Package 'sjmisc'

September 24, 2019

Type Package

Encoding UTF-8

Title Data and Variable Transformation Functions

Version 2.8.2

Date 2019-09-24

Maintainer Daniel Lüdecke <d.luedecke@uke.de>

Description Collection of miscellaneous utility functions, supporting data transformation tasks like recoding, dichotomizing or grouping variables, setting and replacing missing values. The data transformation functions also support labelled data, and all integrate seamlessly into a 'tidyverse'-workflow.

License GPL-3

Depends R (>= 3.2)

Imports dplyr (>= 0.8.0), insight (>= 0.5.0), magrittr, methods, purrr, rlang, sjlabelled (>= 1.1.1), stats, tidyselect (>= 0.2.5), utils

Suggests ggplot2, graphics, haven (>= 1.1.2), mice, sjPlot, sjstats, knitr, rmarkdown, stringdist, testthat, tidyr

URL https://strengejacke.github.io/sjmisc

BugReports https://github.com/strengejacke/sjmisc/issues

RoxygenNote 6.1.1

VignetteBuilder knitr

NeedsCompilation no

Author Daniel Lüdecke [aut, cre] (<https://orcid.org/0000-0002-8895-3206>), Iago Giné-Vázquez [ctb], Alexander Bartel [ctb] (<https://orcid.org/0000-0002-1280-6138>)

Repository CRAN

Date/Publication 2019-09-24 14:30:02 UTC

R topics documented:

`sjmisc-package` *Data and Variable Transformation Functions*

Description

Purpose of this package

Collection of miscellaneous utility functions, supporting data transformation tasks like recoding, dichotomizing or grouping variables, setting and replacing missing values. The data transformation functions also support labelled data, and all integrate seamlessly into a 'tidyverse'-workflow.

Design philosophy - consistent api

The design of this package follows, where appropriate, the *tidyverse-approach*, with the first argument of a function always being the data (either a data frame or vector), followed by variable names that should be processed by the function. If no variables are specified as argument, the function applies to the complete data that was indicated as first function argument.

There are two types of function designs:

transformation/recoding functions Functions like `rec()` or `dicho()`, which transform or recode variables, typically return the complete data frame that was given as first argument, *additionally including* the transformed and recoded variables specified in the `...`-ellipses argument. The variables usually get a suffix, so original variables are preserved in the data.

coercing/converting functions Functions like `to_factor()` or `to_label()`, which convert variables into other types or add additional information like variable or value labels as attribute, also typically return the complete data frame that was given as first argument. However, the variables specified in the `...`-ellipses argument are converted ("overwritten"), all other variables remain unchanged. Hence, these functions do not return any new, additional variables.

Author(s)

Daniel Lüdecke <d.luedecke@uke.de>

add_columns *Add or replace data frame columns*

Description

add_columns() combines two or more data frames, but unlike cbind or dplyr::bind_cols(), this function binds data as last columns of a data frame (i.e., behind columns specified in ...). This can be useful in a "pipe"-workflow, where a data frame returned by a previous function should be appended *at the end* of another data frame that is processed in add_colums().

replace_columns() replaces all columns in data with identically named columns in ..., and adds remaining (non-duplicated) columns from ... to data.

add_id() simply adds an ID-column to the data frame, with values from 1 to nrow(data), respectively for grouped data frames, values from 1 to group size. See 'Examples'.

Usage

```
add_columns(data, ..., replace = TRUE)

replace_columns(data, ..., add.unique = TRUE)

add_id(data, var = "ID")
```

Arguments

data	A data frame. For add_columns(), will be bound after data frames specified in .... For replace_columns(), duplicated columns in data will be replaced by columns in ....
...	More data frames to combine, resp. more data frames with columns that should replace columns in data.
replace	Logical, if TRUE (default), columns in ... with identical names in data will replace the columns in data. The order of columns after replacing is preserved.
add.unique	Logical, if TRUE (default), remaining columns in ... that did not replace any column in data, are appended as new columns to data.
var	Name of new the ID-variable.

Value

For add_columns(), a data frame, where columns of data are appended after columns of ....

For replace_columns(), a data frame where columns in data will be replaced by identically named columns in ..., and remaining columns from ... will be appended to data (if add.unique = TRUE).

For add_id(), a new column with ID numbers. This column is always the first column in the returned data frame.

Note

For add_columns(), by default, columns in data with identical names like columns in one of the data frames in ... will be dropped (i.e. variables with identical names in ... will replace existing variables in data). Use replace = FALSE to keep all columns. Identical column names will then be renamed, to ensure unique column names (which happens by default when using dplyr::bind_cols()). When replacing columns, replaced columns are not added to the end of the data frame. Rather, the original order of columns will be preserved.

Examples

```
data(efc)
d1 <- efc[, 1:3]
d2 <- efc[, 4:6]

library(dplyr)
head(bind_cols(d1, d2))
add_columns(d1, d2)

d1 <- efc[, 1:3]
d2 <- efc[, 2:6]

add_columns(d1, d2, replace = TRUE)
add_columns(d1, d2, replace = FALSE)

# use case: we take the original data frame, select specific
# variables and do some transformations or recodings
# (standardization in this example) and add the new, transformed
# variables *to the end* of the original data frame
efc %>%
  select(e17age, c160age) %>%
  std() %>%
  add_columns(efc)

# new variables with same name will overwrite old variables
# in "efc". order of columns is not changed.
efc %>%
  select(e16sex, e42dep) %>%
  to_factor() %>%
  add_columns(efc)

# keep both old and new variables, automatically
# rename variables with identical name
efc %>%
  select(e16sex, e42dep) %>%
  to_factor() %>%
  add_columns(efc, replace = FALSE)

# create sample data frames
d1 <- efc[, 1:10]
d2 <- efc[, 2:3]
d3 <- efc[, 7:8]
d4 <- efc[, 10:12]
```

```
# show original
head(d1)

library(sjlabelled)
# slightly change variables, to see effect
d2 <- as_label(d2)
d3 <- as_label(d3)

# replace duplicated columns, append remaining
replace_columns(d1, d2, d3, d4)

# replace duplicated columns, omit remaining
replace_columns(d1, d2, d3, d4, add.unique = FALSE)

# add ID to dataset
library(dplyr)
data(mtcars)
add_id(mtcars)

mtcars %>%
  group_by(gear) %>%
  add_id() %>%
  arrange(gear, ID) %>%
  print(n = 100)
```

---

add_rows *Merge labelled data frames*

Description

Merges (full join) data frames and preserve value and variable labels.

Usage

```
add_rows(..., id = NULL)

merge_df(..., id = NULL)
```

Arguments

...	Two or more data frames to be merged.
id	Optional name for ID column that will be created to indicate the source data frames for appended rows.

Details

This function works like dplyr::bind_rows(), but preserves variable and value label attributes. add_rows() row-binds all data frames in ..., even if these have different numbers of columns. Non-matching columns will be column-bound and filled with NA-values for rows in those data frames that do not have this column.

Value and variable labels are preserved. If matching columns have different value label attributes, attributes from first data frame will be used.

merge_df() is an alias for add_rows().

Value

A full joined data frame.

Examples

```
library(dplyr)
data(efc)
x1 <- efc %>% select(1:5) %>% slice(1:10)
x2 <- efc %>% select(3:7) %>% slice(11:20)

mydf <- add_rows(x1, x2)
mydf
str(mydf)

## Not run:
library(sjPlot)
view_df(mydf)
## End(Not run)

x3 <- efc %>% select(5:9) %>% slice(21:30)
x4 <- efc %>% select(11:14) %>% slice(31:40)

mydf <- add_rows(x1, x2, x3, x4, id = "subsets")
mydf
str(mydf)
```

---

add_variables *Add variables or cases to data frames*

Description

add_variables() adds a new column to a data frame, while add_case() adds a new row to a data frame. These are convenient functions to add columns or rows not only at the end of a data frame, but at any column or row position. Furthermore, they allow easy integration into a pipe-workflow.

Usage

```
add_variables(data, ..., .after = Inf, .before = NULL)

add_case(data, ..., .after = Inf, .before = NULL)
```

Arguments

data	A data frame.
...	One or more named vectors that indicate the variables or values, which will be added as new column or row to data. For add_case(), non-matching columns in data will be filled with NA.
.after, .before	Numerical index of row or column, where after or before the new variable or case should be added. If .after = -1, variables or cases are added at the beginning; if .after = Inf, variables and cases are added at the end. In case of add_variables(), .after and .before may also be a character name indicating the column in data, after or infront of what ... should be inserted.

Value

data, including the new variables or cases from ....

Note

For add_case(), if variable does not exist, a new variable is created and existing cases for this new variable get the value NA. See 'Examples'.

Examples

```
d <- data.frame(
  a = c(1, 2, 3),
  b = c("a", "b", "c"),
  c = c(10, 20, 30),
  stringsAsFactors = FALSE
)

add_case(d, b = "d")
add_case(d, b = "d", a = 5, .before = 1)

# adding a new case for a new variable
add_case(d, e = "new case")

add_variables(d, new = 5)
add_variables(d, new = c(4, 4, 4), new2 = c(5, 5, 5), .after = "b")
```

all_na *Check if vector only has NA values*

Description

Check if all values in a vector are NA.

Usage

```
all_na(x)
```

Arguments

x	A vector or data frame.

Value

Logical, TRUE if x has only NA values, FALSE if x has at least one non-missing value.

Examples

```
x <- c(NA, NA, NA)
y <- c(1, NA, NA)

all_na(x)
all_na(y)
all_na(data.frame(x, y))
all_na(list(x, y))
```

big_mark *Format numbers*

Description

big_mark() formats large numbers with big marks, while prcn() converts a numeric scalar between 0 and 1 into a character vector, representing the percentage-value.

Usage

```
big_mark(x, big.mark = ",", ...)

prcn(x)
```

Arguments

x	A vector or data frame. All numeric inputs (including numeric character) vectors) will be prettified. For prcn(), a number between 0 and 1, or a vector or data frame with such numbers.
big.mark	Character, used as mark between every 3 decimals before the decimal point.
...	Other arguments passed down to the prettyNum-function.

Value

For big_mark(), a prettified x as character, with big marks. For prcn, a character vector with a percentage number.

Examples

```
# simple big mark
big_mark(1234567)

# big marks for several values at once, mixed numeric and character
big_mark(c(1234567, "55443322"))

# pre-defined width of character output
big_mark(c(1234567, 55443322), width = 15)

# convert numbers into percentage, as character
prcn(0.2389)
prcn(c(0.2143887, 0.55443, 0.12345))

dat <- data.frame(
  a = c(.321, .121, .64543),
  b = c("a", "b", "c"),
  c = c(.435, .54352, .234432)
)
prcn(dat)
```

---

count_na *Frequency table of tagged NA values*

Description

This method counts tagged NA values (see tagged_na) in a vector and prints a frequency table of counted tagged NAs.

Usage

```
count_na(x, ...)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.

Value

A data frame with counted tagged NA values.

Examples

```
library(haven)
x <- labelled(
  x = c(1:3, tagged_na("a", "c", "z"),
        4:1, tagged_na("a", "a", "c"),
        1:3, tagged_na("z", "c", "c"),
        1:4, tagged_na("a", "c", "z")),
  labels = c("Agreement" = 1, "Disagreement" = 4,
             "First" = tagged_na("c"), "Refused" = tagged_na("a"),
             "Not home" = tagged_na("z"))
)
count_na(x)

y <- labelled(
  x = c(1:3, tagged_na("e", "d", "f"),
        4:1, tagged_na("f", "f", "d"),
        1:3, tagged_na("f", "d", "d"),
        1:4, tagged_na("f", "d", "f")),
  labels = c("Agreement" = 1, "Disagreement" = 4, "An E" = tagged_na("e"),
             "A D" = tagged_na("d"), "The eff" = tagged_na("f"))
)

# create data frame
dat <- data.frame(x, y)

# possible count()-function calls
count_na(dat)
count_na(dat$x)
count_na(dat, x)
count_na(dat, x, y)
```

---

descr *Basic descriptive statistics*

Description

This function prints a basic descriptive statistic, including variable labels.

Usage

```
descr(x, ..., max.length = NULL, weights = NULL, show = "all",
  out = c("txt", "viewer", "browser"), encoding = "UTF-8",
  file = NULL)
```

Arguments

x	A vector or a data frame. May also be a grouped data frame (see 'Note' and 'Examples').
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
max.length	Numeric, indicating the maximum length of variable labels in the output. If variable names are longer than max.length, they will be shortened to the last whole word within the first max.length chars.
weights	Bare name, or name as string, of a variable in x that indicates the vector of weights, which will be applied to weight all observations. Default is NULL, so no weights are used.
show	Character vector, indicating which information (columns) that describe the data should be returned. May be one or more of "type","label","n","NA.prc","mean","sd","se","md", There are two shortcuts: show = "all" (default) shows all information, show = "short" just shows n, missing percentage, mean and standard deviation.
out	Character vector, indicating whether the results should be printed to console (out = "txt") or as HTML-table in the viewer-pane (out = "viewer") or browser (out = "browser").
encoding	Character vector, indicating the charset encoding used for variable and value labels. Default is "UTF-8". Only used when out is not "txt".
file	Destination file, if the output should be saved as file. Only used when out is not "txt".

Value

A data frame with basic descriptive statistics.

Note

data may also be a grouped data frame (see group_by) with up to two grouping variables. Descriptive tables are created for each subgroup then.

Examples

```
data(efc)
descr(efc, e17age, c160age)

efc$weights <- abs(rnorm(nrow(efc), 1, .3))
descr(efc, c12hour, barthtot, weights = weights)
```

```
library(dplyr)
efc %>% select(e42dep, e15relat, c172code) %>% descr()

# show just a few elements
efc %>% select(e42dep, e15relat, c172code) %>% descr(show = "short")

# with grouped data frames
efc %>%
  group_by(e16sex) %>%
  select(e16sex, e42dep, e15relat, c172code) %>%
  descr()

# you can select variables also inside 'descr()'
efc %>%
  group_by(e16sex, c172code) %>%
  descr(e16sex, c172code, e17age, c160age)

# or even use select-helpers
descr(efc, contains("cop"), max.length = 20)
```

---

de_mean *Compute group-meaned and de-meaned variables*

Description

de_mean() computes group- and de-meaned versions of a variable that can be used in regression analysis to model the between- and within-subject effect.

Usage

```
de_mean(x, ..., grp, append = TRUE, suffix.dm = "_dm",
  suffix.gm = "_gm")
```

Arguments

x	A data frame.
...	Names of variables that should be group- and de-meaned.
grp	Quoted or unquoted name of the variable that indicates the group- or cluster-ID.
append	Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.
suffix.dm, suffix.gm	String value, will be appended to the names of the group-meaned and de-meaned variables of x. By default, de-meaned variables will be suffixed with "_dm" and grouped-meaned variables with "_gm".

Details

de_mean() is intended to create group- and de-meaned variables for complex random-effect-within-between models (see *Bell et al. 2018*), where group-effects (random effects) and fixed effects correlate (see *Bafumi and Gelman 2006)*). This violation of one of the *Gauss-Markov-assumptions* can happen, for instance, when analysing panel data. To control for correlating predictors and group effects, it is recommended to include the group-meaned and de-meaned version of *time-varying covariates* in the model. By this, one can fit complex multilevel models for panel data, including time-varying, time-invariant predictors and random effects. This approach is superior to simple fixed-effects models, which lack information of variation in the group-effects or between-subject effects.

A description of how to translate the formulas described in *Bell et al. 2018* into R using lmer() from **lme4** or glmmTMB() from **glmmTMB** can be found here: for lmer() and for glmmTMB().

Value

For append = TRUE, x including the group-/de-meaned variables as new columns is returned; if append = FALSE, only the group-/de-meaned variables will be returned.

References

Bafumi J, Gelman A. 2006. Fitting Multilevel Models When Predictors and Group Effects Correlate. In. Philadelphia, PA: Annual meeting of the American Political Science Association.

Bell A, Fairbrother M, Jones K. 2018. Fixed and Random Effects Models: Making an Informed Choice. Quality & Quantity. doi: 10.1007/s111350180802x

Examples

