Package 'expss'

November 15, 2020

Type Package

Title Tables, Labels and Some Useful Functions from Spreadsheets and 'SPSS' Statistics

Version 0.10.7

Maintainer Gregory Demin <gdemin@gmail.com>

URL https://gdemin.github.io/expss/

BugReports https://github.com/gdemin/expss/issues

Depends R (>= 3.3.0),

Imports foreign, utils, stats, magrittr (>= 1.5), htmlTable (>= 1.11.0), matrixStats (>= 0.51.0), data.table (>= 1.10),

Suggests DT, htmltools, knitr, rmarkdown, repr, ggplot2, testthat, openxlsx, fst, huxtable

Description Package computes and displays tables with support for 'SPSS'-style labels, multiple and nested banners, weights, multiple-response variables and significance testing. There are facilities for nice output of tables in 'knitr', 'Shiny', '*.xlsx' files, R and 'Jupyter' notebooks. Methods for labelled variables add value labels support to base R functions and to some functions from other packages. Additionally, the package brings popular data transformation functions from 'SPSS' Statistics and 'Excel': 'RECODE', 'COUNT', 'COMPUTE', 'DO IF', 'COUNTIF', 'VLOOKUP' and etc. These functions are very useful for data processing in marketing research surveys. Package intended to help people to move data processing from 'Excel' and 'SPSS' to R.

VignetteBuilder knitr

LazyData yes

License GPL (>= 2)

RoxygenNote 7.1.1

NeedsCompilation no

Author Gregory Demin [aut, cre], Sebastian Jeworutzki [ctb] (<https://orcid.org/0000-0002-2671-5253>)

Repository CRAN

Date/Publication 2020-11-15 22:00:06 UTC

R topics documented:

`add_rows` *Add rows to data.frame/matrix/table*

Description

Take a sequence of vector, matrix or data-frame arguments and combine by rows. Contrary to rbind it handles non-matching column names. `%add_rows%` is an infix version of `add_rows`. There is also special method for the results of cro/cro_fun/tables/fre. `.add_rows` is version for adding rows to default dataset. See default_dataset.

Usage

```
add_rows(...)

## S3 method for class 'data.frame'
add_rows(..., nomatch_columns = c("add", "drop", "stop"))

x %add_rows% y

.add_rows(..., nomatch_columns = c("add", "drop", "stop"))
```

Arguments

`...`	data.frame/matrix/table for binding
`nomatch_columns`	action if there are non-matching columns between data.frames. Possible values are `"add"`, `"drop"`, `"stop"`. `"add"` will combine all columns, `"drop"` will leave only common columns, `"stop"` will raise an error.
`x`	data.frame/matrix/table for binding
`y`	data.frame/matrix/table for binding

Value

See rbind, cro, cro_fun, fre, tables

Examples

```
a = data.frame(x = 1:5, y = 6:10)
b = data.frame(y = 6:10, z = 11:15)

add_rows(a, b) # x, y, z
a %add_rows% b # the same result

add_rows(a, b, nomatch_columns = "drop")  # y

# simple tables
data(mtcars)
# apply labels
mtcars = apply_labels(mtcars,
                mpg = "Miles/(US) gallon",
                cyl = "Number of cylinders",
                disp = "Displacement (cu.in.)",
                hp = "Gross horsepower",
                drat = "Rear axle ratio",
                wt = "Weight (lb/1000)",
                qsec = "1/4 mile time",
                vs = "V/S",
                vs = c("V-engine" = 0, "Straight engine" = 1),
                am = "Transmission (0 = automatic, 1 = manual)",
                am = c(automatic = 0, manual = 1),
                gear = "Number of forward gears",
                carb = "Number of carburetors"
)

tbl_mean = calculate(mtcars, cro_mean(cyl, am))
tbl_percent = calculate(mtcars, cro_cpct(cyl, am))

tbl_mean %add_rows% tbl_percent
```

---

apply_labels *Set variable labels/value labels on variables in the data.frame*

Description

apply_labels tries automatically detect what is variable label and what are value labels. .apply_labels is version for working with default dataset. See also var_lab and val_lab.

Usage

```
apply_labels(data, ...)

.apply_labels(...)
```

Arguments

data	data.frame/list
...	named arguments. Name of argument is a variable name in data. Argument values are variable or value labels. Unnamed characters of length 1 are considered as variable labels and named vectors are considered as value labels.

Value

data with applied labels

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      vs = "Engine",
                      vs = num_lab("
                             0 V-engine
                             1 Straight engine
                             "),
                      am = "Transmission",
                      am = num_lab("
                             0 Automatic
                             1 Manual
                             ")
)

# 'table' from base R
table(mtcars$vs, mtcars$am)

# more sofisticated crosstable
calculate(mtcars, cro(vs, am))
```

---

as.category *Convert dichotomy data.frame/matrix to data.frame with category encoding*

Description

Convert dichotomy data.frame/matrix to data.frame with category encoding

Usage

```
as.category(x, prefix = NULL, counted_value = 1, compress = FALSE)

is.category(x)
```

Arguments

x	Dichotomy data.frame/matrix (usually with 0,1 coding).
prefix	If is not NULL then column names will be added in the form prefix+column number.
counted_value	Vector. Values that will be considered as indicator of category presence. By default it equals to 1.
compress	Logical. Should we drop columns with all NA? FALSE by default. TRUE significantly decreases performance of the function.

Value

data.frame of class `category` with numeric values that correspond to column numbers of counted values. Column names of x or variable labels are added as value labels.

See Also

`as.dichotomy` for reverse conversion, mrset, mdset for usage multiple-response variables with tables.

Examples

```
set.seed(123)

# Let's imagine it's matrix of consumed products
dichotomy_matrix = matrix(sample(0:1,40,replace = TRUE,prob=c(.6,.4)),nrow=10)
colnames(dichotomy_matrix) = c("Milk","Sugar","Tea","Coffee")

as.category(dichotomy_matrix, compress = TRUE) # compressed version
category_encoding = as.category(dichotomy_matrix)

 # should be TRUE
identical(val_lab(category_encoding), c(Milk = 1L, Sugar = 2L, Tea = 3L, Coffee = 4L))
all(as.dichotomy(category_encoding, use_na = FALSE) == dichotomy_matrix)

# with prefix
as.category(dichotomy_matrix, prefix = "products_")

# data.frame with variable labels
dichotomy_dataframe = as.data.frame(dichotomy_matrix)
colnames(dichotomy_dataframe) = paste0("product_", 1:4)
var_lab(dichotomy_dataframe[[1]]) = "Milk"
var_lab(dichotomy_dataframe[[2]]) = "Sugar"
var_lab(dichotomy_dataframe[[3]]) = "Tea"
var_lab(dichotomy_dataframe[[4]]) = "Coffee"

as.category(dichotomy_dataframe, prefix = "products_")
```

`as.datatable_widget` *Create an HTML table widget for usage with Shiny*

Description

This is method for rendering results of tables/fre/cro in Shiny. `DT` package should be installed for this feature (`install.packages('DT')`). For detailed description of function and its arguments see datatable.

Usage

```
as.datatable_widget(data, ...)

## S3 method for class 'etable'
as.datatable_widget(
  data,
  ...,
  repeat_row_labels = FALSE,
  show_row_numbers = FALSE,
  digits = get_expss_digits()
)

## S3 method for class 'with_caption'
as.datatable_widget(
  data,
  ...,
  repeat_row_labels = FALSE,
  show_row_numbers = FALSE,
  digits = get_expss_digits()
)
```

Arguments

`data`	a data object (result of tables/fre/cro).
`...`	further parameters for datatable
`repeat_row_labels`	logical Should we repeat duplicated row labels in the every row? Default is FALSE.
`show_row_numbers`	logical Default is FALSE.
`digits`	integer By default, all numeric columns are rounded to one digit after decimal separator. Also you can set this argument by option 'expss.digits' - for example, `expss_digits(2)`. If it is NA than all numeric columns remain unrounded.

Value

Object of class datatable

See Also

htmlTable for knitting

Examples

```
## Not run:

data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

mtcars_table = mtcars %>%
                 tab_cols(total(), am %nest% vs) %>%
                 tab_cells(mpg, hp) %>%
                 tab_stat_mean() %>%
                 tab_cells(cyl) %>%
                 tab_stat_cpct() %>%
                 tab_pivot() %>%
                 set_caption("Table 1. Some variables from mtcars dataset.")

library(shiny)
shinyApp(
    ui = fluidPage(fluidRow(column(12, DT::dataTableOutput('tbl')))),
    server = function(input, output) {
        output$tbl = DT::renderDataTable(
            as.datatable_widget(mtcars_table)
        )
    }
)

## End(Not run)
```

`as.dichotomy` *Convert variable (possibly multiple choice question) to data.frame/matrix of dummy variables.*

Description

This function converts variable/multiple response variable (vector/matrix/data.frame) with category encoding into data.frame/matrix with dichotomy encoding (0/1) suited for most statistical analysis, e. g. clustering, factor analysis, linear regression and so on.

- `as.dichotomy` returns data.frame of class 'dichotomy' with 0, 1 and possibly NA.
- `dummy` returns matrix of class 'dichotomy' with 0, 1 and possibly NA.
- `dummy1` drops last column in dichotomy matrix. It is useful in many cases because any column of such matrix usually is linear combinations of other columns.

Usage

```
as.dichotomy(
  x,
  prefix = "v",
  keep_unused = FALSE,
  use_na = TRUE,
  keep_values = NULL,
  keep_labels = NULL,
  drop_values = NULL,
  drop_labels = NULL,
  presence = 1,
  absence = 0
)

dummy(
  x,
  keep_unused = FALSE,
  use_na = TRUE,
  keep_values = NULL,
  keep_labels = NULL,
  drop_values = NULL,
  drop_labels = NULL,
  presence = 1,
  absence = 0
)

dummy1(
  x,
  keep_unused = FALSE,
  use_na = TRUE,
  keep_values = NULL,
  keep_labels = NULL,
  drop_values = NULL,
  drop_labels = NULL,
  presence = 1,
  absence = 0
)
```

```
is.dichotomy(x)
```

Arguments

x	vector/factor/matrix/data.frame.
prefix	character. By default "v".
keep_unused	Logical. Should we create columns for unused value labels/factor levels? FALSE by default.
use_na	Logical. Should we use NA for rows with all NA or use 0's instead. TRUE by default.
keep_values	Numeric/character. Values that should be kept. By default all values will be kept.
keep_labels	Numeric/character. Labels/levels that should be kept. By default all labels/levels will be kept.
drop_values	Numeric/character. Values that should be dropped. By default all values will be kept. Ignored if keep_values/keep_labels are provided.
drop_labels	Numeric/character. Labels/levels that should be dropped. By default all labels/levels will be kept. Ignored if keep_values/keep_labels are provided.
presence	numeric value which will code presence of the level. By default it is 1. Note that all tables functions need that presence and absence will be 1 and 0.
absence	numeric value which will code absence of the level. By default it is 0. Note that all tables functions need that presence and absence will be 1 and 0.

Value

as.dichotomy returns data.frame of class dichotomy with 0,1. Columns of this data.frame have variable labels according to value labels of original data. If label doesn't exist for particular value then this value will be used as variable label. dummy returns matrix of class dichotomy. Column names of this matrix are value labels of original data.

See Also

as.category for reverse conversion, mrset, mdset for usage multiple-response variables with tables.

Examples

```
# toy example
# brands - multiple response question
# Which brands do you use during last three months?
set.seed(123)
brands = as.sheet(t(replicate(20,sample(c(1:5,NA),4,replace = FALSE))))
# score - evaluation of tested product
score = sample(-1:1,20,replace = TRUE)
var_lab(brands) = "Used brands"
val_lab(brands) = autonum("
```

```
                                Brand A
                                Brand B
                                Brand C
                                Brand D
                                Brand E
                                ")

var_lab(score) = "Evaluation of tested brand"
val_lab(score) = make_labels("
                               -1 Dislike it
                                0 So-so
                                1 Like it
                               ")

cro_cpct(as.dichotomy(brands), score)
# the same as
cro_cpct(mrset(brands), score)

# customer segmentation by used brands
kmeans(dummy(brands), 3)

# model of influence of used brands on evaluation of tested product
summary(lm(score ~ dummy(brands)))

# prefixed data.frame
as.dichotomy(brands, prefix = "brand_")
```

`as.etable` *Convert data.frame/matrix to object of class 'etable'*

Description

If `x` is `data.frame` then `as.etable` just adds `etable` to `class` attribute of `x`. If `x` is matrix then it will be converted to data.frame.

Usage

```
as.etable(x, rownames_as_row_labels = NULL)

is.etable(x)
```

Arguments

`x`	data.frame/matrix
`rownames_as_row_labels`	logical. If it is TRUE than rownames of `x` will be added to result as first column with name `row_labels`. By default row names will be added if they are not NULL and are not sequential numerics.

Value

object of class `etable`

Examples

```
data(mtcars)
etable_mtcars = as.etable(mtcars)
is.etable(etable_mtcars) #TRUE

etable_mtcars #another 'print' method is used

cor(mtcars) %>% as.etable()
```

`as.labelled` *Recode vector into numeric vector with value labels*

Description

Recode vector into numeric vector with value labels

Usage

```
as.labelled(x, label = NULL)

is.labelled(x)
```

Arguments

`x`	numeric vector/character vector/factor
`label`	optional variable label

Value

numeric vector with value labels

Examples

```
character_vector = c("one", "two",  "two", "three")
as.labelled(character_vector, label = "Numbers")
```

`as_huxtable.etable` *Convert table to huxtable*

Description

This function converts a `etable` object to a `huxtable`. The `huxtable-package` needs to be installed to use this function.

Usage

```
as_huxtable.etable(x, ...)

as_hux.etable(x, ...)
```

Arguments

`x`	etable. Table to convert to a huxtable.
`...`	arguments passed on to huxtable.

Details

`huxtable` allows to export formated tables to LaTeX, HTML, Microsoft Word, Microsoft Excel, Microsoft Powerpoint, RTF and Markdown.

Tables in knitr or rmarkdown documents of type LaTeX or Word are converted by default.

Examples

```
## Not run:
library(huxtable)
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

tab = mtcars %>%
```

```
    tab_cols(total(), am %nest% vs) %>%
    tab_cells(mpg, hp) %>%
    tab_stat_mean() %>%
    tab_cells(cyl) %>%
    tab_stat_cpct() %>%
    tab_pivot() %>%
    set_caption("Table 1. Some variables from mtcars dataset.")

ht = as_huxtable(tab)
ht

## End(Not run)
```

`compare_proportions` *Calculate significance (p-values) of differences between proportions/means*

Description

`compare_proportions` calculates p-values (via z-test) for comparison between each proportion in the `prop1` and `prop2`. Results are calculated with the same formula as in prop.test without continuity correction. `compare_means` calculates p-values (via t-test) for comparison between each mean in the `mean1` and `mean2`. Results are calculated on the aggregated statistics (means, std. devs, N) with the same formula as in t.test. These functions mainly intended for usage inside significance_cpct and significance_means.

Usage

```
compare_proportions(prop1, prop2, base1, base2, common_base = 0)

compare_means(
  mean1,
  mean2,
  sd1,
  sd2,
  base1,
  base2,
  common_base = 0,
  var_equal = FALSE
)
```

Arguments

`prop1`	a numeric vector of proportions in the group 1. Values should be between 0 and 1
`prop2`	a numeric vector of proportions in the group 2. Values should be between 0 and 1

base1	a numeric vector for compare_means and single number for compare_proportions. Number of valid cases for each mean in the first group for compare_means and number of cases for compare_proportions.
base2	a numeric vector for compare_means and single number for compare_proportions. Number of valid cases for each mean in the second group for compare_means and number of cases for compare_proportions.
common_base	numeric. Number of cases that belong to both values in the first and the second argument. It can occur in the case of overlapping samples. Calculations are made according to algorithm in IBM SPSS Statistics Algorithms v20, p. 263. Note that with these adjustments t-tests between means are made with equal variance assumed (as with var_equal = TRUE).
mean1	a numeric vector of the means in the first group.
mean2	a numeric vector of the means in the second group.
sd1	a numeric vector of the standard deviations in the first group. Values should be non-negative.
sd2	a numeric vector of the standard deviations in the second group. Values should be non-negative.
var_equal	a logical variable indicating whether to treat the variances in the groups as being equal. For details see t.test.

Value

numeric vector with p-values

See Also

significance_cpct, significance_means, prop.test, t.test

Examples

```
# proportions
data(mtcars)
counts = table(mtcars$am, mtcars$vs)
props = prop.table(counts)
compare_proportions(props[,1], props[,2],
                    colSums(counts)[1], colSums(counts)[1])

# means
t.test(mpg ~ am, data = mtcars)$p.value
# the same result
calculate(mtcars,
          compare_means(
              mean(mpg[am==0]), mean(mpg[am==1]),
              sd(mpg[am==0]),  sd(mpg[am==1]),
              length(mpg[am==0]), length(mpg[am==1])
          ))
```

compute *Modify data.frame/modify subset of the data.frame*

Description

- compute evaluates expression expr in the context of data.frame data and return original data possibly modified.
- calculate evaluates expression expr in the context of data.frame data and return value of the evaluated expression. Function use_labels is shortcut for calculate with argument use_labels set to TRUE. When use_labels is TRUE there is a special shortcut for entire data.frame - ..data.
- do_if modifies only rows for which cond equals to TRUE. Other rows remain unchanged. Newly created variables also will have values only in rows for which cond have TRUE. There will be NA's in other rows. This function tries to mimic SPSS "DO IF(). ... END IF." statement.

Full-featured %to% is available in the expressions for addressing range of variables. There is a special constant .N which equals to number of cases in data for usage in expression inside compute/calculate. Inside do_if .N gives number of rows which will be affected by expressions. For parametrization (variable substitution) see .. or examples. Sometimes it is useful to create new empty variable inside compute. You can use .new_var function for this task. This function creates variable of length .N filled with NA. See examples. modify is an alias for compute, modify_if is an alias for do_if and calc is an alias for calculate.

Usage

```
compute(data, ...)

modify(data, ...)

do_if(data, cond, ...)

modify_if(data, cond, ...)

calculate(data, expr, use_labels = FALSE)

use_labels(data, expr)

calc(data, expr, use_labels = FALSE)

data %calc% expr

data %use_labels% expr

data %calculate% expr
```

Arguments

data	data.frame/list of data.frames. If data is list of data.frames then expression expr will be evaluated inside each data.frame separately.
...	expressions that should be evaluated in the context of data.frame data. It can be arbitrary code in curly brackets or assignments. See examples.
cond	logical vector or expression. Expression will be evaluated in the context of the data.
expr	expression that should be evaluated in the context of data.frame data
use_labels	logical. Experimental feature. If it equals to TRUE then we will try to replace variable names with labels. So many base R functions which show variable names will show labels.

Value

compute and do_if functions return modified data.frame/list of modified data.frames, calculate returns value of the evaluated expression/list of values.

Examples

```
dfs = data.frame(
    test = 1:5,
    a = rep(10, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_3 = rep(13, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)


# compute sum of b* variables and attach it to 'dfs'
compute(dfs, {
    b_total = sum_row(b_1 %to% b_5)
    var_lab(b_total) = "Sum of b"
    random_numbers = runif(.N) # .N usage
})

# calculate sum of b* variables and return it
calculate(dfs, sum_row(b_1 %to% b_5))


# set values to existing/new variables
compute(dfs, {
    (b_1 %to% b_5) %into% text_expand('new_b{1:5}')
})

# .new_var usage
compute(dfs, {
    new_var = .new_var()
    new_var[1] = 1 # this is not possible without preliminary variable creation
```

```
})

# conditional modification
do_if(dfs, test %in% 2:4, {
    a = a + 1
    b_total = sum_row(b_1 %to% b_5)
    random_numbers = runif(.N) # .N usage
})


# variable substitution
name1 = "a"
name2 = "new_var"

compute(dfs, {
     ..$name2 = ..$name1*2
})

compute(dfs, {
     for(name1 in paste0("b_", 1:5)){
         name2 = paste0("new_", name1)
         ..$name2 = ..$name1*2
     }
     rm(name1, name2) # we don't need this variables as columns in 'dfs'
})

# square brackets notation
compute(dfs, {
     ..[(name2)] = ..[(name1)]*2
})

compute(dfs, {
     for(name1 in paste0("b_", 1:5)){
         ..[paste0("new_", name1)] = ..$name1*2
     }
     rm(name1) # we don't need this variable as column in 'dfs'
})

# '..$' doesn't work for case below so we need to use square brackets form
name1 = paste0("b_", 1:5)
name2 = paste0("new_", name1)
compute(dfs, {
     for(i in 1:5){
         ..[name2[i]] = ..[name1[i]]*3
     }
     rm(i) # we don't need this variable as column in 'dfs'
})

# 'use_labels' examples. Utilization of labels in base R.
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
```

```
                        disp = "Displacement (cu.in.)",
                        hp = "Gross horsepower",
                        drat = "Rear axle ratio",
                        wt = "Weight (lb/1000)",
                        qsec = "1/4 mile time",
                        vs = "Engine",
                        vs = c("V-engine" = 0,
                               "Straight engine" = 1),
                        am = "Transmission",
                        am = c("Automatic" = 0,
                               "Manual"=1),
                        gear = "Number of forward gears",
                        carb = "Number of carburetors"
)

use_labels(mtcars, table(am, vs))

## Not run:
use_labels(mtcars, plot(mpg, hp))

## End(Not run)

mtcars %>%
       use_labels(lm(mpg ~ disp + hp + wt)) %>%
       summary()
```

---

count_if *Count/sum/average/other functions on values that meet a criterion*

Description

These functions calculate count/sum/average/etc. on values that meet a criterion that you specify. `apply_if_*` apply custom functions. There are different flavors of these functions: `*_if` work on entire dataset/matrix/vector, `*_row_if` works on each row and `*_col_if` works on each column.

Usage

```
count_if(criterion, ...)

count_row_if(criterion, ...)

count_col_if(criterion, ...)

