Package 'Kmisc'

July 2, 2014

Type Package

Title Kevin Miscellaneous

Version 0.5.0

Date 2013-12-12

Author Kevin Ushey

Maintainer Kevin Ushey <kevinushey@gmail.com>

Description This package contains a collection of functions for common data extraction and reshaping operations, string manipulation, and functions for table and plot generation for R Markdown documents.

License GPL (>= 2)

LazyData TRUE

LinkingTo Rcpp

URL https://github.com/kevinushey/Kmisc

BugReports https://github.com/kevinushey/Kmisc/issues

Imports Rcpp (>= 0.10.5), data.table, lattice, grid, knitr, markdown

Suggests ggplot2, plyr, reshape2, googleVis, testthat, microbenchmark

ByteCompile TRUE

VignetteBuilder knitr

Collate 'Kmisc-md2html.R' 'Kmisc-package.R' 'RcppExports.R' 'Rcpp_gen_makevars.R' 'anat.R' 'any_na.R' 'apply.R' 'awk.R' 'bwplot2.R' 'chunk.R' 'clean_doc.R' 'copy.R' 'counts.R' 'extract.R' 'extract_rows_from_file.R' 'fast_factor.R' 'grid.text2.R' 'html.R' 'htmlTable.R' 'html_attach.R' 'html_extras.R' 'html_tags.R' 'in_interval.R' 'is_sorted.R' 'labeller.R' 'list_to_dataframe.R' 'manhattan_plot.R' 'mat2df.R' 'matches.R' 'melt_.R' 'misc.R' 'nametree.R' 'pMerge.R' 'pad.R' 'par.reset.R' 'pp_plot.R'

'prepare_package.R' 'pymat.R' 'rcpp_apply_generator.R'
'rcpp_tapply_generator.R' 'read.R' 'regex.R'
'registerFunctions.R' 'remove_char_digit.R' 'simp.R' 'size.R'
'split_file.R' 'split_runs.R' 'stack_list.R' 'str_collapse.R'
'str_rev.R' 'str_slice.R' 'str_sort.R' 'str_split.R' 'swap.R'
'sys.R' 'tapply_.R' 'transpose.R' 'unmelt.R' 'update_date.R' 'value_matching.R' 'wrap.R'

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-12-13 06:40:46

R topics documented:

`anat` *Display the Anatomy of a Data Frame*

Description

This function displays the 'anatomy' of a data frame. In practice, it's used to implement a faster version of `str` for data frames built entirely of atomic vectors, as `str.data.frame` is very slow for large data frames. If there are non-atomic vectors in `df`, we fall back to `base::str`.

Usage

```
anat(df, n = 3, cols = 99)

anatomy(df, n = 3, cols = 99)
```

Arguments

`df`	An object inheriting class `data.frame`.
`n`	The number of elements to print from each vector.
`cols`	The number of columns to print from the `data.frame`.

Examples

```
## Not run:
local({
  bigDF <- as.data.frame( matrix( factor(1:1E3), nrow=1E3, ncol=1E3 ) )
  sink( tmp <- tempfile() )
  str <- system.time( str(bigDF, list.len=1E3) )
  anat <- system.time( anat(bigDF) )
```

```
  sink()
  unlink(tmp)
  print( rbind( str, anat ) )
})

## End(Not run)
```

---

any_na *Check whether there are any Missing Values in a Vector*

Description

This function checks whether there are any missing values in an R object. For list objects, we recurse over each entry. This is typically faster than `any(is.na(x))` as we exit and return `TRUE` as soon as an `NA` is discovered.

Usage

```
any_na(x, how = "unlist")
```

Arguments

x	An R object.
how	The simplification to use if x is a list. See `rapply` for more details.

---

attachHTML *Attach Common, Non-Masking HTML Functions*

Description

DEPRECATED: Please use the `html` function to generate HTML.

Usage

```
attachHTML()
```

awk *A Simple Front-end to Awk*

Description

This function provides a simple front-end to awk. It assumes that you have awk available and in your PATH.

Usage

```
awk(code, file, BEGIN = NULL, END = NULL, vars = NULL, fs = NULL,
  out = TRUE, verbose = FALSE)
```

Arguments

code	The awk code you want to put in the main execution block.
file	The file we are running awk on.
BEGIN	A block of code to include as though it were within the BEGIN block.
END	A block of code to include as though it were within the END block.
vars	A named list, whereby variables are assigned so that name=value.
fs	The field separator (passed to -F).
out	The location to output the result of the computation. If this is TRUE, we intern the process and bring the results back into the R session. Otherwise, it should be a string specifying the output path for a file.
verbose	Output the generated awk code?

Examples

```
## Not run:
dat <- data.frame(
  x=1:10,
  y=letters[1:10],
  z=LETTERS[1:10]
)

tempfile <- tempfile()

write.table(dat,
  file=tempfile,
  row.names=FALSE,
  col.names=FALSE,
  quote=FALSE
)

x <- awk("print $1", tempfile)
## note that it is read in as type 'character'
```

```
print( cbind( x, dat$x ) )

## End(Not run)
```

---

awk.set *Set awk*

Description

Use this function to set the string by which awk is called; e.g. if you're using GNU awk (gawk), mawk, and so on.

Usage

```
awk.set(awk)
```

Arguments

awk	String denoting the appropriate call for your flavour of awk.

---

bwplot2 *Custom Lattice Boxplot*

Description

This generates a custom lattice boxplot; we super-impose actual plot points for groups with a small number of points, and also restrict plotting of the boxplot for these as well (since they are really rather mis-representative of the distribution when there are so few points.) The downside is that functionality is not implemented for multi-panel plots.

Usage

```
bwplot2(form, data = NULL, xlab = NULL, ylab = NULL, main = NULL,
  n = 10, ...)
```

Arguments

form	a formula object as expected by lattice's xyplot.
data	see xyplot.
xlab	see xyplot.
ylab	see xyplot.
main	see xyplot.
n	number of points necessary for a boxplot to be drawn.
...	additional arguments passed to xyplot call.

Details

Axis labels are inferred from the `form` object passed in when possible.

Examples

```
set.seed(123)
dat <- data.frame( y=rnorm(100), x=factor( rbinom(100,size=2,p=0.8) ) )
bwplot2( x ~ y , dat)
```

---

cat.cb *Write Data to the Clipboard*

Description

This function writes data to the clipboard, using `cat`.

Usage

```
cat.cb(dat, ...)
```

Arguments

`dat`	data to be written to the clipboard.
`...`	optional arguments passed to `cat`.

See Also

`write.cb`, `cat`

---

cd *Set Working Directory*

Description

A small convenience function that wraps `file.path` into a `setwd` call.

Usage

```
cd(...)
```

Arguments

`...`	the set of strings to paste together. if no arguments are submitted, then we return to the home directory.

Examples

```
x <- "my_favourite_dir"
#setwd( "C:/", x, "really_awesome_stuff" )
## calls setwd( paste( "C:/", x, "really_awesome_stuff", collapse="" ) )
```

---

char_to_factor *Converts Characters to Factors in an Object*

Description

Converts characters to factors in an object. Leaves non-factor elements untouched.

Usage

```
char_to_factor(X, inplace = FALSE, ...)
```

Arguments

X	an object.
inplace	boolean; if TRUE the object is modified in place. Useful if you're modifying a list and don't want to force a copy, but be wary of other symbols pointing as the same data.
...	Ignored.

---

chunk *Generate a Sequence of Integers, and Split into Chunks*

Description

This function takes a set of endpoints, and 'chunks' the sequence from min to max into a list with each element of size size.

Usage

```
chunk(min, max, size, by = 1)
```

Arguments

min	The lower end (start point) of the sequence.
max	The upper end (end point) of the sequence.
size	The number of elements to place in each chunk.
by	The difference between consecutive elements.

Details

If max is not specified, then we generate a chunk of integers from 1 to min, each of size size. This allows you to specify chunks with syntax like chunk(100, size=5).

clean_doc *Clean Documentation in Current Package*

Description

This function removes all the .Rd documentation files present in `<dir>/man`. This function is handy if you've 'polluted' your `man` directory in prototyping different functions – assuming that you're documenting your code with eg. `roxygen`.

Usage

```
clean_doc(dir = getwd(), ask = TRUE)
```

Arguments

`dir`	the project directory.
`ask`	boolean. ask before clearing directory?

counts *Generate Counts of Values in a Vector*

Description

This function uses Rcpp sugar to implement a fast `table`, for unique counts of a single vector.

Usage

```
counts(x)
```

Arguments

`x`	A numeric, integer, character or logical vector, or a (potentially nested) list of such vectors. If `x` is a list, we recursively apply counts throughout elements in the list.

dapply *Apply a Function over a List*

Description

A convenience function that works like lapply, but coerces the output to a data.frame if possible. We set stringsAsFactors=FALSE, and optional=TRUE, to minimize the amount of automatic coersion R might try to do.

