Package 'quest'

September 10, 2021

Type Package

Title Prepare Questionnaire Data for Analysis

Version 0.1.0

Description Offers a suite of functions to prepare questionnaire data for analysis (perhaps other types of data as well). By data preparation, I mean data analytic tasks to get your raw data ready for statistical modeling (e.g., regression). There are functions to investigate missing data, reshape data, validate responses, recode variables, score questionnaires, center variables, aggregate by groups, shift scores (i.e., leads or lags), etc. It provides functions for both single level and multilevel (i.e., grouped) data. With a few exceptions (e.g., ncases()), functions without an ``s'' at the end of their primary word (e.g., center_by()) act on atomic vectors, while functions with an ``s'' at the end of their primary word (e.g., centers_by()) act on multiple columns of a data.frame.

Depends R (>= 4.0.0), datasets, stats, utils, methods

Imports str2str, plyr, car, psych, psychTools

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.1.1

NeedsCompilation no

Author David Disabato [aut, cre] (<`https://orcid.org/0000-0001-7094-4996`>)

Maintainer David Disabato <`ddisab01@gmail.com`>

Repository CRAN

Date/Publication 2021-09-10 11:20:02 UTC

R topics documented:

agg *Aggregate an Atomic Vector by Group*

Description

agg evaluates a function separately for each group and combines the results back together into an atomic vector of data.frame that is returned. Depending on the argument rep, the results of fun are repeated for each element of x in the group (TRUE) or only once for each group (FALSE). Depending on the argument rtn.grp, the return object is a data.frame and the groups within grp are included in the data.frame as columns (TRUE) or the return object is an atomic vector and the groups are the names (FALSE).

Usage

```
agg(x, grp, rep = TRUE, rtn.grp = !rep, sep = "_", fun, ...)
```

Arguments

x	atomic vector.
grp	atomic vector or list of atomic vectors (e.g., data.frame) specifying the groups. The atomic vector(s) must be the length of x or else an error is returned.
rep	logical vector of length 1 specifying whether the result of fun should be repeated for every instance of the group in x (TRUE) or only once for each group (FALSE).
rtn.grp	logical vector of length 1 specifying whether the groups (i.e., grp) should be included in the return object as columns. The default is the opposite of rep as traditionally it is most important to return the group columns when rep = FALSE.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp is a list of atomic vectors (e.g., data.frame) AND rep = FALSE AND rtn.grp = FALSE.
fun	function to use for aggregation. This function is expected to return an atomic vector of length 1.
...	additional named arguments to fun.

Details

If `rep` = TRUE, then `agg` calls `ave`; if `rep` = FALSE, then `agg` calls `aggregate`.

Value

result of `fun` applied to `x` for each group within `grp`. The structure of the return object depends on the arguments `rep` and `rtn.grp`.

then the return object is a data.frame with nrow = `nrow(data)` where the first columns are `grp` and the last column is the result of `fun`. If `grp` is not a list with names, then its colnames will be "Group.1", "Group.2", "Group.3" etc. similar to `aggregate`'s return object. The colname for the result of `fun` will be "x".

If `rep` = TRUE and `rtn.grp` = TRUE:If `rep` = TRUE and `rtn.grp` = FALSE: then the return object is an atomic vector with length = `length(x)` where the values are the result of `fun` and the names = `names(x)`.

If `rep` = FALSE and `rtn.grp` = TRUE: then the return object is a data.frame with nrow = `length(levels(interaction(g` where the first columns are the unique group combinations in `grp` and the last column is the result of `fun`. If `grp` is not a list with names, then its colnames will be "Group.1", "Group.2", "Group.3" etc. similar to `aggregate`'s return object. The colname for the result of `fun` will be "x".

If `rep` = FALSE and codertn.grp = FALSE: then the return object is an atomic vector with length `length(levels(interaction(grp)))` where the values are the result of `fun` and the names are each group value pasted together by `sep` if there are multiple grouping variables within `grp` (i.e., `is.list(grp)` && `length(grp)` > 2).

See Also

`aggs` `agg_dfm` `ave` `aggregate`

Examples

```
# one grouping variable
agg(x = airquality$"Solar.R", grp = airquality$"Month", fun = mean)
agg(x = airquality$"Solar.R", grp = airquality$"Month", fun = mean,
   na.rm = TRUE) # ignoring missing values
agg(x = setNames(airquality$"Solar.R", nm = row.names(airquality)), grp = airquality$"Month",
   fun = mean, na.rm = TRUE) # keeps the names in the return object
agg(x = airquality$"Solar.R", grp = airquality$"Month", rep = FALSE,
   fun = mean, na.rm = TRUE) # do NOT repeat aggregated values
agg(x = airquality$"Solar.R", grp = airquality$"Month", rep = FALSE, rtn.grp = FALSE,
   fun = mean, na.rm = TRUE) # groups are the names of the returned atomic vector

# two grouping variables
tmp_nm <- c("vs","am") # Roxygen2 doesn't like a c() within a []
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = TRUE, fun = sd)
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE,
   fun = sd) # do NOT repeat aggregated values
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE, rtn.grp = FALSE,
   fun = sd) # groups are the names of the returned atomic vector
```

```
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE, rtn.grp = FALSE,
   sep = ".", fun = sd) # change the separater for naming

# error messages
## Not run:
   agg(x = airquality$"Solar.R", grp = mtcars[tmp_nm]) # error returned
   # b/c  atomic vectors within \code{grp} not having the same length as \code{x}

## End(Not run)
```

---

aggs — *Aggregate Data by Group*

Description

aggs evaluates a function separately for each group and combines the results back together into a data.frame that is returned. Depending on `rep`, the results of `fun` are repeated for each element of `data[vrb.nm]` in the group (TRUE) or only once for each group (FALSE). Note, `aggs` evaluates `fun` separately for each variable `vrb.nm` within `data`. If instead, you want to evaluate `fun` for variables as a set `data[vrb.nm]`, then use `agg_dfm`.

Usage

```
aggs(
  data,
  vrb.nm,
  grp.nm,
  rep = TRUE,
  rtn.grp = !rep,
  sep = "_",
  suffix = "_a",
  fun,
  ...
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`rep`	logical vector of length 1 specifying whether the result of `fun` should be repeated for every instance of the group in `data[vrb.nm]` (TRUE) or only once for each group (FALSE).
`rtn.grp`	logical vector of length 1 specifying whether the group columns (i.e., `data[grp.nm]`) should be included in the return object as columns. The default is the opposite of `rep` as traditionally it is most important to return the group columns when `rep` = FALSE.

`sep`	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if `grp.nm` has length > 1 AND `rep` = FALSE AND `rtn.grp` = FALSE.
`suffix`	character vector of length 1 specifying the string to append to the end of the colnames in the return object.
`fun`	function to use for aggregation. This function is expected to return an atomic vector of length 1.
`...`	additional named arguments to `fun`.

Details

If `rep` = TRUE, then `agg` calls `ave`; if `rep` = FALSE, then `agg` calls `aggregate`.

Value

data.frame of aggregated values. If `rep` is TRUE, then nrow = `nrow(data)`. If `rep` = FALSE, then nrow = `length(levels(interaction(data[grp.nm])))`. The names are specified by `paste0(vrb.nm,suffix)`. If `rtn.grp` = TRUE, then the group columns are appended to the begining of the data.frame.

See Also

`agg` `agg_dfm` `ave` `aggregate`

Examples

```
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   fun = mean, na.rm = TRUE)
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rtn.grp = TRUE, fun = mean, na.rm = TRUE) # include the group columns
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, fun = mean, na.rm = TRUE) # do NOT repeat aggregated values
aggs(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, fun = mean, na.rm = TRUE) # with multiple group columns
aggs(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = FALSE, fun = mean, na.rm = TRUE) # without returning groups
```

`agg_dfm` *Data Information by Group*

Description

`agg_dfm` evaluates a function on a set of variables in a data.frame separately for each group and combines the results back together. The `rep` and `rtn.grp` arguments determine exactly how the results are combined together. If `rep` = TRUE, then the result of `fun` is repeated for every row of the group in `data[grp.nm]`; If `rep` = FALSE, then the result of `fun` for each unique combination of `data[grp.nm]` is returned once. If `rtn.grp` = TRUE, then the results are returned in a data.frame where the first columns are the groups from `data[grp.nm]`; If `rtn.grp` = FALSE, then the results are returned in an atomic vector. Note, `agg_dfm` evaluates `fun` on all the variables in `data[vrb.nm]` as a whole, If instead, you want to evaluate `fun` separately for variable `vrb.nm` in `data`, then use `Agg`.

Usage

```
agg_dfm(
  data,
  vrb.nm,
  grp.nm,
  rep = FALSE,
  rtn.grp = !rep,
  sep = ".",
  rtn.result.nm = "result",
  fun,
  ...
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the set of variables to evaluate `fun` on.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`rep`	logical vector of length 1 specifying whether the result of `fun` should be repeated for every instance of the group in `data[vrb.nm]` (TRUE) or only once for each group (FALSE).
`rtn.grp`	logical vector of length 1 specifying whether the group columns (i.e., `data[grp.nm]`) should be included in the return object as columns. The default is the opposite of `rep` as traditionally it is most important to return the group columns when `rep` = FALSE.
`sep`	character vector of length 1 specifying the string to paste the group values together with when there are multiple grouping variables (i.e., `length(grp.nm)` > `1`). Only used if `rep` = FALSE and `rtn.grp` = FALSE.
`rtn.result.nm`	character vector of length 1 specifying the name for the column of results in the return object. Only used if `rtn.grp` = TRUE.
`fun`	function to evaluate each grouping of `data[vrb.nm]` by. This function must return an atomic vector of length 1. If not, then consider using `by2` or `plyr::dlply`.
`...`	additional named arguments to `fun`.

Details

If `rep` = TRUE, then `agg_dfm` calls `ave_dfm`; if `rep` = FALSE, then `agg_dfm` calls `by`. When `rep` = FALSE and `rtn.grp` = TRUE, `agg_dfm` is very similar to `plyr::ddply`; when `rep` = FALSE and `rtn.grp` = FALSE, then `agg_dfm` is very similar to `plyr::daply`.

Value

result of fun applied to each grouping of data[vrb.nm]. The structure of the return object depends on the arguments rep and rtn.grp.

then the return object is a data.frame with nrow = nrow(data) where the first columns are data[grp.nm] and the last column is the result of fun with colname = rtn.result.nm.

If rep = TRUE and rtn.grp = TRUE:If rep = TRUE and rtn.grp = FALSE: then the return object is an atomic vector with length = nrow(data) where the values are the result of fun and the names = row.names(data).

If rep = FALSE and codertn.grp = TRUE: then the return object is a data.frame with nrow = length(levels(interaction(data[grp.nm]))) where the first columns are the unique group combinations in data[grp.nm] and the last column is the result of fun with colname = rtn.result.nm.

If rep = FALSE and codertn.grp = FALSE: then the return object is an atomic vector with length length(levels(interaction(data[grp.nm]))) where the values are the result of fun and the names are each group value pasted together by sep if there are multiple grouping variables (i.e., length(grp.nm) > 2).

See Also

agg aggs by2 ddply daply

Examples

```
### one grouping variable

## by in base R
by(data = airquality[c("Ozone","Solar.R")], INDICES = airquality["Month"],
   simplify = FALSE, FUN = function(dat) cor(dat, use = "complete")[1,2])

## rep = TRUE

# rtn.group = TRUE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = TRUE, rtn.grp = TRUE, fun = function(dat) cor(dat, use = "complete")[1,2])

# rtn.group = FALSE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = TRUE, rtn.grp = FALSE, fun = function(dat) cor(dat, use = "complete")[1,2])

## rep = FALSE

# rtn.group = TRUE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, rtn.grp = TRUE, fun = function(dat) cor(dat, use = "complete")[1,2])
suppressWarnings(plyr::ddply(.data = airquality[c("Ozone","Solar.R","Month")],
   .variables = "Month", .fun = function(dat) cor(dat, use = "complete")[1,2]))

# rtn.group = FALSE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, rtn.grp = FALSE, fun = function(dat) cor(dat, use = "complete")[1,2])
suppressWarnings(plyr::daply(.data = airquality[c("Ozone","Solar.R","Month")],
   .variables = "Month", .fun = function(dat) cor(dat, use = "complete")[1,2]))

### two grouping variables

## by in base R
by(data = mtcars[c("mpg","cyl","disp")], INDICES = mtcars[c("vs","am")],
   FUN = nrow, simplify = FALSE) # with multiple group columns

## rep = TRUE

# rtn.grp = TRUE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = TRUE, rtn.grp = TRUE, fun = nrow)

# rtn.grp = FALSE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = TRUE, rtn.grp = FALSE, fun = nrow)

## rep = FALSE

# rtn.grp = TRUE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = TRUE, fun = nrow)
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = TRUE, rtn.result.nm = "value", fun = nrow)

# rtn.grp = FALSE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = FALSE, fun = nrow)
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = FALSE, sep = "_", fun = nrow)
```

ave_dfm *Repeated Group Statistics for a Data-Frame*

Description

`ave_dfm` evaluates a function on a set of variables `vrb.nm` separately for each group within `grp.nm`. The results are combined back together in line with the rows of `data` similar to `ave`. `ave_dfm` is different than ave or agg because it operates on a data.frame, not an atomic vector.

