Package ‘gdata’

October 16, 2023

Version 3.0.0
Date 2023-10-09
Title Various R Programming Tools for Data Manipulation
Imports gtools, methods, stats, utils
Suggests RUnit

Description Various R programming tools for data manipulation, including medical unit conversions, combining objects, character vector operations, factor manipulation, obtaining information about R objects, generating fixed-width format files, extracting components of date & time objects, operations on columns of data frames, matrix operations, operations on vectors, operations on data frames, value of last evaluated expression, and a resample() wrapper for sample() that ensures consistent behavior for both scalar and vector arguments.

License GPL-2
URL https://github.com/r-gregmisc/gdata

BugReports https://github.com/r-gregmisc/gdata/issues

NeedsCompilation no

Author Gregory R. Warnes [aut], Gregor Gorjanc [aut], Arni Magnusson [aut, cre], Liviu Andronic [aut], Jim Rogers [aut], Don MacQueen [aut], Ales Korosec [aut], Ben Bolker [ctb], Michael Chirico [ctb], Gabor Grothendieck [ctb], Thomas Lumley [ctb], Brian Ripley [ctb], inoui llc [fnd]

Maintainer Arni Magnusson <thisisarni@gmail.com>
Repository CRAN

Date/Publication 2023-10-16 19:30:14 UTC
R topics documented:

gdata-package ................................................................. 3
ans .......................................................... 4
Args .......................................................... 5
bindData .......................................................... 6
case .......................................................... 7
cbindX .......................................................... 8
centerText ......................................................... 9
combine .......................................................... 10
ConvertMedUnits ..................................................... 11
drop.levels ......................................................... 13
duplicated2 .......................................................... 14
eav .......................................................... 15
first .......................................................... 16
frameApply ......................................................... 17
gdata-defunct ....................................................... 19
getYear .......................................................... 19
humanReadable ....................................................... 21
interleave ........................................................ 23
is.what ........................................................ 25
keep ........................................................ 26
left ........................................................ 27
ll ........................................................ 28
ls.funs ........................................................ 30
mapLevels ........................................................ 31
matchcols ......................................................... 33
MedUnits ........................................................ 35
mv ........................................................ 36
nobs ........................................................ 37
nPairs ........................................................ 38
object_size ......................................................... 40
rename.vars ......................................................... 42
reorder.factor ..................................................... 43
resample ........................................................ 44
startsWith ......................................................... 45
trim ........................................................ 47
trimSum ........................................................ 48
unknownToNA ..................................................... 49
unmatrix ........................................................ 51
update.list ......................................................... 52
upperTriangle ....................................................... 53
wideByFactor ....................................................... 55
write.fwf ......................................................... 56

Index 61
Various R Programming Tools for Data Manipulation

Description

Various R programming tools for data manipulation, including:

- Medical unit conversions: ConvertMedUnits, MedUnits
- Combining objects: link(bindData), cbindX, combine, interleave
- Character vector operations: centerText, startsWith, trim
- Factor manipulation: levels, reorder.factor, mapLevels
- Obtaining information about R objects: object_size, env, humanReadable, is.what, ll, keep, ls.funs, Args, nPairs, nobs
- Generating fixed-width format files: write.fwf
- Extracting 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
- Wrapper for sample that ensures consistent behavior for both scalar and vector arguments: resample

Note

browseVignettes() shows package vignettes.

Author(s)

Gregory R. Warnes, Gregor Gorjanc, Arni Magnusson, Liviu Andronic, Jim Rogers, Don MacQueen, and Ales Korosec, with contributions by Ben Bolker, Michael Chirico, Gabor Grothendieck, Thomas Lumley, and Brian Ripley.
Description

The function returns the value of the last evaluated *top-level* expression, which is always assigned to \( \text{.Last.value} \) (in package:base).

Usage

\[
\text{ans()}
\]

Details

This function retrieves \( \text{.Last.value} \). For more details see \( \text{.Last.value} \).

Value

\( \text{.Last.value} \)

Author(s)

Liviu Andronic

See Also

\( \text{.Last.value}, \text{eval} \)

Examples

\begin{verbatim}
2+2    # Trivial calculation
\text{ans()}    # See the answer again
\text{gamma}(1:15)    # Some intensive calculation
\text{fac14} <- \text{ans()}    # store the results into a variable
\text{rnorm}(20)    # Generate some standard normal values
\text{ans()}^2    # Convert to \text{Chi-square}(1) values
\text{stem}(\text{ans()})    # Now show a stem-and-leaf table
\end{verbatim}
**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 appropriate solution, `bindData` might perform an appropriate binding for two data frames. This is especially useful when some variables are measured once, while others are repeated.