Usage

```
dapply(X, FUN, ...)
```

Arguments

X	a vector, expression object, or a data.frame
FUN	function to be applied to each element of X.
...	optional arguments to FUN.

Details

This function is preferable to sapply or lapply when you explicitly want a data frame returned.

See Also

lapply, lapply

Examples

```
dat <- data.frame(
  x = rnorm(10),
  y = rnorm(10)
)

## Calculate 0.025, 0.975 quantiles for each column in a data.frame,
## and return result as data.frame .
dapply( dat, function(x) {
  quantile(x, c(0.025, 0.975))
} )

dapply( dat, summary )
str( dapply( dat, summary ) )
```

detachHTML *Detach Common, Non-Masking HTML Functions*

Description

DEPRECATED: Please use the `html` function to generate HTML.

Usage

```
detachHTML()
```

duplicate *Force a Copy of an R Object*

Description

In R, objects are copied 'lazily'. We use this function to force a copy.

Usage

```
duplicate(x)
```

Arguments

x	An R object.

extract *Extract Elements from a Named Object*

Description

Extracts elements from an R object with the names attribute set in a 'lazy' way. The first argument is the object, while the second is a set of names parsed from .... We return the object, including only the elements with names matched from ....

Usage

```
extract(x, ...)
```

Arguments

x	An R object with a `names` attribute.
...	an optional number of 'names' to match in `dat`.

Details

We can be 'lazy' with how we pass names. The names passed to ... are not evaluated directly; rather, their character representation is taken and used for extraction.

See Also

without, extract.re

Examples

```
dat <- data.frame( x = c(1, 2, 3), y = c("a", "b", "c"), z=c(4, 5, 6) )
## all of these return identical output
dat[ names(dat) %in% c("x","z") ]
extract( dat, x, z)
```

---

extract_rows_from_file

Extract Rows from File

Description

This function reads through a delimited file on disk, determines if the entry at the specified column is in a character vector of items, and writes that line to file (or to R) if it is.

Usage

```
extract_rows_from_file(file, out = NULL, column, sep = NULL, keep)
```

Arguments

file	The input file to extract rows from.
out	The location to output the file. If this is NULL, we redirect output back into the R session.
column	The column to check, indexed from 1.
sep	The delimiter used in file. Must be a single character. If no delimiter is specified, we guess the delimiter from the first row of file.
keep	A character vector containing all items that we want to check and keep within the columnth column of each row.

See Also

split_file

factor_ *Fast Factor Generation*

Description

This function generates factors quickly using faster sorting and matching algorithms available in Rcpp.

Usage

```
factor_(x, levels = NULL)
```

Arguments

x	An object of atomic type integer, numeric, character or logical.
levels	An optional character vector of levels. Is coerced to the same type as x. By default, we compute the levels as sort(unique(x)).

factor_to_char *Converts Factors to Characters in an Object*

Description

Converts factors to characters in an object. Leaves non-character elements untouched.

Usage

```
factor_to_char(X, inplace = FALSE)
```

Arguments

X	an object.
inplace	Boolean; if TRUE we modify the object in place. Useful if you're modifying a list and don't want to force a copy, but be wary of other symbols pointing as the same data.

Details

We iterate through all elements in the object (e.g. if it is a list) and convert anything that is a factor into a character.

getObjects *Get all Objects in Environment*

Description

Get all objects within an environment. Useful for inspecting the objects available in a particular environment.

Usage

```
getObjects(env)
```

Arguments

env	an environment.

Value

a list of the objects contained within that environment.

Examples

```
myenv <- new.env()
assign( "foo", "bar", env=myenv )
assign( "baz", "spam", env=myenv )
getObjects( myenv )
```

gradient *Generate Gradient from Continuous Variable*

Description

Assign colors based on a continuous variable. Useful for plotting functions where you would like to generate a gradient based on (a function of) the continuous variables you are plotting quickly.

Usage

```
gradient(x, m = 10, cols = c("darkorange", "grey60", "darkblue"))
```

Arguments

x	a continuous variable to generate colors over.
m	the number of distinct colors you wish to pull from the pallete.
cols	the colors to interpolate over. passed to colorRampPalette.

See Also

colorRampPalette

Examples

```
dat <- data.frame(y=rnorm(100), x=rnorm(100))
with( dat, plot( y ~ x, col=gradient(y) ) )
```

---

grid.text2 *Grid Text with a Background*

Description

This function produces text as does grid.text, but also generates a background rectangle through grid.rect. Helpful for plotting e.g. overlaying correlation statistics on a plot, where you'd like the element to stand out a little more.

Usage

```
grid.text2(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
  just = "centre", hjust = NULL, vjust = NULL, check.overlap = FALSE,
  default.units = "npc", name = NULL, gp = gpar(col = "black", fill =
  "grey92", lineend = "butt", linejoin = "round"), draw = TRUE, vp = NULL,
  widthAdj = unit(0.05, "npc"), heightAdj = unit(0.05, "npc"))
```

Arguments

label	A character or expression vector. Other objects are coerced by as.graphicsAnnot.
x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
just	The justification of the text relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
check.overlap	A logical value to indicate whether to check for and omit overlapping text.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.

draw	A logical value indiciating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).
widthAdj	A width adjustment parameter, to help control how much horizontal padding there should be between the text and the background rectangle.
heightAdj	A height adjustment parameter, to help control how much vertical padding there should be between the text and the background rectangle.

See Also

grid.text and grid.rect

Examples

```
x <- rnorm(10)
y <- rnorm(10)
if (require(lattice)) xyplot( y ~ x,
        panel = function(x, y, ...) {
          panel.xyplot(x, y, ...)
          grid.text2("some text\nwith a nice\nbackground")
          grid.text2( expression( sum(x[i], i==1, n)^2 ), x=0.8, y=0.8 )
        grid.text2( paste("sum of rnorm(10): ", sum(rnorm(10)) ), x=0.2, y=0.2, just="left" )
         grid.text2( "horizontal justifications work too", x=0.95, y=0.35, just="right" )
        })
## will work for multi-panel plots as well
```

---

hImg *Print Plot to File and Return HTML*

Description

A convenience function that prints a plot to file, and then returns HTML to embed that image in the page.

Usage

```
hImg(my_plot, file, width = 400, height = 300, dpi = 300, dim = NULL,
  scale = 100, device = "png", ...)
```

Arguments

my_plot	a plot object, or code that generates a plot
file	file location to output image to
width	width (in pixels) of the plot
height	height (in pixels) of the plot
dpi	the number of dots per inch used. Default is high to ensure plots are crisp on all displays

dim	passed to par(mfrow); used if making multiple base-R plots
scale	the scale factor to use when scaling plots for web display.
device	the device to use for the plot call.
...	optional arguments passed to png

Details

The dim attribute is passed on to par(mfrow='dim'); ie, it is used if you are calling a plot function that writes more than one plot.

The png device is used.

Examples

```
library(lattice)
## generate an xyplot, write it to file, and return HTML
## code that sources the generated image
dat <- data.frame( x = rnorm(100), y = rnorm(100) )
hImg( file = "plot_output.png",
  xyplot( y ~ x, dat )
)
```

---

hSvg *Print SVG to File and Return HTML*

Description

A convenience function that prints a plot to file, and then returns HTML to embed that image in the page. Used for SVG images.

Usage

```
hSvg(my_plot, file, width = 400, height = 300, dim = NULL, scale = 100,
  ...)
```

Arguments

my_plot	a plot object, or code that generates a plot
file	location to output file
width	width (in pixels) of the plot
height	height (in pixels) of the plot
scale	the scale used to scale the plot from inches to pixels, for display in a web browser
dim	passed to par(mfrow); used if making multiple base-R plots
...	passed to svg

Examples

```
library(lattice)
## generate an xyplot, write it to file, and return HTML code that
## sources the generated image
dat <- data.frame( x = rnorm(100), y = rnorm(100) )
hSvg( file = "plot_output.svg",
  xyplot( y ~ x, dat )
)
```

`html` *Print HTML Elements*

Description

Use this function to output HTML code for use in R Markdown documents or otherwise.

Usage

```
html(..., file = "", .sub = NULL)
```

Arguments

`...`	A set of HTML tag functions. See examples for details.
`file`	Location to output the generated HTML.
`.sub`	A named list of substitutions to perform; we substitute each symbol from the names of `.sub` with its corresponding value.

See Also

`makeHTMLTag`, for making your own tags.

Examples

```
html(
  h1("Welcome!"),
  div(class="header", table( tr( td("nested elements are ok") ) ) ),
  footer(class="foot", "HTML5 footer")
)
```

htmlTable *Generate an HTML Table*

Description

This function is used to generate an HTML table; it wraps to `knitr::kable` but gives some 'extras'; in particular, it allows us to set the class, id, and other HTML attributes.

