

# Package ‘gdata’

January 2, 2012

**Title** Various R programming tools for data manipulation

**Description** Various R programming tools for data manipulation

**Depends** R (>= 2.6.0)

**Imports** gtools

**Version** 2.8.2

**Date** 2011-04-15

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

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

**License** GPL-2

**Repository** CRAN

**Date/Publication** 2011-04-24 06:08:17

## R topics documented:

gdata-package . . . . .	2
.runRUnitTestsGdata . . . . .	3
aggregate.table . . . . .	4
Args . . . . .	5
bindData . . . . .	6
cbindX . . . . .	7
combine . . . . .	8
ConvertMedUnits . . . . .	9
drop.levels . . . . .	11
elem . . . . .	12
env . . . . .	13
frameApply . . . . .	14
getYear . . . . .	16

humanReadable . . . . .	17
installXLSXsupport . . . . .	20
interleave . . . . .	21
is.what . . . . .	23
keep . . . . .	24
ll . . . . .	25
mapLevels . . . . .	26
matchcols . . . . .	29
MedUnits . . . . .	30
nobs . . . . .	32
nPairs . . . . .	33
object.size . . . . .	34
read.xls . . . . .	36
rename.vars . . . . .	38
reorder.factor . . . . .	39
resample . . . . .	41
sheetCount . . . . .	42
trim . . . . .	43
trimSum . . . . .	44
unknownToNA . . . . .	45
unmatrix . . . . .	47
upperTriangle . . . . .	48
wideByFactor . . . . .	50
write.fwf . . . . .	51
xlsFormats . . . . .	55

<b>Index</b>	<b>56</b>
--------------	-----------

---

gdata-package	<i>Various R programming tools for data manipulation</i>
---------------	--

---

## Description

**gdata** package provides various R programming tools for data manipulation.

## Details

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

DESCRIPTION file	<code>library(help="gdata")</code>
This file	<code>package?gdata</code>
Vignette	<code>vignette("unknown")</code> <code>vignette("mapLevels")</code>
Some help files	<a href="#">read.xls</a> <a href="#">write.fwf</a>

```
News          file.show(system.file("NEWS", package="gdata"))
```

## Testing

If you want to perform the validity/unit testing of the installed **ggmisc** package on your own computer, take a look at [.runRUnitTestsGdata](#) function - please note that you need the **RUnit** package for this to work.

---

`.runRUnitTestsGdata` *Run RUnit tests for the gdata package*

---

## Description

Run **RUnit** tests to perform the validity/unit testing of installed **gdata** package on your own computer.

## Usage

```
.runRUnitTestsGdata(testFileRegexp="^runit.+\\. [rR]$")
```

## Arguments

`testFileRegexp` regular expression; see details

## Details

Argument `testFileRegexp` can be used to specify different sets of tests provided by the package. The following values are sensible:

- `"^runit.+\\. [rR]$"` for basic tests

## Value

None, just the print out of **RUnit** testing.

## See Also

[defineTestSuite](#) in **RUnit** package

## Examples

```
## Basic testing
.runRUnitTestsGdata()
```

---

aggregate.table      *Create 2-Way Table of Summary Statistics*

---

**Description**

Splits the data into subsets based on two factors, computes a summary statistic on each subset, and arranges the results in a 2-way table.

**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.

**Value**

Returns a matrix with one element for each combination of by1 and by2.

**Author(s)**

Gregory R. Warnes <greg@warnes.net>

**See Also**

[aggregate](#), [tapply](#), [interleave](#)

**Examples**

```
# 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 <- aggregate.table( dat, g1, g2, mean )
stderrs <- aggregate.table( dat, g1, g2, stderr )
ns <- aggregate.table( dat, 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)
```

---

**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 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

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

### Arguments

<code>x</code>	data.frame
<code>y</code>	data.frame
<code>common</code>	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), ]
```

---

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)
```

---

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 combined vector data is labeled `data`.