```
data(efc)
efc$ID <- sample(1:4, nrow(efc), replace = TRUE) # fake-ID
de_mean(efc, c12hour, barthtot, grp = ID, append = FALSE)
```

---

dicho *Dichotomize variables*

Description

Dichotomizes variables into dummy variables (0/1). Dichotomization is either done by median, mean or a specific value (see dich.by). dicho_if() is a scoped variant of dicho(), where recoding will be applied only to those variables that match the logical condition of predicate.

Usage

```
dicho(x, ..., dich.by = "median", as.num = FALSE, var.label = NULL,
  val.labels = NULL, append = TRUE, suffix = "_d")

dicho_if(x, predicate, dich.by = "median", as.num = FALSE,
  var.label = NULL, val.labels = NULL, append = TRUE,
  suffix = "_d")
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
dich.by	Indicates the split criterion where a variable is dichotomized. Must be one of the following values (may be abbreviated): "median" **or** "md" by default, x is split into two groups at the median. "mean" **or** "m" splits x into two groups at the mean of x. **numeric value** splits x into two groups at the specific value. Note that the value is inclusive, i.e. dich.by = 10 will split x into one group with values from lowest to 10 and another group with values greater than 10.
as.num	Logical, if TRUE, return value will be numeric, not a factor.
var.label	Optional string, to set variable label attribute for the returned variable (see vignette Labelled Data and the sjlabelled-Package). If NULL (default), variable label attribute of x will be used (if present). If empty, variable label attributes will be removed.
val.labels	Optional character vector (of length two), to set value label attributes of dichotomized variable (see set_labels). If NULL (default), no value labels will be set.
append	Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.
suffix	String value, will be appended to variable (column) names of x, if x is a data frame. If x is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call: • recoded variables (rec()) will be suffixed with "_r" • recoded variables (recode_to()) will be suffixed with "_r0" • dichotomized variables (dicho()) will be suffixed with "_d" • grouped variables (split_var()) will be suffixed with "_g" • grouped variables (group_var()) will be suffixed with "_gr" • standardized variables (std()) will be suffixed with "_z" • centered variables (center()) will be suffixed with "_c" • de-meaned variables (de_mean()) will be suffixed with "_dm" • grouped-meaned variables (de_mean()) will be suffixed with "_gm"

If suffix = "" and append = TRUE, existing variables that have been recoded/transformed will be overwritten.

predicate — A predicate function to be applied to the columns. The variables for which predicate returns TRUE are selected.

Details

dicho() also works on grouped data frames (see group_by). In this case, dichotomization is applied to the subsets of variables in x. See 'Examples'.

Value

x, dichotomized. If x is a data frame, for append = TRUE, x including the dichotomized. variables as new columns is returned; if append = FALSE, only the dichotomized variables will be returned. If append = TRUE and suffix = "", recoded variables will replace (overwrite) existing variables.

Note

Variable label attributes are preserved (unless changed via var.label-argument).

Examples

```
data(efc)
summary(efc$c12hour)
# split at median
table(dicho(efc$c12hour))
# split at mean
table(dicho(efc$c12hour, dich.by = "mean"))
# split between value lowest to 30, and above 30
table(dicho(efc$c12hour, dich.by = 30))

# sample data frame, values from 1-4
head(efc[, 6:10])

# dichtomized values (1 to 2 = 0, 3 to 4 = 1)
library(dplyr)
efc %>%
  select(6:10) %>%
  dicho(dich.by = 2) %>%
  head()

# dichtomize several variables in a data frame
dicho(efc, c12hour, e17age, c160age, append = FALSE)

# dichotomize and set labels
frq(dicho(
  efc, e42dep,
  var.label = "Dependency (dichotomized)",
  val.labels = c("lower", "higher"),
  append = FALSE
))
```

```
# works also with gouped data frames
mtcars %>%
  dicho(disp, append = FALSE) %>%
  table()

mtcars %>%
  group_by(cyl) %>%
  dicho(disp, append = FALSE) %>%
  table()

# dichotomizing grouped data frames leads to different
# results for a dichotomized variable, because the split
# value is different for each group.
# compare:
mtcars %>%
  group_by(cyl) %>%
  summarise(median = median(disp))

median(mtcars$disp)

# dichotomize only variables with more than 10 unique values
p <- function(x) dplyr::n_distinct(x) > 10
dicho_if(efc, predicate = p, append = FALSE)
```

`efc` *Sample dataset from the EUROFAMCARE project*

Description

A SPSS sample data set, imported with the `read_spss` function.

Examples

```
# Attach EFC-data
data(efc)

# Show structure
str(efc)

# show first rows
head(efc)
```

empty_cols *Return or remove variables or observations that are completely missing*

Description

These functions check which rows or columns of a data frame completely contain missing values, i.e. which observations or variables completely have missing values, and either 1) returns their indices; or 2) removes them from the data frame.

Usage

```
empty_cols(x)

empty_rows(x)

remove_empty_cols(x)

remove_empty_rows(x)
```

Arguments

x	A data frame.

Value

For `empty_cols` and `empty_rows`, a numeric (named) vector with row or column indices of those variables that completely have missing values.

For `remove_empty_cols` and `remove_empty_rows`, a data frame with "empty" columns or rows removed.

Examples

```
tmp <- data.frame(a = c(1, 2, 3, NA, 5),
                  b = c(1, NA, 3, NA , 5),
                  c = c(NA, NA, NA, NA, NA),
                  d = c(1, NA, 3, NA, 5))

tmp

empty_cols(tmp)
empty_rows(tmp)

remove_empty_cols(tmp)
remove_empty_rows(tmp)
```

find_var *Find variable by name or label*

Description

This functions finds variables in a data frame, which variable names or variable (and value) label attribute match a specific pattern. Regular expression for the pattern is supported.

Usage

```
find_var(data, pattern, ignore.case = TRUE, search = c("name_label",
  "name_value", "label_value", "name", "label", "value", "all"),
  out = c("table", "df", "index"), fuzzy = FALSE, regex = FALSE)

find_in_data(data, pattern, ignore.case = TRUE,
  search = c("name_label", "name_value", "label_value", "name", "label",
  "value", "all"), out = c("table", "df", "index"), fuzzy = FALSE,
  regex = FALSE)
```

Arguments

data	A data frame.
pattern	Character string to be matched in data. May also be a character vector of length > 1 (see 'Examples'). pattern is searched for in column names and variable label attributes of data (see get_label). pattern might also be a regular-expression object, as returned by stringr::regex(). Alternatively, use regex = TRUE to treat pattern as a regular expression rather than a fixed string.
ignore.case	Logical, whether matching should be case sensitive or not. ignore.case is ignored when pattern is no regular expression or regex = FALSE.
search	Character string, indicating where pattern is sought. Use one of following options: "name_label" The default, searches for pattern in variable names and variable labels. "name_value" Searches for pattern in variable names and value labels. "label_value" Searches for pattern in variable and value labels. "name" Searches for pattern in variable names. "label" Searches for pattern in variable labels "value" Searches for pattern in value labels. "all" Searches for pattern in variable names, variable and value labels.
out	Output (return) format of the search results. May be abbreviated and must be one of: "table" A tabular overview (as data frame) with column indices, variable names and labels of matching variables. "df" A data frame with all matching variables.

	"index" A named vector with column indices of all matching variables.
fuzzy	Logical, if TRUE, "fuzzy matching" (partial and close distance matching) will be used to find pattern in data if no exact match was found.
regex	Logical, if TRUE, pattern is treated as a regular expression rather than a fixed string.

Details

This function searches for pattern in data's column names and - for labelled data - in all variable and value labels of data's variables (see get_label for details on variable labels and labelled data). Regular expressions are supported as well, by simply using pattern = stringr::regex(...) or regex = TRUE.

Value

By default (i.e. out = "table", returns a data frame with three columns: column number, variable name and variable label. If out = "index", returns a named vector with column indices of matching variables (variable names are used as names-attribute); if out = "df", returns the matching variables as data frame

Examples

```
data(efc)

# find variables with "cop" in variable name
find_var(efc, "cop")

# return data frame with matching variables
find_var(efc, "cop", out = "df")

# or return column numbers
find_var(efc, "cop", out = "index")

# find variables with "dependency" in names and variable labels
library(sjlabelled)
find_var(efc, "dependency")
get_label(efc$e42dep)

# find variables with "level" in names and value labels
res <- find_var(efc, "level", search = "name_value", out = "df")
res
get_labels(res, attr.only = FALSE)

# use sjPlot::view_df() to view results
## Not run:
library(sjPlot)
view_df(res)
## End(Not run)
```

flat_table *Flat (proportional) tables*

Description

This function creates a labelled flat table or flat proportional (marginal) table.

Usage

```
flat_table(data, ..., margin = c("counts", "cell", "row", "col"),
  digits = 2, show.values = FALSE)
```

Arguments

data	A data frame. May also be a grouped data frame (see 'Note' and 'Examples').
...	One or more variables of data that should be printed as table.
margin	Specify the table margin that should be computed for proportional tables. By default, counts are printed. Use margin = "cell", margin = "col" or margin = "row" to print cell, column or row percentages of the table margins.
digits	Numeric; for proportional tables, digits indicates the number of decimal places.
show.values	Logical, if TRUE, value labels are prefixed by the associated value.

Value

An object of class ftable.

Note

data may also be a grouped data frame (see group_by) with up to two grouping variables. Cross tables are created for each subgroup then.

See Also

frq for simple frequency table of labelled vectors.

Examples

```
data(efc)

# flat table with counts
flat_table(efc, e42dep, c172code, e16sex)

# flat table with proportions
flat_table(efc, e42dep, c172code, e16sex, margin = "row")

# flat table from grouped data frame. You need to select
# the grouping variables and at least two more variables for
```

```
# cross tabulation.
library(dplyr)
efc %>%
  group_by(e16sex) %>%
  select(e16sex, c172code, e42dep) %>%
  flat_table()

efc %>%
  group_by(e16sex, e42dep) %>%
  select(e16sex, e42dep, c172code, n4pstu) %>%
  flat_table()

# now it gets weird...
efc %>%
  group_by(e16sex, e42dep) %>%
  select(e16sex, e42dep, c172code, n4pstu, c161sex) %>%
  flat_table()
```

`frq` *Frequency table of labelled variables*

Description

This function returns a frequency table of labelled vectors, as data frame.

Usage

```
frq(x, ..., sort.frq = c("none", "asc", "desc"), weights = NULL,
  auto.grp = NULL, show.strings = TRUE, show.na = TRUE,
  grp.strings = NULL, min.frq = 0, out = c("txt", "viewer",
  "browser"), title = NULL, encoding = "UTF-8", file = NULL)
```

Arguments

`x`	A vector or a data frame. May also be a grouped data frame (see 'Note' and 'Examples').
`...`	Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
`sort.frq`	Determines whether categories should be sorted according to their frequencies or not. Default is `"none"`, so categories are not sorted by frequency. Use `"asc"` or `"desc"` for sorting categories ascending or descending order.
`weights`	Bare name, or name as string, of a variable in `x` that indicates the vector of weights, which will be applied to weight all observations. Default is `NULL`, so no weights are used.

auto.grp — Numeric value, indicating the minimum amount of unique values in a variable, at which automatic grouping into smaller units is done (see group_var). Default value for auto.group is NULL, i.e. auto-grouping is off.

show.strings — Logical, if TRUE, frequency tables for character vectors will not be printed. This is useful when printing frequency tables of all variables from a data frame, and due to computational reasons character vectors should not be printed.

show.na — Logical, or "auto". If TRUE, the output always contains information on missing values, even if variables have no missing values. If FALSE, information on missing values are removed from the output. If show.na = "auto", information on missing values is only shown when variables actually have missing values, else it's not shown.

grp.strings — Numeric, if not NULL, groups string values in character vectors, based on their similarity. See group_str and str_find for details on grouping, and their precision-argument to get more details on the distance of strings to be treated as equal.

min.frq — Numeric, indicating the minimum frequency for which a value will be shown in the output (except for the missing values, prevailing show.na). Default value for min.frq is 0, so all value frequencies are shown. All values or categories that have less than min.frq occurences in the data will be summarized in a "n < 100" category.

out — Character vector, indicating whether the results should be printed to console (out = "txt") or as HTML-table in the viewer-pane (out = "viewer") or browser (out = "browser").

title — String, will be used as alternative title to the variable label. If x is a grouped data frame, title must be a vector of same length as groups.

encoding — Character vector, indicating the charset encoding used for variable and value labels. Default is "UTF-8". Only used when out is not "txt".

file — Destination file, if the output should be saved as file. Only used when out is not "txt".

Details

The ...-argument not only accepts variable names or expressions from select_helpers. You can also use logical conditions, math operations, or combining variables to produce "crosstables". See 'Examples' for more details.

Value

A list of data frames with values, value labels, frequencies, raw, valid and cumulative percentages of x.

Note

x may also be a grouped data frame (see group_by) with up to two grouping variables. Frequency tables are created for each subgroup then.

The print()-method adds a table header with information on the variable label, variable type,

total and valid N, and mean and standard deviations. Mean and SD are *always* printed, even for categorical variables (factors) or character vectors. In this case, values are coerced into numeric vector to calculate the summary statistics.

See Also

flat_table for labelled (proportional) tables.

Examples

```
# simple vector
data(efc)
frq(efc$e42dep)

# with grouped data frames, in a pipe
library(dplyr)
efc %>%
  group_by(e16sex, c172code) %>%
  frq(e16sex, c172code, e42dep)

# show only categories with a minimal amount of frequencies
frq(mtcars$gear)

frq(mtcars$gear, min.frq = 10)

frq(mtcars$gear, min.frq = 15)

# with select-helpers: all variables from the COPE-Index
# (which all have a "cop" in their name)
frq(efc, contains("cop"))

# all variables from column "c161sex" to column "c175empl"
frq(efc, c161sex:c175empl)

# for non-labelled data, variable name is printed,
# and "label" column is removed from output
data(iris)
frq(iris, Species)

# also works on grouped data frames
efc %>%
  group_by(c172code) %>%
  frq(is.na(nur_pst))

# group variables with large range and with weights
efc$weights <- abs(rnorm(n = nrow(efc), mean = 1, sd = .5))
frq(efc, c160age, auto.grp = 5, weights = weights)

# different weight options
frq(efc, c172code, weights = weights)
frq(efc, c172code, weights = "weights")
frq(efc, c172code, weights = efc$weights)
```

```
frq(efc$c172code, weights = efc$weights)

# group string values
dummy <- efc[1:50, 3, drop = FALSE]
dummy$words <- sample(
  c("Hello", "Helo", "Hole", "Apple", "Ape",
    "New", "Old", "System", "Systemic"),
  size = nrow(dummy),
  replace = TRUE
)

frq(dummy)
frq(dummy, grp.strings = 2)

#### other expressions than variables

# logical conditions
frq(mtcars, cyl ==6)

frq(efc, is.na(nur_pst), contains("cop"))

iris %>%
  frq(starts_with("Petal"), Sepal.Length > 5)

# computation of variables "on the fly"
frq(mtcars, (gear + carb) / cyl)

# crosstables
set.seed(123)
d <- data.frame(
  var_x = sample(letters[1:3], size = 30, replace = TRUE),
  var_y = sample(1:2, size = 30, replace = TRUE),
  var_z = sample(LETTERS[8:10], size = 30, replace = TRUE)
)
table(d$var_x, d$var_z)
frq(d, paste0(var_x, var_z))
frq(d, paste0(var_x, var_y, var_z))
```

---

group_str *Group near elements of string vectors*

Description

This function groups elements of a string vector (character or string variable) according to the element's distance ('similatiry'). The more similar two string elements are, the higher is the chance to be combined into a group.