Usage

```
htmlTable(x, class = "table table-condensed table-hover", id = NULL,
  style = NULL, attr = NULL, output = TRUE, ...)
```

Arguments

`x`	A `data.frame` or `matrix`.
`class`	The CSS class to give the table. By default, we use Twitter bootstrap styling – for this to take effect, your document must include bootstrap CSS.
`id`	The CSS id to give the table.
`style`	Custom styling to apply to the table.
`attr`	Other attributes we wish to apply to the table.
`output`	Whether we should write the output to the console. We hijack the `kable` argument.
`...`	Optional arguments passed to `kable`.

Examples

```
df <- data.frame(`P Values`=runif(1000), Group=1:1000)
htmlTable( head(df[ order(df$P, decreasing=FALSE), ]) )
## wow! look at all that significance! ...
```

in_interval *Determine if Value Lies within Interval*

Description

This function determines whether elements of a numeric vector `x` lie within boundaries `[lo, hi)`. Marginally slower than the R equivalent code `x >= lo & x < hi` for small vectors; much faster for very large vectors.

Usage

```
in_interval(x, lo, hi, include.lower = TRUE, include.upper = FALSE)
```

Arguments

x	numeric. vector of numbers.
lo	numeric, length 1. lower boundary.
hi	numeric, length 1. upper boundary.
include.lower	boolean. include the lower endpoint?
include.upper	boolean. include the upper endpoint?

Examples

```
x <- runif(100); lo <- 0.5; hi <- 1
f <- function(x, lo, hi) {
  return( x >= lo & x < hi )
}
stopifnot( all( in_interval( x, lo, hi ) == f(x, lo, hi) ) )
```

---

is.sorted *Test if an Object is Sorted*

Description

Test if an object is sorted, without the cost of sorting it. Wrapper to `is.unsorted`.

Usage

```
is.sorted(x, na.rm = FALSE, strictly = FALSE)
```

Arguments

x	an R object with a class or a numeric, complex, character or logical vector.
na.rm	logical. Should missing values be removed before checking?
strictly	logical indicating if the check should be for strictly increasing values.

See Also

`is.unsorted`

Examples

```
stopifnot( is.sorted(1, 2, 4) )
```

kAnova *Nicely Formatted ANOVA Table*

Description

Returns a nicely formatted ANOVA table. See kCoef for other details.

Usage

```
kAnova(fit, test = "LRT", swap.periods = TRUE)
```

Arguments

fit	the model fit to generate an ANOVA table for.
test	the type of test to perform. default is likelihood-ratio test (LRT).
swap.periods	swap periods with spaces?

Examples

```
x <- rnorm(100)
y <- ifelse( x + runif(100) > 1, 1, 0 )
myFit <- glm( y ~ x, family="binomial" )
kAnova( myFit )
```

kCoef *Nicely Formatted Model Coefficient Output*

Description

A customized coefficient function that assigns better row names to the coefficient matrix returned by coef() for a model fit. Also includes some arguments for parsing of variable names.

Usage

```
kCoef(fit, remove_underscore = TRUE, remove_dollar = TRUE,
  swap_periods = TRUE)
```

Arguments

fit	the model fit we wish to generate coefficients from.
remove_underscore	remove underscores (and all elements after) in a variable?
remove_dollar	remove all elements before and including a $ in a variable name?
swap_periods	swap periods with spaces?

Details

NOTE:

Models with interaction effects are currently not handled.

Value

a matrix of coefficients with nicely formatted names.

Note

The names given assume default contrasts in your model fit; ie, the default is `contr.treatment`, where each level of a factor is compared to a reference.

Examples

```
## How the remove_underscore and remove_dollar arguments act:
## An example:
##                     kDat$variable_other_stuff
## remove_underscore:  ++++++++++++------------
## remove_dollar:      -----++++++++++++++++++++

x <- rnorm(100); y <- x * runif(100)
z <- as.factor( rep( c("apple", "banana", "cherry", "date"), each=25 ) )
myFit <- lm( y ~ x + z )

## compare the output of these two: which do you prefer?
coef( summary( myFit ) )
kCoef( myFit )
```

---

kFivenum *Fivenum with Names*

Description

A wrapper to `stats::fivenum` that also produces variable names.

Usage

```
kFivenum(x, na.rm = TRUE)
```

Arguments

`x`	numeric, maybe including `NA`s and `Inf`s.
`na.rm`	logical. remove NAs?

Value

`data.frame` version of five number summary

See Also

fivenum

---

kImg *HTML - Source an Image*

Description

Convenience function for cat-ing out HTML markup for an image as .

Usage

```
kImg(x, width = 480, height = 480)
```

Arguments

x	path to an image you want to source
width	width (in pixels) of the image
height	height (in pixels) of the image

See Also

hImg

---

kLoad *Load and Assign an R Object*

Description

The regular load function keeps the old variable name used when saving that object. Often, we would prefer to assign the loaded object to a new variable name. Hence, this function.

Usage

```
kLoad(...)

getload(...)
```

Arguments

...	args to pass to load

Details

If multiple arguments are supplied, they will be concatenated through file.path.

Value

the object stored in the load-ed object

See Also

load

Examples

```
dat <- data.frame( x = c(1,2,3), y=c('a','b','c') )
save( dat, file="dat.rda" )
rm( dat )
my_data <- kLoad( "dat.rda" )
## we protect ourselves from 'forgetting' the name of the
## object we saved
```

---

kmeans_plot *k-means Diagnostic Plot*

Description

Using kmeans, plot percentage variance explained vs. number of clusters. Used as a means of picking k.

Usage

```
kmeans_plot(dat, nmax = 20, ...)
```

Arguments

dat	numeric matrix of data, or an object that can be coerced to such a matrix (such as a numeric vector or a data frame with all numeric columns).
nmax	maximum number of clusters to examine
...	optional arguments passed to xyplot

See Also

kmeans

Examples

```
data(iris)
kmeans_plot(iris[,1:4])
```

kMerge *Merge (Left Join) with Order Retainment*

Description

merge will mangle the order of the data frames it is merging. This is a simple modification to ensure that the order in data frame x is preserved when doing a 'left join'; ie, merge(x, y, all.x=TRUE, ...). That is, if we want to merge a data frame x with another data frame y, we can merge in the parts of y whose index matches with that of x, while preserving the ordering of x.

Usage

```
kMerge(x, y, by = intersect(names(x), names(y)), by.x = by, by.y = by,
  ...)
```

Arguments

x	the data.frame you wish to merge y into.
y	the data.frame to be merged.
by	specifications of the columns used for merging. See 'Details' of merge.
by.x	specifications of the columns used for merging. See 'Details' of merge.
by.y	specifications of the columns used for merging. See 'Details' of merge.
...	optional arguments passed to merge.

Value

data.frame

See Also

merge

Examples

```
x <- data.frame( id=5:1, nums=rnorm(5) )
y <- data.frame( id=1:3, labels=c(1, 2, 2) )
merge(x, y, all.x=TRUE) ## re-ordered the data.frame
merge(x, y, all.x=TRUE, sort=FALSE) ## nope - NAs cause problems
kMerge(x, y, by="id") ## preserves ordering of x, even with NAs

## an id entry appears more than once in y
y <- data.frame( id=c(1, 1, 2), labels=c(1, 2, 3) )
kMerge(x, y, by="id")
```

Kmisc *Kmisc*

Description

This package contains utility function for some common data reshaping operations, and also some HTML utility functions for greater control over R markdown documents.

See Also

html

Kmisc.knit2html *Knit an Rmd File to HTML with Kmisc Styling*

Description

This function 'knits' an R Markdown document with `knitr`, and injects HTML, CSS and JavaScript from CSS for nice, interactive HTML reports.

Usage

```
Kmisc.knit2html(input, ...)
```

Arguments

`input`	An `.Rmd` file.
`...`	Optional arguments passed to `knit`.

kSave *Write out and Save a Tabular File*

Description

A function that both writes a file to table with `write.table`, and saves it with the same name but a separate file extension.

Usage

```
kSave(x, file, lvl = 1, Rext = ".rda", ...)
```

Arguments

x	the R object you want to save / write to file
file	the location to write the file to, with extension desired for object written by write.table
lvl	how many extensions do you want to strip from your output file?
Rext	the extension to use for the saved object.
...	optional arguments passed to write.table

Examples

```
dat <- data.frame( x=c(1,2,3), y=c('a','b','c') )
kSave( dat, file="dat.txt" )
## the file 'dat.rda' is written as well - let's see if it exists
dat2 <- kLoad( "dat.rda" )
stopifnot( identical(dat, dat2) ) ## TRUE
```

---

kSvg *HTML - Source an SVG file*

Description

Convenience function for cat-ing out HTML markup for an SVG image, using <embed>.

