp <- as.POSIXlt(tmp)

getYear(tmp)
getMonth(tmp)
getDay(tmp)
```

---

humanReadable	*Print Byte Size in Human Readable Format*

Description

humanReadable converts integer byte sizes to a human readable units such as kB, MB, GB, etc.

Usage

```
humanReadable(x, units="auto", standard=c("IEC", "SI", "Unix"),
              digits=1, width=NULL, sep=" ", justify=c("right", "left") )
```

Arguments

x	integer, byte size
standard	character, "IEC" for powers of 1024 ('MiB'), "SI" for powers of 1000 ('MB'), or "Unix" for powers of 1024 ('M'). See details.
units	character, unit to use for all values (optional), one of "auto", "bytes", or an appropriate unit corresponding to standard.
digits	integer, number of digits after decimal point
width	integer, width of number string
sep	character, separator between number and unit
justify	two-element vector specifiy the alignment for the number and unit components of the size. Each element should be one of "none", "left", "right", or "center"

Details

The basic unit used to store information in computers is a bit. Bits are represented as zeroes and ones - binary number system. Although, the binary number system is not the same as the decimal number system, decimal prefixes for binary multiples such as kilo and mega are often used. In the decimal system kilo represent 1000, which is close to $1024 = 2^{10}$ in the binary system. This sometimes causes problems as it is not clear which powers (2 or 10) are used in a notation like 1 kB. To overcome this problem International Electrotechnical Commission (IEC) has provided the following solution to this problem:

Name	System	Symbol	Size	Conversion
byte	binary	B	2^3	8 bits
kilobyte	decimal	kB	10^3	1000 bytes
kibibyte	binary	KiB	2^{10}	1024 bytes
megabyte	decimal	MB	$(10^3)^2$	1000 kilobytes
mebibyte	binary	MiB	$(2^{10})^2$	1024 kibibytes
gigabyte	decimal	GB	$(10^3)^3$	1000 megabytes
gibibyte	binary	GiB	$(2^{10})^3$	1024 mebibytes
terabyte	decimal	TB	$(10^3)^4$	1000 gigabytes
tebibyte	binary	TiB	$(2^{10})^4$	1024 gibibytes
petabyte	decimal	PB	$(10^3)^5$	1000 terabytes
pebibyte	binary	PiB	$(2^{10})^5$	1024 tebibytes
exabyte	decimal	EB	$(10^3)^6$	1000 petabytes
exbibyte	binary	EiB	$(2^{10})^6$	1024 pebibytes
zettabyte	decimal	ZB	$(10^3)^7$	1000 exabytes
zebibyte	binary	ZiB	$(2^{10})^7$	1024 exbibytes
yottabyte	decimal	YB	$(10^3)^8$	1000 zettabytes
yebibyte	binary	YiB	$(2^{10})^8$	1024 zebibytes

where Zi and Yi are GNU extensions to IEC. To get the output in the decimal system (powers of 1000) use `standard="SI"`. To obtain IEC standard (powers of 1024) use `standard="IEC"`.

In addition, single-character units are provided that follow (and extend) the Unix pattern (use `standard="Unix"`):

Name	System	Symbol	Size	Conversion
byte	binary	B	2^3	8 bits
kibibyte	binary	K	2^{10}	1024 bytes
mebibyte	binary	M	$(2^{10})^2$	1024 kibibytes
gibibyte	binary	G	$(2^{10})^3$	1024 mebibytes
tebibyte	binary	T	$(2^{10})^4$	1024 gibibytes
pebibyte	binary	P	$(2^{10})^5$	1024 tebibytes
exbibyte	binary	E	$(2^{10})^6$	1024 pebibytes
zebibyte	binary	Z	$(2^{10})^7$	1024 exbibytes
yottabyte	binary	Y	$(2^{10})^8$	1024 zebibytes

For printout both `digits` and `width` can be specified. If `width` is `NULL`, all values have given

number of digits. If `width` is not `NULL`, output is rounded to a given width and formated similar to human readable format of the Unix `ls`, `df` or `du` shell commands.

Value

Byte size in human readable format as character with proper unit symbols added at the end of the string.

Author(s)

Ales Korosec, Gregor Gorjanc, and Gregory R. Warnes <greg@warnes.net>

References

Wikipedia: https://en.wikipedia.org/wiki/Byte https://en.wikipedia.org/wiki/SI_prefix https://en.wikipedia.org/wiki/Binary_prefix

GNU manual for coreutils: http://www.gnu.org/software/coreutils/manual/html_node/Block-size.html#Block-size

See Also

object.size in package 'gdata', object.size in package 'utils', ll

Examples

```
# Simple example: maximum addressible size of 32 bit pointer
humanReadable(2^32-1)
humanReadable(2^32-1, standard="IEC")
humanReadable(2^32-1, standard="SI")
humanReadable(2^32-1, standard="Unix")

humanReadable(2^32-1, unit="MiB")
humanReadable(2^32-1, standard="IEC",  unit="MiB")
humanReadable(2^32-1, standard="SI",   unit="MB")
humanReadable(2^32-1, standard="Unix", unit="M")

# Vector of sizes
matrix(humanReadable(c(60810, 124141, 124, 13412513), width=4))
matrix(humanReadable(c(60810, 124141, 124, 13412513), width=4, unit="KiB"))

# Specify digits rather than width
matrix(humanReadable(c(60810, 124141, 124, 13412513), width=NULL, digits=2))

# Change the justification
matrix(humanReadable(c(60810, 124141, 124, 13412513), width=NULL,
                     justify=c("right", "right") ))
```

---

installXLSXsupport   *Install perl modules needed for read.xls to support Excel 2007+ XLSX*
                            format

Description

Install perl modules needed for read.xls to support Excel 2007+ XLSX format

Usage

```
installXLSXsupport(perl = "perl", verbose = FALSE)
```

Arguments

perl	Path to perl interpreter (optional).
verbose	If TRUE, show additional messages during processing.

Details

This function calls the perl script 'install_modules.pl' located in the perl subdirectory of the gdata package directory (or inst/perl in the source package). This perl script attempts to use the perl 'CPAN' package, which should be included as part of most perl installations, to automatically download, compile, and install the Compress::Raw::Zlib and Spreadsheet::XLSX perl modules needed for read.xls to support support Excel 2007+ XLSX files into the gdata perl subdirectory.

Since the perl modules are installed into the gdata installation directory, the perl modules will be available until the gdata package is replaced or removed. Since this occurs each time a new version of gdata is installed, installXLSXsupport will need to be run each time a new version of the gdata package is installed.

Further, the binary Compress::Raw::Zlib package installed by installXLSXsupport is tied to the particular version of perl used to compile it, therefore, you will need to re-run installXLSXsupport if you install a different perl distribution.

This installation process will fail if 1) perl is not available on the search path and the perl argument is not used to specify the path of the perl executable, 2) the perl installation is not properly configured for installing binary packages*, 3) if the CPAN module is not present, or 4) if the C compiler specified by the perl installation is not present.

In particular, installXLSXsupport *will fail for the version of perl included with the current RTools.zip package, which is not correctly configured to allow installation of additional perl packages. (The RTools version of perl is installed in a different directory than the perl configuration files expect.)*

The function xlsFormats can be used to see whether XLS and XLSX formats are supported by the currently available perl modules.

Value

Either TRUE indicating that the necessary perl modules have been successfully installed, or FALSE indicating that an error has occured.

See Also

[read.xls](), [xls2csv](), [xlsFormats]()

Examples

```
## Not run:
   installXLSXsupport()

## End(Not run)
```

---

interleave	*Interleave Rows of Data Frames or Matrices*

Description

Interleave rows of data frames or Matrices.

Usage

