Package ‘mrdwabmisc’

March 29, 2013

Type Package

Title Miscellaneous R functions, mostly for data processing

Version 1.0

Date 2013-01-22

Author Ananda Mahto

Maintainer Ananda Mahto <mrdwab@gmail.com>

Description Miscellaneous R functions, some utility, and others to clean and organize data.

Depends R (>= 2.10),digest,reshape2

License GPL-2

LazyLoad yes

Collate
'concat.split.R''df.sorter.R''multi.freq.table.R''random.names.R''row.extractor.R''sample.size.R''stratified.R''stringseed.s
package.R''aggregate2.R''CBIND.R''dfcols.list.R''FacsToChars.R''getSize.R''LinearizeNestedList.R''load.scripts.and.d

R topics documented:

`mrdwabmisc-package` *mrdwabmisc*

Description

Miscellaneous R functions, some utility, and others to clean and organize data.

Details

Package:	mrdwabmisc
Type:	Package
Version:	1.0
Date:	2013-01-22
License:	GPL-2

Author(s)

Ananda Mahto

Maintainer: Ananda Mahto <mrdwab@gmail.com>

Examples

```
## concat.split
data(concatenated)
head(concat.test)
head(concat.split(concat.test, "Likes", drop.col = TRUE))


## sample.size
sample.size(population = 300)
sample.size(population = 300, c.lev = 97)

## stringseed.sampling
stringseed.sampling("Santa Barbara", 1920, 100)

## table2df
table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph),
       as.multitable = TRUE, direction = "wide")[[1]]
head(table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph)))
## aggregate2
aggregate2(ToothGrowth, "len", ".", c("sum", "mean"))

## CBIND
df1 <- data.frame(A = 1:5, B = letters[1:5])
df2 <- data.frame(C = 1:3, D = letters[1:3])
df3 <- data.frame(E = 1:8, F = letters[1:8], G = LETTERS[1:8])
#'CBIND(list(df1, df2, df3))

## FacsToChars
dat <- data.frame(title = c("title1", "title2", "title3"),
            author = c("author1", "author2", "author3"),
            customerID = c(1, 2, 1))
str(dat)
FacsToChars(dat, overwrite = TRUE)
str(dat)

## makemeNA
# Some sample data
temp <- data.frame(
V1 = c(1:3),
V2 = c(1, "*", 3),
V3 = c("a", "*", "c"),
V4 = c(".", "*", "3"))
temp
makemeNA(temp, c("*", "."))
```

aggregate2 *Perform multiple aggregation functions on grouped data*

Description

Base R's `aggregate` function allows you to specify multiple functions when aggregating. However, the output of such commands is a `data.frame` where the aggregated "columns" are actually matrices. aggregate2 is a basic wrapper around aggregate that outputs a regular data.frame instead.

Usage

```
aggregate2(data, aggs, ids, funs = NULL, ...)
```

Arguments

data	Your data.frame
aggs	The variables that need to be aggregated, specified as a character vector.
ids	The variables that serve as grouping variables, specified as a character vector.
funs	The functions that you want to apply, specified as a character vector.
...	Further arguments to aggregate. Really only useful for the subset argument.

Note

This function essentially constructs a formula that can be used with aggregate and keeps track of the names of the aggregation functions you have applied to create new variable names. This function is not very useful when the output of FUN would already output a matrix (for example, if FUN = fivenum or FUN = summary). In such cases, it is recommended to use base R's aggregate with a do.call. For example: do.call("data.frame", aggregate(. ~ Species, iris, summary)).

Author(s)

Ananda Mahto

See Also

aggregate

Examples

```
# One-to-one, two functions
(temp1a <- aggregate(weight ~ feed, data = chickwts,
               function(x) cbind(mean(x), sum(x))))
str(temp1a)
(temp1b <- aggregate2(chickwts, "weight", "feed", c("mean", "sum")))
str(temp1b)

# Many-to-many, two functions
(temp2a <- aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, data = esoph,
               function(x) cbind(sum(x), mean(x))))
str(temp2a)
(temp2b <- aggregate2(esoph, c("ncases", "ncontrols"),
                c("alcgp", "tobgp"), c("sum", "mean")))
str(temp2b)

# Dot notation
(temp3a <- aggregate(len ~ ., data = ToothGrowth,
                function(x) cbind(sum(x), mean(x))))
str(temp3a)
(temp3b <- aggregate2(ToothGrowth, "len", ".", c("sum", "mean")))
str(temp3b)
```

CBIND *cbind* data.frames *with different number of rows*

Description

cbind does not work when trying to combine data.frames with differing numbers of rows. This function takes a list of data.frames, identifies how many extra rows are required to make cbind work correctly, and does the combining for you.

Usage

```
CBIND(datalist)
```

Arguments

datalist	A list of data.frames that you want to combine by columns.

Details

The CBIND function also works with nested lists by first "flattening" them using the LinearizeNestedList function by Akhil S Bhel.

Author(s)

Ananda Mahto

See Also

cbind, cbindX, LinearizeNestedList

Examples

```
# Example data
df1 <- data.frame(A = 1:5, B = letters[1:5])
df2 <- data.frame(C = 1:3, D = letters[1:3])
df3 <- data.frame(E = 1:8, F = letters[1:8], G = LETTERS[1:8])

CBIND(list(df1, df2, df3))

# Nested lists:
test1 <- list(list(df1, df2, df3), df1)
str(test1)

CBIND(test1)
```

CensusNames1990 — *List of first names and surnames to generate random names*

Description

This is the default dataset used by the RandomNames function.

Format

A list of first names and surnames, split by gender and how common the first names are.

References

Genealogy Data: Frequently Occurring Surnames from Census 1990–Names Files: http://www.census.gov/genealogy/www/data/1990surnames/names_files.html

concat.split — *Split concatenated cells in a* data.frame

Description

The concat.split function takes a column with multiple values, splits the values into a list or into separate columns, and returns a new data.frame.

Usage

```
concat.split(data, split.col, sep = ",",
  structure = "compact", mode = NULL, drop.col = FALSE,
  fixed = FALSE)
```

Arguments

data	The source data.frame
split.col	The variable that needs to be split; can be specified either by the column number or the variable name.
sep	The character separating each value (defaults to ",").
structure	Can be either "compact", "expanded", or "list". Defaults to "compact". See Details.
mode	Can be either binary or value (where binary is default and it recodes values to 1 or NA, like Boolean data, but without assuming 0 when data is not available). This setting only applies when structure = "expanded"; an warning message will be issued if used with other structures.
drop.col	Logical (whether to remove the original variable from the output or not). Defaults to TRUE.
fixed	Is the input for the sep value *fixed*, or a *regular expression*? See Details.

