Package ‘analyze.stuff’

August 29, 2018

Type Package

Title Tools that help analyze data

Version 0.2.0

Date 2015-05-29

Author info@ejanalysis.com

Maintainer ejanalysis.com <info@ejanalysis.com>

Description Tools that simplify some basic tasks in exploring and analyzing a dataset in a matrix or data.frame. Key functions help to change many fieldnames to new names using a map of old to new names, create many calculated fields based on formulas specified or saved as text fields (character vector), see how many rows or cols have values above certain cutoffs, get rowMaxs, colMaxs, wtd.rowMeans, wtd.colMeans, see a table of values at 100 weighted percentiles, see how many values are NA or non-NA in each column, etc. The analyze.stuff, proxistat, ejanalysis, and countyhealthrankings packages, once made public can be installed from GitHub.com using the devtools package: devtools::install_github(c(``ejanalysis/analyze.stuff", ``ejanalysis/proxistat", ``ejanalysis/ejanalysis", ``ejanalysis/countyhealthrankings"))

Depends data.table

Imports Hmisc,
matrixStats

License MIT + file LICENSE

Suggests sp

URL http://ejanalysis.github.io

http://www.ejanalysis.com/

RoxygenNote 6.1.0

R topics documented:

`analyze.stuff` *Basic Tools for Analyzing Datasets*

Description

This R package provides some useful tools for analyzing data in matrices and data.frames, such as functions to find the weighted mean of each column of data, add leading zeroes, or find what percent of rows are above some cutoff for each column.

Details

Key functions include

- `change.fieldnames`: Change many fieldnames using map of current to new ones
- `calc.fields`: Create many new calculated fields from data.frame fields by specifying a list of formulas
- `similar.p`, `setdiff2`: Compare two datasets or sets
- `rows.above.count`, `rows.above.pct`: How many rows have values above a cutoff
- `cols.above.count`, `cols.above.pct`: How many cols have values above a cutoff
- `rowMaxs`, `colMaxs`, `rowMins`, `colMins`: Max or min of each row or col in data.frame or matrix
- `wtd.rowMeans`, `wtd.colMeans`: Weighted mean of each row or col
- `pctiles`, `wtd.pctiles`: See a table of values at 100 percentiles, for each field.
- `na.check`, `length2`: How many NA or non-NA values in each column
- `mem`: What objects are taking up the most memory
- `dir2`, `dirr`, `dirdirs`: Directory listing with wildcards, just R-related files, subfolders, etc.

May add later:

- cols.below.count
- cols.below.pct
- cols.below.which
- rows.above.count
- rows.above.pct
- rows.above.which
- rows.below.count
- rows.below.pct
- rows.below.which

Author(s)

info@ejanalysis.com <info@ejanalysis.com>

References

http://ejanalysis.github.io
http://www.ejanalysis.com

**Acknowledgements:
The package **sp** function `spDists`. The matrixStats package provides versions of rowMins, rowMax, colMins, colMaxs and related functions. This package could at some point provide data.frame methods that extend those, but for now it replaces them with slower versions that work on data.frames. Source: Henrik Bengtsson (2015). matrixStats: Methods that Apply to Rows and Columns of a Matrix. R package version 0.13.1-9000. `https://github.com/HenrikBengtsson/matrixStats`

`calc.fields` *Create calculated fields by specifying formulas*

Description

Create calculated fields from formulas that are specified as character strings, returning data.frame of specified results (not all intermediate variables necessarily) e.g., create calculated demographic variables from raw American Community Survey counts. This function is useful if you are working with a dataset with numerous fields, and you want to calculate numerous derived fields from those original fields, and you find it convenient to store all of the formulas in a text file, for example. You could read in the formulas from the file, and apply them to a new version of the dataset to calculate a new version of all of your derived fields.

Usage

```
calc.fields(mydf, formulas, keep)
```

Arguments

`mydf`	Required. A data.frame with strings that are field names (input variables) that may appear in formulas. See example.
`formulas`	Required. A vector of strings that are formulas based on input variables and/or variables calculated from previous formulas. See example.
`keep`	Optional. A vector of strings that are the input and/or calculated variables to return, in case not all intermediate variables are needed. Default is all results of formulas but not any input variables.

Details

This function returns a matrix or vector of results of applying specified formulas to the fields in the input data.frame. Each row of data is used in a formula to provide a row of results.

Value

A data.frame of new variables where columns are defined by keep (or all calculated variables if keep is not specified).

See Also

`change.fieldnames`

Examples

```
# formulas could be ejscreenformulas$formula from the ejscreen package, for example.
 myforms <- c('calcvar1 = b+1', 'calcvar2=calcvar1 + a', 'calcvar3<- paste(a,"x",sep="")')
 # Saving to and reading from a file that stores all these formulas:
 # write.csv(myforms, file='testforms.csv', row.names = FALSE)
 # myforms <- read.csv('testforms.csv')
 mydat  <- data.frame(a=1:10, b=2:11)
 x <- calc.fields(mydat, myforms); x
 # Return only some of the input/output variables:
 calc.fields(mydf=mydat, formulas=myforms, keep=c('a', 'calcvar2','calcvar3') )
```

`change.fieldnames` *Change some or all of the colnames of a data.frame or matrix via a 1-1 map*

Description

Returns a new set of field names, based on the old set of names, which can be specified in a file or as parameters. This provides a convenient way to specify which names will be replaced with which new names, via a map of 1-1 relationships between the old names and new names.

Usage

```
change.fieldnames(allnames, oldnames, newnames, file = NA,
  sort = FALSE)
```

Arguments

`allnames`	Character vector, optional. A vector of all the original fieldnames, such as the results of names(mydataframe).
`oldnames`	Character vector, optional. A vector of only those original fieldnames that you want to change, in any order.
`newnames`	Character vector, optional. A vector of new names, sorted in an order corresponding to oldnames.
`file`	Character, optional. A filename (or path with filename) for a mapping file that is a csv file with two columns named with a header row: oldnames, newnames (instead of passing them to the function as parameters).
`sort`	Logical value, optional, FALSE by default. If FALSE, return new fieldnames. If sort=TRUE, return vector of indexes giving new position of given field, based on sort order of oldnames.

Details

This function returns a character vector of length equal to the number of oldnames (the parameter or the field in the file).

Value

A vector of character strings, the full set of fieldnames, with some or all updated if sort=FALSE (default). Uses oldnames and newnames, or file for mapping. If those are not specified, it tries to open an interactive window for editing a mapping table to create and save it as a csv file.

If sort=TRUE, return vector of indexes giving new position of given field, based on sort order of oldnames. If sort=TRUE, names in oldnames that are not in allnames are ignored with warning, & names in allnames that are left out of oldnames left out of new sort order indexes.

See Also

put.first which make it easier to rearrange the order of columns in a data.frame.

Examples

```
oldnames <- c('PCTILE', 'REGION')
newnames <- c('percentile', 'usregion')
df <- data.frame(REGION=301:310, ID=1:10, PCTILE=101:110, OTHER=1:10)
names(df) <- change.fieldnames(names(df), oldnames, newnames); names(df)
names(df) <- change.fieldnames(names(df), "ID", "identification"); names(df)
# names(df) <- change.fieldnames(names(df)); names(df)  # does not work on MacOSX?
# names(df) <- change.fieldnames(names(df), 'saved fieldnames.csv'); names(df)
df[ change.fieldnames(names(df), c('ID', 'OTHER', 'REGION', 'PCTILE'), sort=TRUE)]
# much like df[ , c('ID', 'OTHER', 'REGION', 'PCTILE') ]
#  change.fieldnames is more useful when file specified
```

---

colMaxs *Get the max value of each column of a data.frame or matrix*

Description

Returns maximum value of each column of a data.frame or matrix.

Usage

```
colMaxs(df, na.rm = TRUE)
```

Arguments

df	data.frame or matrix
na.rm	TRUE by default. Should NA values be removed first

Details

** NOTE: The useful matrixStats package will provide the basis for extended rowMins, rowMax, colMins, colMaxs functions to be made available through this package. Source: Henrik Bengtsson (2015). matrixStats: Methods that Apply to Rows and Columns of a Matrix. R package version 0.13.1-9000.
https://github.com/HenrikBengtsson/matrixStats
Initially, separate functions were written here for those four functions, and the versions here were more flexible and convenient for some purposes, e.g., handling data.frames and different na.rm defaults, but the matrixStats versions are much faster (e.g., by 4x or more). Ideally, this analyze.stuff

package would be modified to just extend those functions by providing them methods to handle data.frames, not just matrix class objects, and perhaps provide new or different parameters or defaults, such as defaulting to na.rm=TRUE instead of FALSE, and handling factor class columns in a data.frame. That has not been done yet, so colMaxs() etc. refer to the slower more flexible ones, and the faster matrix-only ones are via matrixStats::colMaxs etc.

** NOTE: max() and min() and matrixStats::colMaxs from matrixStats etc. default to na.rm=FALSE, but this function defaults to na.rm=TRUE because that just seems more frequently useful.

** NOTE: `min` and `max` & this function will handle character elements by coercing all others in the column to character, which can be confusing – e.g., note that min(c(8,10,'txt')) returns '10' not '8' and max returns 'txt' (also see the help for `Comparison`)

If this worked just like max() and min(), cols that are factors would make this fail. max or min of a factor fails, even if as.character() of the factor would return a valid numeric vector. That isn't an issue with a matrix, but a data.frame might have numbers stored as factor. To fix that, this uses `factor.as.numeric` with parameters that try to convert character or factor columns to numeric.

Based on how `min` and `max` behave, return Inf or -Inf if no non-missing arguments to min or max respectively. To suppress that warning when using this function, use `suppressWarnings`(func(x))

Value

vector of numbers with length equal to number of cols in df

See Also

`factor.as.numeric` `rowMaxs` `rowMins` `colMaxs` `colMins` `count.above` `pct.above` `pct.below` `cols.above.which` `cols.above.pct`

Other functions for max and min of rows and columns: `colMins`, `rowMaxs`, `rowMins`

Examples

```
blah <- rbind(NA, data.frame(a=c(0, 0:8), b=c(0.1+(0:9)), c=c(1:10), d=c(rep(NA, 10)),
  e=TRUE, f=factor('factor'), g='words', stringsAsFactors=FALSE) )
cbind(blah, min=rowMins(blah), max=rowMaxs(blah))
rbind(blah, min=colMins(blah), max=colMaxs(blah))
blah <- blah[ , sapply(blah, function(x) is.numeric(x) | is.logical(x)) ]
cbind(blah, min=rowMins(blah), max=rowMaxs(blah),
  mean=rowMeans(blah, na.rm=TRUE), sum=rowSums(blah, na.rm=TRUE))
rbind(blah, min=colMins(blah), max=colMaxs(blah),
  mean=colMeans(blah, na.rm=TRUE), sum=colSums(blah, na.rm=TRUE))
  # ** Actually, matrixStats does this ~4x as quickly,
  # although no practical difference unless large dataset:
  n <- 1e7
t1=Sys.time(); x=analyze.stuff::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
t1=Sys.time(); x=  matrixStats::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
# Note the latter cannot handle a data.frame:
## Not run:
# This would fail:
```

```
matrixStats::colMaxs(   data.frame(a=1:10, b=2))
# This works:
analyze.stuff::colMaxs( data.frame(a=1:10, b=2))

## End(Not run)
```

colMins *Returns the min value of each column of a data.frame or matrix*

Description

Returns minimum value of each column of a data.frame or matrix.

Usage

```
colMins(df, na.rm = TRUE)
```

Arguments

df	data.frame or matrix
na.rm	TRUE by default. Should NA values be removed first

Details

** NOTE: The useful matrixStats package will provide the basis for extended rowMins, rowMax, colMins, colMaxs functions to be made available through this package. Source: Henrik Bengtsson (2015). matrixStats: Methods that Apply to Rows and Columns of a Matrix. R package version 0.13.1-9000.
https://github.com/HenrikBengtsson/matrixStats
Initially, separate functions were written here for those four functions, and the versions here were more flexible and convenient for some purposes, e.g., handling data.frames and different na.rm defaults, but the matrixStats versions are much faster (e.g., by 4x or more). Ideally, this analyze.stuff package would be modified to just extend those functions by providing them methods to handle data.frames, not just matrix class objects, and perhaps provide new or different parameters or defaults, such as defaulting to na.rm=TRUE instead of FALSE, and handling factor class columns in a data.frame. That has not been done yet, so colMaxs() etc. refer to the slower more flexible ones, and the faster matrix-only ones are via matrixStats::colMaxs etc.