Usage

```
kSvg(file = NULL, width = 4, height = 4, class = NULL)
```

Arguments

file	path to the SVG file you want to embed
width	width (in pixels) of the SVG file (or, more accurately, canvas in which that file is displayed)
height	height (in pixels) of the SVG file (or, more accurately, canvas in which that file is displayed)
class	class passed to the <embed> tag

kTable *Make Nicely Formatted Tables*

Description

Function for creating nice 1D and 2D tables. Tables are generated and formatted with both counts and percentages. Primarily intended to be used with R Markdown documents, calling some of the table printing functions. The function returns a `data.frame` in a format that can be used with utility HTML generation functions.

Usage

```
kTable(x, y = NULL, deparse.level = 2, top.left.cell = "",
  col.names = NULL, row.names = NULL, left.label = NULL,
  top.label = NULL, google = FALSE)
```

Arguments

`x`	the `x` variable to build a table on.
`y`	optional: the `y` variable to build a table on. Used for 2x2 contingency tables.
`deparse.level`	passed to `table`; `deparse.level=2` allows us to pass through variable names.
`top.left.cell`	the string to set in the top left cell of the table.
`col.names`	a vector of column names to use on the outputted table; typically this is parsed from the variables passed through.
`row.names`	a vector of row names to use on the outputted table; typically this is parsed from the variables passed through.
`left.label`	the label to use for the rows; typically parsed from `x`. Only used for 2D tables (ie, when `y` is not null).
`top.label`	the label to use for the columns; typically parsed from `y`. Only used for 2D tables (ie, when `y` is not null).
`google`	used if you plan on passing the table to `gvistable` from the `googleVis` package.

Examples

```
x <- rbinom(100, size=2, p=0.1)
y <- rbinom(100, size=2, p=0.1)

## try these in an R markdown document for best results
kTable(x)
my_table <- kTable(x, y, top.left.cell="foo", left.label="bar", top.label="baz")
pxt( my_table )
```

labeller *ggplot2 labeller*

Description

This function works as a labelling mapper for ggplot2, typically used in facet_grid. All arguments must be named. Items are mapped as name => value, where name represents the original levels of the factor used for facetting.

Usage

```
labeller(..., .parse = TRUE)
```

Arguments

...	A set of named arguments.
.parse	boolean; if TRUE we parse the text as though it were an expression.

Examples

```
if (require(ggplot2)) {

  df <- data.frame(
    x=1:100,
    y=rnorm(100),
    grp=rep( c("tau+", "tau-"), each=50 ) ## levels are "tau+", "tau-"
  )

  f <- labeller(
    `tau-` = 'tau["-"]',
    `tau+` = 'tau["+"]'
  )

  ggplot(df, aes(x=x, y=y)) +
    geom_point() +
    facet_grid(". ~ grp", labeller=f)

  df$grp2 <- factor(rep( c("beta+", "beta-"), each=50 ))

  f <- labeller(
    `tau-` = 'tau["-"]',
    `tau+` = 'tau["+"]',
    `beta+` = 'beta["+"]',
    `beta-` = 'beta["-"]'
  )

  ggplot(df, aes(x=x, y=y)) +
    geom_point() +
    facet_grid("grp ~ grp2", labeller=f)
}
```

lg *length(grep(...))*

Description

This is a wrapper to a `length( grep( ... ) )`. See examples for usage. Primarily intended for interactive, not programmatic, use.

Usage

```
lg(pattern, x, perl = TRUE, ...)
```

Arguments

`pattern`	regex pattern passed to `grep`.
`x`	a vector on which we attempt to match `pattern` on.
`perl`	boolean. use perl-compatible regular expressions?
`...`	additional arguments passed to `grep`.

See Also

`re_exists`

Examples

```
x <- c("apple", "banana", "cherry")
if( lg( "^ap", x ) > 0 ) {
  print( "regular expression '^ap' found in 'x'" )
  }
```

list2df *Convert list to data.frame*

Description

This function converts a list to a data frame, assuming that each element of the list is of equal length.

Usage

```
list2df(list, inplace = FALSE)
```

Arguments

`list`	A list.
`inplace`	Boolean. If `TRUE`, we convert the list in place, so that the `list` itself is transformed into a `data.frame`, sans copying.

lu *Number of non-NA unique elements in a vector*

Description

Returns the number of non-NA unique elements in a vector. A wrapper to `length( unique( x[!is.na(x)], ... ) )`. Primarily intended for interactive, not programmatic, use.

Usage

```
lu(x, ...)
```

Arguments

x	a vector
...	passed to unique

makeHTMLTable *Make HTML Table from R 'table-like' Object*

Description

Function for making HTML tables from an R 'table-like' object; ie, a `data.frame` or a `matrix`. It simply parses the item as an HTML table.

Usage

```
makeHTMLTable(x, attr, row.spans = 0, col.spans = 0,
  use.row.names = FALSE, use.col.names = FALSE, clean = TRUE,
  replace.periods = TRUE)
```

Arguments

x	the `data.frame` / `matrix` you want to convert to an HTML table.
attr	attributes to be passed to the <table> tag, as raw HTML.
row.spans	a matrix specifying desired row.spans, for largers cells.
col.spans	a matrix specifying desired column spans, for larger cells.
use.row.names	if you submit an object with row names, use those names in construction of the table.
use.col.names	if you submit an object with column names, use those names in construction of the table.
clean	boolean. if `TRUE`, we print all numeric values with 4 digits. Alternatively, we can pass a format specifier as used by sprintf.
replace.periods	replace periods with spaces?

Details

The row.spans and col.spans argument can be specified as a matrix to set the row or column span of a certain cell to be >1, if desired. See `pxt` for an example implementation. It will also handle 'boxes', e.g. cells with both rowspan and colspan > 1.

Note that the default behavior is to `'clean'` numeric input; this prints numeric values with a maximum of four digits; ie, through the `"%.4G"` format specifier. Alternatively, you can use a format specifier (as used in `sprintf`) to ensure numbers are formatted and displayed as desired.

Examples

```
dat <- data.frame( apple=c(1.2150125, 2, 3), banana=c("a", "b", "c") )
makeHTMLTable( dat )
```

---

makeHTMLTag *Make HTML Elements*

Description

Creates a function that returns a function that can be used to generate HTML elements. See examples for usage.

Usage

```
makeHTMLTag(tag, ...)
```

Arguments

`tag`	the HTML tag to use.
`...`	a collection of named and unnamed arguments; named arguments are parsed as attributes of the tag, unnamed arguments are pasted together into the inner data of the tag.

Details

This function returns a function that can be called as an HTML tag generating function. For example, by calling `p <- makeHTMLTag("p")`, we can generate a function that interprets all named arguments as attributes, and all unnamed arguments as 'data', which is generated for a `p` HTML tag.

See Also

`html`

Examples

```
div <- makeHTMLTag("div")
my_class = "orange"
x <- "some text"
div( class=my_class, id="hello", "This is ", x )
```

make_dummy *Make Dummy Variables from a Factor*

Description

This functions converts a single factor into dummy variables, with one dummy variable for each level of that factor. Names are constructed as <varName>_<level>.

Usage

```
make_dummy(x)
```

Arguments

x	an object coercable to factor.

Examples

```
x <- factor( rep( c("a", "b", "c", "d"), each=25 ) )
make_dummy(x)
```

manhattan_plot *Make a Manhattan Plot*

Description

Generates a manhattan plot (a plot of -log10(p-val)) for a set of markers by chromosome and base-pair position.

Usage

```
manhattan_plot(pval, bp, chr, groups = NULL, cutoff = NULL,
  xlab = "Chromosome (base-pair position)",
  ylab = expression(paste(-log[10](italic(p)))), transform = TRUE,
  cex = 0.5, ...)
```

Arguments

pval	A vector of p-values.
bp	A vector of base-pair positions, corresponding to the genomic location for which that p-value is associated (typically, the SNP location).
chr	The chromosomal location associated with the p-value.
groups	A groups vector: used if you want to overlay multiple manhattan plots.
cutoff	optional. By default, a Bonferroni cutoff line is drawn on the plot; if you want to plot a custom cut-off line you can specify the cutoff here.

xlab	The label to use for the x axis.
ylab	The label to use for the y axis.
transform	boolean; if TRUE, we compute -log10(pval); otherwise, we use pval as-is, assuming that pval has been transformed accordingly.
cex	Multiplier for the point size.
...	Optional arguments passed to xyplot.

Examples

```
pval <- runif(1E4)
bp <- c(1:5E3, 1:5E3)
chr <- rep(1:22, length.out=1E4)
groups=rep( c("Phenotype 1", "Phenotype 2"), each=5E3 )
manhattan_plot( pval, bp, chr, groups, main="Two Phenotype MH Plot" )
manhattan_plot( pval, bp, chr, main="Manhattan Plot" )
```

---

mat2df *Convert a Matrix to a DataFrame*

Description

Identical to as.matrix.data.frame, but faster.