```
interleave(..., append.source=TRUE, sep=": ", drop=FALSE)
```

Arguments

...	objects to be interleaved
append.source	Boolean Flag. When TRUE (the default) the argument name will be appended to the row names to show the source of each row.
sep	Separator between the original row name and the object name.
drop	boolean flag - When TRUE, matrices containing one column will be converted to vectors.

Details

This function creates a new matrix or data frame from its arguments.

The new object will have all of the rows from the source objects interleaved. IE, it will contain row 1 of object 1, followed by row 1 of object 2, .. row 1 of object 'n', row 2 of object 1, row 2 of object 2, ... row 2 of object 'n' ...

Value

Matrix containing the interleaved rows of the function arguments.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

cbind, rbind, combine

Examples

```
# Simple example
a <- matrix(1:10,ncol=2,byrow=TRUE)
b <- matrix(letters[1:10],ncol=2,byrow=TRUE)
c <- matrix(LETTERS[1:10],ncol=2,byrow=TRUE)
interleave(a,b,c)

# Useful example:
#
# Create a 2-way table of means, standard errors, and # obs

g1 <- sample(letters[1:5], 1000, replace=TRUE)
g2 <- sample(LETTERS[1:3], 1000, replace=TRUE )
dat <- rnorm(1000)

stderr <- function(x) sqrt( var(x,na.rm=TRUE) / nobs(x) )

means  <- tapply(dat, list(g1, g2), mean )
stderrs <- tapply(dat, list(g1, g2), stderr )
ns     <- tapply(dat, list(g1, g2), nobs )
blanks <- matrix( " ", nrow=5, ncol=3)

tab <- interleave( "Mean"=round(means,2),
                   "Std Err"=round(stderrs,2),
                   "N"=ns, " " = blanks, sep=" " )

print(tab, quote=FALSE)

# Using drop to control coercion to a lower dimensions:

m1 <- matrix(1:4)
m2 <- matrix(5:8)

interleave(m1, m2, drop=TRUE)  # This will be coerced to a vector

interleave(m1, m2, drop=FALSE) # This will remain a matrix
```

---

is.what   *Run Multiple is.* Tests on a Given Object*

Description

Run multiple is.* tests on a given object: is.na, is.numeric, and many others.

Usage

```
is.what(object, verbose=FALSE)
```

Arguments

object   any R object.

verbose   whether negative tests should be included in output.

Value

A character vector containing positive tests, or when verbose is TRUE, a data frame showing all test results.

Note

The following procedure is used to look for valid tests:

1. Find all objects named is.* in all loaded environments.

2. Discard objects that are not functions.

3. Include test result only if it is of class "logical", not an NA, and of length 1.

Author(s)

Arni Magnusson, inspired by demo(is.things).

See Also

is.na and is.numeric are commonly used tests.

Examples

```
is.what(pi)
is.what(NA, verbose=TRUE)
is.what(lm(1~1))
is.what(is.what)
```

---

keep   *Remove All Objects, Except Those Specified*

Description

Remove all objects from the user workspace, except those specified.

Usage

```
keep(..., list=character(0), all=FALSE, sure=FALSE)
```

Arguments

...	objects to be kept, specified one by one, quoted or unquoted.
list	character vector of object names to be kept.
all	whether hidden objects (beginning with a .) should be removed, unless explicitly kept.
sure	whether to perform the removal, otherwise return names of objects that would have been removed.

Details

Implemented with safety caps: objects whose name starts with a . are not removed unless all=TRUE, and an explicit sure=TRUE is required to remove anything.

Value

A character vector containing object names, or NULL when sure is TRUE.

Author(s)

Arni Magnusson

See Also

keep is a convenient interface to rm when removing most objects from the user workspace.

Examples

```
data(women, cars)
keep(cars)
## To remove all objects except cars, run:
## keep(cars, sure=TRUE)
```

---

left	*Return the leftmost or rightmost columns of a matrix or dataframe*

Description

Return the leftmost or rightmost or columns of a matrix or dataframe

Usage

```
right(x, n = 6)
left(x, n=6)
```

Arguments

x	Matrix or dataframe
n	Number of columns to return

Value

An object consisting of the leftmost or rightmost n columns of x.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

[first](#), [last](#), [head](#), [tail](#)

Examples

```
m <- matrix( 1:100, ncol=10 )
colnames(m) <- paste("Col",1:10, sep="_")
left(m)
right(m)

d <- as.data.frame(m)
left(d)
right(d)
```

---

ll   *Describe Objects or Elements*

Description

Display name, class, size, and dimensions of each object in a given environment. Alternatively, if the main argument is a list-like object, its elements are listed and described.

Usage

```
ll(pos=1, unit="KB", digits=0, dim=FALSE, sort=FALSE, class=NULL,
   invert=FALSE, ...)
```

Arguments

pos	environment position number, environment name, data frame, list, model, or any object that is.list.
unit	unit for displaying object size: "bytes", "KB", "MB", or first letter.
digits	number of decimals to display when rounding object size.
dim	whether object dimensions should be returned.
sort	whether elements should be sorted by name.
class	character vector for limiting the output to specified classes.
invert	whether to invert the class filter, so specified classes are excluded.
...	passed to ls.

Value

A data frame with named rows and the following columns:

Class	object class.
KB	object size *(see note)*.
Dim	object dimensions *(optional)*.

Note

The name of the object size column is the same as the unit used.

Author(s)

Arni Magnusson, with a contribution by Jim Rogers

See Also

ll is a verbose alternative to [ls](objects in an environment) and [names](elements in a list-like object).

[str](#) and [summary](#) also describe elements in a list-like objects.

[env](#) is a related function that describes all loaded environments.

Examples

```
ll()
ll(all=TRUE)
ll("package:base")
ll("package:base", class="function", invert=TRUE)

data(infert)
ll(infert)
model <- glm(case~spontaneous+induced, family=binomial, data=infert)
ll(model, dim=TRUE)
ll(model, sort=TRUE)
ll(model$family)
```

---

ls.funs   *List function objects*

Description

Return a character vector giving the names of function objects in the specified environment.

Usage

```
ls.funs(...)
```

Arguments

...   Arguments passed to `ls`. See the help for `ls` for details.

Details

This function calls `ls` and then returns a character vector containing only the names of only function objects.

Value

character vector

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

`ls`, `is.function`

Examples

```
## List functions defined in the global environment:
ls.funs()

## List functions available in the base package:
ls.funs("package:base")
```

---

mapLevels   *Mapping levels*

Description

`mapLevels` produces a map with information on levels and/or internal integer codes. As such can be conveniently used to store level mapping when one needs to work with internal codes of a factor and later transfrorm back to factor or when working with several factors that should have the same levels and therefore the same internal coding.

Usage