Usage

```
group_str(strings, precision = 2, strict = FALSE,
  trim.whitespace = TRUE, remove.empty = TRUE, verbose = FALSE,
  maxdist)
```

Arguments

strings	Character vector with string elements.
precision	Maximum distance ("precision") between two string elements, which is allowed to treat them as similar or equal. Smaller values mean less tolerance in matching.
strict	Logical; if TRUE, value matching is more strictly. See 'Examples'.
trim.whitespace	Logical; if TRUE (default), leading and trailing white spaces will be removed from string values.
remove.empty	Logical; if TRUE (default), empty string values will be removed from the character vector strings.
verbose	Logical; if TRUE, the progress bar is displayed when computing the distance matrix. Default in FALSE, hence the bar is hidden.
maxdist	Deprecated. Please use precision now.

Value

A character vector where similar string elements (values) are recoded into a new, single value. The return value is of same length as strings, i.e. grouped elements appear multiple times, so the count for each grouped string is still avaiable (see 'Examples').

See Also

str_find

Examples

```
oldstring <- c("Hello", "Helo", "Hole", "Apple",
               "Ape", "New", "Old", "System", "Systemic")
newstring <- group_str(oldstring)

# see result
newstring

# count for each groups
table(newstring)

# print table to compare original and grouped string
frq(oldstring)
frq(newstring)

# larger groups
newstring <- group_str(oldstring, precision = 3)
```

```
frq(oldstring)
frq(newstring)

# be more strict with matching pairs
newstring <- group_str(oldstring, precision = 3, strict = TRUE)
frq(oldstring)
frq(newstring)
```

---

group_var *Recode numeric variables into equal-ranged groups*

Description

Recode numeric variables into equal ranged, grouped factors, i.e. a variable is cut into a smaller number of groups, where each group has the same value range. `group_labels()` creates the related value labels. `group_var_if()` and `group_labels_if()` are scoped variants of `group_var()` and `group_labels()`, where grouping will be applied only to those variables that match the logical condition of `predicate`.

Usage

```
group_var(x, ..., size = 5, as.num = TRUE, right.interval = FALSE,
  n = 30, append = TRUE, suffix = "_gr")

group_var_if(x, predicate, size = 5, as.num = TRUE,
  right.interval = FALSE, n = 30, append = TRUE, suffix = "_gr")

group_labels(x, ..., size = 5, right.interval = FALSE, n = 30)

group_labels_if(x, predicate, size = 5, right.interval = FALSE,
  n = 30)
```

Arguments

`x`	A vector or data frame.
`...`	Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
`size`	Numeric; group-size, i.e. the range for grouping. By default, for each 5 categories of `x` a new group is defined, i.e. `size = 5`. Use `size = "auto"` to automatically resize a variable into a maximum of 30 groups (which is the ggplot-default grouping when plotting histograms). Use `n` to determine the amount of groups.
`as.num`	Logical, if `TRUE`, return value will be numeric, not a factor.
`right.interval`	Logical; if `TRUE`, grouping starts with the lower bound of `size`. See 'Details'.

n — Sets the maximum number of groups that are defined when auto-grouping is on (size = "auto"). Default is 30. If size is not set to "auto", this argument will be ignored.

append — Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.

suffix — String value, will be appended to variable (column) names of x, if x is a data frame. If x is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call:

- recoded variables (rec()) will be suffixed with "_r"
- recoded variables (recode_to()) will be suffixed with "_r0"
- dichotomized variables (dicho()) will be suffixed with "_d"
- grouped variables (split_var()) will be suffixed with "_g"
- grouped variables (group_var()) will be suffixed with "_gr"
- standardized variables (std()) will be suffixed with "_z"
- centered variables (center()) will be suffixed with "_c"
- de-meaned variables (de_mean()) will be suffixed with "_dm"
- grouped-meaned variables (de_mean()) will be suffixed with "_gm"

If suffix = "" and append = TRUE, existing variables that have been recoded/transformed will be overwritten.

predicate — A predicate function to be applied to the columns. The variables for which predicate returns TRUE are selected.

Details

If size is set to a specific value, the variable is recoded into several groups, where each group has a maximum range of size. Hence, the amount of groups differ depending on the range of x.

If size = "auto", the variable is recoded into a maximum of n groups. Hence, independent from the range of x, always the same amount of groups are created, so the range within each group differs (depending on x's range).

right.interval determins which boundary values to include when grouping is done. If TRUE, grouping starts with the **lower bound** of size. For example, having a variable ranging from 50 to 80, groups cover the ranges from 50-54, 55-59, 60-64 etc. If FALSE (default), grouping starts with the upper bound of size. In this case, groups cover the ranges from 46-50, 51-55, 56-60, 61-65 etc. **Note:** This will cover a range from 46-50 as first group, even if values from 46 to 49 are not present. See 'Examples'.

If you want to split a variable into a certain amount of equal sized groups (instead of having groups where values have all the same range), use the split_var function!

group_var() also works on grouped data frames (see group_by). In this case, grouping is applied to the subsets of variables in x. See 'Examples'.

Value

- For `group_var()`, a grouped variable, either as numeric or as factor (see paramter `as.num`). If `x` is a data frame, only the grouped variables will be returned.
- For `group_labels()`, a string vector or a list of string vectors containing labels based on the grouped categories of `x`, formatted as "from lower bound to upper bound", e.g. `"10-19"` `"20-29"` `"30-39"` etc. See 'Examples'.

Note

Variable label attributes (see, for instance, `set_label`) are preserved. Usually you should use the same values for `size` and `right.interval` in `group_labels()` as used in the `group_var` function if you want matching labels for the related recoded variable.

See Also

`split_var` to split variables into equal sized groups, `group_str` for grouping string vectors or `rec_pattern` and `rec` for another convenient way of recoding variables into smaller groups.

Examples

```
age <- abs(round(rnorm(100, 65, 20)))
age.grp <- group_var(age, size = 10)
hist(age)
hist(age.grp)

age.grpvar <- group_labels(age, size = 10)
table(age.grp)
print(age.grpvar)

# histogram with EUROFAMCARE sample dataset
# variable not grouped
library(sjlabelled)
data(efc)
hist(efc$e17age, main = get_label(efc$e17age))

# bar plot with EUROFAMCARE sample dataset
# grouped variable
ageGrp <- group_var(efc$e17age)
ageGrpLab <- group_labels(efc$e17age)
barplot(table(ageGrp), main = get_label(efc$e17age), names.arg = ageGrpLab)

# within a pipe-chain
library(dplyr)
efc %>%
  select(e17age, c12hour, c160age) %>%
  group_var(size = 20)

# create vector with values from 50 to 80
dummy <- round(runif(200, 50, 80))
# labels with grouping starting at lower bound
group_labels(dummy)
```

```
# labels with grouping startint at upper bound
group_labels(dummy, right.interval = TRUE)

# works also with gouped data frames
mtcars %>%
  group_var(disp, size = 4, append = FALSE) %>%
  table()

mtcars %>%
  group_by(cyl) %>%
  group_var(disp, size = 4, append = FALSE) %>%
  table()
```

---

has_na *Check if variables or cases have missing / infinite values*

Description

This functions checks if variables or observations in a data frame have NA, NaN or Inf values.

Usage

```
has_na(x, ..., by = c("col", "row"), out = c("table", "df", "index"))

incomplete_cases(x, ...)

complete_cases(x, ...)

complete_vars(x, ...)

incomplete_vars(x, ...)
```

Arguments

x	A data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
by	Whether to check column- or row-wise for missing and infinite values. If by = "col", has_na() checks for NA/NaN/Inf in *columns*; If by = "row", has_na() checks each row for these values.
out	Output (return) format of the results. May be abbreviated.

Value

If x is a vector, returns TRUE if x has any missing or infinite values. If x is a data frame, returns TRUE for each variable (if by = "col") or observation (if by = "row") that has any missing or infinite values. If out = "table", results are returned as data frame, with column number, variable name and label, and a logical vector indicating if a variable has missing values or not. However, it's printed in colors, with green rows indicating that a variable has no missings, while red rows indicate the presence of missings or infinite values. If out = "index", a named vector is returned.

Note

complete_cases() and incomplete_cases() are convenient shortcuts for has_na(by = "row",out = "index"), where the first only returns case-id's for all complete cases, and the latter only for non-complete cases.

complete_vars() and incomplete_vars() are convenient shortcuts for has_na(by = "col",out = "index"), and again only return those column-id's for variables which are (in-)complete.

Examples

```
data(efc)
has_na(efc$e42dep)
has_na(efc, e42dep, tot_sc_e, c161sex)
has_na(efc)

has_na(efc, e42dep, tot_sc_e, c161sex, out = "index")
has_na(efc, out = "df")

has_na(efc, by = "row")
has_na(efc, e42dep, tot_sc_e, c161sex, by = "row", out = "index")
has_na(efc, by = "row", out = "df")

complete_cases(efc, e42dep, tot_sc_e, c161sex)
incomplete_cases(efc, e42dep, tot_sc_e, c161sex)
complete_vars(efc, e42dep, tot_sc_e, c161sex)
incomplete_vars(efc, e42dep, tot_sc_e, c161sex)
```

---

is_crossed *Check whether two factors are crossed or nested*

Description

These functions checks whether two factors are (fully) crossed or nested, i.e. if each level of one factor occurs in combination with each level of the other factor (is_crossed()) resp. if each category of the first factor co-occurs with only one category of the other (is_nested()). is_cross_classified() checks if one factor level occurs in some, but not all levels of another factor.

Usage

```
is_crossed(f1, f2)

is_nested(f1, f2)

is_cross_classified(f1, f2)
```

Arguments

f1	Numeric vector or factor.
f2	Numeric vector or factor.

Value

Logical. For `is_crossed()`, `TRUE` if factors are (fully) crossed, `FALSE` otherwise. For `is_nested()`, `TRUE` if factors are nested, `FALSE` otherwise. For `is_cross_classified()`, `TRUE`, if one factor level occurs in some, but not all levels of another factor.

Note

If factors are nested, a message is displayed to tell whether `f1` is nested within `f2` or vice versa.

References

Grace, K. The Difference Between Crossed and Nested Factors. (web)

Examples

```
# crossed factors, each category of
# x appears in each category of y
x <- c(1,4,3,2,3,2,1,4)
y <- c(1,1,1,2,2,1,2,2)
# show distribution
table(x, y)
# check if crossed
is_crossed(x, y)

# not crossed factors
x <- c(1,4,3,2,3,2,1,4)
y <- c(1,1,1,2,1,1,2,2)
# show distribution
table(x, y)
# check if crossed
is_crossed(x, y)


# nested factors, each category of
# x appears in one category of y
x <- c(1,2,3,4,5,6,7,8,9)
y <- c(1,1,1,2,2,2,3,3,3)
# show distribution
```

```
table(x, y)
# check if nested
is_nested(x, y)
is_nested(y, x)

# not nested factors
x <- c(1,2,3,4,5,6,7,8,9,1,2)
y <- c(1,1,1,2,2,2,3,3,3,2,3)
# show distribution
table(x, y)
# check if nested
is_nested(x, y)
is_nested(y, x)

# also not fully crossed
is_crossed(x, y)

# but partially crossed
is_cross_classified(x, y)
```

`is_empty` *Check whether string, list or vector is empty*

Description

This function checks whether a string or character vector (of length 1), a list or any vector (numeric, atomic) is empty or not.

Usage

```
is_empty(x, first.only = TRUE, all.na.empty = TRUE)
```

Arguments

`x`	String, character vector, list, data.frame or numeric vector or factor.
`first.only`	Logical, if `FALSE` and `x` is a character vector, each element of `x` will be checked if empty. If `TRUE`, only the first element of `x` will be checked.
`all.na.empty`	Logical, if `x` is a vector with `NA`-values only, `is_empty` will return `FALSE` if `all.na.empty = FALSE`, and will return `TRUE` if `all.na.empty = TRUE` (default).

Value

Logical, `TRUE` if `x` is a character vector or string and is empty, `TRUE` if `x` is a vector or list and of length 0, `FALSE` otherwise.

Note

`NULL`- or `NA`-values are also considered as "empty" (see 'Examples') and will return `TRUE`, unless `all.na.empty==FALSE`.

Examples

```
is_empty("test")
is_empty("")
is_empty(NA)
is_empty(NULL)

# string is not empty
is_empty(" ")

# however, this trimmed string is
is_empty(trim(" "))

# numeric vector
x <- 1
is_empty(x)
x <- x[-1]
is_empty(x)

# check multiple elements of character vectors
is_empty(c("", "a"))
is_empty(c("", "a"), first.only = FALSE)

# empty data frame
d <- data.frame()
is_empty(d)

# empty list
is_empty(list(NULL))

# NA vector
x <- rep(NA,5)
is_empty(x)
is_empty(x, all.na.empty = FALSE)
```

`is_even` *Check whether value is even or odd*

Description

Checks whether `x` is an even or odd number. Only accepts numeric vectors.

Usage

```
is_even(x)

is_odd(x)
```

Arguments

x	Numeric vector or single numeric value, or a data frame or list with such vectors.

Value

`is_even()` returns `TRUE` for each even value of `x`, `FALSE` for odd values. `is_odd()` returns `TRUE` for each odd value of `x` and `FALSE` for even values.

Examples

```
is_even(4)
is_even(5)
is_even(1:4)

is_odd(4)
is_odd(5)
is_odd(1:4)
```

---

is_float *Check if a variable is of (non-integer) double type or a whole number*

Description

`is_float()` checks whether an input vector or value is a numeric non-integer (double), depending on fractional parts of the value(s). `is_whole()` does the opposite and checks whether an input vector is a whole number (without fractional parts).

Usage

```
is_float(x)

is_whole(x)
```

Arguments

x	A value, vector or data frame.

Value

For `is_float()`, `TRUE` if `x` is a floating value (non-integer double), `FALSE` otherwise (also returns `FALSE` for character vectors and factors). For `is_whole()`, `TRUE` if `x` is a vector with whole numbers only, `FALSE` otherwise (returns `TRUE` for character vectors and factors).