Details

structure

- `"compact"` creates as many columns as the maximum length of the resulting split. This is the most useful general-case application of this function.
- When the input is numeric, `"expanded"` creates as many columns as the maximum value of the input data. This is most useful when converting to `mode = "binary"`.
- `"list"` creates a single new column that is structurally a `list` within a `data.frame`.

fixed When `structure = "expanded"` or `structure = "list"`, it is possible to supply a a regular expression containing the characters to split on. For example, to split on `","`, `";"`, or `"|"`, you can set `sep = ",|;|\|"` or `sep = "[,;|]"`, and `fixed = FALSE` to split on any of those characters.

Note

If using `structure = "compact"`, the value for `sep` can only be a single character. See the "Advanced Usage" example of how to specify multiple characters for batch conversion of columns.

Author(s)

Ananda Mahto

References

- See `http://stackoverflow.com/q/10100887/1270695`
- The `"condensed"` setting was inspired by an answer from David Winsemius to a question at Stack Overflow. See: `http://stackoverflow.com/a/13924245/1270695`

Examples

```
## Load some data
data(concatenated)
head(concat.test)

# Split up the second column, selecting by column number
head(concat.split(concat.test, 2))

# ... or by name, and drop the offensive first column
head(concat.split(concat.test, "Likes", drop.col = TRUE))

# The "Hates" column uses a different separator
head(concat.split(concat.test, "Hates", sep = ";", drop.col = TRUE))

# You'll get a warning here, when trying to retain the original values
head(concat.split(concat.test, 2, mode = "value", drop.col = TRUE))

# Try again. Notice the differing number of resulting columns
head(concat.split(concat.test, 2, structure = "expanded",
   mode = "value", drop.col = TRUE))
```

```
# Let's try splitting some strings... Same syntax
head(concat.split(concat.test, 3, drop.col = TRUE))

# Split up the "Likes column" into a list variable; retain original column
head(concat.split(concat.test, 2, structure = "list", drop.col=FALSE))

# View the structure of the output for the first 10 rows to verify
# that the new column is a list; note the difference between "Likes"
# and "Likes_list".
str(concat.split(concat.test, 2, structure = "list",
 drop.col=FALSE)[1:10, c(2, 5)])

# ADVANCED USAGE ###

# Show just the first few lines, compact structure
# Note that the split characters must be specified
#   in the same order that lapply will encounter them
head(do.call(cbind,
           c(concat.test[1],
             lapply(1:(ncol(concat.test)-1),
                    function(x) {
                        splitchars = c(",", ",", ";")
                        concat.split(concat.test[-1][x], 1,
                                     splitchars[x],
                                     drop.col=TRUE)
                                     }))))
```

concat.test — *Example dataset with concatenated cells*

Description

This is a sample dataset to demonstrate the different features of the `concat.split` function.

Format

A data.frame in which many columns contain concatenated cells

df.sorter — *Sort a* `data.frame` *by rows or columns*

Description

The `df.sorter` function allows you to sort a `data.frame` by columns or rows or both. You can also quickly subset data columns by using the `var.order` argument.

Usage

```
df.sorter(data, var.order = names(data), col.sort = NULL,
  at.start = TRUE)
```

Arguments

`data`	The source `data.frame`.
`var.order`	The new order in which you want the variables to appear. See Details
`col.sort`	The columns *within* which there is data that need to be sorted. See Details.
`at.start`	Should the pattern matching be from the start of the variable name? Defaults to `TRUE`.

Details

var.order

- Defaults to `names(data)`, which keeps the variables in the original order.
- Variables can be referred to either by a vector of their index numbers or by a vector of the variable name; partial name matching also works, but requires that the partial match identifies similar columns uniquely (see Examples). Basic subsetting can also be done using `var.order` simply by omitting the variables you want to drop.

col.sort

- Defaults to `NULL`, which means no sorting takes place. Variables can be referred to either by a vector of their index numbers or by a vector of the variable names; full names must be provided.

Note

If you are sorting both by variables and within the columns and using numeric indexes as opposed to variable names, the `col.sort` order should be based on the location of the columns in the new `data.frame`, not the original `data.frame`.

Author(s)

Ananda Mahto

Examples

```
# Make up some data
set.seed(1)
dat = data.frame(id = rep(1:5, each=3), times = rep(1:3, 5),
               measure1 = rnorm(15), score1 = sample(300, 15),
               code1 = replicate(15, paste(sample(LETTERS[1:5], 3),
                                            sep="", collapse="")),
               measure2 = rnorm(15), score2 = sample(150:300, 15),
               code2 = replicate(15, paste(sample(LETTERS[1:5], 3),
                                            sep="", collapse="")))
# Preview your data
```

```
dat

# Change the variable order, grouping related columns
# Note that you do not need to specify full variable names,
#    just enough that the variables can be uniquely identified
head(df.sorter(dat, var.order = c("id", "ti", "cod", "mea", "sco")))

# As above, but sorted by 'times' and then 'id'
head(df.sorter(dat,
             var.order = c("id", "tim", "cod", "mea", "sco"),
             col.sort = c(2, 1)))

# Drop 'measure1' and 'measure2', sort by 'times', and 'score1'
head(df.sorter(dat,
             var.order = c("id", "tim", "sco", "cod"),
             col.sort = c(2, 3)))

# Just sort by columns, first by 'times' then by 'id'
head(df.sorter(dat, col.sort = c("times", "id")))

# Pattern matching anywhere in the variable name
head(df.sorter(dat, var.order= "co", at.start=FALSE))
```

`dfcols.list` *Convert the columns of a* `data.frame` *to a* `list`

Description

Sometimes, it is useful to have the columns of a `data.frame` as separate list items or vectors. `unlist` is useful for creating a single vector, but not for creating multiple vectors. The `dfcols.list` function is a simple convenience function that allows for such transformations.

Usage

```
dfcols.list(data, vectorize = TRUE)
```

Arguments

`data`	The input `data.frame`
`vectorize`	Logical. Should the function return a list of single-column `data.frame`s, or a simple `vector` of values? Defaults to `TRUE`.

Author(s)

Ananda Mahto

Examples

```
dat <- data.frame(A = c(1:2), B = c(3:4), C = c(5:6))
dfcols.list(dat)
dfcols.list(dat, vectorize = FALSE)
```

FacsToChars — *Convert all* `factor` *columns to* `character` *columns in a* `data.frame`

Description

Sometimes, we forget to use the `stringsAsFactors` argument when using `read.table` and related functions. By default, R converts character columns to factors. Instead of re-reading the data, the `FacsToChars` function will identify which columns are currently factors, and convert them all to characters.

