Package 'qol'

March 11, 2026

Title Powerful 'SAS' Inspired Concepts for more Efficient Bigger
Outputs

Version 1.2.2

Description The main goal is to make descriptive evaluations easier to create bigger and more complex outputs in less time with less code. Introducing format containers with multilabels <https://documentation.sas.com/doc/en/pgmsascdc/v_067/proc/p06ciqes4eaqo6n0zyqtz9p21nfb.htm>, a more powerful summarise which is capable to output every possible combination of the provided grouping variables in one go <https://documentation.sas.com/doc/en/pgmsascdc/v_067/proc/p0jvbbqkt0gs2cn1lo4zndbqs1pe.htm>, tabulation functions which can create any table in different styles <https://documentation.sas.com/doc/en/pgmsascdc/v_067/proc/n1ql5xnu0k3kdtn11gwa5hc7u435.htm> and other more readable functions. The code is optimized to work fast even with datasets of over a million observations.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

URL https://github.com/s3rdia/qol, https://s3rdia.github.io/qol/

Imports data.table (>= 1.17.8), collapse (>= 2.1.2), openxlsx2 (>=
1.19)

Depends R (>= 4.1.0)

RoxygenNote 7.3.3

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

Config/Needs/website rmarkdown

NeedsCompilation no

Author Tim Siebenmorgen [aut, cre, cph]

Maintainer Tim Siebenmorgen <qol_package@proton.me>

Repository CRAN

Date/Publication 2026-03-11 00:20:09 UTC

Contents

---

add_extension   *Add Extensions to Variable Names*

Description

Renames variables in a data frame by adding the desired extensions to the original names. This can be useful if you want to use pre summarised data with any_table(), which needs the value variables to have the statistic extensions.

Usage

```
add_extension(data_frame, from, extensions, reuse = "none")
```

Arguments

data_frame	The data frame in which variables should gain extensions to their name.
from	The position of the variable inside the data frame at which to start the renaming.
extensions	The extensions to add.
reuse	"none" by default, meaning only the provided extensions will be set. E.g. if there are two extensions provided, two variables will be renamed. If "last", the last provided extension will be used for every following variable until the end of the data frame. If "repeat", the provided extensions will be repeated from the first one for every following variable until the end of the data frame.

Value

Returns a data frame with extended variable names.

Examples

```
# Example data frame
my_data <- dummy_data(10)

# Add extensions to variable names
new_names1 <- my_data |> add_extension(5, c("sum", "pct"))
new_names2 <- my_data |> add_extension(5, c("sum", "pct"), reuse = "last")
new_names3 <- my_data |> add_extension(5, c("sum", "pct"), reuse = "alternate")
```

---

add_variable_range   *Add Empty Variables In A Given Range*

Description

Add empty variables to a data frame in the provided range. Basically does in a data frame, what paste0("age", 1:10) does for a vector.

Usage

```
add_variable_range(data_frame, var_range)
```

Arguments

data_frame   A data frame to add variables to.

var_range   A range of variables to add, provided in the form: var_name1:var_name10.

Value

Returns a data frame with added variables.

Examples

```
# Example data frames
my_data <- dummy_data(100)

# Add variable range
my_data <- my_data |> add_variable_range(status1:status12)
```

---

any_table   *Compute Any Possible Table*

Description

[any_table()](#) produces any possible descriptive table in 'Excel' format. Any number of variables can be nested and crossed. The output is an individually styled 'Excel' table, which also receives named ranges, making it easier to read the data back in.

Usage

```
any_table(
  data_frame,
  rows,
  columns = "",
  values,
  statistics = "sum",
  pct_group = c(),
  pct_value = list(),
  formats = list(),
  by = c(),
  weight = NULL,
  order_by = "stats",
  titles = .qol_options[["titles"]],
  footnotes = .qol_options[["footnotes"]],
  var_labels = .qol_options[["var_labels"]],
  stat_labels = .qol_options[["stat_labels"]],
  box = "",
  workbook = NULL,
  style = .qol_options[["excel_style"]],
  output = .qol_options[["output"]],
  na.rm = .qol_options[["na.rm"]],
  print_miss = .qol_options[["print_miss"]],
  print = .qol_options[["print"]],
  monitor = .qol_options[["monitor"]]
)
```