Usage

```
mat2df(x)
```

Arguments

x	A matrix.

---

matches *Count Matches*

Description

This function returns a matrix of matches between each argument passed. Each cell x_ij in the output denotes how many times the elements in input i were found in input j.

Usage

```
matches(...)
```

Arguments

...	A set of (possibly named) arguments, all of the same type.

Examples

```
x <- c("a", "b", "c", "d")
y <- c("a", "b", "c")
z <- c("a", "b", "d")
matches(x, y, z)
```

---

melt_ *Make a 'Wide' data set 'Long'*

Description

Inspired by `reshape2::melt`, we melt `data.frames` and `matrixs`. This function is built for speed.

Usage

```
melt_(data, ...)

## S3 method for class 'data.frame'
melt_(data, id.vars, measure.vars,
  variable.name = "variable", ..., value.name = "value")

## S3 method for class 'matrix'
melt_(data, ...)
```

Arguments

`data`	The `data.frame` to melt.
`...`	Arguments passed to other methods.
`id.vars`	Vector of id variables. Can be integer (variable position) or string (variable name). If blank, we use all variables not in `measure.vars`.
`measure.vars`	Vector of measured variables. Can be integer (variable position) or string (variable name). If blank, we use all variables not in `id.vars`.
`variable.name`	Name of variable used to store measured variable names.
`value.name`	Name of variable used to store values.

Details

If items to be stacked are not of the same internal type, they will be promoted in the order `logical` > `integer` > `numeric` > `character`.

Examples

```
n <- 20
tmp <- data.frame( stringsAsFactors=FALSE,
  x=sample(letters, n, TRUE),
  y=sample(LETTERS, n, TRUE),
  za=rnorm(n),
  zb=rnorm(n),
  zc=rnorm(n)
)

stopifnot(
  identical(
    melt_(tmp, id.vars=c('x', 'y')),
    melt_(tmp, measure.vars=c('za', 'zb', 'zc'))
  )
)
```

---

ngrep *Pattern matching and Replacement*

Description

These functions provide simple extensions to base regular expressions in R, primarily intended to assist with extraction of elements based on the result of a regular expression evaluation.

Usage

```
ngrep(pattern, x, ignore.case = FALSE, perl = FALSE, value = FALSE,
  fixed = FALSE, useBytes = FALSE)

fgrep(pattern, x, ignore.case = FALSE, value = FALSE, useBytes = FALSE,
  invert = FALSE)

re_exists(x, pattern, perl = TRUE, fixed = FALSE, ...)

re.exists(pattern, x, perl = TRUE, fixed = FALSE, ...)

re_extract(x, pattern, perl = TRUE, fixed = FALSE, ...)

extract.re(x, pattern, perl = TRUE, fixed = FALSE, ...)

re_without(x, pattern, perl = TRUE, fixed = FALSE, ...)

without.re(x, pattern, perl = TRUE, fixed = FALSE, ...)

re_extract_rows(x, pattern, match_var = rownames(x), perl = TRUE,
  fixed = FALSE, ...)
```

```
extract_rows.re(x, pattern, match_var = rownames(x), perl = TRUE,
  fixed = FALSE, ...)

re_without_rows(x, pattern, match_var = rownames(x), perl = TRUE,
  fixed = FALSE, ...)

without_rows.re(x, pattern, match_var = rownames(x), perl = TRUE,
  fixed = FALSE, ...)
```

Arguments

x	An R object (in the case of re_ methods), or a character vector (in the case of ngrep, fgrep)
pattern	character string containing a character string to be matched in the given character vector; coerced by as.character if possible.
ignore.case	boolean; if TRUE we perform case-insensitive matching.
perl	boolean; if TRUE, we use perl-compatible regular expressions.
value	boolean; if TRUE we return the actual matches; if FALSE we return the indices corresponding to the matches.
fixed	boolean; if TRUE the pattern is matched as-is. Overrides all conflicting arguments.
useBytes	boolean; if TRUE we perform matching byte-by-byte rather than character by character.
match_var	A variable to match on, as used in the re_extract_rows function.
invert	Invert the results of the regular expression match?
...	Optional arguments passed to grep or grepl.

Details

The order is *reversed* for the re_ set of functions; i.e., an R object is expected first, rather than a regular expression pattern.

See Also

grep

grep, regex

Examples

```
ngrep( "abc", c("abc", "babcd", "abcdef", "apple"), value=TRUE )
if( re_exists(c("apple", "banana"), "^ap") ) print("yay!")
dat <- data.frame( x=letters, y=LETTERS )
rownames(dat) <- 1:26
## get all rows in dat with a 1, 2, 3 or 4 in the name
re_extract_rows( dat, "[1-4]" )
dat <- data.frame( x=letters, y=LETTERS )
rownames(dat) <- 1:26
```

```
## get all rows in dat with a 1, 2, 3 or 4 in the name
re_without_rows( dat, "[0-4]" )
```

---

p1t *Make 1D HTML Table*

Description

This tabling function is intended for the output of kTable, as generated when only one 'data' argument is passed.

Usage

```
p1t(x, class = "table table-condensed table-striped table-hover", id = NULL,
  ...)
```

Arguments

x	a data.frame, typically output of kTable.
class	class to be passed to HTML table; used for CSS styling.
id	id to be passed to HTML table; used for CSS styling.
...	optional arguments passed to makeHTMLTable.

See Also

kTable, makeHTMLTable

Examples

```
y <- factor( rbinom( 100, 2, 0.2 ) )
p1t( kTable( y ) )
```

---

pad *Pad an Object with NAs*

Description

This function pads an R object (list, data.frame, matrix, atomic vector) with NAs. For matrices, lists and data.frames, this occurs by extending each (column) vector in the object.

Usage

```
pad(x, n)
```

Arguments

x	An R object (list, data.frame, matrix, atomic vector).
n	The final length of each object.

par.reset *Restore the 'par' settings*

Description

If you have been mucking around with parameter settings, this function will revert to the parameter settings that were available when this package was loaded.

Usage

```
par.reset()
```

pMerge *Merge a Data Frame 'into' Another*

Description

This function will merge a data frame `df2` 'into' a data frame `df1`, preserving `df1` as much as possible in the merger. Hence I call this a 'preserving' merge, or `pMerge`.

Usage

```
pMerge(df1, df2, by = intersect(names(df1), names(df2)), doCheck = FALSE)
```

Arguments

df1	the data.frame which we are preserving
df2	the data.frame we are merging into df1
by	character; name of the variable we are merging over
doCheck	boolean; set this if you want to perform more extensive (but slower) error checking

Examples

```
df1 <- data.frame( stringsAsFactors=FALSE,
  x=1:1000,
  y=sample(LETTERS, size=1000, replace=TRUE)
)

df2 <- data.frame( stringsAsFactors=FALSE,
  x=sample( 1:2000, size=2000, replace=TRUE ),
  z=sample( letters, size=2000, replace=TRUE ),
  q=sample( LETTERS, size=2000, replace=TRUE )
)

dMerged <- pMerge( df1, df2, by="x" )
stopifnot( all.equal( df1, dMerged[1:ncol(df1)] ) )
```

pp_plot *Construct a Probability-Probability Plot from a Set of P-Values*

Description

This function constructs a probability-probability plot as based on a vector of p-values.

Usage

```
pp_plot(x, ...)
```

Arguments

x	A vector of p-values; numbers within the range 0 to 1.
...	Optional arguments passed to xyplot. Note that a custom panel function is used for generating the plot and hence you shouldn't try to generate your own panel function.

Examples

```
pp_plot( runif(100), main="PP-Plot of 100 random uniforms" )
```

prepare_package *Prepare Package*

Description

This function prepares the package such that all the C / C++ source files are concatenated into one file (for each source). This decreases compilation time, and produces a tarball that can be used for submission to CRAN.

Usage

```
prepare_package(build = TRUE, check = TRUE, install = FALSE,
  copy.tarball = TRUE)
```

Arguments

build	Build the package with R CMD build?
check	Check the package with R CMD check?
install	Install the package with R CMD INSTALL? Only done if build is TRUE as well.
copy.tarball	If build && copy.tarball, we copy the tarball generated by R CMD build to the "dist" folder.

print.kHTML *Print kHTML Objects*

Description

By default, we cat out kHTML objects as we typically intend to embed them in R Markdown documents. This is mainly used for printing of items in the environment html.

Usage

```
## S3 method for class 'kHTML'
print(...)
```

Arguments

...	a set of kHTML objects (strings).

See Also

html

Examples

```
Kmisc:::.html$br()
```

pxt *Make 2x2 HTML Contingency Table*

Description

Function for outputting cross-tabulated tables as marked-up HTML. CSS styling can be used to make these tables look especially nice.