Usage

```
ave_dfm(data, vrb.nm, grp.nm, fun, ...)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames in `data` specifying the variables to use for the aggregation function `fun`.
`grp.nm`	character vector of colnames in `data` specifying the grouping variables.
`fun`	function that returns an atomic vector of length 1. Probably makes sense to ensure the function always returns the same typeof as well.
`...`	additional named arguments to `fun`.

Value

atomic vector of length = `nrow(data)` providing the result of the function `fun` for the subset of data with that group value (i.e., `data[levels(interaction(data[grp.nm]))[i],vrb.nm]`) for that row.

See Also

`ave` for the same functionality with atomic vector inputs `agg_dfm` for similar functionality with data.frames, but can return the result for each group once rather than repeating the result for each group value in the data.frame

Examples

```
# one grouping variables
ave_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   fun = function(dat) cor(dat, use = "complete")[1,2])

# two grouping variables
ave_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   fun = nrow) # with multiple group columns
```

---

by2 *Apply a Function to Data by Group*

Description

by2 applies a function to data by group and is an alternative to the base R function `by`. The function is apart of the split-apply-combine type of function discussed in the `plyr` R package and is very similar to `dlply`. It splits up one data.frame `.data[.vrb.nm]`into a data.frame for each group in `.data[.grp.nm]`, applies a function `.fun` to each data.frame, and then returns the results as a list with names equal to the group values `unique(interaction(.data[.grp.nm], sep = .sep))`. `by2` is simply `split.data.frame` + `lapply`. Similar to `dlply`, The arguments all start with `.` so that they do not conflict with arguments from the function `.fun`. If you want to apply a function a (atomic) vector rather than data.frame, then use `tapply2`.

Usage

```
by2(.data, .vrb.nm, .grp.nm, .sep = ".", .fun, ...)
```

Arguments

`.data`	data.frame of data.
`.vrb.nm`	character vector specifying the colnames of `.data` to select the set of variables to apply `.fun` to.
`.grp.nm`	character vector specifying the colnames of `.data` to select the grouping variables.
`.sep`	character vector of length 1 specifying the string to combine the group values together with. `.sep` is only used if there are multiple grouping variables (i.e., `length(.grp.nm)` > 1).
`.fun`	function to apply to the set of variables `.data[.vrb.nm]` for each group.
`...`	additional named arguments to pass to `.fun`.

Value

list of objects containing the return object of `.fun` for each group. The names are the unique combinations of the grouping variables (i.e., `unique(interaction(.data[.grp.nm],sep = .sep))`).

See Also

`by` `tapply2` `dlply`

Examples

```

# one grouping variable
by2(mtcars, .vrb.nm = c("mpg","cyl","disp"), .grp.nm = "vs",
   .fun = cov, use = "complete.obs")

# two grouping variables
x <- by2(mtcars, .vrb.nm = c("mpg","cyl","disp"), .grp.nm = c("vs","am"),
   .fun = cov, use = "complete.obs")
print(x)
str(x)

# compare to by
vrb_nm <- c("mpg","cyl","disp") # Roxygen runs the whole script if I put a c() in a []
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
y <- by(mtcars[vrb_nm], INDICES = mtcars[grp_nm],
   FUN = cov, use = "complete.obs", simplify = FALSE)
str(y) # has dimnames rather than names
```

center *Centering and/or Standardizing a Numeric Vector*

Description

`center` centers and/or standardized a numeric vector. It is an alternative to `scale.default` that returns a numeric vector rather than a numeric matrix.

Usage

```
center(x, center = TRUE, scale = FALSE)
```

Arguments

`x`	numeric vector.
`center`	logical vector with length 1 specifying whether grand-mean centering should be done.
`scale`	logical vector with length 1 specifying whether grand-SD scaling should be done.

Details

`center` first coerces `x` to a matrix in preparation for the call to `scale.default`. If the coercion results in a non-numeric matrix (e.g., `x` is a character vector or factor), then an error is returned.

Value

numeric vector of `x` centered and/or standardized with the same names as `x`.

See Also

`centers` `center_by` `centers_by` `scale.default`

Examples

```
center(x = mtcars$"disp")
center(x = mtcars$"disp", scale = TRUE)
center(x = mtcars$"disp", center = FALSE, scale = TRUE)
center(x = setNames(mtcars$"disp", nm = row.names(mtcars)))
```

centers — *Centering and/or Standardizing Numeric Data*

Description

`centers` centers and/or standardized data. It is an alternative to `scale.default` that returns a data.frame rather than a numeric matrix.

Usage

```
centers(data, vrb.nm, center = TRUE, scale = FALSE, suffix)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`center`	logical vector with length 1 specifying whether grand-mean centering should be done.
`scale`	logical vector with length 1 specifying whether grand-SD scaling should be done.
`suffix`	character vector with a single element specifying the string to append to the end of the colnames of the return object. The default depends on the `center` and `scale` arguments: 1)if `center` = TRUE and `scale` = FALSE, then `suffix` = "_c", 2) if `center` = FALSE and `scale` = TRUE, then `suffix` = "_s", 3) if `center` = TRUE and `scale` = TRUE, then `suffix` = "_z", 4) if `center` = FALSE and `scale` = FALSE, then `suffix` = "".

Details

`centers` first coerces `data[vrb.nm]` to a matrix in preparation for the call to `scale.default`. If the coercion results in a non-numeric matrix (e.g., any columns in `data[vrb.nm]` are character vectors or factors), then an error is returned.

Value

data.frame of centered and/or standardized variables with colnames specified by `paste0(vrb.nm,suffix)`.

See Also

`center` `centers_by` `center_by` `scale.default`

Examples

```
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"))
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   scale = TRUE)
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   center = FALSE, scale = TRUE)
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   scale = TRUE, suffix = "_std")
```

---

centers_by *Centering and/or Standardizing Numeric Data by Group*

Description

`centers_by` centers and/or standardized data by group. This is sometimes called group-mean centering and/or group-SD standardizing. The groups can be specified by multiple columns in `data` (e.g., `grp.nm` with length > 1), and `interaction` will be implicitly called to create the groups.

Usage

```
centers_by(data, vrb.nm, grp.nm, center = TRUE, scale = FALSE, suffix)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`center`	logical vector with length 1 specifying whether group-mean centering should be done.
`scale`	logical vector with length 1 specifying whether group-SD scaling should be done.
`suffix`	character vector with a single element specifying the string to append to the end of the colnames of the return object. The default depends on the `center` and `scale` arguments: 1)if `center` = TRUE and `scale` = FALSE, then `suffix` = "_cw", 2) if `center` = FALSE and `scale` = TRUE, then `suffix` = "_sw", 3) if `center` = TRUE and `scale` = TRUE, then `suffix` = "_zw", 4) if `center` = FALSE and `scale` = FALSE, then `suffix` = "".

Details

`centers_by` first coerces `data[vrb.nm]` to a matrix in preparation for the core of the function, which is essentially `lapply(X = split(x = data[vrb.nm], f = data[grp.nm]), FUN = scale.default)` If the coercion results in a non-numeric matrix (e.g., any columns in `data[vrb.nm]` are character vectors or factors), then an error is returned.

Value

data.frame of centered and/or standardized variables by group with colnames specified by `paste0(vrb.nm,suffix)`.

See Also

`center_by` `centers` `center` `scale.default`

Examples

```
ChickWeight2 <- as.data.frame(ChickWeight) # because the "groupedData" class calls
   # `[.groupedData`, which is different than `[.data.frame`
row.names(ChickWeight2) <- as.numeric(row.names(ChickWeight)) / 1000
centers_by(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick")
centers_by(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick",
   scale = TRUE, suffix = "_within")
centers_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), scale = TRUE) # multiple grouping columns
```

`center_by` *Centering and/or Standardizing a Numeric Vector by Group*

Description

`center_by` centers and/or standardized a numeric vector by group. This is sometimes called group-mean centering and/or group-SD standardizing.

Usage

```
center_by(x, grp, center = TRUE, scale = FALSE)
```

Arguments

`x`	numeric vector.
`grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame) containing the groups. They should each have same length as `x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`center`	logical vector with length 1 specifying whether group-mean centering should be done.
`scale`	logical vector with length 1 specifying whether group-SD scaling should be done.

Details

`center_by` first coerces `x` to a matrix in preparation for the core of the function, which is essentially: `lapply(X = split(x = x, f = grp), FUN = scale.default)`. If the coercion results in a non-numeric matrix (e.g., `x` is a character vector or factor), then an error is returned. An error is also returned if `x` and the elements of `grp` do not have the same length.

Value

numeric vector of `x` centered and/or standardized by group with the same names as `x`.

See Also

`centers_by` `center` `centers` `scale.default`

Examples

```
chick_data <- as.data.frame(ChickWeight) # because the "groupedData" class calls
   # `[.groupedData`, which is different than `[.data.frame`
center_by(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]])
center_by(x = setNames(obj = ChickWeight[["weight"]], nm = row.names(ChickWeight)),
   grp = ChickWeight[["Chick"]]) # with names
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like a c() within a []
center_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   scale = TRUE) # multiple grouping vectors
```

---

change *Change Score from a Numeric Vector*

Description

change creates a change score (aka difference score) from a numeric vector. It is assumed that the vector is already sorted by time such that the first element is earliest in time and the last element is the latest in time.

Usage

```
change(x, n, undefined = NA)
```

Arguments

`x`	numeric vector.
`n`	integer vector with length 1. Specifies how the change score is calculated. If `n` is positive, then the change score is calculated from lead - original; if `n` is negative, then the change score is calculated from original - lag. The magnitude of `n` determines how many elements are shifted for the lead/lag within the calculation. If `n` is zero, then change simply returns a vector or zeros. See details of `shift`.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See details of `shift`.

Details

It is recommended to use `L` when specifying `n` to prevent problems with floating point numbers. `shift` tries to circumvent this issue by a call to `round` within `shift` if `n` is not an integer; however that is not a complete fail safe. The problem is that `as.integer(n)` implicit in `shift` truncates rather than rounds. See details of `shift`.

Value

an atomic vector of the same length as x that is the change score. If x and undefined are different typeofs, then the return will be coerced to the most complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

changes change_by changes_by shift

Examples

```
change(x = attitude[[1]], n = -1L) # use L to prevent problems with floating point numbers
change(x = attitude[[1]], n = -2L) # can specify any integer up to the length of `x`
change(x = attitude[[1]], n = +1L) # can specify negative or positive integers
change(x = attitude[[1]], n = +2L, undefined = -999) # user-specified indefined value
change(x = attitude[[1]], n = -2L, undefined = -999) # user-specified indefined value
change(x = attitude[[1]], n = 0L) # returns a vector of zeros
## Not run:
change(x = setNames(object = letters, nm = LETTERS), n = 3L) # character vector returns an error

## End(Not run)
```

---

changes *Change Scores from Numeric Data*

Description

changes creates change scores (aka difference scores) from numeric data. It is assumed that the data is already sorted by time such that the first row is earliest in time and the last row is the latest in time. changes is a multivariate version of change that operates on multiple variabes rather than just one.

Usage

```
changes(data, vrb.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many rows are shifted for the lead/lag within the calculation. See details of shifts.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shifts.

suffix	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_hg",-n), 2) if n > 0, then suffix = paste0("_hd",+n), 3) if n = 0, then suffix = "".

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shifts tries to circumvent this issue by a call to round within shifts if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shifts truncates rather than rounds. See details of shifts.

Value

data.frame of change scores with colnames specified by paste0(vrb.nm,suffix).

See Also

change changes_by change_by shifts

Examples

```
changes(attitude, vrb.nm = names(attitude),
   n = -1L) # use L to prevent problems with floating point numbers
changes(attitude, vrb.nm = names(attitude),
   n = -2L) # can specify any integer up to the length of `x`
changes(attitude, vrb.nm = names(attitude),
   n = +1L) # can specify negative or positive integers
changes(attitude, vrb.nm = names(attitude),
   n = +2L, undefined = -999) # user-specified indefined value
changes(attitude, vrb.nm = names(attitude),
   n = -2L, undefined = -999) # user-specified indefined value
## Not run:
changes(str2str::d2d(InsectSprays), names(InsectSprays),
  n = 3L) # character vector returns an error

## End(Not run)
```

---

changes_by *Change Scores from Numeric Data by Group*

Description

changes_by creates change scores (aka difference scores) from numeric data separately for each group. It is assumed that the data is already sorted within each group by time such that the first row for that group is earliest in time and the last row for that group is the latest in time.

Usage

```
changes_by(data, vrb.nm, grp.nm, n, undefined = NA, suffix)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`n`	integer vector with length 1. Specifies how the change score is calculated. If `n` is positive, then the change score is calculated from lead - original; if `n` is negative, then the change score is calculated from original - lag. The magnitude of `n` determines how many rows are shifted for the lead/lag within the calculation. See details of `shifts_by`.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See details of `shifts_by`.
`suffix`	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the `n` argument: 1) if `n` < 0, then `suffix` = `paste0("_hgw",-n)`, 2) if `n` > 0, then `suffix` = `paste0("_hdw",+n)`, 3) if `n` = 0, then `suffix` = "".

Details

It is recommended to use `L` when specifying `n` to prevent problems with floating point numbers. `shifts_by` tries to circumvent this issue by a call to `round` within `shifts_by` if `n` is not an integer; however that is not a complete fail safe. The problem is that `as.integer(n)` implicit in `shifts_by` truncates rather than rounds. See details of `shifts_by`.

Value

data.frame of change scores by group with colnames specified by `paste0(vrb.nm,suffix)`.

See Also

`change_by` `changes` `change` `shifts_by`

Examples

```
changes_by(data = ChickWeight, vrb.nm = c("weight","Time"), grp.nm = "Chick", n = -1L)
changes_by(data = mtcars, vrb.nm = c("disp","mpg"), grp.nm = c("vs","am"), n = 1L)
changes_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n = 2L) # multiple grouping columns
```

change_by — *Change Scores from a Numeric Vector by Group*

Description

change_by creates a change score (aka difference score) from a numeric vector separately for each group. It is assumed that the vector is already sorted within each group by time such that the first element for that group is earliest in time and the last element for that group is the latest in time.

Usage

```
change_by(x, grp, n, undefined = NA)
```

Arguments

x	numeric vector.
grp	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as x. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many rows are shifted for the lead/lag within the calculation. See details of shift_by.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shift_by.

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift_by tries to circumvent this issue by a call to round within shift_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift_by truncates rather than rounds. See details of shift_by.

Value

an atomic vector of the same length as x that is the change score by group. If x and undefined are different typeofs, then the return will be coerced to the more complex typoof (i.e., complex to simple: character, double, integer, logical).

See Also

changes_by change changes shift_by

Examples

```
change_by(x = ChickWeight[["Time"]], grp = ChickWeight[["Chick"]], n = -1L)
tmp_nm <- c("vs","am") # multiple grouping vectors
change_by(x = mtcars[["disp"]], grp = mtcars[tmp_nm], n = +1L)
tmp_nm <- c("Type","Treatment") # multiple grouping vectors
change_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm], n = 2L)
```

---

colMeans_if *Column Means Conditional on Frequency of Observed Values*

Description

colMeans_if calculates the mean of every column in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that column is less than (or equal to) that specified by ov.min, then NA is returned for that row.

Usage

```
colMeans_if(x, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

x	numeric or logical matrix. If not a matrix, it will be coerced to one.
ov.min	minimum frequency of observed values required per column. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and nrow(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values in a column is exactly equal to ov.min.

Details

Conceptually this function does: apply(X = x, MARGIN = 2, FUN = mean_if, ov.min = ov.min, prop = prop, inclusive = inclusive). But for computational efficiency purposes it does not because then the missing values conditioning would not be vectorized. Instead, it uses colMeans and then inserts NAs for columns that have too few observed values.

Value

numeric vector of length = ncol(x) with names = colnames(x) providing the mean of each column or NA depending on the frequency of observed values.

See Also

colSums_if rowMeans_if rowSums_if colMeans

Examples

```
colMeans_if(airquality)
colMeans_if(x = airquality, ov.min = 150, prop = FALSE)
```

---

colNA *Frequency of Missing Values by Column*

Description

rowNA compute the frequency of missing values in a matrix by column. This function essentially does apply(X = x,MARGIN = 2,FUN = vecNA). It is also used by other functions in the quest package related to missing values (e.g., colMeans_if).