```
mapLevels(x, codes=TRUE, sort=TRUE, drop=FALSE, combine=FALSE, ...)
mapLevels(x) <- value
```

Arguments

x	object whose levels will be mapped, look into details
codes	boolean, create integer levelsMap (with internal codes) or character levelsMap (with level names)
sort	boolean, sort levels of character x, look into details
drop	boolean, drop unused levels
combine	boolean, combine levels, look into details
...	additional arguments for sort
value	levelsMap or listLevelsMap, output of mapLevels methods or constructed by user, look into details

Value

mapLevels() returns "levelsMap" or "listLevelsMap" objects as described in levelsMap and listLevelsMap section.

Result of mapLevels<- is always a factor with remapped levels or a "list/data.frame" with remapped factors.

mapLevels

mapLevels function was written primarily for work with "factors", but is generic and can also be used with "character", "list" and "data.frame", while "default" method produces error. Here the term levels is also used for unique character values.

When codes=TRUE **integer "levelsMap"** with information on mapping internal codes with levels is produced. Output can be used to transform integer to factor or remap factor levels as described bellow. With codes=FALSE **character "levelsMap"** is produced. The later is usefull, when one would like to remap factors or combine factors with some overlap in levels as described in mapLevels<- section and shown in examples.

sort argument provides possibility to sort levels of "character" x and has no effect when x is a "factor".

Argument combine has effect only in "list" and "data.frame" methods and when codes=FALSE i.e. with **character "levelsMaps"**. The later condition is necesarry as it is not possible to combine maps with different mapping of level names and integer codes. It is assumed that passed "list" and "data.frame" have all components for which methods exist. Otherwise error is produced.

levelsMap and listLevelsMap

Function mapLevels returns a map of levels. This map is of class "levelsMap", which is actually a list of length equal to number of levels and with each component of length 1. Components need not be of length 1. There can be either integer or character "levelsMap". **Integer "levelsMap"** (when codes=TRUE) has names equal to levels and components equal to internal codes. **Character "levelsMap"** (when codes=FALSE) has names and components equal to levels. When mapLevels is applied to "list" or "data.frame", result is of class "listLevelsMap", which is a list of "levelsMap" components described previously. If combine=TRUE, result is a "levelsMap" with all levels in x components.

For ease of inspection, print methods unlists "levelsMap" with proper names. mapLevels<- methods are fairly general and therefore additional convenience methods are implemented to ease the work with maps: is.levelsMap and is.listLevelsMap; as.levelsMap and as.listLevelsMap for coercion of user defined maps; generic "[" and c for both classes (argument recursive can be used in c to coerce "listLevelsMap" to "levelsMap") and generic unique and sort (generic from R 2.4) for "levelsMap".

mapLevels<-

Workhorse under mapLevels<- methods is [levels<-](). mapLevels<- just control the assignment of "levelsMap" (integer or character) or "listLevelsMap" to x. The idea is that map values are changed to map names as indicated in [levels]() examples. **Integer "levelsMap"** can be applied to "integer" or "factor", while **character "levelsMap"** can be applied to "character" or "factor". Methods for "list" and "data.frame" can work only on mentioned atomic components/columns and can accept either "levelsMap" or "listLevelsMap". Recycling occurs, if length of value is not the same as number of components/columns of a "list/data.frame".

Author(s)

Gregor Gorjanc

See Also

[factor](), [levels]() and [unclass]()

Examples

```
## --- Integer levelsMap ---

(f <- factor(sample(letters, size=20, replace=TRUE)))
(mapInt <- mapLevels(f))

## Integer to factor
(int <- as.integer(f))
(mapLevels(int) <- mapInt)
all.equal(int, f)

## Remap levels of a factor
(fac <- factor(as.integer(f)))
(mapLevels(fac) <- mapInt) # the same as levels(fac) <- mapInt
all.equal(fac, f)

## --- Character levelesMap ---

f1 <- factor(letters[1:10])
f2 <- factor(letters[5:14])

## Internal codes are the same, but levels are not
as.integer(f1)
as.integer(f2)
```

```
## Get character levelsMaps and combine them
mapCha1 <- mapLevels(f1, codes=FALSE)
mapCha2 <- mapLevels(f2, codes=FALSE)
(mapCha <- c(mapCha1, mapCha2))

## Remap factors
mapLevels(f1) <- mapCha # the same as levels(f1) <- mapCha
mapLevels(f2) <- mapCha # the same as levels(f2) <- mapCha

## Internal codes are now "consistent" among factors
as.integer(f1)
as.integer(f2)

## Remap characters to get factors
f1 <- as.character(f1); f2 <- as.character(f2)
mapLevels(f1) <- mapCha
mapLevels(f2) <- mapCha

## Internal codes are now "consistent" among factors
as.integer(f1)
as.integer(f2)
```

---

matchcols   *Select columns names matching certain critera*

Description

This function allows easy selection of the column names of an object using a set of inclusion and exclusion critera.

Usage

```
matchcols(object, with, without, method=c("and","or"), ...)
```

Arguments

object	Matrix or dataframe
with, without	Vector of regular expression patterns
method	One of "and" or "or"
...	Optional arguments to grep

Value

Vector of column names which match all (method="and") or any (method="or") of the patterns specified in with, but none of the patterns specified in without.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

grep

Examples

```
# create a matrix with a lot of named columns
x <- matrix( ncol=30, nrow=5 )
colnames(x) <- c("AffyID","Overall Group Means: Control",
                 "Overall Group Means: Moderate",
                 "Overall Group Means: Marked",
                 "Overall Group Means: Severe",
                 "Overall Group StdDev: Control",
                 "Overall Group StdDev: Moderate",
                 "Overall Group StdDev: Marked",
                 "Overall Group StdDev: Severe",
                 "Overall Group CV: Control",
                 "Overall Group CV: Moderate",
                 "Overall Group CV: Marked",
                 "Overall Group CV: Severe",
                 "Overall Model P-value",
                 "Overall Model: (Intercept): Estimate",
                 "Overall Model: Moderate: Estimate",
                 "Overall Model: Marked: Estimate",
                 "Overall Model: Severe: Estimate",
                 "Overall Model: (Intercept): Std. Error",
                 "Overall Model: Moderate: Std. Error",
                 "Overall Model: Marked: Std. Error",
                 "Overall Model: Severe: Std. Error",
                 "Overall Model: (Intercept): t value",
                 "Overall Model: Moderate: t value",
                 "Overall Model: Marked: t value",
                 "Overall Model: Severe: t value",
                 "Overall Model: (Intercept): Pr(>|t|)",
                 "Overall Model: Moderate: Pr(>|t|)",
                 "Overall Model: Marked: Pr(>|t|)",
                 "Overall Model: Severe: Pr(>|t|)")

# Get the columns which give estimates or p-values
# only for marked and severe groups
matchcols(x, with=c("Pr", "Std. Error"),
             without=c("Intercept","Moderate"),
             method="or"
         )

# Get just the column which give the p-value for the intercept
matchcols(x, with=c("Intercept", "Pr") )
```

MedUnits   *Table of conversions between Intertional Standard (SI) and US 'Conventional' Units for common medical measurements.*

Description

Table of conversions between Intertional Standard (SI) and US 'Conventional' Units for common medical measurements.

Usage

```
data(MedUnits)
```

Format

A data frame with the following 5 variables.

Abbreviation Common Abbreviation (mostly missing)

Measurement Measurement Name

ConventionalUnit Conventional Unit

Conversion Conversion factor

SIUnit SI Unit

Details

Medical laboratories and practitioners in the United States use one set of units (the so-called 'Conventional' units) for reporting the results of clinical laboratory measurements, while the rest of the world uses the International Standard (SI) units. It often becomes necessary to translate between these units when participating in international collaborations.

This data set provides constants for converting between SI and US 'Conventional' units.

To perform the conversion from SI units to US 'Conventional' units do:

Measurement in ConventionalUnit = (Measurement in SIUnit) / Conversion

To perform conversion from 'Conventional' to SI units do:

Measurement in SIUnit = (Measurement in ConventionalUnit) * Conversion

Source

https://globalrph.com/medical/conventional-units-international-units/

See Also

The function ConvertMedUnits automates the conversion task.

Examples

```
data(MedUnits)

# show available conversions
MedUnits$Measurement

# utility function
matchUnits <- function(X) MedUnits[ grep(X, MedUnits$Measurement),]

# Convert SI Glucose measurement to 'Conventional' units
GlucoseSI = c(5, 5.4, 5, 5.1, 5.6, 5.1, 4.9, 5.2, 5.5) # in SI Units
GlucoseUS = GlucoseSI / matchUnits("Glucose")$Conversion
cbind(GlucoseSI,GlucoseUS)

# also consider using ConvertMedUnits()
ConvertMedUnits( GlucoseSI, "Glucose", to="US" )
```

---

mv   *Rename an Object*

Description

Rename an object.

Usage