** NOTE: max() and min() and matrixStats::colMaxs from matrixStats etc. default to na.rm=FALSE, but this function defaults to na.rm=TRUE because that just seems more frequently useful.

** NOTE: `min` and `max` & this function will handle character elements by coercing all others in the column to character, which can be confusing – e.g., note that min(c(8,10,'txt')) returns '10' not '8' and max returns 'txt' (also see the help for `Comparison`)

If this worked just like max() and min(), cols that are factors would make this fail. max or min of a factor fails, even if as.character() of the factor would return a valid numeric vector. That isn't an issue with a matrix, but a data.frame might have numbers stored as factor. To fix that, this uses `factor.as.numeric` with parameters that try to convert character or factor columns to numeric.

Based on how min and max behave, return Inf or -Inf if no non-missing arguments to min or max respectively. To suppress that warning when using this function, use suppressWarnings(func(x))

Value

vector of numbers with length equal to number of cols in df

See Also

factor.as.numeric rowMaxs rowMins colMaxs colMins count.above pct.above pct.below cols.above.which cols.above.pct

Other functions for max and min of rows and columns: colMaxs, rowMaxs, rowMins

Examples

```
blah <- rbind(NA, data.frame(a=c(0, 0:8), b=c(0.1+(0:9)), c=c(1:10), d=c(rep(NA, 10)),
  e=TRUE, f=factor('factor'), g='words', stringsAsFactors=FALSE) )
cbind(blah, min=rowMins(blah), max=rowMaxs(blah))
rbind(blah, min=colMins(blah), max=colMaxs(blah))
blah <- blah[ , sapply(blah, function(x) is.numeric(x) | is.logical(x)) ]
cbind(blah, min=rowMins(blah), max=rowMaxs(blah),
  mean=rowMeans(blah, na.rm=TRUE), sum=rowSums(blah, na.rm=TRUE))
rbind(blah, min=colMins(blah), max=colMaxs(blah),
  mean=colMeans(blah, na.rm=TRUE), sum=colSums(blah, na.rm=TRUE))
  # ** Actually, matrixStats does this ~4x as quickly,
  # although no practical difference unless large dataset:
  n <- 1e7
t1=Sys.time(); x=analyze.stuff::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
t1=Sys.time(); x=  matrixStats::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
# Note the latter cannot handle a data.frame:
## Not run:
# This would fail:
matrixStats::colMaxs(   data.frame(a=1:10, b=2))
# This works:
analyze.stuff::colMaxs( data.frame(a=1:10, b=2))

## End(Not run)
```

---

cols.above.count *Number of Columns with Value at or above Cutoff*

Description

Find what number of columns have a value at or above some cutoff.

Usage

```
cols.above.count(x, cutoff, or.tied = FALSE, na.rm = TRUE,
  below = FALSE)
```

Arguments

`x`	Data.frame or matrix of numbers to be compared to cutoff value.
`cutoff`	The numeric threshold or cutoff to which numbers are compared. Default is arithmetic mean of row. Usually one number, but can be a vector of same length as number of rows, in which case each row can use a different cutoff.
`or.tied`	Logical. Default is FALSE, which means we check if number in x is greater than the cutoff (>). If TRUE, check if greater than or equal (>=).
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed before result is found. Otherwise result will be NA when a row has an NA value in any column.
`below`	Logical. Default is FALSE. If TRUE, uses > or >= cutoff. If FALSE, uses < or <= cutoff.

Details

For a matrix with a few cols of related data, find what number of columns are at/above (or below) some cutoff. Returns a vector of number indicating how many of the columns are at/above the cutoff. Can be used in identifying places (rows) where some indicator(s) is/are at/above a cutoff, threshold value.

Value

Returns a vector the same size as the number of rows in x.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: `cols.above.pct`, `cols.above.which`, `count.above`, `count.below`, `pct.above`, `pct.below`, `rows.above.count`, `rows.above.pct`, `rows.above.which`, `rows.below.count`, `rows.below.pct`

Examples

```
out <- cols.above.count(x<-data.frame(a=1:10, b=rep(7,10), c=7:16), cutoff=7)
out
out # default is or.tied=FALSE
out <- cols.above.count(data.frame(a=1:10, b=rep(7,10), c=7:16),
  cutoff=7, or.tied=TRUE, below=TRUE)
out
out <- cols.above.count(data.frame(a=1:10, b=rep(7,10), c=7:16) )
 # Compares each number in each row to the row's mean.
out
```

`cols.above.pct` *Percent of Columns with Value at or above Cutoff*

Description

Find what percent of columns have a value at or above some cutoff.

Usage

```
cols.above.pct(x, cutoff, or.tied = FALSE, na.rm = TRUE,
  below = FALSE)
```

Arguments

`x`	Data.frame or matrix of numbers to be compared to cutoff value. Must have more than one row and one column?
`cutoff`	The numeric threshold or cutoff to which numbers are compared. Default is arithmetic mean of row. Usually one number, but can be a vector of same length as number of rows, in which case each row can use a different cutoff.
`or.tied`	Logical. Default is FALSE, which means we check if number in x is greater than the cutoff (>). If TRUE, check if greater than or equal (>=).
`na.rm`	Logical, default TRUE. Should NA values be removed before analysis.
`below`	Logical. Default is FALSE. If TRUE, uses > or >= cutoff. If FALSE, uses < or <= cutoff.

Details

For a matrix with a few cols of related data, find what percent of columns are at/above (or below) some cutoff. Returns a vector of number indicating what percentage of the columns are at/above the cutoff. Can be used in identifying places (rows) where some indicator(s) is/are at/above a cutoff, threshold value.

Value

Returns a vector the same size as the number of rows in x.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

Author(s)

author

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.which, count.above, count.below, pct.above, pct.below, rows.above.count, rows.above.pct, rows.above.which, rows.below.count, rows.below.pct

Examples

```
out <- cols.above.pct(x<-data.frame(a=1:10, b=rep(7,10), c=7:16), cutoff=7)
out
out # default is or.tied=FALSE
out <- cols.above.pct(data.frame(a=1:10, b=rep(7,10), c=7:16),
  cutoff=7, or.tied=TRUE, below=TRUE)
out
out <- cols.above.pct(data.frame(a=1:10, b=rep(7,10), c=7:16) )
 # Compares each number in each row to the row's mean.
out
```

---

cols.above.which *Does each Column have a Value at or above Cutoff(s)*

Description

Flag which cells are at or above some cutoff(s) or mean.

Usage

```
cols.above.which(x, cutoff, or.tied = FALSE, below = FALSE)
```

Arguments

x Data.frame or matrix of numbers to be compared to cutoff value.

cutoff The numeric threshold or cutoff to which numbers are compared. Default is arithmetic mean of row. Usually one number, but can be a vector of same length as number of rows, in which case each row can use a different cutoff.

or.tied Logical. Default is FALSE, which means we check if number in x is greater than the cutoff (>). If TRUE, check if greater than or equal (>=).

below Logical. Default is FALSE. If TRUE, uses > or >= cutoff. If FALSE, uses < or <= cutoff.

Details

For a matrix with a few cols of related data, find which cells are at or above (or below) some cutoff. Returns a logical matrix, with TRUE for each cell that is at or above the cutoff. Can be used in identifying places (rows) where some indicator(s) is or are at or above a cutoff, threshold value.

Value

Returns a logical matrix the same size as x. ** Note this is different than `which` – That function returns the positions of TRUE elements but this returns TRUE or FALSE for all elements.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: `cols.above.count`, `cols.above.pct`, `count.above`, `count.below`, `pct.above`, `pct.below`, `rows.above.count`, `rows.above.pct`, `rows.above.which`, `rows.below.count`, `rows.below.pct`

Examples

```
out <- cols.above.which(x<-data.frame(a=1:10, b=rep(7,10), c=7:16), cutoff=7)
out
out # default is or.tied=FALSE
out <- cols.above.which(data.frame(a=1:10, b=rep(7,10), c=7:16),
  cutoff=7, or.tied=TRUE, below=TRUE)
out
out <- cols.above.which(data.frame(a=1:10, b=rep(7,10), c=7:16) )
 # Compares each number in each row to the row's mean.
out
```

`count.above` *Number or percent of rows (for each col) where value exceeds cutoff(s)*

Description

Count the number or percent of rows (for each col of a data.frame) where the value exceeds some specified cutoff(s)

Usage

```
count.above(df, benchmarks = "mean", benchnames = "cutoff",
  or.tied = FALSE, below = FALSE, wts = 1, na.rm = TRUE)
```

Arguments

`df` Data.frame or matrix, required.

`benchmarks` Default is ’mean’ but otherwise this must be a number or numeric vector of thresholds to compare values to.

`benchnames` Default is ’cutoff’ and this string is used to create colnames for the results, such as above.cutoff.for.field1 `or.tied` Logical, FALSE by default, reporting on those > cutoff. But, if or.tied=TRUE, this reports on those >= cutoff.

`below` Logical, FALSE by default, which counts how many are above cutoff (or tied if or.tied). If TRUE, counts how many are below (or tied with) cutoff.

`wts` Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.

`na.rm` Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.

Details

- If wts is population counts, for example, this gives the COUNT of people (not rows) for whom value in df[,x] exceeds benchmark for each column x
- If below=FALSE by default, reports on those above (or tied with, if or.tied) cutoff. But if below=TRUE, this reports on those below (or tied with, if or.tied) cutoff.
- If df (passed to the function) is a data.frame or matrix, the function returns a vector of length= length(df) or number of cols in matrix.
- If df is just a vector, it is treated like a 1-column data.frame, so the function returns a single value.
- If benchmarks (passed to the function) is a data.frame matching df in dimensions, each value is used as the cutoff for the corresponding cell in df.
- If benchmarks is a vector of length= length(df), each value in benchmarks is the cutoff for the corresponding column in df.
- If benchmarks is a shorter vector, it is recycled. (e.g., a vector of length 2 would use the first benchmark as the cutoff for all odd columns of df, the second for all even columns of df).
- If benchmarks is a single numeric value, it is used as the cutoff value in every comparison for all of df.
- If benchmarks is omitted, the default behavior is to use the arithmetic mean value a column of df as the cutoff for that column of df.
- If benchnames is omitted, the word "cutoff" is used by default (unless benchmarks is also omitted).
- If benchnames is specified but benchmarks is not, the benchmarks default to the column means, so benchnames is ignored and "mean" is used instead.
- If wts is omitted the default is 1 which means no weighting. Just row counts.
- If wts is a vector of length= length(df[,1]) then each row of df uses the corresponding weight and count is sum of wts not count of rows.
- If wts is shorter than that, it is recycled but # of rows in df must be an integer multiple of length(wts).
- NA values in df are not counted and are not in the numerator of pct.above() but the denominator of pct.above() is a count of all rows of df, not just the non-NA ones.