Examples

```
data(mtcars)
data(iris)

is.double(4)
is_float(4)
is_float(4.2)
is_float(iris)

is_whole(4)
is_whole(4.2)
is_whole(mtcars)
```

`is_num_fac` *Check whether a factor has numeric levels only*

Description

`is_num_fac()` checks whether a factor has only numeric or any non-numeric factor levels, while `is_num_chr()` checks whether a character vector has only numeric strings.

Usage

```
is_num_fac(x)

is_num_chr(x)
```

Arguments

`x`	A factor for `is_num_fac()` and a character vector for `is_num_chr()`

Value

Logical, `TRUE` if factor has numeric factor levels only, or if character vector has numeric strings only, `FALSE` otherwise.

Examples

```
# numeric factor levels
f1 <- factor(c(NA, 1, 3, NA, 2, 4))
is_num_fac(f1)

# not completeley numeric factor levels
f2 <- factor(c(NA, "C", 1, 3, "A", NA, 2, 4))
is_num_fac(f2)

# not completeley numeric factor levels
```

```
f3 <- factor(c("Justus", "Bob", "Peter"))
is_num_fac(f3)

is_num_chr(c("a", "1"))
is_num_chr(c("2", "1"))
```

`merge_imputations` *Merges multiple imputed data frames into a single data frame*

Description

This function merges multiple imputed data frames from `mice::mids()`-objects into a single data frame by computing the mean or selecting the most likely imputed value.

Usage

```
merge_imputations(dat, imp, ori = NULL, summary = c("none", "dens",
  "hist", "sd"), filter = NULL)
```

Arguments

`dat`	The data frame that was imputed and used as argument in the `mice`-function call.
`imp`	The `mice::mids()`-object with the imputed data frames from `dat`.
`ori`	Optional, if `ori` is specified, the imputed variables are appended to this data frame; else, a new data frame with the imputed variables is returned.
`summary`	After merging multiple imputed data, `summary` displays a graphical summary of the "quality" of the merged values, compared to the original imputed values.
	`"dens"` Creates a density plot, which shows the distribution of the mean of the imputed values for each variable at each observation. The larger the areas overlap, the better is the fit of the merged value compared to the imputed value.
	`"hist"` Similar to `summary = "dens"`, however, mean and merged values are shown as histogram. Bins should have almost equal height for both groups (mean and merged).
	`"sd"` Creates a dot plot, where data points indicate the standard deviation for all imputed values (y-axis) at each merged value (x-axis) for all imputed variables. The higher the standard deviation, the less precise is the imputation, and hence the merged value.
`filter`	A character vector with variable names that should be plotted. All non-defined variables will not be shown in the plot.

Details

This method merges multiple imputations of variables into a single variable by computing the (rounded) mean of all imputed values of missing values. By this, each missing value is replaced by those values that have been imputed the most times.

`imp` must be a `mids`-object, which is returned by the `mice()`-function of the **mice**-package. `merge_imputations()` than creates a data frame for each imputed variable, by combining all imputations (as returned by the `complete`-function) of each variable, and computing the row means of this data frame. The mean value is then rounded for integer values (and not for numerical values with fractional part), which corresponds to the most frequent imputed value (mode) for a missing value. Missings in the original variable are replaced by the most frequent imputed value.

Value

A data frame with (merged) imputed variables; or `ori` with appended imputed variables, if `ori` was specified. If `summary` is included, returns a list with the data frame `data` with (merged) imputed variables and some other summary information, which are required for the plot-output.

Note

Typically, further analyses are conducted on pooled results of multiple imputed data sets (see `pool`), however, sometimes (in social sciences) it is also feasible to compute the mean or mode of multiple imputed variables (see *Burns et al. 2011*).

References

Burns RA, Butterworth P, Kiely KM, Bielak AAM, Luszcz MA, Mitchell P, et al. 2011. Multiple imputation was an efficient method for harmonizing the Mini-Mental State Examination with missing item-level data. Journal of Clinical Epidemiology;64:787-93 doi: 10.1016/j.jclinepi.2010.10.011

Examples

```
library(mice)
imp <- mice(nhanes)

# return data frame with imputed variables
merge_imputations(nhanes, imp)

# append imputed variables to original data frame
merge_imputations(nhanes, imp, nhanes)

# show summary of quality of merging imputations
merge_imputations(nhanes, imp, summary = "dens", filter = c("chl", "hyp"))
```

`move_columns` *Move columns to other positions in a data frame*

Description

`move_columns()` moves one or more columns in a data frame to another position.

Usage

```
move_columns(data, ..., .before, .after)
```

Arguments

`data`	A data frame.
`...`	Unquoted names or character vector with names of variables that should be move to another position. You may also use functions like : or tidyselect's `select_helpers`.
`.before`	Optional, column name or numeric index of the position where `col` should be moved to. If not missing, `col` is moved to the position *before* the column indicated by `.before`.
`.after`	Optional, column name or numeric index of the position where `col` should be moved to. If not missing, `col` is moved to the position *after* the column indicated by `.after`.

Value

`data`, with resorted columns.

Note

If neither `.before` nor `.after` are specified, the column is moved to the end of the data frame by default.

Examples

```
## Not run:
data(iris)

iris %>%
  move_columns(Sepal.Width, .after = "Species") %>%
  head()

iris %>%
  move_columns(Sepal.Width, .before = Sepal.Length) %>%
  head()

iris %>%
  move_columns(Species, .before = 1) %>%
```

```
  head()

iris %>%
  move_columns("Species", "Petal.Length", .after = 1) %>%
  head()

library(dplyr)
iris %>%
  move_columns(contains("Width"), .after = "Species") %>%
  head()
## End(Not run)
```

`numeric_to_factor` *Convert numeric vectors into factors associated value labels*

Description

This function converts numeric variables into factors, and uses associated value labels as factor levels.

Usage

```
numeric_to_factor(x, n = 4)
```

Arguments

x	A data frame.
n	Numeric, indicating the maximum amount of unique values in x to be considered as "factor". Variables with more unique values than n are not converted to factor.

Details

If x is a labelled vector, associated value labels will be used as level. Else, the numeric vector is simply coerced using `as.factor()`.

Value

x, with numeric values with a maximum of n unique values being converted to factors.

Examples

```
library(dplyr)
data(efc)
efc %>%
  select(e42dep, e16sex, c12hour, c160age, c172code) %>%
  numeric_to_factor()
```

rec *Recode variables*

Description

rec() recodes values of variables, where variable selection is based on variable names or column position, or on select helpers (see documentation on ...). rec_if() is a scoped variant of rec(), where recoding will be applied only to those variables that match the logical condition of predicate.

Usage

```
rec(x, ..., rec, as.num = TRUE, var.label = NULL, val.labels = NULL,
  append = TRUE, suffix = "_r")

rec_if(x, predicate, rec, as.num = TRUE, var.label = NULL,
  val.labels = NULL, append = TRUE, suffix = "_r")
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
rec	String with recode pairs of old and new values. See 'Details' for examples. rec_pattern is a convenient function to create recode strings for grouping variables.
as.num	Logical, if TRUE, return value will be numeric, not a factor.
var.label	Optional string, to set variable label attribute for the returned variable (see vignette Labelled Data and the sjlabelled-Package). If NULL (default), variable label attribute of x will be used (if present). If empty, variable label attributes will be removed.
val.labels	Optional character vector, to set value label attributes of recoded variable (see vignette Labelled Data and the sjlabelled-Package). If NULL (default), no value labels will be set. Value labels can also be directly defined in the rec-syntax, see 'Details'.
append	Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.
suffix	String value, will be appended to variable (column) names of x, if x is a data frame. If x is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call: • recoded variables (rec()) will be suffixed with "_r" • recoded variables (recode_to()) will be suffixed with "_r0"

- dichotomized variables (`dicho()`) will be suffixed with `"_d"`
- grouped variables (`split_var()`) will be suffixed with `"_g"`
- grouped variables (`group_var()`) will be suffixed with `"_gr"`
- standardized variables (`std()`) will be suffixed with `"_z"`
- centered variables (`center()`) will be suffixed with `"_c"`
- de-meaned variables (`de_mean()`) will be suffixed with `"_dm"`
- grouped-meaned variables (`de_mean()`) will be suffixed with `"_gm"`

If `suffix = ""` and `append = TRUE`, existing variables that have been recoded/transformed will be overwritten.

`predicate` A predicate function to be applied to the columns. The variables for which `predicate` returns `TRUE` are selected.

Details

The `rec` string has following syntax:

recode pairs each recode pair has to be separated by a `;`, e.g. `rec = "1=1; 2=4; 3=2; 4=3"`

multiple values multiple old values that should be recoded into a new single value may be separated with comma, e.g. `"1,2=1; 3,4=2"`

value range a value range is indicated by a colon, e.g. `"1:4=1; 5:8=2"` (recodes all values from 1 to 4 into 1, and from 5 to 8 into 2)

value range for doubles for double vectors (with fractional part), all values within the specified range are recoded; e.g. `1:2.5=1;2.6:3=2` recodes 1 to 2.5 into 1 and 2.6 to 3 into 2, but 2.55 would not be recoded (since it's not included in any of the specified ranges)

`"min"` **and** `"max"` minimum and maximum values are indicates by *min* (or *lo*) and *max* (or *hi*), e.g. `"min:4=1; 5:max=2"` (recodes all values from minimum values of `x` to 4 into 1, and from 5 to maximum values of `x` into 2)

`"else"` all other values, which have not been specified yet, are indicated by *else*, e.g. `"3=1; 1=2; else=3"` (recodes 3 into 1, 1 into 2 and all other values into 3)

`"copy"` the `"else"`-token can be combined with *copy*, indicating that all remaining, not yet recoded values should stay the same (are copied from the original value), e.g. `"3=1; 1=2; else=copy"` (recodes 3 into 1, 1 into 2 and all other values like 2, 4 or 5 etc. will not be recoded, but copied, see 'Examples')

`NA`**'s** `NA` values are allowed both as old and new value, e.g. `"NA=1; 3:5=NA"` (recodes all NA into 1, and all values from 3 to 5 into NA in the new variable)

`"rev"` `"rev"` is a special token that reverses the value order (see 'Examples')

direct value labelling value labels for new values can be assigned inside the recode pattern by writing the value label in square brackets after defining the new value in a recode pair, e.g. `"15:30=1 [young aged]; 31:55=2 [middle aged]; 56:max=3 [old aged]"`. See 'Examples'.

Value

`x` with recoded categories. If `x` is a data frame, for `append = TRUE`, `x` including the recoded variables as new columns is returned; if `append = FALSE`, only the recoded variables will be returned. If `append = TRUE` and `suffix = ""`, recoded variables will replace (overwrite) existing variables.

Note

Please note following behaviours of the function:

- the "else"-token should always be the last argument in the rec-string.
- Non-matching values will be set to NA, unless captured by the "else"-token.
- Tagged NA values (see tagged_na) and their value labels will be preserved when copying NA values to the recoded vector with "else=copy".
- Variable label attributes (see, for instance, get_label) are preserved (unless changed via var.label-argument), however, value label attributes are removed (except for "rev", where present value labels will be automatically reversed as well). Use val.labels-argument to add labels for recoded values.
- If x is a data frame, all variables should have the same categories resp. value range (else, see second bullet, NAs are produced).

See Also

set_na for setting NA values, replace_na to replace NA's with specific value, recode_to for re-shifting value ranges and ref_lvl to change the reference level of (numeric) factors.

Examples

```
data(efc)
table(efc$e42dep, useNA = "always")

# replace NA with 5
table(rec(efc$e42dep, rec = "1=1;2=2;3=3;4=4;NA=5"), useNA = "always")

# recode 1 to 2 into 1 and 3 to 4 into 2
table(rec(efc$e42dep, rec = "1,2=1; 3,4=2"), useNA = "always")

# keep value labels. variable label is automatically preserved
library(dplyr)
efc %>%
  select(e42dep) %>%
  rec(rec = "1,2=1; 3,4=2",
      val.labels = c("low dependency", "high dependency")) %>%
  str()

# works with mutate
efc %>%
  select(e42dep, e17age) %>%
  mutate(dependency_rev = rec(e42dep, rec = "rev")) %>%
  head()

# recode 1 to 3 into 4 into 2
table(rec(efc$e42dep, rec = "min:3=1; 4=2"), useNA = "always")

# recode 2 to 1 and all others into 2
table(rec(efc$e42dep, rec = "2=1; else=2"), useNA = "always")
```

```
# reverse value order
table(rec(efc$e42dep, rec = "rev"), useNA = "always")

# recode only selected values, copy remaining
table(efc$e15relat)
table(rec(efc$e15relat, rec = "1,2,4=1; else=copy"))

# recode variables with same category in a data frame
head(efc[, 6:9])
head(rec(efc[, 6:9], rec = "1=10;2=20;3=30;4=40"))

# recode multiple variables and set value labels via recode-syntax
dummy <- rec(
  efc, c160age, e17age,
  rec = "15:30=1 [young]; 31:55=2 [middle]; 56:max=3 [old]",
  append = FALSE
)
frq(dummy)

# recode variables with same value-range
lapply(
  rec(
    efc, c82cop1, c83cop2, c84cop3,
    rec = "1,2=1; NA=9; else=copy",
    append = FALSE
  ),
  table,
  useNA = "always"
)

# recode character vector
dummy <- c("M", "F", "F", "X")
rec(dummy, rec = "M=Male; F=Female; X=Refused")

# recode numeric to character
rec(efc$e42dep, rec = "1=first;2=2nd;3=third;else=hi")

# recode non-numeric factors
data(iris)
table(rec(iris, Species, rec = "setosa=huhu; else=copy", append = FALSE))

# recode floating points
table(rec(
  iris, Sepal.Length, rec = "lo:5=1;5.01:6.5=2;6.501:max=3", append = FALSE
))

# preserve tagged NAs
library(haven)
x <- labelled(c(1:3, tagged_na("a", "c", "z"), 4:1),
              c("Agreement" = 1, "Disagreement" = 4, "First" = tagged_na("c"),
                "Refused" = tagged_na("a"), "Not home" = tagged_na("z")))
# get current value labels
x
```

```
# recode 2 into 5; Values of tagged NAs are preserved
rec(x, rec = "2=5;else=copy")
na_tag(rec(x, rec = "2=5;else=copy"))

# use select-helpers from dplyr-package
rec(
  efc, contains("cop"), c161sex:c175empl,
  rec = "0,1=0; else=1",
  append = FALSE
)

# recode only variables that have a value range from 1-4
p <- function(x) min(x, na.rm = TRUE) > 0 && max(x, na.rm = TRUE) < 5
rec_if(efc, predicate = p, rec = "1:3=1;4=2;else=copy")
```

---

recode_to *Recode variable categories into new values*

Description

Recodes (or "renumbers") the categories of variables into new category values, beginning with the lowest value specified by `lowest`. Useful when recoding dummy variables with 1/2 values to 0/1 values, or recoding scales from 1-4 to 0-3 etc. `recode_to_if()` is a scoped variant of `recode_to()`, where recoding will be applied only to those variables that match the logical condition of `predicate`.

Usage