Usage

```
FacsToChars(mydf, overwrite = FALSE)
```

Arguments

`mydf`	The name of your `data.frame`
`overwrite`	Logical. Should the current object be overwritten? Defaults to `FALSE`

Author(s)

Ananda Mahto

See Also

`read.table`

Examples

```
## Some example data
dat <- data.frame(title = c("title1", "title2", "title3"),
            author = c("author1", "author2", "author3"),
            customerID = c(1, 2, 1))
## Make a copy of dat for later
dat_copy <- dat
str(dat) # current structure
dat2 <- FacsToChars(dat)
str(dat2) # Your new object
str(dat)  # Original object is unaffected

## You can also overwrite the existing object
str(dat_copy) # Before applying the function
FacsToChars(dat_copy, overwrite=TRUE)
```

```
str(dat_copy) # After applying the function
```

getSize *Get the size of multiple objects in your workspace*

Description

This is a convenience wrapper around `object.size` to get the sizes of multiple objects in your workspace. By default, it will list all the objects in your workspaces, but a specific pattern to match can also be specified.

Usage

```
getSize(pattern = NULL, sort.by = "size")
```

Arguments

`pattern`	The pattern to be used by `ls`. Defaults to `"*"`, meaning to match anything.
`sort.by`	Should the output be sorted by object size (`"size"`) or name (`"name"`)? Defaults to `sort.by = "size"`.

Author(s)

Ananda Mahto

See Also

`ls`, `object.size`

Examples

```
AA <- rnorm(10000)
AB <- rnorm(100)
CB <- rnorm(50000)

getSize()
getSize("*B", "name")
getSize("*B", "size")
getSize("^A", "name")
getSize("^A", "size")
```

LinearizeNestedList *Linearize (un-nest) nested lists*

Description

Implements a recursive algorithm to linearize nested `lists` upto any arbitrary level of nesting (limited by R's allowance for recursion-depth). By linearization, it is meant to bring all list branches emanating from any nth-nested trunk upto the top-level trunk such that the return value is a simple non-nested list having all branches emanating from this top-level branch.

Usage

```
LinearizeNestedList(NList, LinearizeDataFrames = FALSE,
  NameSep = "/", ForceNames = FALSE)
```

Arguments

`NList`	The input `list`
`LinearizeDataFrames`	Logical. Should columns in `data.frames` in the list be "linearized" as vectors? Defaults to `FALSE`.
`NameSep`	Character to be used when creating names. Defaults to "/" to mimic directory listings.
`ForceNames`	Logical. Should the present names be discarded and new simplified names be created? Defaults to `FALSE`

Details

Since `data.frames` are essentially `lists` a boolean option is provided to switch on/off the linearization of `data.frames`. This has been found desirable in the author's experience.

Also, one would typically want to preserve names in the lists in a way as to clearly denote the association of any list element to its nth-level history. As such we provide a clean and simple method of preserving names information of list elements. The names at any level of nesting are appended to the names of all preceding trunks using the `NameSep` option string as the seperator. The default "/" has been chosen to mimic the unix tradition of filesystem hierarchies. The default behavior works with existing names at any n-th level trunk, if found; otherwise, coerces simple numeric names corresponding to the position of a list element on the nth-trunk. Note, however, that this naming pattern does not ensure unique names for all elements in the resulting list. If the nested lists had non-unique names in a trunk the same would be reflected in the final list. Also, note that the function does not at all handle cases where *some* names are missing and some are not.

Clearly, preserving the n-level hierarchy of branches in the element names may lead to names that are too long. Often, only the depth of a list element may only be important. To deal with this possibility a boolean option called `ForceNames` has been provided. `ForceNames` shall drop all original names in the lists and coerce simple numeric names which simply indicate the position of an element at the nth-level trunk as well as all preceding trunk numbers.

Author(s)

Akhil S Bhel

References

https://sites.google.com/site/akhilsbehl/geekspace/articles/r/linearize_nested_lists_in_r

See Also

unlist

Examples

```
# Create some sample data
NList <- list(a = "a", # Atom
        b = 1:5, # Vector
        c = data.frame(x = runif(5), y = runif(5)), # Add a data.frame
        d = matrix(runif(4), nrow = 2), # Throw in a matrix for good measure
        e = list(l = list("a", "b"), # Introduce nesting
                 m = list(1:5, 5:10),
                 n = list(list(1), list(2)))) # More nesting

LinearizeNestedList(NList)
LinearizeNestedList(NList, LinearizeDataFrames = TRUE)
LinearizeNestedList(NList, ForceNames = TRUE)
LinearizeNestedList(NList, ForceNames = TRUE, NameSep = "_")
```

`load.scripts.and.data` *Load all script and data files from specified directories*

Description

A convenience function to read all the data files and scripts from specified directories. In general, should only need to specify the directories. Specify directories without trailing slashes.

Usage

```
load.scripts.and.data(path,
  pattern = list(scripts = "*.R$", data = "*.rda$|*.Rdata$"),
  ignore.case = TRUE)
```

Arguments

`path`	A character vector of file paths.
`pattern`	A named list of patterns to match for loading scripts and data files. See "Notes".
`ignore.case`	Logical. Should letter case be considered when searching for data files and script files? Defaults to `FALSE`.

Note

The pre-defined pattern is `list(scripts = "*.R$", data = "*.rda$|*.Rdata$")`. This should match most conventionally used file extensions for R's native script and data files. Alternative patterns should be specified in the same form.

Author(s)

Ananda Mahto

Examples

```
## Not run:
load.scripts.and.data(c("~/Dropbox/Public",
                        "~/Dropbox/Public/R Functions"))

## End(Not run)
```

---

makemeNA — *Make certain values in a* `data.frame` `NA`

Description

Sometimes, after having read in data, one needs to replace certain values by `NA`. One approach is to use `mydf[mydf == "some-character"] <- NA`. However, in many cases that results in a `data.frame` where variables which should be numeric end up as characters or factors if the NA string was a character to begin with. This function is a convenience wrapper around `type.convert` to address such problems.

Usage

```
makemeNA(mydf, NAStrings, fixed = TRUE,
  overwrite = FALSE)
```

Arguments

`mydf`	A `data.frame` in which some values need to be converted to `NA`
`NAStrings`	The values which have been used to represent `NA`
`fixed`	Logical. Is the `NAStrings` argument a fixed character (or vector of characters) or a regular expression? Defaults to `TRUE`.
`overwrite`	Logical. Should the current object be overwritten? Defaults to `FALSE`

Author(s)

Ananda Mahto

See Also

`type.convert`

Examples

```
# Some sample data
temp <- data.frame(
V1 = c(1:3),
V2 = c(1, "*", 3),
V3 = c("a", "*", "c"),
V4 = c(".", "*", "3"))
temp
str(temp)

temp1 <- makemeNA(temp, c("*", "."))
temp1
str(temp1)

# Can make anything NA. Useful for -999 type of NA values
makemeNA(temp, "1")
```

---

multi.freq.table — *Tabulates columns from a* data.frame *containing multiple-response data*

Description

The multi.freq.table function takes a data.frame containing Boolean responses to multiple response questions and tabulates the number of responses by the possible combinations of answers.