x %row_in% criterion

x %has% criterion

x %col_in% criterion
```

```
sum_if(criterion, ..., data = NULL)

sum_row_if(criterion, ..., data = NULL)

sum_col_if(criterion, ..., data = NULL)

mean_if(criterion, ..., data = NULL)

mean_row_if(criterion, ..., data = NULL)

mean_col_if(criterion, ..., data = NULL)

sd_if(criterion, ..., data = NULL)

sd_row_if(criterion, ..., data = NULL)

sd_col_if(criterion, ..., data = NULL)

median_if(criterion, ..., data = NULL)

median_row_if(criterion, ..., data = NULL)

median_col_if(criterion, ..., data = NULL)

max_if(criterion, ..., data = NULL)

max_row_if(criterion, ..., data = NULL)

max_col_if(criterion, ..., data = NULL)

min_if(criterion, ..., data = NULL)

min_row_if(criterion, ..., data = NULL)

min_col_if(criterion, ..., data = NULL)

apply_row_if(fun, criterion, ..., data = NULL)

apply_col_if(fun, criterion, ..., data = NULL)
```

Arguments

criterion	Vector with counted values or function. See details and examples.
...	Data on which criterion will be applied. Vector, matrix, data.frame, list.
x	Data on which criterion will be applied. Vector, matrix, data.frame, list.
data	Data on which function will be applied. Doesn't applicable to `count_*_if` functions. If omitted then function will be applied on the ... argument.

`fun` Custom function that will be applied based on criterion.

Details

Possible type for criterion argument:

- vector/single value All values in `...` which equal to the elements of vector in the criteria will be used as function `fun` argument.
- function Values for which function gives TRUE will be used as function `fun` argument. There are some special functions for convenience (e. g. `gt(5)` is equivalent ">5" in spreadsheet) - see criteria.

`count*` and `%in*%` never returns NA's. Other functions remove NA's before calculations (as `na.rm = TRUE` in base R functions).

Function criterion should return logical vector of same size and shape as its argument. This function will be applied to each column of supplied data and TRUE results will be used. There is asymmetrical behavior in `*_row_if` and `*_col_if` for function criterion: in both cases function criterion will be applied columnwise.

Value

`*_if` return single value (vector of length 1). `*_row_if` returns vector for each row of supplied arguments. `*_col_if` returns vector for each column of supplied arguments. `%row_in%/%col_in%` return logical vector - indicator of presence of criterion in each row/column. `%has%` is an alias for `%row_in%`.

Examples

```
set.seed(123)
sheet1 = as.sheet(
       matrix(sample(c(1:10,NA), 30, replace = TRUE), 10)
)

result  = compute(sheet1, {
             # count 8
             exact = count_row_if(8, V1, V2, V3)
             # count values greater than 8
             greater = count_row_if(gt(8), V1, V2, V3)
             # count integer values between 5 and 8, e. g. 5, 6, 7, 8
             integer_range = count_row_if(5:8, V1, V2, V3)
             # count values between 5 and 8
             range = count_row_if(5 %thru% 8, V1, V2, V3)
             # count NA
             na = count_row_if(is.na, V1, V2, V3)
             # count not-NA
             not_na = count_row_if(not_na, V1, V2, V3)
             # are there any 5 in each row?
             has_five = cbind(V1, V2, V3) %row_in% 5
         })
result
```

```
mean_row_if(6, sheet1$V1, data = sheet1)
median_row_if(gt(2), sheet1$V1, sheet1$V2, sheet1$V3)
sd_row_if(5 %thru% 8, sheet1$V1, sheet1$V2, sheet1$V3)

if_na(sheet1) = 5 # replace NA

# custom apply
apply_col_if(prod, gt(2), sheet1$V1, data = sheet1) # product of all elements by columns
apply_row_if(prod, gt(2), sheet1$V1, data = sheet1) # product of all elements by rows

# Examples borrowed from Microsoft Excel help for COUNTIF
sheet1 = text_to_columns(
    "
       a       b
    apples    32
    oranges   54
    peaches   75
    apples    86
    "
)

count_if("apples", sheet1$a) # 2

count_if("apples", sheet1) # 2

calc(sheet1, count_if("apples", a, b)) # 2

count_if(gt(55), sheet1$b) # greater than 55 = 2

count_if(ne(75), sheet1$b) # not equal 75 = 3

count_if(ge(32), sheet1$b) # greater than or equal 32 = 4

count_if(gt(32) & lt(86), sheet1$b) # 2

# count only integer values between 33 and 85
count_if(33:85, sheet1$b) # 2

# values with letters
count_if(regex("^[A-z]+$"), sheet1) # 4

# values that started on 'a'
count_if(regex("^a"), sheet1) # 2

# count_row_if
count_row_if(regex("^a"), sheet1) # c(1,0,0,1)

sheet1 %row_in% 'apples'  # c(TRUE,FALSE,FALSE,TRUE)

# Some of Microsoft Excel examples for SUMIF/AVERAGEIF/etc
sheet1 = text_to_columns(
    "
    property_value commission data
```

```
    100000              7000  250000
    200000             14000
    300000             21000
    400000             28000
    "
)

# Sum of commision for property value greater than 160000
calc(sheet1, sum_if(gt(160000), property_value, data = commission)) # 63000

# Sum of property value greater than 160000
calc(sheet1, sum_if(gt(160000), property_value)) # 900000

# Sum of commision for property value equals to 300000
calc(sheet1, sum_if(300000, property_value, data = commission)) # 21000

# Sum of commision for property value greater than first value of data
calc(sheet1, sum_if(gt(data[1]), property_value, data = commission)) # 49000

sheet1 = text_to_columns(
       "
         category     food sales
       Vegetables Tomatoes  2300
       Vegetables   Celery  5500
           Fruits  Oranges   800
             NA     Butter   400
       Vegetables  Carrots  4200
           Fruits   Apples  1200
       "
       )

# Sum of sales for Fruits
calc(sheet1, sum_if("Fruits", category, data = sales)) # 2000

# Sum of sales for Vegetables
calc(sheet1, sum_if("Vegetables", category, data = sales)) # 12000

# Sum of sales for food which is ending on 'es'
calc(sheet1, sum_if(perl("es$"), food, data = sales)) # 4300

# Sum of sales for empty category
calc(sheet1, sum_if(NA, category, data = sales))  # 400


sheet1 = text_to_columns(
    "
    property_value commission data
    100000              7000  250000
    200000             14000
    300000             21000
    400000             28000
    "
)
```

```
# Commision average for comission less than 23000
calc(sheet1, mean_if(lt(23000), commission)) # 14000


# Property value average for property value less than 95000
calc(sheet1, mean_if(lt(95000), property_value)) #  NaN

# Commision average for property value greater than 250000
calc(sheet1, mean_if(gt(250000), property_value, data = commission)) # 24500


sheet1 = text_to_columns(
    '
                region  profits
                  East   45678
                  West   23789
                 North   -4789
    "South (New Office)"     0
               MidWest    9678
    ',
    quote = '"'
)


# Mean profits for 'west' regions
calc(sheet1, mean_if(contains("West"), region, data = profits)) # 16733.5


# Mean profits for regions wich doesn't contain New Office
calc(sheet1, mean_if(not(contains("New Office")), region, data = profits))  # 18589


sheet1 = text_to_columns(
    "
    grade weight
    89      1
    93      2
    96      2
    85      3
    91      1
    88      1
    "
)

# Minimum grade for weight equals to 1
calc(sheet1, min_if(1, weight, data = grade)) # 88


# Maximum grade for weight equals to 1
calc(sheet1, max_if(1, weight, data = grade)) #91
```

```
# Example with offset
sheet1 = text_to_columns(
    "
    weight grade
       10    b
       11    a
      100    a
      111    b
        1    a
        1    a
    "
)

calc(sheet1, min_if("a", grade[2:5], data = weight[1:4])) # 10
```

`criteria` *Criteria functions*

Description

Produce criteria which could be used in the different situations - see 'recode', 'na_if', 'count_if', 'match_row', '%i%' and etc. For example, `'greater(5)'` returns function which tests whether its argument greater than five. `'fixed("apple")'` returns function which tests whether its argument contains "apple". For criteria logical operations (|, &, !, xor) are defined, e. g. you can write something like: `'greater(5) | equals(1)'`. List of functions:

- comparison criteria - `'equals'`, `'greater'` and etc. return functions which compare its argument against value.
- `'thru'` checks whether a value is inside interval. `'thru(0,1)'` is equivalent to `'x>=0 & x<=1'`
- `'%thru%'` is infix version of `'thru'`, e. g. `'0 %thru% 1'`
- `'is_max'` and `'is_min'` return TRUE where vector value is equals to maximum or minimum.
- `'contains'` searches for the pattern in the strings. By default, it works with fixed patterns rather than regular expressions. For details about its arguments see grepl
- `'like'` searches for the Excel-style pattern in the strings. You can use wildcards: '*' means any number of symbols, '?' means single symbol. Case insensitive.
- `'fixed'` alias for contains.
- `'perl'` such as `'contains'` but the pattern is perl-compatible regular expression (`'perl = TRUE'`). For details see grepl
- `'regex'` use POSIX 1003.2 extended regular expressions (`'fixed = FALSE'`). For details see grepl
- `'has_label'` searches values which have supplied label(-s). We can used criteria as an argument for 'has_label'.
- `'to'` returns function which gives TRUE for all elements of vector before the first occurrence of `'x'` and for `'x'`.

- 'from' returns function which gives TRUE for all elements of vector after the first occurrence of 'x' and for 'x'.
- 'not_na' returns TRUE for all non-NA vector elements.
- 'other' returns TRUE for all vector elements. It is intended for usage with 'recode'.
- 'items' returns TRUE for the vector elements with the given sequential numbers.
- 'and', 'or', 'not' are spreadsheet-style boolean functions.

Shortcuts for comparison criteria:

- 'equals' - 'eq'
- 'not_equals' - 'neq', 'ne'
- 'greater' - 'gt'
- 'greater_or_equal' - 'gte', 'ge'
- 'less' - 'lt'
- 'less_or_equal' - 'lte', 'le'

Usage

```
as.criterion(crit)

is.criterion(x)

equals(x)

not_equals(x)

less(x)

less_or_equal(x)

greater(x)

greater_or_equal(x)

thru(lower, upper)

lower %thru% upper

when(x)

is_max(x)

is_min(x)

contains(
  pattern,
  ignore.case = FALSE,
```

```
  perl = FALSE,
  fixed = TRUE,
  useBytes = FALSE
)

like(pattern)

fixed(
  pattern,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = TRUE,
  useBytes = FALSE
)

perl(
  pattern,
  ignore.case = FALSE,
  perl = TRUE,
  fixed = FALSE,
  useBytes = FALSE
)

regex(
  pattern,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)

has_label(x)

from(x)

to(x)

items(...)

not_na(x)

is_na(x)

other(x)

and(...)

or(...)
```

```
not(x)
```

Arguments

crit	vector of values/function which returns logical or logical vector. It will be converted to function of class criterion.
x	vector
lower	vector/single value - lower bound of interval
upper	vector/single value - upper bound of interval
pattern	character string containing a regular expression (or character string for 'fixed') to be matched in the given character vector. Coerced by as.character to a character string if possible.
ignore.case	logical see grepl
perl	logical see grepl
fixed	logical see grepl
useBytes	logical see grepl
...	numeric indexes of desired items for items, logical vectors or criteria for boolean functions.

Value

function of class 'criterion' which tests its argument against condition and return logical value

See Also

recode, count_if, match_row, na_if, %i%

Examples

```
# operations on vector, '%d%' means 'diff'
1:6 %d% greater(4) # 1:4
1:6 %d% (1 | greater(4)) # 2:4
# '%i%' means 'intersect
1:6 %i% (is_min() | is_max()) # 1, 6
# with Excel-style boolean operators
1:6 %i% or(is_min(), is_max()) # 1, 6

letters %i% (contains("a") | contains("z")) # a, z

letters %i% perl("a|z") # a, z

letters %i% from("w")  # w, x, y, z

letters %i% to("c")  # a, b, c

letters %i% (from("b") & to("e"))  # b, d, e
```

```
c(1, 2, NA, 3) %i% not_na() # c(1, 2, 3)

# examples with count_if
df1 = data.frame(
    a=c("apples", "oranges", "peaches", "apples"),
    b = c(32, 54, 75, 86)
)

count_if(greater(55), df1$b) # greater than 55 = 2

count_if(not_equals(75), df1$b) # not equals 75 = 3

count_if(greater(32) & less(86), df1$b) # greater than 32 and less than 86 = 2
count_if(and(greater(32), less(86)), df1$b) # the same result

# infix version
count_if(35 %thru% 80, df1$b) # greater than or equals to 35 and less than or equals to 80 = 2

# values that started on 'a'
count_if(like("a*"), df1) # 2

# the same with Perl-style regular expression
count_if(perl("^a"), df1) # 2

# count_row_if
count_row_if(perl("^a"), df1) # c(1,0,0,1)

# examples with 'n_intersect' and 'n_diff'
data(iris)
iris %>% n_intersect(to("Petal.Width")) # all columns up to 'Species'

# 'Sepal.Length', 'Sepal.Width' will be left
iris %>% n_diff(from("Petal.Length"))

# except first column
iris %n_d% items(1)

# 'recode' examples
qvar = c(1:20, 97, NA, NA)
recode(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, 11 %thru% hi ~ 3, other ~ 0)
# the same result
recode(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, greater_or_equal(11) ~ 3, other ~ 0)
```

---

cro *Cross tabulation with support of labels, weights and multiple response variables.*

Description

- `cro`, `cro_cases` build a contingency table of the counts.
- `cro_cpct`, `cro_cpct_responses` build a contingency table of the column percent. These functions give different results only for multiple response variables. For `cro_cpct` base of percent is number of valid cases. Case is considered as valid if it has at least one non-NA value. So for multiple response variables sum of percent may be greater than 100. For `cro_cpct_responses` base of percent is number of valid responses. Multiple response variables can have several responses for single case. Sum of percent of `cro_cpct_responses` always equals to 100%.
- `cro_rpct` build a contingency table of the row percent. Base for percent is number of valid cases.
- `cro_tpct` build a contingency table of the table percent. Base for percent is number of valid cases.
- `calc_cro_*` are the same as above but evaluate their arguments in the context of the first argument `data`.
- `total` auxiliary function - creates variables with 1 for valid case of its argument `x` and NA in opposite case.

You can combine tables with add_rows and merge.etable. For sorting table see tab_sort_asc. To provide multiple-response variables as arguments use mrset for multiples with category encoding and mdset for multiples with dichotomy (dummy) encoding. To compute statistics with nested variables/banners use nest. For more sophisticated interface with modern piping via `magrittr` see tables.

Usage

```
cro(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

cro_cases(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)
```

```
cro_cpct(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

cro_rpct(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

cro_tpct(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

cro_cpct_responses(
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_responses",
  total_row_position = c("below", "above", "none")
)

calc_cro(
  data,
  cell_vars,
```

```
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

calc_cro_cases(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

calc_cro_cpct(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

calc_cro_rpct(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

calc_cro_tpct(
  data,
  cell_vars,
```

```
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none")
)

calc_cro_cpct_responses(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = NULL,
  weight = NULL,
  subgroup = NULL,
  total_label = NULL,
  total_statistic = "u_responses",
  total_row_position = c("below", "above", "none")
)

total(x = 1, label = "#Total")
```

Arguments

cell_vars	vector/data.frame/list. Variables on which percentage/cases will be computed. Use mrset/mdset for multiple-response variables.
col_vars	vector/data.frame/list. Variables which breaks table by columns. Use mrset/mdset for multiple-response variables.
row_vars	vector/data.frame/list. Variables which breaks table by rows. Use mrset/mdset for multiple-response variables.
weight	numeric vector. Optional cases weights. Cases with NA's, negative and zero weights are removed before calculations.
subgroup	logical vector. You can specify subgroup on which table will be computed.
total_label	By default "#Total". You can provide several names - each name for each total statistics.
total_statistic	By default it is "u_cases" (unweighted cases). Possible values are "u_cases", "u_responses", "u_cpct", "u_rpct", "u_tpct", "w_cases", "w_responses", "w_cpct", "w_rpct", "w_tpct". "u_" means unweighted statistics and "w_" means weighted statistics.
total_row_position	Position of total row in the resulting table. Can be one of "below", "above", "none".
data	data.frame in which context all other arguments will be evaluated (for calc_cro_*).
x	vector/data.frame of class 'category'/'dichotomy'.
label	character. Label for total variable.

Value

object of class 'etable'. Basically it's a data.frame but class is needed for custom methods.

See Also

tables, fre, cro_fun.

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

calculate(mtcars, cro(am, vs))
calc_cro(mtcars, am, vs) # the same result

# column percent with multiple banners
calculate(mtcars, cro_cpct(cyl, list(total(), vs, am)))
calc_cro_cpct(mtcars, cyl, list(total(), vs, am)) # the same result

# nested banner
calculate(mtcars, cro_cpct(cyl, list(total(), vs %nest% am)))

# stacked variables
calculate(mtcars, cro(list(cyl, carb), list(total(), vs %nest% am)))

# nested variables
calculate(mtcars, cro_cpct(am %nest% cyl, list(total(), vs)))

# row variables
calculate(mtcars, cro_cpct(cyl, list(total(), vs), row_vars = am))

# several totals above table
calculate(mtcars, cro_cpct(cyl,
              list(total(), vs),
              row_vars = am,
              total_row_position = "above",
```

```
                total_label = c("number of cases", "row %"),
                total_statistic = c("u_cases", "u_rpct")
                ))

  # multiple-choice variable
  # brands - multiple response question
  # Which brands do you use during last three months?
  set.seed(123)
  brands = data.frame(t(replicate(20,sample(c(1:5,NA),4,replace = FALSE))))
  # score - evaluation of tested product
  score = sample(-1:1,20,replace = TRUE)
  var_lab(brands) = "Used brands"
  val_lab(brands) = make_labels("
                                1 Brand A
                                2 Brand B
                                3 Brand C
                                4 Brand D
                                5 Brand E
                                ")

  var_lab(score) = "Evaluation of tested brand"
  val_lab(score) = num_lab("
                               -1 Dislike it
                               0 So-so
                               1 Like it
                               ")

  cro_cpct(mrset(brands), list(total(), score))
  # responses
  cro_cpct_responses(mrset(brands), list(total(), score))
```

`cro_fun` *Cross-tabulation with custom summary function.*

Description

- `cro_mean`, `cro_sum`, `cro_median` calculate mean/sum/median by groups. NA's are always omitted.
- `cro_mean_sd_n` calculates mean, standard deviation and N simultaneously. Mainly intended for usage with significance_means.
- `cro_pearson`, `cro_spearman` calculate correlation of first variable in each data.frame in `cell_vars` with other variables. NA's are removed pairwise.
- `cro_fun`, `cro_fun_df` return table with custom summary statistics defined by `fun` argument. NA's treatment depends on your `fun` behavior. To use weight you should have formal `weight` argument in `fun` and some logic for its processing inside. Several functions with weight support are provided - see w_mean. `cro_fun` applies `fun` on each variable in `cell_vars` separately, `cro_fun_df` gives to `fun` each data.frame in `cell_vars` as a whole. So `cro_fun(iris[,-5],iris$Species,fun = mean)` gives the same result as `cro_fun_df(iris[,-5],iris$Species,f`

= colMeans). For cro_fun_df names of cell_vars will converted to labels if they are available before the fun will be applied. Generally it is recommended that fun will always return object of the same form. Row names/vector names of fun result will appear in the row labels of the table and column names/names of list will appear in the column labels. If your fun returns data.frame/matrix/list with element named 'row_labels' then this element will be used as row labels. And it will have precedence over rownames.

- calc_cro_* are the same as above but evaluate their arguments in the context of the first argument data.
- combine_functions is auxiliary function for combining several functions into one function for usage with cro_fun/cro_fun_df. Names of arguments will be used as statistic labels. By default, results of each function are combined with c. But you can provide your own method function with method argument. It will be applied as in the expression do.call(method,list_of_functions_results). Particular useful method is list. When it used then statistic labels will appear in the column labels. See examples. Also you may be interested in data.frame, rbind, cbind methods.

Usage

```
cro_fun(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL,
  fun,
  ...,
  unsafe = FALSE
)

cro_fun_df(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL,
  fun,
  ...,
  unsafe = FALSE
)

cro_mean(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)
```

```
cro_mean_sd_n(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL,
  weighted_valid_n = FALSE,
  labels = NULL
)

cro_sum(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

cro_median(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

cro_pearson(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

cro_spearman(
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

calc_cro_fun(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
```

```
  subgroup = NULL,
  fun,
  ...,
  unsafe = FALSE
)

calc_cro_fun_df(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL,
  fun,
  ...,
  unsafe = FALSE
)

calc_cro_mean(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

calc_cro_mean_sd_n(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL,
  weighted_valid_n = FALSE,
  labels = NULL
)

calc_cro_sum(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

calc_cro_median(
```

```
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

calc_cro_pearson(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

calc_cro_spearman(
  data,
  cell_vars,
  col_vars = total(),
  row_vars = total(label = ""),
  weight = NULL,
  subgroup = NULL
)

combine_functions(..., method = c)
```

Arguments

cell_vars	vector/data.frame/list. Variables on which summary function will be computed.
col_vars	vector/data.frame/list. Variables which breaks table by columns. Use mrset/mdset for multiple-response variables.
row_vars	vector/data.frame/list. Variables which breaks table by rows. Use mrset/mdset for multiple-response variables.
weight	numeric vector. Optional cases weights. Cases with NA's, negative and zero weights are removed before calculations.
subgroup	logical vector. You can specify subgroup on which table will be computed.
fun	custom summary function. Generally it is recommended that fun will always return object of the same form. Rownames/vector names of fun result will appear in the row labels of the table and column names/names of list will appear in the column labels. To use weight you should have formal weight argument in fun and some logic for its processing inside. For cro_fun_df fun will receive data.table with all names converted to variable labels (if labels exists). So it is not recommended to rely on original variables names in your fun.
...	further arguments for fun in cro_fun/cro_fun_df or functions for combine_functions. Ignored in cro_fun/cro_fun_df if unsafe is TRUE.

unsafe	logical/character If not FALSE than fun will be evaluated as is. It can lead to significant increase in the performance. But there are some limitations. For cro_fun it means that your function fun should return vector. If length of this vector is greater than one than you should provide with unsafe argument vector of unique labels for each element of this vector. There will be no attempts to automatically make labels for the results of fun. For cro_fun_df your function should return vector or list/data.frame (optionally with 'row_labels' element - statistic labels). If unsafe is TRUE or not logical then further arguments (...) for fun will be ignored.
weighted_valid_n	logical. Should we show weighted valid N in cro_mean_sd_n? By default it is FALSE.
labels	character vector of length 3. Labels for mean, standard deviation and valid N in cro_mean_sd_n.
data	data.frame in which context all other arguments will be evaluated (for calc_cro_*).
method	function which will combine results of multiple functions in combine_functions. It will be applied as in the expression do.call(method,list_of_functions_results). By default it is c.

Value

object of class 'etable'. Basically it's a data.frame but class is needed for custom methods.

See Also

tables, fre, cro.