When the arguments consist of a mix of matrixes 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 omitted, 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	<i>Convert medical measurements between International Standard (SI) and US 'Conventional' Units.</i>
-----------------	--

---

## 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**

[http://www.globalrph.com/conv\\_si.htm](http://www.globalrph.com/conv_si.htm)

**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 <a href="#">reorder.factor?</a>
...	additional arguments to <a href="#">reorder.factor</a>

## 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(1)
str(df)
str(drop.levels(df))
```

---

elem

*Display Information about Elements in a Given Object*


---

### Description

*This function is deprecated. Please use [l1](#) 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 ( <i>see notes</i> ).
Dim	element dimensions ( <i>optional</i> ).

### 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 ( <i>see notes</i> ).

**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](#).

[l1](#) is a related function that describes objects in an environment.

**Examples**

```
## Not run:
env()

## End(Not run)
```

---

 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 <a href="#">try</a> ), 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 `i`th component gives the result for the `i`th 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))
```

---

`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 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

<code>x</code>	generic, date/time object
<code>format</code>	character, format
<code>...</code>	arguments passed 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 byte size in human readable format such as kB, MB, GB, etc.

## Usage

```
humanReadable(x, standard="SI", digits=1, width=3, sep=" ")
```

## Arguments

x	integer, byte size
standard	character, "SI" for powers of 1000 or anything else for powers of 1024, see details
digits	integer, number of digits after decimal point
width	integer, width of number string
sep	character, separator between number and unit

## Details

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". Otherwise IEC standard (powers of 1024) is used.

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 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 and Gregor Gorjanc

### References

Wikipedia: <http://en.wikipedia.org/wiki/Byte> [http://en.wikipedia.org/wiki/SI\\_prefix](http://en.wikipedia.org/wiki/SI_prefix)  
[http://en.wikipedia.org/wiki/Binary\\_prefix](http://en.wikipedia.org/wiki/Binary_prefix)

GNU manual for coreutils: [http://www.gnu.org/software/coreutils/manual/html\\_node/Block-size.html#Block-size](http://www.gnu.org/software/coreutils/manual/html_node/Block-size.html#Block-size)

### See Also

[object.size, ll](#)

**Examples**

```

baseSI <- 10
powerSI <- seq(from=3, to=27, by=3)
SI0 <- (baseSI)^powerSI
k <- length(SI0) - 1
SI1 <- SI0 - SI0 / c(2, runif(n=k, min=1.01, max=5.99))
SI2 <- SI0 + SI0 / c(2, runif(n=k, min=1.01, max=5.99))

baseIEC <- 2
powerIEC <- seq(from=10, to=90, by=10)
IEC0 <- (baseIEC)^powerIEC
IEC1 <- IEC0 - IEC0 / c(2, runif(n=k, min=1.01, max=5.99))
IEC2 <- IEC0 + IEC0 / c(2, runif(n=k, min=1.01, max=5.99))

cbind(humanReadable(x=SI1, width=NULL, digits=3),
      humanReadable(x=SI0, width=NULL, digits=2),
      humanReadable(x=SI2, width=NULL, digits=1),
      humanReadable(x=IEC1, standard="IEC", width=7, digits=3),
      humanReadable(x=IEC0, standard="IEC", width=7, digits=2),
      humanReadable(x=IEC2, standard="IEC", width=7, digits=1))

```

---

installXLSXsupport	<i>Install perl modules needed for read.xls to support Excel 2007+ XLSX format</i>
--------------------	--

---

**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 occurred.

### See Also

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

### Examples

```
installXLSXsupport()
```

---

interleave	<i>Interleave Rows of Data Frames or Matrices</i>
------------	---

---

### Description

Interleave rows of data frames or Matrices.