Usage

```
multi.freq.table(data, sep = "", boolean = TRUE,
  factors = NULL, NAto0 = TRUE, basic = FALSE,
  dropzero = TRUE, clean = TRUE)
```

Arguments

data	The multiple responses that need to be tabulated.
sep	The desired separator for collapsing the combinations of options; defaults to "" (collapsing with no space between each option name).
boolean	Are you tabulating boolean data (see dat Examples)? Defaults to TRUE.
factors	If you are trying to tabulate non-boolean data, and the data are not factors, you can specify the factors here (see dat2 Examples). Defaults to NULL and is not used when boolean = TRUE.
NAto0	Should NA values be converted to 0? Defaults to TRUE, in which case, the number of valid cases should be the same as the number of cases overall. If set to FALSE, any rows with NA values will be dropped as invalid cases. Only applies when boolean = TRUE.

`basic`	Should a basic table of each item, rather than combinations of items, be created? Defaults to `FALSE`.
`dropzero`	Should combinations with a frequency of zero be dropped from the final table? Defaults to `TRUE`. Does not apply when `boolean = TRUE`.
`clean`	Should the original tabulated data be retained or dropped from the final table? Defaults to `TRUE` (drop). Does not apply when `boolean = TRUE`.

Details

In addition to tabulating the *frequency* (`Freq`), there are two other columns in the output: *Percent of Responses* (`Pct.of.Resp`) and *Percent of Cases* (`Pct.of.Cases`).

Percent of Responses is the frequency divided by the total number of answers provided; this column should sum to 100 table is generated and there are cases where a respondent did not select any option, the Percent of Responses value would be more than 100 frequency divided by the total number of valid cases; this column would most likely sum to more than 100 a basic table is produced since each respondent (case) can select multiple answers, but should sum to 100 other tables.

Author(s)

Ananda Mahto

References

`apply` shortcut for creating the `Combn` column in the output by Justin. See: http://stackoverflow.com/q/11348391/1270695 and http://stackoverflow.com/q/11622660/1270695

Examples

```
## ========================================== ##
## ============= BOOLEAN DATA ============== ##

# Make up some data
set.seed(1)
dat <- data.frame(A = sample(c(0, 1), 20, replace=TRUE),
                 B = sample(c(0, 1, NA), 20,
                            prob=c(.3, .6, .1), replace=TRUE),
                 C = sample(c(0, 1, NA), 20,
                            prob=c(.7, .2, .1), replace=TRUE),
                 D = sample(c(0, 1, NA), 20,
                            prob=c(.3, .6, .1), replace=TRUE),
                 E = sample(c(0, 1, NA), 20,
                            prob=c(.4, .4, .2), replace=TRUE))

# View your data
dat

# How many cases have "NA" values?
table(is.na(rowSums(dat)))

# Apply the function with all defaults accepted
```

```
multi.freq.table(dat)

# Tabulate only on variables "A", "B", and "D", with a different
# separator, keep any zero frequency values, and keeping the
# original tabulations. There are no solitary "D" responses.
multi.freq.table(dat[c(1, 2, 4)], sep="-", dropzero=FALSE, clean=FALSE)

# As above, but without converting "NA" to "0".
# Note the difference in the number of valid cases.
multi.freq.table(dat[c(1, 2, 4)], NAto0=FALSE,
               sep="-", dropzero=FALSE, clean=FALSE)

# View a basic table.
multi.freq.table(dat, basic=TRUE)

## ========================================= ##
## =========== NON-BOOLEAN DATA ============ ##

# Make up some data
dat2 <- structure(list(Reason.1 = c("one", "one", "two", "one", "two",
                                 "three", "one", "one", NA, "two"),
                    Reason.2 = c("two", "three", "three", NA, NA,
                                 "two", "three", "two", NA, NA),
                    Reason.3 = c("three", NA, NA, NA, NA,
                                 NA, NA, "three", NA, NA)),
                    .Names = c("Reason.1", "Reason.2", "Reason.3"),
                    class = "data.frame",
                    row.names = c(NA, -10L))

# View your data
dat2

## Not run: # The following will not work.
# The data are not factored.
multi.freq.table(dat2, boolean=FALSE)
## End(Not run)

# Factor create the factors.
multi.freq.table(dat2, boolean=FALSE,
               factors = c("one", "two", "three"))

# And, a basic table.
multi.freq.table(dat2, boolean=FALSE,
               factors = c("one", "two", "three"),
               basic=TRUE)
```

mv   *Rename an object in the workspace*

Description

Renames an object in the workspace, "removing" the orinal object. This does so without creating a copy of the original object. If an object in the workspace currently exists with the new name specified, the function prompts the user to verify that they want to overwrite that object before proceeding.

Usage

```
mv(currentName, newName)
```

Arguments

`currentName`	The current name of the object
`newName`	The new name for the object

Author(s)

Rolf Turner

References

A good amount of discussion on when R makes a copy in memory in this discussion thread: `https://stat.ethz.ch/pipermail/r-help/2008-March/156028.html`.

Examples

```
x <- runif(1e7)
ls()
x.add <- tracemem(x)
mv(x, y)
identical(x.add, tracemem(y))
ls()
```

`mySOreputation` *Parse your reputation page from any of the Stack Exchange sites*

Description

It is very easy to *view* a detailed account of your reputation at any of the Stack Exchange sites by visiting `http://"sitename"/reputation` (obviously substituting "sitename" for the actual site of interest, for example, `http://stackoverflow.com/reputation`). However, that format is not very user-friendly if you want to do any analysis with it. This function parses that page into an R `data.frame`.

Usage

```
mySOreputation(rep_file)
```

Arguments

`rep_file`	The path to a text version of your reputation page. Windows and Linux users can copy the text on the page with select all + copy, and simply use "clipboard" instead of saving the contents to a local file.

Author(s)

Ananda Mahto

References

Values for the "actions" variable determined after visiting `http://meta.stackoverflow.com/questions/43004/how-do-i-audit-my-reputation/43005#43005`. There is one value not mentioned at that page, coded as `action_id == 99` and `action == Bonus` that corresponds to the bonus that a user gets when they have above a certain reputation and are active on multiple Stack Exchange sites.

Examples

```
## This is a real reputation file,
##    but the "question_id" variable is
##    made up.
rep_file <- system.file("soreputation.txt", package = "mrdwabmisc")
readLines(rep_file, 15)
mydf <- mySOreputation(rep_file)
head(mydf, 15)
str(mydf)
plot(mydf$date, cumsum(mydf$rep_change))

## Not run:
library(xts)
mydfx <- xts(mydf$rep_change, mydf$date)
apply.monthly(mydfx, sum)
plot(apply.monthly(mydfx, sum))

## End(Not run)
```

`RandomNames` *Generate random names*

Description

The `RandomNames` function uses data from the *Genealogy Data: Frequently Occurring Surnames from Census 1990–Names Files* web page to generate a `data.frame` with random names.