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)


# Simple example - there is special shortcut for it - 'cro_mean'
```

```
calculate(mtcars, cro_fun(list(mpg, disp, hp, wt, qsec),
                          col_vars = list(total(), am),
                          row_vars = vs,
                          fun = mean)
)

# the same result
calc_cro_fun(mtcars, list(mpg, disp, hp, wt, qsec),
                     col_vars = list(total(), am),
                     row_vars = vs,
                     fun = mean
)

# The same example with 'subgroup'
calculate(mtcars, cro_fun(list(mpg, disp, hp, wt, qsec),
                          col_vars = list(total(), am),
                          row_vars = vs,
                          subgroup = vs == 0,
                          fun = mean)
)

# 'combine_functions' usage
calculate(mtcars, cro_fun(list(mpg, disp, hp, wt, qsec),
                          col_vars = list(total(), am),
                          row_vars = vs,
                          fun = combine_functions(Mean = mean,
                                                  'Std. dev.' = sd,
                                                  'Valid N' = valid_n)
))
# 'combine_functions' usage - statistic labels in columns
calculate(mtcars, cro_fun(list(mpg, disp, hp, wt, qsec),
                          col_vars = list(total(), am),
                          row_vars = vs,
                          fun = combine_functions(Mean = mean,
                                                  'Std. dev.' = sd,
                                                  'Valid N' = valid_n,
                                                  method = list
                                                  )
))

# 'summary' function
calculate(mtcars, cro_fun(list(mpg, disp, hp, wt, qsec),
                          col_vars = list(total(), am),
                          row_vars = list(total(), vs),
                          fun = summary
))

# comparison 'cro_fun' and 'cro_fun_df'
calculate(mtcars, cro_fun(
                        sheet(mpg, disp, hp, wt, qsec),
                        col_vars = am,
                        fun = mean
                        )
```

```
)

# same result
calculate(mtcars, cro_fun_df(
                       sheet(mpg, disp, hp, wt, qsec),
                       col_vars = am,
                       fun = colMeans
                       )
)

# usage for 'cro_fun_df' which is not possible for 'cro_fun'
# linear regression by groups
calculate(mtcars, cro_fun_df(
                      sheet(mpg, disp, hp, wt, qsec),
                      col_vars = am,
                      fun = function(x){
                            frm = reformulate(".", response = names(x)[1])
                            model = lm(frm, data = x)
                            sheet(
                                'Coef. estimate' = coef(model),
                                 confint(model)
                                 )
                      }
))
```

`default_dataset` *Get or set reference to default dataset. Experimental feature.*

Description

Use data.frame or data.frame name to set it as default. Use NULL as an argument to disable default dataset. If argument is missing then function will return reference to default dataset. Use ref to modify it. Also see .compute for usage patterns.

Usage

```
default_dataset(x)
```

Arguments

x	data.frame or data.frame name which we want to make default for some operations.

Value

formula reference to default dataset or NULL

See Also

ref

Examples

```
data(iris)
default_iris = iris
default_dataset(default_iris) # set default dataset

.compute({
    new_col = 1
    Sepal.Length = Sepal.Length*2
})

# for comparison

iris$new_col = 1
iris$Sepal.Length = iris$Sepal.Length*2
identical(iris, default_iris) # should be TRUE

default_dataset(NULL) # disable default dataset
```

`do_repeat` *Repeats the same transformations on a specified set of variables/values*

Description

Repeats the same transformations on a specified set of variables/values

Usage

```
do_repeat(data, ...)

.do_repeat(...)

as_is(...)
```

Arguments

`data`	data.frame/list. If `data` is list then `do_repeat` will be applied to each element of the list.
`...`	stand-in name(s) followed by equals sign and a vector/list of replacement variables or values. They can be numeric/characters or variables names. Names at the top-level can be unquoted (non-standard evaluation). Quoted characters also considered as variables names. To avoid this behavior use `as_is` function. For standard evaluation of parameters you can surround them by round brackets. Also you can use %to% operator and other criteria functions. Last argument should be expression in curly brackets which will be evaluated in the scope of data.frame `data`. See examples.

Details

There is a special constant `.N` which equals to number of cases in `data` for usage in expression inside `do_repeat`. Also there are a variables `.item_num` which is equal to the current iteration number and `.item_value` which is named list with current stand-in variables values.

Value

transformed data.frame `data`

See Also

compute, do_if

Examples

```
data(iris)
scaled_iris = do_repeat(iris,
                        i = Sepal.Length %to% Petal.Width,
                        {
                            i = scale(i)
                        })
head(scaled_iris)

# several stand-in names and standard evaluattion
old_names = qc(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
new_names = paste0("scaled_", old_names)
scaled_iris = do_repeat(iris,
                        orig = ((old_names)),
                        scaled = ((new_names)),
                        {
                            scaled = scale(orig)
                        })
head(scaled_iris)

# numerics
new_df = data.frame(id = 1:20)
# note the automatic creation of the sequence of variables
new_df = do_repeat(new_df,
                   item = i1 %to% i3,
                   value = c(1, 2, 3),
                   {
                       item = value
                   })
head(new_df)

# the same result with internal variable '.item_num'
new_df = data.frame(id = 1:20)
new_df = do_repeat(new_df,
                   item = i1 %to% i3,
                   {
                       item = .item_num
                   })
```

```
head(new_df)

# functions
set.seed(123)
new_df = data.frame(id = 1:20)
new_df = do_repeat(new_df,
                   item = c(i1, i2, i3),
                   fun = c("rnorm", "runif", "rexp"),
                   {
                       item = fun(.N)
                   })
head(new_df)
```

`drop_empty_rows` *Drop empty (with all NA's) rows/columns from data.frame/table*

Description

By default tables produced by functions tables, cro, cro_fun and cro_fun_df are created with all possible value labels. If values for this labels are absent in variable there are NA's in rows and columns. `drop_empty_rows`/`drop_empty_columns` are intended to remove these empty rows/columns. `drop_r` and `drop_c` are the same functions with shorter names. `drop_rc` drops rows and columns simultaneously.

Usage

```
drop_empty_rows(x, excluded_rows = NULL, excluded_columns = NULL)

drop_empty_columns(x, excluded_rows = NULL, excluded_columns = NULL)

drop_r(x, excluded_rows = NULL, excluded_columns = NULL)

drop_c(x, excluded_rows = NULL, excluded_columns = NULL)

drop_rc(x)
```

Arguments

`x`	data.frame/etable(result of cro and etc.)
`excluded_rows`	character/logical/numeric rows which won't be dropped and in which NAs won't be counted. If it is characters then they will be considered as pattern/vector of patterns. Patterns will be matched with Perl-style regular expression with values in the first column of `x` (see grep, `perl = TRUE` argument). Rows which have such patterns will be excluded. By default for class 'etable' pattern is "#" because "#" marks totals in the result of cro.

excluded_columns
: logical/numeric/characters columns which won't be dropped and in which NAs won't be counted. By default for class 'etable' it is first column - column with labels in table.

Value

data.frame with removed rows/columns

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
            vs = "Engine",
            vs = num_lab("
                      0 V-engine
                      1 Straight engine
                      9 Other
                      "),
            am = "Transmission",
            am = num_lab("
                     0 Automatic
                     1 Manual
                     9 Other
                     ")
         )
with_empty = calculate(mtcars, cro(am, vs))

drop_empty_rows(with_empty)
drop_empty_columns(with_empty)
drop_rc(with_empty)
```

---

experimental *Experimental functions for operations with default dataset*

Description

Workflow for these functions is rather simple. You should set up default data.frame with default_dataset and then operate with it without any reference to your data.frame. There are two kinds of operations. The first kind modify default dataset, the second kind will be evaluated in the context of the default dataset but doesn't modify it. It is not recommended to use one of these functions in the scope of another of these functions. By now their performance is not so high, especially `.do_if`/`.modify_if` can be very slow.

Usage

```
.compute(expr)
```

```
.do_if(cond, expr)

.modify_if(cond, expr)

.modify(expr)

.calculate(expr, use_labels = FALSE)

.calc(expr, use_labels = FALSE)

.val_lab(...)

.var_lab(...)

.recode(x, ...)

.fre(...)

.cro(...)

.cro_cases(...)

.cro_cpct(...)

.cro_rpct(...)

.cro_tpct(...)

.cro_mean(...)

.cro_sum(...)

.cro_median(...)

.cro_mean_sd_n(...)

.cro_fun(...)

.cro_fun_df(...)
```

Arguments

expr	set of expressions in curly brackets which will be evaluated in the context of default dataset
cond	logical vector/expression
use_labels	logical. Experimental feature. If it equals to TRUE then we will try to replace variable names with labels. Many base R functions which show variable names will show labels.

...	further arguments
x	vector/data.frame - variable names in the scope of default dataset

Details

Functions which modify default dataset:

- `.modify` Add and modify variables inside default data.frame. See modify.
- `.compute` Shortcut for `.modify`. Name is inspired by SPSS COMPUTE operator. See modify.
- `.modify_if` Add and modify variables inside subset of default data.frame. See modify_if.
- `.do_if` Shortcut for `.modify_if`. Name is inspired by SPSS DO IF operator. See modify_if.
- `.where` Leave subset of default data.frame which meet condition. See where, subset.
- `.recode` Change, rearrange or consolidate the values of an existing variable inside default data.frame. See recode.

Other functions:

- `.var_lab` Return variable label from default dataset. See var_lab.
- `.val_lab` Return value labels from default dataset. See val_lab.
- `.fre` Simple frequencies of variable in the default data.frame. See fre.
- `.cro`/`.cro_cpct`/`.cro_rpct`/`.cro_tpct` Simple crosstabulations of variable in the default data.frame. See cro.
- `.cro_mean`/`.cro_sum`/`.cro_median`/`.cro_fun`/`.cro_fun_df` Simple crosstabulations of variable in the default data.frame. See cro_fun.
- `.calculate` Evaluate arbitrary expression in the context of data.frame. See calculate.

Examples

```
data(mtcars)

default_dataset(mtcars) # set mtcars as default dataset

# calculate new variables
.compute({
    mpg_by_am = ave(mpg, am, FUN = mean)
    hi_low_mpg = ifs(mpg<mean(mpg) ~ 0, TRUE ~ 1)
})

# set labels
.apply_labels(
    mpg = "Miles/(US) gallon",
    cyl = "Number of cylinders",
    disp = "Displacement (cu.in.)",
    hp = "Gross horsepower",
    mpg_by_am = "Average mpg for transimission type",
    hi_low_mpg = "Miles per gallon",
    hi_low_mpg = num_lab("
                    0 Low
```

```
                        1 High
                        "),

      vs = "Engine",
      vs = num_lab("
                        0 V-engine
                        1 Straight engine
                  "),

      am = "Transmission",
      am = num_lab("
                        0 Automatic
                        1 Manual
                             ")
)
# calculate frequencies
.fre(hi_low_mpg)
.cro(cyl, hi_low_mpg)
.cro_mean(data.frame(mpg, disp, hp), vs)

# disable default dataset
default_dataset(NULL)

# Example of .recode

data(iris)

default_dataset(iris) # set iris as default dataset

.recode(Sepal.Length, lo %thru% median(Sepal.Length) ~ "small", other ~ "large")

.fre(Sepal.Length)

# example of .do_if

.do_if(Species == "setosa",{
     Petal.Length = NA
     Petal.Width = NA
})

.cro_mean(data.frame(Petal.Length, Petal.Width), Species)

# disable default dataset
default_dataset(NULL)
```

---

expss *expss: Tables with Labels and Some Useful Functions from Spreadsheets and SPSS Statistics*

Description

'expss' package implements some popular functions from spreadsheets and SPSS Statistics software. Implementations are not complete copies of their originals. I try to make them consistent with other R functions. See examples in the vignette and in the help.

Excel

- IF ifelse
- AVERAGE mean_row
- SUM sum_row
- MIN min_row
- MAX max_row
- VLOOKUP vlookup
- COUNTIF count_if
- AVERAGEIF mean_row_if
- SUMIF sum_row_if
- MINIF min_row_if
- MAXIF max_row_if
- IFS ifs
- IFNA if_na
- MATCH match_row
- INDEX index_row
- PIVOT TABLES tables, cro_fun, cro

SPSS

- COMPUTE compute
- RECODE recode
- COUNT count_row_if
- DO IF do_if
- DO REPEAT do_repeat
- VARIABLE LABELS var_lab
- VALUE LABELS val_lab
- ANY any_in_row
- FREQUENCIES fre
- CROSSTABS cro
- CUSTOM TABLES tables

expss.options *Options for controlling behavior of the package*

Description

All options can be set with `options(option.name = option.value)` or with special functions (see below). You can get value of option with `getOption("option.name")`.

- `expss.digits` Number of digits after decimal separator which will be shown for tables. This parameter is supported in the as.datatable_widget, htmlTable.etable and `print` methods. `NULL` is default and means one digit. `NA` means no rounding. There is a convenience function for this option: `expss_digits`.
- `expss.enable_value_labels_support` By default, all labelled variables will use labels as labels for factor levels when factor is called. So, any function which calls factor/as.factor will use value labels. In details this option changes behavior of two methods for class `labelled` - `as.character` and `unique` - on which `factor` depends entirely. If you have compatibility problems set this option to zero: `options(expss.enable_value_labels_support = 0)`. Additionally there is an option for extreme value labels support: `options(expss.enable_value_labels_support = 2)`. With this value `factor`/`as.factor` will take into account empty levels. See example. It is recommended to turn off this option immediately after usage because `unique.labelled` will give weird result. Labels without values will be added to unique values. There are shortcuts for these options: `expss_enable_value_labels_support()`, `expss_enable_value_labels_support_extreme` and `expss_disable_value_labels_support()`.
- `expss.output` By default tables are printed in the console. You can change this behavior by setting this option. There are five possible values: `'rnotebook'`, `'viewer'`, `'commented'`, `'raw'` or `'huxtable'`. First option is useful when you run your code in the R Notebook - output will be rendered to nice HTML. The second option will render tables to RStudio viewer. `knitr` is supported automatically via `knit_print` method. `'commented'` prints default output to the console with comment symbol (#) at the beginning of the each line. With comment symbol you can easily copy and paste your output into the script. Option `raw` disables any formatting and all tables are printed as data.frames. Option `huxtable` print output via the huxtable library. Shortcuts for options: `expss_output_default()`, `expss_output_raw()`, `expss_output_viewer()`, `expss_output_commented()`, `expss_output_rnotebook()` and `expss_output_huxtable()`.
- `expss_fix_encoding_on`/`expss_fix_encoding_off` If you expreience problems with character encoding in RStudio Viewer/RNotebooks under Windows try `expss_fix_encoding_on()`. In some cases, it can help.

Usage

```
expss_digits(digits = NULL)

get_expss_digits()

expss_enable_value_labels_support()
```

```
expss_enable_value_labels_support_extreme()

expss_disable_value_labels_support()

expss_output_default()

expss_output_commented()

expss_output_raw()

expss_output_viewer()

expss_output_rnotebook()

expss_output_huxtable(...)

expss_fix_encoding_on()

expss_fix_encoding_off()

expss_fre_stat_lab(
  label = c("Count", "Valid percent", "Percent", "Responses, %",
    "Cumulative responses, %")
)
```

Arguments

digits	integer. Number of digits after decimal point. NULL is default and means 1 digit. NA means no rounding.
...	list of parameters for huxtable::set_default_properties. See set_default_properties.
label	character vector of length 5. Default labels for fre.

Examples

```
# example of different levels of value labels support
my_scale = c(1, 2, 2, 2)
# note that we have label 'Hard to say' for which there are no values in 'my_scale'
val_lab(my_scale) = num_lab("
                            1 Yes
                            2 No
                            3 Hard to say
                            ")
# disable labels support
expss_disable_value_labels_support()
table(my_scale) # there is no labels in the result
unique(my_scale)
# default value labels support
expss_enable_value_labels_support()
# table with labels but there are no label "Hard to say"
```

```
table(my_scale)
unique(my_scale)
# extreme value labels support
expss_enable_value_labels_support_extreme()
# now we see "Hard to say" with zero counts
table(my_scale)
# weird 'unique'! There is a value 3 which is absent in 'my_scale'
unique(my_scale)
# return immediately to defaults to avoid issues
expss_enable_value_labels_support()
```

`fctr` *Convert labelled variable to factor*

Description

`fctr` converts variable to factor. It force labels usage as factor labels for labelled variables even if 'expss.enable_value_labels_support' set to 0. For other types of variables base factor is called. Factor levels are constructed as values labels. If label doesn't exist for particular value then this value remain as is - so there is no information lost. This levels look like as "Variable_labelValue label" if argument `prepend` set to TRUE.

Usage

```
fctr(x, ..., drop_unused_labels = FALSE, prepend_var_lab = TRUE)
```

Arguments

`x`	a vector of data with labels.
`...`	optional arguments for `factor`
`drop_unused_labels`	logical. Should we drop unused value labels? Default is FALSE.
`prepend_var_lab`	logical. Should we prepend variable label before value labels? Default is TRUE.

Value

an object of class factor. For details see base factor documentation.

See Also

values2labels, names2labels, val_lab, var_lab. Materials for base functions: `factor`, `as.factor`, `ordered`, `as.ordered`

Examples

```
data(mtcars)

var_lab(mtcars$am) = "Transmission"
val_lab(mtcars$am) = c(automatic = 0, manual=1)

summary(lm(mpg ~ am, data = mtcars)) # no labels
summary(lm(mpg ~ fctr(am), data = mtcars)) # with labels
summary(lm(mpg ~ fctr(unvr(am)), data = mtcars)) # without variable label
```

`fre` *Simple frequencies with support of labels, weights and multiple response variables.*

Description

`fre` returns data.frame with six columns: labels or values, counts, valid percent (excluding NA), percent (with NA), percent of responses(for single-column `x` it equals to valid percent) and cumulative percent of responses.

Usage

```
fre(
  x,
  weight = NULL,
  drop_unused_labels = TRUE,
  prepend_var_lab = FALSE,
 stat_lab = getOption("expss.fre_stat_lab", c("Count", "Valid percent", "Percent",
    "Responses, %", "Cumulative responses, %"))
)
```

Arguments

`x`	vector/data.frame/list. data.frames are considered as multiple response variables. If `x` is list then vertically stacked frequencies for each element of list will be generated,
`weight`	numeric vector. Optional case weights. NA's and negative weights treated as zero weights.
`drop_unused_labels`	logical. Should we drop unused value labels? Default is TRUE.
`prepend_var_lab`	logical. Should we prepend variable label before value labels? By default we will add variable labels to value labels only if `x` or predictor is list (several variables).
`stat_lab`	character. Labels for the frequency columns.

Value

object of class 'etable'. Basically it's a data.frame but class is needed for custom methods.

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
    var_lab(vs) = "Engine"
    val_lab(vs) = c("V-engine" = 0,
                    "Straight engine" = 1)
    var_lab(am) = "Transmission"
    val_lab(am) = c(automatic = 0,
                    manual=1)
})

fre(mtcars$vs)

# stacked frequencies
fre(list(mtcars$vs, mtcars$am))

# multiple-choice variable
# brands - multiple response question
# Which brands do you use during last three months?
set.seed(123)
brands = data.frame(t(replicate(20,sample(c(1:5,NA),4,replace = FALSE))))
# score - evaluation of tested product
score = sample(-1:1,20,replace = TRUE)
var_lab(brands) = "Used brands"
val_lab(brands) = make_labels("
                              1 Brand A
                              2 Brand B
                              3 Brand C
                              4 Brand D
                              5 Brand E
                              ")

var_lab(score) = "Evaluation of tested brand"
val_lab(score) = make_labels("
                            -1 Dislike it
                            0 So-so
                            1 Like it
                            ")

fre(brands)

# stacked frequencies
fre(list(score, brands))
```

`htmlTable.etable` *Outputting HTML tables in RStudio viewer/R Notebooks*

Description

This is method for rendering results of fre/cro/tables in Shiny/RMarkdown/Jupyter notebooks and etc. For detailed description of function and its arguments see htmlTable. You can pack your tables in the list and render them all simultaneously. See examples. You may be interested in `expss_output_viewer()` for automatical rendering tables in the RStudio viewer or `expss_output_rnotebook()` for rendering in the R notebooks. See expss.options. `repr_html` is method for rendering table in the Jupyter notebooks and `knit_print` is method for rendering table in the `knitr` HTML-documents. Jupyter notebooks and `knitr` documents are supported automatically but in the R notebooks it is needed to set output to notebook via `expss_output_rnotebook()`.

Usage

```
## S3 method for class 'etable'
htmlTable(
  x,
  header = NULL,
  rnames = NULL,
  rowlabel = NULL,
  caption = NULL,
  tfoot = NULL,
  label = NULL,
  rgroup = NULL,
  n.rgroup = NULL,
  cgroup = NULL,
  n.cgroup = NULL,
  tspanner = NULL,
  n.tspanner = NULL,
  total = NULL,
  ctable = TRUE,
  compatibility = getOption("htmlTableCompat", "LibreOffice"),
  cspan.rgroup = "all",
  escape.html = FALSE,
  ...,
  digits = get_expss_digits(),
  row_groups = TRUE
)

## S3 method for class 'with_caption'
htmlTable(
  x,
  header = NULL,
  rnames = NULL,
  rowlabel = NULL,
```

```
  caption = NULL,
  tfoot = NULL,
  label = NULL,
  rgroup = NULL,
  n.rgroup = NULL,
  cgroup = NULL,
  n.cgroup = NULL,
  tspanner = NULL,
  n.tspanner = NULL,
  total = NULL,
  ctable = TRUE,
  compatibility = getOption("htmlTableCompat", "LibreOffice"),
  cspan.rgroup = "all",
  escape.html = FALSE,
  ...,
  digits = get_expss_digits(),
  row_groups = TRUE
)

## S3 method for class 'list'
htmlTable(
  x,
  header = NULL,
  rnames = NULL,
  rowlabel = NULL,
  caption = NULL,
  tfoot = NULL,
  label = NULL,
  rgroup = NULL,
  n.rgroup = NULL,
  cgroup = NULL,
  n.cgroup = NULL,
  tspanner = NULL,
  n.tspanner = NULL,
  total = NULL,
  ctable = TRUE,
  compatibility = getOption("htmlTableCompat", "LibreOffice"),
  cspan.rgroup = "all",
  escape.html = FALSE,
  ...,
  digits = get_expss_digits(),
  row_groups = TRUE,
  gap = "<br>"
)

knit_print.etable(x, ..., digits = get_expss_digits(), escape.html = FALSE)

knit_print.with_caption(
```

```
  x,
  ...,
  digits = get_expss_digits(),
  escape.html = FALSE
)

repr_html.etable(obj, ..., digits = get_expss_digits(), escape.html = FALSE)

repr_html.with_caption(
  obj,
  ...,
  digits = get_expss_digits(),
  escape.html = FALSE
)

repr_text.etable(obj, ..., digits = get_expss_digits())

repr_text.with_caption(obj, ..., digits = get_expss_digits())
```

Arguments

x	a data object of class 'etable' - result of fre/cro and etc.
header	Ignored.
rnames	Ignored.
rowlabel	Ignored.
caption	See manual for htmlTable.
tfoot	See manual for htmlTable.
label	See manual for htmlTable.
rgroup	Ignored.
n.rgroup	Ignored.
cgroup	Ignored.
n.cgroup	Ignored.
tspanner	See manual for htmlTable.
n.tspanner	See manual for htmlTable.
total	See manual for htmlTable.
ctable	See manual for htmlTable.
compatibility	See manual for htmlTable.
cspan.rgroup	See manual for htmlTable.
escape.html	logical: should HTML characters be escaped? Defaults to FALSE.
...	further parameters for htmlTable.
digits	integer By default, all numeric columns are rounded to one digit after decimal separator. Also you can set this argument by setting option 'expss.digits' - for example, `expss_digits(2)`. If it is NA than all numeric columns remain unrounded.

row_groups	logical Should we create row groups? TRUE by default.
gap	character Separator between tables if we output list of tables. By default it is line break ' '.
obj	a data object of class 'etable' - result of fre/cro and etc.

Value

Returns a string of class htmlTable

Examples

```
## Not run:
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

expss_output_viewer()
mtcars %>%
     tab_cols(total(), am %nest% vs) %>%
     tab_cells(mpg, hp) %>%
     tab_stat_mean() %>%
     tab_cells(cyl) %>%
     tab_stat_cpct() %>%
     tab_pivot() %>%
     set_caption("Table 1. Some variables from mtcars dataset.")