Usage

```
colNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	matrix with any typeof. If not a matrix, it will be coerced to a matrix via as.matrix. The function allows for colnames to carry over for non-matrix objects (e.g., data.frames).
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length = ncol(x), and names = colnames(x) providing the frequency of missing values (or observed values if ov = TRUE) per column. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 1 to nrow(x).

See Also

is.na vecNA rowNA rowsNA

Examples

```
colNA(as.matrix(airquality)) # count of missing values
colNA(as.matrix(airquality), prop = TRUE) # proportion of missing values
colNA(as.matrix(airquality), ov = TRUE) # count of observed values
colNA(as.data.frame(airquality), prop = TRUE, ov = TRUE) # proportion of observed values
```

colSums_if *Column Sums Conditional on Frequency of Observed Values*

Description

`colSums_if` calculates the sum of every column in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that column is less than (or equal to) that specified by `ov.min`, then NA is returned for that column. It also has the option to return a value other than 0 (e.g., NA) when all columns are NA, which differs from `colSums(x,na.rm = TRUE)`.

Usage

```
colSums_if(
  x,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  allNA = NA_real_
)
```

Arguments

`x`	numeric or logical matrix. If not a matrix, it will be coerced to one.
`ov.min`	minimum frequency of observed values required per column. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `nrow(x)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the sum should be calculated if the frequency of observed values in a column is exactly equal to `ov.min`.
`impute`	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of `x[,i]`. If TRUE (default), this will make sums over the same rows with different amounts of observed data comparable.
`allNA`	numeric vector of length 1 specifying what value should be returned for columns that are all NA. This is most applicable when `ov.min = 0` and `inclusive = TRUE`. The default is NA, which differs from `colSums` with `na.rm = TRUE` where 0 is returned. Note, the value is overwritten by NA if the frequency of observed values in that column is less than (or equal to) that specified by `ov.min`.

Details

Conceptually this function does: `apply(X = x,MARGIN = 2,FUN = sum_if,ov.min = ov.min,prop = prop,inclusive = inclusive)`. But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses `colSums` and then inserts NAs for columns that have too few observed values.

Value

numeric vector of length = `ncol(x)` with names = `colnames(x)` providing the sum of each column or NA depending on the frequency of observed values.

See Also

`colMeans_if` `rowSums_if` `rowMeans_if` `colSums`

Examples

```
colSums_if(airquality)
colSums_if(x = airquality, ov.min = 150, prop = FALSE)
x <- data.frame("x" = c(1, 2, NA), "y" = c(1, NA, NA), "z" = c(NA, NA, NA))
colSums_if(x)
colSums_if(x, ov.min = 0)
colSums_if(x, ov.min = 0, allNA = 0)
identical(x = colSums(x, na.rm = TRUE),
   y = colSums_if(x, impute = FALSE, ov.min = 0, allNA = 0)) # identical to
   # colSums(x, na.rm = TRUE)
```

---

decompose — *Decompose a Numeric Vector by Group*

Description

`decompose` decomposes a numeric vector into within-group and between-group components via within-group centering and group-mean aggregation. There is an option to create a grand-mean centered version of the between-person component as well as lead/lag versions of the original vector and the within-group component.

Usage

```
decompose(x, grp, grand = TRUE, n.shift = NULL, undefined = NA)
```

Arguments

`x`	numeric vector.
`grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as `x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`grand`	logical vector of length 1 specifying whether a grand-mean centered version of the the between-group component should be computed.
`n.shift`	integer vector specifying the direction and magnitude of the shifts. For example a one-lead is +1 and a two-lag is -2. See `shift` details.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See `shift` details.

Value

data.frame with nrow = `length(x)` and `row.names` = `names(x)`. The first two columns correspond to the within-group component (i.e., "wth") and the between-group component (i.e., "btw"). If grand = TRUE, then the third column corresponds to the grand-mean centered between-group component (i.e., "btw_c"). If shift != NULL, then the last columns are the shifts indicated by n.shift, where the shifts of `x` are first (i.e., "tot") and then the shifts of the within-group component are second (i.e., "wth"). The naming of the shifted columns is based on the default behavior of `Shift_by`. See the details of `Shift_by`. If you don't like the default naming, then call `Decompose` instead and use the different suffix arguments.

See Also

`decomposes` `center_by` `agg` `shift_by`

Examples

```
# single grouping variable
chick_data <- as.data.frame(ChickWeight) # because the "groupedData" class
   # calls `[.groupedData`, which is different than `[.data.frame`
decompose(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]])
decompose(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]],
   grand = FALSE) # no grand-mean centering
decompose(x = setNames(obj = ChickWeight[["weight"]],
  nm = paste0(row.names(ChickWeight),"_row")), grp = ChickWeight[["Chick"]]) # with names

# multiple grouping variables
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like c() in a []
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm])
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n.shift = 1)
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n.shift = c(+2, +1, -1, -2))
```

`decomposes` *Decompose Numeric Data by Group*

Description

`decomposes` decomposes numeric data by group into within-group and between- group components via within-group centering and group-mean aggregation. There is an option to create a grand-mean centered version of the between-group components.

Usage

```
decomposes(
  data,
  vrb.nm,
  grp.nm,
  grand = TRUE,
  n.shift = NULL,
  undefined = NA,
  suffix.wth = "_w",
  suffix.btw = "_b",
  suffix.grand = "c",
  suffix.lead = "_dw",
  suffix.lag = "_gw"
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`grand`	logical vector of length 1 specifying whether grand-mean centered versions of the the between-group components should be computed.
`n.shift`	integer vector specifying the direction and magnitude of the shifts. For example a one-lead is +1 and a two-lag is -2. See `Shift_by` details.
`undefined`	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in `data[vrb.nm]`). Specifies what to insert for undefined values after the shifting takes place. See details of `Shift_by`.
`suffix.wth`	character vector with a single element specifying the string to append to the end of the within-group component colnames of the return object.
`suffix.btw`	character vector with a single element specifying the string to append to the end of the between-group component colnames of the return object.
`suffix.grand`	character vector with a single element specifying the string to append to the end of the grand-mean centered version of the between-group component colnames of the return object. Note, this is a string that is appended after `suffix.btw` has already been appended.
`suffix.lead`	character vector with a single element specifying the string to append to the end of the positive shift colnames of the return object. Note, `decomposes` will add `abs(n.shift)` to the end of `suffix.lead`.
`suffix.lag`	character vector with a single element specifying the string to append to the end of the negative shift colnames of the return object. Note, `decomposes` will add `abs(n.shift)` to the end of `suffix.lag`.

Value

data.frame with nrow = `nrow(data` and rownames = `row.names(data)`. The first set of columns correspond to the within-group components, followed by the between-group components. If grand = TRUE, then the next set of columns correspond to the grand-mean centered between-group components. If shift != NULL, then the last columns are the shifts by group indicated by n.shift, where the shifts of `data[vrb.nm]` are first and then the shifts of the within-group components are second.

See Also

decompose centers_by aggs shifts_by

Examples

```
ChickWeight2 <- as.data.frame(ChickWeight)
row.names(ChickWeight2) <- as.numeric(row.names(ChickWeight)) / 1000
decomposes(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick")
decomposes(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick",
   suffix.wth = ".wth", suffix.btw = ".btw", suffix.grand = ".grand")
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment")) # multiple grouping columns
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n.shift = 1) # with lead
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"), grp.nm = c("Type","Treatment"),
   n.shift = c(+2, +1, -1, -2)) # with multiple lead/lags
```

---

dum2nom *Dummy Variables to a Nominal Variable*

Description

dum2nom converts dummy variables to a nominal variable. The information from the dummy columns in a data.frame are combined into a character vector (or factor if rtn.fct = TRUE) representing a nominal variable. The unique values of the nominal variable will be the dummy colnames (i.e., dum.nm). Note, *all* the dummy variables associated with a nominal variable are required for this function to work properly. In regression-like models, data analysts will exclude one dummy variable for the category that is the reference group. If d = number of categories in the nominal variable, then that leads to d - 1 dummy variables in the model. dum2nom requires all d dummy variables.

Usage

```
dum2nom(data, dum.nm, yes = 1L, rtn.fct = FALSE)
```

Arguments

data	data.frame of data.
dum.nm	character vector of colnames from data specifying the dummy variables.
yes	atomic vector of length 1 specifying the unique value of the category in each dummy column. This must be the same value for all the dummy variables.
rtn.fct	logical vector of length 1 specifying whether the return object should be a factor (TRUE) or a character vector (FALSE).

Details

dum2nom tests to ensure that data[dum.nm] are indeed a set of dummy columns. First, the dummy columns are expected to have the same mode such that there is one yes unique value across the dummy columns. Second, each row in data[dum.nm] is expected to have either 0 or 1 instance of yes. If there is more than one instance of yes in a row, then an error is returned. If there is 0 instances of yes in a row (e.g., all missing values), NA is returned for that row. Note, any value other than yes will be treated as a no.

Value

character vector (or factor if rtn.fct = TRUE) containing the unique values of dum.nm - one for each dummy variable.

See Also

nom2dum

Examples

```
dum <- data.frame(
   "Quebec_nonchilled" = ifelse(CO2$"Type" == "Quebec" & CO2$"Treatment" == "nonchilled",
      yes = 1L, no = 0L),
   "Quebec_chilled" = ifelse(CO2$"Type" == "Quebec" & CO2$"Treatment" == "chilled",
      yes = 1L, no = 0L),
  "Mississippi_nonchilled" = ifelse(CO2$"Type" == "Mississippi" & CO2$"Treatment" == "nonchilled",
      yes = 1L, no = 0L),
  "Mississippi_chilled" = ifelse(CO2$"Type" == "Mississippi" & CO2$"Treatment" == "chilled",
      yes = 1L, no = 0L)
)
dum2nom(data = dum, dum.nm = names(dum)) # default
dum2nom(data = dum, dum.nm = names(dum), rtn.fct = TRUE) # return as a factor
## Not run:
dum2nom(data = npk, dum.nm = c("N","P","K")) # error due to overlapping dummy columns
dum2nom(data = mtcars, dum.nm = c("vs","am"))# error due to overlapping dummy columns

## End(Not run)
```

---

freq *Univariate Frequency Table*

Description

freq creates univariate frequency tables similar to table. It differs from table by allowing for custom sorting by something other than the alphanumerics of the unique values as well as returning an atomic vector rather than a 1D-array.

Usage

```
freq(
  x,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = "always",
  prop = FALSE,
  sort = "frequency",
  decreasing = TRUE,
  na.last = TRUE
)
```

Arguments

`x`	atomic vector
`exclude`	unique values of `x` to exclude from the returned table. If NULL, then missing values are always included in the returned table. See `table` for documentation on the same argument.
`useNA`	character vector of length 1 specifying how to handle missing values (i.e., whether to include NA as an element in the returned table). There are three options: 1) "no" = don't include missing values in the table, 2) "ifany" = include missing values if there are any, 3) "always" = include missing values in the table, regardless of whether there are any or not. See `table` for documentation on the same argument.
`prop`	logical vector of length 1 specifying whether the returned table should include counts (FALSE) or proportions (TRUE). If NAs are excluded (e.g., useNA = "no" or exclude = c(NA, NaN)), then the proportions will be based on the number of observed elements.
`sort`	character vector of length 1 specifying how the returned table will be sorted. There are three options: 1) "frequency" = the frequency of the unique values in `x`, 2) "position" = the position when each unique value first appears in `x`, 3) "alphanum" = alphanumeric ordering of the unique values in `x` (the sorting used by `table`). When "frequency" is specified and there are ties, then the ties are sorted alphanumerically.
`decreasing`	logical vector of length 1 specifying whether the table should be sorted in decreasing (TRUE) or increasing (FALSE) order.
`na.last`	logical vector of length 1 specifying whether the table should have NAs last or in whatever position they end up at. This argument is only relevant if NAs exist in `x` and are included in the table (e.g., useNA = "always" or exclude = NULL).

Details

The name for the table element giving the frequency of missing values is "(NA)". This is different from `table` where the name is `NA_character_`. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with `NA_character_` names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

numeric vector of frequencies as either counts (if `prop` = FALSE) or proportions (if `prop` = TRUE) with the unique values of `x` as names (missing values have name = "(NA)"). Note, this is different from `table`, which returns a 1D-array and has class "table".

See Also

`freqs` `freq_by` `freqs_by` `table`

Examples

```
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "frequency", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "frequency", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "frequency", decreasing = FALSE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "frequency", decreasing = FALSE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "position", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "position", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "position", decreasing = FALSE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "position", decreasing = FALSE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "alphanum", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "alphanum", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "alphanum", decreasing = FALSE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "alphanum", decreasing = FALSE, na.last = FALSE)
```

`freqs` *Multiple Univariate Frequency Tables*

Description

`freqs` creates a frequency table for a set of variables in a data.frame. Depending on `total`, frequencies for all the variables together can be returned. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freqs(data, vrb.nm, prop = FALSE, useNA = "always", total = "no")
```

Arguments

`data`	data.fame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`prop`	logical vector of length 1 specifying whether the frequencies should be counts (FALSE) or proportions (TRUE). Note, whether the proportions include missing values depends on the `useNA` argument.
`useNA`	character vector of length 1 specifying how missing values should be handled. The three options are 1) "no" = do not include NA frequencies in the return object, 2) "ifany" = only NA frequencies if there are any missing values (in any variable from `data[vrb.nm]`), or 3) "always" = do include NA frequencies regardless of whether there are missing values or not.
`total`	character vector of length 1 specifying whether the frequencies for the set of variables as a whole should be returned. The name "total" refers to tabulating the frequencies for the variables from `data[vrb.nm]` together as a set. The three options are 1) "no" = do not include a row for the total frequencies in the return object, 2) "yes" = do include the total frequencies as the first row in the return object, or 3) "only" = only include the total frequencies as a single row in the return object and do not include rows for each of the individual column frequencies in `data[vrb.nm]`.

Details

`freqs` uses `plyr::rbind.fill` to combine the results from `table` applied to each variable into a single data.frame. If a variable from `data[vrb.nm]` does not have values present in other variables from `data[vrb.nm]`, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from `table` where the name is `NA_character_`. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with `NA_character_` names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

data.frame of frequencies for the variables in `data[vrb.nm]`. Depending on `prop`, the frequencies are either counts (FALSE) or proportions (TRUE). Depending on `total`, the nrow is either 1) `length(vrb.nm)` (if `total` = "no"), 1 + `length(vrb.nm)` (if `total` = "yes"), or 3) 1 (if `total` = "only"). The rownames are `vrb.nm` for each variable in `data[vrb.nm]` and "_total_" for the total row (if present). The colnames are the unique values present in `data[vrb.nm]`, potentially including "(NA)" depending on `useNA`.

See Also

`freq` `freqs_by` `freq_by` `table`

Examples

```
vrb_nm <- str2str::inbtw(names(psych::bfi), "A1","O5")
freqs(data = psych::bfi, vrb.nm = vrb_nm) # default
freqs(data = psych::bfi, vrb.nm = vrb_nm, prop = TRUE) # proportions by row
freqs(data = psych::bfi, vrb.nm = vrb_nm, useNA = "no") # without NA counts
freqs(data = psych::bfi, vrb.nm = vrb_nm, total = "yes") # include total counts
```

---

freqs_by *Multiple Univariate Frequency Tables*

Description

`freqs_by` creates a frequency table for a set of variables in a data.frame by group. Depending on `total`, frequencies for all the variables together can be returned by group. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freqs_by(
  data,
  vrb.nm,
  grp.nm,
  prop = FALSE,
  useNA = "always",
  total = "no",
  sep = "."
)
```

Arguments

`data`	data.fame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`prop`	logical vector of length 1 specifying whether the frequencies should be counts (FALSE) or proportions (TRUE). Note, whether the proportions include missing values depends on the `useNA` argument.
`useNA`	character vector of length 1 specifying how missing values should be handled. The three options are 1) "no" = do not include NA frequencies in the return object, 2) "ifany" = only NA frequencies if there are any missing values (in any variable from `data[vrb.nm]`), or 3) "always" = do include NA frequencies regardless of whether there are missing values or not.
`total`	character vector of length 1 specifying whether the frequencies for the set of variables as a whole should be returned. The name "total" refers to tabulating the frequencies for the variables from `data[vrb.nm]` together as a set. The

three options are 1) "no" = do not include a row for the total frequencies in the return object, 2) "yes" = do include the total frequencies as the first row in the return object, or 3) "only" = only include the total frequencies as a single row in the return object and do not include rows for each of the individual column frequencies in `data[vrb.nm]`.

`sep`	character vector of length 1 specifying the string to combine the group values together with. `sep` is only used if there are multiple grouping variables (i.e., `length(grp.nm)` > 1).