Usage

```
RandomNames(N = 100, cat = NULL, gender = NULL,
  MFprob = NULL, dataset = NULL)
```

Arguments

`N`	The number of random names you want. Defaults to 100.
`cat`	Do you want "common" names, "rare" names, names with an "average" frequency, or some combination of these? Should be specified as a character vector (for example, `c("rare", "common")`). Defaults to `NULL`, in which case all names are used as the sample frame.
`gender`	Do you want first names from the "male" dataset, the "female" dataset, or from all available names? Should be specified as a quoted string (for example, `"male"`). Defaults to `NULL`, in which case all available first names are used as the sample frame.
`MFprob`	What proportion of the sample should be male names and what proportion should be female? Specify as a numeric vector that sums to 1 (for example, `c(.6, .4)`). The first number represents the probability of sampling a "male" first name, and the second number represents the probability of sampling a "female" name. This argument is not used if only one gender has been specified in the previous argument. Defaults to `NULL`, in which case, the probability used is `c(.5, .5)`.
`dataset`	What do you want to use as the dataset of names from which to sample? A default dataset is provided that can generate over 400 million unique names. See the "Dataset Details" note for more information.

Note

Dataset Details This function samples from a provided dataset of names. By default, it uses the data from the *Genealogy Data: Frequently Occurring Surnames from Census 1990–Names Files* web page. Those data have been converted to `list` named `"CensusNames1990"` containing three `data.frames` (named `"surnames"`, `"malenames"`, and `"femalenames"`).

Alternatively, you may provide your own data in a `list` formatted according to the following specifications (see the `"myCustomNames"` data in the "Examples*" section). *Please remember that R is case sensitive!*

- This must be a named list with three items: `"surnames"`, `"malenames"`, and `"femalenames"`.
- The contents of each list item is a `data.frame` with at least the following named columns: `"Name"` and `"Category"`.
- Acceptable values for `"Category"` are `"common"`, `"rare"`, and `"average"`.

Author(s)

Ananda Mahto

References

- See http://www.census.gov/genealogy/www/data/1990surnames/names_files.html for source of data.
- Inspired by the online Random Name Generator http://random-name-generator.info/

Examples

```
# Generate 20 random names
RandomNames(N = 20)

# Generate a reproducible list of 100 random names with approximately
#   80% of the names being female names, and 20% being male names.
set.seed(1)
temp <- RandomNames(cat = "common", MFprob = c(.2, .8))
list(head(temp), tail(temp))
table(temp$Gender)

# Cleanup
rm(.Random.seed, envir=globalenv()) # Resets your seed
rm(temp)

# Generate 10 names from the common and rare categories of names
RandomNames(N = 10, cat = c("common", "rare"))

## ===================================== ##
## ======== USING YOUR OWN DATA ======== ##

myCustomNames <- list(
surnames = data.frame(
 Name = LETTERS[1:26],
 Category = c(rep("rare", 10), rep("average", 10), rep("common", 6))),
malenames = data.frame(
 Name = letters[1:10],
 Category = c(rep("rare", 4), rep("average", 4), rep("common", 2))),
femalenames = data.frame(
 Name = letters[11:26],
 Category = c(rep("rare", 8), rep("average", 4), rep("common", 4))))
str(myCustomNames)

RandomNames(N = 15, dataset = myCustomNames)
```

RBIND   *Append* `data.frame`*s by row, even when columns differ*

Description

The default `rbind` function will produce an error if you attempt to use it on `data.frame`s with differing numbers of columns. The `RBIND` function appends a list of `data.frame`s together by row, filling missing columns with `NA`.

Usage

```
RBIND(datalist, keep.rownames = TRUE)
```

Arguments

`datalist`	A `list` of `data.frames` which need to be appended together by row.
`keep.rownames`	Logical. Should the original `rownames` be retained? Defaults to `TRUE`.

Author(s)

Ananda Mahto

See Also

`rbind` and `cbind` for other base R functions to combine `data.frames`; `rbind.fill` for a function with almost identical functionality (does not preserve the `rownames`); `CBIND`.

Examples

```
## Make up some data
x <- data.frame(a = 1:2, b = 2:3, c = 3:4, d = 4:5,
           row.names = c("row_1", "another_row1"))
y <- data.frame(a = c(10, 20), b = c(20, 30), c = c(30, 40),
           row.names=c("row_2", "another_row2"))
z <- data.frame(a = c(11, 21), b = c(22, 32), d = c(33, 43),
           row.names = c("row_3", "another_row3"))
xx <- data.frame(a = 1:2, b = 3:4)
yy <- data.frame(a = 5:6, b = 7:8)
zz <- data.frame(a = 9:10, b = 11:12)
zz2 <- data.frame(a = 9:10, w = 11:12)
temp1 <- list(x, y, z)
temp2 <- list(xx, yy, zz)
temp3 <- list(xx, yy, zz2)
temp4 <- list(x, y, z, xx, yy, zz, zz2)

## Apply the function
RBIND(temp1)
RBIND(temp1, keep.rownames = FALSE)
RBIND(temp2)
RBIND(temp3)
RBIND(temp4)
RBIND(temp4, keep.rownames = FALSE)
```

read.so *Read displayed text at Stack Overflow*

Description

For many questions at Stack Overflow, the question asker does not properly share their question (for example, using `dput` or by sharing some commands to make up the data). Most of the time, you can just copy and paste the text into R using `read.table(text = "clipboard", header = TRUE, stringsAsFactors = F` This function is basically a convenience function for the above.

Usage

```
read.so(sep = "", header = TRUE, out = "mydf")
```

Arguments

`sep`	Most of the time, the code shared is space separated (which is the default for this function). If the separator is any other character, it can be specified here.
`header`	Are headers included?
`out`	Desired output object name. Defaults to `mydf`.

Details

The output of `read.so` is automatically assigned to an object in your workspace called "`mydf`" unless specified using the `out` argument.

Author(s)

Ananda Mahto

See Also

`dput`, `read.table`

Examples

```
## Not run:
## Copy the following text (select and ctrl-c)

# A B
# 1 2
# 3 4
# 5 6

## Now, just type:

read.so()

## End(Not run)
```

round2 *Round numbers the way you learned in school*

Description

The `round2` function rounds numbers in the way you probably learned in school, that is, round up to the next number for values of 5 and above.

Usage

```
round2(x, digits = 0)
```

Arguments

`x`	The number (or vector of numbers) that needs rounding.
`digits`	The number of decimal places in the output.

Details

To reduce bias in rounding, R's `round` function uses a "round-to-even" approach. Still, many people are surprised when they find that R's `round` function will return the same value for `round(1.5)` and `round(2.5)`. This function uses the rounding approach found in most school lessons and in software like Excel to make the results comparable.

Author(s)

Unknown (see "References")

References

Function originally found in an anonymous comment at the Statistically Significant blog. See `http://www.webcitation.org/68djeLBtJ`

See Also

`round`

Examples

```
input <- seq(from = 0.5, by = 1, length.out = 10)
round(input)
round2(input)
round(input/10, digits = 1)
round2(input/10, digits = 1)
```

row.extractor *Extract min/median/max/quantile rows from a* `data.frame`

Description

The `row.extractor` function takes a `data.frame` and extracts rows with the min, median, or max values of a given variable, or extracts rows with specific quantiles of a given variable.