# several tables in a list
mtcars %>%
    calc(list(
      cro_cpct(list(am, vs, cyl), list(total(), am)) %>% set_caption("Table 1. Percent."),
     cro_mean_sd_n(list(mpg, hp, qsec), list(total(), am)) %>% set_caption("Table 2. Means.")
    )) %>%
    htmlTable()

expss_output_default()
```

```
## End(Not run)
```

`if_na` *Replace values with NA and vice-versa*

Description

- `if_na` replaces NA values in vector/data.frame/matrix/list with supplied value. For single value argument label can be provided with `label` argument. If replacement value is vector then `if_na` uses for replacement values from appropriate positions. An opposite operation is `na_if`.
- `na_if` replaces values with NA in vector/data.frame/matrix/list. Another alias for this is `mis_val`.
- `valid` returns logical vector which indicate the presence of at least one not-NA value in row. For vector or single column data.frame result is the same as with complete.cases. There is a special case for data.frame of class `dichotomy`. In this case result indicate the presence of at least one 1 in a row.

Usage

```
if_na(x, value, label = NULL)

if_na(x, label = NULL) <- value

x %if_na% value

na_if(x, value, with_labels = FALSE)

na_if(x, with_labels = FALSE) <- value

x %na_if% value

mis_val(x, value, with_labels = FALSE)

mis_val(x, with_labels = FALSE) <- value



valid(x)
```

Arguments

`x`	vector/matrix/data.frame/list
`value`	single value, vector of the same length as number of rows in `x`, or function (criteria) for `na_if`. See recode for details.
`label`	a character of length 1. Label for `value` which replace NA.
`with_labels`	logical. FALSE by default. Should we also remove labels of values which we recode to NA?

Format

An object of class `character` of length 1.

Value

object of the same form and class as `x`. `valid` returns logical vector.

Examples

```
# simple case
a = c(NA, 2, 3, 4, NA)
if_na(a, 99)

# the same result
a %if_na% 99

# with label
a = c(NA, 2, 3, 4, NA)
if_na(a, 99, label = "Hard to say")

# in-place replacement. The same result:
if_na(a, label = "Hard to say") = 99
a # c(99, 2, 3, 4, 99)

# replacement with values from other variable
a = c(NA, 2, 3, 4, NA)
b = 1:5
if_na(a, b)

# replacement with group means
# make data.frame
set.seed(123)
group = sample(1:3, 30, replace = TRUE)
param = runif(30)
param[sample(30, 10)] = NA # place 10 NA's
df = data.frame(group, param)

# replace NA's with group means
if_na(df$param) = window_fun(df$param, df$group, mean_col)
df

######################
### na_if examples ###
######################

a = c(1:5, 99)
# 99 to NA
na_if(a, 99)    # c(1:5, NA)

a %na_if% 99    # same result

# values which greater than 4 to NA
```

```
na_if(a, gt(4)) # c(1:4, NA, NA)

# alias 'mis_val', with_labels = TRUE
a = c(1, 1, 2, 2, 99)
val_lab(a) = c(Yes = 1, No = 2, "Hard to say" = 99)
mis_val(a, 99, with_labels = TRUE)

set.seed(123)
dfs = data.frame(
      a = c("bad value", "bad value", "good value", "good value", "good value"),
      b = runif(5)
)

# rows with 'bad value' will be filled with NA
# logical argument and recycling by columns
na_if(dfs, dfs$a=="bad value")

a = rnorm(50)
# values greater than 1 or less than -1 will be set to NA
# special functions usage
na_if(a, lt(-1) | gt(1))

# values inside [-1, 1] to NA
na_if(a, -1 %thru% 1)
```

`info` *Provides variables description for dataset*

Description

`info` returns data.frame with variables description and some summary statistics. Resulting data.frame mainly intended to keep in front of eyes in RStudio viewer or to be saved as csv to view in the spreadsheet software as reference about working dataset.

Usage

```
info(x, stats = TRUE, frequencies = TRUE, max_levels = 10)
```

Arguments

`x`	vector/factor/list/data.frame.
`stats`	Logical. Should we calculate summary for each variable?
`frequencies`	Logical. Should we calculate frequencies for each variable? This calculation can take significant amount of time for large datasets.
`max_levels`	Numeric. Maximum levels for using in frequency calculations. Levels above this value will convert to 'Other values'.

Value

data.frame with following columns: Name, Class, Length, NotNA, NA, Distincts, Label, ValueLabels, Min., 1st Qu., Median, Mean, 3rd Qu., Max., Frequency.

Examples

```
data(mtcars)
var_lab(mtcars$am) = "Transmission"
val_lab(mtcars$am) = c("Automatic"=0, "Manual"=1)
info(mtcars, max_levels = 5)
```

---

keep *Keep or drop elements by name/criteria in data.frame/matrix*

Description

keep selects variables/elements from data.frame by their names or by criteria (see criteria). except drops variables/elements from data.frame by their names or by criteria. Names at the top-level can be unquoted (non-standard evaluation). For standard evaluation of parameters you can surround them by round brackets. See examples. Methods for list will apply keep/except to each element of the list separately. .keep/.except are versions which works with default_dataset.

Usage

```
keep(data, ...)

.keep(...)

except(data, ...)

.except(...)
```

Arguments

data	data.frame/matrix/list
...	column names of type character/numeric or criteria/logical functions

Value

object of the same type as data

Examples

```
data(iris)
keep(iris, Sepal.Length, Sepal.Width)
except(iris, Species)

keep(iris, Species, other()) # move 'Species' to the first position
keep(iris, to("Petal.Width")) # keep all columns up to 'Species'

except(iris, perl("^Petal")) # remove columns which names start with 'Petal'

except(iris, 5) # remove fifth column

data(mtcars)
keep(mtcars, from("mpg") & to("qsec")) # keep columns from 'mpg' to 'qsec'
keep(mtcars, mpg %to% qsec) # the same result

 # standard and non-standard evaluation
 many_vars = c("am", "vs", "cyl")
 ## Not run:
 keep(mtcars, many_vars) # error - names not found: 'many_vars'

## End(Not run)
 keep(mtcars, (many_vars)) # ok

# character expansion
dfs = data.frame(
     a = 10 %r% 5,
     b_1 = 11 %r% 5,
     b_2 = 12 %r% 5,
     b_3 = 12 %r% 5,
     b_4 = 14 %r% 5,
     b_5 = 15 %r% 5
 )
 i = 1:5
 keep(dfs, b_1 %to% b_5)
 keep(dfs, text_expand("b_{i}")) # the same result
```

`match_row` *Match finds value in rows or columns/index returns value by index from rows or columns*

Description

`match` finds value in rows or columns. `index` returns value by index from row or column. One can use functions as criteria for `match`. In this case position of first value on which function equals to TRUE will be returned. For convenience there are special predefined functions - see criteria. If value is not found then NA will be returned.

Usage

```
match_row(criterion, ...)

match_col(criterion, ...)

index_row(index, ...)

index_col(index, ...)

value_row_if(criterion, ...)

value_col_if(criterion, ...)
```

Arguments

criterion	Vector of values to be matched, or function.
...	data. Vectors, matrixes, data.frames, lists. Shorter arguments will be recycled.
index	vector of positions in rows/columns from which values should be returned.

Value

vector with length equals to number of rows for *_row and equals to number of columns for *_col.

Examples

```
# toy data
v1 = 1:3
v2 = 2:4
v3 = 7:5

# postions of 1,3,5 in rows
match_row(c(1, 3, 5), v1, v2, v3) # 1:3
# postions of 1,3,5 in columns
match_col(1, v1, v2, v3) # c(v1 = 1, v2 = NA, v3 = NA)

# postion of first value greater than 2
ix = match_row(gt(2), v1, v2, v3)
ix # c(3,2,1)
# return values by result of previous 'match_row'
index_row(ix, v1, v2, v3) # c(7,3,3)

# the same actions with data.frame
dfs = data.frame(v1, v2, v3)

# postions of 1,3,5 in rows
match_row(c(1, 3, 5), dfs) # 1:3
# postions of 1,3,5 in columns
match_col(1, dfs) # c(v1 = 1, v2 = NA, v3 = NA)

# postion of first value greater than 2
```

```
ix = match_row(gt(2), dfs)
ix # c(3,2,1)
# return values by result of previous 'match_row'
index_row(ix, dfs) # c(7,3,3)
```

`merge.etable` *Merge two tables/data.frames*

Description

`%merge%` is infix shortcut for base merge with `all.x = TRUE` and `all.y = FALSE` (left join). There is also special method for combining results of `cro_*` and `fre`. For them `all = TRUE` (full join). It allows make complex tables from simple ones. See examples. Strange result is possible if one or two arguments have duplicates in first column (column with labels).

Usage

```
x %merge% y
```

Arguments

x	data.frame or results of `fre`/`cro_*`/`table_*`
y	data.frame or results of `fre`/`cro_*`/`table_*`

Value

data.frame

See Also

fre, cro, cro, cro_fun, merge

Examples

```
data(mtcars)
# apply labels
mtcars = apply_labels(mtcars,
                mpg = "Miles/(US) gallon",
                cyl = "Number of cylinders",
                disp = "Displacement (cu.in.)",
                hp = "Gross horsepower",
                drat = "Rear axle ratio",
                wt = "Weight (lb/1000)",
                qsec = "1/4 mile time",
                vs = "V/S",
                vs = c("V-engine" = 0, "Straight engine" = 1),
                am = "Transmission (0 = automatic, 1 = manual)",
                am = c(automatic = 0, manual = 1),
                gear = "Number of forward gears",
```

```
                carb = "Number of carburetors"
)

# table by 'am'
tab1 = calculate(mtcars, cro_cpct(gear, am))
# table with percents
tab2 = calculate(mtcars, cro_cpct(gear, vs))

# combine tables
tab1 %merge% tab2

# complex tables
# table with counts
counts = calculate(mtcars, cro(list(vs, am, gear, carb), list("Count")))
# table with percents
percents = calculate(mtcars, cro_cpct(list(vs, am, gear, carb), list("Column, %")))

# combine tables
counts %merge% percents
```

`mrset` *Create multiple response set/multiple dichotomy set from variables*

Description

These functions are intended for usage with tables - tables, cro, cro_fun. Result of `mrset` is considered as muliple-response set with category encoding and result of `mdset` is considered as multiple response set with dichotomy (dummy) encoding e. g. with 0 or 1 in the each column. Each column in the `dichotomy` is indicator of absence or presence of particular feature. Both functions don't convert its arguments to anything - it is supposed that arguments already have appropriate encoding. For conversation see as.dichotomy or as.category.

- `mrset_f` and `mdset_f` select variables by fixed pattern. Fixed pattern can be unquoted. For details see ..f.
- `mrset_p` and `mdset_p` select variables for multiple-responses by perl-style regular expression. For details see ..p.
- `mrset_t` and `mdset_t` select variables by expanding text arguments. For details see ..t and text_expand.

Usage

```
mrset(..., label = NULL)

mdset(..., label = NULL)

mrset_f(..., label = NULL)

mdset_f(..., label = NULL)
```

```
mrset_p(..., label = NULL)

mdset_p(..., label = NULL)

mrset_t(..., label = NULL)

mdset_t(..., label = NULL)
```

Arguments

...	variables
label	character optional label for multiple response set

Value

data.frame of class category/dichotomy

See Also

as.dichotomy, as.category

Examples

```
data(product_test)

calc_cro_cpct(product_test, mrset(a1_1 %to% a1_6))

# same result
calc_cro_cpct(product_test, mrset_f(a1_))

# same result
calc_cro_cpct(product_test, mrset_p("a1_"))

# same result
calc_cro_cpct(product_test, mrset_t("a1_{1:6}"))
```

---

names2labels *Replace data.frame/list names with corresponding variables labels.*

Description

names2labels replaces data.frame/list names with corresponding variables labels. If there are no labels for some variables their names remain unchanged. n2l is just shortcut for names2labels.

Usage

```
names2labels(x, exclude = NULL, keep_names = FALSE)

n2l(x, exclude = NULL, keep_names = FALSE)
```

Arguments

x	data.frame/list.
exclude	logical/integer/character columns which names should be left unchanged. Only applicable to list/data.frame.
keep_names	logical. If TRUE original column names will be kept with labels. Only applicable to list/data.frame.

Value

Object of the same type as x but with variable labels instead of names.

See Also

values2labels, val_lab, var_lab

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "V/S"
                var_lab(am) = "Transmission (0 = automatic, 1 = manual)"
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

# without original names
# note: we exclude dependent variable 'mpg' from conversion to use its short name in formula
summary(lm(mpg ~ ., data = names2labels(mtcars, exclude = "mpg")))
# with names
summary(lm(mpg ~ ., data = names2labels(mtcars, exclude = "mpg", keep_names = TRUE)))
```

---

name_dots *Bug workaround*

Description

Function is added to workaround strange bug with data.table (issue #10).

Usage

```
name_dots(...)
```

Arguments

...	arguments

Value

list

---

nest *Compute nested variable(-s) from several variables*

Description

`nest` mainly intended for usage with table functions such as cro. See examples. `%nest%` is infix version of this function. You can apply `nest` on multiple-response variables/list of variables and data.frames.

Usage

```
nest(...)

x %nest% y
```

Arguments

...	vectors/data.frames/lists
x	vector/data.frame/list
y	vector/data.frame/list

Value

vector/data.frame/list

See Also

See also interaction

Examples

```
data(mtcars)

mtcars = apply_labels(mtcars,
                      cyl = "Number of cylinders",
                      vs = "Engine",
                      vs = num_lab("
                             0 V-engine
                             1 Straight engine
                             "),
                      am = "Transmission",
```

```
                        am = num_lab("
                               0 Automatic
                               1 Manual
                               "),
                        carb = "Number of carburetors"
)

calc(mtcars, cro(cyl, am %nest% vs))

# list of variables
calc(mtcars, cro(cyl, am %nest% list(vs, cyl)))

# list of variables - multiple banners/multiple nesting
calc(mtcars, cro(cyl, list(total(), list(am, vs) %nest% cyl)))

# three variables
calc(mtcars, cro(am %nest% vs %nest% carb, cyl))

# the same with usual version
calc(mtcars, cro(cyl, nest(am, vs)))

# three variables
calc(mtcars, cro(nest(am, vs, carb), cyl))
```

`net` *Add subtotal to a set of categories*

Description

'subtotal' adds subtotal to set of categories, 'net' replaces categories with their net value. If you provide named arguments then name will be used as label for subtotal. In other case labels will be automatically generated taking into account arguments 'new_label' and 'prefix'. Note that if you provide overlapping categories then net and subtotals will also be overlapping. 'subtotal' and 'net' are intended for usage with cro and friends. 'tab_subtotal_*' and 'tab_net_*' are intended for usage with custom tables - see tables. There are auxiliary functions 'hide' and 'unhide'. 'hide' is used with 'subtotal' when you need to leave only subtotal for some specific items. And 'unhide' is used with 'net' when you want to show items for some nets. See examples.

Usage

```
net(
  x,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last"),
  add = FALSE
)
```

```
subtotal(
  x,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last"),
  add = TRUE
)

tab_net_cells(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)

tab_net_cols(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)

tab_net_rows(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)

tab_subtotal_cells(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)

tab_subtotal_cols(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)
```

```
tab_subtotal_rows(
  data,
  ...,
  position = c("below", "above", "top", "bottom"),
  prefix = "TOTAL ",
  new_label = c("all", "range", "first", "last")
)

hide(category)

unhide(category)
```

Arguments

x	variable, list, data.frame or multiple response set
...	list of categories for grouping. It can be numeric vectors (for example, 1:2), ranges (for example, 4 greater(5)). If an argument is named then this name will be used as label for subtotal.
position	position of the subtotal or net relative to original categories. "below" by default. One of the "below", "above", "top", "bottom". "top" and "bottom" place nets and subtotals above or below all other categories. For nets "below" and "above" have no difference because original categories are removed.
prefix	character, "TOTAL " by default. It is a prefix to automatically created labels for nets and subtotals.
new_label	how we will combine original values for automatically generated subtotal labels. Possible values are "all", "range", "first", "last". "all" collapse all labels, "range" take only first and last label,
add	logical. Should we add subtotal to categories or replace categories with a net?
data	intermediate table. See tables.
category	category (numeric vectors, ranges, criteria) which you want to 'hide' or 'unhide'.

Value

multiple response set or list of the multiple response sets

Examples

```
ol = c(1:7, 99)
var_lab(ol) = "Liking"
val_lab(ol)  = num_lab("
                    1 Disgusting
                    2 Very Poor
                    3 Poor
                    4 So-so
                    5 Good
                    6 Very good
```

```
                    7 Excellent
                    99 Hard to say
                    ")

cro(subtotal(ol, BOTTOM = 1:3, TOP = 6:7, position = "top"))
# example with hide
cro(subtotal(ol, TOP1 = hide(7), TOP2 = hide(6:7), TOP3 = 5:7, BOTTOM = 1:3, position = "top"))
# autolabelling
cro(subtotal(ol, 1:3, 6:7))
# replace original codes and another way of autolabelling
cro(net(ol, 1:3, 6:7, new_label = "range", prefix = "NET "))
# unhide
cro(net(ol, 1:3, unhide(6:7), new_label = "range", prefix = "NET "))
# character variable and criteria usage
items = c("apple", "banana", "potato", "orange", "onion", "tomato", "pineapple")
cro(
    subtotal(items,
             "TOTAL FRUITS"     = like("*ap*") | like("*an*"),
             "TOTAL VEGETABLES" = like("*to*") | like("*on*"),
             position = "bottom")
)

# 'tab_net_*' usage
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      gear = c(
                          One = 1,
                          Two = 2,
                          Three = 3,
                          Four = 4,
                          Five = 5
                      )
)
mtcars %>%
    tab_cells(mpg) %>%
   tab_net_cells("Low mpg" = less(mean(mpg)), "High mpg" = greater_or_equal(mean(mpg))) %>%
    tab_cols(total(), am) %>%
    tab_stat_cases() %>%
    tab_pivot()

mtcars %>%
    tab_cells(mpg) %>%
    tab_rows(gear) %>%
    tab_subtotal_rows(1:2, 3:4, "5 and more" = greater(4)) %>%
    tab_stat_mean() %>%
    tab_pivot()
```

prepend_values *Prepend values/variable names to value/variable labels*

Description

These functions add values/variable names as prefixes to value/variable labels. Functions which start with tab_ intended for usage inside table creation sequences. See examples and tables. It is recommended to use tab_prepend_* at the start of sequence of tables creation. If you use it in the middle of the sequence then previous statements will not be affected.

Usage

```
prepend_values(x)

prepend_names(x)

prepend_all(x)

tab_prepend_values(data)

tab_prepend_names(data)

tab_prepend_all(data)
```

Arguments

x	vector/data.frame. prepend_names can be applied only to data.frames.
data	data.frame/intermediate result of tables construction. See tables.

Value

original object with prepended names/values to labels

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (lb/1000)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
```

```
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

# prepend names and 'cro_cpct'
mtcars %>%
       prepend_names %>%
       calculate(
              cro_cpct(list(cyl, gear), list(total(), vs, am))
          )

# prepend values to value labels
mtcars %>%
   tab_prepend_values %>%
   tab_cols(total(), vs, am) %>%
   tab_cells(cyl, gear) %>%
   tab_stat_cpct() %>%
   tab_pivot()

# prepend names and labels
mtcars %>%
   tab_prepend_all %>%
   tab_cols(total(), vs, am) %>%
   tab_cells(cyl, gear) %>%
   tab_stat_cpct() %>%
   tab_pivot()

# variable in rows without prefixes
mtcars %>%
   tab_cells(cyl, gear) %>%
   tab_prepend_all %>%
   tab_cols(total(), vs, am) %>%
   tab_stat_cpct() %>%
   tab_pivot()
```

`product_test` *Data from product test of chocolate confectionary*

Description

It is truncated dataset with data from product test of two samples of chocolate sweets. 150 respondents tested two kinds of sweets (codenames: VSX123 and SDF546). Sample was divided into two groups (cells) of 75 respondents in each group. In cell 1 product VSX123 was presented first and then SDF546. In cell 2 sweets were presented in reversed order. Questions about respondent impressions about first product are in the block A (and about second tested product in the block B). At the end of the questionnaire there is a question about preferences between sweets.

Usage

```
product_test
```

Format

A data frame with 150 rows and 18 variables:

id Respondent Id.

cell First tested product (cell number).

s2a Age.

a1_1 What did you like in these sweets? Multiple response. First tested product.

a1_2 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_3 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_4 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_5 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_6 (continue) What did you like in these sweets? Multiple response. First tested product.

a22 Overall liking. First tested product.

b1_1 What did you like in these sweets? Multiple response. Second tested product.

b1_2 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_3 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_4 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_5 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_6 (continue) What did you like in these sweets? Multiple response. Second tested product.

b22 Overall liking. Second tested product.

c1 Preferences.

---

prop *Compute proportions from numeric vector/matrix/data.frame*

Description

`prop` returns proportion to sum of entire `x`. `prop_col` returns proportion to sum of each column of `x`. `prop_row` returns proportion to sum of each row of `x`. Non-numeric columns in the data.frame are ignored. NA's are also ignored.

Usage

```
prop(x)

prop_col(x)

prop_row(x)
```

Arguments

x	numeric vector/matrix/data.frame

Value

the same structure as x but with proportions of original values from sum of original values.

Examples

```
a = c(25, 25, NA)
prop(a)

# data.frame with non-numeric columns
fac = factor(c("a", "b", "c"))
char = c("a", "b", "c")
dat = as.POSIXct("2016-09-27")
a = sheet(fac, a = c(25, 25, NA), b = c(100, NA, 50), char, dat)

prop(a)
prop_row(a)
prop_col(a)

# the same as result as with 'prop.table'
tbl = table(state.division, state.region)

prop(tbl)
prop_row(tbl)
prop_col(tbl)
```

---

qc *Create vector of characters from unquoted strings (variable names)*

Description

- qc It is often needed to address variables in the data.frame in the such manner: `dfs[ ,c("var1","var2","var3")]`. qc ("quoted c") is a shortcut for the such cases to reduce keystrokes. With qc you can write: `dfs[ ,qc(var1,var2,var3)]`.
- qe returns list of expression. It is useful to create substitution list for `..$arg`.
- `text_expand` is simple string interpolation function. It searches in its arguments expressions in curly brackets `{expr}`, evaluate them and substitute with the result of evaluation. See examples.

Usage

```
qc(...)

qe(...)

text_expand(..., delim = c("\\{", "\\}"))
```

Arguments

...	characters in text_expand, unquoted names of variables in qc or unquoted expressions in qe.
delim	character vector of length 2 - pair of opening and closing delimiters for the templating tags. By default it is curly brackets. Note that delim will be used in the perl-style regular expression so you need to escape special characters, e. g. use "\\{" instead of "\{".

Value

Vector of characters

Examples

```
## qc
qc(a, b, c)
identical(qc(a, b, c), c("a", "b", "c"))

mtcars[, qc(am, mpg, gear)]

## text_expand
i = 1:5
text_expand("q{i}")

i = 1:3
j = 1:3
text_expand("q1_{i}_{j}")

data(iris)
text_expand("'iris' has {nrow(iris)} rows.")