These could be renamed rows.above.count(), rows.above.pct(), rows.above.which() to follow convention of cols.above.count(), cols.above.pct(), cols.above.which() and same using below too, like rows.below.pct() etc. and *** should make param names consistent, like x not df, cutoff(s) not benchmarks?, or.tied not gte but *** cols versions and all should have wts, na.rm, benchmarks as vector not just 1 number, benchnames, params and ** should have a "below" version for each variant

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.below, pct.above, pct.below, rows.above.count, rows.above.pct, rows.above.which, rows.below.count, rows.below.pct

Examples

```
x <- data.frame(a=1:20, b=10, c=c(1:9,100:110))
mywts <- c(rep(1,10), rep(2,10))
mybench <- c(3,100,10)
mynames <- c("HI","USavg","HealthStandard")

count.above(x, 0, wts=mywts)
count.above(x, 100, wts=mywts)
count.above(x, 10, wts=mywts)
count.above(x, mybench, wts=mywts)
cbind(count= count.above(x, mybench, mynames, wts=mywts))
cbind(pct= pct.above(x, benchmarks=mybench, benchnames=mynames, wts=mywts) )
cbind(
  count= count.above(x, mybench, mynames, wts=mywts),
  pct= pct.above(x, benchmarks=mybench, benchnames=mynames, wts=mywts) )
cbind(stat= pct.above(as.matrix(x), mybench, mynames, wts=mywts) )
cbind(stat= pct.above(1:100, 98 , wts=mywts))
 # If only a single vector is passed,
 # not a data.frame "Warning: df is a vector... converting to data.frame"

# to find how many PLACES are at/above the 95th population-weighted percentile
#  (won't be exactly 5% of places, just 5% of people):
mybench2 <- sapply(x, function(z) Hmisc::wtd.quantile(z, mywts, probs=0.95, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('pop.95th.', names(x), sep=''), wts=1 )
# to find how many PLACES are at/above the MEDIAN pop-wtd place
#  (won't be exactly half of places, just half of people):
mybench2 <- sapply(x, function(z) Hmisc::wtd.quantile(z, mywts, probs=0.50, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('pop.median.', names(x), sep=''), wts=1 )

# to find how many PEOPLE are at/above the 95th percentile place
#  (won't be exactly 5% of people, just 5% of places):
mybench2 <- sapply(x, function(z) quantile(z, probs=0.95, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('95th.', names(x), sep=''), wts=mywts )
#
## Not run:
# to find how many PEOPLE are at/above the MEDIAN place
#  (won't be exactly 50% of people, just 50% of places):
mybench2 <- sapply(x, function(z) quantile(z, probs=0.50, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('median.', names(x), sep=''), wts=mywts )
##not run## cbind( pct.above(1:100, wts=mywts) )
# That does not recycle weights in this situation of a single vector argument
count.above(data.frame(a=c(1:10, NA)), 2, wts=mywts)   # does not work if NA values
cbind( pct.above(data.frame(a=c(1:10, NA)), 0 , wts=mywts))
  # Gives "Error: wts must be a vector whose length is a factor of # rows in df,
  # so length(df[,1]) is an integer multiple of length(wts) "
pct.above(data.frame(a=c(NA, NA, NA)), 3, wts=mywts)
  # Gives "Error - df is a single NA value or single column with only NA values"
count.above(x, c(3,1), wts=mywts) # 3,1 is recycled as 3,1,3 since x has 3 cols
pct.above(x, benchnames=mynames, wts=mywts)
  # ignores names since default benchmarks are column means

## End(Not run)
```

---

count.below *Number or percent of rows (for each col) where value is below cutoff(s)*

Description

Count the number or percent of rows (for each col of a data.frame) where the value is below some specified cutoff(s)

Usage

```
count.below(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = TRUE, or.tied = FALSE, below = TRUE, wts = 1)
```

Arguments

`df`	Data.frame or matrix, required.
`benchmarks`	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
`benchnames`	Default is 'cutoff' and this string is used to create colnames for the results
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.
`or.tied`	Logical, FALSE by default, reporting on those < cutoff. But, if or.tied=TRUE, this reports on those <= cutoff.
`below`	Logical, TRUE by default, which counts how many are below cutoff (or tied if or.tied). If FALSE, counts how many are above (or tied with) cutoff.
`wts`	Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.

Details

See `count.above` for details, for which this is a wrapper.

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: `cols.above.count`, `cols.above.pct`, `cols.above.which`, `count.above`, `pct.above`, `pct.below`, `rows.above.count`, `rows.above.pct`, `rows.above.which`, `rows.below.count`, `rows.below.pct`

`count.words` *Word Frequency in a Text File*

Description

Simple way to count how many times each word appears in a text file.

Usage

```
count.words(file, wordclump = 1, ignore.case = TRUE, stopwords = "",
  string, numbers.keep = TRUE, ...)
```

Arguments

`file`	Character string filename, with or without path, for text file to be analyzed. Words assumed to be separated by spaces.
`wordclump`	number of words per clump, so if wordclump=2, it counts how often each 2-word phrase appears.
`ignore.case`	Logical, default TRUE which means not case-sensitive.
`stopwords`	Vector of words to ignore and not count. Default is none, optional.
`string`	A single character string containing text to analyze. Not yet implemented.
`numbers.keep`	Not yet implemented. Would ignore numbers.
`...`	Any other parameters used by `scan` may be passed through. See `http://stat.ethz.ch/R-manual/R-devel/library/base/html/scan.html`

Value

Returns a data.frame with term (term) and frequencies (freq) sorted by frequency, showing the number of times a given word appears in the file. The rownames are also the words found.

Examples

```
## Not run:
  counts <- count.words('speech.txt'); tail(counts, 15)
counts <- count.words('speech.txt', ignore.case=FALSE); head(counts[order(counts$term), ], 15)
counts <- count.words('speech.txt', stopwords=c('The', 'the', 'And', 'and', 'A', 'a'))
tail(counts, 15)
counts <- count.words('speech.txt', 3); tail(counts, 30)
#
counts['the', ]
counts[c('the', 'and', 'notfoundxxxxx'), ] # works only if you are sure all are found
counts[rownames(counts) %in% c('the', 'and', 'notfoundxxxxx'), ]
  # that works even if specified word wasn't found
counts[counts$term %in% c('the', 'and', 'notfoundxxxxx'), ]
  # that works even if specified word wasn't found
counts <- count.words('C:/mypath/speech.txt')
counts <- count.words('speech.txt', sep='.')
  # that is for whole sentences (sort of - splits up at decimal places as well)

## End(Not run)
```

dir2 *Directory listing using wildcard search*

Description

Function to let you see directory listing using wildcard search syntax like '*.R'

Usage

```
dir2(x, ignore.case = TRUE, ...)
```

Arguments

`x`	Query string that can use wildcards to search directory
`ignore.case`	Logical, TRUE by default, optional. If FALSE, then this is case-sensitive.
`...`	Optional other parameters passed to `dir`

Value

A directory listing.

See Also

`dirdirs` `dirr`

Examples

```
dir2('*.txt')
dir2('*.txt', path='~')
dir2()   # shows only files, not folders, if no x is specified.
dir2(path='~')  # shows only files, not folders, if no x is specified.
```

dirdirs *Directory listing of R-related files/folders*

Description

Function to let you see directory listing of files/folders ending in r, R, or RData

Usage

```
dirdirs(path = ".", recursive = FALSE, ...)
```

Arguments

`path`	Path as character string, optional. Default is current working directory.
`recursive`	Logical value, optional, FALSE by default. Should subdirectories be shown.
`...`	Optional other parameters passed to `list.dirs`

Value

A directory listing

See Also

`dir2` `dirr`

Examples

```
dirdirs()
```

dirr *Directory listing of R-related files/folders*

Description

Function to let you see directory listing of files/folders ending in r, R, or RData

Usage

```
dirr(path = ".", ignore.case = TRUE, ...)
```

Arguments

`path`	A file path string, optional, default is current working directory.
`ignore.case`	Logical, TRUE by default, optional. If FALSE, then this is case-sensitive.
`...`	Optional other parameters passed to `dir`

Value

A directory listing.

See Also

`dir2` `dirdirs`

`download.files` *Try to download one or more files*

Description

Attempts to download files, given name(s) from specified url, saving them in specified folder. Just a wrapper that Uses `download.file` which only downloads a single file.

Usage

```
download.files(url, files, todir, silent = FALSE, overwrite = FALSE)
```

Arguments

`url`	The url of folder with files to download, as character string
`files`	A character vector of file names.
`todir`	The folder where downloaded files will be placed, as a character string.
`silent`	Logical, optional, FALSE by default. Prints a message using cat() if TRUE.
`overwrite`	Optional, logical, FALSE by default. If FALSE, checks to see if file already exists in local folder and does not download if already exists. But note that may cause problems if zero size file exists already due to earlier failed download.

Value

Returns vector of numbers, each being 1 or 0 or 2 to signify success or failure or no attempt because file already seems to exist locally.

See Also

`download.file`

`expand.gridMatrix` *Similar to expand.grid, but returns a matrix not data.frame*

Description

This function is similar to `expand.grid`, in the sense that it returns a matrix that has 2 columns, one for each input, and one row per combination, cycling through the first field first. It differs from expand.grid in that this returns a matrix not data.frame, only accepts two parameters creating two columns, for now, and lacks the other parameters of expand.grid

Usage

```
expand.gridMatrix(x, y)
```

Arguments

`x`	required vector
`y`	required vector

Value

This function returns a matrix and tries to assign colnames based on the two input parameters. If they are variables, it uses those names as colnames. Otherwise it uses "x" and "y" as colnames.

See Also

`expand.grid`

Examples

```
expand.gridMatrix(99:103, 1:2)
zz <- 1:10; top <- 1:2
expand.gridMatrix(zz, top)
```

`factor.as.numeric` *Handle Numbers Stored as Factors*

Description

Try to convert back to numeric any numbers stored as factors, e.g., in a data.frame that did not use stringsAsFactors.

Usage

```
factor.as.numeric(x, stringsAsFactors = TRUE)
```

Arguments

`x`	Data.frame or vector, required. (If matrix, it is returned unaltered as a matrix).
`stringsAsFactors`	Logical, TRUE by default, in which case a factor vector or col that has character elements, and thus cannot be coerced to numeric without creating NA values, is left as a factor. If FALSE, such a vector or col is converted to character class.

Details

Uses as.numeric(as.character(x)) on the factor cols or vector, but if there are both numbers and characters, it leaves it as factor, not numeric (which would put NA values in place of character elements). NOTE: ** Not optimized for speed yet, so it is slow.

Value

Returns a data.frame or vector, same shape as x (or matrix if given a matrix). Any column that was integer or numeric is returned as numeric.
Any character column or vector is returned as numeric if it could be coerced to numeric without creating any NA values because it has only numbers stored as text.
Logical is returned as logical.
When stringsAsFactors is TRUE, factor is returned as factor if it has any text that cannot be coerced to non-NA numeric.
When stringsAsFactors is FALSE, factor is returned as character if it has any text that cannot be coerced to non-NA numeric.

See Also

`as.vector`, `factor`, `data.table`, `matrix`

Examples

```
a=factor(c(2,3,5)); b=factor(c('2', '3', '5')); c=factor(c('two','three','five'))
d=factor(c(2,'3','5')); e=factor(c(2,'three','five')); f=factor(c('2','three','5'))
g=factor(c(2,'3','five')); h=factor(c(NA, 3, 'five')); i=1:3;
j=rep('nonfactor',3); k=c(1,2,'text'); l=c(TRUE, FALSE, TRUE); m=c('2','3','5')
x=data.frame(a,b,c,d,e,f,g,h,i,j,k,l,m, stringsAsFactors=FALSE)
cat('\n')
cat('\n'); x; cat('\n'); cat('\n')
z=factor.as.numeric(x)
cat('\n'); z
cat('\n'); str(x)
cat('\n'); str(z);
cat('\n'); str( factor.as.numeric(x, stringsAsFactors=FALSE) )
for (i in 1:length(x)) {out<-factor.as.numeric(x[,i]);cat(class(out), out,'\n') }
for (i in 1:length(x)) {
  out<-factor.as.numeric(x[,i], stringsAsFactors = FALSE)
  cat(class(out), out,'\n')
}
```

`findArgs` *Get the arguments of a function*

Description

Get the arguments of a function as shown in help for `grep`

Usage

```
findArgs(env, pattern)
```

Arguments

`env`	e.g., 'package:analyze.stuff'
`pattern`	search query as regular expression

Value

arguments

Examples

```
findArgs("package:base", "warn")
```

`formatcomma` *Print numbers with commas and 0-2 decimal places*

Description

Wrapper for `format` making it easier to use with my typical settings

Usage

```
formatcomma(x, big.m = ",", nsmall = 2, drop0trailing = FALSE, ...)
```

Arguments

`x`	Numeric vector
`big.m`	Default is a comma at every three digits (1,000,000)
`nsmall`	Default is 2 digits after the decimal (3.14)
`drop0trailing`	Default is FALSE, which means zeroes after the last nonzero digit in the decimal portion are still shown (TRUE would mean 1.2 is shown, not 1.20)
`...`	other parameters passed to `format`

Value

Same as `format` but with some preselected defaults

See Also

`format`

Examples

```
cbind(mynum=formatcomma(c(1452345, 1.2, 4234.12345)))
cbind(mynum=formatcomma(c(1452345, 1.2, 4234.12345), drop0trailing=FALSE))
```

geomean *Geometric mean*

Description

Returns the geometric mean of a vector of numbers, which is the nth root of their product.