Arguments

`data_frame`	A data frame in which are the variables to tabulate.
`rows`	A vector that provides single variables or variable combinations that should appear in the table rows. To nest variables use the form: "var1 + var2 + var3 + ...".
`columns`	A vector that provides single variables or variable combinations that should appear in the table rows. To nest variables use the form: "var1 + var2 + var3 + ...".
`values`	A vector containing all variables that should be summarised.
`statistics`	Available functions:

- "sum" -> Weighted and unweighted sum
- "sum_wgt" -> Sum of all weights
- "freq" -> Unweighted frequency
- "freq_g0" -> Unweighted frequency of all values greater than zero
- "pct_group" -> Weighted and unweighted percentages within the respective group
- "pct_value" -> Weighted and unweighted percentages between value variables or of a variable expression

- "pct_total" -> Weighted and unweighted percentages compared to the grand total
- "mean" -> Weighted and unweighted mean
- "median" -> Weighted and unweighted median
- "mode" -> Weighted and unweighted mode
- "min" -> Minimum
- "max" -> Maximum
- "sd" -> Weighted and unweighted standard deviation
- "variance" -> Weighted and unweighted standard variance
- "first" -> First value
- "last" -> Last value
- "pn" -> Weighted and unweighted percentiles (any p1, p2, p3, ... possible)
- "missing" -> Missings generated by the value variables

pct_group	If pct_group is specified in the statistics, this option is used to determine which variable of the row and column variables should add up to 100 %. Multiple variables can be specified in a vector to generate multiple group percentages. You can also use the keywords "row_pct" or "col_pct" to calculate total percentages for rows and columns regardless of the respective other dimension.
pct_value	If pct_value is specified in the statistics, you can pass a list here which contains the information for a new variable name and between which of the value variables percentages should be computed.
formats	A list in which is specified which formats should be applied to which variables.
by	Compute tables stratified by the expressions of the provided variables.
weight	Put in a weight variable to compute weighted results.
order_by	Determine how the columns will be ordered. "values" orders the results by the order you provide the variables in values. "stats" orders them by the order under statistics. "values_stats" is a combination of both. "columns" keeps the order as given in columns and "interleaved" alternates the stats.
titles	Specify one or more table titles.
footnotes	Specify one or more table footnotes.
var_labels	A list in which is specified which label should be printed for which variable instead of the variable name.
stat_labels	A list in which is specified which label should be printed for which statistic instead of the statistic name.
box	Provide a text for the upper left box of the table.
workbook	Insert a previously created workbook to expand the sheets instead of creating a new file.
style	A list of options can be passed to control the appearance of 'Excel' outputs. Styles can be created with [excel_output_style()]().
output	The following output formats are available: excel and excel_nostyle.
na.rm	FALSE by default. If TRUE removes all NA values from the variables.

print_miss	FALSE by default. If TRUE outputs all possible categories of the grouping variables based on the provided formats, even if there are no observations for a combination.
print	TRUE by default. If TRUE prints the output, if FALSE doesn't print anything. Can be used if one only wants to catch the output data frame and workbook with meta information.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

[any_table()](#) is based on the 'SAS' procedure Proc Tabulate, which provides efficient and readable ways to perform complex tabulations.

With this function you can combine any number of variables in any possible way, all at once. You just define which variables or variable combinations should end up in the table rows and columns with a simple syntax. Listing variables in a vector like c("var1", "var2", "var3",...) means to put variables below (in case of the row variables) or besides (in case of the column variables) each other. Nesting variables is as easy as putting a plus sign between them, e.g. c("var1 + var2", "var2" + "var3" + "var4", etc.). And of course you can combine both versions.

The real highlight is, that this function not only creates all the desired variable combinations and exports them to an 'Excel' file, it prints a fully custom styled table to a workbook. Setting up a custom, reusable style is as easy as setting up options like: provide a color for the table header, set the font size for the row header, should borders be drawn for the table cells yes/no, and so on. Merging doubled header texts, happens automatically.

With this function you basically can fully concentrate on designing a table, instead of thinking hard about how to calculate where to put a border or to even manually prepare a designed workbook.