## qe
qe(mrset(a1 %to% a6), mrset(b1 %to% b6), mrset(c1 %to% c6))
```

---

read_spss *Read an SPSS Data File*

Description

read_spss reads data from a file stored in SPSS *.sav format. It returns data.frame and never converts string variables to factors. Also it prepares SPSS values/variables labels for working with val_lab/var_lab functions. User-missings values are ignored. read_spss is simple wrapper around read.spss function from package foreign.

Usage

```
read_spss(file, reencode = TRUE)

read_spss_to_list(file, reencode = TRUE)
```

Arguments

file	Character string: the name of the file or URL to read.
reencode	logical: should character strings be re-encoded to the current locale. The default is TRUE. NA means to do so in a UTF-8 locale, only. Alternatively, a character string specifying an encoding to assume for the file.

Value

read_spss returns data.frame.

read_spss_to_list returns list of variables from SPSS files.

See Also

read.spss in package foreign, val_lab, var_lab

Examples

```
## Not run:

w = read_spss("project_123.sav") # to data.frame
list_w = read_spss_to_list("project_123.sav") # to list


## End(Not run)
```

---

recode *Change, rearrange or consolidate the values of an existing or new variable. Inspired by the RECODE command from SPSS.*

Description

recode change, rearrange or consolidate the values of an existing variable based on conditions. Design of this function inspired by RECODE from SPSS. Sequence of recodings provided in the form of formulas. For example, 1:2 ~ 1 means that all 1's and 2's will be replaced with 1. Each value will be recoded only once. In the assignment form recode(...) = ... of this function values which doesn't meet any condition remain unchanged. In case of the usual form ... = recode(...) values which doesn't meet any condition will be replaced with NA. One can use values or more sophisticated logical conditions and functions as a condition. There are several special functions for usage as criteria - for details see criteria. Simple common usage looks like: recode(x,1:2 ~ -1,3 ~ 0,1:2 ~ 1,99 ~ NA). For more information, see details and examples. The ifs function checks whether one or more conditions are met and returns a value that corresponds to the first TRUE condition. ifs can take the place of multiple nested ifelse statements and is much easier to read with multiple conditions. ifs works in the same manner as recode - e. g. with formulas. But conditions should be only logical and it doesn't operate on multicolumn objects.

Usage

```
recode(
  x,
  ...,
  with_labels = FALSE,
  new_label = c("all", "range", "first", "last")
)

rec(x, ..., with_labels = TRUE, new_label = c("all", "range", "first", "last"))

recode(x, with_labels = FALSE, new_label = c("all", "range", "first", "last")) <- value

rec(x, with_labels = TRUE, new_label = c("all", "range", "first", "last")) <- value

ifs(...)

lo

hi

copy(x)

from_to(from, to)

values %into% names
```

Arguments

x	vector/matrix/data.frame/list
...	sequence of formulas which describe recodings. They are used when from/to arguments are not provided.
with_labels	logical. FALSE by default for 'recode' and TRUE for 'rec'. Should we also recode value labels with the same recodings as variable?
new_label	one of "all", "range", "first", or "last". If we recode value labels ('with_labels = TRUE') how we will combine labels for duplicated values? "all" will use all labels, "range" will use first and last labels. See examples.
value	list with formulas which describe recodings in assignment form of function/to list if from/to notation is used.
from	list of conditions for values which should be recoded (in the same format as LHS of formulas).
to	list of values into which old values should be recoded (in the same format as RHS of formulas).
values	object(-s) which will be assigned to names for %into% operation. %into% supports multivalue assignments. See examples.
names	name(-s) which will be given to values expression. For %into%.

Format

An object of class `numeric` of length 1.

An object of class `numeric` of length 1.

Details

Input conditions - possible values for left-hand side (LHS) of formula or element of `from` list:

- vector/single value All values in `x` which equal to elements of the vector in LHS will be replaced with RHS.
- function Values for which function gives TRUE will be replaced with RHS. There are some special functions for the convenience - see criteria.
- single logical value `TRUE` It means all other unrecoded values (ELSE in SPSS RECODE). All other unrecoded values will be changed to RHS of the formula or appropriate element of `to`.

Output values - possible values for right-hand side (RHS) of formula or element of `to` list:

- value replace elements of `x`. This value will be recycled across rows and columns of `x`.
- vector values of this vector will replace values in the corresponding position in rows of `x`. Vector will be recycled across columns of `x`.
- function This function will be applied to values of `x` which satisfy recoding condition. There is a special auxiliary function `copy` which just returns its argument. So, in the `recode` it just copies old value (COPY in SPSS RECODE). See examples.

`%into%` tries to mimic SPSS 'INTO'. Values from left-hand side will be assigned to right-hand side. You can use `%to%` expression in the RHS of `%into%`. See examples. `lo` and `hi` are shortcuts for `-Inf` and `Inf`. They can be useful in expressions with `%thru%`, e. g. `1 %thru% hi`.

Value

object of the same form as `x` with recoded values

Examples

```
# examples from SPSS manual
# RECODE V1 TO V3 (0=1) (1=0) (2, 3=-1) (9=9) (ELSE=SYSMIS)
v1  = c(0, 1, 2, 3, 9, 10)
recode(v1) = c(0 ~ 1, 1 ~ 0, 2:3 ~ -1, 9 ~ 9, TRUE ~ NA)
v1

# RECODE QVAR(1 THRU 5=1)(6 THRU 10=2)(11 THRU HI=3)(ELSE=0).
qvar = c(1:20, 97, NA, NA)
recode(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, 11 %thru% hi ~ 3, TRUE ~ 0)
# the same result
recode(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, ge(11) ~ 3, TRUE ~ 0)

# RECODE STRNGVAR ('A', 'B', 'C'='A')('D', 'E', 'F'='B')(ELSE=' ').
strngvar = LETTERS
recode(strngvar, c('A', 'B', 'C') ~ 'A', c('D', 'E', 'F') ~ 'B', TRUE ~ ' ')
```

```
# recode in place. Note that we recode only first six letters
recode(strngvar) = c(c('A', 'B', 'C') ~ 'A', c('D', 'E', 'F') ~ 'B')
strngvar

# RECODE AGE (MISSING=9) (18 THRU HI=1) (0 THRU 18=0) INTO VOTER.
age = c(NA, 2:40, NA)
voter = recode(age, NA ~ 9, 18 %thru% hi ~ 1, 0 %thru% 18 ~ 0)
voter
# the same result with '%into%'
recode(age, NA ~ 9, 18 %thru% hi ~ 1, 0 %thru% 18 ~ 0) %into% voter2
voter2
# recode with adding labels
voter = recode(age, "Refuse to answer" = NA ~ 9,
                    "Vote" = 18 %thru% hi ~ 1,
                    "Don't vote" = 0 %thru% 18 ~ 0)
voter

# recoding with labels
ol = c(1:7, 99)
var_lab(ol) = "Liking"
val_lab(ol)  = num_lab("
                    1 Disgusting
                    2 Very Poor
                    3 Poor
                    4 So-so
                    5 Good
                    6 Very good
                    7 Excellent
                    99 Hard to say
                    ")

recode(ol, 1:3 ~ 1, 5:7 ~ 7, TRUE ~ copy, with_labels = TRUE)
# 'rec' is a shortcut for recoding with labels. Same result:
rec(ol, 1:3 ~ 1, 5:7 ~ 7, TRUE ~ copy)
# another method of combining labels
recode(ol, 1:3 ~ 1, 5:7 ~ 7, TRUE ~ copy, with_labels = TRUE, new_label = "range")
# example with from/to notation
# RECODE QVAR(1 THRU 5=1)(6 THRU 10=2)(11 THRU HI=3)(ELSE=0).
list_from = list(1 %thru% 5, 6 %thru% 10, ge(11), TRUE)
list_to = list(1, 2, 3, 0)
recode(qvar, from_to(list_from, list_to))


list_from = list(NA, 18 %thru% hi, 0 %thru% 18)
list_to = list("Refuse to answer" = 9, "Vote" = 1, "Don't vote" = 0)
voter = recode(age, from_to(list_from, list_to))
voter

# 'ifs' examples
a = 1:5
b = 5:1
ifs(b>3 ~ 1)                      # c(1, 1, NA, NA, NA)
ifs(b>3 ~ 1, TRUE ~ 3)            # c(1, 1, 3, 3, 3)
```

```
ifs(b>3 ~ 1, a>4 ~ 7, TRUE ~ 3)    # c(1, 1, 3, 3, 7)
ifs(b>3 ~ a, TRUE ~ 42)            # c(1, 2, 42, 42, 42)

# advanced usage
#' # multiple assignment with '%into%'
set.seed(123)
x1 = runif(30)
x2 = runif(30)
x3 = runif(30)
# note nessesary brackets around RHS of '%into%'
recode(x1 %to% x3, gt(0.5) ~ 1, other ~ 0) %into% (x_rec_1 %to% x_rec_3)
fre(x_rec_1)
# the same operation with characters expansion
i = 1:3
recode(x1 %to% x3, gt(0.5) ~ 1, other ~ 0) %into% text_expand('x_rec2_{i}')
fre(x_rec2_1)

# factor recoding
a = factor(letters[1:4])
recode(a, "a" ~ "z", TRUE ~ copy) # we get factor

# example with function in RHS
data(iris)
new_iris = recode(iris, is.numeric ~ scale, other ~ copy)
str(new_iris)

set.seed(123)
a = rnorm(20)
# if a<(-0.5) we change it to absolute value of a (abs function)
recode(a, lt(-0.5) ~ abs, other ~ copy)

# the same example with logical criteria
recode(a, when(a<(-.5)) ~ abs, other ~ copy)
```

`ref` *Auxiliary functions to pass arguments to function by reference*

Description

These two functions aimed to simplify build functions with side-effects (e. g. for modifying variables in place). Of cause it is not the R way of doing things but sometimes it can save several keystrokes.

Usage

```
ref(x)

ref(x) <- value
```

Arguments

x	Reference to variable, it is formula, ~var_name.
value	Value that should be assigned to modified variable.

Details

To create reference to variable one can use formula: b = ~a. b is reference to a. So ref(b) returns value of a and ref(b) = new_val will modify a. If argument x of these functions is not formula then these functions have no effect e. g. ref(a) is identical to a and after ref(a) = value a is identical to value. It is not possible to use function as argument x in assignment form. For example, ref(some_function(x)) = some_value will rise error. Use y = some_function(x); ref(y) = some_value instead.

Value

ref returns value of referenced variable. ref<- modifies referenced variable.

Examples

```
# Simple example
a = 1:3
b = ~a  # b is reference to 'a'
identical(ref(b),a) # TRUE

ref(b)[2] = 4 # here we modify 'a'
identical(a, c(1,4,3)) # TRUE

# usage inside function

# top 10 rows
head10 = function(x){
 ds = head(ref(x), 10)
 ref(x) = ds
 invisible(ds) # for usage without references
}

data(iris)
ref_to_iris = ~iris
head10(ref_to_iris) # side-effect
nrow(iris) # 10

# argument is not formula - no side-effect
data(mtcars)
mtcars10 = head10(mtcars)

nrow(mtcars10) # 10
nrow(mtcars) # 32
```

`set_caption` *Add caption to the table*

Description

To drop caption use `set_caption` with `caption = NULL`. Captions are supported by htmlTable.etable, xl_write and as.datatable_widget functions.

Usage

```
set_caption(obj, caption)

get_caption(obj)

is.with_caption(obj)
```

Arguments

`obj`	object of class `etable` - result of `cro_cpct` and etc.
`caption`	character caption for the table.

Value

object of class `with_caption`.

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      vs = "Engine",
                      vs = num_lab("
                             0 V-engine
                             1 Straight engine
                             "),
                      am = "Transmission",
                      am = num_lab("
                             0 Automatic
                             1 Manual
                             ")
)
tbl_with_caption = calc_cro(mtcars, am, vs) %>%
    set_caption("Table 1. Type of transimission.")

tbl_with_caption
```

sheet *Make data.frame without conversion to factors and without fixing names*

Description

sheet and as.sheet are shortcuts to data.frame and as.data.frame with stringsAsFactors = FALSE, check.names = FALSE. .sheet is the same as above but works in the scope of default dataset.

Usage

```
sheet(...)

as.sheet(x, ...)

.sheet(...)
```

Arguments

...	objects, possibly named
x	object to be coerced to data.frame

Value

data.frame/list

See Also

default_dataset, data.frame, as.data.frame

Examples

```
# see the difference
df1 = data.frame(a = letters[1:3], "This is my long name" = 1:3)
df2 = sheet(a = letters[1:3], "This is my long name" = 1:3)

str(df1)
str(df2)


data(iris)
default_dataset(iris)

.sheet(Sepal.Width,  Sepal.Length)
```

sort_asc *Sort data.frames/matrices/vectors*

Description

`sort_asc` sorts in ascending order and `sort_desc` sorts in descending order. `.sort_asc`/`.sort_desc` are versions for working with default_dataset.

Usage

```
sort_asc(data, ..., na.last = FALSE)

.sort_asc(..., na.last = FALSE)

sort_desc(data, ..., na.last = TRUE)

.sort_desc(..., na.last = TRUE)
```

Arguments

`data`	data.frame/matrix/vector
`...`	character/numeric or criteria/logical functions (see criteria). Column names/numbers for data.frame/matrix by which object will be sorted. Names at the top-level can be unquoted (non-standard evaluation). For standard evaluation of parameters you can surround them by round brackets. See examples. Ignored for vectors.
`na.last`	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.

Value

sorted `data`

Examples

```
data(mtcars)
sort_asc(mtcars, mpg)
sort_asc(mtcars, cyl, mpg) # by two column

# same results with column nums
sort_asc(mtcars, 1)
sort_asc(mtcars, 2:1) # by two column
sort_asc(mtcars, 2, 1) # by two column

# call with parameter
sorting_columns = c("cyl", "mpg")
sort_asc(mtcars, (sorting_columns))
```

split_by *Splits data.frame into list of data.frames that can be analyzed separately*

Description

Splits data.frame into list of data.frames that can be analyzed separately. These data.frames are sets of cases that have the same values for the specified split variables. Any missing values in split variables are dropped together with the corresponding values of `data`. `split_off` works with lists of data.frames or objects that can be coerced to data.frame and assumed to have compatible structure. Resulting rows will be sorted in order of the split variables.

Usage

```
split_by(data, ..., drop = TRUE)

split_off(data, groups = NULL, rownames = NULL)
```

Arguments

`data`	data.frame for `split_by`/list for `split_off`
`...`	unquoted variables names (see keep) by which `data` will be split into list.
`drop`	should we drop combination of levels with zero observation? TRUE by default.
`groups`	character If it is not `NULL` then we add list names as variable to result of `split_off` with the name specified by `groups`. If it is `TRUE` then name will be `.groups`.
`rownames`	character If it is not `NULL` then we add data.frames rownames as variable to result of `split_off` with the name specified by `rownames`. If it is `TRUE` then name will be `.rownames`.

Value

`split_by` returns list of data.frames/`split_off` returns data.frame

See Also

split, compute, calculate, do_repeat, where

Examples

```
# example from base R 'split'
data(airquality)
airquality2 = airquality %>%
    split_by(Month) %>%
    compute({
        Ozone_zscore = scale(Ozone)
    }) %>%
    split_off()
```

```
head(airquality2)

# usage of 'groups', 'rownames'
data(mtcars)
# add labels to dataset
mtcars %>%
    apply_labels(mpg = "Miles/(US) gallon",
                 disp = "Displacement (cu.in.)",
                 wt = "Weight",
                 hp = "Gross horsepower",
                 vs = "Engine",
                 vs = num_lab("
                                  0 V-engine
                                  1 Straight engine
                                  "),

                 am = "Transmission",
                 am = num_lab("
                                  0 Automatic
                                  1 Manual
                                  ")
    ) %>%
    split_by(am, vs) %>%
    use_labels({
        res = lm(mpg ~ hp + disp + wt)
        cbind(Coef. = coef(res), confint(res))
    }) %>%
    split_off(groups = TRUE, rownames = "variable")
```

`split_labels` *Split character vector to matrix/split columns in data.frame*

Description

`split_labels`/`split_columns` are auxiliary functions for post-processing tables resulted from cro/cro_fun and etc. In these tables all labels collapsed in the first column with "|" separator. `split_columns` split first column into multiple columns with separator (`split` argument). `split_table_to_df` split first column of table and column names. Result of this operation is data.frame with character columns.

Usage

```
split_labels(
  x,
  remove_repeated = TRUE,
  split = "\\|",
  fixed = FALSE,
  perl = FALSE
```

```
)

split_columns(
  data,
  columns = 1,
  remove_repeated = TRUE,
  split = "\\|",
  fixed = FALSE,
  perl = FALSE
)

split_table_to_df(
  data,
  digits = get_expss_digits(),
  remove_repeated = TRUE,
  split = "\\|",
  fixed = FALSE,
  perl = FALSE
)

make_subheadings(data, number_of_columns = 1)
```

Arguments

x	character vector which will be split
remove_repeated	logical. Default is TRUE. Should we remove repeated labels?
split	character vector (or object which can be coerced to such) containing regular expression(s) (unless fixed = TRUE) to use for splitting.
fixed	logical. If TRUE match split exactly, otherwise use regular expressions. Has priority over perl.
perl	logical. Should Perl-compatible regexps be used?
data	data.frame vector which will be split
columns	character/numeric/logical columns in the data.frame data which should be split
digits	numeric. How many digits after decimal point should be left in split_table_to_df?
number_of_columns	integer. Number of columns from row labels which will be used as subheadings in table.

Value

split_labels returns character matrix, split_columns returns data.frame with columns replaced by possibly multiple columns with split labels. split_table_to_df returns data.frame with character columns.

See Also

strsplit

Examples

```
data(mtcars)

# apply labels
mtcars = apply_labels(mtcars,
    cyl = "Number of cylinders",
    vs = "Engine",
    vs = c("V-engine" = 0,
                   "Straight engine" = 1),
    am = "Transmission",
    am = c(automatic = 0,
                  manual=1),
    gear = "Number of forward gears",
    carb = "Number of carburetors"
)

# all row labels in the first column
tabl = mtcars %>%
       calculate(cro_cpct(list(cyl, gear, carb), list(total(), vs, am)))

tabl # without subheadings

make_subheadings(tabl) # with subheadings

split_labels(tabl[[1]])
split_labels(colnames(tabl))

# replace first column with new columns
split_columns(tabl) # remove repeated

split_columns(tabl, remove_repeated = FALSE)

split_columns(tabl)

split_table_to_df(tabl)

split_table_to_df(tabl)
```

`sum_row` *Compute sum/mean/sd/median/max/min/custom function on rows/columns*

Description

These functions are intended for usage inside compute, and do_if. sum/mean/sd/median/max/min by default omits NA. `any_in_*` checks existence of any TRUE in each row/column. It is equivalent of any applied to each row/column. `all_in_*` is equivalent of all applied to each row/column.

Usage

```
sum_row(..., na.rm = TRUE)

sum_col(..., na.rm = TRUE)

mean_row(..., na.rm = TRUE)

mean_col(..., na.rm = TRUE)

sd_row(..., na.rm = TRUE)

sd_col(..., na.rm = TRUE)

median_row(..., na.rm = TRUE)

median_col(..., na.rm = TRUE)

max_row(..., na.rm = TRUE)

max_col(..., na.rm = TRUE)

min_row(..., na.rm = TRUE)

min_col(..., na.rm = TRUE)

apply_row(fun, ...)

apply_col(fun, ...)

any_in_row(..., na.rm = TRUE)

any_in_col(..., na.rm = TRUE)

all_in_row(..., na.rm = TRUE)

all_in_col(..., na.rm = TRUE)
```

Arguments

`...`	data. Vectors, matrixes, data.frames, list. Shorter arguments will be recycled.
`na.rm`	logical. Contrary to the base 'sum' it is TRUE by default. Should missing values (including NaN) be removed?
`fun`	custom function that will be applied to ...

Value

All functions except `apply_*` return numeric vector of length equals the number of argument columns/rows. Value of `apply_*` depends on supplied `fun` function.

See Also

compute, do_if, %to%, count_if, sum_if, mean_if, median_if, sd_if, min_if, max_if

Examples

```
iris = compute(iris, {
  new_median = median_row(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
  new_mean = mean_row(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
  })

dfs = data.frame(
    test = 1:5,
    aa = rep(10, 5),
    b_ = rep(20, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)

# calculate sum of b* variables
compute(dfs, {
    b_total = sum_row(b_, b_1 %to% b_5)
})

# conditional modification
do_if(dfs, test %in% 2:4, {
    b_total = sum_row(b_, b_1 %to% b_5)
})
```

---

tables *Functions for custom tables construction*

Description

Table construction consists of at least of three functions chained with `magrittr` pipe operator. At first we need to specify variables for which statistics will be computed with `tab_cells`. Secondary, we calculate statistics with one of `tab_stat_*` functions. And last, we finalize table creation with `tab_pivot`: `dataset %>% tab_cells(variable) %>% tab_stat_cases() %>% tab_pivot()`. After that we can optionally sort table with tab_sort_asc, drop empty rows/columns with drop_rc and transpose with `tab_transpose`. Generally, table is just a data.frame so we can use arbitrary operations on it. Statistic is always calculated with the last cell, column/row variables, weight, missing values and subgroup. To define new cell/column/row variables we can call appropriate function one more time. `tab_pivot` defines how we combine different statistics and where statistic labels will appear - inside/outside rows/columns. See examples. For significance testing see significance.