Usage

```
geomean(x, na.rm = FALSE)
```

Arguments

x	Vector of numbers, required.
na.rm	Logical value, optional, FALSE by default. If FALSE, result is NA if any of the values in x is NA. If TRUE, remove the NA values first.

Details

The geomean is one type of average, used in working with lognormal distributions, for example. Is not as strongly influenced by extreme outliers as the arithmetic mean. See http://en.wikipedia.org/wiki/Geometric_mean for many applications.

Value

Returns a single number that is the geometric mean of the numbers in x.

See Also

harmean mean rms

Examples

```
geomean(c(4,9)) # is the square root of 4 * 9
```

get.os *Windows or Mac?*

Description

This function returns a character string "win" or "mac" depending on which operating system is being used (that's all it does right now)

Usage

```
get.os()
```

Value

Returns "win" or "mac" currently.

See Also

`os` which is more powerful and easier to type

`harmean` *Harmonic mean*

Description

Returns the harmonic mean of a vector of numbers.

Usage

```
harmean(x, na.rm = FALSE)
```

Arguments

x	Vector of numbers, required.
na.rm	Logical value, optional, FALSE by default. If FALSE, result is NA if any of the values in x is NA. If TRUE, remove the NA values first.

Details

The harmonic mean is one type of average. It is the reciprocal of the arithmetic mean of the reciprocals. See `http://en.wikipedia.org/wiki/Harmonic_mean` for many applications of the harmonic mean.

Value

Returns a single number

See Also

`geomean` `mean` `rms`

Examples

```
harmean(c(1,2,4))
```

installrequired *Require a list of packages, downloading and installing if necessary*

Description

Convenient way to specify packages to attach, and install any that are not already installed. It only installs a package if that package is not already available locally.

Usage

```
installrequired(x, github, gitlatest = FALSE)
```

Arguments

`x` vector of package names e.g., c("Hmisc", "data.table")

`github` optional vector of user slash package names e.g., "rstudio/shiny" but those can just be in x now. If github = 'ej' it installs several specific ones from github (also see `http://www.ejanalysis.com`):

- 'rstudio/shiny'
- 'ejanalysis/analyze.stuff'
- 'ejanalysis/ejanalysis'
- 'ejanalysis/proxistat'
- 'ejanalysis/ejscreen'
- 'ejanalysis/ACSdownload'
- 'ejanalysis/countyhealthrankings'
- 'ejanalysis/UScensus2010blocks'

`gitlatest` Optional logical, default is FALSE which means not downloaded from github if pkg of that name is already installed. If TRUE, download latest from github even if already installed.

Details

Uses `require` and if necessary uses `install.packages` or `install_github` If no parameters, prints an example.

Examples

```
  ## Not run:
  installrequired('stringr')
  installrequired('rstudio/shiny')
  #
  installrequired(c('Hmisc' , 'ejanalysis/analyze.stuff'))
  # or
  installrequired('ej') # for several specific ones used in e

## End(Not run)
```

intersperse — *Intersperse the elements of a vector, mixing 2d half of the list in with the 1st half*

Description

This function will take a vector and split it in half (it must have an even # of elements) and then will intersperse the elements, so for example, if the vector's starting order is 1,2,3, 4,5,6 the function returns the vector ordered as 1,4, 2,5, 3,6

Usage

```
intersperse(x)
```

Arguments

x	A vector with an even number of elements, required, character or numeric works.

Details

This is useful for example in reformatting a data.frame of Census data where the first n fields are estimates and the next n fields are margin of error values corresponding to those estimates. This function applied to the field names can reorder them to pair each estimate followed by its MOE.

Value

Returns a vector that contains all the elements of the original, but reordered.

Examples

```
mydf <- data.frame(e1=101:120, e2=102:121, e3=111:130,
  m1=(101:120)*0.01, m2=(102:121)*0.01, m3=(111:130)*0.01)
mydf
mydf <- mydf[ , intersperse(names(mydf))]
mydf
```

lead.zeroes — *Add leading zeroes as needed*

Description

Returns the vector that was supplied, but with leading zeroes added where needed to make all elements have specified number of characters.

Usage

```
lead.zeroes(fips, length.desired)
```

Arguments

`fips`	Character vector, which can be FIPS codes or other data. Required.
`length.desired`	A single numeric value (recycled), or vector of numbers, required, specifying how many characters long each returned string should be.

Details

This function can be useful in working with Census data where FIPS codes are often used. Moving data to and from a spreadsheet can remove leading zeroes that may be necessary for proper data management. This can apply to e.g., FIPS code for a block, block group, tract, county, or state.
Note: Number of digits in FIPS codes, assuming leading zeroes are there:
state 2 (2 cumulative)
county 3 (5 cum)
tract 6 (11 cum) (note 11 digits is ambiguous if not sure leading zero is there)
block group 1 (12 cum) (note 12 digits is ambiguous if not sure leading zero is there)
block 1 (13 cum)

Value

Returns a vector of same length as input parameter

Examples

```
lead.zeroes(c('234','01234','3'), 5)
```

`length2` *Length of a list with or without NA values*

Description

Replacement for length(). Finds count of items like length(), but if set na.rm=TRUE then it doesn't count the items that are NA

Usage

```
length2(x, na.rm = FALSE)
```

Arguments

`x`	A vector, required.
`na.rm`	Logical value, optional, FALSE by default. Should NA values be left out of the count?

Value

Returns a single number.

Examples

```
length2(c(1,2,3,NA))
length2(c(1,2,3,NA), na.rm=TRUE)
```

`linefit` *Add fit lines to a scatter plot*

Description

Convenient wrapper for lowess(), lm(), and coef(line())

Usage

```
linefit(x, y, type = "b", cex = 4, show.lowess = TRUE,
  show.lm = TRUE, show.line = TRUE)
```

Arguments

`x`	x values, required
`y`	y values, required
`type`	passed through to lines() for the lowess
`cex`	scaling for lowess
`show.lowess`	Logical value, optional, TRUE by default. Defines if lowess is shown
`show.lm`	Logical value, optional, TRUE by default. Defines if lm line is shown
`show.line`	Logical value, optional, TRUE by default. Defines if should show abline(coef(line(x,y)))

Details

This function adds lines to a scatter plot, using lines(`lowess`(x,y)), abline(`lm`(y~x)), and abline(`coef`(`line`(x,y))) DOESN'T SEEM TO WORK IF log='xy' was used in original plot() NOTE: coef(line()) and lm() give different results

Value

Provides a plot just as a side effect

Examples

```
## Not run:
    # see
#?lm  or  ?aov   or  ?glm
# ?line
require(graphics)
plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))
# ?predict
# ?lowess
# ?scatterplot
#The scatterplot( ) function in the car package offers many enhanced features, including
#fit lines, marginal box plots, conditioning on a factor, and interactive point identification.
#Each of these features is optional.
# Enhanced Scatterplot of MPG vs. Weight
# by Number of Car Cylinders
 library(car)
 scatterplot(mpg ~ wt | cyl, data=mtcars,
             xlab="Weight of Car", ylab="Miles Per Gallon",
             main="Enhanced Scatter Plot",
             labels=row.names(mtcars))

## End(Not run)
```

linesofcode *Counts lines of source code in .R files of package source*

Description

This is just a way to summarize how many lines of code appear to be in the .R files in the folder that a package is built from.

Usage

```
linesofcode(folder = getwd(), packages, recursive = TRUE,
  sums = FALSE)
```

Arguments

folder	Default is current working directory. **This is NOT the base path of the package itself! It is the full path of the folder within which is a folder for each package of interest. For example, folder= '~/Documents/R PACKAGES' works but folder= '~/Documents/R PACKAGES/mypkg' does not see the package called mypkg
packages	Default is all found in folder. Can specify a subset of those by name as character vector.
recursive	Default is TRUE, searches subfolders within specified folder.
sums	Default is FALSE, but if TRUE it returns the count of .R files and lines of code for each package found.

Value

Returns a data.frame of results, with details depending on sums parameter. Also prints summary info if sums=FALSE, and returns detailed info.

Examples

```
 ## Not run:
 linesofcode(folder= '~/Documents/R PACKAGES', packages=c('analyze.stuff', 'proxistat') )
 x <- linesofcode(folder= '~/Documents/R PACKAGES')
 x[order(x$code), c('filename', 'package', 'code')]

## End(Not run)
```

`logposneg` *log10(x) if positive, 0 if 0, -log10(-x) if negative*

Description

Function that transforms a vector of numbers x into log10(x) if positive, 0 if 0, -log10(-x) if negative, useful for graphing something on a log scale when it has negative values. This log scale expands outward from zero in both directions.

Usage

```
logposneg(x)
```

Arguments

`x`	numeric vector, required

Value

A numeric vector of same length as x

`mem` *See what is using up memory*

Description

See a list of the largest objects in memory, and how much RAM they are using up Uses `object.size` to return info on memory consumption for largest n objects

Usage

```
mem(n = 10)
```

Arguments

`n`	Numeric, default is 10. How many objects to show (e.g., top 10)

Value

Results in printing a list of objects and their sizes

Examples

```
## Not run:
 mem()
mem(15)

# draw pie chart
pie(object.sizes(), main="Memory usage by object")

# draw bar plot
barplot(object.sizes(),
        main="Memory usage by object", ylab="Bytes", xlab="Variable name",
        col=heat.colors(length(object.sizes())))

# draw dot chart
dotchart(object.sizes(), main="Memory usage by object", xlab="Bytes")

###################################
# memory.size() and memory.limit() and object.sizes() comparison:
###################################

# memory.size() to print aggregate memory usage statistics

print(paste('R is using', memory.size(), 'MB out of limit', memory.limit(), 'MB'))

# object.sizes() to see memory total used by objects:

# NOTE: THIS DOES NOT MATCH TOTAL GIVEN BY memory.size();
#  it is only about half as much in the case I tried:
sum(as.numeric(object.sizes()))
# same, in MEGABYTES:
unclass(sum(as.numeric(object.sizes())))/1e6
# print to console in table format
object.sizes()
# see a list of the top few variables:
head(cbind(object.sizes()))

## End(Not run)
```

`minNonzero` *Find minimum non-zero number(s) - BUT EXCLUDES COLUMNS THAT ARE NOT NUMERIC OR ARE FACTOR***

Description

Returns minimum nonzero numbers in vector, matrix, or data.frame

Usage

```
minNonzero(mydf)
```

Arguments

`mydf` Required. Must be vector, matrix, or data.frame

Value

A number or vector of numbers

Examples

```
minNonzero(-1:6)
minNonzero(data.frame(a=0:10, b=1:11, c=c(0,1:9,NA), d='text', stringsAsFactors = FALSE))
minNonzero(data.frame(a=0:10, b=1:11, c=c(0,1:9,NA), d='3', stringsAsFactors = TRUE))
```

---

na.check *Basic info on each col of data.frame*

Description

Returns basic information on each field in a data.frame, like count of rows that are zero, negative, NA, infinite, etc.

Slow - work in progress Leaves out logical, complex?, character, etc. cols

Usage

```
na.check(df, min.text = FALSE)
```

Arguments

df	Matrix or data.frame to examine. Cannot be a single vector currently.
min.text	Logical, optional, defaults to FALSE. If TRUE, tries to find minimum of numbers stored as text? Slows it down.

Value

Returns a vector of results, one per col of df

See Also

signTabulate in **matrixStats** minNonzero and experimental variations on na.check: na.check na.check2

Examples

```
## Not run:
system.time(x= na.check(data.frame(a=-1:1e6, b='text', c=c(NA, 1, 2)), min.text=FALSE) )
system.time(x= na.check2(data.frame(a=-1:1e6, b='text', c=c(NA, 1, 2)), min.text=TRUE) )
na.check(data.frame(a=-1:10, b='text', c=c(NA, 1, 2)))
na.check2(data.frame(a=-1:10, b='text', c=c(NA, 1, 2)))

## End(Not run)
```

na.check2 *Basic info on each col of data.frame - testing faster way, but returns text*

Description

Returns basic information on each field in a data.frame, like count of rows that are zero, negative, NA, infinite, etc.

Slow - work in progress Leaves out logical, complex?, character, etc. cols this version fails to handle fields that are factor class!?

Usage

```
na.check2(df)
```

Arguments

df	Matrix or data.frame to examine. Cannot be a single vector currently.