Value

Returns a list with the data table containing the results for the table, the formatted 'Excel' workbook and the meta information needed for styling the final table.

See Also

Creating a custom table style: [excel_output_style()](#), [modify_output_style()](#), [number_format_style()](#), [modify_number_formats()](#).

Global style options: [set_style_options()](#), [set_variable_labels()](#), [set_stat_labels()](#).

Creating formats: [discrete_format()](#) and [interval_format()](#).

Functions that can handle formats and styles: [frequencies()](#), [crosstabs()](#).

Additional functions that can handle styles: [export_with_style()](#)

Additional functions that can handle formats: [summarise_plus()](#), [recode()](#), [recode_multi()](#), [transpose_plus()](#), [sort_plus()](#)

Examples

```
# Example data frame
my_data <- dummy_data(1000)
my_data[["person"]] <- 1

# Formats
age. <- discrete_format(
    "Total"         = 0:100,
    "under 18"      = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"  = 65:100)

sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

education. <- discrete_format(
    "Total"            = c("low", "middle", "high"),
    "low education"    = "low",
    "middle education" = "middle",
    "high education"   = "high")

state. <- discrete_format(
    "Germany"                       = 1:16,
    "Schleswig-Holstein"            = 1,
    "Hamburg"                       = 2,
    "Lower Saxony"                  = 3,
    "Bremen"                        = 4,
    "North Rhine-Westphalia"        = 5,
    "Hesse"                         = 6,
    "Rhineland-Palatinate"          = 7,
    "Baden-Württemberg"             = 8,
    "Bavaria"                       = 9,
    "Saarland"                      = 10,
    "West"                          = 1:10,
    "Berlin"                        = 11,
    "Brandenburg"                   = 12,
    "Mecklenburg-Western Pomerania" = 13,
    "Saxony"                        = 14,
    "Saxony-Anhalt"                 = 15,
    "Thuringia"                     = 16,
    "East"                          = 11:16)

# Define style
set_style_options(column_widths = c(2, 15, 15, 15, 9))

# Define titles and footnotes. If you want to add hyperlinks you can do so by
# adding "link:" followed by the hyperlink to the main text.
set_titles("This is title number 1 link: https://cran.r-project.org/",
```

```
              "This is title number 2",
              "This is title number 3")

  set_footnotes("This is footnote number 1",
                "This is footnote number 2",
                "This is footnote number 3 link: https://cran.r-project.org/")

  # Output complex tables with different percentages
  my_data |> any_table(rows      = c("sex + age", "sex", "age"),
                       columns    = c("year", "education + year"),
                       values     = weight,
                       statistics = c("sum", "pct_group"),
                       pct_group  = c("sex", "age", "row_pct", "col_pct"),
                       formats    = list(sex = sex., age = age.,
                                             education = education.),
                       na.rm      = TRUE)

  # If you want to get a clearer vision of what the result table looks like, in terms
  # of the row and column categories, you can write the code like this, to make out
  # the variable crossings and see the order.
  my_data |> any_table(columns = c(              "year", "education + year"),
                       rows     = c("sex + age",
                                        "sex",
                                        "age"),
                       values     = weight,
                       statistics = c("sum", "pct_group"),
                       pct_group  = c("sex", "age", "education", "year"),
                       formats    = list(sex = sex., age = age.,
                                             education = education.),
                       na.rm      = TRUE)

  # Percentages based on value variables instead of categories
  my_data |> any_table(rows      = c("age + year"),
                       columns    = "sex",
                       values     = c(probability, person),
                       statistics = c("pct_value", "sum", "freq"),
                       pct_value  = list(rate = "probability / person"),
                       weight     = weight,
                       formats    = list(sex = sex., age = age.),
                       na.rm      = TRUE)

  # Percentages based on a formatted variable expression
  my_data |> any_table(rows      = c("age + year"),
                       columns    = "sex",