Details

`freqs_by` uses `plyr::rbind.fill` to combine the results from `table` applied to each variable into a single data.frame for each group. If a variable from `data[vrb.nm]` for each group does not have values present in other variables from `data[vrb.nm]` for that group, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from `table` where the name is `NA_character_`. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with `NA_character_` names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

list of data.frames containing the frequencies for the variables in `data[vrb.nm]` by group. The number of list elements are the groups specified by `unique(interaction(data[grp.nm],sep = sep))`. Depending on `prop`, the frequencies are either counts (FALSE) or proportions (TRUE) by group. Depending on `total`, the nrow for each data.frame is either 1) `length(vrb.nm)` (if `total` = "no"), 1 + `length(vrb.nm)` (if `total` = "yes"), or 3) 1 (if `total` = "only"). The rownames are `vrb.nm` for each variable in `data[vrb.nm]` and "_total_" for the total row (if present). The colnames for each data.frame are the unique values present in `data[vrb.nm]`, potentially including "(NA)" depending on `useNA`.

See Also

`freqs` `freq_by` `freqs_by` `table`

Examples

```
vrb_nm <- str2str::inbtw(names(psych::bfi), "A1","O5")
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender") # default
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   prop = TRUE) # proportions by row
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   useNA = "no") # without NA counts
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   total = "yes") # include total counts
freqs_by(data = psych::bfi, vrb.nm = vrb_nm,
   grp.nm = c("gender","education")) # multiple grouping variables
```

freq_by *Univariate Frequency Table By Group*

Description

tables_by creates a frequency table for a set of variables in a data.frame by group. Depending on total, frequencies for all the variables together can be returned by group. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freq_by(
  x,
  grp,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = "always",
  prop = FALSE,
  sort = "frequency",
  decreasing = TRUE,
  na.last = TRUE
)
```

Arguments

x	atomic vector.
grp	atomic vector or list of atomic vectors (e.g., data.frame) specifying the groups. The atomic vector(s) must be the length of x or else an error is returned.
exclude	unique values of x to exclude from the returned table. If NULL, then missing values are always included in the returned table. See table for documentation on the same argument.
useNA	character vector of length 1 specifying how to handle missing values (i.e., whether to include NA as an element in the returned table). There are three options: 1) "no" = don't include missing values in the table, 2) "ifany" = include missing values if there are any, 3) "always" = include missing values in the table, regardless of whether there are any or not. See table for documentation on the same argument.
prop	logical vector of length 1 specifying whether the returned table should include counts (FALSE) or proportions (TRUE). If NAs are excluded (e.g., useNA = "no" or exclude = c(NA, NaN)), then the proportions will be based on the number of observed elements.
sort	character vector of length 1 specifying how the returned table will be sorted. There are three options: 1) "frequency" = the frequency of the unique values in x, 2) "position" = the position when each unique value first appears in x, 3) "alphanum" = alphanumeric ordering of the unique values in x (the sorting used

	by table). When "frequency" is specified and there are ties, then the ties are sorted alphanumerically.
decreasing	logical vector of length 1 specifying whether the table should be sorted in decreasing (TRUE) or increasing (FALSE) order.
na.last	logical vector of length 1 specifying whether the table should have NAs last or in whatever position they end up at. This argument is only relevant if NAs exist in x and are included in the table (e.g., useNA = "always" or exclude = NULL).

Details

tables_by uses plyr::rbind.fill to combine the results from table applied to each variable into a single data.frame for each group. If a variable from data[vrb.nm] for each group does not have values present in other variables from data[vrb.nm] for that group, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from table where the name is NA_character_. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with NA_character_ names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

list of numeric vector of frequencies by group. The number of list elements are the groups specified by unique(interaction(grp,sep = sep)). The frequencies either counts (if prop = FALSE) or proportions (if prop = TRUE) with the unique values of x as names (missing values have name = "(NA)"). Note, this is different from table, which returns a 1D-array and has class "table".

See Also

freq freq_by freqs_by table

Examples

```
x <- freq_by(mtcars$"gear", grp = mtcars$"vs")
str(x)
y <- freq_by(mtcars$"am", grp = mtcars$"vs", useNA = "no")
str(y)
str2str::lv2m(lapply(X = y, FUN = rev), along = 1) # ready to pass to prop.test()
```

---

long2wide *Reshape Multiple Scores From Long to Wide*

Description

long2wide reshapes data from long to wide. This if often necessary to do with multilevel data where variables in the long format seek to be reshaped to multiple sets of variables in the wide format. If only one column needs to be reshaped, then you can use unstack2 or cast - but that does not work for *multiple* columns.

Usage

```
long2wide(
  data,
  vrb.nm,
  grp.nm,
  obs.nm,
  sep = ".",
  colnames.by.obs = TRUE,
  keep.attr = FALSE
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables to be reshaped. In longitudinal panel data, this would be the scores.
`grp.nm`	character vector of colnames from `data` specifying the groups. In longitudnal panel data, this would be the participant ID variable.
`obs.nm`	character vector of length 1 with a colname from `data` specifying the observation within each group. In longitudinal panel data, this would be the time variable.
`sep`	character vector of length 1 specifying the string that separates the name prefix (e.g., score) from it's number suffix (e.g., timepoint). If `sep = ""`, then that implies there is no string separating the name prefix and the number suffix (e.g., "outcome1").
`colnames.by.obs`	logical vector of length 1 specifying whether to sort the return object colnames by the observation label (TRUE) or by the order of `vrb.nm`. See the example at the end of the "MULTIPLE GROUPING VARIABLES" section of the examples.
`keep.attr`	logical vector of length 1 specifying whether to keep the "reshapeWide" attribute (from `reshape`) in the return object.

Details

`long2wide` uses `reshape(direction = "wide")` to reshape the data. It attempts to streamline the task of reshaping long to wide as the `reshape` arguments can be confusing because the same arguments are used for wide vs. long reshaping. See `reshape` if you are curious.

Value

data.frame with nrow equal to `nrow(unique(data[grp.nm]))` and number of reshaped columns equal to `length(vrb.nm) * unique(data[[obs.nm]])`. The colnames will have the structure `paste0(vrb.nm, sep, unique(data[[obs.nm]]))`. The reshaped colnames are sorted by the observation labels if `colnames.by.obs` = TRUE and sorted by `vrb.nm` if `colnames.by.obs` = FALSE. Overall, the columns are in the following order: 1) `grp.nm` of the groups, 2) reshaped columns, 3) additional columns that were not reshaped.

See Also

wide2long reshape unstack2

Examples

```
# SINGLE GROUPING VARIABLE
dat_long <- as.data.frame(ChickWeight) # b/c groupedData class does weird things...
w1 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time") # NAs inserted for missing observations in some groups
w2 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time", sep = "_")
head(w1); head(w2)
w3 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time", sep = "_T", keep.attr = TRUE)
attributes(w3)

# MULTIPLE GROUPING VARIABLE
tmp <- psychTools::sai
grps <- interaction(tmp[1:3], drop = TRUE)
dups <- duplicated(grps)
dat_long <- tmp[!(dups), ] # for some reason there are duplicate groups in the data
vrb_nm <- str2str::pick(names(dat_long), val = c("study","time","id"), not = TRUE)
w4 <- long2wide(data = dat_long, vrb.nm = vrb_nm, grp.nm = c("study","id"),
   obs.nm = "time")
w5 <- long2wide(data = dat_long, vrb.nm = vrb_nm, grp.nm = c("study","id"),
   obs.nm = "time", colnames.by.obs = FALSE) # colnames sorted by `vrb.nm` instead
head(w4); head(w5)
```

---

make.dummy *Make Dummy Columns*

Description

make.dummy creates dummy columns (i.e., dichotomous numeric vectors coded 0 and 1) from logical conditions. If you want to make logical conditions from columns of a data.frame, you will need to call the data.frame and its columns explicitly as this function does not use non-standard evaluation.

Usage

```
make.dummy(..., rtn.lgl = FALSE)
```

Arguments

...	logical conditions that evaluate to logical vectors of the same length. If the logical vectors are not the same length, an error is returned. The names of the arguments are the colnames in the return object. If unnamed, then default R data.frame naming is used, which can get ugly.

`rtn.lgl`	logical vector of length 1 specifying whether the dummy columns should be logical vectors (TRUE) rather than numeric vectors (FALSE).

Value

data.frame of dummy columns based on the logical conditions n `...`. If `rtn.lgl` = TRUE, then the columns are logical vectors. If `out.lgl` = FALSE, then the columns are numeric vectors where 0 = FALSE and 1 = TRUE. The colnames are the names of the arguments in `...`. If not specified, then default data.frame names are created from the logical conditions themselves (which can get ugly).

See Also

`make.dumNA`

Examples

```
make.dummy(attitude$"rating" > 50) # ugly colnames
make.dummy("rating_50plus" = attitude$"rating" > 50,
   "advance_50minus" = attitude$"advance" < 50)
make.dummy("rating_50plus" = attitude$"rating" > 50,
   "advance_50minus" = attitude$"advance" < 50, rtn.lgl = TRUE)
## Not run:
   make.dummy("rating_50plus" = attitude$"rating" > 50,
      "mpg_20plus" = mtcars$"mpg" > 20)

## End(Not run)
```

`make.dumNA` *Make Dummy Columns For Missing Data.*

Description

`make.dumNA` makes dummy columns (i.e., dichomotous numeric vectors coded 0 and 1) for missing data. Each variable is treated in isolation.

Usage

```
make.dumNA(data, vrb.nm, ov = FALSE, rtn.lgl = FALSE, suffix = "_m")
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`ov`	logical vector of length 1 specifying whether the dummy columns should be reverse coded such that missing values = 0/FALSE and observed values = 1/TRUE.
`rtn.lgl`	logical vector of length 1 specifying whether the dummy columns should be logical vectors (TRUE) rather than numeric vectors (FALSE).
`suffix`	character vector of length 1 specifying the string that should be appended to the end of the colnames in the return object.

Value

data.frame of numeric (logical if `rtn.lgl` = TRUE) columns where missing = 1 and observed = 0 (flipped if `ov` = TRUE) for each variable. The colnames are created by `paste0(vrb.nm,suffix)`.

See Also

`make.dummy`

Examples

```
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"))
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"),
   rtn.lgl = TRUE) # logical vectors returned
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"),
   ov = TRUE, suffix = "_o") # 1 = observed value
```

`make.fun_if` *Make a Function Conditional on Frequency of Observed Values*

Description

`make.fun_if` makes a function that evaluates conditional on a specified minimum frequency of observed values. Within the function, if the frequency of observed values is less than (or equal to) `ov.min`, then `false` is returned rather than the return value.

Usage

```
make.fun_if(
  fun,
  ...,
  ov.min.default = 1,
  prop.default = TRUE,
  inclusive.default = TRUE,
  false = NA
)
```

Arguments

`fun`	function that takes an atomic vector as its first argument. The first argument does not have to be named "x" within `fun`, but it will be named "x" in the returned function.
`...`	additional arguments with parameters to `fun`. This would be similar to `impute` in `sum_if`. However in the current version of `make.fun_if`, the parameters you provide will always be used within the returned function and cannot be specified by the user of the returned function. Unfortunately, I cannot figure out how to include user-specified arguments (with defaults) within the returned function other than `ov.min.default`, `prop.default`, and `inclusive.default`.

ov.min.default	numeric vector of length 1 specifying what the default should be for the argument ov.min within the returned function, which specifies the minimum frequency of observed values required. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(x).
prop.default	logical vector of length 1 specifying what the default should be for the argument prop within the returned function, which specifies whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive.default	logical vector of length 1 speicfying what the default should be for the argument inclusive within the returned function, which specifies whether the function should be evaluated if the frequency of observed values is exactly equal to ov.min.
false	vector of length 1 specifying what should be returned if the observed values condition is not met within the returned function. The default is NA. Whatever the value is, it will be coerced to the same mode as x within the returned function.

Value

function that takes an atomic vector x as its first argument, ... as other arguments, ending with ov.min, prop, and inclusive as final arguments with defaults specified by ov.min.default, prop.default, and inclusive.default, respectively.

See Also

sum_if mean_if

Examples

```
# SD
sd_if <- make.fun_if(fun = sd, na.rm = TRUE) # always have na.rm = TRUE
sd_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
sd_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observed values
sd_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min values itself

# skewness
skew_if <- make.fun_if(fun = psych::skew, type = 1) # always have type = 1
skew_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
skew_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observed values
skew_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min values itself

# mode
popular <- function(x) names(sort(table(x), decreasing = TRUE))[1]
popular_if <- make.fun_if(fun = popular) # works with character vectors too
popular_if(x = c(unlist(dimnames(HairEyeColor)), rep.int(x = NA, times = 10)),
   ov.min = .50)
popular_if(x = c(unlist(dimnames(HairEyeColor)), rep.int(x = NA, times = 10)),
   ov.min = .60)
```

`make.product` *Make Product Terms (e.g., interactions)*

Description

`make.product` creates product terms (i.e., interactions) from various components. `make.product` uses `Center` for the optional of centering and/or scaling the predictors and/or moderators before making the product terms.

Usage

```
make.product(
  data,
  x.nm,
  m.nm,
  center.x = FALSE,
  center.m = FALSE,
  scale.x = FALSE,
  scale.m = FALSE,
  suffix.x = "",
  suffix.m = "",
  sep = ":",
  combo = TRUE
)
```

Arguments

`data`	data.frame of data.
`x.nm`	character vector of colnames from `data` specifying the predictor columns.
`m.nm`	character vector of colnames from `data` specifying the moderator columns.
`center.x`	logical vector of length 1 specifying whether the predictor columns should be grand-mean centered before making the product terms.
`center.m`	logical vector of length 1 specifying whether the moderator columns should be grand-mean centered before making the product terms.
`scale.x`	logical vector of length 1 specifying whether the predictor columns should be grand-SD scaled before making the product terms.
`scale.m`	logical vector of length 1 specifying whether the moderator columns should be grand-SD scaled before making the product terms.
`suffix.x`	character vector of length 1 specifying any suffix to add to the end of the predictor colnames `x.nm` when creating the colnames of the return object.

`suffix.m`	character vector of length 1 specifying any suffix to add to the end of the moderator colnames `m.nm` when creating the colnames of the return object.
`sep`	character vector of length 1 specifying the string to connect `x.nm` and `m.nm` when specifying the colnames of the return object.
`combo`	logical vector of length 1 specifying whether all combinations of the predictors and moderators should be calculated or only those in parallel to each other (i.e., `x.nm[i]` and `m.nm[i]`). This argument is only applicable when multiple predictors AND multiple moderators are given.

Value

data.frame with product terms (e.g., interactions) as columns. The colnames are created by `paste(paste0(x.nm,suffix.x),` `= sep)`.

Examples

```
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = "learning", center.x = TRUE, center.m = TRUE,
   suffix.x = "_c", suffix.m = "_c") # with grand-mean centering
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = c("learning","raises"), combo = TRUE) # all possible combinations
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = c("learning","raises"), combo = FALSE) # only combinations "in parallel"
```

mean_if *Mean Conditional on Minimum Frequency of Observed Values*

Description

`mean_if` calculates the mean of a numeric or logical vector conditional on a specified minimum frequency of observed values. If the frequency of observed values is less than (or equal to) `ov.min`, then `NA` is returned rather than the mean.

Usage

```
mean_if(x, trim = 0, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

`x`	numeric or logical vector.
`trim`	numeric vector of length 1 specifying the proportion of values from each end of `x` to trim. Trimmed values are recoded to their endpoint for calculation of the mean. See `mean.default`.
`ov.min`	minimum frequency of observed values required. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(x)`.

`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values is exactly equal to `ov.min`.

Value

numeric vector of length 1 providing the mean of `x` or `NA` conditional on if the frequency of observed data is greater than (or equal to) `ov.min`.