Value

Returns a vector of results, one per col of df

See Also

signTabulate in **matrixStats** minNonzero and experimental variations on na.check: na.check na.check2

Examples

```
## Not run:
system.time(x= na.check(data.frame(a=-1:1e6, b='text', c=c(NA, 1, 2)), min.text=FALSE) )
system.time(x= na.check2(data.frame(a=-1:1e6, b='text', c=c(NA, 1, 2)), min.text=TRUE) )
na.check(data.frame(a=-1:10, b='text', c=c(NA, 1, 2)))
na.check2(data.frame(a=-1:10, b='text', c=c(NA, 1, 2)))

## End(Not run)
```

names2 *Print names(data.frame) commented out for easy pasting into code*

Description

Uses cat() to print names of data.frame, but in a column with # before each. Make it convenient to copy/paste into .R code as comments

Usage

```
names2(x)
```

Arguments

x	Data.frame, required

Value

Prints results

---

normalized *Normalize raw scores as ratio of score to wtd mean*

Description

Provides a data.frame that takes the matrix or data.frame and finds the weighted mean of each column and then divides each column of values by the column's weighted mean.

Usage

```
normalized(df, wts = NULL, na.rm = TRUE)
```

Arguments

df	numeric Data.frame of one or more columns of values to be normalized, or matrix or vector to be coerced to data.frame
wts	numeric Weights to use when computing weighted mean of given column, one weight per row in df (default=1) or per element of vector df. If omitted, default is unweighted mean.
na.rm	logical Whether to exclude rows where weight or value or both = NA.

Details

Uses scale

Value

matrix same size as df, but with all values in given column divided by weighted mean of that column

See Also

scale

Examples

```
## Not run:
 mydf_norm <- tbd
 ###

## End(Not run)
```

os *Windows, Mac, or other Unix?*

Description

Answers query about whether operating system is a certain type, or just reports type of operating system.

Usage

```
os(x)
```

Arguments

x	Optional query, must be among these: 'mac', 'apple', 'osx', 'darwin', 'win', 'windows', 'pc', 'microsoft', 'unix'

Value

If queried, returns TRUE or FALSE (or NA if query not recognized). If no query, returns 'win', 'mac', or 'unix'

See Also

get.os which is a bit more limited

Examples

```
os()
if (os('mac')) {cat("Hi, I'm a Mac\n")} else {cat('I am not a Mac\n')}
```

overlaps *Counts for Intersect, Union, etc. for Two Sets*

Description

This is just a convenient way to compare two sets (vectors) that overlap, to count how many are in each set, how many are in a not b, in b not a, in both, etc.

Usage

```
overlaps(a, b, values = FALSE)
```

Arguments

a	Required vector, such as list of FIPS character codes.
b	Required vector
values	Default is FALSE. If TRUE, output is logical data.frame with union of only the unique elements as rownames, indicating which of those meet each criterion.

Value

Returns a data.frame of counts by default, formatted for viewing as a small table. If values = TRUE, returns a larger data.frame (see `values` parameter). See examples.

See Also

`setdiff2`, `setops`, `intersectDiagram`

Examples

```
 overlaps( c('Selectric 251','Selectric 245'),
  c('Selectric 245','Selectric 255','Selectric 255'))
 overlaps( c('Selectric 251','Selectric 245'),
  c('Selectric 245','Selectric 255','Selectric 255'), values = TRUE)
 overlaps(state.abb[1:3], state.abb[3:4])
 colSums( overlaps(state.abb[1:3], state.abb[3:4], values = TRUE) )
 colSums( overlaps(state.abb[1:3], state.abb[c(3:4,4,4,4,4,4)], values = TRUE) )
 overlaps(state.abb[1:3], state.abb[c(3:4,4,4,4,4,4)])
 overlaps(state.abb[1:3], state.abb[3:4], values = TRUE)
 ## Not run:
  overlaps(ejanalysis::get.state.info()$ST, state.abb)
  data(fips.state, package='acs')
  overlaps(lead.zeroes(fips.state$STATE,2), ejanalysis::get.state.info()$FIPS.ST)
  data(fips.county, package='acs')
  overlaps(ejanalysis::get.county.info()$FIPS.COUNTY,
    paste(analyze.stuff::lead.zeroes(fips.county$State.ANSI,2),
    analyze.stuff::lead.zeroes(fips.county$County.ANSI,3), sep=''))

  colSums( overlaps(ejanalysis::get.state.info()$ST, c(999, state.abb), values = TRUE) [ , 2:8])


## End(Not run)
```

`pause` *Pause and wait specified number of seconds*

Description

Do nothing until time is up. Pause for some reason, wait for a download, etc.

Usage

```
pause(seconds = 1)
```

Arguments

`seconds`	Time in seconds. Optional, default is 1 second.

Details

The word pause is easier to remember than Sys.sleep, and Sys.sleep does not work on all systems apparently.

Value

No value is returned.

See Also

Sys.sleep

---

pct.above *Number or percent of rows (for each col) where value exceeds cutoff(s)*

Description

Count the number or percent of rows (for each col of a data.frame) where the value exceeds some specified cutoff(s)

Usage

```
pct.above(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = FALSE, or.tied = FALSE, below = FALSE, wts = 1,
  of.what = "all")
```

Arguments

df	Data.frame or matrix, required.
benchmarks	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
benchnames	Default is 'cutoff' and this string is used to create colnames for the results, such as above.cutoff.for.field1
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed before value is found. Otherwise result will be NA when any NA is in a col.
or.tied	Logical, FALSE by default, reporting on those > cutoff. But, if or.tied=TRUE, this reports on those >= cutoff.
below	Logical, FALSE by default, which counts how many are above cutoff (or tied if or.tied). If TRUE, counts how many are below (or tied with) cutoff.
wts	Number or vector, default is 1. Length must be a factor of # rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.
of.what	Optional, character, 'all' by default, defines xxx as the text used in "pct.above.xxx" (or below) for fieldnames in results

Details

below=FALSE by default, reports on those above (or tied with, if or.tied) cutoff. But if below=TRUE, this reports on those below (or tied with, if or.tied) cutoff.

- If df (passed to the function) is a data.frame or matrix, the function returns a vector of length= length(df) or number of cols in matrix.

- If df is just a vector, it is treated like a 1-column data.frame, so the function returns a single value.
- If benchmarks (passed to the function) is a data.frame matching df in dimensions, each value is used as the cutoff for the corresponding cell in df.
- If benchmarks is a vector of length= length(df), each value in benchmarks is the cutoff for the corresponding column in df.
- If benchmarks is a shorter vector, it is recycled. (e.g., a vector of length 2 would use the first benchmark as the cutoff for all odd columns of df, the second for all even columns of df).
- If benchmarks is a single numeric value, it is used as the cutoff value in every comparison for all of df.
- If benchmarks is omitted, the default behavior is to use the arithmetic mean value a column of df as the cutoff for that column of df.
- If benchnames is omitted, the word "cutoff" is used by default (unless benchmarks is also omitted).
- If benchnames is specified but benchmarks is not, the benchmarks default to the column means, so benchnames is ignored and "mean" is used instead.
- If wts is omitted the default is 1 which means no weighting. Just row counts.
- If wts is a vector of length= length(df[,1]) then each row of df uses the corresponding weight and count is sum of wts not count of rows.
- If wts is shorter than that, it is recycled but # of rows in df must be an integer multiple of length(wts).

NA values in df are not counted and are not in the numerator of pct.above() but the denominator of pct.above() is a count of all rows of df, not just the non-NA ones. These could be renamed rows.above.count(), rows.above.pct(), rows.above.which() to follow convention of cols.above.count(), cols.above.pct(), cols.above.which() and same using below too, like rows.below.pct() etc. and *** should make param names consistent, like x not df, cutoff(s) not benchmarks?, or.tied not gte but *** cols versions and all should have wts, na.rm, benchmarks as vector not just 1 number, benchnames, params and ** should have a "below" version for each variant

Note Hmisc::wtd.mean is not exactly same as stats::weighted.mean since na.rm defaults differ
Hmisc::wtd.mean(x, weights=NULL, normwt="ignored", na.rm = TRUE) # Note na.rm defaults differ.
weighted.mean(x, w, ..., na.rm = FALSE)

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.below, rows.above.count, rows.above.pct, rows.above.which, rows.below.count, rows.below.pct

Examples

```
x <- data.frame(a=1:20, b=10, c=c(1:9,100:110))
mywts <- c(rep(1,10), rep(2,10))
mybench <- c(3,100,10)
mynames <- c("HI","USavg","HealthStandard")

count.above(x, 0, wts=mywts)
count.above(x, 100, wts=mywts)
count.above(x, 10, wts=mywts)
count.above(x, mybench, wts=mywts)
cbind(count= count.above(x, mybench, mynames, wts=mywts))
cbind(pct= pct.above(x, benchmarks=mybench, benchnames=mynames, wts=mywts) )
cbind(
  count= count.above(x, mybench, mynames, wts=mywts),
  pct= pct.above(x, benchmarks=mybench, benchnames=mynames, wts=mywts) )
cbind(stat= pct.above(as.matrix(x), mybench, mynames, wts=mywts) )
cbind(stat= pct.above(1:100, 98 , wts=mywts))
# If only a single vector is passed, not a data.frame
  #"Warning: df is a vector... converting to data.frame"

# to find how many PLACES are at/above the 95th population-weighted percentile
#  (won't be exactly 5% of places, just 5% of people):
mybench2 <- sapply(x, function(z) Hmisc::wtd.quantile(z, mywts, probs=0.95, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('pop.95th.', names(x), sep=''), wts=1 )
## Not run:
# to find how many PLACES are at/above the MEDIAN pop-wtd place
#  (won't be exactly half of places, just half of people):
mybench2 <- sapply(x, function(z) Hmisc::wtd.quantile(z, mywts, probs=0.50, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('pop.median.', names(x), sep=''), wts=1 )

# to find how many PEOPLE are at/above the 95th percentile place
#  (won't be exactly 5% of people, just 5% of places):
mybench2 <- sapply(x, function(z) quantile(z, probs=0.95, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('95th.', names(x), sep=''), wts=mywts )
# to find how many PEOPLE are at/above the MEDIAN place
#  (won't be exactly 50% of people, just 50% of places):
mybench2 <- sapply(x, function(z) quantile(z, probs=0.50, na.rm=TRUE))
count.above(x, benchmarks=mybench2, benchnames=paste('median.', names(x), sep=''), wts=mywts)

cbind( pct.above(1:100, wts=mywts) )
# that does not recycle weights in this situation of a single vector argument
count.above(data.frame(a=c(1:10, NA)), 2, wts=mywts)   # does not work if NA values
cbind( pct.above(data.frame(a=c(1:10, NA)), 0 , wts=mywts))
# Gives "Error: wts must be a vector whose length is a factor of # rows in df,
# so length(df[,1]) is an integer multiple of length(wts) "
pct.above(data.frame(a=c(NA, NA, NA)), 3, wts=mywts)
# Gives "Error - df is a single NA value or single column with only NA values"
count.above(x, c(3,1), wts=mywts) # 3,1 is recycled as 3,1,3 since x has 3 cols
pct.above(x, benchnames=mynames, wts=mywts)
# that ignores names since default benchmarks are column means

## End(Not run)
```

pct.below *Number or percent of rows (for each col) where value is below cutoff(s)*

Description

Count the number or percent of rows (for each col of a data.frame) where the value is below some specified cutoff(s)

Usage

```
pct.below(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = FALSE, or.tied = FALSE, below = TRUE, wts = 1,
  of.what = "all")
```

Arguments

`df`	Data.frame or matrix, required.
`benchmarks`	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
`benchnames`	Default is 'cutoff' and this string is used to create colnames for the results
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.
`or.tied`	Logical, FALSE by default, reporting on those < cutoff. But, if or.tied=TRUE, this reports on those <= cutoff.
`below`	Logical, TRUE by default, which counts how many are below cutoff (or tied if or.tied). If FALSE, counts how many are above (or tied with) cutoff.
`wts`	Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.
`of.what`	Optional, character, 'all' by default, defines xxx as the text used in "pct.above.xxx" (or below) for fieldnames in results

Details

See `pct.above` for details, for which this is a wrapper.