Usage

```
row.extractor(data, extract.by, what = "all")
```

Arguments

`data`	The source `data.frame`.
`extract.by`	The column which will be used as the reference for extraction; can be specified either by the column number or the variable name.
`what`	Options are `"min"` (for all rows matching the minimum value), `"median"` (for the median row or rows), `"max"` (for all rows matching the maximum value), or `"all"` (for min, median, and max); alternatively, a numeric vector can be specified with the desired quantiles, for instance `c(0, .25, .5, .75, 1)`.

Author(s)

Ananda Mahto

References

- `which.quantile` function by cbeleites: http://stackoverflow.com/users/755257/cbeleites
- See: http://stackoverflow.com/q/10256503/1270695

See Also

`min`, `max`, `median`, `which.min`, `which.max`, `quantile`

Examples

```
# Make up some data
set.seed(1)
dat = data.frame(V1 = 1:50, V2 = rnorm(50),
              V3 = round(abs(rnorm(50)), digits=2),
              V4 = sample(1:30, 50, replace=TRUE))
# Get a sumary of the data
summary(dat)

# Get the rows corresponding to the 'min', 'median', and 'max' of 'V4'
row.extractor(dat, 4)
```

```
# Get the 'min' rows only, referenced by the variable name
row.extractor(dat, "V4", "min")

# Get the 'median' rows only. Notice that there are two rows
#    since we have an even number of cases and true median
#    is the mean of the two central sorted values
row.extractor(dat, "V4", "median")

# Get the rows corresponding to the deciles of 'V3'
row.extractor(dat, "V3", seq(0.1, 1, 0.1))
```

`sample.size` *Determine the optimal sample size of a given population*

Description

The sample.size function either calculates the optimum survey sample size when provided with a population size, or the confidence interval of using a certain sample size with a given population. It can be used to generate tables (data.frames) of different combinations of inputs of the following arguments, which can be useful for showing the effect of each of these in sample size calculation.

Usage

```
sample.size(population, samp.size = NULL, c.lev = 95,
  c.int = NULL, what = "sample", distribution = 50)
```

Arguments

`population`	The population size for which a sample size needs to be calculated.
`samp.size`	The sample size. This argument is only used when calculating the confidence interval, and defaults to `NULL`.
`c.lev`	The desired confidence level. Defaults to a reasonable 95%.
`c.int`	The confidence interval. This argument is only used when calculating the sample size. If not specified when calculating the sample size, defaults to 5% and a message is provided indicating this; this is also the default action if `c.int = NULL`.
`what`	Should the function calculate the desired sample size or the confidence interval? Accepted values are `"sample"` and `"confidence"` (quoted), and defaults to `"sample"`.
`distribution`	Response distribution. Defaults to 50% (`distribution = 50`), which will give you the largest sample size.

Note

From a teaching perspective, the function can be used to easily make tables which demonstrate how the sample size or confidence interval change when different inputs change. See the "Advanced Usage" examples. The following formulae were used in this function:

$$ss = \frac{-Z^2 \times p \times (1 - p)}{c^2}$$

$$pss = \frac{ss}{1 + \frac{ss-1}{pop}}$$

Author(s)

Ananda Mahto

References

- See the 2657 Productions News site for how this function progressively developed: http://news.mrdwab.com/2010/09/10/a-sample-size-calculator-function-for-r/
- The `sample.size` function is based on the following formulas from the Creative Research Systems web page *Sample size formulas for our sample size calculator*: http://www.webcitation.org/69kNjMuKe

Examples

```
# What should our sample size be for a population of 300?
# All defaults accepted.
sample.size(population = 300)

# What sample should we take for a population of 300
#   at a confidence level of 97%?
sample.size(population = 300, c.lev = 97)

# What about if we change our confidence interval?
sample.size(population = 300, c.int = 2.5, what = "sample")

# What about if we want to determine the confidence interval
#   of a sample of 140 from a population of 300? A confidence
#   level of 95% is assumed.
sample.size(population = 300, samp.size = 140, what = "confidence")

## ========================================== ##
## =========== ADVANCED USAGE ============== ##

# What should the sample be for populations of 300 to 500 by 50?
sample.size(population=c(300, 350, 400, 450, 500))

# How does varying confidence levels or confidence intervals
#   affect the sample size?
sample.size(population=300,
          c.lev=rep(c(95, 96, 97, 98, 99), times = 3),
          c.int=rep(c(2.5, 5, 10), each=5))

# What is are the confidence intervals for a sample of
#   150, 160, and 170 from a population of 300?
sample.size(population=300,
          samp.size = c(150, 160, 170),
          what = "confidence")
```

`SampleToSum` *Draw a random sample that sums to a specified amount*

Description

This function creates a random sample of numbers drawn from a specified range which sum to a specified amount.

Usage

```
SampleToSum(Target = 100, VecLen = 10, InRange = 1:100,
  Tolerance = 2, writeProgress = NULL)
```

Arguments

`Target`	The desired sum of all the samples
`VecLen`	How many numbers should be in your resulting vector?
`InRange`	What is the acceptable range of values to be sampled from?
`Tolerance`	What is the maximum difference allowed between the target and the sum? Set to "0" to match the target exactly. In general, the difference is within 5 anyway, which is reasonable.
`writeProgress`	If you want a log-file to be written that includes *all* the variations tried before arriving at a vector that satisfies all the user's conditions, specify the output file name (quoted) with this argument. Note that in some cases, this might be quite a large file with tens-of-thousands of lines!

Note

This function can be notoriously slow, particularly if your range is too narrow and your tolerance is too high.

Author(s)

Ananda Mahto

References

This function was written as a response to the following Stack Overflow question: `http://stackoverflow.com/q/14684539/1270695`

See Also

sample, runif

Examples

```
set.seed(1)
SampleToSum()
SampleToSum(Tolerance = 0)
replicate(5,
    SampleToSum(Target = 1376,
                VecLen = 13,
                InRange = 10:200,
                Tolerance = 0),
    simplify = FALSE)
replicate(5,
    SampleToSum(Target = 1376,
                VecLen = 13,
                InRange = 10:200),
    simplify = FALSE)
```

`stratified` *Sample from a* `data.frame` *according to a stratification variable*

Description

The `stratified` function samples from a `data.frame` in which one of the columns can be used as a "stratification" or "grouping" variable. The result is a new `data.frame` with the specified number of samples from each group.

Usage

```
stratified(df, group, size, select = NULL, seed = NULL,
  ...)
```

Arguments

`df`	The source `data.frame`.
`group`	Your grouping variables. Generally, if you are using more than one variable to create your "strata", you should list them in the order of *slowest* varying to *quickest* varying. This can be a vector of names or column indexes.
`size`	The desired sample size. • If `size` is a value between `0` and `1` expressed as a decimal, size is set to be proportional to the number of observations per group. • If `size` is a single positive integer, it will be assumed that you want the same number of samples from each group.