```
mv(from, to, envir = parent.frame())
```

Arguments

from	Character scalar giving the source object name
to	Character scalar giving the desination object name
envir	Environment in which to do the rename

Details

This function renames an object by the value of object a to the name b, and removing a.

Value

Invisibly returns the value of the object.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

rm, assign

Examples

```
a <- 1:10
a
mv("a", "b")
b
exists("a")
```

---

nobs   *Compute the Number of Non-missing Observations*

Description

Compute the number of non-missing observations. Provides a 'default' method to handle vectors, and a method for data frames.

Usage

```
nobs(object, ...)
## Default S3 method:
nobs(object, ...)
## S3 method for class 'data.frame'
nobs(object, ...)
## S3 method for class 'lm'
nobs(object, ...)
```

Arguments

object	Target Object
...	Optional parameters (currently ignored)

Details

Calculate the number of observations in object.

- For numeric vectors, this is simply the number of non-NA elements, as computed by sum(!is.na(object)).

- For dataframe objects, the result is a vector containing the number of non-NA elementes of each column.

The nobs and nobs.lm functions defined in gtools are simply aliases for the functions in the base R stats package, provided for backwards compatibility.

Value

Either single numeric value (for vectors) or a vector of numeric values (for data.frames) giving the number of non-missing values.

Note

The base R package stats now provides a S3 dispatch function for nobs, and methods for for objects of classes "lm", "glm", "nls" and "logLik", as well as a default method.

Since they provided a subset of the the functionality, the base method dispatch (nobs) function and method for "lm" objects (nobs.lm) are, as of gdata version 2.10.1, simply aliases for the equivalent functions in the base R stats package.

Since gdata's default method (nobs.default) processes vectors and hands any other data/object types to stats:::nobs.default.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

[is.na](), [length]()

Examples

```
x <- c(1,2,3,5,NA,6,7,1,NA )
length(x)
nobs(x)

df <- data.frame(x=rnorm(100), y=rnorm(100))
df[1,1] <- NA
df[1,2] <- NA
df[2,1] <- NA

nobs(df)

fit <- lm(y ~ x, data=df)
nobs(fit)
```

---

nPairs   *Number of variable pairs*

Description

nPairs counts the number of pairs between variables.

Usage

```
nPairs(x, margin=FALSE, names=TRUE, abbrev=TRUE, ...)
```

Arguments

x	data.frame or a matrix
margin	logical, calculate the cumulative number of "pairs"
names	logical, add row/col-names to the output
abbrev	logical, abbreviate names
...	other arguments passed to [abbreviate](#)

Details

The class of returned matrix is nPairs and matrix. There is a summary method, which shows the opposite information - counts how many times each variable is known, while the other variable of a pair is not. See examples.

Value

Matrix of order k, where k is the number of columns in x. Values in a matrix represent the number of pairs between columns/variables in x. If margin=TRUE, the number of columns is $k + 1$ and the last column represents the cumulative number of pairing all variables.

Author(s)

Gregor Gorjanc

See Also

[abbreviate](#)

Examples

```
## Test data
test <- data.frame(V1=c(1, 2, 3, 4, 5),
                   V2=c(NA, 2, 3, 4, 5),
                   V3=c(1, NA, NA, NA, NA),
                   V4=c(1, 2, 3, NA, NA))

## Number of variable pairs
nPairs(x=test)

## Without names
nPairs(x=test, names=FALSE)

## Longer names
colnames(test) <- c("Variable1", "Variable2", "Variable3", "Variable4")
nPairs(x=test)

## Margin
nPairs(x=test, margin=TRUE)

## Summary
```

```
summary(object=nPairs(x=test))
```

---

object.size   *Report the Space Allocated for Objects*

Description

Provides an estimate of the memory that is being used to store R objects.

Usage

```
object.size(...)

## S3 method for class 'object_sizes'
is(x)

## S3 method for class 'object_sizes'
as(x)

## S3 method for class 'object_sizes'
c(..., recursive=FALSE)

## S3 method for class 'object_sizes'
format(x, humanReadable=getOption("humanReadable"), standard="IEC", units,
       digits=1, width=NULL, sep=" ", justify = c("right", "left"),
...)

## S3 method for class 'object_sizes'
print(x, quote=FALSE, humanReadable=getOption("humanReadable"),
      standard="IEC", units, digits=1, width=NULL, sep=" ",
      justify = c("right", "left"), ...)
```

Arguments

...	object.size: R objects; print and format: arguments to be passed to other methods.
x	output from object.size
quote	logical, indicating whether or not the result should be printed with surrounding quotes.
humanReadable	logical, use the "human readable" format.
standard,units,digits,width,sep,justify	
	See the man page for [humanReadable]().
recursive	See the man page for [c]().

Details

This is a modified copy of the man page for utils::object.size in R 2.2.1.

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (Sharing amongst elements of a character vector is taken into account, but not that between character vectors in a single object.)

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Associated space (e.g., the environment of a function and what the pointer in a EXTPTRSXP points to) is not included in the calculation.

Object sizes are larger on 64-bit builds than 32-bit ones, but will very likely be the same on different platforms with the same word length and pointer size.

Changes

Class of returned object is c("object_sizes", "numeric") with appropriate print and c methods.

By default object.size outputs size in bytes, but human readable format similar to ls, df or du shell commands can be displayed by calling humanReadable directly, calling print with the argument humanReadable=TRUE, or by setting options(humanReadable=TRUE).

Value

A numeric vector class c("object_sizes", "numeric") containing estimated memory allocation attributable to the objects in bytes.

See Also

object.size in package 'utils' for the standard version of this function, Memory-limits for the design limitations on object size, humanReadable for human readable format.

Examples

```
object.size(letters)
object.size(ls)
## find the 10 largest objects in the base package
allObj <- sapply(ls("package:base"),
            function(x)
                object.size(get(x, envir = baseenv()))
            )

( bigObj <- as.object_sizes(rev(sort(allObj))[1:10] ) )
print(bigObj, humanReadable=TRUE)


as.object_sizes(14567567)



options(humanReadable=TRUE)
```

```
(
    z <- object.size(letters,
                     c(letters, letters),
                     rep(letters, 100),
                     rep(letters, 10000)
                     )
)
is.object_sizes(z)
as.object_sizes(14567567)
```

---

read.xls   *Read Excel files*

Description

Read a Microsoft Excel file into a data frame

Usage

```
read.xls(xls, sheet=1, verbose=FALSE, pattern, na.strings=c("NA","#DIV/0!"),
         ..., method=c("csv","tsv","tab"), perl="perl")
xls2csv(xls, sheet=1, verbose=FALSE, blank.lines.skip=TRUE, ..., perl="perl")
xls2tab(xls, sheet=1, verbose=FALSE, blank.lines.skip=TRUE, ..., perl="perl")
xls2tsv(xls, sheet=1, verbose=FALSE, blank.lines.skip=TRUE, ..., perl="perl")
xls2sep(xls, sheet=1, verbose=FALSE, blank.lines.skip=TRUE, ...,
        method=c("csv","tsv","tab"), perl="perl")
```

Arguments

xls	path to the Microsoft Excel file. Supports "http://", "https://", and "ftp://" URLS.
sheet	name or number of the worksheet to read
verbose	logical flag indicating whether details should be printed as the file is processed.
pattern	if specified, them skip all lines before the first containing this string
perl	name of the perl executable to be called.
method	intermediate file format, "csv" for comma-separated and "tab" for tab-separated
na.strings	a character vector of strings which are to be interpreted as 'NA' values. See [read.table](#) for details.
blank.lines.skip	
	logical flag indicating whether blank lines in the orginal file should be ignored.
...	additional arguments to read.table. The defaults for read.csv() are used.

Details

This function works translating the named Microsoft Excel file into a temporary .csv or .tab file, using the xls2csv or xls2tab Perl script installed as part of this (gdata) package.

Caution: In the conversion to csv, strings will be quoted. This can be problem if you are trying to use the `comment.char` option of `read.table` since the first character of all lines (including comment lines) will be "\"" after conversion.

If you have quotes in your data which confuse the process you may wish to use `read.xls(...,` `quote = '')`. This will cause the quotes to be regarded as data and you will have to then handle the quotes yourself after reading the file in.

Caution: If you call "xls2csv" directly, is your responsibility to close and delete the file after using it.

Value

"read.xls" returns a data frame.

"xls2sep" returns a temporary file in the specified format. "xls2csv" and "xls2tab" are simply wrappers for "xls2sep" specifying method as "csv" or "tab", respectively.

Note

Either a working version of Perl must be present in the executable search path, or the exact path of the perl executable must be provided via the `perl` argument. See the examples below for an illustration.

Author(s)

Gregory R. Warnes <greg@warnes.net>, Jim Rogers <james.a.rogers@pfizer.com>, and Gabor Grothendieck <ggrothendieck@gmail.com>.

References

http://www.analytics.washington.edu/statcomp/downloads/xls2csv

See Also

read.csv

Examples