                       values     = c(probability, person),
                       statistics = "pct_value",
                       pct_value  = list(sex = "Total", age = "Total"),
                       weight     = weight,
                       formats    = list(sex = sex., age = age.),
                       na.rm      = TRUE)

  # Customize the visual appearance by adding variable and statistic labels. Both
  # can also be set as a global option, if labels should be reused over multiple
```

```
# tables.
# Note: You don't have to describe every element. Sometimes a table can be more
# readable with less text. To completely remove a variable label just put in an
# empty text "" as label.
my_data |> any_table(rows       = c("age + year"),
                     columns    = "sex",
                     values     = weight,
                     statistics = c("sum", "pct_group"),
                     order_by   = "interleaved",
                     formats    = list(sex = sex., age = age.),
                     var_labels = list(age = "Age categories",
                                       sex = "", weight = ""),
                     stat_labels = list(pct = "%"),
                     na.rm      = TRUE)


# Individual styling can also be passed directly
my_style <- excel_output_style(header_back_color = "0077B6",
                               font              = "Times New Roman")


my_data |> any_table(rows       = c("age + year"),
                     columns    = "sex",
                     values     = c(probability, person),
                     statistics = c("pct_value", "sum", "freq"),
                     pct_value  = list(rate = "probability / person"),
                     weight     = weight,
                     formats    = list(sex = sex., age = age.),
                     style      = my_style,
                     na.rm      = TRUE)


# Pass on workbook to create more sheets in the same file
my_style <- my_style |> modify_output_style(sheet_name = "age_sex")


result_list <- my_data |>
          any_table(rows       = "age",
                    columns    = "sex",
                    values     = weight,
                    statistics = "sum",
                    formats    = list(sex = sex., age = age.),
                    style      = my_style,
                    na.rm      = TRUE,
                    print      = FALSE)


my_style <- my_style |> modify_output_style(sheet_name = "edu_year")


my_data |> any_table(workbook   = result_list[["workbook"]],
                     rows       = "education",
                     columns    = "year",
                     values     = weight,
                     statistics = "pct_group",
                     formats    = list(education = education.),
                     style      = my_style,
                     na.rm      = TRUE)
```

```
# The result list from above also carries the transformed data frame if
# needed for further usage
any_table_df <- result_list[["table"]]

# Output multiple complex tables by expressions of another variable.
# If you specify the sheet name as "by" in the output style, the sheet
# names are named by the variable expressions of the by-variable. Otherwise
# the given sheet named gets a running number.
my_style <- my_style |> modify_output_style(sheet_name = "by")

my_data |> any_table(rows      = c("sex", "age"),
                     columns   = "education + year",
                     values    = weight,
                     by        = state,
                     statistics = c("sum", "pct_group"),
                     pct_group = "education",
                     formats   = list(sex = sex., age = age., state = state.,
                                       education = education.),
                     na.rm     = TRUE)

# To save a table as xlsx file you have to set the path and filename in the
# style element
# Example files paths
table_file <- tempfile(fileext = ".xlsx")

# Note: Normally you would directly input the path ("C:/MyPath/") and name ("MyFile.xlsx").
set_style_options(save_path  = dirname(table_file),
                  file       = basename(table_file),
                  sheet_name = "MyTable")

my_data |> any_table(rows      = "sex",
                     columns   = "year",
                     values    = weight,
                     formats   = list(sex = sex.))

# Manual cleanup for example
unlink(table_file)

# Global options are permanently active until the current R session is closed.
# There are also functions to reset the values manually.
reset_style_options()
reset_qol_options()
close_file()
```