Usage

```
pxt(x, class = "twoDtable", id = NULL, ...)
```

Arguments

x	a 2x2 table; typically something returned from kTable(x,y)
class	class to be passed to HTML table; used for CSS styling.
id	id to be passed to HTML table; used for CSS styling.
...	optional arguments passed to makeHTMLTable.

See Also

kTable

Examples

```
x <- rbinom( 100, 2, 0.2 )
y <- rbinom( 100, 2, 0.2 )
pxt( kTable(x, y) )
```

---

pymat *Python-style Formatting of Strings.*

Description

This function allows Python-style formatting of strings, whereby text of the form `{0}, {1}, ..., {n}` is substituted according to the matching argument passed to `....` `0` corresponds to the first argument, `1` corresponds to the second, and so on.

Usage

```
pymat(x, ..., collapse = ", ")
```

Arguments

`x`	A string with arguments to be replaced in the form of `{0}, {1}, ..., {n}`.
`...`	Arguments to be substituted into `x`.
`collapse`	If vectors of length greater than 1 are passed to `...`, then we collapse the vectors with this separator.

Examples

```
pymat(
  "My favourite fruits are: {0}, {1}, and {2}.",
  "apple", "banana", "orange"
)

pymat(
  "My favourite fruits are: {0}.",
  c("apple", "banana", "orange"), collapse=", "
)
```

`rcpp_apply_generator` *Rcpp Apply Generator*

Description

Use this function as a generator for your own `apply` functions; that is, functions you would like to apply over rows or columns of a matrix.

Usage

```
rcpp_apply_generator(fun, includes = NULL, depends = NULL, inline = TRUE,
  returnType = "double", name = NULL, file = NULL, additional = NULL)

Rcpp_apply_generator(fun, includes = NULL, depends = NULL, inline = TRUE,
  returnType = "double", name = NULL, file = NULL, additional = NULL)
```

Arguments

`fun`	A character string defining the C++ function. It must be in terms of a variable `x`, and it must return a `double`. `x` is a reference to the current row/column being iterated over.
`includes`	Other C++ libraries to include. For example, to include boost/math.hpp, you could pass c("<boost/math.hpp>"). Rcpp is included by default, unless RcppArmadillo is included as well (since Rcpp is included as part of the RcppArmadillo include)
`depends`	Other libraries to link to. Linking is done through Rcpp attributes.
`inline`	boolean; mark this function as inline? This may or may not increase execution speed.
`returnType`	The return type of your function; must be a scalar that is `wrap`pable by Rcpp. Currently, the supported choices are `double`, `int`, and `bool`.
`name`	An internal name for the function.
`file`	A location to output the file. Defaults to a temporary file as generated by `tempfile()`.
`additional`	Other C++ code you want to include; e.g. helper functions. This code will be inserted as-is above the code in `fun`.

Examples

```
## Not run:
x <- matrix(1:16, nrow=4)
cvApply <- rcpp_apply_generator("return mean(x) / sd(x);")
squaredSumApply <- rcpp_apply_generator("
  double out = 0;
  for( int i=0; i < x.size(); i++ ) {
    out += x[i];
  }
```

```
  out = out*out;
  return out;
  ")
cvApply(x, 2)
apply(x, 2, mean) / apply(x, 2, sd)
if( require(microbenchmark) ) {
 f <- function(x) { mean(x) / sd(x) }
 microbenchmark( cvApply(x, 2), apply(x, 2, f) )
}
## example with bool
anyBig <- rcpp_apply_generator( returnType="bool", '
  return is_true( any( x > 10 ) );
  ')
anyBig(x, 2)
anyBig(x, 1)
## example with boost's gcd. silly but demonstrative.
## intended to be applied to matrices with 2 rows and n columns
gcdApply <- rcpp_apply_generator( returnType="int",
  includes="<boost/math/common_factor.hpp>",
  fun='
  return boost::math::gcd( (int)x[0], (int)x[1] );
  ')
M <- matrix( c(4, 6, 20, 25, 10, 100), nrow=2 )
gcdApply(M, 2)

## End(Not run)
```

`Rcpp_gen_makevars` *Reproduce Rcpp Makevars Files*

Description

If you're building a package and want a simple set of Makevars files to export, this function will handle it for you. Borrowed from the Rcpp `rcpp.package.skeleton` function.

Usage

```
Rcpp_gen_makevars(src = file.path(getwd(), "src"))

rcpp_gen_makevars(src = file.path(getwd(), "src"))
```

Arguments

`src`	the location to output the Makevars.

`rcpp_tapply_generator` *Rcpp tapply Generator*

Description

Use this function as a generator for your own `tapply` functions; that is, functions you would like to apply split over some grouping variable. Note that we restrict ourselves to the case where we return a scalar as output; rather than the more general output of `tapply`.

Usage

```
rcpp_tapply_generator(fun, includes = NULL, depends = NULL, inline = TRUE,
  returnType = "double", name = NULL, file = NULL, additional = NULL)

Rcpp_tapply_generator(fun, includes = NULL, depends = NULL, inline = TRUE,
  returnType = "double", name = NULL, file = NULL, additional = NULL)
```

Arguments

`fun`	A character string defining the C++ function. It must be in terms of a variable `x`.
`includes`	Other C++ libraries to include. For example, to include `boost/math.hpp`, you could pass `c("<boost/math.hpp>")`. Rcpp is included by default, unless RcppArmadillo is included as well (since Rcpp is included as part of the RcppArmadillo include)
`depends`	Other libraries to link to. Linking is done through Rcpp attributes.
`returnType`	The return type of your function; note that we require that the generator returns a scalar value of type `double`, `int`, or `bool`.
`inline`	boolean; mark this function as inline? This may or may not increase execution speed.
`name`	An internal name for the function.
`file`	A location to output the file. Defaults to a temporary file as generated by `tempfile()`.
`additional`	Other C++ code you want to include; e.g. helper functions. This code will be inserted as-is above the code in `fun`.

Details

Note that we simplify differently than base R `tapply`: when `simplify=TRUE`, we call `unlist` on the output; hence, a named vector is returned.

read *Read a File*

Description

These functions read a file into memory. We memory map the file for fast I/O. The file is read in as a character vector (length one for `read`, length `n` for `readlines`).

Usage

```
read(file)

readlines(file)
```

Arguments

`file`	Path to a file.

Examples

```
p <- file.path( R.home(), "NEWS" )
if (file.exists(p))
  stopifnot( identical( readLines(p), readlines(p) ) )
```

read.cb *Read Tabular Data from the Clipboard*

Description

Convenience function for reading tabular data from the clipboard. The function checks the system OS and provides the appropriate wrapper call to `read.table`.

Usage

```
read.cb(sep = "\t", header = TRUE, ...)
```

Arguments

`sep`	the delimiter used in the copied text.
`header`	boolean; does the first row contain column names?
`...`	optional arguments passed to `read.table`.

See Also

`read.table`

Examples

```
## with some data on the clipboard, simply write
# x <- read.cb()
```

`remove_chars` *Remove Alphabetic Characters from a Character Vector*

Description

Removes all alphabetic characters from a character vector.

Usage

```
remove_chars(x, remove_spaces = TRUE)
```

Arguments

`x`	A character vector, or vector coercable to character.
`remove_spaces`	boolean; if `TRUE` we remove all white-space as well.

`remove_digits` *Remove Digits from a Character Vector*

Description

Removes all digits from a character vector.

Usage

```
remove_digits(x, remove_spaces = TRUE)
```

Arguments

`x`	A character vector, or vector coercable to character.
`remove_spaces`	boolean; if `TRUE` we remove all white-space as well.

remove_na *Remove NA Entries from a Vector*

Description

This function removes all NA entries from a vector.

Usage

```
remove_na(x)
```

Arguments

x	An (atomic) vector, or a list / data.frame.

Details

For data.frames, we use complete.cases to remove NAs, and hence remove all rows for which an NA value in encountered.

rowApply *Apply Wrappers*

Description

These are thin but clearer wrappers to apply(x, 1,FUN, ...) (row apply) and apply(x, 2, FUN, ...) (column apply). Intended for use with 2D R matrixs. We do a bit more work to ensure row names, column names are passed along if appropriate.

Usage

```
rowApply(X, FUN, ..., drop = TRUE)

colApply(X, FUN, ..., drop = TRUE)
```

Arguments

X	A matrix, or a 2D array.
FUN	The function to be applied.
...	Optional arguments to FUN.
drop	Boolean. If TRUE, we 'drop' dimensions so that results of dimension n x 1 or 1 x n are coerced to vectors.

Details

See apply for more info.

scan.cb *Read Data from the Clipboard*

Description

Convenience function for reading data from the clipboard. Wraps to scan. By default, we assume the data is character, and delimit by new lines.

Usage

```
scan.cb(what = character(), sep = "\n", quiet = TRUE, ...)
```

Arguments

what	passed to scan.
sep	passed to scan.
quiet	passed to scan.
...	passed to scan.