```
# iris.xls is included in the gregmisc package for use as an example
xlsfile <- file.path(path.package('gdata'),'xls','iris.xls')
xlsfile

iris <- read.xls(xlsfile) # defaults to csv format
iris <- read.xls(xlsfile,method="csv") # specify csv format
iris <- read.xls(xlsfile,method="tab") # specify tab format

head(iris)  # look at the top few rows
```

```
  ## Not run:
   ## Example specifying exact Perl path for default MS-Windows install of
   ## ActiveState perl
   iris <- read.xls(xlsfile, perl="C:/perl/bin/perl.exe")

## End(Not run)

   ## Not run:
   ## Example specifying exact Perl path for Unix systems
   iris <- read.xls(xlsfile, perl="/usr/bin/perl")

   ## finding perl
   ## (read.xls automatically calls findPerl so this is rarely needed)
   perl <- gdata:::findPerl("perl")
   iris <- read.xls(xlsfile, perl=perl)

## End(Not run)

   ## Not run:
   ## read xls file from net
   nba.url <- "http://mgtclass.mgt.unm.edu/Bose/Excel/Tutorial.05/Cases/NBA.xls"
   nba <- read.xls(nba.url)

## End(Not run)

   ## Not run:
   ## read xls file ignoring all lines prior to first containing State
   crime.url <- "http://www.jrsainfo.org/jabg/state_data2/Tribal_Data00.xls"
   crime <- read.xls(crime.url, pattern = "State")

   ## use of xls2csv - open con, print two lines, close con
   con <- xls2csv(crime.url)
   print(readLines(con, 2))
   file.remove(summary(con)$description)

## End(Not run)

   ## Examples demonstrating selection of specific 'sheets'
   ## from the example XLS file 'ExampleExcelFile.xls'
   exampleFile <- file.path(path.package('gdata'),'xls',
                            'ExampleExcelFile.xls')
   exampleFile2007 <- file.path(path.package('gdata'),'xls',
                            'ExampleExcelFile.xlsx')

   ## see the number and names of sheets:
   sheetCount(exampleFile)
   if( 'XLSX' %in% xlsFormats() )  ## if XLSX is supported..
     sheetCount(exampleFile2007)
```

```
## show names of shets in the file
sheetNames(exampleFile)
if( 'XLSX' %in% xlsFormats() )  ## if XLSX is supported..
sheetNames(exampleFile2007)

data <- read.xls(exampleFile)          ## default is first worksheet
data <- read.xls(exampleFile, sheet=2) ## second worksheet by number
data <- read.xls(exampleFile, sheet="Sheet Second",v=TRUE) ## and by name

## load the third worksheet, skipping the first two non-data lines...
if( 'XLSX' %in% xlsFormats() )  ## if XLSX is supported..
  data <- read.xls(exampleFile2007, sheet="Sheet with initial text", skip=2)

## load a file containing data and column names using latin-1
## characters
latinFile <- file.path(path.package('gdata'),'xls','latin-1.xls')
latin1 <- read.xls(latinFile, fileEncoding="latin1")
colnames(latin1)
```

---

rename.vars   *Remove or rename variables in a dataframe*

Description

Remove or rename a variables in a data frame.

Usage

```
rename.vars(data, from="", to="", info=TRUE)
remove.vars(data, names="", info=TRUE)
```

Arguments

data	dataframe to be modified.
from	character vector containing the current name of each variable to be renamed.
to	character vector containing the new name of each variable to be renamed.
names	character vector containing the names of variables to be removed.
info	boolean value indicating whether to print details of the removal/renaming. Defaults to TRUE.

Value

The updated data frame with variables listed in from renamed to the corresponding element of to.

Author(s)

Code by Don MacQueen <macq@llnl.gov>. Documentation by Gregory R. Warnes <greg@warnes.net>

See Also

[names](#), [colnames](#), [data.frame](#)

Examples

```
data <- data.frame(x=1:10,y=1:10,z=1:10)
names(data)
data <- rename.vars(data, c("x","y","z"), c("first","second","third"))
names(data)

data <- remove.vars(data, "second")
names(data)
```

---

reorder.factor   *Reorder the Levels of a Factor*

Description

Reorder the levels of a factor

Usage

```
## S3 method for class 'factor'
reorder(x, X, FUN, ..., order=is.ordered(x), new.order, sort=mixedsort)
```

Arguments

x	factor
X	auxillary data vector
FUN	function to be applied to subsets of X determined by x, to determine factor order
...	optional parameters to FUN
order	logical value indicating whether the returned object should be an [ordered](#) factor
new.order	a vector of indexes or a vector of label names giving the order of the new factor levels
sort	function to use to sort the factor level names, used only when new.order is missing

Details

This function changes the order of the levels of a factor. It can do so via three different mechanisms, depending on whether, X *and* FUN, new.order or sort are provided.

If X *and* Fun are provided: The data in X is grouped by the levels of x and FUN is applied. The groups are then sorted by this value, and the resulting order is used for the new factor level names.

If new.order is a numeric vector, the new factor level names are constructed by reordering the factor levels according to the numeric values. If new.order is a chraccter vector, new.order gives the list of new factor level names. In either case levels omitted from new.order will become missing (NA) values.

If sort is provided (as it is by default): The new factor level names are generated by calling the function specified by sort to the existing factor level *names*. With sort=mixedsort (the default) the factor levels are sorted so that combined numeric and character strings are sorted in according to character rules on the character sections (including ignoring case), and the numeric rules for the numeric sections. See mixedsort for details.

Value

A new factor with reordered levels

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

factor and reorder

Examples

```
# Create a 4 level example factor
trt <- factor( sample( c("PLACEBO", "300 MG", "600 MG", "1200 MG"),
                100, replace=TRUE ) )
summary(trt)
# Note that the levels are not in a meaningful order.