See Also

`mean.default` `sum_if` `make.fun_if`

Examples

```
mean_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
mean_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observe values
mean_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min value itself
mean_if(x = c(TRUE, NA, FALSE, NA),
   ov.min = .50) # works with logical vectors as well as numeric
```

`mode2` *Statistical Mode of a Numeric Vector*

Description

`mode2` calculates the statistical mode - a measure of central tendancy - of a numeric vector. This is in contrast to `mode` in base R, which returns the storage mode of an object. In the case multiple modes exist, the `multiple` argument allows the user to specify if they want the multiple modes returned or just one.

Usage

```
mode2(x, na.rm = FALSE, multiple = FALSE)
```

Arguments

`x`	atomic vector
`na.rm`	logical vector of length 1 specifying if missing values should be removed from `x` before calculating its frequencies.
`multiple`	logical vector of length 1 specifying if multiple modes should be returned in the case they exist. If multiple modes exist and `multiple` = TRUE, the multiple modes will be returned in alphanumeric order. If multiple modes exist and `multiple` = TRUE, the first mode in alphanumeric order will be returned. Note, NA is always last in the alphanumeric order. If only one mode exists, then the `multiple` argument is not used.

Value

atomic vector of the same storage mode as `x` providing the statistical mode(s).

See Also

`freq` `table`

Examples

```
# ONE MODE
vec <- c(7,8,9,7,8,9,9)
mode2(vec)
mode2(vec, multiple = TRUE)

# TWO MODES
vec <- c(7,8,9,7,8,9,8,9)
mode2(vec)
mode2(vec, multiple = TRUE)

# WITH NA
vec <- c(7,8,9,7,8,9,NA,9)
mode2(vec)
mode2(vec, na.rm = TRUE)
vec <- c(7,8,9,7,8,9,NA,9,NA,NA)
mode2(vec)
mode2(vec, multiple = TRUE)
```

`ncases` *Number of Cases in Data*

Description

`ncases` counts how many cases in a data.frame there are that have a specified frequency of observed values across a set of columns. This function is similar to `nrow` and is essentially `partial.cases` + `sum`. The user can have `ncases` return the number of complete cases by calling `ov.min` = 1, `prop` = TRUE, and `inclusive` = TRUE (the default).

Usage

```
ncases(data, vrb.nm = names(data), ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

`data`	data.frame or matrix of data.
`vrb.nm`	a character vector of colnames from `data` specifying the variables.
`ov.min`	minimum frequency of observed values required per row. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(vrb.nm)`.

`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to `ov.min`.

Value

integer vector of length 1 providing the nrow in `data` with the given amount of observed values.

See Also

`partial.cases` `nrow`

Examples

```
vrb_nm <- c("Ozone","Solar.R","Wind")
nrow(airquality[vrb_nm]) # number of cases regardless of missing data
sum(complete.cases(airquality[vrb_nm])) # number of complete cases
ncases(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind"),
   ov.min = 2/3) # number of rows with at least 2 of the 3 variables observed
```

`nom2dum` *Nominal Variable to Dummy Variables*

Description

`nom2dum` converts a nominal variable into a set of dummy variables. There is one dummy variable for each unique value in the nominal variable. Note, base R does this recoding internally through the `model.matrix.default` function, but it is used in the context of regression-like models and it is not clear how to simplify it for general use cases outside that context.

Usage

```
nom2dum(nom, yes = 1L, no = 0L, prefix = "", rtn.fct = FALSE)
```

Arguments

`nom`	character vector (or any atomic vector, including factors, which will be then coerced to a character vector) specifying the nominal variable.
`yes`	atomic vector of length 1 specifying what unique value should represent rows when the nominal category of interest is present. For a traditional dummy variable this value would be 1.
`no`	atomic vector of length 1 specifying what unique value should represent rows when the nominal category of interest is absent. For a traditional dummy variable this value would be 0.
`prefix`	character vector of length 1 specifying the string that should be appended to the beginning of each colname in the return object.
`rtn.fct`	logical vector of length 1 specifying whether the columns of the return object should be factors where the first level is `no` and the second level is `yes`.

Details

Note, that yes and no are assumed to be the same typeof. If they are not, then the columns in the return object will be coerced to the most complex typeof (i.e., most to least: character, double, integer, logical).

Value

data.frame of dummy columns with colnames specified by `paste0(prefix,unique(nom))` and rownames specified by `names(nom)` or default `data.frame` rownames (i.e., c("1","2","3", etc.) if `names(nom)` is `NULL`.

See Also

`model.matrix.default` `dum2nom`

Examples

```
nom2dum(infert$"education") # default
nom2dum(infert$"education", prefix = "edu_") # use of the `prefix` argument
nom2dum(nom = infert$"education", yes = "one", no = "zero",
   rtn.fct = TRUE) # returns factor columns
```

`partial.cases` *Find Partial Cases*

Description

`partial.cases` indicates which cases are at least partially observed, given a specified frequency of observed values across a set of columns. This function builds off `complete.cases`. While `complete.cases` requires completely observed cases, `partial.cases` allows the user to specify the frequency of columns required to be observed. The default arguments are equal to `complete.cases`.

Usage

```
partial.cases(data, vrb.nm, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

`data`	data.frame or matrix of data.
`vrb.nm`	a character vector of colnames from `data` specifying the variables which will be used to determine the partially observed cases.
`ov.min`	minimum frequency of observed values required per row. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(vrb.nm)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to `ov.min`.

Value

logical vector of length = `nrow(data)` with names = `rownames(data)` specifying if the frequency of observed values is greater than (or equal to, if `inclusive` = TRUE) `ov.min`.

See Also

`complete.cases` `rowNA` `ncases`

Examples

```
cases2keep <- partial.cases(data = airquality,
   vrb.nm = c("Ozone","Solar.R","Wind"), ov.min = .66)
airquality2 <- airquality[cases2keep, ] # all cases with 2/3 variables observed
cases2keep <- partial.cases(data = airquality,
   vrb.nm = c("Ozone","Solar.R","Wind"), ov.min = 1, prop = TRUE, inclusive = TRUE)
complete_cases <- complete.cases(airquality)
identical(x = unname(cases2keep),
   y = complete_cases) # partial.cases(ov.min = 1, prop = TRUE,
   # inclusive = TRUE) = complete.cases()
```

---

pomp *Recode a Numeric Vector to Percentage of Maximum Possible (POMP) Units*

Description

`pomp` recodes a numeric vector to percentage of maximum possible (POMP) units. This can be useful when data is measured with arbitrary units (e.g., Likert scale).

Usage

```
pomp(x, mini, maxi, relative = FALSE, unit = 1)
```

Arguments

`x`	numeric vector.
`mini`	numeric vector of length 1 specifying the minimum numeric value possible.
`maxi`	numeric vector of length 1 specifying the maximum numeric value possible.
`relative`	logical vector of length 1 specifying whether relative POMP scores (rather than absolute POMP scores) should be created. If TRUE, then the `mini` and `maxi` arguments are ignored. See details for the distinction between absolute and relative POMP scores.
`unit`	numeric vector of length 1 specifying how many percentage points is desired for the units. Traditionally, POMP scores use `unit` = 1 (default) such that one unit is one percentage point. However, another option is to use `unit` = 100 such that one unit is all 100 percentage points (i.e., proportion of maximum possible). This argument also gives the flexibility of specifying units in between

1 and 100 percentage points. For example, `unit` = 50 would mean that one unit represents going from low (i.e., 25th percentile) to high (i.e., 75th percentile) on the variable.

Details

There are too common approaches to POMP scores: 1) absolute POMP units where the minimum and maximum are the smallest/largest values possible from the measurement instrument (e.g., 1 to 7 on a Likert scale) and 2) relative POMP units where the minimum and maximum are the smallest/largest values observed in the data (e.g., 1.3 to 6.8 on a Likert scale). Both will be correlated perfectly with the original units as they are each linear transformations.

Value

numeric vector from recoding `x` to percentage of maximum possible (pomp) with units specified by `unit`.

See Also

`pomps`

Examples

```
vec <- psych::bfi[[1]]
pomp(x = vec, mini = 1, maxi = 6) # absolute POMP units
pomp(x = vec, relative = TRUE) # relative POMP units
pomp(x = vec, mini = 1, maxi = 6, unit = 100) # unit = 100
pomp(x = vec, mini = 1, maxi = 6, unit = 50) # unit = 50
```

`pomps` *Recode Numeric Data to Percentage of Maximum Possible (POMP) Units*

Description

pomps recodes numeric data to percentage of maximum possible (POMP) units. This can be useful when data is measured with arbitrary units (e.g., Likert scale).

Usage

```
pomps(
  data,
  vrb.nm,
  mini,
  maxi,
  relative = FALSE,
  unit = 1,
  suffix = paste0("_p", unit)
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`mini`	numeric vector of length 1 specifying the minimum numeric value possible. Note, this is assumed to be the same for each variable.
`maxi`	numeric vector of length 1 specifying the maximum numeric value possible. Note, this is assumed to be the same for each variable.
`relative`	logical vector of length 1 specifying whether relative POMP scores (rather than absolute POMP scores) should be created. If TRUE, then the `mini` and `maxi` arguments are ignored. See details for the distinction between absolute and relative POMP scores.
`unit`	numeric vector of length 1 specifying how many percentage points is desired for the units. Traditionally, POMP scores use `unit` = 1 (default) such that one unit is one percentage point. However, another option is to use `unit` = 100 such that one unit is all 100 percentage points (i.e., proportion of maximum possible). This argument also gives the flexibility of specifying units in between 1 and 100 percentage points. For example, `unit` = 50 would mean that one unit represents going from low (i.e., 25th percentile) to high (i.e., 75th percentile) on the variable.
`suffix`	character vector of length 1 specifying the string to add to the end of the column names in the return object.

Details

There are too common approaches to POMP scores: 1) absolute POMP units where the minimum and maximum are the smallest/largest values possible from the measurement instrument (e.g., 1 to 7 on a Likert scale) and 2) relative POMP units where the minimum and maximum are the smallest/largest values observed in the data (e.g., 1.3 to 6.8 on a Likert scale). Both will be correlated perfectly with the original units as they are each linear transformations.

Value

data.frame of variables recoded to percentage of maximum possible (pomp) with units specified by `unit` and names specified by `paste0(vrb.nm,suffix)`.

See Also

`pomp`

Examples

```
vrb_nm <- names(psych::bfi)[grepl(pattern = "A", x = names(psych::bfi))]
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6) # absolute POMP units
pomps(data = psych::bfi, vrb.nm = vrb_nm, relative = TRUE) # relative POMP units
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, unit = 100) # unit = 100
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, unit = 50) # unit = 50
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, suffix = "_pomp")
```

`quest` *Pre-processing Questionnaire Data*

Description

`quest` is a package for pre-processing questionnaire data to get it ready for statistical modeling. It contains functions for investigating missing data (e.g., `rowNA`), reshaping data (e.g., `wide2long`), validating responses (e.g., `revalids`), recoding variables (e.g., `recodes`), scoring (e.g., `scores`), centering (e.g., `centers`), aggregating (e.g., `aggs`), shifting (e.g., `shifts`), etc. Functions whose first phrases end with an `s` are vectorized versions of their functions without an `s` at the end of the first phrase. For example, `center` inputs a (atomic) vector and outputs a atomic vector to center and/or scale a single variable; `centers` inputs a data.frame and outputs a data.frame to center and/or scale multiple variables. Functions that end in `_by` are calculated by group. For example, `center` does grand-mean centering while `center_by` does group-mean centering. Putting the two together, `centers_by` inputs a data.frame and outputs a data.frame to center and/or scale multiple variables by group. Functions that end in `_if` are calculated dependent on the frequency of observed values (aka amount of missing data). The `quest` package uses the `str2str` package internally to convert R objects from one structure to another. See `str2str` for details.

Types of functions

There are two main types of functions. 1) Helper functions that primarily exist to save a few lines of code and are primarily for convenience (e.g., `vecNA`). 2) Functions for wrangling questionnaire data (e.g., `nom2dum`, `reverses`).

Abbreviations

See the table below

names

ovm	observed values
NA	missing values
prop	proportion
sep	separator
vrb	variable
grp	group
id	identifier
rtn	return
fun	function
dfm	data.frame
fct	factor
nom	nominal variable
dum	dummy variable

pomp percentage of maximum possible

std standardize

wth within-groups

btw between-groups

`recode2other` *Recode Unique Values in a Character Vector to 0ther (or NA)*

Description

`recode2other` recodes multiple unique values in a character vector to the same new value (e.g., "other", NA_character_). It's primary use is to recode based on the minimum frequency of the unique values so that low frequency values can be combined into the same category; however, it also allows for recoding particular unique values given by the user (see details). This function is a wrapper for `car::recode`, which can handle general recoding of character vectors.

Usage

```
recode2other(
  x,
  freq.min,
  prop = FALSE,
  inclusive = TRUE,
  other.nm = "other",
  extra.nm = NULL
)
```

Arguments

`x`	character vector. If not a character vector, it will be coarced to one via `as.character`.
`freq.min`	numeric vector of length 1 specifying the minimum frequency of a unique value to keep it unchanged and consequentially recode any unique values with frequencues less than (or equal to) it.
`prop`	logical vector of length 1 specifying if `freq.min` provides the frequency as a count (FALSE) or proportion (TRUE).
`inclusive`	logical vector of length 1 specifying whether the frequency of a unique value exactly equal to `freq.min` should be kept unchanged (and not recoded to `other.nm`).
`other.nm`	character vector of length 1 specifying what value the other unique values should be recoded to. This can be NA_character_ resulting in recoding to a missing value.
`extra.nm`	character vector specifying extra unique values that should be recoded to `other.nm` that are not included based on the minimum frequency from the combination of `freq.min`, `prop`, `inclusive`. The default is NULL, meaning no extra unique values are recoded.

Details

The `extra.nm` argument allows for `recode2other` to be used as simpler function that just recodes particular unique values to the same new value (although arguably this is easier to do using `car::recode` directly). To do so set `freq.min = 0` and provide the unique values to `extra.nm`. Note, that the current version of this function does not allow for NA_character_ to be included in `extra.nm` as it will end up treating it as "NA" (see examples).

Value

character vector of the same length as `x` with unique values with frequency less than `freq.nm` recoded to `other.nm` as well as any unique values in `extra.nm`. While the current version of the function allows for recoding *to* NA values via `other.nm`, it does not allow for recoding *from* NA values via `extra.nm` (see examples).

See Also

recode ifelse

Examples

```
# based on minimum frequency unique values
state_region <- as.character(state.region)
recode2other(state_region, freq.min = 13) # freq.min as a count
recode2other(state_region, freq.min = 0.26, prop = TRUE) # freq.min as a proportion
recode2other(state_region, freq.min = 13, other.nm = "_blank_")
recode2other(state_region, freq.min = 13,
   other.nm = NA) # allows for other.nm to be NA
recode2other(state_region, freq.min = 13,
   extra.nm = "South") # add an extra unique value to recode
recode2other(state_region, freq.min = 13,
   inclusive = FALSE) # recodes "West" to "other"

# based on user given unique values
recode2other(state_region, freq.min = 0,
   extra.nm = c("South","West")) # recodes manually rather than by freq.min
# current version does NOT allow for NA to be a unique value that is converted to other
state_region2 <- c(NA, state_region, NA)
recode2other(state_region2, freq.min = 13) # NA remains in the character vector
recode2other(state_region2, freq.min = 0,
   extra.nm = c("South","West",NA)) # NA remains in the character vector
```

---

recodes *Recode Data*

Description

`recodes` recodes data based on specified recodes using the `car::recode` function. This can be used for numeric or character (including factors) data. See `recode` for details. The `levels` argument from `car::recode` is excluded because there is no easy way to vectorize it when only a subset of the variables are factors.

Usage

```
recodes(data, vrb.nm, recodes, suffix = "_r", as.factor, as.numeric = TRUE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`recodes`	character vector of length 1 specifying the recodes. See details of `recode` for how to use this argument.
`suffix`	character vector of length 1 specifying the string to add to the end of the colnames in the return object.
`as.factor`	logical vector of length 1 specifying if the recoded columns should be returned as factors. The default depends on the column in `data[vrb.nm]`. If the column is a factor, then `as.factor` = TRUE for that column. If the column is not a factor, then `as.factor` = FALSE for that column. Any non-default, specified value for this argument will result in `as.factor` being universally applied to all columns in `data[vrb.nm]`.
`as.numeric`	logical vector of length 1 specifying if the recoded columns should be returned as numeric vectors when possible. This can be useful when having character vectors converted to numeric, such that numbers with typeof character (e.g., "1") will be coerced to typeof numeric (e.g., 1). Note, this argument has no effect on columns in `data[vrb.nm]` which are typeof character and have letters in their values (e.g., "1a"). Note, this argument is often not needed as you can directly recode to a numeric by excluding quotes from the number in the `recodes` argument.