See Also

scan

simp *Area Under the Curve with Simpson's Rule*

Description

This function computes the area under the curve using Simpson's (composite) rule, for a function f(x) evaluated over equally spaced points x.

Usage

```
simp(x, y)
```

Arguments

x	A vector of values x.
y	A vector of values f(x).

size *Print the Object Size, with Auto Units*

Description

Provides an estimate of the memory that is being used to store an R object. Similar to `object.size`, but we set `units="auto"` as default.

Usage

```
size(x, quote = FALSE, units = "auto", ...)
```

Arguments

`x`	An R object.
`quote`	logical, indicating whether or not the result should be printed with surrounding quotes.
`units`	The units to be used in printing the size. Other allowed values are `"Kb"`, `"Mb"`, `"Gb"` and `"auto"`. See `object.size` for more details.
`...`	Arguments to be passed to or from other methods.

split_file *Split a File by Unique Entries in a Column*

Description

This script splits a delimited file by unique entries in a selected column. The name of the entry being split over is appended to the file name (before the file extension).

Usage

```
split_file(file, column, sep = NULL, outDir = file.path(dirname(file),
  "split"), prepend = "", dots = 1, skip = 0, verbose = TRUE)
```

Arguments

`file`	The location of the file we are splitting.
`column`	The column (by index) to split over.
`sep`	The file separator. Must be a single character. If `''`, we guess the delimiter from the first line.
`outDir`	The directory to output the files.
`prepend`	A string to prepend to the output file names; typically an identifier for what the column is being split over.

dots	The number of dots used in making up the file extension. If there are no dots in the file name, this argument is ignored.
skip	Integer; number of rows to skip (e.g. to avoid a header).
verbose	Be chatty?

Details

This function should help users out in the unfortunate case that the data they have attempted to read is too large to fit into RAM. By splitting the file into multiple, smaller files, we hope that each file, post-splitting, is now small enough to fit into RAM.

The focus is on efficient splitting of 'well-mannered' files, so if you have comments, quoted delimiters, cell entries that have paragraphs of unicode text, or other wacky things this is probably not the function for you.

See Also

extract_rows_from_file

---

split_runs *Split by Runs*

Description

Split a vector into a list of runs, such that each entry in the output list is a set of runs encountered. This function accepts two forms of inputs: either a vector where each element of the vector is of length 1 (e.g. `c("A","A", "C", "T")`), or a vector of length 1 interpretted as a long string (e.g. `"AAAACCAGGGACGCCGCGGTTGG"`).

Usage

```
split_runs(x)
```

Arguments

x	A numeric or character vector.

Details

Factors will be coerced to character before splitting.

See Also

rle, for a similar function with different output.

Examples

```
x <- rbinom( 100, 2, 0.5 )
stopifnot( all( x == unlist( split_runs(x) ) ) )
stopifnot( all( as.character(x) == unlist( split_runs( as.character(x) ) ) ) )
y <- paste( collapse="", sample( LETTERS[1:5], 1E5, replace=TRUE ) )
stopifnot( y == paste( collapse="", split_runs(y) ) )
z <- replicate( 25, paste( collapse="", sample( LETTERS[1:5], 1E3, replace=TRUE ) ) )
system.time( lapply(z, split_runs) )
```

---

stack_list *Stack a List of DataFrame-like Objects*

Description

Function for stacking a list, where each component of the list is a `data.frame` containing potentially differing number of rows, but the same number of columns, and with all columns of equivalent class.

Usage

```
stack_list(list, name = "row_names", make_row_names = typeof(attr(list[[1]],
  "row.names")) == "character", keep_list_index = TRUE,
  index_name = "list_index", coerce_factors = TRUE)
```

Arguments

`list`	a list of data frames, or a list of lists with each element of the same length.
`name`	a name to assign to the column of row names generated.
`make_row_names`	boolean. add a column built from the row names? Defaults to `TRUE` only if the row names are of type `character`.
`keep_list_index`	boolean; if `TRUE` we include a vector that indicates which list a particular entry came from.
`index_name`	a name to assign to the column of indices.
`coerce_factors`	boolean; if `TRUE`, we convert factors to their character representation; otherwise, we take the internal integer representation.

Details

When stacking `data.frames` with row names, they are passed into a column called `row_names` to protect from problems with non-unique row names, and also to avoid appending numbers onto these row names as well.

Note that information on factors is lost by default; they will be converted either to characters or their internal integer representations.

Examples

```
x <- data.frame( x=c(1, 2, 3), y=letters[1:3], z=rnorm(3), stringsAsFactors=FALSE )
rownames(x) <- c("apple", "banana", "cherry")
y <- data.frame( x=c(4, 5, 6), y=LETTERS[1:3], z=runif(3), stringsAsFactors=FALSE )
rownames(y) <- c("date", "eggplant", "fig")
tmp1 <- stack_list( list(x, y) )
tmp2 <- do.call( rbind, list(x, y) )
rownames(tmp2) <- 1:nrow(tmp2)
all.equal( tmp1[,1:3], tmp2 )
```

---

strip_extension *Strip File Extension*

Description

Strips the extension from a file name. By default, we assume the extension is separated from the file name by a single period; however, the `lvl` argument lets us specify how many periods are used in forming the file extension.

Usage

```
strip_extension(x, lvl = 1)
```

Arguments

`x`	the file name, including extension.
`lvl`	the number of `'.'` used in defining the file extension.

Examples

```
x <- "path_to_file.tar.gz"
strip_extension(x, lvl=2)
```

---

str_collapse *Collapse a String*

Description

This function collapses a string using Rcpp sugar, and operates similarily to `paste0(..., collapse="")`.

Usage

```
str_collapse(x)
```

Arguments

`x`	A list of character vectors.

str_rev *Reverse a Vector of Strings*

Description

Reverses a vector of 'strings' (a character vector). Not safe for unicode (UTF-8) characters.

Usage

```
str_rev(x)
```

Arguments

x	a character vector.

Details

This function is written in C for fast execution; however, we do not handle non-ASCII characters. For a 'safe' version of `str_rev` that handles unicode characters, see `str_rev2`.

See Also

`str_rev2`

Examples

```
x <- c("ABC", "DEF", "GHIJ")
str_rev(x)
```

str_rev2 *Reverse a Vector of Strings (UTF-8)*

Description

Reverses a vector of 'strings' (a character vector). This will safely reverse a vector of unicode (UTF-8) characters.

Usage

```
str_rev2(x)
```

Arguments

x	a character vector.

Details

This function will handle UTF-8 characters safely. If you are working only with ASCII characters and require speed, see str_rev2.

See Also

str_rev

Examples

```
x <- c("ABC", "DEF", "GHIJ")
str_rev(x)
```

`str_slice` *Slice a Vector at Consecutive Indices*

Description

This function 'slices' the strings of a character vector x at consecutive indices n, thereby generating consecutive substrings of length n and returning the result as a list. Not safe for use with unicode characters.

Usage

```
str_slice(x, n = 1)
```

Arguments

x	a character vector.
n	integer (or numeric coercible to integer); index at which to slice.

Value

A list of length equal to the length of x, with each element made up of the substrings generated from x[i].

Note

Underlying code is written in C for fast execution.

See Also

str_slice2, for slicing a UTF-8 encoded vector.

Examples

```
x <- c("ABCD", "EFGH", "IJKLMN")
str_slice(x, 2)
```

str_slice2 *Slice a Vector at Consecutive Indices*

Description

This function 'slices' the strings of a character vector `x` at consecutive indices `n`, thereby generating consecutive substrings of length `n` and returning the result as a list. This function will safely 'slice' a UTF-8 encoded vector.

Usage

```
str_slice2(x, n = 1, USE.NAMES = TRUE)
```

Arguments

`x`	a character vector.
`n`	integer (or numeric coercible to integer); index at which to slice.
`USE.NAMES`	logical. if names attribute already exists on `x`, pass this through to the result?

Value

A list of length equal to the length of `x`, with each element made up of the substrings generated from `x[i]`.

Note

Safe for use with UTF-8 characters, but slower than `str_slice`.

See Also

`str_slice`, for slicing an ASCII vector.

str_sort *Sort a Vector of Strings*

Description

Sorts a vector of strings lexically, as based on their UTF-8 ordering scheme. Lower-case letters are, by default, 'larger' than upper-case letters. This function will safely sort a UTF-8 vector.

Usage

```
str_sort(x, increasing = TRUE, ignore.case = FALSE, USE.NAMES = FALSE)
```

Arguments

x	a character vector (a vector of 'strings' to sort)
increasing	boolean. sort the string in increasing lexical order?
ignore.case	boolean. ignore case (so that, eg, a < A < b)
USE.NAMES	logical. if names attribute already exists on x, pass this through to the result?