# Change the order to something useful..
# - default "mixedsort" ordering
trt2 <- reorder(trt)
summary(trt2)
# - using indexes:
trt3 <- reorder(trt, new.order=c(4, 2, 3, 1))
summary(trt3)
# - using label names:
trt4 <- reorder(trt, new.order=c("PLACEBO", "300 MG", "600 MG", "1200 MG"))
summary(trt4)
# - using frequency
trt5 <- reorder(trt, X=rnorm(100), FUN=mean)
```

```
summary(trt5)

# Drop out the '300 MG' level
trt6 <- reorder(trt, new.order=c("PLACEBO", "600 MG", "1200 MG"))
summary(trt6)
```

---

resample   *Consistent Random Samples and Permutations*

Description

resample takes a sample of the specified size from the elements of x using either with or without replacement.

Usage

```
resample(x, size, replace = FALSE, prob = NULL)
```

Arguments

x	A numeric, complex, character or logical vector from which to choose.
size	Non-negative integer giving the number of items to choose.
replace	Should sampling be with replacement?
prob	A vector of probability weights for obtaining the elements of the vector being sampled.

Details

resample differs from the S/R sample function in resample always considers x to be a vector of elements to select from, while sample treats a vector of length one as a special case and samples from 1:x. Otherwise, the functions have identical behavior.

Value

Vector of the same length as the input, with the elements permuted.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

sample

Examples

```
## sample behavior differs if first argument is scalar vs vector
sample( c(10) )
sample( c(10,10) )

## resample has the consistent behavior for both cases
resample( c(10) )
resample( c(10,10) )
```

---

sheetCount   *Count or list sheet names in Excel spreadsheet files.*

Description

Count or list sheet names in Excel spreadsheet files.

Usage

```
sheetCount(xls, verbose = FALSE, perl = "perl")
sheetNames(xls, verbose = FALSE, perl = "perl")
```

Arguments

xls	File path to spreadsheet. Supports "http://", "https://", and "ftp://" URLS.
verbose	If TRUE, show additional messages during processing.
perl	Path to perl interpreter.

Value

sheetCount returns the number of sheets in the spreadsheet. sheetNames returns the names of sheets in the spreadsheet.

See Also

[read.xls](#), [xls2csv](#).

Examples

```
    xlsfile <- system.file("xls", "iris.xls", package = "gdata")
    xlsfile

    sheetCount(xlsfile)


    exampleFile <- file.path(path.package('gdata'),'xls',
                             'ExampleExcelFile.xls')
```

```
exampleFile2007 <- file.path(path.package('gdata'),'xls',
                             'ExampleExcelFile.xlsx')

sheetCount(exampleFile)

if( 'XLSX' %in% xlsFormats() )  # if XLSX is supported..
  sheetNames(exampleFile2007)
```

startsWith	*Determine if a character string "starts with" with the specified characters.*

Description

Determine if a character string "starts with" with the specified characters.

Usage

```
startsWith(str, pattern, trim=FALSE, ignore.case=FALSE)
```

Arguments

str	character vector to test
pattern	characters to check for
trim	Logical flag indicating whether leading whitespace should be removed from str before testing for a match.
ignore.case	Logical flag indicating whether case should be ignored when testing for a match.

Details

This function returns TRUE for each element of the vector str where pattern occurs at the beginning of the string. If trim is TRUE, leading whitespace is removed from the elements of str before the test is performed. If ignore.case is TRUE, character case is ignored.

Value

Boolean vector of the same length as str.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

substr, trim

Examples

```
## simplest example:
startsWith( 'Testing', 'Test')

## vector examples
s <- c('Testing', ' Testing', 'testing', 'Texting')
names(s) <- s

startsWith(s, 'Test')                    # ' Testing', 'testing', and 'Texting' do not match
startsWith(s, 'Test', trim=TRUE)         # Now ' Testing' matches
startsWith(s, 'Test', ignore.case=TRUE) # Now 'testing' matches
```

---

trim   *Remove leading and trailing spaces from character strings*

Description

Remove leading and trailing spaces from character strings and other related objects.

Usage

```
trim(s, recode.factor=TRUE, ...)
```

Arguments

s	object to be processed
recode.factor	should levels of a factor be recoded, see below
...	arguments passed to other methods, currently only to reorder.factor for factors

Details

trim is a generic function, where default method does nothing, while method for character s trims its elements and method for factor s trims levels. There are also methods for list and data.frame.

Trimming character strings can change the sort order in some locales. For factors, this can affect the coding of levels. By default, factor levels are recoded to match the trimmed sort order, but this can be disabled by setting recode.factor=FALSE. Recoding is done with reorder.factor.

Value

s with all leading and trailing spaces removed in its elements.

Author(s)

Gregory R. Warnes <greg@warnes.net> with contributions by Gregor Gorjanc

See Also

[trimws](), [sub,gsub]() as well as argument `strip.white` in [read.table]() and [reorder.factor]()

Examples

```
s <- "   this is an example string   "
trim(s)

f <- factor(c(s, s, " A", " B ", "  C ", "D "))
levels(f)

trim(f)
levels(trim(f))

trim(f, recode.factor=FALSE)
levels(trim(f, recode.factor=FALSE))

l <- list(s=rep(s, times=6), f=f, i=1:6)
trim(l)

df <- as.data.frame(l)
trim(df)
```

---

trimSum	*Trim a vector such that the last/first value represents the sum of trimmed values*

Description

`trimSum` trims (shortens) a vector in such a way that the last or first value represents the sum of trimmed values. User needs to specify the desired length of a trimmed vector.

Usage

```
trimSum(x, n, right=TRUE, na.rm=FALSE, ...)
```

Arguments

x	numeric, a vector of numeric values
n	numeric, desired length of the output
right	logical, trim on the right/bottom or the left/top side
na.rm	logical, remove NA values when applying a function
...	arguments passed to other methods - currently not used

Value

Trimmed vector with a last/first value representing the sum of trimmed values

Author(s)

Gregor Gorjanc

See Also

[trim](#)

Examples

```
x <- 1:10
trimSum(x, n=5)
trimSum(x, n=5, right=FALSE)

x[9] <- NA
trimSum(x, n=5)
trimSum(x, n=5, na.rm=TRUE)
```

---

unknownToNA   *Change unknown values to NA and vice versa*

Description

Unknown or missing values (NA in R) can be represented in various ways (as 0, 999, etc.) in different programs. isUnknown, unknownToNA, and NAToUnknown can help to change unknown values to NA and vice versa.

Usage

```
isUnknown(x, unknown=NA, ...)
unknownToNA(x, unknown, warning=FALSE, ...)
NAToUnknown(x, unknown, force=FALSE, call.=FALSE, ...)
```

Arguments

x	generic, object with unknown value(s)
unknown	generic, value used instead of NA
warning	logical, issue warning if x already has NA
force	logical, force to apply already existing value in x
...	arguments pased to other methods (as.character for POSIXlt in case of isUnknown)
call.	logical, look in [warning](#)

Details

This functions were written to handle different variants of "other NA" like representations that are usually used in various external data sources. unknownToNA can help to change unknown values to NA for work in R, while NAToUnknown is meant for the opposite and would usually be used prior to export of data from R. isUnknown is utility function for testing for unknown values.

All functions are generic and the following classes were tested to work with latest version: "integer", "numeric", "character", "factor", "Date", "POSIXct", "POSIXlt", "list", "data.frame" and "matrix". For others default method might work just fine.

unknownToNA and isUnknown can cope with multiple values in unknown, but those should be given as a "vector". If not, coercing to vector is applied. Argument unknown can be feed also with "list" in "list" and "data.frame" methods.

If named "list" or "vector" is passed to argument unknown and x is also named, matching of names will occur.

Recycling occurs in all "list" and "data.frame" methods, when unknown argument is not of the same length as x and unknown is not named.

Argument unknown in NAToUnknown should hold value that is not already present in x. If it does, error is produced and one can bypass that with force=TRUE, but be warned that there is no way to distinguish values after this action. Use at your own risk! Anyway, warning is issued about new value in x. Additionally, caution should be taken when using NAToUnknown on factors as additional level (value of unknown) is introduced. Then, as expected, unknownToNA removes defined level in unknown. If unknown="NA", then "NA" is removed from factor levels in unknownToNA due to consistency with conversions back and forth.

Unknown representation in unknown should have the same class as x in NAToUnknown, except in factors, where unknown value is coerced to character anyway. Silent coercing is also applied, when "integer" and "numeric" are in question. Otherwise warning is issued and coercing is tried. If that fails, R introduces NA and the goal of NAToUnknown is not reached.

NAToUnknown accepts only single value in unknown if x is atomic, while "list" and "data.frame" methods accept also "vector" and "list".

"list/data.frame" methods can work on many components/columns. To reduce the number of needed specifications in unknown argument, default unknown value can be specified with component ".default". This matches component/column ".default" as well as all other undefined components/columns! Look in examples.

Value

unknownToNA and NAToUnknown return modified x. isUnknown returns logical values for object x.

Author(s)

Gregor Gorjanc

See Also

[is.na](is.na)

Examples