Value

data.frame of recoded variables with colnames specified by `paste0(vrb.nm,suffix)`. In general, the columns of the data.frame are the same typeof as those in `data` except for instances when `as.factor` and/or `as.numeric` change the typeof.

See Also

`recode` `reverses`

Examples

```
recodes(data = psych::bfi, vrb.nm = c("A1","C4","C5","E1","E2","O2","O5"),
   recodes = "1=6; 2=5; 3=4; 4=3; 5=2; 6=1")
re_codes <- "'Quebec' = 'canada'; 'Mississippi' = 'usa'; 'nonchilled' = 'no'; 'chilled' = 'yes'"
```

```
recodes(data = CO2, vrb.nm = c("Type","Treatment"), recodes = re_codes,
   as.factor = FALSE) # convert from factors to characters
```

---

renames *Rename Data Columns from a Codebook*

Description

`renames` renames columns in a data.frame from a codebook. The codebook is assumed to be a list of data.frames containing the old and new column names. See details for how the codebook should be structured. The idea is that the codebook has been imported as an excel workbook with different sets of column renaming information in different workbook sheets. This function is simply a wrapper for `plyr::rename`.

Usage

```
renames(
  data,
  codebook,
  old = 1L,
  new = 2L,
  warn_missing = TRUE,
  warn_duplicated = TRUE
)
```

Arguments

`data`	data.frame of data.
`codebook`	list of data.frames containing the old and new column names.
`old`	numeric vector or character vector of length 1 specifying the position or name of the column in the `codebook` data.frames that contains the old column names present in `data`.
`new`	numeric vector or character vector of length 1 specifying the position or name of the column in the `codebook` data.frames that contains the new column names to rename to in `data`.
`warn_missing`	logical vector of length 1 specifying whether `renames` should return a warning if any old names in `codebook` are not present in `data`.
`warn_duplicated`	logical vector of length 1 specifying whether `renames` should return a warning if the renaming process results in duplicate column names in the return object.

Details

`codebook` is a list of data.frames where one column refers to the old names and another column refers to the new names. Therefore, each row of the data.frames refers to a column in `data`. The position or names of the columns in the `codebook` data.frames that contain the old (i.e., `old`) and new (i.e., `new`) `data` columns must be the same for each data.frame in `codebook`.

Value

data.frame identical to `data` except that the old names in `codebook` have been replaced by the new names in `codebook`.

See Also

`rename`

Examples

```
code_book <- list(
   data.frame("old" = c("rating","complaints"), "new" = c("RATING","COMPLAINTS")),
   data.frame("old" = c("privileges","learning"), "new" = c("PRIVILEGES","LEARNING"))
)
renames(data = attitude, codebook = code_book, old = "old", new = "new")
```

`reorders` *Reorder Levels of Factor Data*

Description

`reorders` re-orders the levels of factor data. The factors are columns in a data.frame where the same reordering scheme is desired. This is often useful before using factor data in a statistical analysis (e.g., `lm`) or a graph (e.g., `ggplot`). It is essentially a vectorized version of `reorder.default`.

Usage

```
reorders(data, fct.nm, ord.nm = NULL, fun, ..., suffix = "_r")
```

Arguments

`data`	data.frame of data.
`fct.nm`	character vector of colnames in `data` that specify the factor columns. If any of the columns specified by `fct.nm` are not factors, then an error is returned.
`ord.nm`	character vector of length 1 or `NULL`. If a character vector of length 1, it is a colname in `data` specifying the column in `data` that will be used in conjunction with `fun` to re-order the factor columns. If `NULL` (default), it is assumed that each factor column itself will be used in conjunction with `fun` to re-order the factor columns.
`fun`	function that will be used to re-order the factor columns. The function is expected to input an atomic vector of length = `nrow(data)` and return an atomic vector of length 1. `fun` is applied to `data[[ord.nm]]` if `ord.nm` is a character vector of length 1 or applied to each column in `data[fct.nm]` if `ord.nm` = `NULL`.
`...`	additional named arguments used by `fun`. For example, if `fun` is `mean`, the user might specify an argument `na.rm` = `TRUE` to set the `na.rm` argument in the `mean` function.
`suffix`	character vector of length 1 specifying the string that will be appended to the end of the colnames in the return object.

Value

data.frame of re-ordered factor columns with colnames = `paste0(fct.nm,suffix)`.

See Also

`reorder.default`

Examples

```
# factor vector
reorder(x = state.region, X = state.region,
   FUN = length) # least frequent to most frequent
reorder(x = state.region, X = state.region,
   FUN = function(vec) {-1 * length(vec)}) # most frequent to least frequent

# data.frame of factors
infert_fct <- infert
fct_nm <- c("education","parity","induced","case","spontaneous")
infert_fct[fct_nm] <- lapply(X = infert[fct_nm], FUN = as.factor)
x <- reorders(data = infert_fct, fct.nm = fct_nm,
   fun = length) # least frequent to most frequent
lapply(X = x, FUN = levels)
y <- reorders(data = infert_fct, fct.nm = fct_nm,
   fun = function(vec) {-1 * length(vec)}) # most frequent to least frequent
lapply(X = y, FUN = levels)
# ord.nm specified as a different column in data.frame
z <- reorders(data = infert_fct, fct.nm = fct_nm, ord.nm = "pooled.stratum",
   fun = mean) # category with highest mean for pooled.stratum to
   # category with lowest mean for pooled.stratum
lapply(X = z, FUN = levels)
```

`revalid` *Recode Invalid Values from a Vector*

Description

`revalid` recodes invalid data to specified values. For example, sometimes invalid values are present in a vector of data (e.g., age = -1). This function allows you to specify which values are possible and will then recode any impossible values to `undefined`. This function is a useful wrapper for the function `car::recode`, tailored for the specific use of recoding invalid values.

Usage

```
revalid(x, valid, undefined = NA)
```

Arguments

`x`	atomic vector.
`valid`	atomic vector of valid values for `x`.
`undefined`	atomic vector of length 1 specifying what the invalid values should be recoded to.

Value

atomic vector with the same typeof as `x` where any values not present in `valid` have been recoded to `undefined`.

See Also

`revalids` `valid_test` `valids_test`

Examples

```
revalid(x = attitude[[1]], valid = 25:75, undefined = NA) # numeric vector
revalid(x = as.character(ToothGrowth[["supp"]]), valid = c('VC'),
   undefined = NA) # character vector
revalid(x = ToothGrowth[["supp"]], valid = c('VC'),
   undefined = NA) # factor
```

`revalids` *Recode Invalid Values from Data*

Description

`revalids` recodes invalid data to specified values. For example, sometimes invalid values are present in a vector of data (e.g., age = -1). This function allows you to specify which values are possible and will then recode any impossible values to `undefined`. `revalids` is simply a vectorized version of `revalid` to more easily revalid multiple columns of a data.frame at the same time.

Usage

```
revalids(data, vrb.nm, valid, undefined = NA, suffix = "_v")
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`valid`	atomic vector of valid values for the data. Note, the valid values must be the same for each variable.
`undefined`	atomic vector of length 1 specifying what the invalid values should be recoded to.
`suffix`	character vector of length 1 specifying the string to add to the end of the colnames in the return object.

Value

data.frame of recoded variables where any values not present in `valid` have been recoded to `undefined` with colnames specified by `paste0(vrb.nm,suffix)`.

See Also

`revalid` `valids_test` `valid_test`

Examples

```
revalids(data = attitude, vrb.nm = names(attitude),
   valid = 25:75) # numeric data
revalids(data = as.data.frame(CO2), vrb.nm = c("Type","Treatment"),
   valid = c('Quebec','nonchilled')) # factors
```

`reverse` *Reverse Code a Numeric Vector*

Description

`reverse` reverse codes a numeric vector based on minimum and maximum values. For example, say numerical values of response options can range from 1 to 4. The function will change 1 to 4, 2 to 3, 3 to 2, and 4 to 1. If there are an odd number of response options, the middle in the sequence will be unchanged.

Usage

```
reverse(x, mini, maxi)
```

Arguments

`x`	numeric vector.
`mini`	numeric vector of length 1 specifying the minimum numeric value.
`maxi`	numeric vector of length 1 specifying the maximum numeric value.

Value

numeric vector that correlates exactly -1 with `x`.

See Also

`reverses` `reverse.code` `recode`

Examples

```
x <- psych::bfi[[1]]
head(x, n = 15)
y <- reverse(x = psych::bfi[[1]], min = 1, max = 6)
head(y, n = 15)
cor(x, y, use = "complete.obs")
```

`reverses` *Reverse Code Numeric Data*

Description

`reverses` reverse codes numeric data based on minimum and maximum values. For example, say numerical values of response options can range from 1 to 4. The function will change 1 to 4, 2 to 3, 3 to 2, and 4 to 1. If there are an odd number of response options, the middle in the sequence will be unchanged.

Usage

```
reverses(data, vrb.nm, mini, maxi, suffix = "_r")
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`mini`	numeric vector of length 1 specifying the minimum numeric value.
`maxi`	numeric vector of length 1 specifying the maximum numeric value.
`suffix`	character vector of length 1 specifying the string to add to the end of the colnames in the return object.

Details

`reverses` is simply a vectorized version of `reverse` to more easily reverse code multiple columns of a data.frame at the same time.

Value

data.frame of reverse coded variables with colnames specified by `paste0(vrb.nm,suffix)`.

See Also

`reverse` `reverse.code` `recodes`

Examples

```
tmp <- !(is.element(el = names(psych::bfi) , set = c("gender","education","age")))
vrb_nm <- names(psych::bfi)[tmp]
reverses(data = psych::bfi, vrb.nm = vrb_nm, mini = 1, maxi = 6)
```

rowMeans_if — *Row Means Conditional on Frequency of Observed Values*

Description

rowMean_if calculates the mean of every row in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that row is less than (or equal to) that specified by ov.min, then NA is returned for that row.

Usage

```
rowMeans_if(x, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

x	numeric or logical matrix. If not a matrix, it will be coerced to one.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and ncol(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values in a row is exactly equal to ov.min.

Details

Conceptually this function does: apply(X = x,MARGIN = 1,FUN = mean_if,ov.min = ov.min,prop = prop,inclusive = inclusive). But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses rowMeans and then inserts NAs for rows that have too few observed values

Value

numeric vector of length = nrow(x) with names = rownames(x) providing the mean of each row or NA depending on the frequency of observed values.

See Also

rowSums_if colMeans_if colSums_if rowMeans

Examples

```
rowMeans_if(airquality)
rowMeans_if(x = airquality, ov.min = 5, prop = FALSE)
```

rowNA *Frequency of Missing Values by Row*

Description

rowNA compute the frequency of missing values in a matrix by row. This function essentially does apply(X = x,MARGIN = 1,FUN = vecNA). It is also used by other functions in the quest package related to missing values (e.g., rowMeans_if).

Usage

```
rowNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	matrix with any typeof. If not a matrix, it will be coerced to a matrix via as.matrix. The argument rownames.force is set to TRUE to allow for rownames to carry over for non-matrix objects (e.g., data.frames).
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length = nrow(x), and names = rownames(x), providing the frequency of missing values (or observed values if ov = TRUE) per row. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 1 to ncol(x).

See Also

is.na vecNA colNA rowsNA

Examples

```
rowNA(as.matrix(airquality)) # count of missing values
rowNA(as.data.frame(airquality)) # with rownames
rowNA(as.matrix(airquality), prop = TRUE) # proportion of missing values
rowNA(as.matrix(airquality), ov = TRUE) # count of observed values
rowNA(as.data.frame(airquality), prop = TRUE, ov = TRUE) # proportion of observed values
```

rowsNA *Frequency of Multiple Sets of Missing Values by Row*

Description

`rowsNA` computes the frequency of missing values for multiple sets of columns from a data.frame. The arguments `prop` and `ov` allow the user to specify if they want to sum or mean the missing values as well as compute the frequency of observed values rather than missing values. This function is essentially a vectorized version of `rowNA` that inputs and outputs a data.frame.

Usage

```
rowsNA(data, vrb.nm.list, prop = FALSE, ov = FALSE)
```

Arguments

`data`	data.frame of data.
`vrb.nm.list`	list where each element is a character vector of colnames in `data` specifying the variables for that set of columns. The names of `vrb.nm.list` will be the colnames of the return object.
`prop`	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
`ov`	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

data.frame with the frequency of missing values (or observed values if `ov` = TRUE) for each set of variables. The names are specified by `names(vrb.nm.list)`; if `vrb.nm.list` does not have any names, then the first element from `vrb.nm.list[[i]]` is used.

See Also

`rowNA` `colNA` `vecNA` `is.na`

Examples

```
vrb_list <- lapply(X = c("O","C","E","A","N"), FUN = function(chr) {
   tmp <- grepl(pattern = chr, x = names(psych::bfi))
   names(psych::bfi)[tmp]
})
rowsNA(data = psych::bfi,
   vrb.nm.list = vrb_list) # names set to first elements in `vrb.nm.list`[[i]]
names(vrb_list) <- paste0(c("O","C","E","A","N"), "_m")
rowsNA(data = psych::bfi, vrb.nm.list = vrb_list) # names set to names(`vrb.nm.list`)
```

rowSums_if *Row Sums Conditional on Frequency of Observed Values*

Description

`rowSums_if` calculates the sum of every row in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that row is less than (or equal to) that specified by `ov.min`, then NA is returned for that row. It also has the option to return a value other than 0 (e.g., NA) when all rows are NA, which differs from `rowSums(x,na.rm = TRUE)`.

Usage

```
rowSums_if(
  x,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  allNA = NA_real_
)
```

Arguments

`x`	numeric or logical matrix. If not a matrix, it will be coerced to one.
`ov.min`	minimum frequency of observed values required per row. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `ncol(x)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the sum should be calculated if the frequency of observed values in a row is exactly equal to `ov.min`.
`impute`	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of `x[i,]`. If TRUE (default), this will make sums over the same columns with different amounts of observed data comparable.
`allNA`	numeric vector of length 1 specifying what value should be returned for rows that are all NA. This is most applicable when `ov.min = 0` and `inclusive = TRUE`. The default is NA, which differs from `rowSums` with `na.rm = TRUE` where 0 is returned. Note, the value is overwritten by NA if the frequency of observed values in that row is less than (or equal to) that specified by `ov.min`.