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.above, rows.above.count, rows.above.pct, rows.above.which, rows.below.count, rows.below.pct

pctiles *Show the rounded values at 100 percentiles*

Description

Get a quick look at a distribution by seeing the 100 values that are the percentiles 1-100

Usage

```
pctiles(x, probs = (1:100)/100, na.rm = TRUE, digits = 3)
```

Arguments

x	Required numeric vector of values whose distribution you want to look at.
probs	Optional vector of fractions specifying percentiles. (1:100)/100 by default.
na.rm	TRUE by default, specifies if NA values should be removed first.
digits	Number, 3 by default, how many decimal places to round to

Details

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES !# Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. This is how should be calculating percentiles as of 2/2013. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################
*** WARNING: The wtd.quantile function DOES interpolate between values, even if type='i/n'
There does not seem to be a way to fix that for the wtd.quantile() function. For example,
wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame

See Also

pctiles pctiles.exact pctiles.a.over.b wtd.pctiles.exact wtd.pctiles wtd.pctiles.fast

Examples

```
#
```

pctiles.a.over.b *Show the rounded values at 100 percentiles for a/b (or zero if b=0)*

Description

Get a quick look at a distribution by seeing the rounded values at 100 percentiles for a/b (setting a/b to zero if b=0)

Usage

```
pctiles.a.over.b(a, b, digits = 3)
```

Arguments

a	Required numeric vector of values that are numerator of ratio whose distribution you want to look at.
b	Required numeric vector of values that are denominator of ratio whose distribution you want to look at.
digits	Number, 3 by default, specifying how many decimal places to round to

Details

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES !# Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. This is how should be calculating percentiles as of 2/2013. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################
*** WARNING: The Hmisc::wtd.quantile function DOES interpolate between values, even if type='i/n'
There does not seem to be a way to fix that for the Hmisc::wtd.quantile() function. For example,
Hmisc::wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame

See Also

pctiles pctiles.exact pctiles.a.over.b wtd.pctiles.exact wtd.pctiles wtd.pctiles.fast

Examples

```
#
```

---

pctiles.exact *Show the not-rounded values at 100 percentiles*

Description

Get a quick look at a distribution by seeing the 100 values that are the percentiles 1-100

Usage

```
pctiles.exact(x)
```

Arguments

x	Required numeric vector of values whose distribution you want to look at.

Details

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES !# Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. This is how should be calculating percentiles as of 2/2013. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################
*** WARNING: The Hmisc::wtd.quantile function DOES interpolate between values, even if type='i/n'
There does not seem to be a way to fix that for the Hmisc::wtd.quantile() function. For example,
Hmisc::wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame

See Also

pctiles pctiles.exact pctiles.a.over.b wtd.pctiles.exact wtd.pctiles wtd.pctiles.fast

Examples

```
#
```

pdf2 *Overlay two simple histograms (pdf=probability density functions, not pdf filetype)*

Description

Overlay two simple histograms, for data below vs above a cutoff

Usage

```
pdf2(x, binx, threshold, n = 100, colors = c("gray", "red"), ...)
```

Arguments

x	Variable for histogram
binx	Variable that defines subsets
threshold	Variable that defines cutoff, so binx<threshold is plotted first, in one color, then binx>=threshold in other color is drawn over that
n	Default is 100. n is just passed to the hist function.
colors	Character vector length 2, with colors for first and second histogram
...	Other named parameters sent to hist, such as main, xlab, ylab

Value

Just plots using hist.

Examples

```
  ## Not run:
  pdf2(
  log10(bg$proximity.npl),
  bg$bin.EJ.DISPARITY.proximity.npl.eo, 10,
  main ='npl for high top 10 pct vs low EJ', xlab='NPL score log10'
  )

## End(Not run)
```

put.first *Simple way to put certain cols first, in a data.frame*

Description

Returns a data.frame with specified columns put first, before the others.

Usage

```
put.first(x, fields)
```

Arguments

x	Required data.frame that will have its columns reordered
fields	required character vector of strings that are among the elements of names(x)

Value

Returns a transformed data.frame with cols in new order

See Also

change.fieldnames

Examples

```
before <- data.frame(year=c(2,2,2), ID=3, numbers=4, last=1)
put.first(before, c('ID', 'numbers'))
after <- put.first(before, names(before)[length(before)] ) # put last column first
before; after
```

rmall *Help removing all objects from memory*

Description

A simple way to get a reminder of how to clear all objects from memory because I always forget how

Usage

```
rmall()
```

Value

prints how to do that

rms — *Root Mean Square (RMS), or Quadratic Mean*

Description

Returns the RMS, or quadratic mean of a vector of numbers.

Usage

```
rms(x, na.rm = FALSE)
```

Arguments

`x`	Vector of numbers, required.
`na.rm`	Logical value, optional, FALSE by default. If FALSE, result is NA if any of the values in x is NA. If TRUE, remove the NA values first.

Details

The quadratic mean is one type of average. It is the square root of the arithmetic mean of the squares. See http://en.wikipedia.org/wiki/Root_mean_square or http://mathworld.wolfram.com/Root-Mean-Square.html for many applications

Value

Returns a single number

See Also

geomean mean harmean

Examples

```
rms(c(1,2,4))
```

rowMaxs — *Returns the max value of each row of a data.frame or matrix*

Description

Returns maximum value of each row of a data.frame or matrix.

Usage

```
rowMaxs(df, na.rm = TRUE)
```

Arguments

`df`	Data.frame or matrix, required.
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in the given vector.

Details

** NOTE: The useful matrixStats package will provide the basis for extended rowMins, rowMax, colMins, colMaxs functions to be made available through this package. Source: Henrik Bengtsson (2015). matrixStats: Methods that Apply to Rows and Columns of a Matrix. R package version 0.13.1-9000.
https://github.com/HenrikBengtsson/matrixStats
Initially, separate functions were written here for those four functions, and the versions here were more flexible and convenient for some purposes, e.g., handling data.frames and different na.rm defaults, but the matrixStats versions are much faster (e.g., by 4x or more). Ideally, this analyze.stuff package would be modified to just extend those functions by providing them methods to handle data.frames, not just matrix class objects, and perhaps provide new or different parameters or defaults, such as defaulting to na.rm=TRUE instead of FALSE, and handling factor class columns in a data.frame. That has not been done yet, so colMaxs() etc. refer to the slower more flexible ones, and the faster matrix-only ones are via matrixStats::colMaxs etc.

** NOTE: max() and min() and matrixStats::colMaxs from matrixStats etc. default to na.rm=FALSE, but this function defaults to na.rm=TRUE because that just seems more frequently useful.

** NOTE: `min` and `max` & this function will handle character elements by coercing all others in the column to character, which can be confusing – e.g., note that min(c(8,10,'txt')) returns '10' not '8' and max returns 'txt' (also see the help for `Comparison`)

If this worked just like max() and min(), cols that are factors would make this fail. max or min of a factor fails, even if as.character() of the factor would return a valid numeric vector. That isn't an issue with a matrix, but a data.frame might have numbers stored as factor. To fix that, this uses `factor.as.numeric` with parameters that try to convert character or factor columns to numeric.

Based on how `min` and `max` behave, return Inf or -Inf if no non-missing arguments to min or max respectively. To suppress that warning when using this function, use `suppressWarnings`(func(x))

Value

Returns a vector of numbers of length equal to number of rows in df.

See Also

`factor.as.numeric` `rowMaxs` `rowMins` `colMaxs` `colMins` `count.above` `pct.above` `pct.below` `cols.above.which` `cols.above.pct`

Other functions for max and min of rows and columns: `colMaxs`, `colMins`, `rowMins`

Examples

```
blah <- rbind(NA, data.frame(a=c(0, 0:8), b=c(0.1+(0:9)), c=c(1:10), d=c(rep(NA, 10)),
  e=TRUE, f=factor('factor'), g='words', stringsAsFactors=FALSE) )
cbind(blah, min=rowMins(blah), max=rowMaxs(blah))
rbind(blah, min=colMins(blah), max=colMaxs(blah))
blah <- blah[ , sapply(blah, function(x) is.numeric(x) | is.logical(x)) ]
cbind(blah, min=rowMins(blah), max=rowMaxs(blah),
  mean=rowMeans(blah, na.rm=TRUE), sum=rowSums(blah, na.rm=TRUE))
rbind(blah, min=colMins(blah), max=colMaxs(blah),
  mean=colMeans(blah, na.rm=TRUE), sum=colSums(blah, na.rm=TRUE))
  # ** Actually, matrixStats does this ~4x as quickly,
  # although no practical difference unless large dataset:
  n <- 1e7
t1=Sys.time(); x=analyze.stuff::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
t1=Sys.time(); x=  matrixStats::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
# Note the latter cannot handle a data.frame:
## Not run:
# This would fail:
matrixStats::colMaxs(   data.frame(a=1:10, b=2))
# This works:
analyze.stuff::colMaxs( data.frame(a=1:10, b=2))

## End(Not run)
```

rowMins *Returns the min value of each row of a data.frame or matrix*

Description

Returns minimum value of each row of a data.frame or matrix.

Usage

```
rowMins(df, na.rm = TRUE)
```

Arguments

df	Data.frame or matrix, required.
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in the given vector.

Details

** NOTE: The useful matrixStats package will provide the basis for extended rowMins, rowMax, colMins, colMaxs functions to be made available through this package. Source: Henrik Bengtsson (2015). matrixStats: Methods that Apply to Rows and Columns of a Matrix. R package version 0.13.1-9000.
https://github.com/HenrikBengtsson/matrixStats
Initially, separate functions were written here for those four functions, and the versions here were more flexible and convenient for some purposes, e.g., handling data.frames and different na.rm defaults, but the matrixStats versions are much faster (e.g., by 4x or more). Ideally, this analyze.stuff package would be modified to just extend those functions by providing them methods to handle data.frames, not just matrix class objects, and perhaps provide new or different parameters or defaults, such as defaulting to na.rm=TRUE instead of FALSE, and handling factor class columns in a data.frame. That has not been done yet, so colMaxs() etc. refer to the slower more flexible ones, and the faster matrix-only ones are via matrixStats::colMaxs etc.

** NOTE: max() and min() and matrixStats::colMaxs from matrixStats etc. default to na.rm=FALSE, but this function defaults to na.rm=TRUE because that just seems more frequently useful.

** NOTE: min and max & this function will handle character elements by coercing all others in the column to character, which can be confusing – e.g., note that min(c(8,10,'txt')) returns '10' not '8' and max returns 'txt' (also see the help for Comparison)

If this worked just like max() and min(), cols that are factors would make this fail. max or min of a factor fails, even if as.character() of the factor would return a valid numeric vector. That isn't an issue with a matrix, but a data.frame might have numbers stored as factor. To fix that, this uses factor.as.numeric with parameters that try to convert character or factor columns to numeric.

Based on how min and max behave, return Inf or -Inf if no non-missing arguments to min or max respectively. To suppress that warning when using this function, use suppressWarnings(func(x))

Value

Returns a vector of numbers of length equal to number of rows in df.