```
xInt <- c(0, 1, 0, 5, 6, 7, 8, 9, NA)
isUnknown(x=xInt, unknown=0)
isUnknown(x=xInt, unknown=c(0, NA))
(xInt <- unknownToNA(x=xInt, unknown=0))
(xInt <- NAToUnknown(x=xInt, unknown=0))

xFac <- factor(c("0", 1, 2, 3, NA, "NA"))
isUnknown(x=xFac, unknown=0)
isUnknown(x=xFac, unknown=c(0, NA))
isUnknown(x=xFac, unknown=c(0, "NA"))
isUnknown(x=xFac, unknown=c(0, "NA", NA))
(xFac <- unknownToNA(x=xFac, unknown="NA"))
(xFac <- NAToUnknown(x=xFac, unknown="NA"))

xList <- list(xFac=xFac, xInt=xInt)
isUnknown(xList, unknown=c("NA", 0))
isUnknown(xList, unknown=list("NA", 0))
tmp <- c(0, "NA")
names(tmp) <- c(".default", "xFac")
isUnknown(xList, unknown=tmp)
tmp <- list(.default=0, xFac="NA")
isUnknown(xList, unknown=tmp)

(xList <- unknownToNA(xList, unknown=tmp))
(xList <- NAToUnknown(xList, unknown=999))
```

---

unmatrix	*Convert a matrix into a vector, with appropriate names*

Description

Convert a matrix into a vector, with element names constructed from the row and column names of
the matrix.

Usage

```
unmatrix(x, byrow=FALSE)
```

Arguments

x   matrix

byrow   Logical. If `FALSE`, the elements within columns will be adjacent in the resulting
                vector, otherwise elements within rows will be adjacent.

Value

A vector with names constructed from the row and column names from the matrix. If the the row or column names are missing, ('r1', 'r2', ..,) or ('c1', 'c2', ..) will be used as appropriate.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

as.vector

Examples

```
# simple, useless example
m <- matrix( letters[1:10], ncol=5)
m
unmatrix(m)

# unroll model output
x <- rnorm(100)
y <- rnorm(100, mean=3+5*x, sd=0.25)
m <- coef( summary(lm( y ~ x )) )
unmatrix(m)
```

---

update.list   *Update the elements of a list*

Description

Function to update the elements of a list to contain all of the named elements of a new list, over-writing elements with the same name, and (optionally) copying unnamed elements.

Usage

```
## S3 method for class 'list'
update(object,
       new,
       unnamed=FALSE,
       ...)
```

Arguments

`object`	Object to be updated.
`new`	List or dataframe containing new elements.
`unnamed`	Logical. If `TRUE`, unnamed elements of `new` will be appended to `object`.
`...`	optional method arguments (ignored)

Value

A new list constructed from the elements of `object` by merging the elements of `old` and `new` with named items of `new` replacing the corresponding elements of `old`. Unnamed elements of `new` will be appened unless unnamed=FALSE.

Note

This method can be called directly, or as via the S3 base method for `update`.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

[update](#), [merge](#)

Examples

```
old <- list(a=1,b="red",c=1.37)
new <- list(b="green",c=2.4)

update(old, new)
update.list(old,new)  # equivalent

older <- list(a=0, b="orange", 4, 5, 6)
newer <- list(b="purple", 7, 8, 9)
update(older, newer)              # ignores unnamed elements of newer
update(older, newer, unnamed=TRUE) # appends unnamed elements of newer
```

---

upperTriangle	*Extract or replace the upper/lower triangular portion of a matrix*

Description

Extract or replace the upper/lower triangular portion of a matrix

Usage

```
upperTriangle(x, diag=FALSE, byrow=FALSE)
upperTriangle(x, diag=FALSE, byrow=FALSE) <- value
lowerTriangle(x, diag=FALSE, byrow=FALSE)
lowerTriangle(x, diag=FALSE, byrow=FALSE) <- value
```

Arguments

x	Matrix
diag	Logical. If TRUE, include the matrix diagonal.
byrow	Logical. If FALSE, return/replace elements in column-wise order. If TRUE, return/replace elements in row-wise order.
value	Either a single value or a vector of length equal to that of the current upper/lower triangular. Should be of a mode which can be coerced to that of x.

Value

upperTriangle(x) and lowerTriangle(x) return the upper or lower triangle of matrix x, respectively. The assignment forms replace the upper or lower triangular area of the matrix with the provided value(s).

Note

By default, the elements are returned/replaced in R's default column-wise order. Thus

```
lowerTriangle(x) <- upperTriangle(x)
```

will not yield a symmetric matrix. Instead use:

```
lowerTriangle(x) <- upperTriangle(x, byrow=TRUE)
```

or equivalently:

```
lowerTriangle(x, byrow=TRUE) <- upperTriangle(x)
```

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

diag, lower.tri, upper.tri

Examples

```
x <- matrix( 1:25, nrow=5, ncol=5)
x
upperTriangle(x)
upperTriangle(x, diag=TRUE)
upperTriangle(x, diag=TRUE, byrow=TRUE)


lowerTriangle(x)
lowerTriangle(x, diag=TRUE)
lowerTriangle(x, diag=TRUE, byrow=TRUE)

upperTriangle(x) <- NA
x

upperTriangle(x, diag=TRUE) <- 1:15
x

lowerTriangle(x) <- NA
x

lowerTriangle(x, diag=TRUE) <- 1:15
x

## Copy lower triangle into upper triangle to make
## the matrix (diagonally) symmetric
x <- matrix(LETTERS[1:25], nrow=5, ncol=5, byrow=TRUE)
x
lowerTriangle(x) = upperTriangle(x, byrow=TRUE)
x
```

---

wideByFactor   *Create multivariate data by a given factor*

Description

wideByFactor modifies data.frame in such a way that variables are "separated" into several columns by factor levels.

Usage

```
wideByFactor(x, factor, common, sort=TRUE, keepFactor=TRUE)
```

Arguments

`x`	data frame
`factor`	character, column name of a factor by which variables will be divided
`common`	character, column names of (common) columns that should not be divided
`sort`	logical, sort resulting data frame by factor levels
`keepFactor`	logical, keep the 'factor' column

Details

Given data frame is modified in such a way, that output represents a data frame with $c + f + n * v$ columns, where c is a number of common columns for all levels of a factor, f is a factor column, n is a number of levels in factor f and v is a number of variables that should be divided for each level of a factor. Number of rows stays the same!

Value

A data frame where divided variables have sort of "diagonalized" structure

Author(s)

Gregor Gorjanc

See Also

reshape in the **stats** package, melt and cast in the **reshape** package

Examples

```
n <- 10
f <- 2
tmp <- data.frame(y1=rnorm(n=n),
                  y2=rnorm(n=n),
                  f1=factor(rep(letters[1:f], n/2)),
                  f2=factor(c(rep(c("M"), n/2), rep(c("F"), n/2))),
                  c1=1:n,
                  c2=2*(1:n))

wideByFactor(x=tmp, factor="f1", common=c("c1", "c2", "f2"))
wideByFactor(x=tmp, factor="f1", common=c("c1", "c2"))
```

---

write.fwf   *Write object in fixed width format*

Description

write.fwf writes object in *f*ixed *w*idth *f*ormat.

Usage