---

build_master   *Build a Master Script From Folder*

Description

[build_master()](build_master) reads a given folder structure, which contains scripts, and builds a master script as a markdown file.

Usage

```
build_master(
  dir,
  master_name = "Master",
  author = "",
  with_structure = TRUE,
  with_run_all = TRUE,
  with_run_folder = TRUE
)
```

Arguments

dir	The folder structure which contains the scripts to build upon.
master_name	The file name which should be written.
author	Authors name to be put in the header.
with_structure	Whether the folder structure as tree should be written to the master script.
with_run_all	Whether a section, which let's the user run all scripts, should be written to the master script.
with_run_folder	
	Whether a section, which let's the user run all scripts from a specific folder, should be written to the master script.

Details

The function works with folder structures that look like this:

root/

  subfolder1/

      script1.R

      script2.R

      ....R

  subfolder2/

      script3.R

      script4.R

      ....R

```
.../

    ....R
```

Value

Returns the script as character vector and saves it as markdown file.

Examples

```
# Example export file paths
# NOTE: These tempfiles are only for the examples. In reality you just call the
# main function and put in your desired path and name directly.
temp_file <- tempfile(fileext = ".rstheme")
file_name <- basename(tools::file_path_sans_ext(temp_file))

# Example master
build_master(dir        = dirname(temp_file),
             master_name = file_name)

# Manual cleanup for example
unlink(temp_file)
```

---

build_rstheme   *Build a Theme From Scratch*

Description

Build your own theme by just setting up the colors for the different parts of RStudio. A theme file
will be exported which can be added by going to:

Tools -> Global Options -> Appearance -> Add

Usage

```
build_rstheme(
  file_path,
  theme_name = "qol_green",
  dark_theme = TRUE,
  editor_background = "#062625",
  editor_headline = "#3B3B3B",
  editor_font = "#C3B79D",
  toolbar = "#2E2E2E",
  tab = "#3B3B3B",
  selected_tab = "#062625",
  line_number = "#C3B79D",
  print_margin = "#3B3B3B",
```

```
    cursor = "#CCCCCC",
    selection = "#1B436E",
    smart_highlight = "#3686dc",
    bracket_highlight = "#595959",
    active_line = "#202324",
    whitespace = "#CCCCCC",
    debug_line = "#F18889",
    scrollbar = "#3B3B3B",
    scrollbar_hover = "#595959",
    scrollbar_active = "#BFBFBF",
    class_name = "#BEDD1A",
    keyword = "#FFC90E",
    language_constant = "#FFC90E",
    function_name = "#C3B79D",
    numeric = "#C93F3F",
    string = "#63C2C9",
    regex = "#E8E6E3",
    variable = "#E8E6E3",
    comment = "#32CD32",
    symbol = "#C3B79D",
    console_code = "#C3B79D",
    markdown_code = "#083332"
)
```

Arguments

`file_path`	The path to which the theme file should be saved.
`theme_name`	The themes name.
`dark_theme`	Handles some elements not covered with the other parameters.
`editor_background`	
	Base background color in the editor.
`editor_headline`	
	Mostly used for the headlines of the environment panel.
`editor_font`	Base font color of the editor.
`toolbar`	Base toolbar and frame color.
`tab`	Color of inactive tabs.
`selected_tab`	Color of active tabs.
`line_number`	The color of the line numbers on the left.
`print_margin`	Color of the vertical line showing the print margin.
`cursor`	Cursor color.
`selection`	The background color of the current selection.
`smart_highlight`	
	Background color of smart highlighted words.
`bracket_highlight`	
	Background color of highlighted bracket pairs.

`active_line`	Color for the active line the cursor is in.
`whitespace`	Color for whitespace characters.
`debug_line`	Color of the current debug line.
`scrollbar`	Color of the scrollbars.
`scrollbar_hover`	
	Highlight color when hovering over a scrollbar.
`scrollbar_active`	
	Highlight color when clicking on a scrollbar.
`class_name`	Code color for class names (like package names).
`keyword`	Code color for fixed keywords (like function, if, else).
`language_constant`	
	Code color for language constants (like the @ keywords).
`function_name`	Code color for base and package functions.
`numeric`	Code color for numeric values.
`string`	Code color for string values.
`regex`	Code color for regex expressions.
`variable`	Code color for variables, parameters and arguments.
`comment`	Code color for comments.
`symbol`	Code Color of symbols (like <-, brackets).
`console_code`	Color of executed Code in the Console.
`markdown_code`	Background color of code passages in a markdown file.

Details

In the 'SAS Enterprise Guide' the user is able to not only choose a given theme, but to also pick the colors for the different parts of the editor by themselves. Everyone has a different taste of what colors look pleasing to the eyes, so you should be able to choose them by yourself.

Value

Saves a complete theme file.

Examples

```
# Example export file paths
# NOTE: These tempfiles are only for the examples. In reality you just call the
# main function and put in your desired path and name directly.
temp_file <- tempfile(fileext = ".rstheme")
file_name <- basename(tools::file_path_sans_ext(temp_file))

# Example theme
build_rstheme(file_path         = dirname(temp_file),
              theme_name        = file_name,
              editor_background = "#417291",
              editor_headline   = "#602BCA",
```

```
            editor_font       = "#C75C48")

# Manual cleanup for example
unlink(temp_file)
```

---

combine_into_workbook   *Combine Multiple Tables Into One Workbook*

Description

Combines any number of tables created with any_table() into one workbook and styles them according to their meta information.