Usage

```r
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 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 Gorjanc

See Also

`merge`, `wideByFactor`

Examples

```r
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],
                   ch2=rep(c("a", "b"), n2/2)),
                   ch3=rep(c("c", "d"), n3/2)))
```
```r
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=(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

```r
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

```r
## 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.
centerText

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)
```

---

<table>
<thead>
<tr>
<th>centerText</th>
<th>Center Text Strings</th>
</tr>
</thead>
</table>

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

```r
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")
```

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

```r
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.
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 Logical 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

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 elements, 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**

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

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

**Usage**

```r
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` is a related function that describes objects in an environment.

**Examples**

```r
## 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,...)
lstart(x, n=1,...)
first(x, n=1,...) <- value
lstart(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

## Vector
v <- 1:10
first(v)
lstart(v)
first(v) <- 9
v
lstart(v) <- 20
v
## List
```r
l <- list(a=1, b=2, c=3)
first(l)
last(l)

first(l) <- "apple"
last(l) <- "banana"
l
```

## Data frame
```r
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
```

## Matrix
```r
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
```r
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, na.rm=TRUE),
          SD = sd(x, na.rm=TRUE),
          N = sum(x, na.rm=TRUE)),
  subset = !is.na(Concentration))
}

---

`aggregate.table` is now replaced by `tapply(X=x, INDEX=list(by1, by2), FUN=FUN, ...)`. 

---

getYear

`getYear` is an experimental approach for extracting the date/time parts from objects of a date/time class. They are designed to be intuitive 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 passed to other methods

Value

Character

Author(s)

Gregor Gorjanc

See Also

Date, DateTimeClasses, strftime

Examples

## Date

tmp <- Sys.Date()  
tmp  
getYear(tmp)  
getMonth(tmp)  
getDay(tmp)

## POSIXct

tmp <- as.POSIXct(tmp)  
getYear(tmp)  
getMonth(tmp)  
getDay(tmp)

## POSIXlt

tmp <- as.POSIXlt(tmp)  
getYear(tmp)  
getMonth(tmp)  
getDay(tmp)
humanReadable

Print Byte Size in Human Readable Format

Description

Convert 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 specify 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 $10^3 = 2^{10}$ 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:

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

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"`):

<table>
<thead>
<tr>
<th>Name</th>
<th>System</th>
<th>Symbol</th>
<th>Size</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>binary</td>
<td>B</td>
<td>(2^3)</td>
<td>8 bits</td>
</tr>
<tr>
<td>kibibyte</td>
<td>binary</td>
<td>K</td>
<td>(2^{10})</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>mebibyte</td>
<td>binary</td>
<td>M</td>
<td>(2^{10})^2)</td>
<td>1024 kibibytes</td>
</tr>
<tr>
<td>gibibyte</td>
<td>binary</td>
<td>G</td>
<td>(2^{10})^3)</td>
<td>1024 mebibytes</td>
</tr>
<tr>
<td>tebibyte</td>
<td>binary</td>
<td>T</td>
<td>(2^{10})^4)</td>
<td>1024 gibibytes</td>
</tr>
<tr>
<td>pebibyte</td>
<td>binary</td>
<td>P</td>
<td>(2^{10})^5)</td>
<td>1024 tebibytes</td>
</tr>
<tr>
<td>exbibyte</td>
<td>binary</td>
<td>E</td>
<td>(2^{10})^6)</td>
<td>1024 pebibytes</td>
</tr>
<tr>
<td>zebibyte</td>
<td>binary</td>
<td>Z</td>
<td>(2^{10})^7)</td>
<td>1024 exbibytes</td>
</tr>
<tr>
<td>yottabyte</td>
<td>binary</td>
<td>Y</td>
<td>(2^{10})^8)</td>
<td>1024 yebibytes</td>
</tr>
</tbody>
</table>

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 formatted 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**

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

Interleave Rows of Data Frames or Matrices

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. Starting with 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', etc.

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)

# Create a 2-way table of means, standard errors, and nobs
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)
Description

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

Usage

keep(..., list=character(), 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 be 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 that are deleted if sure=TRUE.