### Usage

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

### Arguments

<code>...</code>	objects to be interleaved
<code>append.source</code>	Boolean Flag. When TRUE (the default) the argument name will be appended to the row names to show the source of each row.
<code>sep</code>	Separator between the original row name and the object name.
<code>drop</code>	logical - If the number of columns in output matrix is 1, whether matrix should be returned or a vector

**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 <- aggregate.table( dat, g1, g2, mean )
stderrs <- aggregate.table( dat, g1, g2, stderr )
ns <- aggregate.table( dat, 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

<code>object</code>	any R object.
<code>verbose</code>	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 <code>.</code> ) 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)
```

---

**l1** *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**

```
l1(pos=1, unit="KB", digits=0, dim=FALSE, sort=FALSE, class=NULL,
   invert=FALSE, ...)
```

**Arguments**

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

**Value**

A data frame with named rows and the following columns:

<code>Class</code>	object class.
<code>KB</code>	object size ( <i>see note</i> ).
<code>Dim</code>	object dimensions ( <i>optional</i> ).

**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**

`l1` 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)

```

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**

```

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 below. 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 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 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 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	<i>Select columns names matching certain criteria</i>
-----------	---

---

**Description**

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

**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

**SIUnit** SI Unit

**Conversion** Conversion factor

**ConventionalUnit** Conventional 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

[http://www.globalrph.com/conv\\_si.htm](http://www.globalrph.com/conv_si.htm)

## 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" )
```

---

`nobs`*Compute the Number of Non-missing Observations*

---

**Description**

Compute the number of non-missing observations. Special methods exist for data frames, and lm objects.

**Usage**

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

**Arguments**

<code>x</code>	Target Object
<code>...</code>	Optional parameters (currently ignored)

**Details**

In the simplest case, this is really just wrapper code for `sum(!is.na(x))`.

**Value**

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

**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)
```

---

nPairs	<i>Number of variable pairs</i>
--------	---------------------------------

---

### 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 <a href="#">abbreviate</a>

### 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 an Object*


---

**Description**

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

**Usage**

```
object.size(...)

## S3 method for class 'object_size'
print(x, quote=FALSE, humanReadable, ...)
```

**Arguments**

...	object.size: R objects; print; arguments to be passed to or from 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.

## Details

This is a modified copy from the `utils` package in R as of 2008-12-15.

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 platforms than 32-bit ones, but will very likely be the same on different platforms with the same word length and pointer size.

Class of returned object is `c("byte", "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 invoked with `options(humanReadable=TRUE)`.

## Value

An object of class `"object.size"` with a length-one double value, an estimate of the memory allocation attributable to the object in bytes.

## See Also

[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
z <- sapply(ls("package:base"), function(x)
  object.size(get(x, envir = baseenv())))
(tmp <- as.matrix(rev(sort(z))[1:10]))

as.object_size(14567567)
options(humanReadable=TRUE)
(z <- object.size(letters, c(letters, letters), rep(letters, 100), rep(letters, 10000)))
is.object_size(z)
as.object_size(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, ...,
         method=c("csv","tsv","tab"), perl="perl")
xls2csv(xls, sheet=1, verbose=FALSE, ..., perl="perl")
xls2tab(xls, sheet=1, verbose=FALSE, ..., perl="perl")
xls2tsv(xls, sheet=1, verbose=FALSE, ..., perl="perl")
xls2sep(xls, sheet=1, verbose=FALSE, ..., method=c("csv","tsv","tab"),
        perl="perl")
```

**Arguments**

xls	path to the Microsoft Excel file. Supports "http://", "https://", and "ftp://" URLs.
sheet	number of the sheet within the Excel file from which data are to be read
verbose	logical flag indicating whether details should be printed as the file is processed.
pattern	if specified, then 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
...	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")

# 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)

# read xls file from net
nba.url <- "http://mgtclass.mgt.unm.edu/Bose/Excel/Tutorial.05/Cases/NBA.xls"
nba <- read.xls(nba.url)
```

```

# 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..
  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)

```

---

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 <a href="#">ordered</a> 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 provided: For a numeric vector, the new factor level names are constructed by reordering the factor levels according to the numeric values. For vectors, 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 applying the supplied function to the existing factor level names. With sort=mixedsort 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=as.numeric(trt), FUN=length)
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

<code>x</code>	A numeric, complex, character or logical vector from which to choose.
<code>size</code>	Non-negative integer giving the number of items to choose.
<code>replace</code>	Should sampling be with replacement?
<code>prob</code>	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)

```

---

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 <a href="#">reorder.factor</a> 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**

[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	<i>Trim a vector such that the last/first value represents the sum of trimmed values</i>
---------	--