- If `size` is a vector, the function will check to see whether the length of the vector matches the number of groups and use those specified values as the desired sample sizes. The values in the vector should be in the same order as you would get if you tabulated the grouping variable (usually alphabetic order); alternatively, you can name each value to ensure it is properly matched.

`select`	A named list containing levels from the "group" variables in which you are interested. The list names must be present as variable names for the input `data.frame`.
`seed`	The seed that you want to use (using `set.seed` *within* the function, if any. Defaults to `NULL`.
`...`	Further arguments to be passed to the `sample` function.

Note

Slightly different sizes than requested

Because of how computers deal with floating-point arithmetic, and because R uses a "round to even" approach, the size per strata that results when specifying a proportionate sample may be slightly higher or lower per strata than you might have expected.

"Seed" argument

This is different from using `set.seed` before using the function. Setting a seed using this argument is equivalent to using `set.seed` each time that you go to take a sample from a different group (in other words, the same seed is used for each group).

The inclusion of a `seed` argument is mostly a matter of convenience, to be able to have a single seed with which the samples can be verified later. However, by using the `seed` argument, the same seed is used to sample from each group. This may be a problem if there are many groups that have the same number of observations, since it means that the same observation number will be selected from each of those grops. For instance, if group "AA" and "DD" both had the same number of observations (say, 5) and you were sampling 3 cases using a seed of 1, the second, fifth, and fourth observation would be taken from each of those groups. To avoid this, you can set the seed using `set.seed` *before* you run the `stratified` function.

As a user, you need to weigh the benefits and drawbacks of setting the seed *before* running the function as opposed to setting the seed *with* the function. Setting the seed before would be useful if there are several groups with the same number of observations; however, in the slim chance that you need to verify the samples manually, you *may* run into problems.

Author(s)

Ananda Mahto

References

The evolution of this function can be traced at the following links. The version in this package is entirely reworked and does not require an additional package to be loaded.

- `http://news.mrdwab.com/2011/05/15/stratified-random-sampling-in-r-beta/`
- `http://news.mrdwab.com/2011/05/20/stratified-random-sampling-in-r-from-a-data-frame/`

- http://stackoverflow.com/a/9714207/1270695

See Also

strata

Examples

```
# Generate a couple of sample data.frames to play with
set.seed(1)
dat1 <- data.frame(ID = 1:100,
                 A = sample(c("AA", "BB", "CC", "DD", "EE"), 100, replace = TRUE),
                 B = rnorm(100), C = abs(round(rnorm(100), digits=1)),
                 D = sample(c("CA", "NY", "TX"), 100, replace = TRUE),
                 E = sample(c("M", "F"), 100, replace = TRUE))
dat2 <- data.frame(ID = 1:20,
                 A = c(rep("AA", 5), rep("BB", 10),
                       rep("CC", 3), rep("DD", 2)))
# What do the data look like in general?
summary(dat1)
summary(dat2)

# Let's take a 10% sample from all -A- groups in dat1, seed = 1
stratified(dat1, "A", .1, seed = 1)

# Let's take a 10% sample from only "AA" and "BB" groups from -A- in dat1, seed = 1
stratified(dat1, "A", .1, select = list(A = c("AA", "BB")), seed = 1)

# Let's take 5 samples from all -D- groups in dat1,
#   seed = 1, specified by column number
stratified(dat1, group = 5, size = 5, seed = 1)

# Let's take a sample from all -A- groups in dat1, seed = 1,
#   where we specify the number wanted from each group
stratified(dat1, "A", size = c(3, 5, 4, 5, 2), seed = 1)

# Use a two-column strata: -E- and -D-
#   -E- varies more slowly, so it is better to put that first
stratified(dat1, c("E", "D"), size = .15, seed = 1)

# Use a two-column strata (-E- and -D-) but only interested in
#   cases where -E- == "M"
stratified(dat1, c("E", "D"), .15, select = list(E = "M"), seed = 1)

## As above, but where -E- == "M" and -D- == "CA" or "TX"
stratified(dat1, c("E", "D"), .15,
        select = list(E = "M", D = c("CA", "TX")), seed = 1)

# Use a three-column strata: -E-, -D-, and -A-
s.out <- stratified(dat1, c("E", "D", "A"), size = 2, seed = 1)

list(head(s.out), tail(s.out))
```

```
# How many samples were taken from each strata?
table(interaction(s.out[c("E", "D", "A")]))

# Can we verify the message about group sizes?
names(which(table(interaction(dat1[c("E", "D", "A")])) < 2))

names(which(table(interaction(s.out[c("E", "D", "A")])) < 2))
```

`stringseed.sampling` *Use any alphanumeric input as a seed*

Description

The `stringseed.sampling` function is designed as a batch sampling function that allows the user to specify any alphanumeric input as the seed *per sample in the batch*.

Usage

```
stringseed.sampling(seedbase, N, n, write.output = FALSE)
```

Arguments

`seedbase`	A vector of seeds to be used for sampling.
`N`	The "population" from which to draw the sample.
`n`	The desired number of samples.
`write.output`	Logical. Should the output be written to a file? Defaults to `FALSE`. If `TRUE`, a csv file is written with the sample "metadata", and a plain text file is written with the details of the resulting sample. The names of the files written are `"Sample frame generated on {date the script was run} .csv"` and `"Samples generated on {date the script was run} .txt"` and will be found in your current working directory.

Value

This function returns a list with the `class` "stringSeedSampling" and uses a custom `print` method. The list items are:

- `input`: The "metadata" to remind you of your input parameters.
- `samples`: The samples resulting from the specified input parameters. In the case of batch sampling being used, `samples` will be a named list.

Use `print.default(your-object-here)` to view the underlying list.

Author(s)

Ananda Mahto

References

See: http://stackoverflow.com/q/10910698/1270695

See Also

digest

Examples

```
# We'll use a data.frame with a list of village names, the population,
#   and the desired samples as our columns. The function will use the
#   village names to generate a unique seed for each village before
#   drawing the sample.
myListOfPlaces <- data.frame(
villageName = c("Melakkal", "Sholavandan", "T. Malaipatti"),
population = c(120, 130, 140),
requiredSample = c(30, 25, 12))
myListOfPlaces

stringseed.sampling(seedbase = myListOfPlaces$villageName,
                  N = myListOfPlaces$population,
                  n = myListOfPlaces$requiredSample)

# Manual verification of the samples generated for Melakkal village
#   (for which the automatically generated seed was 1331891848)
set.seed(1331891848)
sample(120, 30)

# What about using the function on a single input?
stringseed.sampling("Santa Barbara", 1920, 100)
```

---

subsequence *Identify sequences in a vector*

Description

The subsequence function is like the inverse of rep, and is somewhat related to rle. It detects the sequence in a vector and returns the period of the sequence, the actual sequence, the number of times the sequence is repeated, and optionally, a "Groups" vector the same length as the input vector that can be used as a grouping variable.