Author(s)

Arni Magnusson

See Also

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

Examples

data(trees, CO2)
keep(trees)
# To remove all objects except trees, run:
# keep(trees, sure=TRUE)
left

Return the leftmost or rightmost columns of a matrix or data frame

Description

Return the leftmost or rightmost or columns of a matrix or data frame

Usage

right(x, n = 6L, ...)  
left(x, n=6L, ...)

## S3 method for class 'matrix'
right(x, n=6L, add.col.nums=TRUE, ...)
## S3 method for class 'matrix'
left(x, n=6L, add.col.nums=TRUE, ...)

## S3 method for class 'data.frame'
right(x, n=6L, add.col.nums=TRUE, ...)
## S3 method for class 'data.frame'
left(x, n=6L, add.col.nums=TRUE, ...)

Arguments

x Matrix or data frame
n If positive, number of columns to return. If negative, number of columns to omit. See examples.
add.col.nums Logical. If no column names are present, add names giving original column number. (See example below.)
... Additional arguments used by methods

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

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

left(m)
right(m)

# When no column names are present, they are added by default
colnames(m) <- NULL
left(m)
colnames(left(m))
right(m)
colnames(right(m))

# Prevent addition of column numbers
left(m, add.col.nums = FALSE)
colnames(left(m, add.col.nums = FALSE))

right(m, add.col.nums = FALSE) # columns are labeled 1:6
colnames(right(m, add.col.nums = FALSE)) # instead of 5:10

# Works for data frames too!
d <- data.frame(m)
left(d)
right(d)

# Use negative n to specify number of columns to omit
left(d, -3)
right(d, -3)
```

---

**Describe Objects or Elements**

**Description**

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

**Usage**

```r
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, S4 object, or any object that is.list.
unit  unit for displaying object size: "B", "KB", "MB", "GB", or first letter (case-insensitive).
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 contributions by Jim Rogers and Michael Chirico.

See Also

- `ls` (objects in an environment), `names` (elements in a list-like object), and `slotNames` (S4 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)`

- `ll(infert)`
- `model <- glm(case~spontaneous+induced, family=binomial, data=infert)`
- `ll(model, dim=TRUE)`
- `ll(model, sort=TRUE)`
- `ll(model$family)`
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 transform back to factor or when working with several factors that should have the same levels and therefore the same internal coding.

### Usage

```r
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`

The `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 below. With `codes=FALSE` **character “levelsMap”** is produced. The later is useful, 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 necessary 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 an 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**

```r
## 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 levelsMap
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 data frame
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 many 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 International Standard (SI) and US 'Conventional' Units for common medical measurements.*

**Description**

Table of conversions between International 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:

\[
\text{Measurement in ConventionalUnit} = \frac{\text{Measurement in SIUnit}}{\text{Conversion}}
\]

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

\[
\text{Measurement in SIUnit} = \text{Measurement in ConventionalUnit} \times \text{Conversion}
\]
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 destination 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.
nobs

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

rm, assign

Examples

```r
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 new default method to handle numeric and logical vectors, and a method for data frames.

Usage

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

Arguments

- `object` Numeric or logical vector, data frame, or a model object.
- `...` Further arguments to be passed to methods.

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` provides a generic `nobs` function with methods for fitted model objects. The `gdata` package adds methods for numeric and logical vectors, as well as data frames.
An alias function `n_obs` is also provided, equivalent to `gdata::nobs`. Using `n_obs` in scripts makes it explicitly clear that the `gdata` implementation is being used.

Author(s)
Gregory R. Warnes <greg@warnes.net>

See Also
- `nobs` in package 'stats' for the base R implementation, `is.na`, `length`

Examples
```r
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)
n_obs(fit)

# Comparison
# gdata
nobs(x)
nobs(df)
# stats
length(na.omit(x))
sapply(df, function(x) length(na.omit(x)))
```

---

### nPairs

**Number of variable pairs**

#### Description
Count the number of pairs between variables.

#### Usage
```r
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 \( \text{margin=TRUE} \), 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

```r
# 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))
```
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        whether or not the result should be printed with surrounding quotes.
humanReadable whether to use the "human readable" format.
standard,units,digits,width,sep,justify
    passed to humanReadable.
recursive     passed to the c method.

Details

The base R package utils provides an object.size function that handles a single object. The gdata package provides a similar object.size function that handles multiple objects.
Both the utils and gdata implementations store the object size in bytes, but offer a slightly different user interface for showing the object size in other formats. The gdata implementation offers human readable format similar to ls, df or du shell commands, by calling humanReadable directly, calling print with the argument humanReadable=TRUE, or by setting options(humanReadable=TRUE). The 3.0.0 release of gdata renamed this function to object_size, allowing users to explicitly call the gdata implementation. This eliminates ambiguity and makes it less likely to get errors when running parts of an existing script without first loading the gdata package. The old object.size function name is now deprecated in the gdata package, pointing users to object_size and utils::gdata instead.