Examples

```
stopifnot( all( str_sort(c("cba", "fed")) == c("abc", "def") ) )
```

---

str_split *Split a Vector of Strings Following a Regular Structure*

Description

This function takes a vector of strings following a regular structure, and converts that vector into a data.frame, split on that delimiter. A nice wrapper to strsplit, essentially - the primary bonus is the automatic coersion to a data.frame.

Usage

```
str_split(x, sep, fixed = FALSE, perl = TRUE, useBytes = FALSE,
  names = NULL)

split2df(x, sep, fixed = FALSE, perl = TRUE, useBytes = FALSE,
  names = NULL)
```

Arguments

x	a vector of strings.
sep	the delimiter / regex you wish to split your strings on.
fixed	logical. If TRUE, we match sep exactly; otherwise, we use regular expressions. Has priority over perl.
perl	logical. Should perl-compatible regexps be used? Ignored when fixed is TRUE.
useBytes	logical. If TRUE, matching is done byte-by-byte rather than character-by-character.
names	optional: a vector of names to pass to the returned data.frame.

Details

Note that the preferred method for reading text data with a single, one character delimiter is through read.table(text=...) or data.table::fread; however, this function is helpful in the case of non-regular delimiters (that you wish to specify with a regex)

See Also

strsplit

Examples

```
str_split(
  c("regular_structure", "in_my", "data_here"),
  sep="_",
  names=c("apple", "banana")
)
x <- c("somewhat_different.structure", "in_this.guy")
str_split( x, "[_\\.]", names=c("first", "second", "third") )
```

---

swap *Swap Elements in a Vector*

Description

This function swaps elements in a vector. See examples for usage.

Usage

```
swap(vec, from, to = names(from))
```

Arguments

vec	the vector of items whose elements you will be replacing.
from	the items you will be mapping 'from'.
to	the items you will be mapping 'to'. must be same length and order as from.

Details

If to is of different type than from, it will be coerced to be of the same type.

See Also

swap_

Examples

```
x <- c(1, 2, 2, 3)
from <- c(1, 2)
to <- c(10, 20)
swap( x, from, to )

## alternatively, we can submit a named character vector
## we translate from value to name. note that this forces
## a conversion to character
```

```
names(from) <- to
swap( x, from )

## NAs are handled sensibly. Types are coerced as needed.
x <- c(1, NA, 2, 2, 3)
swap(x, c(1, 2), c("a", "b") )
```

---

swap_ *Swap Elements in a Vector*

Description

This function swaps elements in a vector. See examples for usage.

Usage

```
swap_(vec, ...)
```

Arguments

vec	the vector of items whose elemetns you will be replacing.
...	A set of named arguments, whereby we translate from names to values of those arguments.

Details

If to is of different type than from, it will be coerced to be of the same type.

See Also

swap

Examples

```
x <- c('a', 'a', 'b', 'c')
swap_(x, a="A")
```

sys *Invoke a System Command*

Description

This function wraps to `system`, but interprets all un-named arguments as things to be `paste`-ed. See `system` for further details.

Usage

```
sys(..., intern = FALSE, ignore.stdout = FALSE, ignore.stderr = FALSE,
  wait = TRUE, input = NULL, show.output.on.console = TRUE,
  minimized = FALSE, invisible = TRUE)
```

Arguments

`...`	System command to be invoked; this gets passed into `paste(..., sep='', collapse='')`.
`intern`	A logical (not `NA`) which indicates whether to capture the output of the command as an R character vector.
`ignore.stdout`	Ignore `stdout`?
`ignore.stderr`	Ignore `stderr`?
`wait`	Should the R interpreter wait for the program to finish execution?
`input`	If a character vector is supplied, this is copied one string per line to a temporary file, and the standard input of `...` is redirected to the file.
`show.output.on.console`	Windows only – show output on console?
`minimized`	Windows only – run the shell minimized?
`invisible`	Windows only – run invisibly?

tapply_ *Faster tapply*

Description

This function acts as a faster version of `tapply` for the common case of splitting an atomic vector by another atomic vector, and then applying a function.

Usage

```
tapply_(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Arguments

X	An atomic vector.
INDEX	A vector coercable to factor; must be one of the common atomic types: factor, integer, numeric, or character.
FUN	The function to be applied. See more details at lapply.
...	Optional arguments to pass to FUN.
simplify	boolean; if TRUE, we unlist the output and hence return a named vector of values.

Examples

```
x <- rnorm(100)
gp <- sample( 1:10, 100, TRUE )
stopifnot( all(
  tapply(x, gp, mean) == tapply_(x, gp, mean)
) )
```

---

transpose *Tranpose an Object*

Description

This functions similarily to R's t, but we add a new method, transpose.list, for transposing lists in a specific way.

Usage

```
transpose(x)

## S3 method for class 'list'
transpose(x)

## S3 method for class 'data.frame'
transpose(x)

## Default S3 method:
transpose(x)
```

Arguments

x	A matrix, data.frame, or matrix-like list.

tree — *Print a Tree Representation of an Object of Nested Lists*

Description

This function returns output similar to that of the command line tool `tree`, except rather than directory/file structure, we simply print the names of lists.

Usage

```
tree(x)
```

Arguments

`x`	A (named) list.

See Also

http://stackoverflow.com/questions/18122548/display-names-of-column-of-recursive-list-as-tree

u — *Unique elements in a vector*

Description

Returns the unique elements in a vector. A wrapper to `unique()`. Primarily intended for interactive, not programmatic, use.

Usage

```
u(...)
```

Arguments

`...`	passed to `unique`.

unmelt_ *Unmelt a Melted Data Frame*

Description

This function undoes the melting process done by either reshape2::melt or melt_.

Usage

```
unmelt_(data, variable = "variable", value = "value")
```

Arguments

data	A data.frame.
variable	The index, or name, of the variable vector; analogous to the vector produced with name variable.name.
value	The value of the value vector; analogous to the vector produced with name value.name.

update_date *Update Date in DESCRIPTION File*

Description

This function for package authors updates the time in the DESCRIPTION file to the current date, as discovered through Sys.Date().

Usage

```
update_date(file = "DESCRIPTION")
```

Arguments

file	The path to the DESCRIPTION file.

us *unlist(strsplit(...))*

Description

This is a thin wrapper to `unlist( strsplit( ... ) )` . Primarily intended for interactive, not programmatic, use.

Usage

```
us(x, split = "", ...)
```

Arguments

x	vector of items, as passed to strsplit
split	the delimiter to split on
...	optional arguments passed to strsplit

See Also

unlist, strsplit

Examples

```
x <- "apple_banana_cherry"
us(x, "_")
```

value_matching *Value Matching*

Description

These are a couple of mostly self-explanatory wrappers around `%in%`.

Usage

```
x %nin% y

x %kin% y

x %knin% y
```

Arguments

x	Vector or NULL: the values to be matched.
y	Vector or NULL: the values to be matched against.

Details

`%nin%` returns a logical vector indicating if there is no match for its left operand. It is the inverse of `x %in% y`.

`%kin%` returns the actual values of `x` for which `x %in% y`.

`%knin%` returns the actual values of `x` for which `x %nin% y`.

`without` *Remove Elements from a Named Object*

Description

Removes elements from an R object with the names attribute set in a 'lazy' way. The first argument is the object, while the second is a set of names parsed from `...`. We return the object, including only the elements with names not matched in `...`.

Usage

```
without(x, ...)
```

Arguments

`x`	An R object with a `names` attribute.
`...`	an optional number of 'names' to match in `x`.

Details

We can be 'lazy' with how we pass names. The `names` passed to `...` are not evaluated directly; rather, their character representation is taken and used for extraction.

See Also

`extract`

Examples

```
dat <- data.frame( x=c(1, 2, 3), y=c("a", "b", "c"), z=c(4, 5, 6) )
## all of these return identical output
dat[ !( names(dat) %in% c("x","z") ) ]
without(dat, x, z)
```

wrap *Wrap a String*

Description

This function operates similarily to strwrap, but pastes the wrapped text back together with line separators. Useful for automatically wrapping long labels.

Usage

```
wrap(x, width = 8, ...)
```

Arguments

x	A character vectors, or an object which can be converted to a character vector by as.character.
width	A positive integer giving the number of characters a line can reach before we wrap and introduce a new line.
...	Optional arguments passed to strwrap.

Examples

```
long_label <- "This is a very long label which needs wrapping."
wrap(long_label)
```

write.cb *Write Tabular Data to the Clipboard*

Description

Directs output of write.table to the clipboard. This can be useful if you want to quickly write some R table out and paste it into some other file, eg. a Word document, Excel table, and so on.

Usage

```
write.cb(dat, row.names = FALSE, col.names = TRUE, sep = "\t",
  quote = FALSE)
```

Arguments

dat	the data file you want to write out; passed to write.table.
row.names	logical. include the row names of dat?
col.names	logical. include the column names of dat?
sep	the delimiter used to separate elements after exporting dat.
quote	logical. include quotes around character vectors in dat?

See Also

write.table

Index