```
recode_to(x, ..., lowest = 0, highest = -1, append = TRUE,
  suffix = "_r0")

recode_to_if(x, predicate, lowest = 0, highest = -1, append = TRUE,
  suffix = "_r0")
```

Arguments

`x`	A vector or data frame.
`...`	Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
`lowest`	Indicating the lowest category value for recoding. Default is 0, so the new variable starts with value 0.
`highest`	If specified and greater than `lowest`, all category values larger than `highest` will be set to `NA`. Default is `-1`, i.e. this argument is ignored and no NA's will be produced.

append — Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.

suffix — String value, will be appended to variable (column) names of x, if x is a data frame. If x is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call:

- recoded variables (rec()) will be suffixed with "_r"
- recoded variables (recode_to()) will be suffixed with "_r0"
- dichotomized variables (dicho()) will be suffixed with "_d"
- grouped variables (split_var()) will be suffixed with "_g"
- grouped variables (group_var()) will be suffixed with "_gr"
- standardized variables (std()) will be suffixed with "_z"
- centered variables (center()) will be suffixed with "_c"
- de-meaned variables (de_mean()) will be suffixed with "_dm"
- grouped-meaned variables (de_mean()) will be suffixed with "_gm"

If suffix = "" and append = TRUE, existing variables that have been recoded/transformed will be overwritten.

predicate — A predicate function to be applied to the columns. The variables for which predicate returns TRUE are selected.

Value

x with recoded category values, where lowest indicates the lowest value; If x is a data frame, for append = TRUE, x including the recoded variables as new columns is returned; if append = FALSE, only the recoded variables will be returned. If append = TRUE and suffix = "", recoded variables will replace (overwrite) existing variables.

Note

Value and variable label attributes are preserved.

See Also

rec for general recoding of variables and set_na for setting NA values.

Examples

```
# recode 1-4 to 0-3
dummy <- sample(1:4, 10, replace = TRUE)
recode_to(dummy)

# recode 3-6 to 0-3
# note that numeric type is returned
dummy <- as.factor(3:6)
recode_to(dummy)

# lowest value starting with 1
dummy <- sample(11:15, 10, replace = TRUE)
recode_to(dummy, lowest = 1)
```

```
# lowest value starting with 1, highest with 3
# all others set to NA
dummy <- sample(11:15, 10, replace = TRUE)
recode_to(dummy, lowest = 1, highest = 3)

# recode multiple variables at once
data(efc)
recode_to(efc, c82cop1, c83cop2, c84cop3, append = FALSE)

library(dplyr)
efc %>%
  select(c82cop1, c83cop2, c84cop3) %>%
  mutate(
    c82new = recode_to(c83cop2, lowest = 5),
    c83new = recode_to(c84cop3, lowest = 3)
  ) %>%
  head()
```

`rec_pattern` *Create recode pattern for 'rec' function*

Description

Convenient function to create a recode pattern for the `rec` function, which recodes (numeric) vectors into smaller groups.

Usage

```
rec_pattern(from, to, width = 5, other = NULL)
```

Arguments

`from`	Minimum value that should be recoded.
`to`	Maximum value that should be recoded.
`width`	Numeric, indicating the range of each group.
`other`	String token, indicating how to deal with all other values that have not been captured by the recode pattern. See 'Details' on the `else`-token in `rec`.

Value

A list with two values:

`pattern` string pattern that can be used as `rec` argument for the `rec`-function.

`labels` the associated values labels that can be used with `set_labels`.

See Also

group_var for recoding variables into smaller groups, and group_labels to create the asssociated value labels.

Examples

```
rp <- rec_pattern(1, 100)
rp

# sample data, inspect age of carers
data(efc)
table(efc$c160age, exclude = NULL)
table(rec(efc$c160age, rec = rp$pattern), exclude = NULL)

# recode carers age into groups of width 5
x <- rec(
  efc$c160age,
  rec = rp$pattern,
  val.labels = rp$labels
)
# watch result
frq(x)
```

---

ref_lvl *Change reference level of (numeric) factors*

Description

Changes the reference level of (numeric) factor.

Usage

```
ref_lvl(x, ..., lvl = NULL)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
lvl	Eiher numeric, indicating the new reference level, or a string, indicating the value label from the new reference level. If x is a factor with non-numeric factor levels, relevel(x,ref = lvl) is returned. See 'Examples'.

Details

Unlike relevel, this function behaves differently for factor with numeric factor levels or for labelled data, i.e. factors with value labels for the values. ref_lvl() changes the reference level by recoding the factor's values using the rec function. Hence, all values from lowest up to the reference level indicated by lvl are recoded, with lvl starting as lowest factor value. For factors with non-numeric factor levels, the function simply returns relevel(x,ref = lvl). See 'Examples'.

Value

x with new reference level. If x is a data frame, the complete data frame x will be returned, where variables specified in ... will be re-leveled; if ... is not specified, applies to all variables in the data frame.

See Also

to_factor to convert numeric vectors into factors; rec to recode variables.

Examples

```
data(efc)
x <- to_factor(efc$e42dep)
str(x)
frq(x)

# see column "val" in frq()-output, which indicates
# how values/labels were recoded after using ref_lvl()
x <- ref_lvl(x, lvl = 3)
str(x)
frq(x)

library(dplyr)
dat <- efc %>%
  select(c82cop1, c83cop2, c84cop3) %>%
  to_factor()

frq(dat)
ref_lvl(dat, c82cop1, c83cop2, lvl = 2) %>% frq()

# compare numeric and string value for "lvl"-argument
x <- to_factor(efc$e42dep)
frq(x)
ref_lvl(x, lvl = 2) %>% frq()
ref_lvl(x, lvl = "slightly dependent") %>% frq()

# factors with non-numeric factor levels
data(iris)
levels(iris$Species)
levels(ref_lvl(iris$Species, lvl = 3))
levels(ref_lvl(iris$Species, lvl = "versicolor"))
```

remove_var *Remove variables from a data frame*

Description

This function removes variables from a data frame, and is intended to use within a pipe-workflow. `remove_cols()` is an alias for `remove_var()`.

Usage

```
remove_var(x, ...)

remove_cols(x, ...)
```

Arguments

`x`	A vector or data frame.
`...`	Character vector with variable names, or unquoted names of variables that should be removed from the data frame. You may also use functions like : or tidyselect's `select_helpers`.

Value

`x`, with variables specified in `...` removed.

Examples

```
mtcars %>% remove_var("disp", "cyl")
mtcars %>% remove_var(c("wt", "vs"))
mtcars %>% remove_var(drat:am)
```

replace_na *Replace NA with specific values*

Description

This function replaces (tagged) NA's of a variable, data frame or list of variables with `value`.

Usage

```
replace_na(x, ..., value, na.label = NULL, tagged.na = NULL)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
value	Value that will replace the NA's.
na.label	Optional character vector, used to label the the former NA-value (i.e. adding a labels attribute for value to x).
tagged.na	Optional single character, specifies a tagged_na value that will be replaced by value. Herewith it is possible to replace only specific NA values of x.

Details

While regular NA values can only be *completely* replaced with a single value, tagged_na allows to differentiate between different qualitative values of NAs. Tagged NAs work exactly like regular R missing values except that they store one additional byte of information: a tag, which is usually a letter ("a" to "z") or character number ("0" to "9"). Therewith it is possible to replace only specific NA values, while other NA values are preserved.

Value

x, where NA's are replaced with value. If x is a data frame, the complete data frame x will be returned, with replaced NA's for variables specified in ...; if ... is not specified, applies to all variables in the data frame.

Note

Value and variable label attributes are preserved.

See Also

set_na for setting NA values, rec for general recoding of variables and recode_to for re-shifting value ranges.

Examples

```
library(sjlabelled)
data(efc)
table(efc$e42dep, useNA = "always")
table(replace_na(efc$e42dep, value = 99), useNA = "always")

# the original labels
get_labels(replace_na(efc$e42dep, value = 99))
# NA becomes "99", and is labelled as "former NA"
get_labels(
  replace_na(efc$e42dep, value = 99, na.label = "former NA"),
  values = "p"
)
```

```
dummy <- data.frame(
  v1 = efc$c82cop1,
  v2 = efc$c83cop2,
  v3 = efc$c84cop3
)
# show original distribution
lapply(dummy, table, useNA = "always")
# show variables, NA's replaced with 99
lapply(replace_na(dummy, v2, v3, value = 99), table, useNA = "always")

library(haven)
x <- labelled(c(1:3, tagged_na("a", "c", "z"), 4:1),
              c("Agreement" = 1, "Disagreement" = 4, "First" = tagged_na("c"),
                "Refused" = tagged_na("a"), "Not home" = tagged_na("z")))
# get current NA values
x
get_na(x)

# replace only the NA, which is tagged as NA(c)
replace_na(x, value = 2, tagged.na = "c")
get_na(replace_na(x, value = 2, tagged.na = "c"))

table(x)
table(replace_na(x, value = 2, tagged.na = "c"))

# tagged NA also works for non-labelled class
# init vector
x <- c(1, 2, 3, 4)
# set values 2 and 3 as tagged NA
x <- set_na(x, na = c(2, 3), as.tag = TRUE)
# see result
x
# now replace only NA tagged with 2 with value 5
replace_na(x, value = 5, tagged.na = "2")
```

`reshape_longer` *Reshape data into long format*

Description

`reshape_longer()` reshapes one or more columns from wide into long format.

Usage

```
reshape_longer(x, columns = colnames(x), names.to = "key",
  values.to = "value", labels = NULL, numeric.timevar = FALSE,
  id = ".id")
```

Arguments

x	A data frame.
columns	Names of variables (as character vector), or column index of variables, that should be reshaped. If multiple column groups should be reshaped, use a list of vectors (see 'Examples').
names.to	Character vector with name(s) of key column(s) to create in output. Either one name per column group that should be gathered, or a single string. In the latter case, this name will be used as key column, and only one key column is created.
values.to	Character vector with names of value columns (variable names) to create in output. Must be of same length as number of column groups that should be gathered. See 'Examples'.
labels	Character vector of same length as values.to with variable labels for the new variables created from gathered columns. See 'Examples'.
numeric.timevar	Logical, if TRUE, the values of the names.to column will be recoded to numeric values, in sequential ascending order.
id	Name of ID-variable.

Value

A reshaped data frame.

See Also

to_long

Examples

```
# Reshape one column group into long format
mydat <- data.frame(
  age = c(20, 30, 40),
  sex = c("Female", "Male", "Male"),
  score_t1 = c(30, 35, 32),
  score_t2 = c(33, 34, 37),
  score_t3 = c(36, 35, 38)
)

reshape_longer(
  mydat,
  columns = c("score_t1", "score_t2", "score_t3"),
  names.to = "time",
  values.to = "score"
)


# Reshape multiple column groups into long format
mydat <- data.frame(
  age = c(20, 30, 40),
  sex = c("Female", "Male", "Male"),
```

```
  score_t1 = c(30, 35, 32),
  score_t2 = c(33, 34, 37),
  score_t3 = c(36, 35, 38),
  speed_t1 = c(2, 3, 1),
  speed_t2 = c(3, 4, 5),
  speed_t3 = c(1, 8, 6)
)

reshape_longer(
  mydat,
  columns = list(
    c("score_t1", "score_t2", "score_t3"),
    c("speed_t1", "speed_t2", "speed_t3")
  ),
  names.to = "time",
  values.to = c("score", "speed")
)

# or ...
reshape_longer(
  mydat,
  list(3:5, 6:8),
  names.to = "time",
  values.to = c("score", "speed")
)

# gather multiple columns, label columns
x <- reshape_longer(
  mydat,
  list(3:5, 6:8),
  names.to = "time",
  values.to = c("score", "speed"),
  labels = c("Test Score", "Time needed to finish")
)

library(sjlabelled)
str(x$score)
get_label(x$speed)
```

`rotate_df` *Rotate a data frame*

Description

This function rotates a data frame, i.e. columns become rows and vice versa.

Usage

```
rotate_df(x, rn = NULL, cn = FALSE)
```

Arguments

x	A data frame.
rn	Character vector (optional). If not NULL, the data frame's rownames will be added as (first) column to the output, with rn being the name of this column.
cn	Logical (optional), if TRUE, the values of the first column in x will be used as column names in the rotated data frame.

Value

A (rotated) data frame.

Examples

```
x <- mtcars[1:3, 1:4]
rotate_df(x)
rotate_df(x, rn = "property")

# use values in 1. column as column name
rotate_df(x, cn = TRUE)
rotate_df(x, rn = "property", cn = TRUE)

# also works on list-results
library(purrr)

dat <- mtcars[1:3, 1:4]
tmp <- purrr::map(dat, function(x) {
  sdev <- stats::sd(x, na.rm = TRUE)
  ulsdev <- mean(x, na.rm = TRUE) + c(-sdev, sdev)
  names(ulsdev) <- c("lower_sd", "upper_sd")
  ulsdev
})
tmp
as.data.frame(tmp)
rotate_df(tmp)

tmp <- purrr::map_df(dat, function(x) {
  sdev <- stats::sd(x, na.rm = TRUE)
  ulsdev <- mean(x, na.rm = TRUE) + c(-sdev, sdev)
  names(ulsdev) <- c("lower_sd", "upper_sd")
  ulsdev
})
tmp
rotate_df(tmp)
```

---

round_num *Round numeric variables in a data frame*

Description

round_num() rounds numeric variables in a data frame that also contains non-numeric variables. Non-numeric variables are ignored.

Usage

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

Arguments

x	A vector or data frame.
digits	Numeric, number of decimals to round to.

Value

x with all numeric variables rounded.

Examples

```
data(iris)
round_num(iris)
```

---

row_count *Count row or column indices*

Description

row_count() mimics base R's rowSums(), with sums for a specific value indicated by count. Hence, it is equivalent to rowSums(x == count,na.rm = TRUE). However, this function is designed to work nicely within a pipe-workflow and allows select-helpers for selecting variables and the return value is always a data frame (with one variable).

col_count() does the same for columns. The return value is a data frame with one row (the column counts) and the same number of columns as x.

Usage

```
row_count(x, ..., count, var = "rowcount", append = TRUE)

col_count(x, ..., count, var = "colcount", append = TRUE)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
count	The value for which the row or column sum should be computed. May be a numeric value, a character string (for factors or character vectors), NA, Inf or NULL to count missing or infinite values, or null-values.
var	Name of new the variable with the row or column counts.
append	Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.

Value

For row_count(), a data frame with one variable: the sum of count appearing in each row of x; for col_count(), a data frame with one row and the same number of variables as in x: each variable holds the sum of count appearing in each variable of x. If append = TRUE, x including this variable will be returned.

Examples

```
dat <- data.frame(
  c1 = c(1, 2, 3, 1, 3, NA),
  c2 = c(3, 2, 1, 2, NA, 3),
  c3 = c(1, 1, 2, 1, 3, NA),
  c4 = c(1, 1, 3, 2, 1, 2)
)

row_count(dat, count = 1, append = FALSE)
row_count(dat, count = NA, append = FALSE)
row_count(dat, c1:c3, count = 2, append = TRUE)

col_count(dat, count = 1, append = FALSE)
col_count(dat, count = NA, append = FALSE)
col_count(dat, c1:c3, count = 2, append = TRUE)
```

---

row_sums *Row sums and means for data frames*

Description

row_sums() and row_means() compute row sums or means for at least n valid values per row. The functions are designed to work nicely within a pipe-workflow and allow select-helpers for selecting variables.