Usage

```
tab_cols(data, ...)

tab_cells(data, ...)

tab_rows(data, ...)

tab_weight(data, weight = NULL)

tab_mis_val(data, ...)

tab_total_label(data, ...)

tab_total_statistic(data, ...)

tab_total_row_position(data, total_row_position = c("below", "above", "none"))

tab_subgroup(data, subgroup = NULL)

tab_row_label(data, ..., label = NULL)

tab_stat_fun(data, ..., label = NULL, unsafe = FALSE)

tab_stat_mean_sd_n(
  data,
  weighted_valid_n = FALSE,
 labels = c("Mean", "Std. dev.", ifelse(weighted_valid_n, "Valid N", "Unw. valid N")),
  label = NULL
)

tab_stat_mean(data, label = "Mean")

tab_stat_median(data, label = "Median")

tab_stat_se(data, label = "S. E.")

tab_stat_sum(data, label = "Sum")

tab_stat_min(data, label = "Min.")

tab_stat_max(data, label = "Max.")

tab_stat_sd(data, label = "Std. dev.")

tab_stat_valid_n(data, label = "Valid N")

tab_stat_unweighted_valid_n(data, label = "Unw. valid N")
```

```
tab_stat_fun_df(data, ..., label = NULL, unsafe = FALSE)

tab_stat_cases(
  data,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none"),
  label = NULL
)

tab_stat_cpct(
  data,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none"),
  label = NULL
)

tab_stat_cpct_responses(
  data,
  total_label = NULL,
  total_statistic = "u_responses",
  total_row_position = c("below", "above", "none"),
  label = NULL
)

tab_stat_tpct(
  data,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none"),
  label = NULL
)

tab_stat_rpct(
  data,
  total_label = NULL,
  total_statistic = "u_cases",
  total_row_position = c("below", "above", "none"),
  label = NULL
)

tab_last_vstack(
  data,
  stat_position = c("outside_rows", "inside_rows"),
  stat_label = c("inside", "outside"),
  label = NULL
)
```

```
tab_last_hstack(
  data,
  stat_position = c("outside_columns", "inside_columns"),
  stat_label = c("inside", "outside"),
  label = NULL
)

tab_pivot(
  data,
 stat_position = c("outside_rows", "inside_rows", "outside_columns", "inside_columns"),
  stat_label = c("inside", "outside")
)

tab_transpose(data)

tab_caption(data, ...)
```

Arguments

data	data.frame/intermediate_table
...	vector/data.frame/list. Variables for tables. Use mrset/mdset for multiple-response variables.
weight	numeric vector in tab_weight. Cases with NA's, negative and zero weights are removed before calculations.
total_row_position	Position of total row in the resulting table. Can be one of "below", "above", "none".
subgroup	logical vector in tab_subgroup. You can specify subgroup on which table will be computed.
label	character. Label for the statistic in the tab_stat_*.
unsafe	logical If TRUE than fun will be evaluated as is. It can lead to significant increase in the performance. But there are some limitations. For tab_stat_fun it means that your function fun should return vector of length one. Also there will be no attempts to make labels for statistic. For tab_stat_fun_df your function should return vector of length one or list/data.frame (optionally with 'row_labels' element - statistic labels). If unsafe is TRUE then further arguments (. . .) for fun will be ignored.
weighted_valid_n	logical. Sould we show weighted valid N in tab_stat_mean_sd_n? By default it is FALSE.
labels	character vector of length 3. Labels for mean, standard deviation and valid N in tab_stat_mean_sd_n.
total_label	By default "#Total". You can provide several names - each name for each total statistics.

`total_statistic`
: By default it is "u_cases" (unweighted cases). Possible values are "u_cases", "u_responses", "u_cpct", "u_rpct", "u_tpct", "w_cases", "w_responses", "w_cpct", "w_rpct", "w_tpct". "u_" means unweighted statistics and "w_" means weighted statistics.

`stat_position`
: character one of the values `"outside_rows"`, `"inside_rows"`, `"outside_columns"` or `"inside_columns"`. It defines how we will combine statistics in the table.

`stat_label`
: character one of the values `"inside"` or `"outside"`. Where will be placed labels for the statistics relative to column names/row labels? See examples.

Details

- `tab_cells` variables on which percentage/cases/summary functions will be computed. Use mrset/mdset for multiple-response variables.
- `tab_cols` optional variables which breaks table by columns. Use mrset/mdset for multiple-response variables.
- `tab_rows` optional variables which breaks table by rows. Use mrset/mdset for multiple-response variables.
- `tab_weight` optional weight for the statistic.
- `tab_mis_val` optional missing values for the statistic. It will be applied on variables specified by `tab_cells`. It works in the same manner as na_if.
- `tab_subgroup` optional logical vector/expression which specify subset of data for table.
- `tab_row_label` Add to table empty row with specified row labels. It is usefull for making section headings and etc.
- `tab_total_row_position` Default value for `total_row_position` argument in `tab_stat_cases` and etc. Can be one of "below", "above", "none".
- `tab_total_label` Default value for `total_label` argument in `tab_stat_cases` and etc. You can provide several names - each name for each total statistics.
- `tab_total_statistic` Default value for `total_statistic` argument in `tab_stat_cases` and etc. You can provide several values. Possible values are "u_cases", "u_responses", "u_cpct", "u_rpct", "u_tpct", "w_cases", "w_responses", "w_cpct", "w_rpct", "w_tpct". "u_" means unweighted statistics and "w_" means weighted statistics.
- `tab_stat_fun`, `tab_stat_fun_df` `tab_stat_fun` applies function on each variable in cells separately, `tab_stat_fun_df` gives to function each data.frame in cells as a whole data.table with all names converted to variable labels (if labels exists). So it is not recommended to rely on original variables names in your `fun`. For details see cro_fun. You can provide several functions as arguments. They will be combined as with combine_functions. So you can use `method` argument. For details see documentation for combine_functions.
- `tab_stat_cases` calculate counts.
- `tab_stat_cpct`, `tab_stat_cpct_responses` calculate column percent. These functions give different results only for multiple response variables. For `tab_stat_cpct` base of percent is number of valid cases. Case is considered as valid if it has at least one non-NA value. So for multiple response variables sum of percent may be greater than 100. For `tab_stat_cpct_responses` base of percent is number of valid responses. Multiple response variables can have several responses for single case. Sum of percent of `tab_stat_cpct_responses` always equals to 100%.

- tab_stat_rpct calculate row percent. Base for percent is number of valid cases.
- tab_stat_tpct calculate table percent. Base for percent is number of valid cases.
- tab_stat_mean, tab_stat_median, tab_stat_se, tab_stat_sum, tab_stat_min, tab_stat_max, tab_stat_sd, tab_stat_valid_n, tab_stat_unweighted_valid_n different summary statistics. NA's are always omitted.
- tab_pivot finalize table creation and define how different tab_stat_* will be combined
- tab_caption set caption on the table. Should be used after the tab_pivot.
- tab_transpose transpose final table after tab_pivot or last statistic.

Value

All of these functions return object of class intermediate_table except tab_pivot which returns final result - object of class etable. Basically it's a data.frame but class is needed for custom methods.

See Also

fre, cro, cro_fun, tab_sort_asc, drop_empty_rows, significance.

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (1000 lbs)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)
# some examples from 'cro'
# simple example - generally with 'cro' it can be made with less typing
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(vs) %>%
    tab_stat_cpct() %>%
    tab_pivot()

# split rows
mtcars %>%
    tab_cells(cyl) %>%
```

```
    tab_cols(vs) %>%
    tab_rows(am) %>%
    tab_stat_cpct() %>%
    tab_pivot()

# multiple banners
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(total(), vs, am) %>%
    tab_stat_cpct() %>%
    tab_pivot()

# nested banners
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(total(), vs %nest% am) %>%
    tab_stat_cpct() %>%
    tab_pivot()

# summary statistics
mtcars %>%
    tab_cells(mpg, disp, hp, wt, qsec) %>%
    tab_cols(am) %>%
    tab_stat_fun(Mean = w_mean, "Std. dev." = w_sd, "Valid N" = w_n) %>%
    tab_pivot()

# summary statistics - labels in columns
mtcars %>%
    tab_cells(mpg, disp, hp, wt, qsec) %>%
    tab_cols(am) %>%
    tab_stat_fun(Mean = w_mean, "Std. dev." = w_sd, "Valid N" = w_n, method = list) %>%
    tab_pivot()

# subgroup with droping empty columns
mtcars %>%
    tab_subgroup(am == 0) %>%
    tab_cells(cyl) %>%
    tab_cols(total(), vs %nest% am) %>%
    tab_stat_cpct() %>%
    tab_pivot() %>%
    drop_empty_columns()

# total position at the top of the table
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(total(), vs) %>%
    tab_rows(am) %>%
    tab_stat_cpct(total_row_position = "above",
                  total_label = c("number of cases", "row %"),
                  total_statistic = c("u_cases", "u_rpct")) %>%
    tab_pivot()

# this example cannot be made easily with 'cro'
```

```
mtcars %>%
    tab_cells(am) %>%
    tab_cols(total(), vs) %>%
    tab_total_row_position("none") %>%
    tab_stat_cpct(label = "col %") %>%
    tab_stat_rpct(label = "row %") %>%
    tab_stat_tpct(label = "table %") %>%
    tab_pivot(stat_position = "inside_rows")

# statistic labels inside columns
mtcars %>%
    tab_cells(am) %>%
    tab_cols(total(), vs) %>%
    tab_total_row_position("none") %>%
    tab_stat_cpct(label = "col %") %>%
    tab_stat_rpct(label = "row %") %>%
    tab_stat_tpct(label = "table %") %>%
    tab_pivot(stat_position = "inside_columns")

# stacked statistics
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(total(), am) %>%
    tab_stat_mean() %>%
    tab_stat_se() %>%
    tab_stat_valid_n() %>%
    tab_stat_cpct() %>%
    tab_pivot()

# stacked statistics with section headings
mtcars %>%
    tab_cells(cyl) %>%
    tab_cols(total(), am) %>%
    tab_row_label("#Summary statistics") %>%
    tab_stat_mean() %>%
    tab_stat_se() %>%
    tab_stat_valid_n() %>%
    tab_row_label("#Column percent") %>%
    tab_stat_cpct() %>%
    tab_pivot()

# stacked statistics with different variables
mtcars %>%
    tab_cols(total(), am) %>%
    tab_cells(mpg, hp, qsec) %>%
    tab_stat_mean() %>%
    tab_cells(cyl, carb) %>%
    tab_stat_cpct() %>%
    tab_pivot()

# stacked statistics - label position outside row labels
mtcars %>%
    tab_cells(cyl) %>%
```

```
    tab_cols(total(), am) %>%
    tab_stat_mean() %>%
    tab_stat_se %>%
    tab_stat_valid_n() %>%
    tab_stat_cpct(label = "Col %") %>%
    tab_pivot(stat_label = "outside")

# example from 'cro_fun_df' - linear regression by groups with sorting
mtcars %>%
    tab_cells(sheet(mpg, disp, hp, wt, qsec)) %>%
    tab_cols(total(), am) %>%
    tab_stat_fun_df(
        function(x){
            frm = reformulate(".", response = as.name(names(x)[1]))
            model = lm(frm, data = x)
            sheet('Coef.' = coef(model),
                  confint(model)
            )
        }
    ) %>%
    tab_pivot() %>%
    tab_sort_desc()

# multiple-response variables and weight
data(product_test)
codeframe_likes = num_lab("
                          1 Liked everything
                          2 Disliked everything
                          3 Chocolate
                          4 Appearance
                          5 Taste
                          6 Stuffing
                          7 Nuts
                          8 Consistency
                          98 Other
                          99 Hard to answer
                          ")

set.seed(1)
product_test = compute(product_test, {
    # recode age by groups
    age_cat = recode(s2a, lo %thru% 25 ~ 1, lo %thru% hi ~ 2)

    var_lab(age_cat) = "Age"
    val_lab(age_cat) = c("18 - 25" = 1, "26 - 35" = 2)

    var_lab(a1_1) = "Likes. VSX123"
    var_lab(b1_1) = "Likes. SDF456"
    val_lab(a1_1) = codeframe_likes
    val_lab(b1_1) = codeframe_likes

    wgt = runif(.N, 0.25, 4)
    wgt = wgt/sum(wgt)*.N
```

```
})

product_test %>%
    tab_cells(mrset(a1_1 %to% a1_6), mrset(b1_1 %to% b1_6)) %>%
    tab_cols(total(), age_cat) %>%
    tab_weight(wgt) %>%
    tab_stat_cpct() %>%
    tab_sort_desc() %>%
    tab_pivot()

# trick to place cell variables labels inside columns
# useful to compare two variables
# '|' is needed to prevent automatic labels creation from argument
# alternatively we can use list(...) to avoid this
product_test %>%
    tab_cols(total(), age_cat) %>%
    tab_weight(wgt) %>%
    tab_cells("|" = unvr(mrset(a1_1 %to% a1_6))) %>%
    tab_stat_cpct(label = var_lab(a1_1)) %>%
    tab_cells("|" = unvr(mrset(b1_1 %to% b1_6))) %>%
    tab_stat_cpct(label = var_lab(b1_1)) %>%
    tab_pivot(stat_position = "inside_columns")

# if you need standard evaluation, use 'vars'
tables = mtcars %>%
      tab_cols(total(), am %nest% vs)

for(each in c("mpg", "disp", "hp", "qsec")){
    tables = tables %>% tab_cells(vars(each)) %>%
        tab_stat_fun(Mean = w_mean, "Std. dev." = w_sd, "Valid N" = w_n)
}
tables %>% tab_pivot()
```

`tab_significance_options` *Mark significant differences between columns in the table*

Description

- `significance_cpct` conducts z-tests between column percent in the result of cro_cpct. Results are calculated with the same formula as in prop.test without continuity correction.
- `significance_means` conducts t-tests between column means in the result of cro_mean_sd_n. Results are calculated with the same formula as in t.test.
- `significance_cases` conducts chi-squared tests on the subtable of table with counts in the result of cro_cases. Results are calculated with the same formula as in chisq.test.
- `significance_cell_chisq` compute cell chi-square test on table with column percent. The cell chi-square test looks at each table cell and tests whether it is significantly different from its expected value in the overall table. For example, if it is thought that variations in political

opinions might depend on the respondent's age, this test can be used to detect which cells contribute significantly to that dependence. Unlike the chi-square test (`significance_cases`), which is carried out on a whole set of rows and columns, the cell chi-square test is carried out independently on each table cell. Although the significance level of the cell chi-square test is accurate for any given cell, the cell tests cannot be used instead of the chi-square test carried out on the overall table. Their purpose is simply to point to the parts of the table where dependencies between row and column categories may exist.

For `significance_cpct` and `significance_means` there are three type of comparisons which can be conducted simultaneously (argument `compare_type`):

- `subtable` provide comparisons between all columns inside each subtable.
- `previous_column` is a comparison of each column of the subtable with the previous column. It is useful if columns are periods or survey waves.
- `first_column` provides comparison the table first column with all other columns in the table. `adjusted_first_column` is also comparison with the first column but with adjustment for common base. It is useful if the first column is total column and other columns are subgroups of this total. Adjustments are made according to algorithm in IBM SPSS Statistics Algorithms v20, p. 263. Note that with these adjustments t-tests between means are made with equal variance assumed (as with `var_equal = TRUE`).

By now there are no adjustments for multiple-response variables (results of mrset) in the table columns so significance tests are rather approximate for such cases. Also, there are functions for the significance testing in the sequence of custom tables calculations (see tables):

- `tab_last_sig_cpct`, `tab_last_sig_means` and `tab_last_sig_cpct` make the same tests as their analogs mentioned above. It is recommended to use them after appropriate statistic function: tab_stat_cpct, tab_stat_mean_sd_n and tab_stat_cases.
- `tab_significance_options` With this function we can set significance options for the entire custom table creation sequence.
- `tab_last_add_sig_labels` This function applies `add_sig_labels` to the last calculated table - it adds labels (letters by default) for significance to columns header. It may be useful if you want to combine a table with significance with table without it.
- `tab_last_round` This function rounds numeric columns in the last calculated table to specified number of digits. It is sometimes needed if you want to combine table with significance with table without it.

Usage

```
tab_significance_options(
  data,
  sig_level = 0.05,
  min_base = 2,
  delta_cpct = 0,
  delta_means = 0,
  correct = TRUE,
  compare_type = "subtable",
  bonferroni = FALSE,
  subtable_marks = "greater",
```

```
  inequality_sign = "both" %in% subtable_marks,
  sig_labels = LETTERS,
  sig_labels_previous_column = c("v", "^"),
  sig_labels_first_column = c("-", "+"),
  sig_labels_chisq = c("<", ">"),
  keep = c("percent", "cases", "means", "sd", "bases"),
  row_margin = c("auto", "sum_row", "first_column"),
  total_marker = "#",
  total_row = 1,
  digits = get_expss_digits(),
  na_as_zero = FALSE,
  var_equal = FALSE,
  mode = c("replace", "append")
)

tab_last_sig_cpct(
  data,
  sig_level = 0.05,
  delta_cpct = 0,
  min_base = 2,
  compare_type = "subtable",
  bonferroni = FALSE,
  subtable_marks = c("greater", "both", "less"),
  inequality_sign = "both" %in% subtable_marks,
  sig_labels = LETTERS,
  sig_labels_previous_column = c("v", "^"),
  sig_labels_first_column = c("-", "+"),
  keep = c("percent", "bases"),
  na_as_zero = FALSE,
  total_marker = "#",
  total_row = 1,
  digits = get_expss_digits(),
  mode = c("replace", "append"),
  label = NULL
)

tab_last_sig_means(
  data,
  sig_level = 0.05,
  delta_means = 0,
  min_base = 2,
  compare_type = "subtable",
  bonferroni = FALSE,
  subtable_marks = c("greater", "both", "less"),
  inequality_sign = "both" %in% subtable_marks,
  sig_labels = LETTERS,
  sig_labels_previous_column = c("v", "^"),
  sig_labels_first_column = c("-", "+"),
```

```
  keep = c("means", "sd", "bases"),
  var_equal = FALSE,
  digits = get_expss_digits(),
  mode = c("replace", "append"),
  label = NULL
)

tab_last_sig_cases(
  data,
  sig_level = 0.05,
  min_base = 2,
  correct = TRUE,
  keep = c("cases", "bases"),
  total_marker = "#",
  total_row = 1,
  digits = get_expss_digits(),
  mode = c("replace", "append"),
  label = NULL
)

tab_last_sig_cell_chisq(
  data,
  sig_level = 0.05,
  min_base = 2,
  subtable_marks = c("both", "greater", "less"),
  sig_labels_chisq = c("<", ">"),
  correct = TRUE,
  keep = c("percent", "bases", "none"),
  row_margin = c("auto", "sum_row", "first_column"),
  total_marker = "#",
  total_row = 1,
  total_column_marker = "#",
  digits = get_expss_digits(),
  mode = c("replace", "append"),
  label = NULL
)

tab_last_round(data, digits = get_expss_digits())

tab_last_add_sig_labels(data, sig_labels = LETTERS)

significance_cases(
  x,
  sig_level = 0.05,
  min_base = 2,
  correct = TRUE,
  keep = c("cases", "bases"),
  total_marker = "#",
```

```
  total_row = 1,
  digits = get_expss_digits()
)

significance_cell_chisq(
  x,
  sig_level = 0.05,
  min_base = 2,
  subtable_marks = c("both", "greater", "less"),
  sig_labels_chisq = c("<", ">"),
  correct = TRUE,
  keep = c("percent", "bases", "none"),
  row_margin = c("auto", "sum_row", "first_column"),
  total_marker = "#",
  total_row = 1,
  total_column_marker = "#",
  digits = get_expss_digits()
)

cell_chisq(cases_matrix, row_base, col_base, total_base, correct)

significance_cpct(
  x,
  sig_level = 0.05,
  delta_cpct = 0,
  min_base = 2,
  compare_type = "subtable",
  bonferroni = FALSE,
  subtable_marks = c("greater", "both", "less"),
  inequality_sign = "both" %in% subtable_marks,
  sig_labels = LETTERS,
  sig_labels_previous_column = c("v", "^"),
  sig_labels_first_column = c("-", "+"),
  keep = c("percent", "bases"),
  na_as_zero = FALSE,
  total_marker = "#",
  total_row = 1,
  digits = get_expss_digits()
)

add_sig_labels(x, sig_labels = LETTERS)

significance_means(
  x,
  sig_level = 0.05,
  delta_means = 0,
  min_base = 2,
  compare_type = "subtable",
```

```
  bonferroni = FALSE,
  subtable_marks = c("greater", "both", "less"),
  inequality_sign = "both" %in% subtable_marks,
  sig_labels = LETTERS,
  sig_labels_previous_column = c("v", "^"),
  sig_labels_first_column = c("-", "+"),
  keep = c("means", "sd", "bases"),
  var_equal = FALSE,
  digits = get_expss_digits()
)
```

Arguments

data	data.frame/intermediate_table for tab_* functions.
sig_level	numeric. Significance level - by default it equals to 0.05.
min_base	numeric. Significance test will be conducted if both columns have bases greater or equal to min_base. By default, it equals to 2.
delta_cpct	numeric. Minimal delta between percent for which we mark significant differences (in percent points) - by default it equals to zero. Note that, for example, for minimal 5 percent point difference delta_cpct should be equals 5, not 0.05.
delta_means	numeric. Minimal delta between means for which we mark significant differences - by default it equals to zero.
correct	logical indicating whether to apply continuity correction when computing the test statistic for 2 by 2 tables. Only for significance_cases and significance_cell_chisq. For details see chisq.test. TRUE by default.
compare_type	Type of compare between columns. By default, it is subtable - comparisons will be conducted between columns of each subtable. Other possible values are: first_column, adjusted_first_column and previous_column. We can conduct several tests simultaneously.
bonferroni	logical. FALSE by default. Should we use Bonferroni adjustment by the number of comparisons in each row?
subtable_marks	character. One of "greater", "both" or "less". By deafult we mark only values which are significantly greater than some other columns. For significance_cell_chisq default is "both".We can change this behavior by setting an argument to less or both.
inequality_sign	logical. FALSE if subtable_marks is "less" or "greater". Should we show > or < before significance marks of subtable comparisons.
sig_labels	character vector. Labels for marking differences between columns of subtable.
sig_labels_previous_column	a character vector with two elements. Labels for marking a difference with the previous column. First mark means 'lower' (by default it is v) and the second means greater (^).
sig_labels_first_column	a character vector with two elements. Labels for marking a difference with the first column of the table. First mark means 'lower' (by default it is -) and the second means 'greater' (+).

sig_labels_chisq
: a character vector with two labels for marking a difference with row margin of the table. First mark means 'lower' (by default it is <) and the second means 'greater' (>). Only for `significance_cell_chisq`.

keep
: character. One or more from "percent", "cases", "means", "bases", "sd" or "none". This argument determines which statistics will remain in the table after significance marking.

row_margin
: character. One of values "auto" (default), "sum_row", or "first_column". If it is "auto" we try to find total column in the subtable by `total_column_marker`. If the search is failed, we use the sum of each rows as row total. With "sum_row" option we always sum each row to get margin. Note that in this case result for multiple response variables in banners may be incorrect. With "first_column" option we use table first column as row margin for all subtables. In this case result for the subtables with incomplete bases may be incorrect. Only for `significance_cell_chisq`.

total_marker
: character. Total rows mark in the table. "#" by default.

total_row
: integer/character. In the case of the several totals per subtable it is a number or name of total row for the significance calculation.

digits
: an integer indicating how much digits after decimal separator will be shown in the final table.

na_as_zero
: logical. `FALSE` by default. Should we treat `NA`'s as zero cases?

var_equal
: a logical variable indicating whether to treat the two variances as being equal. For details see t.test.

mode
: character. One of `replace`(default) or `append`. In the first case the previous result in the sequence of table calculation will be replaced with result of significance testing. In the second case result of the significance testing will be appended to sequence of table calculation.

label
: character. Label for the statistic in the `tab_*`. Ignored if the `mode` is equals to `replace`.

total_column_marker
: character. Mark for total columns in the subtables. "#" by default.

x
: table (class `etable`): result of cro_cpct with proportions and bases for `significance_cpct`, result of cro_mean_sd_n with means, standard deviations and valid N for `significance_means`, and result of cro_cases with counts and bases for `significance_cases`.

cases_matrix
: numeric matrix with counts size R*C

row_base
: numeric vector with row bases, length R

col_base
: numeric vector with col bases, length C

total_base
: numeric single value, total base

Value

`tab_last_*` functions return objects of class `intermediate_table`. Use tab_pivot to get the final result - `etable` object. Other functions return `etable` object with significant differences.

See Also

cro_cpct, cro_cases, cro_mean_sd_n, tables, compare_proportions, compare_means, prop.test, t.test, chisq.test

Examples