---

**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**

<code>x</code>	numeric, a vector of numeric values
<code>n</code>	numeric, desired length of the output
<code>right</code>	logical, trim on the right/bottom or the left/top side
<code>na.rm</code>	logical, remove NA values when applying a function
<code>...</code>	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  $\mathbb{R}$ ) 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 passed to other methods (as.character for POSIXlt in case of isUnknown)
call.	logical, look in <a href="#">warning</a>

## 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](#)

**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)
```

---

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

**Arguments**

x	Matrix
diag	Logical. If TRUE, include the matrix diagonal.
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).

**Author(s)**

Gregory R. Warnes <greg@warnes.net>

**See Also**

[diag](#)

**Examples**

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

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

upperTriangle(x) <- NA
x

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

lowerTriangle(x) <- NA
x

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

<code>x</code>	data frame
<code>factor</code>	character, column name of a factor by which variables will be divided
<code>common</code>	character, column names of (common) columns that should not be divided
<code>sort</code>	logical, sort resulting data frame by factor levels
<code>keepFactor</code>	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))

write.fwf(x=tmp, factor="f1", common=c("c1", "c2", "f2"))
write.fwf(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"), ...)

```

## Arguments

x	data.frame or matrix, the object to be written
file	character, name of file or connection, look in <a href="#">write.table</a> 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 <a href="#">format</a>
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.
...	further arguments to <a href="#">format.info</a> and <a href="#">format</a>

## Details

\*F\*ixed \*w\*idth \*f\*ormat is not used widely anymore. Use some other format (say \*c\*omma \*s\*eparated \*v\*alues; see [read.csv](#)) if you can. However, if you need fixed width format then `write.fwf` can help you.

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 too 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:

<code>colname</code>	name of the column
<code>nlevels</code>	number of unique values (unused levels of factors are dropped), 0 for numeric column
<code>position</code>	starting column number in the output
<code>width</code>	width of the column
<code>digits</code>	number of digits after the decimal point
<code>exp</code>	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 $2 \cdot 1e+06$ 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)

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

## 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 - 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**

<code>perl</code>	Path to perl interpreter (optional).
<code>verbose</code>	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

- \*Topic **NA**
  - is.what, 23
  - unknownToNA, 45
- \*Topic **array**
  - combine, 8
  - interleave, 21
  - upperTriangle, 48
- \*Topic **attribute**
  - elem, 12
  - ll, 25
  - nobs, 32
- \*Topic **category**
  - aggregate.table, 4
  - interleave, 21
- \*Topic **character**
  - trim, 43
- \*Topic **classes**
  - elem, 12
  - is.what, 23
  - ll, 25
- \*Topic **datasets**
  - MedUnits, 30
- \*Topic **data**
  - env, 13
  - keep, 24
  - ll, 25
- \*Topic **documentation**
  - Args, 5
- \*Topic **environment**
  - env, 13
  - keep, 24
  - ll, 25
- \*Topic **error**
  - is.what, 23
- \*Topic **file**
  - read.xls, 36
  - write.fwf, 51
- \*Topic **iteration**
  - aggregate.table, 4
- \*Topic **list**
  - elem, 12
  - ll, 25
- \*Topic **manip**
  - bindData, 6
  - combine, 8
  - ConvertMedUnits, 9
  - drop.levels, 11
  - frameApply, 14
  - getYear, 16
  - mapLevels, 26
  - matchcols, 29
  - rename.vars, 38
  - reorder.factor, 39
  - trim, 43
  - trimSum, 44
  - unknownToNA, 45
  - unmatrix, 47
  - wideByFactor, 50
- \*Topic **misc**
  - .runRUnitTestsGdata, 3
  - bindData, 6
  - cbindX, 7
  - getYear, 16
  - humanReadable, 17
  - installXLSXsupport, 20
  - mapLevels, 26
  - nPairs, 33
  - resample, 41
  - sheetCount, 42
  - wideByFactor, 50
  - xlsFormats, 55
- \*Topic **package**
  - gdata-package, 2
- \*Topic **print**
  - elem, 12
  - ll, 25
  - write.fwf, 51
- \*Topic **programming**