See Also

factor.as.numeric rowMaxs rowMins colMaxs colMins count.above pct.above pct.below cols.above.which cols.above.pct

Other functions for max and min of rows and columns: colMaxs, colMins, rowMaxs

Examples

```
blah <- rbind(NA, data.frame(a=c(0, 0:8), b=c(0.1+(0:9)), c=c(1:10), d=c(rep(NA, 10)),
  e=TRUE, f=factor('factor'), g='words', stringsAsFactors=FALSE) )
cbind(blah, min=rowMins(blah), max=rowMaxs(blah))
rbind(blah, min=colMins(blah), max=colMaxs(blah))
blah <- blah[ , sapply(blah, function(x) is.numeric(x) | is.logical(x)) ]
cbind(blah, min=rowMins(blah), max=rowMaxs(blah),
  mean=rowMeans(blah, na.rm=TRUE), sum=rowSums(blah, na.rm=TRUE))
rbind(blah, min=colMins(blah), max=colMaxs(blah),
  mean=colMeans(blah, na.rm=TRUE), sum=colSums(blah, na.rm=TRUE))
  # ** Actually, matrixStats does this ~4x as quickly,
  # although no practical difference unless large dataset:
  n <- 1e7
t1=Sys.time(); x=analyze.stuff::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
t1=Sys.time(); x=  matrixStats::colMaxs( cbind(a=1:n, b=2, c=3, d=4, e=5)); t2=Sys.time()
print(difftime(t2,t1))
# Note the latter cannot handle a data.frame:
## Not run:
# This would fail:
matrixStats::colMaxs(   data.frame(a=1:10, b=2))
# This works:
analyze.stuff::colMaxs( data.frame(a=1:10, b=2))

## End(Not run)
```

`rows.above.count` *Number or percent of rows (for each col) where value exceeds cutoff(s)*

Description

Alias for `count.above`

Usage

```
rows.above.count(df, benchmarks = "mean", benchnames = "cutoff",
  or.tied = FALSE, below = FALSE, wts = 1, na.rm = TRUE)
```

Arguments

`df`	Data.frame or matrix, required.
`benchmarks`	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
`benchnames`	Default is 'cutoff' and this string is used to create colnames for the results, such as above.cutoff.for.field1
`or.tied`	Logical, FALSE by default, reporting on those > cutoff. But, if or.tied=TRUE, this reports on those >= cutoff.
`below`	Logical, FALSE by default, which counts how many are above cutoff (or tied if or.tied). If TRUE, counts how many are below (or tied with) cutoff.
`wts`	Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: `cols.above.count`, `cols.above.pct`, `cols.above.which`, `count.above`, `count.below`, `pct.above`, `pct.below`, `rows.above.pct`, `rows.above.which`, `rows.below.count`, `rows.below.pct`

rows.above.pct *Number or percent of rows (for each col) where value exceeds cutoff(s)*

Description

Alias for `pct.above`

Usage

```
rows.above.pct(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = FALSE, or.tied = FALSE, below = FALSE, wts = 1,
  of.what = "all")
```

Arguments

`df`	Data.frame or matrix, required.
`benchmarks`	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
`benchnames`	Default is 'cutoff' and this string is used to create colnames for the results, such as above.cutoff.for.field1
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed before value is found. Otherwise result will be NA when any NA is in a col.
`or.tied`	Logical, FALSE by default, reporting on those > cutoff. But, if or.tied=TRUE, this reports on those >= cutoff.
`below`	Logical, FALSE by default, which counts how many are above cutoff (or tied if or.tied). If TRUE, counts how many are below (or tied with) cutoff.
`wts`	Number or vector, default is 1. Length must be a factor of # rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.
`of.what`	Optional, character, 'all' by default, defines xxx as the text used in "pct.above.xxx" (or below) for fieldnames in results

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

`count.above` `pct.above` `pct.below` to see, for each column, the count or percent of rows that have values above or below a cutoff.

`cols.above.count` `cols.above.which` `cols.above.pct` to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.above, pct.below, rows.above.count, rows.above.which, rows.below.count, rows.below.pct

---

rows.above.which *Does each Row have a Value at or above Cutoff(s)*

Description

Flag which cells are at or above some cutoff(s) or mean.

Usage

```
rows.above.which(x, cutoff, or.tied = FALSE, below = FALSE)
```

Arguments

x	Data.frame or matrix of numbers to be compared to cutoff value.
cutoff	The numeric threshold or cutoff to which numbers are compared. Default is arithmetic mean of row. Usually one number, but can be a vector of same length as number of rows, in which case each row can use a different cutoff.
or.tied	Logical. Default is FALSE, which means we check if number in x is greater than the cutoff (>). If TRUE, check if greater than or equal (>=).
below	Logical. Default is FALSE. If TRUE, uses > or >= cutoff. If FALSE, uses < or <= cutoff.

Details

For a matrix with a few cols of related data, find which cells are at/above (or below) some cutoff. Returns a logical matrix, with TRUE for each cell that is at/above the cutoff. Can be used in identifying places (rows) where some indicator(s) is/are at/above a cutoff, threshold value.

Value

Returns a logical matrix the same size as x. ** Note this is different than which – That function returns the positions of TRUE elements but this returns TRUE or FALSE for all elements.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.above, pct.below, rows.above.count, rows.above.pct, rows.below.count, rows.below.pct

Examples

```
out <- cols.above.which(x<-data.frame(a=1:10, b=rep(7,10), c=7:16), cutoff=7)
out
out # default is or.tied=FALSE
out <- cols.above.which(data.frame(a=1:10, b=rep(7,10), c=7:16),
  cutoff=7, or.tied=TRUE, below=TRUE)
out
out <- cols.above.which(data.frame(a=1:10, b=rep(7,10), c=7:16) )
 # Compares each number in each row to the row's mean.
out
```

`rows.below.count` *Number or percent of rows (for each col) where value is below cutoff(s)*

Description

Alias for `count.below`

Usage

```
rows.below.count(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = TRUE, or.tied = FALSE, below = TRUE, wts = 1)
```

Arguments

`df`	Data.frame or matrix, required.
`benchmarks`	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
`benchnames`	Default is 'cutoff' and this string is used to create colnames for the results
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.
`or.tied`	Logical, FALSE by default, reporting on those < cutoff. But, if or.tied=TRUE, this reports on those <= cutoff.
`below`	Logical, TRUE by default, which counts how many are below cutoff (or tied if or.tied). If FALSE, counts how many are above (or tied with) cutoff.
`wts`	Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.

Details

See `count.below` for details, for which this is a wrapper.

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.above, pct.below, rows.above.count, rows.above.pct, rows.above.which, rows.below.pct

---

rows.below.pct *Number or percent of rows (for each col) where value is below cutoff(s)*

Description

Alias for pct.below

Usage

```
rows.below.pct(df, benchmarks = "mean", benchnames = "cutoff",
  na.rm = FALSE, or.tied = FALSE, below = FALSE, wts = 1,
  of.what = "all")
```

Arguments

df	Data.frame or matrix, required.
benchmarks	Default is 'mean' but otherwise this must be a number or numeric vector of thresholds to compare values to.
benchnames	Default is 'cutoff' and this string is used to create colnames for the results
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed first. Otherwise result will be NA when any NA is in a col.
or.tied	Logical, FALSE by default, reporting on those < cutoff. But, if or.tied=TRUE, this reports on those <= cutoff.
below	Logical, TRUE by default, which counts how many are below cutoff (or tied if or.tied). If FALSE, counts how many are above (or tied with) cutoff.
wts	Number or vector, default is 1. Length must be a factor of number of rows in df, so length(df[,1]) is an integer multiple of length(wts) Applies weights to when counting how many.
of.what	Optional, character, 'all' by default, defines xxx as the text used in "pct.above.xxx" (or below) for fieldnames in results

Value

Returns a vector of numbers of length equal to number of columns in df.

Note

Future work: these functions could have wts, na.rm, & allow cutoffs or benchmarks as a vector (not just 1 number), & have benchnames.

See Also

count.above pct.above pct.below to see, for each column, the count or percent of rows that have values above or below a cutoff.

cols.above.count cols.above.which cols.above.pct to see, for each row, the count or which or fraction of columns with numbers at/above/below cutoff.

Other functions for above and below: cols.above.count, cols.above.pct, cols.above.which, count.above, count.below, pct.above, pct.below, rows.above.count, rows.above.pct, rows.above.which, rows.below.count

setdiff2 *Differences between sets a and b*

Description

Returns the elements that in a or b but not in both (i.e., the differences between sets a and b)

Usage

```
setdiff2(a, b)
```

Arguments

a	Required vector
b	Required vector

Value

Vector of elements

See Also

setdiff which is a bit different

Examples

```
setdiff2(1:10, 3:12)
setdiff2(c('a','b','c'), c('b','c','d'))
```

signifarray *Specify Significant Digits for Each Column*

Description

Given a matrix or numeric data.frame, round each column to a specified column-specific number of significant digits.

Usage

```
signifarray(dat, digits = 6)
```

Arguments

dat	Required, matrix or numeric data.frame with the values to be rounded.
digits	Optional, 6 by default. Can be a vector as long as the number of columns in dat, where each elements specifies the number of significant digits to retain for numbers in the corresponding column of dat.

Value

Returns dat, but with numbers rounded based on digits parameter.

See Also

signif

Examples

```
signifarray(matrix(rnorm(9*5), ncol=5), 1:5)
signifarray(data.frame(a=rnorm(10), b=rnorm(10), c=rnorm(10)), 1:3)
```

similar *See how closely numeric values match in 2 datasets*

Description

Compare two vectors, matrices, or data.frames of numbers to see how often they are similar.

Usage

```
similar(a, b, tol = 99.99, na.rm = FALSE, shownames = TRUE)
```

Arguments

`a`	Required first vector, data.frame, or matrix
`b`	Required second vector, data.frame, or matrix
`tol`	Number, 99.99 by default, specifying tolerance as a percentage 0-100, such that "similar" is defined as the two values being within 100-tol percent of each other.
`na.rm`	Logical value, optional, FALSE by default. not implemented here yet. Should NA values be removed first, or compared and treated as NA matches NA.
`shownames`	Logical value, optional, TRUE by default. Not used. Should names be shown in results?

Details

This function returns a matrix or vector showing how many rows in vector a are within 100-tol percent of the value in vector b. May want to add a 3d case, where NA can match NA.

Value

Data.frame showing what # of rows are "similar" in dataset a vs b, for each column.

See Also

`similar.p`, `all.equal`, `identical`, `isTRUE`, `==`, `all`

Examples

```
similar.p(1:10, (1:10) * 1.001 )
similar.p(data.frame(x=1:10, y=101:110), data.frame(other=1.001*(1:10),
  other2=c(101:109, 110.01) ))
```

`similar.p` *See how closely numeric values match in 2 datasets*

Description

Compare two vectors, matrices, or data.frames of numbers to see how often they are similar.

Usage

```
similar.p(a, b, tol = 99.99, na.rm = FALSE)
```

Arguments

`a`	Required first vector, data.frame, or matrix
`b`	Required second vector, data.frame, or matrix
`tol`	Number, 99.99 by default, specifying tolerance as a percentage 0-100, such that "similar" is defined as the two values being within 100-tol percent of each other.
`na.rm`	Logical value, optional, FALSE by default. not implemented here yet. Should NA values be removed first, or compared and treated as NA matches NA.

Details

This function returns a matrix or vector showing how many rows in vector a are within 100-tol percent of the value in vector b. May want to add a 3d case, where NA can match NA.

Value

Data.frame showing what

See Also

`similar`, `all.equal`, `identical`, `isTRUE`, `==`, `all`

Examples

```
similar(1:10, (1:10) * 1.001 )
similar(
 data.frame(x=1:10, y=101:110),
 data.frame(other=1.001*(1:10), other2=c(101:109, 110.01) )
)
```

`tabular` *Format a table in roxygen documentation of function in a package*

Description

modified version of func in help section on formatting in **roxygen2** package

Usage

```
tabular(df, ...)
```

Arguments

`df`	data.frame required
`...`	optional parameters passed through to lapply(df, format, ...)

Value

Returns text that can be pasted into documentation of a function or data in a package

See Also

Help on formatting in **roxygen2**

Examples

```
 tabular(mtcars[1:5, 1:5])
 tabular(df = data.frame(a=7:16, b='stuff', c=999, d=c('blah','junk')) )
```

tb *wrapper for table() that sorts by counts, decreasing*

Description

wrapper for table() that sorts by counts, decreasing

Usage

```
tb(x, useNA = "always", ...)
```

Arguments

`x`	required, passed to table(x)
`useNA`	default is 'always', passed to table()
`...`	other parameters passed to table() – cannot pass anything to cbind or sort like decreasing=FALSE

Value

like cbind

unzip.files *Unzip multiple zip files*

Description

Wrapper for `unzip` which unzips a single file.