Usage

```
row_sums(x, ...)

## Default S3 method:
row_sums(x, ..., n, var = "rowsums", append = TRUE)

## S3 method for class 'mids'
row_sums(x, ..., var = "rowsums", append = TRUE)

row_means(x, ...)

total_mean(x, ...)

## Default S3 method:
row_means(x, ..., n, var = "rowmeans", append = TRUE)

## S3 method for class 'mids'
row_means(x, ..., var = "rowmeans", append = TRUE)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
n	May either be • a numeric value that indicates the amount of valid values per row to calculate the row mean or sum; • a value between 0 and 1, indicating a proportion of valid values per row to calculate the row mean or sum (see 'Details'). • or Inf. If n = Inf, all values per row must be non-missing to compute row mean or sum. If a row's sum of valid (i.e. non-NA) values is less than n, NA will be returned as value for the row mean or sum.
var	Name of new the variable with the row sums or means.
append	Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.

Details

For n, must be a numeric value from 0 to ncol(x). If a *row* in x has at least n non-missing values, the row mean or sum is returned. If n is a non-integer value from 0 to 1, n is considered to indicate the proportion of necessary non-missing values per row. E.g., if n = .75, a row must have at least ncol(x) * n non-missing values for the row mean or sum to be calculated. See 'Examples'.

Value

For row_sums(), a data frame with a new variable: the row sums from x; for row_means(), a data frame with a new variable: the row means from x. If append = FALSE, only the new variable with row sums resp. row means is returned. total_mean() returns the mean of all values from all specified columns in a data frame.

Examples

```
data(efc)
efc %>% row_sums(c82cop1:c90cop9, n = 3, append = FALSE)

library(dplyr)
row_sums(efc, contains("cop"), n = 2, append = FALSE)

dat <- data.frame(
  c1 = c(1,2,NA,4),
  c2 = c(NA,2,NA,5),
  c3 = c(NA,4,NA,NA),
  c4 = c(2,3,7,8),
  c5 = c(1,7,5,3)
)
dat

row_means(dat, n = 4)
row_sums(dat, n = 4)

row_means(dat, c1:c4, n = 4)
# at least 40% non-missing
row_means(dat, c1:c4, n = .4)
row_sums(dat, c1:c4, n = .4)

# total mean of all values in the data frame
total_mean(dat)

# create sum-score of COPE-Index, and append to data
efc %>%
  select(c82cop1:c90cop9) %>%
  row_sums(n = 1)

# if data frame has only one column, this column is returned
row_sums(dat[, 1, drop = FALSE], n = 0)
```

---

seq_col *Sequence generation for column or row counts of data frames*

Description

seq_col(x) is a convenient wrapper for seq_len(ncol(x)), while seq_row(x) is a convenient wrapper for seq_len(nrow(x)).

Usage

```
seq_col(x)

seq_row(x)
```

Arguments

x	A data frame.

Value

A numeric sequence from 1 to number of columns or rows.

Examples

```
data(iris)
seq_col(iris)
seq_row(iris)
```

---

set_na_if *Replace specific values in vector with NA*

Description

`set_na_if()` is a scoped variant of `set_na`, where values will be replaced only with NA's for those variables that match the logical condition of `predicate`.

Usage

```
set_na_if(x, predicate, na, drop.levels = TRUE, as.tag = FALSE)
```

Arguments

x	A vector or data frame.
predicate	A predicate function to be applied to the columns. The variables for which `predicate` returns `TRUE` are selected.
na	Numeric vector with values that should be replaced with NA values, or a character vector if values of factors or character vectors should be replaced. For labelled vectors, may also be the name of a value label. In this case, the associated values for the value labels in each vector will be replaced with `NA`. `na` can also be a named vector. If `as.tag = FALSE`, values will be replaced only in those variables that are indicated by the value names (see 'Examples').
drop.levels	Logical, if `TRUE`, factor levels of values that have been replaced with `NA` are dropped. See 'Examples'.
as.tag	Logical, if `TRUE`, values in `x` will be replaced by `tagged_na`, else by usual `NA` values. Use a named vector to assign the value label to the tagged NA value (see 'Examples').

Value

x, with all values in na being replaced by NA. If x is a data frame, the complete data frame x will be returned, with NA's set for variables specified in ...; if ... is not specified, applies to all variables in the data frame.

See Also

replace_na to replace NA's with specific values, rec for general recoding of variables and recode_to for re-shifting value ranges. See get_na to get values of missing values in labelled vectors.

Examples

```
dummy <- data.frame(var1 = sample(1:8, 100, replace = TRUE),
                    var2 = sample(1:10, 100, replace = TRUE),
                    var3 = sample(1:6, 100, replace = TRUE))

p <- function(x) max(x, na.rm = TRUE) > 7
tmp <- set_na_if(dummy, predicate = p, na = 8:9)
head(tmp)
```

---

shorten_string *Shorten character strings*

Description

This function shortens strings that are longer than max.length chars, without cropping words.

Usage

```
shorten_string(s, max.length = NULL, abbr = "...")
```

Arguments

s	A string.
max.length	Maximum length of chars for the string.
abbr	String that will be used as suffix, if s was shortened.

Details

If the string length defined in max.length happens to be inside a word, this word is removed from the returned string (see 'Examples'), so the returned string has a *maximum length* of max.length, but might be shorter.

Value

A shortened string.

Examples

```
s <- "This can be considered as very long string!"

# string is shorter than max.length, so returned as is
shorten_string(s, 60)

# string is shortened to as many words that result in
# a string of maximum 20 chars
shorten_string(s, 20)

# string including "considered" is exactly of length 22 chars
shorten_string(s, 22)
```

`split_var` *Split numeric variables into smaller groups*

Description

Recode numeric variables into equal sized groups, i.e. a variable is cut into a smaller number of groups at specific cut points. `split_var_if()` is a scoped variant of `split_var()`, where transformation will be applied only to those variables that match the logical condition of `predicate`.

Usage

```
split_var(x, ..., n, as.num = FALSE, val.labels = NULL,
  var.label = NULL, inclusive = FALSE, append = TRUE,
  suffix = "_g")

split_var_if(x, predicate, n, as.num = FALSE, val.labels = NULL,
  var.label = NULL, inclusive = FALSE, append = TRUE,
  suffix = "_g")
```

Arguments

`x`	A vector or data frame.
`...`	Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
`n`	The new number of groups that `x` should be split into.
`as.num`	Logical, if `TRUE`, return value will be numeric, not a factor.
`val.labels`	Optional character vector, to set value label attributes of recoded variable (see vignette Labelled Data and the sjlabelled-Package). If `NULL` (default), no value labels will be set. Value labels can also be directly defined in the `rec`-syntax, see 'Details'.

var.label — Optional string, to set variable label attribute for the returned variable (see vignette Labelled Data and the sjlabelled-Package). If NULL (default), variable label attribute of x will be used (if present). If empty, variable label attributes will be removed.

inclusive — Logical; if TRUE, cut point value are included in the preceeding group. This may be necessary if cutting a vector into groups does not define proper ("equal sized") group sizes. See 'Note' and 'Examples'.

append — Logical, if TRUE (the default) and x is a data frame, x including the new variables as additional columns is returned; if FALSE, only the new variables are returned.

suffix — String value, will be appended to variable (column) names of x, if x is a data frame. If x is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call:

- recoded variables (rec()) will be suffixed with "_r"
- recoded variables (recode_to()) will be suffixed with "_r0"
- dichotomized variables (dicho()) will be suffixed with "_d"
- grouped variables (split_var()) will be suffixed with "_g"
- grouped variables (group_var()) will be suffixed with "_gr"
- standardized variables (std()) will be suffixed with "_z"
- centered variables (center()) will be suffixed with "_c"
- de-meaned variables (de_mean()) will be suffixed with "_dm"
- grouped-meaned variables (de_mean()) will be suffixed with "_gm"

If suffix = "" and append = TRUE, existing variables that have been recoded/transformed will be overwritten.

predicate — A predicate function to be applied to the columns. The variables for which predicate returns TRUE are selected.

Details

split_var() splits a variable into equal sized groups, where the amount of groups depends on the n-argument. Thus, this functions cuts a variable into groups at the specified quantiles.

By contrast, group_var recodes a variable into groups, where groups have the same value range (e.g., from 1-5, 6-10, 11-15 etc.).

split_var() also works on grouped data frames (see group_by). In this case, splitting is applied to the subsets of variables in x. See 'Examples'.

Value

A grouped variable with equal sized groups. If x is a data frame, for append = TRUE, x including the grouped variables as new columns is returned; if append = FALSE, only the grouped variables will be returned. If append = TRUE and suffix = "", recoded variables will replace (overwrite) existing variables.

Note

In case a vector has only few number of unique values, splitting into equal sized groups may fail. In this case, use the `inclusive`-argument to shift a value at the cut point into the lower, preceeding group to get equal sized groups. See 'Examples'.

See Also

`group_var` to group variables into equal ranged groups, or `rec` to recode variables.

Examples

```
data(efc)
# non-grouped
table(efc$neg_c_7)

# split into 3 groups
table(split_var(efc$neg_c_7, n = 3))

# split multiple variables into 3 groups
split_var(efc, neg_c_7, pos_v_4, e17age, n = 3, append = FALSE)
frq(split_var(efc, neg_c_7, pos_v_4, e17age, n = 3, append = FALSE))

# original
table(efc$e42dep)

# two groups, non-inclusive cut-point
# vector split leads to unequal group sizes
table(split_var(efc$e42dep, n = 2))

# two groups, inclusive cut-point
# group sizes are equal
table(split_var(efc$e42dep, n = 2, inclusive = TRUE))

# Unlike dplyr's ntile(), split_var() never splits a value
# into two different categories, i.e. you always get a clean
# separation of original categories
library(dplyr)

x <- dplyr::ntile(efc$neg_c_7, n = 3)
table(efc$neg_c_7, x)

x <- split_var(efc$neg_c_7, n = 3)
table(efc$neg_c_7, x)

# works also with gouped data frames
mtcars %>%
  split_var(disp, n = 3, append = FALSE) %>%
  table()

mtcars %>%
  group_by(cyl) %>%
  split_var(disp, n = 3, append = FALSE) %>%
```

```
table()
```

---

spread_coef *Spread model coefficients of list-variables into columns*

Description

This function extracts coefficients (and standard error and p-values) of fitted model objects from (nested) data frames, which are saved in a list-variable, and spreads the coefficients into new columnns.

Usage

```
spread_coef(data, model.column, model.term, se, p.val, append = TRUE)
```

Arguments

data	A (nested) data frame with a list-variable that contains fitted model objects (see 'Details').
model.column	Name or index of the list-variable that contains the fitted model objects.
model.term	Optional, name of a model term. If specified, only this model term (including p-value) will be extracted from each model and added as new column.
se	Logical, if TRUE, standard errors for estimates will also be extracted.
p.val	Logical, if TRUE, p-values for estimates will also be extracted.
append	Logical, if TRUE (default), this function returns data with new columns for the model coefficients; else, a new data frame with model coefficients only are returned.

Details

This function requires a (nested) data frame (e.g. created by the nest-function of the **tidyr**-package), where several fitted models are saved in a list-variable (see 'Examples'). Since nested data frames with fitted models stored as list-variable are typically fit with an identical formula, all models have the same dependent and independent variables and only differ in their subsets of data. The function then extracts all coefficients from each model and saves each estimate in a new column. The result is a data frame, where each *row* is a model with each model's coefficients in an own *column*.

Value

A data frame with columns for each coefficient of the models that are stored in the list-variable of data; or, if model.term is given, a data frame with the term's estimate. If se = TRUE or p.val = TRUE, the returned data frame also contains columns for the coefficients' standard error and p-value. If append = TRUE, the columns are appended to data, i.e. data is also returned.

Examples

```
library(dplyr)
library(tidyr)
library(purrr)
data(efc)

# create nested data frame, grouped by dependency (e42dep)
# and fit linear model for each group. These models are
# stored in the list variable "models".
model.data <- efc %>%
  filter(!is.na(e42dep)) %>%
  group_by(e42dep) %>%
  nest() %>%
  mutate(
    models = map(data, ~lm(neg_c_7 ~ c12hour + c172code, data = .x))
  )

# spread coefficients, so we can easily access and compare the
# coefficients over all models. arguments `se` and `p.val` default
# to `FALSE`, when `model.term` is not specified
spread_coef(model.data, models)
spread_coef(model.data, models, se = TRUE)

# select only specific model term. `se` and `p.val` default to `TRUE`
spread_coef(model.data, models, c12hour)

# spread_coef can be used directly within a pipe-chain
efc %>%
  filter(!is.na(e42dep)) %>%
  group_by(e42dep) %>%
  nest() %>%
  mutate(
    models = map(data, ~lm(neg_c_7 ~ c12hour + c172code, data = .x))
  ) %>%
  spread_coef(models)

# spread_coef() makes it easy to generate bootstrapped
# confidence intervals, using the 'bootstrap()' and 'boot_ci()'
# functions from the 'sjstats' package, which creates nested
# data frames of bootstrap replicates
library(sjstats)
efc %>%
  # generate bootstrap replicates
  bootstrap(100) %>%
  # apply lm to all bootstrapped data sets
  mutate(
    models = map(strap, ~lm(neg_c_7 ~ e42dep + c161sex + c172code, data = .x))
  ) %>%
  # spread model coefficient for all 100 models
  spread_coef(models, se = FALSE, p.val = FALSE) %>%
  # compute the CI for all bootstrapped model coefficients
  boot_ci(e42dep, c161sex, c172code)
```

`std` *Standardize and center variables*

Description

`std()` computes a z-transformation (standardized and centered) on the input. `center()` centers the input. `std_if()` and `center_if()` are scoped variants of `std()` and `center()`, where transformation will be applied only to those variables that match the logical condition of `predicate`.

Usage

```
std(x, ..., robust = c("sd", "2sd", "gmd", "mad"), include.fac = FALSE,
  append = TRUE, suffix = "_z")

std_if(x, predicate, robust = c("sd", "2sd", "gmd", "mad"),
  include.fac = FALSE, append = TRUE, suffix = "_z")

center(x, ..., include.fac = FALSE, append = TRUE, suffix = "_c")

center_if(x, predicate, include.fac = FALSE, append = TRUE,
  suffix = "_c")
```

Arguments

`x`	A vector or data frame.
`...`	Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
`robust`	Character vector, indicating the method applied when standardizing variables with `std()`. By default, standardization is achieved by dividing the centered variables by their standard deviation (`robust = "sd"`). However, for skewed distributions, the median absolute deviation (MAD, `robust = "mad"`) or Gini's mean difference (`robust = "gmd"`) might be more robust measures of dispersion. For the latter option, **sjstats** needs to be installed. `robust = "2sd"` divides the centered variables by two standard deviations, following a suggestion by *Gelman (2008)*, so the rescaled input is comparable to binary variables.
`include.fac`	Logical, if `TRUE`, factors will be converted to numeric vectors and also standardized or centered.
`append`	Logical, if `TRUE` (the default) and `x` is a data frame, `x` including the new variables as additional columns is returned; if `FALSE`, only the new variables are returned.
`suffix`	String value, will be appended to variable (column) names of `x`, if `x` is a data frame. If `x` is not a data frame, this argument will be ignored. The default value to suffix column names in a data frame depends on the function call:

- recoded variables (`rec()`) will be suffixed with `"_r"`
- recoded variables (`recode_to()`) will be suffixed with `"_r0"`
- dichotomized variables (`dicho()`) will be suffixed with `"_d"`
- grouped variables (`split_var()`) will be suffixed with `"_g"`
- grouped variables (`group_var()`) will be suffixed with `"_gr"`
- standardized variables (`std()`) will be suffixed with `"_z"`
- centered variables (`center()`) will be suffixed with `"_c"`
- de-meaned variables (`de_mean()`) will be suffixed with `"_dm"`
- grouped-meaned variables (`de_mean()`) will be suffixed with `"_gm"`

If `suffix = ""` and `append = TRUE`, existing variables that have been recoded/transformed will be overwritten.

`predicate` A predicate function to be applied to the columns. The variables for which `predicate` returns `TRUE` are selected.

Details

`std()` and `center()` also work on grouped data frames (see `group_by`). In this case, standardization or centering is applied to the subsets of variables in `x`. See 'Examples'.

For more complicated models with many predictors, Gelman and Hill (2007) suggest leaving binary inputs as is and only standardize continuous predictors by dividing by two standard deviations. This ensures a rough comparability in the coefficients.

Value

If `x` is a vector, returns a vector with standardized or centered variables. If `x` is a data frame, for `append = TRUE`, `x` including the transformed variables as new columns is returned; if `append = FALSE`, only the transformed variables will be returned. If `append = TRUE` and `suffix = ""`, recoded variables will replace (overwrite) existing variables.

Note

`std()` and `center()` only return a vector, if `x` is a vector. If `x` is a data frame and only one variable is specified in the `...`-ellipses argument, both functions do return a data frame (see 'Examples').

References

Gelman A (2008) Scaling regression inputs by dividing by two standard deviations. *Statistics in Medicine 27: 2865-2873.* http://www.stat.columbia.edu/~gelman/research/published/standardizing7.pdf

Gelman A, Hill J (2007) Data Analysis Using Regression and Multilevel/Hierarchical Models. Cambdridge, Cambdrige University Press: 55-57

Examples