```
data(mtcars)
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (lb/1000)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

mtcars_table = calculate(mtcars,
                    cro_cpct(list(cyl, gear),
                             list(total(), vs, am))
                         )

significance_cpct(mtcars_table)
## Not run:
# comparison with first column
significance_cpct(mtcars_table, compare_type = "first_column")

# comparison with first column and inside subtable
significance_cpct(mtcars_table,
            compare_type = c("first_column", "subtable"))

# only significance marks
significance_cpct(mtcars_table, keep = "none")

# means
mtcars_means = calculate(mtcars,
                    cro_mean_sd_n(list(mpg, wt, hp),
                                  list(total(), vs, cyl))
                        )

significance_means(mtcars_means)

# mark values which are less and greater
significance_means(mtcars_means, subtable_marks = "both")
```

```
# chi-squared test
mtcars_cases = calculate(mtcars,
                  cro_cases(list(cyl, gear),
                            list(total(), vs, am))
                        )

significance_cases(mtcars_cases)

# cell chi-squared test
# increase number of cases to avoid warning about chi-square approximation
mtcars2 = add_rows(mtcars, mtcars, mtcars)

tbl = calc_cro_cpct(mtcars2, gear, am)
significance_cell_chisq(tbl)

# table with multiple variables
tbl = calc_cro_cpct(mtcars2, list(gear, cyl), list(total(), am, vs))
significance_cell_chisq(tbl, sig_level = .0001)

# custom tables with significance
mtcars %>%
    tab_significance_options(subtable_marks = "both") %>%
    tab_cells(mpg, hp) %>%
    tab_cols(total(), vs, am) %>%
    tab_stat_mean_sd_n() %>%
    tab_last_sig_means(keep = "means") %>%
    tab_cells(cyl, gear) %>%
    tab_stat_cpct() %>%
    tab_last_sig_cpct() %>%
    tab_pivot()

# Overcomplicated examples - we move significance marks to
# separate columns. Columns with statistics remain numeric
mtcars %>%
    tab_significance_options(keep = "none",
                         sig_labels = NULL,
                         subtable_marks = "both",
                         mode = "append") %>%
    tab_cols(total(), vs, am) %>%
    tab_cells(mpg, hp) %>%
    tab_stat_mean_sd_n() %>%
    tab_last_sig_means() %>%
    tab_last_hstack("inside_columns") %>%
    tab_cells(cyl, gear) %>%
    tab_stat_cpct() %>%
    tab_last_sig_cpct() %>%
    tab_last_hstack("inside_columns") %>%
    tab_pivot(stat_position = "inside_rows") %>%
    drop_empty_columns()

## End(Not run)
```

tab_sort_asc *Partially (inside blocks) sort tables/data.frames*

Description

tab_sort_asc/tab_sort_desc sort tables (usually result of cro/tables) in ascending/descending order between specified rows (by default, it is rows which contain '#' in the first column).

Usage

```
tab_sort_asc(x, ..., excluded_rows = "#", na.last = FALSE)

tab_sort_desc(x, ..., excluded_rows = "#", na.last = TRUE)
```

Arguments

x	data.frame
...	character/numeric or criteria/logical functions (see criteria). Column names/numbers for data.frame/matrix by which object will be sorted. Names at the top-level can be unquoted (non-standard evaluation). For standard evaluation of parameters you can surround them by round brackets. See examples. If this argument is missing then table will be sorted by second column. Usually second column is the first column with numbers in the table (there are row labels in the first column).
excluded_rows	character/logical/numeric rows which won't be sorted. Rows of the table will be sorted between excluded rows. If it is characters then they will be considered as pattern/vector of patterns. Patterns will be matched with Perl-style regular expression with values in the first column of x (see grep, perl = TRUE argument). Rows which have such patterns will be excluded. By default, pattern is "#" because "#" marks totals in the result of cro.
na.last	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.

Value

sorted table('etable')/data.frame

Examples

```
data(mtcars)

# apply labels
mtcars = apply_labels(mtcars,
    cyl = "Number of cylinders",
    vs = "Engine",
    vs = c("V-engine" = 0,
                   "Straight engine" = 1),
```

```
    am = "Transmission",
    am = c(automatic = 0,
                   manual=1),
    gear = "Number of forward gears",
    carb = "Number of carburetors"
)

# without sorting
mtcars %>% calculate(cro_cpct(list(cyl, gear, carb), list("#total", vs, am)))

# with sorting
mtcars %>%
    calculate(cro_cpct(list(cyl, gear, carb), list("#total", vs, am))) %>%
    tab_sort_desc

# sort by parameter
sorting_column = "Engine|V-engine"

mtcars %>%
    calculate(cro_cpct(list(cyl, gear, carb), list("#total", vs, am))) %>%
    tab_sort_desc((sorting_column))
```

`text_to_columns` *Make data.frame from text*

Description

Convert delimited text lines to data.frame. Blank lines are always skipped, trailing whitespaces are trimmed. You can use comments with '#' inside your text. For details see read.table.

Usage

```
text_to_columns(
  text,
  header = TRUE,
  sep = "",
  quote = "",
  dec = ".",
  encoding = "unknown",
  ...
)

text_to_columns_csv(
  text,
  header = TRUE,
  sep = ",",
  quote = "",
  dec = ".",
```

```
  encoding = "unknown",
  ...
)

text_to_columns_csv2(
  text,
  header = TRUE,
  sep = ";",
  quote = "",
  dec = ",",
  encoding = "unknown",
  ...
)

text_to_columns_tab(
  text,
  header = TRUE,
  sep = "\t",
  quote = "",
  dec = ".",
  encoding = "unknown",
  ...
)

text_to_columns_tab2(
  text,
  header = TRUE,
  sep = "\t",
  quote = "",
  dec = ",",
  encoding = "unknown",
  ...
)
```

Arguments

`text`	character/vector of characters
`header`	a logical value indicating whether the `text` contains the names of the variables as its first line.
`sep`	the field separator character. Values on each line of the file are separated by this character. If sep = "" (the default for `text_to_columns`) the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.
`quote`	the set of quoting characters. To disable quoting altogether, use quote = "".
`dec`	the character used in the file for decimal points.
`encoding`	encoding to be assumed for input strings. It is used to mark character strings as known to be in Latin-1 or UTF-8 (see read.table).
`...`	further parameters which will be passed to read.table.

Value

data.frame

Examples

```
text_to_columns("
# simple data.frame
     a b   c
     1 2.5 a
     4 5.5 b
     7 8.5 c
")
```

---

unlab *Drop variable label and value labels*

Description

unlab returns variable x without variable labels and value labels

Usage

```
unlab(x)

drop_all_labels(x)
```

Arguments

x	Variable(s). Vector/data.frame/list.

Value

unlab returns original variable x without variable label, value labels and class.

See Also

unvr unvl

Examples

```
raw_var = rep(1:2,5)
var_with_lab = set_var_lab(raw_var,"Income")
val_lab(var_with_lab) = c("Low"=1,"High"=2)
identical(raw_var,unlab(var_with_lab)) # should be TRUE
```

`values2labels` *Replace vector/matrix/data.frame/list values with corresponding value labels.*

Description

`values2labels` replaces vector/matrix/data.frame/list values with corresponding value labels. If there are no labels for some values they are converted to characters in most cases. If there are no labels at all for variable it remains unchanged. `v2l` is just shortcut to `values2labels`.

Usage

```
values2labels(x)

v2l(x)
```

Arguments

x	vector/matrix/data.frame/list

Value

Object of the same form as x but with value labels instead of values.

See Also

names2labels, val_lab, var_lab

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = NULL
                val_lab(am) = c(" automatic" = 0, " manual" =  1)
})

summary(lm(mpg ~ ., data = values2labels(mtcars[,c("mpg","am")])))
```

`val_lab` *Set or get value labels*

Description

These functions set/get/drop value labels. Duplicated values are not allowed. If argument `x` is data.frame or list then labels applied to all elements of data.frame/list. To drop value labels, use `val_lab(var) <-NULL` or `unvl(var)`. `make_labels` converts text from the form that usually used in questionnaires to named vector. For variable labels see var_lab. For working with entire data.frame see apply_labels.

- `val_lab` returns value labels or NULL if labels doesn't exist.
- `val_lab<-` set value labels.
- `set_val_lab` returns variable with value labels.
- `add_val_lab<-` add value labels to already existing value labels.
- `unvl` drops value labels.
- `make_labels` makes named vector from text for usage as value labels.
- `num_lab`, `lab_num` and `autonum` are shortcuts for `make_labels` with `code_postion` 'left', 'right' and 'autonum' accordingly.

Usage

```
val_lab(x)

val_lab(x) <- value

set_val_lab(x, value, add = FALSE)

add_val_lab(x, value)

add_val_lab(x) <- value

unvl(x)

drop_val_labs(x)

make_labels(text, code_position = c("left", "right", "autonum"))

drop_unused_labels(x)

num_lab(text)

lab_num(text)

autonum(text)
```

Arguments

x	Variable(s). Vector/data.frame/list.
value	Named vector. Names of vector are labels for the appropriate values of variable x.
add	Logical. Should we add value labels to old labels or replace it? Deafult is FALSE - we completely replace old values. If TRUE new value labels will be combined with old value labels.
text	text that should be converted to named vector
code_position	Possible values "left", "right" - position of numeric code in text. "autonum" - makes codes by autonumbering lines of text.

Details

Value labels are stored in attribute "labels" (attr(x,"labels")). We set variable class to "labelled" for preserving labels from dropping during some operations (such as c and `[`).

Value

val_lab return value labels (named vector). If labels doesn't exist it return NULL . val_lab<- and set_val_lab return variable (vector x) of class "labelled" with attribute "labels" which contains value labels. make_labels return named vector for usage as value labels.

Examples

```
# toy example
set.seed(123)
# score - evaluation of tested product

score = sample(-1:1,20,replace = TRUE)
var_lab(score) = "Evaluation of tested brand"
val_lab(score) = c("Dislike it" = -1,
                   "So-so" = 0,
                   "Like it" = 1
                   )

# frequency of product scores
fre(score)

# brands - multiple response question
# Which brands do you use during last three months?

brands = as.sheet(t(replicate(20,sample(c(1:5,NA),4,replace = FALSE))))

var_lab(brands) = "Used brands"
val_lab(brands) = make_labels("
                              1 Brand A
                              2 Brand B
                              3 Brand C
                              4 Brand D
                              5 Brand E
```

```
                                ")

# percentage of used brands
fre(brands)

# percentage of brands within each score
cro_cpct(brands, score)

## make labels from text copied from questionnaire

age = c(1, 2, 1, 2)

val_lab(age) = num_lab("
 1. 18 - 26
 2. 27 - 35
")

# note support of value lables in base R
table(age)

# or, if in original codes is on the right side

products = 1:8

val_lab(products) = lab_num("
 Chocolate bars    1
 Chocolate sweets (bulk) 2
 Slab chocolate(packed) 3
 Slab chocolate (bulk) 4
 Boxed chocolate sweets 5
 Marshmallow/pastilles in chocolate coating 6
 Marmalade in chocolate coating 7
 Other 8
")

table(products)
```

`vars` *Get variables/range of variables by name/by pattern.*

Description

- `vars` returns data.frame with all variables by their names or by criteria (see criteria). There is no non-standard evaluation in this function by design so use quotes for names of your variables. This function is intended to get variables by parameter/criteria. The only exception with non-standard evaluation is `%to%`. You can use `%to%` inside `vars` or independently.
- `..p` returns data.frame with all variables which names satisfy supplied perl-style regular expression. Arguments for this function is quoted characters. It is a shortcut for `vars(perl(pattern))`.

- `..f` returns data.frame with all variables which names contain supplied pattern. Arguments for this function can be unquoted. It is a shortcut for `vars(fixed(pattern))`.
- `..t` returns data.frame with variables which names are stored in the supplied arguments. Expressions in characters in curly brackets are expanded. See text_expand.
- `..[]` returns data.frame with all variables by their names or by criteria (see criteria). Names at the top-level can be unquoted (non-standard evaluation). For standard evaluation of parameters you can surround them by round brackets. You can assign to this expression. If there are several names inside square brackets then each element of list/data.frame from right side will be assigned to appropriate name from left side. You can use `item1 %to% item2` notation to get/create sequence of variables. If there are no arguments inside square brackets than from each item of RHS will be created separate variable in the parent frame. In this case RHS should be named list or data.frame.
- `..$name` sets/returns object which name is stored in the variable `name`. It is convenient wrapper around get/assign functions.
- `%to%` returns range of variables between `e1` and `e2` (similar to SPSS 'to'). modify, modify_if, calculate, keep, except and where support `%to%`.
- `indirect/indirect_list` are aliases for `vars/vars_list`.

Functions with word 'list' in name return lists of variables instead of dataframes.

Usage

```
vars(...)

vars_list(...)

indirect(...)

indirect_list(...)

e1 %to% e2

e1 %to_list% e2

..

..f(...)

..p(...)

..t(...)
```

Arguments

`...`	characters names of variables or criteria/logical functions
`e1`	unquoted name of start variable (e. g. a_1)
`e2`	unquoted name of start variable (e. g. a_5)

Format

An object of class `parameter` of length 1.

Value

data.frame/list with variables

See Also

keep, except, do_repeat, compute, calculate, where

Examples

```
# In data.frame
dfs = data.frame(
    a = rep(1, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_3 = rep(13, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)

# calculate sum of b_* variables
compute(dfs, {
    b_total = sum_row(b_1 %to% b_5)
})

# identical result
compute(dfs, {
    b_total = sum_row(..f(b_))
})

compute(dfs, {
    b_total = sum_row(..t("b_{1:5}"))
})

# In global environement
a = rep(10, 5)
a1 = rep(1, 5)
a2 = rep(2, 5)
a3 = rep(3, 5)
a4 = rep(4, 5)
a5 = rep(5, 5)

# identical results
a1 %to% a5
vars(perl("^a[0-9]$"))
..[perl("^a[0-9]$")]
..p("^a[0-9]$")
..t("a{1:5}")
```

```
# sum each row
sum_row(a1 %to% a5)

# variable substitution
name1 = "a"
name2 = "new_var"

# in global environment
..$name1 # give as variable 'a'

..$name2 = ..$name1 * 2 # create variable 'new_var' which is equal to 'a' times 2
new_var

# inside data.frame
compute(dfs, {
     ..$name2 = ..$name1*2
})

compute(dfs, {
     for(name1 in paste0("b_", 1:5)){
         name2 = paste0("new_", name1)
         ..$name2 = ..$name1*2
     }
     rm(name1, name2) # we don't need this variables as columns in 'dfs'
})

# square brackets notation - multi-assignment
name1 = paste0("b_", 1:5)
compute(dfs, {
          # round brackets about 'name1' is needed to avoid using it 'as is'
         ..[paste0("new_", name1)] = ..[(name1)]*2
})

# the same result
# note the automatic creation of sequence of variables
compute(dfs, {
         ..[new_b_1 %to% new_b_5] = ..[b_1 %to% b_5]*2
})

# assignment form of 'recode' on multiple variables
compute(dfs, {
         recode(..[b_1 %to% b_5]) = 13 %thru% hi ~ 20
})

# empty brackets - unboxing of dichotomy.
compute(dfs, {
         ..[] =  as.dichotomy(b_1 %to% b_5, prefix = "v_")
})
```

---

var_lab *Set or get variable label*

Description

These functions set/get/drop variable labels. For value labels see val_lab. For working with entire data.frame see apply_labels.

- `var_lab` returns variable label or NULL if label doesn't exist.
- `var_lab<-` set variable label.
- `set_var_lab` returns variable with label.
- `unvr` drops variable label.
- `add_labelled_class` Add missing 'labelled' class. This function is needed when you load SPSS data with packages which in some cases don't set 'labelled' class for variables with labels. For example, `haven` package doesn't set 'labelled' class for variables which have variable label but don't have value labels. Note that to use 'expss' with 'haven' you need to load 'expss' strictly after 'haven' to avoid conflicts.

Usage

```
var_lab(x, default = NULL)

var_lab(x) <- value

set_var_lab(x, value)

unvr(x)

drop_var_labs(x)

add_labelled_class(
  x,
  remove_classes = c("haven_labelled", "spss_labelled", "haven_labelled_spss",
    "vctrs_vctr")
)
```

Arguments

`x`	Variable. In the most cases it is numeric vector.
`default`	A character scalar. What we want to get from 'var_lab' if there is no variable label. NULL by default.
`value`	A character scalar - label for the variable x.
`remove_classes`	A character vector of classes which should be removed from the class attribute of the `x`.

Details

Variable label is stored in attribute "label" (`attr(x,"label")`). For preserving from dropping this attribute during some operations (such as `c`) variable class is set to "labelled". There are special methods of subsetting and concatenation for this class. To drop variable label use `var_lab(var) <-NULL` or `unvr(var)`.

Value

var_lab return variable label. If label doesn't exist it return NULL . var_lab<- and set_var_lab return variable (vector x) of class "labelled" with attribute "label" which equals submitted value.

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "V/S"
                var_lab(am) = "Transmission (0 = automatic, 1 = manual)"
                val_lab(am) = c(automatic = 0, manual=1)
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

fre(mtcars$am)

calculate(mtcars,
     cro_mean(list(mpg, disp, hp, qsec), list(total(), am))
     )


## Not run:
if(FALSE){ # to prevent execution
# you need to load packages strictly in this order to avoid conflicts
library(haven)
library(expss)
spss_data = haven::read_spss("spss_file.sav")
# add missing 'labelled' class
spss_data = add_labelled_class(spss_data)
}

## End(Not run)
```

---

vectors *Infix operations on vectors - append, diff, intersection, union, replication*

Description

All these functions except %n_d%, %n_i% preserve names of vectors and don't remove duplicates.

- %a% a(ppends) second argument to the first argument. See also append.

- %u% and v_union u(nite) first and second arguments. Remove elements from the second argument which exist in the first argument.
- %d% and v_diff d(iff) second argument from the first argument. Second argument could be a function which returns logical value. In this case elements of the first argument which give TRUE will be removed.
- %i% and v_intersect i(ntersect) first argument and second argument. Second argument could be a function which returns logical value. In this case elements of the first argument which give FALSE will be removed.
- %e% and v_xor e(xclusive OR). Returns elements that contained only in one of arguments.
- %r% r(epeats) first argument second argument times. See also rep.
- %n_d% and n_diff n(ames) d(iff) - diff second argument from names of first argument. Second argument could be a function which returns logical value. In this case elements of the first argument which names give TRUE will be removed.
- %n_i% and n_intersect n(ames) i(ntersect) - intersect names of the first argument with the second argument. Second argument could be a function which returns logical value. In this case elements of the first argument which names give FALSE will be removed.

For %d%, %i%, %n_d%, %n_i% one can use criteria functions. See criteria for details.

Usage

```
e1 %a% e2

v_union(e1, e2)

e1 %u% e2

v_diff(e1, e2)

e1 %d% e2

v_intersect(e1, e2)

e1 %i% e2

v_xor(e1, e2)

e1 %e% e2

e1 %r% e2

n_intersect(e1, e2)

e1 %n_i% e2

n_diff(e1, e2)

e1 %n_d% e2
```

Arguments

e1	vector or data.frame, matrix, list for %n_d%, %n_i%)
e2	vector or function for %d%, %i%

Value

vector or data.frame, matrix, list for %n_d%, %n_i%)

Examples

```
1:4 %a% 5:6   # 1:6

1:4 %a% 4:5   # 1,2,3,4,4,5

1:4 %u% 4:5   # 1,2,3,4,5

1:6 %d% 5:6   # 1:4

# function as criterion
1:6 %d% greater(4) # 1:4

1:4 %i% 4:5   # 4

# with criteria functions
letters %i% (contains("a") | contains("z")) # a, z

letters %i% perl("[a-d]") # a,b,c,d

1:4 %e% 4:5   # 1, 2, 3, 5

1:2 %r% 2     # 1, 2, 1, 2

# %n_i%, %n_d%

# remove column Species
iris %n_d% "Species"

# leave only columns which names start with "Sepal"
iris %n_i% like("Sepal*")

# leave column "Species" and columns which names start with "Sepal"
iris %n_i% ("Species" | like("Sepal*"))
iris %n_i% or("Species", like("Sepal*")) # same result
```

---

vlookup *Look up values in dictionary.*

Description

vlookup/vlookup_df function is inspired by VLOOKUP spreadsheet function. It looks for a lookup_value in the lookup_column of the dict, and then returns values in the same rows from result_column. add_columns inspired by MATCH FILES (Add variables...) from SPSS Statistics. It works similar to SQL left join but number of cases in the left part always remain the same. If there are duplicated keys in the dict then error will be raised by default. .add_columns is the same function for default dataset.

Usage

```
vlookup(lookup_value, dict, result_column = 2, lookup_column = 1)

vlookup_df(lookup_value, dict, result_column = NULL, lookup_column = 1)

add_columns(data, dict, by = NULL, ignore_duplicates = FALSE)

.add_columns(dict, by = NULL, ignore_duplicates = FALSE)
```

Arguments

lookup_value	Vector of looked up values
dict	data.frame/matrix. Dictionary. Can be vector for vlookup/vlookup_df.
result_column	numeric or character. Resulting columns of dict. There are special values: 'row.names', 'rownames', 'names'. If result_column equals to one of these special values and dict is matrix/data.frame then row names of dict will be returned. If dict is vector then names of vector will be returned. For vlookup_df default result_column is NULL and result will be entire rows. For vlookup defaut result_column is 2 - for frequent case of dictionary with keys in the first column and results in the second column.
lookup_column	Column of dict in which lookup value will be searched. By default it is the first column of the dict. There are special values: 'row.names', 'rownames', 'names'. If lookup_column equals to one of these special values and dict is matrix/data.frame then values will be searched in the row names of dict. If dict is vector then values will be searched in names of the dict.
data	data.frame to be joined with dict.
by	character vector or NULL(default) or 1. Names of common variables in the data and dict by which we will attach dict to data. If it is NULL then common names will be used. If it is equals to 1 then we will use the first column from both dataframes. To add columns by different variables on data and dict use a named vector. For example, by = c("a" = "b") will match data.a to dict.b.
ignore_duplicates	logical Should we ignore duplicates in the by variables in the dict? If it is TRUE than first occurence of duplicated key will be used.

Value

vlookup always return vector, vlookup_df always returns data.frame. row.names in result of vlookup_df are not preserved.

Examples

```
# with data.frame
dict = data.frame(num=1:26, small=letters, cap=LETTERS, stringsAsFactors = FALSE)
rownames(dict) = paste0('rows', 1:26)
identical(vlookup_df(1:3, dict), dict[1:3,]) # should be TRUE
vlookup(c(45,1:3,58), dict, result_column='cap')
vlookup_df(c('z','d','f'), dict, lookup_column = 'small')
vlookup_df(c('rows7', 'rows2', 'rows5'), dict, lookup_column = 'row.names')

# with vector
dict=1:26
names(dict) = letters

vlookup(c(2,4,6), dict, result_column='row.names')