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 base R implementation, 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)

oldopt <- options(humanReadable=TRUE)
(z <- object_size(letters,
  c(letters, letters),
  rep(letters, 100),
  rep(letters, 10000)))
is.object_sizes(z)
as.object_sizes(14567567)
options(oldopt)

# Comparison
# gdata
print(object_size(loadNamespace), humanReadable=TRUE)
print(bigObj, humanReadable=TRUE)
# utils
print(utils::object.size(loadNamespace), units="auto")
sapply(bigObj, utils:::format.object_size, units="auto")
# ll
ll("package:base")[order(-ll("package:base")$KB)[1:10]],]
rename.vars

Remove or rename variables in a data frame

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`: data frame 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

```r
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 the Levels of a Factor

### Description
Reorder the levels of a factor.

### Usage
```r
## 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 character 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
resample

Consistent Random Samples and Permutations

Description

Take 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)
**startsWith**

**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**

A vector of the same length as the input, with the elements permuted.

**Author(s)**

Gregory R. Warnes <greg@warnes.net>

**See Also**

`sample`

**Examples**

```r
## 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))
```

---

**startsWith**

*Does String Start or End With Another String?*

**Description**

Determines if entries of `x` start with a string `prefix`, where strings are recycled to common lengths.

**Usage**

```r
startsWith(x, prefix, trim=FALSE, ignore.case=FALSE)
```
Arguments

x character vector whose “starts” are considered.
prefix character vector, typically of length one, i.e., a string.
trim whether leading and trailing spaces should be removed from x before testing for a match.
ignore.case whether case should be ignored when testing for a match.

Value

A logical vector, of “common length” of x and prefix, i.e., of the longer of the two lengths unless one of them is zero when the result is also of zero length. A shorter input is recycled to the output length.

Note

The base package provides the underlying startsWith function that performs the string comparison. The gdata package adds the trim and ignore.case features.

An alias function starts_with is also provided, equivalent to gdata::startsWith. Using starts_with in scripts makes it explicitly clear that the gdata implementation is being used.

Author(s)

Gregory R. Warnes <greg@warnes.net>

See Also

startsWith for the 'base' package implementation, grepl, substring

Examples

## Simple 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

# Comparison
# gdata
startsWith(s, "Test", trim=TRUE)
startsWith(s, "Test", ignore.case=TRUE)
# base
startsWith(trimws(s), "Test")
startsWith(tolower(s), tolower("Test"))
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

```r
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**

Trim (shorten) 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**

```r
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`
unknownToNA

Examples

```r
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)
```

<table>
<thead>
<tr>
<th>unknownToNA</th>
<th>Change unknown values to NA and vice versa</th>
</tr>
</thead>
</table>

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

```r
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 passed 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 a 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

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"))
```
unmatrix <- 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 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 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, or rows of a data frame

Description

For a list, update the elements of a list to contain all of the named elements of a new list, overwriting elements with the same name, and (optionally) copying unnamed elements. For a data frame, replace the rows of a data frame by corresponding rows in 'new' with the same value for 'by'.

Usage

```r
## S3 method for class 'list'
update(object, new, unnamed=FALSE, ...)
## S3 method for class 'data.frame'
update(object, new, by, by.x=by, by.y=by,
       append=TRUE, verbose=FALSE, ...)
```

Arguments

- `object` List or data frame to be updated.
- `new` List or data frame containing new elements/rows.
- `unnamed` Logical. If TRUE, unnamed list elements of new will be appended to object.
- `by, by.x, by.y` Character. Name of column to use for matching data frame rows.
- `append` Logical. If TRUE, items in new with no match in object will be appended to the data frame.
- `verbose` Logical. If TRUE progress messages will be displayed.
- `...` optional method arguments (ignored).

Value

`update.list` a list a constructed from the elements of object, with named elements of new replacing corresponding named elements from object, and non-corresponding elements of new appended. If unnamed=TRUE, unnamed elements of new will be appended.
update.data.frame

A data frame constructed from the rows of object with rows where values in by.x equal the values in by.y replaced by the corresponding row in new. If append=TRUE, any elements of new without no matching rows in object will be appended.