```
data(efc)
std(efc$c160age) %>% head()
std(efc, e17age, c160age, append = FALSE) %>% head()

center(efc$c160age) %>% head()
center(efc, e17age, c160age, append = FALSE) %>% head()

# NOTE!
std(efc$e17age) # returns a vector
std(efc, e17age) # returns a data frame

# with quasi-quotation
x <- "e17age"
center(efc, !!x, append = FALSE) %>% head()

# works with mutate()
library(dplyr)
efc %>%
  select(e17age, neg_c_7) %>%
  mutate(age_std = std(e17age), burden = center(neg_c_7)) %>%
  head()

# works also with grouped data frames
mtcars %>% std(disp)

# compare new column "disp_z" w/ output above
mtcars %>%
  group_by(cyl) %>%
  std(disp)

data(iris)
# also standardize factors
std(iris, include.fac = TRUE, append = FALSE)
# don't standardize factors
std(iris, include.fac = FALSE, append = FALSE)

# standardize only variables with more than 10 unique values
p <- function(x) dplyr::n_distinct(x) > 10
std_if(efc, predicate = p, append = FALSE)
```

---

str_contains *Check if string contains pattern*

Description

This functions checks whether a string or character vector `x` contains the string `pattern`. By default, this function is case sensitive.

Usage

```
str_contains(x, pattern, ignore.case = FALSE, logic = NULL,
  switch = FALSE)
```

Arguments

x	Character string where matches are sought. May also be a character vector of length > 1 (see 'Examples').
pattern	Character string to be matched in x. May also be a character vector of length > 1 (see 'Examples').
ignore.case	Logical, whether matching should be case sensitive or not.
logic	Indicates whether a logical combination of multiple search pattern should be made. • Use "or", "OR" or "|" for a logical or-combination, i.e. at least one element of pattern is in x. • Use "and", "AND" or "&" for a logical AND-combination, i.e. all elements of pattern are in x. • Use "not", "NOT" or "!" for a logical NOT-combination, i.e. no element of pattern is in x. • By default, logic = NULL, which means that TRUE or FALSE is returned for each element of pattern separately.
switch	Logical, if TRUE, x will be sought in each element of pattern. If switch = TRUE, x needs to be of length 1.

Details

This function iterates all elements in pattern and looks for each of these elements if it is found in *any* element of x, i.e. which elements of pattern are found in the vector x.

Technically, it iterates pattern and calls grep(x,pattern[i],fixed = TRUE) for each element of pattern. If switch = TRUE, it iterates pattern and calls grep(pattern[i],x,fixed = TRUE) for each element of pattern. Hence, in the latter case (if switch = TRUE), x must be of length 1.

Value

TRUE if x contains pattern.

Examples

```
str_contains("hello", "hel")
str_contains("hello", "hal")

str_contains("hello", "Hel")
str_contains("hello", "Hel", ignore.case = TRUE)

# which patterns are in "abc"?
str_contains("abc", c("a", "b", "e"))
```

```
# is pattern in any element of 'x'?
str_contains(c("def", "abc", "xyz"), "abc")
# is "abcde" in any element of 'x'?
str_contains(c("def", "abc", "xyz"), "abcde") # no...
# is "abc" in any of pattern?
str_contains("abc", c("defg", "abcde", "xyz12"), switch = TRUE)

str_contains(c("def", "abcde", "xyz"), c("abc", "123"))

# any pattern in "abc"?
str_contains("abc", c("a", "b", "e"), logic = "or")

# all patterns in "abc"?
str_contains("abc", c("a", "b", "e"), logic = "and")
str_contains("abc", c("a", "b"), logic = "and")

# no patterns in "abc"?
str_contains("abc", c("a", "b", "e"), logic = "not")
str_contains("abc", c("d", "e", "f"), logic = "not")
```

`str_find` *Find partial matching and close distance elements in strings*

Description

This function finds the element indices of partial matching or similar strings in a character vector. Can be used to find exact or slightly mistyped elements in a string vector.

Usage

```
str_find(string, pattern, precision = 2, partial = 0,
  verbose = FALSE)
```

Arguments

`string`	Character vector with string elements.
`pattern`	String that should be matched against the elements of `string`.
`precision`	Maximum distance ("precision") between two string elements, which is allowed to treat them as similar or equal. Smaller values mean less tolerance in matching.
`partial`	Activates similar matching (close distance strings) for parts (substrings) of the `string`. Following values are accepted: • 0 for no partial distance matching • 1 for one-step matching, which means, only substrings of same length as `pattern` are extracted from `string` matching • 2 for two-step matching, which means, substrings of same length as `pattern` as well as strings with a slightly wider range are extracted from `string` matching

Default value is 0. See 'Details' for more information.

verbose — Logical; if TRUE, the progress bar is displayed when computing the distance matrix. Default in FALSE, hence the bar is hidden.

Details

Computation Details

Fuzzy string matching is based on regular expressions, in particular `grep(pattern = "(<pattern>){~<precision>}",x = string)`. This means, `precision` indicates the number of chars inside `pattern` that may differ in `string` to cosinder it as "matching". The higher `precision` is, the more tolerant is the search (i.e. yielding more possible matches). Furthermore, the higher the value for `partial` is, the more matches may be found.

Partial Distance Matching

For `partial = 1`, a substring of `length(pattern)` is extracted from `string`, starting at position 0 in `string` until the end of `string` is reached. Each substring is matched against `pattern`, and results with a maximum distance of `precision` are considered as "matching". If `partial = 2`, the range of the extracted substring is increased by 2, i.e. the extracted substring is two chars longer and so on.

Value

A numeric vector with index position of elements in `string` that partially match or are similar to `pattern`. Returns `-1` if no match was found.

Note

This function does *not* return the position of a matching string *inside* another string, but the element's index of the `string` vector, where a (partial) match with `pattern` was found. Thus, searching for "abc" in a string "this is abc" will not return 9 (the start position of the substring), but 1 (the element index, which is always 1 if `string` only has one element).

See Also

group_str

Examples

```
string <- c("Hello", "Helo", "Hole", "Apple", "Ape", "New", "Old", "System", "Systemic")
str_find(string, "hel")   # partial match
str_find(string, "stem")  # partial match
str_find(string, "R")     # no match
str_find(string, "saste") # similarity to "System"

# finds two indices, because partial matching now
# also applies to "Systemic"
str_find(string,
        "sytsme",
```

```
        partial = 1)

# finds partial matching of similarity
str_find("We are Sex Pistols!", "postils")
```

`str_start` *Find start and end index of pattern in string*

Description

`str_start()` finds the beginning position of `pattern` in each element of `x`, while `str_end()` finds the stopping position of `pattern` in each element of `x`.

Usage

```
str_start(x, pattern, ignore.case = TRUE, regex = FALSE)

str_end(x, pattern, ignore.case = TRUE, regex = FALSE)
```

Arguments

`x`	A character vector.
`pattern`	Character string to be matched in `x`. `pattern` might also be a regular-expression object, as returned by `stringr::regex()`. Alternatively, use `regex = TRUE` to treat `pattern` as a regular expression rather than a fixed string.
`ignore.case`	Logical, whether matching should be case sensitive or not. `ignore.case` is ignored when `pattern` is no regular expression or `regex = FALSE`.
`regex`	Logical, if `TRUE`, `pattern` is treated as a regular expression rather than a fixed string.

Value

A numeric vector with index of start/end position(s) of `pattern` found in `x`, or `-1`, if `pattern` was not found in `x`.

Examples

```
path <- "this/is/my/fileofinterest.csv"
str_start(path, "/")

path <- "this//is//my//fileofinterest.csv"
str_start(path, "//")
str_end(path, "//")

x <- c("my_friend_likes me", "your_friend likes_you")
str_start(x, "_")
```

```
# pattern "likes" starts at position 11 in first, and
# position 13 in second string
str_start(x, "likes")

# pattern "likes" ends at position 15 in first, and
# position 17 in second string
str_end(x, "likes")

x <- c("I like to move it, move it", "You like to move it")
str_start(x, "move")
str_end(x, "move")

x <- c("test1234testagain")
str_start(x, "\\d+4")
str_start(x, "\\d+4", regex = TRUE)
str_end(x, "\\d+4", regex = TRUE)
```

`tidy_values` *Clean values of character vectors.*

Description

This function "cleans" values of a character vector or levels of a factor by removing space and punctuation characters.

Usage

```
tidy_values(x, ...)

clean_values(x, ...)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.

Value

x, with "cleaned" values or levels.

Examples

```
f1 <- sprintf("Char %s", sample(LETTERS[1:5], size = 10, replace = TRUE))
f2 <- as.factor(sprintf("F / %s", sample(letters[1:5], size = 10, replace = TRUE)))
f3 <- sample(1:5, size = 10, replace = TRUE)

x <- data.frame(f1, f2, f3, stringsAsFactors = FALSE)

clean_values(f1)
clean_values(f2)
clean_values(x)
```

---

to_character *Convert variable into character vector and replace values with associated value labels*

Description

This function converts (replaces) variable values (also of factors or character vectors) with their associated value labels and returns them as character vector. This is just a convenient wrapper for `as.character(to_label(x))`.

Usage

```
to_character(x, ..., add.non.labelled = FALSE, prefix = FALSE,
  var.label = NULL, drop.na = TRUE, drop.levels = FALSE)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
add.non.labelled	Logical, if TRUE, values without associated value label will also be converted to labels (as is). See 'Examples'.
prefix	Logical, if TRUE, the value labels used as factor levels or character values will be prefixed with their associated values. See 'Examples'.
var.label	Optional string, to set variable label attribute for the returned variable (see vignette Labelled Data and the sjlabelled-Package). If NULL (default), variable label attribute of x will be used (if present). If empty, variable label attributes will be removed.
drop.na	Logical, if TRUE, tagged NA values with value labels will be converted to regular NA's. Else, tagged NA values will be replaced with their value labels. See 'Examples' and get_na.
drop.levels	Logical, if TRUE, unused factor levels will be dropped (i.e. droplevels will be applied before returning the result).

Value

A character vector with the associated value labels as values. If x is a data frame, the complete data frame x will be returned, where variables specified in ... are coerced to character variables; if ... is not specified, applies to all variables in the data frame.

Note

Value labels will be removed when converting variables to factors, variable labels, however, are preserved.

This function is kept for backwards-compatibility. It is preferred to use `sjlabelled::as_character()`.

Examples

```
library(sjlabelled)
data(efc)
print(get_labels(efc)['c161sex'])
head(efc$c161sex)
head(to_character(efc$c161sex))

# Find more examples at '?sjlabelled::as_label'
```

---

to_dummy *Split (categorical) vectors into dummy variables*

Description

This function splits categorical or numeric vectors with more than two categories into 0/1-coded dummy variables.

Usage

```
to_dummy(x, ..., var.name = "name", suffix = c("numeric", "label"))
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's `select_helpers`. See 'Examples' or package-vignette.
var.name	Indicates how the new dummy variables are named. Use "name" to use the variable name or any other string that will be used as is. Only applies, if x is a vector. See 'Examples'.
suffix	Indicates which suffix will be added to each dummy variable. Use "numeric" to number dummy variables, e.g. *x_1*, *x_2*, *x_3* etc. Use "label" to add value label, e.g. *x_low*, *x_mid*, *x_high*. May be abbreviated.

Value

A data frame with dummy variables for each category of x. The dummy coded variables are of type atomic.

Note

NA values will be copied from x, so each dummy variable has the same amount of NA's at the same position as x.

Examples

```
data(efc)
head(to_dummy(efc$e42dep))

# add value label as suffix to new variable name
head(to_dummy(efc$e42dep, suffix = "label"))

# use "dummy" as new variable name
head(to_dummy(efc$e42dep, var.name = "dummy"))

# create multiple dummies, append to data frame
to_dummy(efc, c172code, e42dep)

# pipe-workflow
library(dplyr)
efc %>%
  select(e42dep, e16sex, c172code) %>%
  to_dummy()
```

---

to_factor *Convert variable into factor and keep value labels*

Description

This function converts a variable into a factor, but preserves variable and value label attributes. See 'Examples'.