Usage

```
unzip.files(zipfile, files = NULL, exdir = ".", unzip = "internal",
  overwrite = TRUE, ...)
```

Arguments

`zipfile`	vector of names of files to unzip
`files`	Optional, NULL by default which signifies all files in each zipfile will be extracted. Otherwise, a list, with the nth element being a vector (length 1 or more) of character string names of files to extract from the nth zipfile.
`exdir`	The directory to extract files to (the equivalent of unzip -d). It will be created if necessary.
`unzip`	See help for unzip
`overwrite`	Logical, optional, TRUE by default which means the local file is not overwritten if it already exists.
`...`	Other arguments passed through to unzip

Value

Returns a list of the filepaths extracted to, from each zipfile. Names of list are the zip file names.

wtd.colMeans *Weighted Mean of each Column - WORK IN PROGRESS (NA HANDLING NOT YET TESTED)*

Description

Returns weighted mean of each column of a data.frame or matrix, based on specified weights, one weight per row. Now based on `data.table` unlike `wtd.colMeans2`

Usage

```
wtd.colMeans(x, wts, by, na.rm = TRUE, dims = 1)
```

Arguments

`x`	Data.frame or matrix, required.
`wts`	Weights, optional, defaults to 1 which is unweighted, numeric vector of length equal to number of rows
`by`	Optional vector, default is none, that can provide a single column name (as character) or character vector of column names, specifying what to group by, producing the weighted mean within each group. See help for `data.table`
`na.rm`	Logical value, optional, TRUE by default. Defines whether NA values should be removed before result is found. Otherwise result will be NA when any NA is in a vector.
`dims`	dims=1 is default. Not used. integer: Which dimensions are regarded as 'rows' or 'columns' to sum over. For row, the sum or mean is over dimensions dims+1, ...; for col it is over dimensions 1:dims.

Details

** not yet handling factor or character fields well. For cols with NA values, mean uses total number of rows (or sum of non-NA weights) as denominator?***, not just rows where the actual value is non-NA!
Note Hmisc::wtd.mean is not exactly same as stats::weighted.mean since na.rm defaults differ
Hmisc::wtd.mean(x, weights=NULL, normwt="ignored", na.rm = TRUE) # Note na.rm defaults differ.
weighted.mean(x, w, ..., na.rm = FALSE)

Value

If by is not specified, returns a vector of numbers of length equal to number of columns in df. If by is specified, returns weighted mean for each column in each subset defined via by.

Examples

```
library(analyze.stuff)
n <- 1e6
mydf <- data.frame(pop=1000 + abs(rnorm(n, 1000, 200)), v1= runif(n, 0, 1),
 v2= rnorm(n, 100, 15), REGION=c('R1','R2',sample(c('R1', 'R2', 'R3'), n-2, replace=TRUE)),
  stringsAsFactors = FALSE)
  mydf$pop[mydf$REGION=='R2'] <- 4 * mydf$pop[mydf$REGION=='R2']
```

```
mydf$v1[mydf$REGION=='R2'] <- 4 * mydf$v1[mydf$REGION=='R2']
wtd.colMeans(mydf[,1:3])
wtd.colMeans(mydf[,1:3], wts=mydf$pop)
wtd.colMeans(mydf, by='REGION')
# R HANGS/STUCK: # wtd.colMeans(mydf[1:100,1:3], by=mydf$REGION, wts=mydf$pop)
mydf2 <- data.frame(a=1:3, b=c(1,2,NA))
wtd.colMeans(mydf2)
wtd.colMeans(mydf2, na.rm=TRUE)
```

wtd.colMeans2 *Weighted Mean of each Column - WORK IN PROGRESS*

Description

Returns weighted mean of each column of a data.frame or matrix, based on specified weights, one weight per row. But also see `data.table` used for `wtd.colMeans`

Usage

```
wtd.colMeans2(x, wts, by, na.rm = FALSE, dims = 1)
```

Arguments

x	Data.frame or matrix, required.
wts	Weights, optional, defaults to nothing i.e. unweighted, and if specified must be vector of weights recycled to be same length as NROW(x) # not the name of the weights field in data.frame x, as single character string, e.g., "weightcol"
by	Optional vector, default is none, that can provide a single column name (as character) or character vector of column names,
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed before result is found. Otherwise result will be NA when any NA is in a vector.
dims	dims=1 is default. **Not used.** integer: Which dimensions are regarded as 'rows' or 'columns' to sum over. For row*, the sum or mean is over dimensions dims+1, ...; for col* it is over dimensions 1:dims.

Value

Returns a vector of numbers of length equal to number of columns in df.

See Also

`wtd.colMeans` `wtd.rowMeans` `wtd.rowSums` `rowMaxs` `rowMins` `colMins`

Examples

```
x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA)
cbind(x, wtd.rowMeans(x, w) )
cbind(x, wtd.rowSums(x, w) )
x=data.frame(a=c(NA, 2:4), b=rep(100,4), c=rep(3,4))
w=c(1.1, 2, NA, 0)
```

```
print(cbind(x,w, wtd=w*x))
print(wtd.colMeans(x, w, na.rm=TRUE))
#rbind(cbind(x,w,wtd=w*x), c(wtd.colMeans(x,w,na.rm=TRUE), 'wtd.colMeans', rep(NA,length(w))))

x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA, rep(1, 7))
print(cbind(x,w, wtd=w*x))
rbind(cbind(x, w), cbind(wtd.colMeans(x, w, na.rm=TRUE), w='wtd.colMeans') )
print(w*cbind(x,w))
```

---

wtd.pctiles *Show the rounded values at 100 weighted percentiles*

Description

Get a quick look at a weighted distribution by seeing the 100 values that are the weighted percentiles 1-100

Usage

```
wtd.pctiles(x, wts = NULL, na.rm = TRUE, type = "i/n",
  probs = (1:100)/100, digits = 3)
```

Arguments

x	Required, numeric vector (or data.frame) of values whose distribution(s) you want to look at.
wts	NULL by default, or vector of numbers (same length as x vector or as a column of x) to use as weights in Hmisc::wtd.quantile
na.rm	Logical optional TRUE by default, in which case NA values are removed first.
type	'i/n' is default. See help for wtd.quantile[Hmisc]()
probs	fractions 0-1, optional, (1:100)/100 by default, define quantiles to use
digits	Number, 3 by default, specifying how many decimal places to round to in results

Details

Provides weighted percentiles using wtd.quantile

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES ! Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################

*** WARNING: The Hmisc::wtd.quantile function DOES interpolate between values, even if type='i/n' There does not seem to be a way to fix that for the Hmisc::wtd.quantile() function. For example, Hmisc::wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame, one row per probs, so 100 by default (1

See Also

`pctiles` `pctiles.exact` `pctiles.a.over.b` `wtd.pctiles.exact` `wtd.pctiles` `wtd.pctiles.fast`

`wtd.pctiles.exact` *Show the values at 100 weighted percentiles*

Description

Get a quick look at a weighted distribution by seeing the 100 values that are the percentiles 1-100

Usage

```
wtd.pctiles.exact(x, wts = NULL, na.rm = TRUE, type = "i/n",
  probs = (1:100)/100)
```

Arguments

`x`	Required numeric vector of values whose distribution you want to look at.
`wts`	NULL by default, or vector of numbers to use as weights in Hmisc::wtd.quantile
`na.rm`	Logical optional TRUE by default, in which case NA values are removed first.
`type`	'i/n' is default. See help for `wtd.quantile[Hmisc]()`
`probs`	fractions 0-1, optional, (1:100)/100 by default, define quantiles to use

Details

Provides weighted percentiles using `wtd.quantile`

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES !# Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. This is how should be calculating percentiles as of 2/2013. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################

*** WARNING: The Hmisc::wtd.quantile function DOES interpolate between values, even if type='i/n'
There does not seem to be a way to fix that for the Hmisc::wtd.quantile() function. For example,
Hmisc::wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame

See Also

`pctiles` `pctiles.exact` `pctiles.a.over.b` `wtd.pctiles.exact` `wtd.pctiles` `wtd.pctiles.fast`

wtd.pctiles.fast *Show the values at 100 weighted percentiles*

Description

Get a quick look at a weighted distribution by seeing the 100 values that are the weighted percentiles 1-100

Usage

```
wtd.pctiles.fast(x, wts = NULL, na.rm = TRUE)
```

Arguments

`x`	Required numeric vector of values whose distribution you want to look at.
`wts`	NULL by default, or vector of numbers to use as weights in Hmisc::wtd.quantile
`na.rm`	Logical optional TRUE by default, in which case NA values are removed first.

Details

Provides weighted percentiles without using `wtd.quantile`

# NOTE: THIS ONLY SHOWS PERCENTILES AND MEAN FOR THE VALID (NOT NA) VALUES ! Defining these types as type=1 and type="i/n" will create simple discontinuous quantiles, without interpolation where there are jumps in the values analyzed. *** WARNING: Unless set type=1, the default type=7 in which case quantile() FUNCTION INTERPOLATES, WHICH ISN'T OBVIOUS IN EVERY DATASET! use type=1 to avoid interpolation. and pctiles() rounded results so interpolation would be even less apparent.
The quantile function will NOT interpolate between values if type=1:
quantile(1:12, probs=(1:10)/10, type=1)
10 2 3 4 5 6 8 9 10 11 12
############################
**** IMPORTANT ***
############################
*** WARNING: The Hmisc::wtd.quantile function DOES interpolate between values, even if type='i/n'
There does not seem to be a way to fix that for the Hmisc::wtd.quantile() function. For example,
Hmisc::wtd.quantile(1:12, probs=(1:10)/10, type='i/n', weights=rep(1,12))
10 1.2 2.4 3.6 4.8 6.0 7.2 8.4 9.6 10.8 12.0

Value

Returns a data.frame

See Also

pctiles pctiles.exact pctiles.a.over.b wtd.pctiles.exact wtd.pctiles wtd.pctiles.fast

---

wtd.rowMeans *Weighted Mean of each Row - WORK IN PROGRESS*

Description

Returns weighted mean of each row of a data.frame or matrix, based on specified weights, one weight per column.

Usage

```
wtd.rowMeans(x, wts = 1, na.rm = FALSE, dims = 1)
```

Arguments

x	Data.frame or matrix, required.
wts	Weights, optional, defaults to 1 which is unweighted, numeric vector of length equal to number of columns
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed before result is found. Otherwise result will be NA when any NA is in a vector.
dims	dims=1 is default. Not used. integer: Which dimensions are regarded as ’rows’ or ’columns’ to sum over. For row*, the sum or mean is over dimensions dims+1, ...; for col* it is over dimensions 1:dims.

Value

Returns a vector of numbers of length equal to number of rows in df.

See Also

wtd.colMeans wtd.rowMeans wtd.rowSums rowMaxs rowMins colMins

Examples

```
x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA)
cbind(x, wtd.rowMeans(x, w) )
cbind(x, wtd.rowSums(x, w) )
x=data.frame(a=c(NA, 2:4), b=rep(100,4), c=rep(3,4))
w=c(1.1, 2, NA, 0)
print(cbind(x,w, wtd=w*x))
print(wtd.colMeans(x, w, na.rm=TRUE))
#rbind(cbind(x,w,wtd=w*x), c(wtd.colMeans(x,w,na.rm=TRUE), 'wtd.colMeans', rep(NA,length(w))))
```

```
x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA, rep(1, 7))
print(cbind(x,w, wtd=w*x))
rbind(cbind(x, w), cbind(wtd.colMeans(x, w, na.rm=TRUE), w='wtd.colMeans') )
print(w*cbind(x,w))
```

wtd.rowSums *Weighted Sum of each Row*

Description

Returns weighted sum of each row of a data.frame or matrix, based on specified weights, one weight per column.

Usage

```
wtd.rowSums(x, wts = 1, na.rm = TRUE)
```

Arguments

x	Data.frame or matrix, required.
wts	Weights, optional, defaults to 1 which is unweighted, numeric vector of length equal to number of columns
na.rm	Logical value, optional, TRUE by default. Defines whether NA values should be removed before result is found. Otherwise result will be NA when any NA is in a vector.

Value

Returns a vector of numbers of length equal to number of rows in df.

See Also

wtd.colMeans wtd.rowMeans wtd.rowSums rowMaxs rowMins colMins

Examples

```
x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA)
cbind(x, wtd.rowMeans(x, w) )
cbind(x, wtd.rowSums(x, w) )
x=data.frame(a=c(NA, 2:4), b=rep(100,4), c=rep(3,4))
w=c(1.1, 2, NA, 0)
print(cbind(x,w, wtd=w*x))
print(wtd.colMeans(x, w, na.rm=TRUE))
#rbind(cbind(x,w,wtd=w*x), c(wtd.colMeans(x,w,na.rm=TRUE), 'wtd.colMeans', rep(NA,length(w))))

x=data.frame(a=c(NA, 2:10), b=rep(100,10), c=rep(3,10))
w=c(1.1, 2, NA, rep(1, 7))
print(cbind(x,w, wtd=w*x))
rbind(cbind(x, w), cbind(wtd.colMeans(x, w, na.rm=TRUE), w='wtd.colMeans') )
print(w*cbind(x,w))
```

Index