Details

Conceptually this function is doing: `apply(X = x,MARGIN = 1,FUN = sum_if,ov.min = ov.min,prop = prop,inclusive = inclusive)`. But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses `rowSums` and then inserts NAs for rows that have too few observed values.

Value

numeric vector of length = `nrow(x)` with names = `rownames(x)` providing the sum of each row or NA (or `allNA`) depending on the frequency of observed values.

See Also

`rowMeans_if` `colSums_if` `colMeans_if` `rowSums`

Examples

```
rowSums_if(airquality)
rowSums_if(x = airquality, ov.min = 5, prop = FALSE)
x <- data.frame("x" = c(1, 1, NA), "y" = c(2, NA, NA), "z" = c(NA, NA, NA))
rowSums_if(x)
rowSums_if(x, ov.min = 0)
rowSums_if(x, ov.min = 0, allNA = 0)
identical(x = rowSums(x, na.rm = TRUE),
   y = unname(rowSums_if(x, impute = FALSE, ov.min = 0, allNA = 0))) # identical to
   # rowSums(x, na.rm = TRUE)
```

---

score *Observed Unweighted Scoring of a Set of Variables/Items*

Description

`score` calculates observed unweighted scores across a set of variables/items. If a row's frequency of observed data is less than (or equal to) `ov.min`, then NA is returned for that row. `data[vrb.nm]` is coerced to a matrix before scoring. If the coercion leads to a character matrix, an error is returned.

Usage

```
score(
  data,
  vrb.nm,
  avg = TRUE,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  std = FALSE,
  std.data = std,
  std.score = std
)
```

Arguments

`data`	data.frame or numeric/logical matrix
`vrb.nm`	character vector of colnames in `data` specifying the set of variables/items.
`avg`	logical vector of length 1 specifying whether mean scores (TRUE) or sum scores (FALSE) should be created.
`ov.min`	minimum frequency of observed values required per row. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(vrb.nm)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the score should be calculated (rather than NA) if the frequency of observed values in a row is exactly equal to `ov.min`.
`impute`	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values from each row of `data[vrb.nm]` (i.e., row mean imputation). If TRUE (default), this will make sums over the same rows with different frequencies of missing values comparable. Note, this argument is only used when `avg` = FALSE since when `avg` = TRUE row mean imputation is always done implicitly.
`std`	logical vector of length 1 specifying whether 1) `data[vrb.nm]` should be standardized before scoring and 2) the score standardized after creation. This argument is for convenience as these two standardization processes are often used together. However, this argument will be overwritten by any non-default value for `std.data` and `std.score`.
`std.data`	logical vector of length 1 specifying whether `data[vrb.nm]` should be standardized before scoring.
`std.score`	logical vector of length 1 specifying whether the score should be standardized after creation.

Value

numeric vector of the mean/sum of each row or `NA` if the frequency of observed values is less than (or equal to) `ov.min`. The names are the rownames of `data`.

See Also

`scores` `rowMeans_if` `rowSums_if` `scoreItems`

Examples

```
score(data = attitude, vrb.nm = c("complaints","privileges","learning","raises"))
score(data = attitude, vrb.nm = c("complaints","privileges","learning","raises"),
   std = TRUE) # standardized scoring
score(data = airquality, vrb.nm = c("Ozone","Solar.R","Temp"),
   ov.min = 0.75) # conditional on observed values
```

scores *Observed Unweighted Scoring of Multiple Sets of Variables/Items*

Description

`scores` calculates observed unweighted scores across multiple sets of variables/items. If a row's frequency of observed data is less than (or equal to) `ov.min`, then NA is returned for that row. Each set of variables/items are coerced to a matrix before scoring. If the coercion leads to a character matrix, an error is returned. This can be tested with `lapply(X = vrb.nm.list,FUN = function(nm) is.character(as.matrix(data[nm])))`.

Usage

```
scores(
  data,
  vrb.nm.list,
  avg = TRUE,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  std = FALSE,
  std.data = std,
  std.score = std
)
```

Arguments

`data`	data.frame or numeric/logical matrix
`vrb.nm.list`	list where each element is a character vector of colnames in `data` specifying the variables/items for that score. The names of `vrb.nm.list` will be the names of the scores in the return object.
`avg`	logical vector of length 1 specifying whether mean scores (TRUE) or sum scores (FALSE) should be created.
`ov.min`	minimum frequency of observed values required per row. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(vrb.nm.list[[i]])`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE). If the multiple sets of variables/items contain different numbers of variables, it probably makes the most sense to use the proportion of observed values (TRUE).
`inclusive`	logical vector of length 1 specifying whether the scores should be calculated (rather than NA) if the frequency of observed values in a row is exactly equal to `ov.min`.

`impute`	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values from each row of `data[vrb.nm.list[[i]] ]` (i.e., row mean imputation). If TRUE (default), this will make sums over the same rows with different frequencies of missing values comparable. Note, this argument is only used when `avg` = FALSE since when `avg` = TRUE row mean imputation is always done implicitly.
`std`	logical vector of length 1 specifying whether 1) the variables should be standardized before scoring and 2) the score standardized after creation. This argument is for convenience as these two standardization processes are often used together. However, this argument will be overwritten by any non-default value for `std.data` and `std.score`.
`std.data`	logical vector of length 1 specifying whether the variables/items should be standardized before scoring.
`std.score`	logical vector of length 1 specifying whether the scores should be standardized after creation.

Value

data.frame of mean/sum scores with `NA` for any row with the frequency of observed values less than (or equal to) `ov.min`. The colnames are specified by `names(vrb.nm.list)` and rownames by `row.names(data)`.

See Also

`score` `rowMeans_if` `rowSums_if` `scoreItems`

Examples

```
list_colnames <- list("first" = c("rating","complaints","privileges"),
    "second" = c("learning","raises","critical"))
scores(data = attitude, vrb.nm.list = list_colnames)
list_colnames <- list("first" = c("Ozone","Wind"),
    "second" = c("Solar.R","Temp"))
scores(data = airquality, vrb.nm.list = list_colnames, ov.min = .50,
   inclusive = FALSE) # scoring conditional on observed values
```

`shift` *Shift a Vector (i.e., lag/lead)*

Description

`shift` shifts elements of a vector right ($n < 0$) for lags or left ($n > 0$) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of elements shifted is equal to `abs(n)`. It is assumed that `x` is already sorted by time such that the first element is earliest in time and the last element is the latest in time.

Usage

```
shift(x, n, undefined = NA)
```

Arguments

`x`	atomic vector or list vector.
`n`	integer vector with length 1. Specifies the direction and magnitude of the shift. See details.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See details.

Details

If `n` is negative, then `shift` inserts `undefined` into the first `abs(n)` elements of `x`, shifting all other values of `x` to the right `abs(n)` positions, and then dropping the last `abs(n)` elements of `x` to preserve the original length of `x`. If `n` is positive, then `shift` drops the first `abs(n)` elements of `x`, shifting all other values of `x` left `abs(n)` positions, and then inserts `undefined` into the last `abs(n)` elements of `x` to preserve the original length of `x`. If `n` is zero, then `shift` simply returns `x`.

It is recommended to use `L` when specifying `n` to prevent problems with floating point numbers. `shift` tries to circumvent this issue by a call to `round` within `shift` if `n` is not an integer; however that is not a complete fail safe. The problem is that `as.integer(n)` implicit in `shift` truncates rather than rounds.

Value

an atomic vector of the same length as `x` that is shifted. If `x` and `undefined` are different typeofs, then the return will be coerced to the more complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

`shifts` `shift_by` `shifts_by`

Examples

```
shift(x = attitude[[1]], n = -1L) # use L to prevent problems with floating point numbers
shift(x = attitude[[1]], n = -2L) # can specify any integer up to the length of `x`
shift(x = attitude[[1]], n = +1L) # can specify negative or positive integers
shift(x = attitude[[1]], n = +2L, undefined = -999) # user-specified indefined value
shift(x = setNames(object = letters, nm = LETTERS), n = 3L) # names are kept
```

shifts — *Shift Data (i.e., lag/lead)*

Description

`shifts` shifts rows of data down (`n` < 0) for lags or up (`n` > 0) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of rows shifted is equal to `abs(n)`. It is assumed that `data[vrb.nm]` is already sorted by time such that the first row is earliest in time and the last row is the latest in time.

Usage

```
shifts(data, vrb.nm, n, undefined = NA, suffix)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`n`	integer vector of length 1. Specifies the direction and magnitude of the shift. See details.
`undefined`	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in `data[vrb.nm]`). Specifies what to insert for undefined values after the shifting takes place. See details.
`suffix`	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the `n` argument: 1) if `n` < 0, then `suffix` = `paste0("_g",-n)`, 2) if `n` > 0, then `suffix` = `paste0("_d",+n)`, 3) if `n` = 0, then `suffix` = "".

Details

If `n` is negative, then `shifts` inserts `undefined` into the first `abs(n)` rows of `data[vrb.nm]`, shifting all other rows of `x` down `abs(n)` positions, and then dropping the last `abs(n)` row of `data[vrb.nm]` to preserve the original nrow of `data`. If `n` is positive, then `shifts` drops the first `abs(n)` rows of `x`, shifting all other rows of `data[vrb.nm]` up `abs(n)` positions, and then inserts `undefined` into the last `abs(n)` rows of `x` to preserve the original length of `data`. If `n` is zero, then `shifts` simply returns `data[vrb.nm]`.

It is recommended to use `L` when specifying `n` to prevent problems with floating point numbers. `shifts` tries to circumvent this issue by a call to `round` within `shifts` if `n` is not an integer; however that is not a complete fail safe. The problem is that `as.integer(n)` implicit in `shifts` truncates rather than rounds.

Value

data.frame of shifted data with colnames specified by `suffix`.

See Also

shift shifts_by shift_by

Examples

```
shifts(data = attitude, vrb.nm = colnames(attitude), n = -1L)
shifts(data = mtcars, vrb.nm = colnames(mtcars), n = 2L)
```

---

shifts_by *Shift Data (i.e., lag/lead) by Group*

Description

shifts_by shifts rows of data down (n < 0) for lags or up (n > 0) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of rows shifted is equal to abs(n). It is assumed that data[vrb.nm] is already sorted within each group by time such that the first row for that group is earliest in time and the last row for that group is the latest in time. The groups can be specified by multiple columns in data (e.g., grp.nm with length > 1), and interaction will be implicitly called to create the groups.

Usage

```
shifts_by(data, vrb.nm, grp.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
n	integer vector of length 1. Specifies the direction and magnitude of the shift. See details.
undefined	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in data[vrb.nm]). Specifies what to insert for undefined values after the shifting takes place. See details.
suffix	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_gw", -n), 2) if n > 0, then suffix = paste0("_dw", +n), 3) if n = 0, then suffix = "".

Details

If n is negative, then shifts_by inserts undefined into the first abs(n) rows of data[vrb.nm] for each group, shifting all other rows of x down abs(n) positions, and then dropping the last abs(n) row of data[vrb.nm] to preserve the original nrow of each group. If n is positive, then shifts_by drops the first abs(n) rows of x for each group, shifting all other rows of data[vrb.nm] up abs(n) positions, and then inserts `undefined` into the last `abs(n)` rows of `x` to preserve the original length of each group. If `n` is zero, then `shifts_by` simply returns `data[vrb.nm]`.

It is recommended to use `L` when specifying `n` to prevent problems with floating point numbers. `shifts_by` tries to circumvent this issue by a call to `round` within `shifts_by` if `n` is not an integer; however that is not a complete fail safe. The problem is that `as.integer(n)` implicit in `shifts_by` truncates rather than rounds.

Value

data.frame of shifted data by group with colnames specified by `suffix`.

See Also

`shift_by` `shifts` `shift`

Examples

```
shifts_by(data = ChickWeight, vrb.nm = c("weight","Time"), grp.nm = "Chick", n = -1L)
shifts_by(data = mtcars, vrb.nm = c("disp","mpg"), grp.nm = c("vs","am"), n = 1L)
shifts_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n = 2L) # multiple grouping columns
```

`shift_by` *Shift a Vector (i.e., lag/lead) by Group*

Description

`shift_by` shifts elements of a vector right (`n` < 0) for lags or left (`n` > 0) for leads by group, replacing the undefined data with a user-defined value (e.g., NA). The number of elements shifted is equal to `abs(n)`. It is assumed that `x` is already sorted within each group by time such that the first element for that group is earliest in time and the last element for that group is the latest in time.

Usage

```
shift_by(x, grp, n, undefined = NA)
```

Arguments

`x`	atomic vector or list vector.
`grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as `x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`n`	integer vector with length 1. Specifies the direction and magnitude of the shift. See details.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See details.

Details

If n is negative, then shift_by inserts undefined into the first abs(n) elements of x for each group, shifting all other values of x to the right abs(n) positions, and then dropping the last abs(n) elements of x to preserve the original length of each group. If n is positive, then shift_by drops the first abs(n) elements of x for each group, shifting all other values of x left abs(n) positions, and then inserts undefined into the last abs(n) elements of x to preserve the original length of each group. If n is zero, then shift_by simply returns x.

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift_by tries to circumvent this issue by a call to round within shift_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift_by truncates rather than rounds.

Value

an atomic vector of the same length as x that is shifted by group. If x and undefined are different typeofs, then the return will be coerced to the most complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

shifts_by shift shifts

Examples

```
shift_by(x = ChickWeight[["Time"]], grp = ChickWeight[["Chick"]], n = -1L)
tmp_nm <- c("vs","am") # b/c Roxygen2 doesn't like c() in a []
shift_by(x = mtcars[["disp"]], grp = mtcars[tmp_nm], n = 1L)
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like c() in a []
shift_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n = 2L) # multiple grouping vectors
```

---

sum_if *Sum Conditional on Minimum Frequency of Observed Values*

Description

sum_if calculates the sum of a numeric or logical vector conditional on a specified minimum frequency of observed values. If the amount of observed data is less than (or equal to) ov.min, then NA is returned rather than the sum.

Usage

```
sum_if(x, impute = TRUE, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

`x`	numeric or logical vector.
`impute`	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of `x`. If TRUE (default), this will make sums over the same vectors with different amounts of missing data comparable.
`ov.min`	minimum frequency of observed values required. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `length(x)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the sum should be calculated (rather than NA) if the frequency of observed values is exactly equal to `ov.min`.

Value

numeric vector of length 1 providing the sum of `x` or `NA` conditional on if the frequency of observed data is greater than (or equal to) `ov.min`.

See Also

`sum` `mean_if` `make.fun_if`

Examples

```
sum_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
sum_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observe values
sum_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min value itself
sum_if(x = c(TRUE, NA, FALSE, NA),
   ov.min = .50) # works with logical vectors as well as numeric
```

`tapply2` *Apply a Function to a (Atomic) Vector by Group*

Description

`tapply2` applies a function to a (atomic) vector by group and is an alternative to the base R function `tapply`. The function is apart of the split-apply-combine type of function discussed in the `plyr` R package and is somewhat similar to `dlply`. It splits up one (atomic) vector `.x`into a (atomic) vector for each group in `.grp`, applies a function `.fun` to each (atomic) vector, and then returns the results as a list with names equal to the group values `unique(interaction(.grp.nm,sep = .sep))`. `tapply2` is simply `split.default` + `lapply`. Similar to `dlply`, The arguments all start with `.` so that they do not conflict with arguments from the function `.fun`. If you want to apply a function a data.frame rather than a (atomic) vector, then use `by2`.

Usage

```
tapply2(.x, .grp, .sep = ".", .fun, ...)
```

Arguments

`.x`	atomic vector
`.grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame) containing the groups. They should each have same length as `.x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`.sep`	character vector of length 1 specifying the string to combine the group values together with. `.sep` is only used if there are multiple grouping variables (i.e., `.grp` is a list with multiple elements).
`.fun`	function to apply to `.x` for each group.
`...`	additional named arguments to pass to `.fun`.

Value

list of objects containing the return object of `.fun` for each group. The names are the unique combinations of the grouping variables (i.e., `unique(interaction(.grp,sep = .sep))`).

See Also

`tapply` `by2` `dlply`

Examples

```
# one grouping variable
tapply2(mtcars$"cyl", .grp = mtcars$"vs", .fun = median, na.rm = TRUE)

# two grouping variables
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
x <- tapply2(mtcars$"cyl", .grp = mtcars[grp_nm], .fun = median, na.rm = TRUE)
print(x)
str(x)

# compare to tapply
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
y <- tapply(mtcars$"cyl", INDEX = mtcars[grp_nm],
   FUN = median, na.rm = TRUE, simplify = FALSE)
print(y)
str(y) # has dimnames rather than names
```

valids_test *Test for Invalid Elements in Data*

Description

`Valid.test` tests whether data has any invalid elements. Valid values are specified by `valid`. Each variable is tested independently. If the variable in `data[vrb.nm]` has any values other than `valid`, then FALSE is returned for that variable; If the variable in `data[vrb.nm]` only has values in `valid`, then TRUE is returned for that variable.

Usage

```
valids_test(data, vrb.nm, valid, na.rm = TRUE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables
`valid`	atomic vector or list vector of valid values.
`na.rm`	logical vector of length 1 specifying whether NA should be ignored from the validity test. If TRUE (default), then any NAs are treated as valid.