Note

These methods 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

# Update list
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, unamed=TRUE) # appends unnamed elements of newer

# Update data frame
old <- data.frame(letter=letters[1:5], number=1:5)
new <- data.frame(letter=letters[c(5, 1, 7)], number=c(-5, -1, -7))

update(old, new, by="letter") # default is append=TRUE
update(old, new, by="letter", append=FALSE)
update(old, new, by="letter", verbose=FALSE)
upperTriangle

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

```r
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**

Modify data frame in such a way that variables are “separated” into several columns by factor levels.

**Usage**

```r
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 so that the output represents a data frame with \( c + f + n \times v \) 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.

Examples

```r
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("M", n/2), rep("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"))
```

Usage

```r
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 (NA) in the output.
- **sep**: character, the string to use for separating columns in output.
- **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, allow numeric values to be formatted using scientific notation.
- **...**: further arguments to `format.info` and `format`.

Details

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 characters, 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 \( \text{rowCol} \) is not NULL and \( \text{rownames=True} \), rownames will also have column name with \( \text{rowCol} \) value. This is mainly for flexibility with tools outside R. Note that it may not be easy to import data back to R with \text{read.fwf} if you also export rownames. This is the reason, that default is \( \text{rownames=False} \).

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

If \( \text{quote=True} \), the output is of course wider due to quotes. Return value (with \( \text{formatInfo=True} \)) can account for this in two ways; controlled with argument \text{quoteInfo}. However, note that there is no way to properly read the data back to R if \( \text{quote=True} \) and \( \text{quoteInfo=False} \) arguments were used for export. \text{quoteInfo} applies only when \( \text{quote=True} \). Assume that there is a file with quoted data as shown below (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 \text{quoteInfo=True}
1 12345 1234 # for width with \text{quoteInfo=False}
"a" "hsgdh" " 9"
" " " bb" " 123"
```

With \( \text{quoteInfo=True} \) \text{write.fwf} will return

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

or (with \( \text{quoteInfo=False} \))

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

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

### Value

Besides its effect to write/export data \text{write.fwf} can provide information on format and width. If \( \text{formatInfo=False} \), then 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
```
write.fwf

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

```r
## Some data
num <- round(c(733070.345678, 1214213.78765456, 553823.798765678,
1085022.8876545678, 571063.88765456, 686718.3876545678,
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