- Args, 5
- is.what, 23
- \*Topic **utilities**
  - Args, 5
  - elem, 12
  - env, 13
  - is.what, 23
  - keep, 24
  - ll, 25
  - object.size, 34
- .checkLevelsMap (mapLevels), 26
- .checkListLevelsMap (mapLevels), 26
- .runRUnitTestsGdata, 3, 3
  
- abbreviate, 33
- aggregate, 4
- aggregate.table, 4
- Args, 5
- args, 5
- as.levelsMap (mapLevels), 26
- as.listLevelsMap (mapLevels), 26
- as.object\_size (object.size), 34
- as.vector, 48
  
- bindData, 6
- by, 15
  
- c.levelsMap (mapLevels), 26
- c.listLevelsMap (mapLevels), 26
- c.object\_size (object.size), 34
- cast, 50
- cbind, 7, 8, 22
- cbindX, 7
- colnames, 39
- combine, 8, 22
- ConvertMedUnits, 9, 31
  
- data.frame, 39
- Date, 16
- DateTimeClasses, 16
- defineTestSuite, 3
- diag, 49
- drop.levels, 11
  
- elem, 12
- env, 13, 13, 25
  
- factor, 28, 40
- formals, 5
- format, 51–53
- format.info, 51–53
- frameApply, 14
  
- gdata (gdata-package), 2
- gdata-package, 2
- getDateTimeParts (getYear), 16
- getDay (getYear), 16
- getHour (getYear), 16
- getMin (getYear), 16
- getMonth (getYear), 16
- getSec (getYear), 16
- getYear, 16
- grep, 29
- gsub, 44
  
- help, 5
- humanReadable, 17, 35
  
- installXLSXsupport, 20
- interleave, 4, 21
- is.levelsMap (mapLevels), 26
- is.listLevelsMap (mapLevels), 26
- is.na, 23, 32, 47
- is.numeric, 23
- is.object\_size (object.size), 34
- is.what, 23
- isUnknown (unknownToNA), 45
  
- keep, 24
  
- length, 32
- levels, 27, 28, 43
- levels<-, 27
- ll, 12–14, 19, 25
- lowerTriangle (upperTriangle), 48
- lowerTriangle<- (upperTriangle), 48
- ls, 25
  
- mapLevels, 26
- mapLevels<- (mapLevels), 26
- matchcols, 29
- MedUnits, 10, 30
- melt, 50
- Memory-limits, 35
- merge, 6, 9
- mixedsort, 40
  
- names, 13, 25, 39
- NAToUnknown, 52, 53
- NAToUnknown (unknownToNA), 45

nchar, 52  
nobs, 32  
nPairs, 33  
  
object.size, 19, 34  
ordered, 40  
  
print.levelsMap (mapLevels), 26  
print.listLevelsMap (mapLevels), 26  
print.object\_size (object.size), 34  
  
rbind, 7–9, 22  
read.csv, 37, 52  
read.fwf, 52, 53  
read.table, 44, 52, 53  
read.xls, 2, 21, 36, 42, 55  
remove.vars (rename.vars), 38  
rename.vars, 38  
reorder, 40  
reorder.factor, 11, 39, 43, 44  
resample, 41  
reshape, 50  
rm, 24  
  
sample, 42  
search, 14  
sheetCount, 42  
sheetNames (sheetCount), 42  
sort.levelsMap (mapLevels), 26  
str, 13, 25  
strptime, 16  
sub, 44  
summary, 13, 25  
  
tapply, 4  
trim, 43, 45, 53  
trimSum, 44  
try, 14  
  
unclass, 28  
unique.levelsMap (mapLevels), 26  
unknownToNA, 45  
unmatrix, 47  
upperTriangle, 48  
upperTriangle<- (upperTriangle), 48  
  
warning, 46  
wideByFactor, 6, 50  
write.fwf, 2, 51  
write.table, 51–53  
  
xls2csv, 21, 42, 55  
xls2csv (read.xls), 36  
xls2sep (read.xls), 36  
xls2tab (read.xls), 36  
xls2tsv (read.xls), 36  
xlsFormats, 21, 55