Value

logical vector with length = `length(vrb.nm)` and names = `vrb.nm` specifying whether all elements in each variable of `data[vrb.nm]` are valid. If FALSE, then (at least one) invalid values are present in that variable of `data[vrb.nm]`.

See Also

`valid_test` `revalids` `revalid`

Examples

```
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 1:6) # return TRUE
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 0:5) # 6 is not present in `valid`
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 1:6, na.rm = FALSE) # NA is not present in `valid`
valids_test(data = ToothGrowth, vrb.nm = c("supp","dose"),
   valid = list("VC", "OJ", 0.5, 1.0, 2.0)) # list vector as `valid` to allow for
   # elements of different typeof
```

valid_test *Test for Invalid Elements in a Vector*

Description

`valid_test` tests whether a vector has any invalid elements. Valid values are specified by `valid`. If the vector `x` has any values other than `valid`, then FALSE is returned; If the vector `x` only has values in `valid`, then TRUE is returned. This function can be useful for checking data after manual human entry.

Usage

```
valid_test(x, valid, na.rm = TRUE)
```

Arguments

`x`	atomic vector or list vector.
`valid`	atomic vector or list vector of valid values.
`na.rm`	logical vector of length 1 specifying whether NA should be ignored from the validity test. If TRUE (default), then any NAs are treated as valid.

Value

logical vector of length 1 specifying whether all elements in `x` are valid values. If FALSE, then (at least one) invalid values are present.

See Also

`valids_test` `revalid` `revalids`

Examples

```
valid_test(x = psych::bfi[[1]], valid = 1:6) # return TRUE
valid_test(x = psych::bfi[[1]], valid = 0:5) # 6 is not present in `valid`
valid_test(x = psych::bfi[[1]], valid = 1:6,
   na.rm = FALSE) # NA is not present in `valid`
```

vecNA — *Frequency of Missing Values in a Vector*

Description

vecNA computes the frequency of missing values in an atomic vector. vecNA is essentially a wrapper for sum or mean + is.na or !is.na and can be useful for functional programming (e.g., lapply(FUN = vecNA)). It is also used by other functions in the quest package related to missing values (e.g., mean_if).

Usage

```
vecNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	atomic vector or list vector. If not a vector, it will be coerced to a vector via as.vector.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length 1 providing the frequency of missing values (or observed values if ov = TRUE). If prop = TRUE, the value will range from 0 to 1. If prop = FALSE, the value will range from 1 to length(x).

See Also

is.na rowNA colNA rowsNA

Examples

```
vecNA(airquality[[1]]) # count of missing values
vecNA(airquality[[1]], prop = TRUE) # proportion of missing values
vecNA(airquality[[1]], ov = TRUE) # count of observed values
vecNA(airquality[[1]], prop = TRUE, ov = TRUE) # proportion of observed values
```

`wide2long` *Reshape Multiple Sets of Variables From Wide to Long*

Description

`wide2long` reshapes data from wide to long. This if often necessary to do with multilevel data where multiple sets of variables in the wide format seek to be reshaped to multiple rows in the long format. If only one set of variables needs to be reshaped, then you can use `stack2` or `melt.data.frame` - but that does not work for *multiple* sets of variables. See details for more information.

Usage

```
wide2long(
  data,
  vrb.nm.list,
  grp.nm = NULL,
  sep = ".",
  rtn.obs.nm = "obs",
  order.by.grp = TRUE,
  keep.attr = FALSE
)
```

Arguments

`data`	data.frame of multilevel data in the wide format.
`vrb.nm.list`	A unique argument for the `quest` package such that it can take on different types of inputs. The conventional use is to provide a list of character vectors specifying each set of colnames to be reshaped. In longitudinal panel data, each list element would contain a score with multiple timepoints. The advanced use is to provide a single character vector specifying the colnames to be reshaped (not organized by sets). See details.
`grp.nm`	character vector specifying the colnames in `data` corresponding to the groups. Because `data` is in the wide format, `data[grp.nm]` must have unique rows (aka groups); if this is not the case, an error is returned. `grp.nm` can be NULL, in which case the rownames of `data` will be used. In longitudinal panel data this variable would be the participant ID variable.
`sep`	character vector of length 1 specifying the string in the column names provided by `vrb.nm.list` that separates out the name prefix from the number suffix. If `sep` = "", then that implies there is no string separating the name prefix and the number suffix (e.g., "outcome1").
`rtn.obs.nm`	character vector of length 1 specifying the new colname in the return object indicating which observation within each group the row refers to. In longitudinal panel data, this would be the returned time variable.
`order.by.grp`	logical vector of length 1 specifying whether to sort the return object first by `grp.nm` and then `obs.nm` (TRUE) or by `obs.nm` and then `grp.nm` (FALSE).
`keep.attr`	logical vector of length 1 specifying whether to keep the "reshapeLong" attribute (from `reshape`) in the return object.

Details

`wide2long` uses `reshape(direction = "long")` to reshape the data. It attempts to streamline the task of reshaping wide to long as the `reshape` arguments can be confusing because the same arguments are used for wide vs. long reshaping. See `reshape` if you are curious.

IF `vrb.nm.list` IS A LIST OF CHARACTER VECTORS: The conventional use of `vrb.nm.list` is to provide a list of character vectors, which specify each set of variables to be reshaped. For example, if `data` contains data from a longitudinal panel study with the same scores at different waves, then there might be a column for each score at each wave. `vrb.nm.list` would then contain an element for each score with each element containing a character vector of the colnames for that score at each wave (see examples). The names of the list elements would then be the colnames in the return object for those scores.

IF `vrb.nm.list` IS A CHARACTER VECTOR: The advanced use of `vrb.nm.list` is to provide a single character vector, which specify the variables to be reshaped (not organized by sets). In this case (i.e., if `vrb.nm.list` is not a list), then `wide2long` (really `reshape`) will attempt to guess which colnames go together as a set. It is assumed the following column naming scheme has been used: 1) have the same name prefix for columns within a set, 2) have the same number suffixes for each set of columns, 3) use, *and only use*, `sep` in the colnames to separate the name prefix and the number suffix. For example, the name prefixes might be "predictor" and "outcome" while the number suffixes might be "0", "1", and "2", and the separator might be ".", resulting in column names such as "outcome.1". The name prefix could include separators other than `sep` (e.g., "outcome_item.1"), but it cannot include `sep` (e.g., "outcome.item.1"). So "outcome_item1.1" could be acceptable, but "outcome.item1.1" would not.

Value

data.frame with nrow equal to `nrow(data) * length(vrb.nm.list[[1]])` if `vrb.nm.list` is a list (i.e., conventional use) or `nrow(data)` * number of unique number suffixes in `vrb.nm.list` if `vrb.nm.list` is not a list (i.e., advanced use). The columns will be in the following order: 1) `grp.nm` of the groups, 2) `rtn.obs.nm` of the observation labels, 3) the reshaped columns, 4) the additional columns that were not reshaped and instead repeated. How the returned data.frame is sorted depends on `order.by.grp`.

See Also

`long2wide` `reshape` `stack2`

Examples

```
# SINGLE GROUPING VARIABLE
dat_wide <- data.frame(
   x_1.1 = runif(5L),
   x_2.1 = runif(5L),
   x_3.1 = runif(5L),
   x_4.1 = runif(5L),
   x_1.2 = runif(5L),
   x_2.2 = runif(5L),
   x_3.2 = runif(5L),
   x_4.2 = runif(5L),
   x_1.3 = runif(5L),
   x_2.3 = runif(5L),
   x_3.3 = runif(5L),
   x_4.3 = runif(5L),
   y_1.1 = runif(5L),
   y_2.1 = runif(5L),
   y_1.2 = runif(5L),
   y_2.2 = runif(5L),
   y_1.3 = runif(5L),
   y_2.3 = runif(5L))
row.names(dat_wide) <- letters[1:5]
print(dat_wide)

# vrb.nm.list = list of character vectors (conventional use)
vrb_pat <- c("x_1","x_2","x_3","x_4","y_1","y_2")
vrb_nm_list <- lapply(X = setNames(vrb_pat, nm = vrb_pat), FUN = function(pat) {
   str2str::pick(x = names(dat_wide), val = pat, pat = TRUE)})
# without `grp.nm`
z1 <- wide2long(dat_wide, vrb.nm = vrb_nm_list)
# with `grp.nm`
dat_wide$"ID" <- letters[1:5]
z2 <- wide2long(dat_wide, vrb.nm = vrb_nm_list, grp.nm = "ID")
dat_wide$"ID" <- NULL

# vrb.nm.list = character vector + guessing (advanced use)
vrb_nm <- str2str::pick(x = names(dat_wide), val = "ID", not = TRUE)
# without `grp.nm`
z3 <- wide2long(dat_wide, vrb.nm.list = vrb_nm)
# with `grp.nm`
dat_wide$"ID" <- letters[1:5]
z4 <- wide2long(dat_wide, vrb.nm = vrb_nm, grp.nm = "ID")
dat_wide$"ID" <- NULL

# comparisons
head(z1); head(z3); head(z2); head(z4)
all.equal(z1, z3)
all.equal(z2, z4)
# keeping the reshapeLong attributes
z7 <- wide2long(dat_wide, vrb.nm = vrb_nm_list, keep.attr = TRUE)
attributes(z7)

# MULTIPLE GROUPING VARIABLES
bfi2 <- psych::bfi
bfi2$"person" <- unlist(lapply(X = 1:400, FUN = rep.int, times = 7))
bfi2$"day" <- rep.int(1:7, times = 400L)
head(bfi2, n = 15)

# vrb.nm.list = list of character vectors (conventional use)
vrb_pat <- c("A","C","E","N","O")
vrb_nm_list <- lapply(X = setNames(vrb_pat, nm = vrb_pat), FUN = function(pat) {
   str2str::pick(x = names(bfi2), val = pat, pat = TRUE)})
z5 <- wide2long(bfi2, vrb.nm.list = vrb_nm_list, grp = c("person","day"),
   rtn.obs.nm = "item")
```

```
# vrb.nm.list = character vector + guessing (advanced use)
vrb_nm <- str2str::pick(x = names(bfi2),
   val = c("person","day","gender","education","age"), not = TRUE)
z6 <- wide2long(bfi2, vrb.nm.list = vrb_nm, grp = c("person","day"),
   sep = "", rtn.obs.nm = "item") # need sep = "" because no character separating
   # scale name and item number
all.equal(z5, z6)
```

winsor *Winsorize a Numeric Vector*

Description

`winsor` winsorizes a numeric vector by recoding extreme values as a user-identified boundary value, which is defined by z-score units. The `to.na` argument provides the option of recoding the extreme values as missing.

Usage

```
winsor(x, z.min = -3, z.max = 3, rtn.int = FALSE, to.na = FALSE)
```

Arguments

`x`	numeric vector
`z.min`	numeric vector of length 1 specifying the lower boundary value in z-score units.
`z.max`	numeric vector of length 1 specifying the upper boundary value in z-score units.
`rtn.int`	logical vector of length 1 specifying whether the recoded values should be rounded to the nearest integer. This can be useful when working with count data and decimal values are impossible.
`to.na`	logical vector of length 1 specifying whether the extreme values should be recoded to NA rather than winsorized to the boundary values.

Details

Note, the psych package also has a function called `winsor`, which offers the option to winsorize a numeric vector by quantiles rather than z-scores. If you have both the quest package and the psych package attached in your current R session (e.g., using `library`), depending on which package you attached first, R might default to using the `winsor` function in either the quest package or the psych package. One way to deal with this issue is to explicitly call which package you want to use the `winsor` package from. You can do this using the `::` function in base R where the package name comes before the `::` and the function names comes after it (e.g., `quest::winsor`).

Value

numeric vector of the same length as `x` with extreme values recoded as either the boundary values or NA.

See Also

winsors winsor # psych package

Examples

```
# winsorize
table(quakes$"stations")
new <- winsor(quakes$"stations")
table(new)

# recode as NA
vecNA(quakes$"stations")
new <- winsor(quakes$"stations", to.na = TRUE)
vecNA(new)

# rtn.int = TRUE
winsor(x = cars[[1]], z.min = -2, z.max = 2, rtn.int = FALSE)
winsor(x = cars[[1]], z.min = -2, z.max = 2, rtn.int = TRUE)
```

---

winsors *Winsorize Numeric Data*

Description

`winsors` winsorizes numeric data by recoding extreme values as a user identified boundary value, which is defined by z-score units. The `to.na` argument provides the option of recoding the extreme values as missing.

Usage

```
winsors(
  data,
  vrb.nm,
  z.min = -3,
  z.max = 3,
  rtn.int = FALSE,
  to.na = FALSE,
  suffix = "_win"
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`z.min`	numeric vector of length 1 specifying the lower boundary value in z-score units.
`z.max`	numeric vector of length 1 specifying the upper boundary value in z-score units.

`rtn.int`	logical vector of length 1 specifying whether the recoded values should be rounded to the nearest integer. This can be useful when working with count data and decimal values are impossible.
`to.na`	logical vector of length 1 specifying whether the extreme values should be recoded to NA rather than winsorized to the boundary values.
`suffix`	character vector of length 1 specifying the string to append to the end of the colnames in the return object.

Value

data.frame of winsorized data with extreme values recoded as either the boundary values or NA and colnames = `paste0(vrb.nm,suffix)`.

See Also

`winsor` `winsor` # psych package

Examples

```
# winsorize
lapply(X = quakes[c("mag","stations")], FUN = table)
new <- winsors(quakes, vrb.nm = names(quakes))
lapply(X = new, FUN = table)

# recode as NA
vecNA(quakes)
new <- winsors(quakes, vrb.nm = names(quakes), to.na = TRUE)
vecNA(new)

# rtn.int = TRUE
winsors(data = cars, vrb.nm = names(cars), z.min = -2, z.max = 2, rtn.int = FALSE)
winsors(data = cars, vrb.nm = names(cars), z.min = -2, z.max = 2, rtn.int = TRUE)
```

Index