Usage

```
subsequence(data, groups = FALSE)
```

Arguments

data	The input vector
groups	Logical. Should the grouping vector be returned?

Author(s)

Ananda Mahto

References

This function was written as an answer to the following Stack Overflow question: http://stackoverflow.com/q/12824931/1270695

See Also

`rep`, `rle`,

Examples

```
## Some sample data
s1a <- rep(c(1, 2, 3), 3)
s1b <- c(s1a, 1)
s2 <- rep(c(1, 2, 3), 50)
s3 <- c(1, 2, 3, 4, 2, 3, 4, 1, 2, 3, 4, 2, 3, 4)
set.seed(1)
s4 <- rep(sample(300, 15), 5)

subsequence(s1a)
## Note the creation of a grouping variable
subsequence(s1b, groups = TRUE)
subsequence(s2)
subsequence(s3)
subsequence(s4)
```

`table2df` *Convert* `table` *objects to* `data.frame`*s*

Description

The `table2df` function takes an object of "class" `table`, `ftable`, and `xtabs` and converts them to `data.frame`s, while retaining as many of the name details as possible.

Usage

```
  table2df(mytable, as.multitable = FALSE,
    direction = "wide")
```

Arguments

`mytable`	The `table` object you want to convert into a `data.frame`. This can be an object in your workspace, or you can make the call to `table`, `ftable`, or `xtabs` as the `mytable` argument to this function.

as.multitable	Logical; defaults to FALSE. Some methods, for instance xtabs and table, will create an array of tables as the output when more than two variables are being tabulated. • If as.multitable is TRUE, the function will return a list of data.frames. • If as.multitable is FALSE, the function will convert the object to an ftable object before performing the transformation.
direction	Can be either "long" or "wide". • If "long", the frequencies will all be tabulated into a single column. This is the same behavior you will generally get if you used as.data.frame on a table object. • If "wide", the tabular format is retained.

Author(s)

Ananda Mahto

References

The expand.grid method for remaking the columns from an ftable was described by Kohske at http://stackoverflow.com/a/6463137/1270695.

See Also

table, ftable, xtabs

Examples

```
# Make up some data:
set.seed(1)
handedness <- data.frame(
gender = sample(c("Female", "Male", "Unknown"), 200, replace = TRUE),
handedness = sample(c("Right", "Left", "Ambidextrous"),
                    200, replace = TRUE, prob = c(.7, .2, .1)),
fav.col = sample(c("Red", "Orange", "Yellow", "Green", "Blue",
                 "Indigo", "Violet", "Black", "White"),
                 200, replace = TRUE),
fav.shape = sample(c("Triangle", "Circle", "Square", "Pentagon", "Hexagon",
                   "Oval", "Octagon", "Rhombus", "Trapezoid"),
                   200, replace = TRUE),
computer = sample(c("Win", "Mac", "Lin"), 200, replace = TRUE,
                  prob = c(.5, .25, .25)))
# Preview the data
list(head(handedness), tail(handedness))

# A very basic table
HT1 <- with(handedness, table(gender, handedness))
HT1
table2df(HT1)
#'table2df(HT1, direction = "long")
```

```
# Another basic table
HT2 <- with(handedness, table(fav.col, computer))
HT2
table2df(HT2)

# This will create multiple tables, one for each possible computer value
HT3 <- with(handedness, table(gender, fav.col, computer))
HT3

# Default settings
table2df(HT3)

# As a list of data.frames
table2df(HT3, as.multitable = TRUE)

# As above, but with the output in long format
#   Only showing the first three lines of each data.frame
lapply(table2df(HT3, as.multitable = TRUE, direction = "long"), head, 3)

# Applied to an ftable
HT4 <- ftable(handedness,
            col.vars="fav.col",
            row.vars=c("gender", "computer"))
HT4
table2df(HT4)

# Applied to a single-row table
table2df(xtabs(breaks ~ wool, warpbreaks))

## ======================================= ##
## ========== OTHER EXAMPLES ============= ##

## Not run:
table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph))
table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph),
       direction = "long")
table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph),
       as.multitable = TRUE, direction = "long")
table2df(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph),
       as.multitable = TRUE, direction = "wide")

## End(Not run)
```

`unbalancedReshape` *Reshape unbalanced wide data to a semi-long data form*

Description

Base R's reshape function cannot easily handle unbalanced wide data. melt, from "reshape2" puts everything into a very long data.frame. This function returns a semi-long data.frame after automatically "balancing" the wide measures.

Usage

```
unbalancedReshape(data, id.vars, sep = ".",
  dropNA = TRUE)
```

Arguments

data	The source data.frame
id.vars	The variables that should be treated as id variables
sep	The separator used. Defaults to .
dropNA	Logical. If all the measure variables for some cases results in a row of NA values, shoud that row be dropped? Defaults to TRUE

Note

Attributes from the output of reshape are retained, allowing you to easily re-convert to the wide form, if necessary.

Author(s)

Ananda Mahto

See Also

reshape, melt

Examples

```
data(concatenated)
temp <- head(do.call(cbind,
           c(concat.test[1],
             lapply(1:(ncol(concat.test)-1),
             function(x) {
             splitchars = c(",", ",", ";")
             concat.split(concat.test[-1][x], 1,
                          splitchars[x],
                          drop.col=TRUE)
                          }))))
temp

## Not run:
## Because the data are unbalanced, we can't just use this
reshape(temp, direction = "long", idvar = "Name",
      varying = 2:ncol(temp), sep = "_")
```

```
## End(Not run)

unbalancedReshape(temp, id.vars = "Name", sep = "_")

## Not run:
## Compare with melt from reshape2
library(reshape2)
melt(temp, id.vars = "Name")

## End(Not run)
```

---

unlistDF *"Unlist" a `list` of `data.frame`s to your workspace*

Description

Many people like the convenience that a `list` of `data.frames` offer; however, some would prefer to have each `data.frame` as a separate object in their workspace. This function "unlists" a `list` of `data.frames`, creating objects named after the `list` and the list item's names (or index position, if names are not available).

Usage

```
unlistDF(mylist)
```

Arguments

`mylist`	The name of your `list` object

Author(s)

Ananda Mahto

See Also

`unlist`

Examples

```
## Some example data
## List with named items
qwerty <- list(A = data.frame(A = 1:2, B = 3:4),
         B = data.frame(C = 5:6, D = 7:8))

## List with unnamed items
ytrewq <- list(data.frame(A = 1:2, B = 3:4),
         data.frame(C = 5:6, D = 7:8))

ls(pattern = "qwer|ytre")
```

```
unlistDF(qwerty)
unlistDF(ytrewq)

ls(pattern = "qwer|ytre")
```

Index