```
write.fwf(x, file="", append=FALSE, quote=FALSE, sep=" ", na="",
  rownames=FALSE, colnames=TRUE, rowCol=NULL, justify="left",
  formatInfo=FALSE, quoteInfo=TRUE, width=NULL, eol="\n",
  qmethod=c("escape", "double"),  scientific=TRUE, ...)
```

Arguments

x	data.frame or matrix, the object to be written
file	character, name of file or connection, look in `write.table` for more
append	logical, append to existing data in file
quote	logical, quote data in output
na	character, the string to use for missing values i.e. NA in the output
sep	character, separator between columns in output
rownames	logical, print row names
colnames	logical, print column names
rowCol	character, rownames column name
justify	character, alignment of character columns; see `format`
formatInfo	logical, return information on number of levels, widths and format
quoteInfo	logical, should `formatInfo` account for quotes
width	numeric, width of the columns in the output
eol	the character(s) to print at the end of each line (row). For example, 'eol="\r\n"' will produce Windows' line endings on a Unix-alike OS, and 'eol="\r"' will produce files as expected by Mac OS Excel 2004.
qmethod	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of '"escape"' (default), in which case the quote character is escaped in C style by a backslash, or '"double"', in which case it is doubled. You can specify just the initial letter.
scientific	logical, if TRUE, allow numeric values to be formatted using scientific notation.
...	further arguments to `format.info` and `format`

Details

While *F*ixed *w*idth *f*ormat is no longer widely used, it remains common in some disciplines.

Output is similar to print(x) or format(x). Formatting is done completely by [format](#) on a column basis. Columns in the output are by default separated with a space i.e. empty column with a width of one character, but that can be changed with sep argument as passed to [write.table](#) via ....

As mentioned formatting is done completely by [format](#). Arguments can be passed to format via ... to further modify the output. However, note that the returned formatInfo might not properly account for this, since [format.info](#) (which is used to collect information about formatting) lacks the arguments of [format](#).

quote can be used to quote fields in the output. Since all columns of x are converted to character (via [format](#)) during the output, all columns will be quoted! If quotes are used, [read.table](#) can be easily used to read the data back into R. Check examples. Do read the details about quoteInfo argument.

Use only *true* character, i.e., avoid use of tabs, i.e., "\t", or similar separators via argument sep. Width of the separator is taken as the number of characters evaluated via [nchar](#)(sep).

Use argument na to convert missing/unknown values. Only single value can be specified. Use [NAToUnknown](#) prior to export if you need greater flexibility.

If rowCol is not NULL and rownames=TRUE, rownames will also have column name with rowCol value. This is mainly for flexibility with tools outside R. Note that (at least in R 2.4.0) it is not "easy" to import data back to R with [read.fwf](#) if you also export rownames. This is the reason, that default is rownames=FALSE.

Information about format of output will be returned if formatInfo=TRUE. Returned value is described in value section. This information is gathered by [format.info](#) and care was taken to handle numeric properly. If output contains rownames, values account for this. Additionally, if rowCol is not NULL returned values contain also information about format of rownames.

If quote=TRUE, the output is of course wider due to quotes. Return value (with formatInfo=TRUE) can account for this in two ways; controlled with argument quoteInfo. However, note that there is no way to properly read the data back to R if quote=TRUE & quoteInfo=FALSE arguments were used for export. quoteInfo applies only when quote=TRUE. Assume that there is a file with quoted data as shown bellow (column numbers in first three lines are only for demonstration of the values in the output).

```
123456789 12345678 # for position
123 1234567 123456 # for width with quoteInfo=TRUE
 1   12345   1234  # for width with quoteInfo=FALSE
"a" "hsgdh" "   9"
" " "   bb" " 123"
```

With quoteInfo=TRUE write.fwf will return

```
colname position width
V1             1     3
V2             5     7
V3            13     6
```

or (with quoteInfo=FALSE)

```
colname position width
V1             2     1
V2             6     5
V3            14     4
```

Argument `width` can be used to increase the width of the columns in the output. This argument
is passed to the width argument of format function. Values in `width` are recycled if there is less
values than the number of columns. If the specified width is to short in comparison to the "width"
of the data in particular column, error is issued.

Value

Besides its effect to write/export data `write.fwf` can provide information on format and width. A
data.frame is returned with the following columns:

colname	name of the column
nlevels	number of unique values (unused levels of factors are dropped), 0 for numeric column
position	starting column number in the output
width	width of the column
digits	number of digits after the decimal point
exp	width of exponent in exponential representation; 0 means there is no exponential representation, while 1 represents exponent of length one i.e. `1e+6` and 2 `1e+06` or `1e+16`

Author(s)

Gregor Gorjanc

See Also

format.info, format, NAToUnknown, write.table, read.fwf, read.table and trim

Examples

```
## Some data
num <- round(c(733070.345678, 1214213.78765456, 553823.798765678,
               1085022.8876545678,  571063.88765456, 606718.3876545678,
             1053686.6, 971024.187656, 631193.398765456, 879431.1),
          digits=3)

testData <- data.frame(num1=c(1:10, NA),
                       num2=c(NA, seq(from=1, to=5.5, by=0.5)),
                       num3=c(NA, num),
                       int1=c(as.integer(1:4), NA, as.integer(4:9)),
                       fac1=factor(c(NA, letters[1:9], "hjh")),
                       fac2=factor(c(letters[6:15], NA)),
                       cha1=c(letters[17:26], NA),
```

```
                            cha2=c(NA, "longer", letters[25:17]),
                            stringsAsFactors=FALSE)
levels(testData$fac1) <- c(levels(testData$fac1), "unusedLevel")
testData$Date <- as.Date("1900-1-1")
testData$Date[2] <- NA
testData$POSIXt <- as.POSIXct(strptime("1900-1-1 01:01:01",
                                        format="%Y-%m-%d %H:%M:%S"))
testData$POSIXt[5] <- NA

## Default
write.fwf(x=testData)

## NA should be -
write.fwf(x=testData, na="-")
## NA should be -NA-
write.fwf(x=testData, na="-NA-")

## Some other separator than space
write.fwf(x=testData[, 1:4], sep="-mySep-")

## Force wider columns
write.fwf(x=testData[, 1:5], width=20)

## Show effect of 'scienfic' option
testData$num3 <- testData$num3 * 1e8
write.fwf(testData, scientific=TRUE)
write.fwf(testData, scientific=FALSE)
testData$num3 <- testData$num3 / 1e8

## Write to file and report format and fixed width information
file <- tempfile()
formatInfo <- write.fwf(x=testData, file=file, formatInfo=TRUE)
formatInfo

## Read exported data back to R (note +1 due to separator)
## ... without header
read.fwf(file=file, widths=formatInfo$width + 1, header=FALSE, skip=1,
         strip.white=TRUE)

## ... with header - via postimport modfication
tmp <- read.fwf(file=file, widths=formatInfo$width + 1, skip=1,
                strip.white=TRUE)
colnames(tmp) <- read.table(file=file, nrow=1, as.is=TRUE)
tmp

## ... with header - persuading read.fwf to accept header properly
## (thanks to Marc Schwartz)
read.fwf(file=file, widths=formatInfo$width + 1, strip.white=TRUE,
         skip=1, col.names=read.table(file=file, nrow=1, as.is=TRUE))

## ... with header - with the use of quotes
write.fwf(x=testData, file=file, quote=TRUE)
read.table(file=file, header=TRUE, strip.white=TRUE)
```

```
## Tidy up
unlink(file)
```

---

xlsFormats   *Check which file formats are supported by read.xls*

Description

Check which file formats are supported by read.xls

Usage

```
xlsFormats(perl = "perl", verbose = FALSE)
```

Arguments

perl   Path to perl interpreter (optional).

verbose   If TRUE, show additional messages during processing.

Value

Vector of supported formats, possible elements are 'XLS' and 'XLSX'.

See Also

[read.xls](), [xls2csv]().

Examples

```
xlsFormats()
```

Index