Usage

```
to_factor(x, ..., add.non.labelled = FALSE, ref.lvl = NULL)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.

add.non.labelled
: Logical, if TRUE, non-labelled values also get value labels.

ref.lvl
: Numeric, specifies the reference level for the new factor. Use this parameter if a different factor level than the lowest value should be used as reference level. If NULL, lowest value will become the reference level. See ref_lvl for details.

Details

to_factor converts numeric values into a factor with numeric levels. as_label, however, converts a vector into a factor and uses value labels as factor levels.

Value

A factor, including variable and value labels. If x is a data frame, the complete data frame x will be returned, where variables specified in ... are coerced to factors (including variable and value labels); if ... is not specified, applies to all variables in the data frame.

Note

This function is intended for use with vectors that have value and variable label attributes. Unlike as.factor, to_factor converts a variable into a factor and preserves the value and variable label attributes.

Adding label attributes is automatically done by importing data sets with one of the read_*-functions, like read_spss. Else, value and variable labels can be manually added to vectors with set_labels and set_label.

This function is kept for backwards-compatibility. It is preferred to use as_factor.

See Also

as_numeric to convert a factor into a numeric vector and as_label to convert a vector into a factor with labelled factor levels.

Examples

```
library(sjlabelled)
data(efc)
# normal factor conversion, loses value attributes
x <- as.factor(efc$e42dep)
frq(x)

# factor conversion, which keeps value attributes
x <- to_factor(efc$e42dep)
frq(x)

# create parially labelled vector
x <- set_labels(efc$e42dep,
                labels = c(`1` = "independent", `4` = "severe dependency",
                           `9` = "missing value"))
```

```
# only copy existing value labels
to_factor(x)
get_labels(to_factor(x), values = "p")

# also add labels to non-labelled values
to_factor(x, add.non.labelled = TRUE)
get_labels(to_factor(x, add.non.labelled = TRUE), values = "p")


# Convert to factor, using different reference level
x <- to_factor(efc$e42dep)
str(x)
table(x)

x <- to_factor(efc$e42dep, ref.lvl = 3)
str(x)
table(x)


# easily coerce specific variables in a data frame to factor
# and keep other variables, with their class preserved
to_factor(efc, e42dep, e16sex, c172code)

# use select-helpers from dplyr-package
library(dplyr)
to_factor(efc, contains("cop"), c161sex:c175empl)
```

`to_label` *Convert variable into factor with associated value labels*

Description

This function converts (replaces) values of a variable (also of factors or character vectors) with their associated value labels. Might be helpful for factor variables. For instance, if you have a Gender variable with 0/1 value, and associated labels are male/female, this function would convert all 0 to male and all 1 to female and returns the new variable as factor.

Usage

```
to_label(x, ..., add.non.labelled = FALSE, prefix = FALSE,
  var.label = NULL, drop.na = TRUE, drop.levels = FALSE)
```

Arguments

x	A vector or data frame.

`...` Optional, unquoted names of variables that should be selected for further processing. Required, if `x` is a data frame (and no vector) and only selected variables from `x` should be processed. You may also use functions like `:` or tidyselect's `select_helpers`. See 'Examples' or package-vignette.

`add.non.labelled` Logical, if `TRUE`, values without associated value label will also be converted to labels (as is). See 'Examples'.

`prefix` Logical, if `TRUE`, the value labels used as factor levels or character values will be prefixed with their associated values. See 'Examples'.

`var.label` Optional string, to set variable label attribute for the returned variable (see vignette Labelled Data and the sjlabelled-Package). If `NULL` (default), variable label attribute of `x` will be used (if present). If empty, variable label attributes will be removed.

`drop.na` Logical, if `TRUE`, tagged `NA` values with value labels will be converted to regular NA's. Else, tagged `NA` values will be replaced with their value labels. See 'Examples' and `get_na`.

`drop.levels` Logical, if `TRUE`, unused factor levels will be dropped (i.e. `droplevels` will be applied before returning the result).

Value

A factor with the associated value labels as factor levels. If `x` is a data frame, the complete data frame `x` will be returned, where variables specified in `...` are coerced to factors; if `...` is not specified, applies to all variables in the data frame.

Note

Value label attributes will be removed when converting variables to factors.

This function is kept for backwards-compatibility. It is preferred to use `as_label`.

Examples

```
library(sjlabelled)
data(efc)
print(get_labels(efc)['c161sex'])
head(efc$c161sex)
head(to_label(efc$c161sex))

# Find more examples at '?sjlabelled::as_label'
```

to_long *Convert wide data to long format*

Description

This function converts wide data into long format. It allows to transform multiple key-value pairs to be transformed from wide to long format in one single step.

Usage

```
to_long(data, keys, values, ..., labels = NULL, recode.key = FALSE)
```

Arguments

data	A data.frame that should be tansformed from wide to long format.
keys	Character vector with name(s) of key column(s) to create in output. Either one key value per column group that should be gathered, or a single string. In the latter case, this name will be used as key column, and only one key column is created. See 'Examples'.
values	Character vector with names of value columns (variable names) to create in output. Must be of same length as number of column groups that should be gathered. See 'Examples'.
...	Specification of columns that should be gathered. Must be one character vector with variable names per column group, or a numeric vector with column indices indicating those columns that should be gathered. See 'Examples'.
labels	Character vector of same length as values with variable labels for the new variables created from gathered columns. See 'Examples' and 'Details'.
recode.key	Logical, if TRUE, the values of the key column will be recoded to numeric values, in sequential ascending order.

Details

This function reshapes data from wide to long format, however, you can gather multiple column groups at once. Value and variable labels for non-gathered variables are preserved. Attributes from gathered variables, such as information about the variable labels, are lost during reshaping. Hence, the new created variables from gathered columns don't have any variable label attributes. In such cases, use labels argument to set back variable label attributes.

See Also

reshape_longer

Examples

```
# create sample
mydat <- data.frame(age = c(20, 30, 40),
                    sex = c("Female", "Male", "Male"),
                    score_t1 = c(30, 35, 32),
                    score_t2 = c(33, 34, 37),
                    score_t3 = c(36, 35, 38),
                    speed_t1 = c(2, 3, 1),
                    speed_t2 = c(3, 4, 5),
                    speed_t3 = c(1, 8, 6))

# gather multiple columns. both time and speed are gathered.
to_long(
  data = mydat,
  keys = "time",
  values = c("score", "speed"),
  c("score_t1", "score_t2", "score_t3"),
  c("speed_t1", "speed_t2", "speed_t3")
)

# alternative syntax, using "reshape_longer()"
reshape_longer(
  mydat,
  columns = list(
    c("score_t1", "score_t2", "score_t3"),
    c("speed_t1", "speed_t2", "speed_t3")
  ),
  names.to = "time",
  values.to = c("score", "speed")
)

# or ...
reshape_longer(
  mydat,
  list(3:5, 6:8),
  names.to = "time",
  values.to = c("score", "speed")
)

# gather multiple columns, use numeric key-value
to_long(
  data = mydat,
  keys = "time",
  values = c("score", "speed"),
  c("score_t1", "score_t2", "score_t3"),
  c("speed_t1", "speed_t2", "speed_t3"),
  recode.key = TRUE
)

# gather multiple columns by colum names and colum indices
to_long(
  data = mydat,
```

```
  keys = "time",
  values = c("score", "speed"),
  c("score_t1", "score_t2", "score_t3"),
  6:8,
  recode.key = TRUE
)

# gather multiple columns, use separate key-columns
# for each value-vector
to_long(
  data = mydat,
  keys = c("time_score", "time_speed"),
  values = c("score", "speed"),
  c("score_t1", "score_t2", "score_t3"),
  c("speed_t1", "speed_t2", "speed_t3")
)

# gather multiple columns, label columns
mydat <- to_long(
  data = mydat,
  keys = "time",
  values = c("score", "speed"),
  c("score_t1", "score_t2", "score_t3"),
  c("speed_t1", "speed_t2", "speed_t3"),
  labels = c("Test Score", "Time needed to finish")
)

library(sjlabelled)
str(mydat$score)
get_label(mydat$speed)
```

`to_value` *Convert factors to numeric variables*

Description

This function converts (replaces) factor levels with the related factor level index number, thus the factor is converted to a numeric variable.

Usage

```
to_value(x, ..., start.at = NULL, keep.labels = TRUE,
  use.labels = FALSE)
```

Arguments

x	A vector or data frame.

...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
start.at	Starting index, i.e. the lowest numeric value of the variable's value range. By default, this argument is NULL, hence the lowest value of the returned numeric variable corresponds to the lowest factor level (if factor levels are numeric) or to 1 (if factor levels are not numeric).
keep.labels	Logical, if TRUE, former factor levels will be added as value labels. For numeric factor levels, values labels will be used, if present. See 'Examples' and set_labels for more details.
use.labels	Logical, if TRUE and x has numeric value labels, these value labels will be set as numeric values.

Value

A numeric variable with values ranging either from start.at to start.at + length of factor levels, or to the corresponding factor levels (if these were numeric). If x is a data frame, the complete data frame x will be returned, where variables specified in ... are coerced to numeric; if ... is not specified, applies to all variables in the data frame.

Note

This function is kept for backwards-compatibility. It is preferred to use as_numeric.

Examples

```
library(sjlabelled)
data(efc)
test <- as_label(efc$e42dep)
table(test)
table(to_value(test))

# Find more examples at '?sjlabelled::as_numeric'
```

---

trim *Trim leading and trailing whitespaces from strings*

Description

Trims leading and trailing whitespaces from strings or character vectors.

Usage

```
trim(x)
```

Arguments

x	Character vector or string, or a list or data frame with such vectors. Function is vectorized, i.e. vector may have a length greater than 1. See 'Examples'.

Value

Trimmed x, i.e. with leading and trailing spaces removed.

Examples

```
trim("white space at end ")
trim(" white space at start and end ")
trim(c(" string1 ", "   string2", "string 3   "))

tmp <- data.frame(a = c(" string1 ", "   string2", "string 3   "),
                  b = c(" strong one  ", "    string two", "  third string "),
                  c = c(" str1 ", "   str2", "str3   "))
tmp
trim(tmp)
```

---

typical_value *Return the typical value of a vector*

Description

This function returns the "typical" value of a variable.

Usage

```
typical_value(x, fun = "mean", weights = NULL, ...)
```

Arguments

x	A variable.
fun	Character vector, naming the function to be applied to x. Currently, "mean", "weighted.mean", "median" and "mode" are supported, which call the corresponding R functions (except "mode", which calls an internal function to compute the most common value). "zero" simply returns 0. **Note:** By default, if x is a factor, only fun = "mode" is applicable; for all other functions (including the default, "mean") the reference level of x is returned. For character vectors, only the mode is returned. You can use a named vector to apply other different functions to numeric and categorical x, where factors are first converted to numeric vectors, e.g. fun = c(numeric = "median",factor = "mean"). See 'Examples'.
weights	Name of variable in x that indicated the vector of weights that will be applied to weight all observations. Default is NULL, so no weights are used.
...	Further arguments, passed down to fun.

Details

By default, for numeric variables, `typical_value()` returns the mean value of `x` (unless changed with the `fun`-argument).

For factors, the reference level is returned or the most common value (if `fun = "mode"`), unless `fun` is a named vector. If `fun` is a named vector, specify the function for numeric and categorical variables as element names, e.g. `fun = c(numeric = "median", factor = "mean")`. In this case, factors are converted to numeric values (using `to_value`) and the related function is applied. You may abbreviate the names `fun = c(n = "median", f = "mean")`. See also 'Examples'.

For character vectors the most common value (mode) is returned.

Value

The "typical" value of `x`.

Examples

```
data(iris)
typical_value(iris$Sepal.Length)

library(purrr)
map(iris, ~ typical_value(.x))

# example from ?stats::weighted.mean
wt <- c(5,  5,  4,  1) / 15
x <- c(3.7, 3.3, 3.5, 2.8)

typical_value(x, fun = "weighted.mean")
typical_value(x, fun = "weighted.mean", weights = wt)

# for factors, return either reference level or mode value
set.seed(123)
x <- sample(iris$Species, size = 30, replace = TRUE)
typical_value(x)
typical_value(x, fun = "mode")

# for factors, use a named vector to apply other functions than "mode"
map(iris, ~ typical_value(.x, fun = c(n = "median", f = "mean")))
```

---

var_rename *Rename variables*

Description

This function renames variables in a data frame, i.e. it renames the columns of the data frame.

Usage

```
var_rename(x, ..., verbose = TRUE)

rename_variables(x, ..., verbose = TRUE)

rename_columns(x, ..., verbose = TRUE)
```

Arguments

x	A data frame.
...	A named vector, or pairs of named vectors, where the name (lhs) equals the column name that should be renamed, and the value (rhs) is the new column name.
verbose	Logical, if TRUE, a warning is displayed when variable names do not exist in x.

Value

x, with new column names for those variables specified in ....

Examples

```
dummy <- data.frame(
  a = sample(1:4, 10, replace = TRUE),
  b = sample(1:4, 10, replace = TRUE),
  c = sample(1:4, 10, replace = TRUE)
)

rename_variables(dummy, a = "first.col", c = "3rd.col")

# using quasi-quotation
library(rlang)
v1 <- "first.col"
v2 <- "3rd.col"
rename_variables(dummy, a = !!v1, c = !!v2)

x1 <- "a"
x2 <- "b"
rename_variables(dummy, !!x1 := !!v1, !!x2 := !!v2)

# using a named vector
new_names <- c(a = "first.col", c = "3rd.col")
rename_variables(dummy, new_names)
```

var_type *Determine variable type*

Description

This function returns the type of a variable as character. It is similar to type_sum, however, the return value is not truncated, and var_type() works on data frames and within pipe-chains.

Usage

```
var_type(x, ..., abbr = FALSE)
```

Arguments

x	A vector or data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.
abbr	Logical, if TRUE, returns a shortened, abbreviated value for the variable type (as returned by type_sum). If FALSE (default), a longer "description" is returned.

Value

The variable type of x, as character.

Examples

```
data(efc)

var_type(1)
var_type(1L)
var_type("a")

var_type(efc$e42dep)
var_type(to_factor(efc$e42dep))

library(dplyr)
var_type(efc, contains("cop"))
```

word_wrap *Insert line breaks in long labels*

Description

Insert line breaks in long character strings. Useful if you want to wordwrap labels / titles for plots or tables.

Usage

```
word_wrap(labels, wrap, linesep = NULL)
```

Arguments

labels	Label(s) as character string, where a line break should be inserted. Several strings may be passed as vector (see 'Examples').
wrap	Maximum amount of chars per line (i.e. line length). If wrap = Inf or wrap = 0, no word wrap will be performed (i.e. labels will be returned as is).
linesep	By default, this argument is NULL and a regular new line string ("\n") is used. For HTML-purposes, for instance, linesep could be " ".

Value

New label(s) with line breaks inserted at every wrap's position.

Examples

```
word_wrap(c("A very long string", "And another even longer string!"), 10)

message(word_wrap("Much too long string for just one line!", 15))
```

zap_inf *Convert infiite or NaN values into regular NA*

Description

Replaces all infinite (Inf and -Inf) or NaN values with regular NA.

Usage

```
zap_inf(x, ...)
```

Arguments

x	A vector or a data frame.
...	Optional, unquoted names of variables that should be selected for further processing. Required, if x is a data frame (and no vector) and only selected variables from x should be processed. You may also use functions like : or tidyselect's select_helpers. See 'Examples' or package-vignette.

Value

x, where all Inf, -Inf and NaN are converted to NA.

Examples

```
x <- c(1, 2, NA, 3, NaN, 4, NA, 5, Inf, -Inf, 6, 7)
zap_inf(x)

data(efc)
# produce some NA and NaN values
efc$e42dep[1] <- NaN
efc$e42dep[2] <- NA
efc$c12hour[1] <- NaN
efc$c12hour[2] <- NA
efc$e17age[2] <- NaN
efc$e17age[1] <- NA

# only zap NaN for c12hour
zap_inf(efc$c12hour)

# only zap NaN for c12hour and e17age, not for e42dep,
# but return complete data framee
zap_inf(efc, c12hour, e17age)

# zap NaN for complete data frame
zap_inf(efc)
```

%nin% *Value matching*

Description

%nin% is the complement to %in%. It looks which values in x do *not* match (hence, are *not in*) values in y.

Usage

```
x %nin% y
```

Arguments

x	Vector with values to be matched.
y	Vector with values to be matched against.

Details

See 'Details' in match.

Value

A logical vector, indicating if a match was *not* located for each element of x, thus the values are TRUE or FALSE and never NA.

Examples

```
c("a", "B", "c") %in% letters
c("a", "B", "c") %nin% letters

c(1, 2, 3, 4) %in% c(3, 4, 5, 6)
c(1, 2, 3, 4) %nin% c(3, 4, 5, 6)
```

Index