# The same results
vlookup(c(2,4,6), dict, result_column='rownames')
vlookup(c(2,4,6), dict, result_column='names')

# example for 'add_columns' from base 'merge'
authors = sheet(
    surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4))
)

books = sheet(
    surname = c("Tukey", "Venables", "Tierney",
                "Ripley", "Ripley", "McNeil", "R Core"),
    title = c("Exploratory Data Analysis",
              "Modern Applied Statistics ...",
              "LISP-STAT",
              "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis",
              "An Introduction to R")
)

add_columns(books, authors)

# Just for fun. Examples borrowed from Microsoft Excel.
# It is not the R way of doing things.

# Example 2

ex2 = utils::read.table(header = TRUE, text = "
    Item_ID Item Cost Markup
    ST-340 Stroller 145.67  0.30
    BI-567 Bib 3.56  0.40
    DI-328 Diapers  21.45  0.35
    WI-989 Wipes  5.12  0.40
    AS-469 Aspirator 2.56  0.45
", stringsAsFactors = FALSE)
```

```
# Calculates the retail price of diapers by adding the markup percentage to the cost.
vlookup("DI-328", ex2, 3) * (1 + vlookup("DI-328", ex2, 4)) # 28.9575

# Calculates the sale price of wipes by subtracting a specified discount from
# the retail price.
(vlookup("WI-989", ex2, "Cost") * (1 + vlookup("WI-989", ex2, "Markup"))) * (1 - 0.2)  # 5.7344

A2 = ex2[1, "Item_ID"]
A3 = ex2[2, "Item_ID"]

# If the cost of an item is greater than or equal to $20.00, displays the string
# "Markup is nn%"; otherwise, displays the string "Cost is under $20.00".
ifelse(vlookup(A2, ex2, "Cost") >= 20,
       paste0("Markup is " , 100 * vlookup(A2, ex2, "Markup"),"%"),
       "Cost is under $20.00") # Markup is 30%


# If the cost of an item is greater than or equal to $20.00, displays the string
# Markup is nn%"; otherwise, displays the string "Cost is $n.nn".
ifelse(vlookup(A3, ex2, "Cost") >= 20,
       paste0("Markup is: " , 100 * vlookup(A3, ex2, "Markup") , "%"),
       paste0("Cost is $", vlookup(A3, ex2, "Cost"))) #Cost is $3.56


# Example 3

ex3 = utils::read.table(header = TRUE, text = "
    ID  Last_name  First_name  Title Birth_date
    1 Davis Sara 'Sales Rep.'  12/8/1968
    2 Fontana Olivier 'V.P. of Sales' 2/19/1952
    3 Leal Karina 'Sales Rep.' 8/30/1963
    4 Patten Michael 'Sales Rep.' 9/19/1958
    5 Burke Brian 'Sales Mgr.' 3/4/1955
    6 Sousa Luis 'Sales Rep.'  7/2/1963
", stringsAsFactors = FALSE)

# If there is an employee with an ID of 5, displays the employee's last name;
# otherwise, displays the message "Employee not found".
if_na(vlookup(5, ex3, "Last_name"), "Employee not found") # Burke

# Many employees
if_na(vlookup(1:10, ex3, "Last_name"), "Employee not found")

# For the employee with an ID of 4, concatenates the values of three cells into
# a complete sentence.
paste0(vlookup(4, ex3, "First_name"), " ",
       vlookup(4, ex3, "Last_name"), " is a ",
       vlookup(4, ex3, "Title")) # Michael Patten is a Sales Rep.
```

---

where *Subset (filter) data.frames/matrices/vectors/lists*

Description

For the data frame cond will be evaluated in the data.frame's context. So columns can be referred as variables in the expression (see the examples). If data is list then where will be applied to each element of the list. For other types (vector/matrix) there is no non-standard evaluation. There is a special constant .N which equals to number of rows in data for usage in cond expression. .where is version for working with default dataset. See default_dataset.

Usage

```
where(data, cond)

.where(cond)
```

Arguments

data	data.frame/matrix/vector/list to be subsetted
cond	logical or numeric expression indicating elements or rows to keep: missing values (NA) are taken as FALSE. If data is data.frame then cond will be evaluated in the scope of the data.

Value

data.frame/matrix/vector/list which contains just selected rows.

Examples

```
# leave only 'setosa'
where(iris, Species == "setosa")
# leave only first five rows
where(iris, 1:5)


# example of .N usage.
set.seed(42)
train = where(iris, sample(.N, 100))
str(train)

set.seed(42)
test = where(iris, -sample(.N, 100))
str(test)

# list example
set.seed(123)
rand_matr = matrix(sample(10, 60, replace = TRUE), ncol = 3)
rand_vec = sample(10, 20, replace = TRUE)
my_list = list(iris, rand_matr, rand_vec)
# two random elements from the each list item
where(my_list, sample(.N, 2))
```

window_fun *Function over grouping variables (window function)*

Description

This is faster version of ave. window_fun applies function to every subset of x and return vector of the same length as x.

Usage

```
window_fun(x, ...)
```

Arguments

x	A vector
...	Grouping variables all of the same length as x or length 1 and function as last argument.

Value

vector of the same length as x

Examples

```
window_fun(1:3, mean)  # no grouping -> grand mean

attach(warpbreaks)

window_fun(breaks, wool, mean)
window_fun(breaks, tension, function(x) mean(x, trim = 0.1))

detach(warpbreaks)
```

write_labelled_csv *Write labelled data to file or export file to SPSS syntax.*

Description

- write_labelled_csv and read_labelled_csv writes csv file with labels. By default labels are stored in the commented lines at the beginning of the file before the data part. *_csv2 write and read data with a semicolon separator and comma as decimal delimiter. *_tab/*_tab2 write and read data with 'tab' separator and "."/"," as decimal delimiter.
- write_labelled_xlsx and read_labelled_xlsx write and read labelled 'xlsx' format. It is a simple Excel file with data and labels on separate sheets. It can help you with labelled data exchange in the corporate environment.

- `write_labelled_fst` and `read_labelled_fst` write and read labelled data in the 'fst' format. See Fst Package. Data and labels are stored in the separate files. With 'fst' format you can read and write a huge amount of data very quickly.
- write_labelled_spss write 'csv' file with SPSS syntax for reading it. You can use it for the data exchange with SPSS.
- `create_dictionary` and `apply_dictionary` make data.frame with dictionary, e. g. variable and value labels for each variable. See format description in the 'Details' section.
- `write_labels` and `write_labels_spss` Write R code and SPSS syntax for labelling data. It allows to extract labels from *.sav files that come without accompanying syntax.
- `old_write_labelled_csv` and `old_read_labelled_csv` Read and write labelled 'csv' in format of the 'expss' version before 0.9.0.

Usage

```
write_labelled_csv(
  x,
  filename,
  remove_new_lines = TRUE,
  single_file = TRUE,
  ...
)

write_labelled_csv2(
  x,
  filename,
  remove_new_lines = TRUE,
  single_file = TRUE,
  ...
)

write_labelled_tab(
  x,
  filename,
  remove_new_lines = TRUE,
  single_file = TRUE,
  ...
)

write_labelled_tab2(
  x,
  filename,
  remove_new_lines = TRUE,
  single_file = TRUE,
  ...
)

write_labelled_xlsx(
  x,
```

```
  filename,
  data_sheet = "data",
  dict_sheet = "dictionary",
  remove_repeated = FALSE,
  use_references = TRUE
)

write_labelled_fst(x, filename, ...)

read_labelled_csv(filename, undouble_quotes = TRUE, ...)

read_labelled_csv2(filename, undouble_quotes = TRUE, ...)

read_labelled_tab(filename, undouble_quotes = TRUE, ...)

read_labelled_tab2(filename, undouble_quotes = TRUE, ...)

read_labelled_xlsx(filename, data_sheet = 1, dict_sheet = "dictionary")

read_labelled_fst(filename, ...)

write_labelled_spss(
  x,
  filename,
  fileEncoding = "",
  remove_new_lines = TRUE,
  ...
)

write_labels_spss(x, filename)

write_labels(x, filename, fileEncoding = "")

create_dictionary(x, remove_repeated = FALSE, use_references = TRUE)

apply_dictionary(x, dict)

old_write_labelled_csv(
  x,
  filename,
  fileEncoding = "",
  remove_new_lines = TRUE,
  ...
)

old_read_labelled_csv(filename, fileEncoding = "", undouble_quotes = TRUE, ...)
```

Arguments

x	data.frame to be written/data.frame whose labels to be written
filename	the name of the file which the data are to be read from/write to.
remove_new_lines	A logical indicating should we replace new lines with spaces in the character variables. TRUE by default.
single_file	logical. TRUE by default. Should we write labels into the same file as data? If it is FALSE dictionary will be written in the separate file.
...	additional arguments for fwrite/fread, e. g. column separator, decimal separator, encoding and etc.
data_sheet	character "data" by default. Where data will be placed in the '*.xlsx' file.
dict_sheet	character "dictionary" by default. Where dictionary will be placed in the '*.xlsx' file.
remove_repeated	logical. FALSE by default. If TRUE then we remove repeated variable names. It makes a dictionary to look nicer for humans but less convenient for usage.
use_references	logical. When TRUE (default) then if the variable has the same value labels as the previous variable, we use reference to this variable. It makes dictionary significantly more compact for datasets with many variables with the same value labels.
undouble_quotes	A logical indicating should we undouble quotes which were escaped by doubling. TRUE by default. Argument will be removed when data.table issue #1109 will be fixed.
fileEncoding	character string: if non-empty declares the encoding to be used on a file (not a connection) so the character data can be re-encoded as they are written. Used for writing dictionary. See file.
dict	data.frame with labels - a result of `create_dictionary`.

Details

Dictionary is a data.frame with the following columns:

- variable variable name in the data set. It can be omitted (NA). In this case name from the previous row will be taken.
- value code for label in the column `'label'`.
- label in most cases it is value label but its meaning can be changed by the column `'meta'`.
- meta if it is NA then we have value label in the `'label'` column. If it is `'varlab'`, then there is a variable label in the `'label'` column and column `'value'` is ignored. If it is `'reference'`, then there is a variable name in the `'label'` column and we use value labels from this variable, column `'value'` is ignored.

Value

Functions for writing invisibly return NULL. Functions for reading return labelled data.frame.

Examples

```
## Not run:
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "Engine"
                val_lab(vs) = c("V-engine" = 0,
                                "Straight engine" = 1)
                var_lab(am) = "Transmission"
                val_lab(am) = c(automatic = 0,
                                manual=1)
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

write_labelled_csv(mtcars, "mtcars.csv")
new_mtcars = read_labelled_csv("mtcars.csv")
str(new_mtcars)

# identically, for xlsx
write_labelled_xlsx(mtcars, "mtcars.xlsx")
new_mtcars = read_labelled_xlsx("mtcars.xlsx")
str(new_mtcars)

# to SPSS syntax
write_labelled_spss(mtcars, "mtcars.csv")


## End(Not run)
```

`w_mean` *Compute various weighted statistics*

Description

- `w_mean` weighted mean of a numeric vector
- `w_sd` weighted sample standard deviation of a numeric vector
- `w_var` weighted sample variance of a numeric vector
- `w_se` weighted standard error of a numeric vector
- `w_median` weighted median of a numeric vector
- `w_mad` weighted mean absolute deviation from median of a numeric vector

- w_sum weighted sum of a numeric vector
- w_n weighted number of values of a numeric vector
- w_cov weighted covariance matrix of a numeric matrix/data.frame
- w_cor weighted Pearson correlation matrix of a numeric matrix/data.frame
- w_pearson shortcut for w_cor. Weighted Pearson correlation matrix of a numeric matrix/data.frame
- w_spearman weighted Spearman correlation matrix of a numeric matrix/data.frame

Usage

```
w_mean(x, weight = NULL, na.rm = TRUE)

w_median(x, weight = NULL, na.rm = TRUE)

w_var(x, weight = NULL, na.rm = TRUE)

w_sd(x, weight = NULL, na.rm = TRUE)

w_se(x, weight = NULL, na.rm = TRUE)

w_mad(x, weight = NULL, na.rm = TRUE)

w_sum(x, weight = NULL, na.rm = TRUE)

w_n(x, weight = NULL, na.rm = TRUE)

unweighted_valid_n(x, weight = NULL)

valid_n(x, weight = NULL)

w_max(x, weight = NULL, na.rm = TRUE)

w_min(x, weight = NULL, na.rm = TRUE)

w_cov(x, weight = NULL, use = c("pairwise.complete.obs", "complete.obs"))

w_cor(x, weight = NULL, use = c("pairwise.complete.obs", "complete.obs"))

w_pearson(x, weight = NULL, use = c("pairwise.complete.obs", "complete.obs"))

w_spearman(x, weight = NULL, use = c("pairwise.complete.obs", "complete.obs"))
```

Arguments

x	a numeric vector (matrix/data.frame for correlations) containing the values whose weighted statistics is to be computed.
weight	a vector of weights to use for each element of x. Cases with missing, zero or negative weights will be removed before calculations. If weight is missing then unweighted statistics will be computed.

na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds. Note that contrary to base R statistic functions the default value is TRUE (remove missing values).
use	"pairwise.complete.obs" (default) or "complete.obs". In the first case the correlation or covariance between each pair of variables is computed using all complete pairs of observations on those variables. If use is "complete.obs" then missing values are handled by casewise deletion.

Details

If argument of correlation functions is data.frame with variable labels then variables names will be replaced with labels. If this is undesirable behavior use unvr function: w_cor(unvr(x)). Weighted spearman correlation coefficients are calculated with rounded to nearest integer weights. It gives the same result as in SPSS Statistics software. By now this algorithm is not memory efficient.

Value

a numeric value of length one/correlation matrix

Examples

```
data(mtcars)
dfs = mtcars %>% keep(mpg, disp, hp, wt)

with(dfs, w_mean(hp, weight = 1/wt))

# apply labels
dfs = modify(dfs, {
         var_lab(mpg) = "Miles/(US) gallon"
         var_lab(disp) = "Displacement (cu.in.)"
         var_lab(hp) = "Gross horsepower"
         var_lab(wt) = "Weight (1000 lbs)"
})

# weighted correlations with labels
w_cor(dfs, weight = 1/dfs$wt)

# without labels
w_cor(unvr(dfs), weight = 1/dfs$wt)
```

---

xl_write *Write tables and other objects to an xlsx file with formatting*

Description

Note that openxlsx package is required for these functions. It can be install by printing install.packages('openxlsx') in the console. On Windows system you also may need to install rtools. You can export several tables at once by combining them in a list. See examples. If you need to write all tables to the single sheet you can use xl_write_file. It automatically creates workbook, worksheet and save *.xlsx file for you.

Usage

```
xl_write(obj, wb, sheet, row = 1, col = 1, ...)

xl_write_file(obj, filename, sheetname = "Tables", ...)

## Default S3 method:
xl_write(
  obj,
  wb,
  sheet,
  row = 1,
  col = 1,
  rownames = FALSE,
  colnames = !is.atomic(obj),
  ...
)

## S3 method for class 'list'
xl_write(obj, wb, sheet, row = 1, col = 1, gap = 1, ...)

## S3 method for class 'etable'
xl_write(
  obj,
  wb,
  sheet,
  row = 1,
  col = 1,
  remove_repeated = c("all", "rows", "columns", "none"),
  format_table = TRUE,
  borders = list(borderColour = "black", borderStyle = "thin"),
 header_format = openxlsx::createStyle(fgFill = "#EBEBEB", halign = "left", wrapText =
    FALSE),
 main_format = openxlsx::createStyle(halign = "right", numFmt = format(0, nsmall =
    get_expss_digits())),
 row_labels_format = openxlsx::createStyle(halign = "left"),
 total_format = openxlsx::createStyle(fgFill = "#EBEBEB", border = "TopBottom",
    borderStyle = "thin", halign = "right", numFmt = "0"),
 total_row_labels_format = openxlsx::createStyle(fgFill = "#EBEBEB", border =
    "TopBottom", borderStyle = "thin", halign = "left"),
 top_left_corner_format = header_format,
 row_symbols_to_remove = NULL,
 col_symbols_to_remove = NULL,
 other_rows_formats = NULL,
 other_row_labels_formats = NULL,
 other_cols_formats = NULL,
 other_col_labels_formats = NULL,
 additional_cells_formats = NULL,
 ...
```

```
)

## S3 method for class 'with_caption'
xl_write(
  obj,
  wb,
  sheet,
  row = 1,
  col = 1,
  remove_repeated = c("all", "rows", "columns", "none"),
  format_table = TRUE,
  borders = list(borderColour = "black", borderStyle = "thin"),
 header_format = openxlsx::createStyle(fgFill = "#EBEBEB", halign = "left", wrapText =
    FALSE),
 main_format = openxlsx::createStyle(halign = "right", numFmt = format(0, nsmall =
    get_expss_digits())),
  row_labels_format = openxlsx::createStyle(halign = "left"),
  total_format = openxlsx::createStyle(fgFill = "#EBEBEB", border = "TopBottom",
    borderStyle = "thin", halign = "right", numFmt = "0"),
  total_row_labels_format = openxlsx::createStyle(fgFill = "#EBEBEB", border =
    "TopBottom", borderStyle = "thin", halign = "left"),
  top_left_corner_format = header_format,
  row_symbols_to_remove = NULL,
  col_symbols_to_remove = NULL,
  other_rows_formats = NULL,
  other_row_labels_formats = NULL,
  other_cols_formats = NULL,
  other_col_labels_formats = NULL,
  additional_cells_formats = NULL,
 caption_format = openxlsx::createStyle(textDecoration = "bold", halign = "left"),
  ...
)
```

Arguments

obj	table - result of cro, fre and etc. obj also can be data.frame, list or other objects.
wb	xlsx workbook object, result of createWorkbook function.
sheet	character or numeric - worksheet name/number in the workbook wb
row	numeric - starting row for writing data
col	numeric - starting column for writing data
...	further arguments for xl_write
filename	A character string naming an xlsx file. For xl_write_file.
sheetname	A character name for the worksheet. For xl_write_file.
rownames	logical should we write data.frame row names?
colnames	logical should we write data.frame column names?
gap	integer. Number of rows between list elements.

`remove_repeated`
: Should we remove duplicated row or column labels in the rows/columns of the etable? Possible values: "all", "rows", "columns", "none".

`format_table`
: logical should we format table? If FALSE all format arguments will be ignored.

`borders`
: list Style of the table borders. List with two named elements: `borderColour` and `borderStyle`. For details see createStyle function. If it is NULL then no table borders will be produced.

`header_format`
: table header format - result of the createStyle function.

`main_format`
: result of the createStyle function. Format of the table main area except total rows. Total rows is rows which row labels contain '#'.

`row_labels_format`
: result of the createStyle function. Format of the row labels area except total rows. Total rows is rows which row labels contain '#'.

`total_format`
: result of the createStyle function. Format of the total rows in the table main area. Total rows is rows which row labels contain '#'.

`total_row_labels_format`
: result of the createStyle function. Format of the total rows in the row labels area. Total rows is rows which row labels contain '#'.

`top_left_corner_format`
: result of the createStyle function.

`row_symbols_to_remove`
: character vector. Perl-style regular expressions for substrings which will be removed from row labels.

`col_symbols_to_remove`
: character vector. Perl-style regular expressions for substrings which will be removed from column names.

`other_rows_formats`
: named list. Names of the list are perl-style regular expression patterns, items of the list are results of the createStyle function. Rows in the main area which row labels contain pattern will be formatted according to the appropriate style.

`other_row_labels_formats`
: named list. Names of the list are perl-style regular expression patterns, items of the list are results of the createStyle function. Rows in the row labels area which row labels contain pattern will be formatted according to the appropriate style.

`other_cols_formats`
: named list. Names of the list are perl-style regular expression patterns, items of the list are results of the createStyle function. Columns in the main area which column labels contain pattern will be formatted according to the appropriate style.

`other_col_labels_formats`
: named list. Names of the list are perl-style regular expression patterns, items of the list are results of the createStyle function. Columns in the header area which column labels contain pattern will be formatted according to the appropriate style.

additional_cells_formats
: list Each item of the list is list which consists of two elements. First element is two columns matrix or data.frame with row number and column numbers in the main area of the table. Such matrix can be produced with code `which(logical_condition,arr.ind = TRUE)`. Instead of matrix one can use function which accepts original table (`obj`) and return such matrix. Second element is result of the createStyle function. Cells in the main area will be formatted according to this style.

caption_format
: result of the createStyle function.

Value

invisibly return vector with rows and columns (`c(rows,columns)`) occupied by outputted object.

Examples

```
## Not run:
library(openxlsx)
data(mtcars)
# add labels to dataset
mtcars = apply_labels(mtcars,
                      mpg = "Miles/(US) gallon",
                      cyl = "Number of cylinders",
                      disp = "Displacement (cu.in.)",
                      hp = "Gross horsepower",
                      drat = "Rear axle ratio",
                      wt = "Weight (lb/1000)",
                      qsec = "1/4 mile time",
                      vs = "Engine",
                      vs = c("V-engine" = 0,
                             "Straight engine" = 1),
                      am = "Transmission",
                      am = c("Automatic" = 0,
                             "Manual"=1),
                      gear = "Number of forward gears",
                      carb = "Number of carburetors"
)

# create table with caption
mtcars_table = calc_cro_cpct(mtcars,
                             cell_vars = list(cyl, gear),
                             col_vars = list(total(), am, vs)
) %>%
    set_caption("Table 1")


wb = createWorkbook()
sh = addWorksheet(wb, "Tables")
# export table
xl_write(mtcars_table, wb, sh)
saveWorkbook(wb, "table1.xlsx", overwrite = TRUE)
```

```
## quick export
xl_write_file(mtcars_table, "table1.xlsx")

## custom cells formatting
wb = createWorkbook()
sh = addWorksheet(wb, "Tables")

# we want to mark cells which are greater than total column
my_formatter = function(tbl){
    greater_than_total = tbl[,-1]>tbl[[2]]
    which(greater_than_total, arr.ind = TRUE)
}
# export table
xl_write(mtcars_table, wb, sh,
    additional_cells_formats = list(
        list(my_formatter, createStyle(textDecoration =  "bold", fontColour = "blue"))
    )
)
saveWorkbook(wb, "table_with_additional_format.xlsx", overwrite = TRUE)

## automated report generation on multiple variables with the same banner

banner = calc(mtcars, list(total(), am, vs))

# create list of tables
list_of_tables = lapply(mtcars, function(variable) {
    if(length(unique(variable))<7){
        cro_cpct(variable, banner) %>% significance_cpct()
    } else {
        # if number of unique values greater than seven we calculate mean
        cro_mean_sd_n(variable, banner) %>% significance_means()

    }

})


wb = createWorkbook()
sh = addWorksheet(wb, "Tables")
# export list of tables with additional formatting
xl_write(list_of_tables, wb, sh,
         # remove '#' sign from totals
         col_symbols_to_remove = "#",
         row_symbols_to_remove = "#",
         # format total column as bold
         other_col_labels_formats = list("#" = createStyle(textDecoration = "bold")),
         other_cols_formats = list("#" = createStyle(textDecoration = "bold")),
)
saveWorkbook(wb, "report.xlsx", overwrite = TRUE)

## End(Not run)
```

Index