## Default
write.fwf(x=testData)
```
```r
## NA should be -
write.fwf(x=testData, na="-")
## NA should be -NA-
write.fwf(x=testData, na="-NA-")
```
```r
## Some other separator than space
write.fwf(x=testData[, 1:4], sepe="-mySep-")
```
```r
## Force wider columns
```
write.fwf(x=testData[, 1:5], width=20)

## Show effect of 'scientific' 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 modification
 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, using quotes
 write.fwf(x=testData, file=file, quote=TRUE)
 read.table(file=file, header=TRUE, strip.white=TRUE)

## Tidy up
 unlink(file)
Index

* NA
  is.what, 25
  unknownToNA, 49
* array
  combine, 10
  interleave, 23
  upperTriangle, 53
* attribute
  ll, 28
  nobs, 37
* category
  interleave, 23
* character
  centerText, 9
  startsWith, 45
  trim, 47
* classes
  is.what, 25
  ll, 28
* data export
  write.fwf, 56
* data output
  write.fwf, 56
* datasets
  MedUnits, 35
* data
  env, 15
  keep, 26
  ll, 28
  mv, 36
  update.list, 52
* documentation
  Args, 5
* environment
  env, 15
  keep, 26
  ll, 28
  ls.funs, 30
  mv, 36
* error
  is.what, 25
* file
  write.fwf, 56
* list
  ll, 28
* logic
  duplicated2, 14
* manip
  bindData, 6
  case, 7
  centerText, 9
  combine, 10
  ConvertMedUnits, 11
  drop.levels, 13
  duplicated2, 14
  first, 16
  frameApply, 17
  getYear, 19
  left, 27
  mapLevels, 31
  matchcols, 33
  rename.vars, 42
  reorder.factor, 43
  trim, 47
  trimSum, 48
  unknownToNA, 49
  unmatrix, 51
  update.list, 52
  wideByFactor, 55
* misc
  bindData, 6
  cbindX, 8
  getYear, 19
  humanReadable, 21
  ls.funs, 30
  mapLevels, 31
  nPairs, 38
  resample, 44
wideByFactor, 55

* **missing**
  unknownToNA, 49

* **package**
  gdata-package, 3

* **pairs**
  nPairs, 38

* **print**
  ll, 28
  write.fwf, 56

* **programming**
  ans, 4
  Args, 5
  is.what, 25

* **utilities**
  Args, 5
  env, 3, 15
  keep, 26
  ll, 28
  object_size, 40
  .Last.value, 4
  .checkLevelsMap (mapLevels), 31
  .checkListLevelsMap (mapLevels), 31

abbreviate, 39
aggregate.table (gdata-defunct), 19
ans, 3, 4
Args, 3, 5
args, 5
as.levelsMap (mapLevels), 31
as.listLevelsMap (mapLevels), 31
as.object_sizes (object_size), 40
as.vector, 51
assign, 37
bindData, 6
by, 18
c, 40
c.levelsMap (mapLevels), 31
c.listLevelsMap (mapLevels), 31
c.object_sizes (object_size), 40
case, 3, 7
cbind, 8, 9, 24
cbindX, 3, 8
centerText, 3, 9
colnames, 42
combine, 3, 10, 24

ConvertMedUnits, 3, 11, 36
data.frame, 42
Date, 20
DateClasses, 20
diag, 54
drop.levels, 13
duplicated, 14
duplicated2, 3, 14
env, 3, 15, 29
eval, 4
factor, 32, 44
first, 16, 27
first<- (first), 16
formals, 5
format, 57–59
format.info, 57–59
format.object_sizes (object_size), 40

frameApply, 3, 17
gdata (gdata-package), 3
gdata-defunct, 19
gdata-package, 3
getDateTimeParts (getYear), 19
gerDay, 3
getDay (getYear), 19
getHour, 3
getHour (getYear), 19
getMin, 3
getMin (getYear), 19
getMonth, 3
getMonth (getYear), 19
getSec, 3
gSec (getYear), 19
getYear, 3, 19
grep, 34
grepl, 46
gsub, 47

head, 16, 27
help, 5
humanReadable, 3, 21, 40, 41
interleave, 3, 23
is.function, 30
is.levelsMap (mapLevels), 31
is.listLevelsMap (mapLevels), 31
is.na, 25, 38, 50
is.numeric, 25
is.object_sizes (object_size), 40
is.what, 3, 25
isUnknown (unknownToNA), 49

keep, 3, 26

last, 27
last (first), 16
last<- (first), 16
left, 16, 27
length, 38
levels, 3, 32, 47
ll, 3, 15, 23, 28
lower.tri, 54
lowerTriangle, 3
lowerTriangle (upperTriangle), 53
lowerTriangle<- (upperTriangle), 53
ls, 29, 30
ls.funs, 3, 30

mapLevels, 3, 31
mapLevels<- (mapLevels), 31
matchcols, 3, 33
MedUnits, 3, 12, 35
merge, 6, 11, 53
mixedsort, 43
mv, 36

n_obs (nobs), 37
names, 29, 42
NAToUnknown, 57, 59
NAToUnknown (unknownToNA), 49
nchar, 57
nobs, 3, 37, 38
nPairs, 3, 38

object.size, 23, 41
object.size (object_size), 40
object.size, 3, 40
ordered, 43

print.levelsMap (mapLevels), 31
print.listLevelsMap (mapLevels), 31
print.object_sizes (object_size), 40

rbind, 8, 9, 11, 24
read.fwf, 58, 59
read.table, 47, 57, 59
remove.vars (rename.vars), 42
rename.vars, 3, 42
reorder, 44
reorder.factor, 3, 13, 43, 47
resample, 3, 44
reshape, 56
right, 16
right (left), 27
rm, 26, 37

sample, 45
search, 15
slotNames, 29
sort.levelsMap (mapLevels), 31
starts_with (startsWith), 45
startsWith, 3, 45, 46
str, 29
strptime, 20
strwrap, 10
sub, 47
substring, 46
summary, 29
tail, 16, 27
trim, 3, 47, 48, 59
trimSum, 3, 48
trimws, 47
try, 18

unclass, 32
unique, 14
unique.levelsMap (mapLevels), 31
unknownToNA, 3, 49
unmatrix, 3, 51
update, 53
update.data.frame (update.list), 52
update.list, 52
upper.tri, 54
upperTriangle, 3, 53
upperTriangle<- (upperTriangle), 53

warning, 49
wideByFactor, 3, 6, 55
write.fwf, 3, 56
write.table, 57, 59