Usage

```
combine_into_workbook(
  ...,
  file = NULL,
  output = "excel",
  print = TRUE,
  monitor = FALSE
)
```

Arguments

...	Provide any number of result lists output by any_table().
file	If NULL, opens the output as temporary file. If a filename with path is specified, saves the output to the specified path.
output	The following output formats are available: excel and excel_nostyle.
print	TRUE by default. If TRUE prints the output, if FALSE doesn't print anything. Can be used if one only wants to catch the combined workbook.
monitor	FALSE by default. If TRUE outputs two charts to visualize the functions time consumption.

Value

A fully styled workbook containing the provided tables.

Examples

```
# Example data frame
my_data <- dummy_data(1000)
my_data[["person"]] <- 1

# Formats
age. <- discrete_format(
```

```
    "Total"          = 0:100,
    "under 18"       = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"   = 65:100)

sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

education. <- discrete_format(
    "Total"            = c("low", "middle", "high"),
    "low education"    = "low",
    "middle education" = "middle",
    "high education"   = "high")

# Define style
set_style_options(column_widths = c(2, 15, 15, 15, 9))

# Define titles and footnotes. If you want to add hyperlinks you can do so by
# adding "link:" followed by the hyperlink to the main text.
set_titles("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")
set_footnotes("This is footnote number 1",
              "This is footnote number 2",
              "This is footnote number 3 link: https://cran.r-project.org/")

# Catch the output and additionally use the options:
# pint = FALSE and output = "excel_nostyle".
# This skips the styling and output part, so that the function runs faster.
# The styling is done later on.
set_print(FALSE)
set_output("excel_nostyle")
set_style_options(sheet_name = "big table")

tab1 <- my_data |> any_table(rows       = c("sex + age", "sex", "age"),
                             columns    = c("year", "education + year"),
                             values     = weight,
                             statistics = c("sum", "pct_group"),
                             pct_group  = c("sex", "age", "education", "year"),
                             formats    = list(sex = sex., age = age.,
                                               education = education.),
                             na.rm      = TRUE)

set_style_options(sheet_name = "age_sex")

tab2 <- my_data |> any_table(rows       = "age",
                             columns    = "sex",
                             values     = weight,
                             statistics = "sum",
```

```
                                       formats    = list(sex = sex., age = age.),
                                       na.rm      = TRUE)

set_style_options(sheet_name = "edu_year")

tab3 <- my_data |> any_table(rows       = "education",
                             columns    = "year",
                             values     = weight,
                             statistics = "pct_group",
                             formats    = list(education = education.),
                             na.rm      = TRUE)

# Every of the above tabs is a list, which contains the data table, an unstyled
# workbook and the meta information needed for the individual styling. These
# tabs can be input into the following function, which reads the meta information,
# styles each table individually and combines them as separate sheets into a single workbook.
combine_into_workbook(tab1, tab2, tab3)

# Reset the global options afterwards
set_print(TRUE)
set_output("excel")
```

---

concat   *Concatenate Multiple Variables With Padding*

Description

Concatenate multiple variables inside a data frame into a new variable. An automatic or individual
padding can be applied. The padding character can be chosen freely.

The function can also be used to give a single variable a padding.

Usage

```
concat(
  data_frame,
  ...,
  padding_char = NULL,
  padding_length = NULL,
  padding_right = FALSE
)
```

Arguments

data_frame	A data frame which contains the the variables to concatenate.
...	The names of the variables to concatenate.
padding_char	A single character which will be used to fill up the empty places.
padding_length	A numeric vector containing the individual padding length per variable.

padding_right   FALSE by default. If TRUE insert padding characters on the right side instead
of the left side.

Value

Returns a character vector.

See Also

Other character manipulating functions: [sub_string()](), [remove_blanks()]()

Examples

```
# Example data frame
my_data <- dummy_data(100)

# Concatenate variables as provided
my_data[["id1"]] <- my_data |> concat(household_id, state, age)

# Concatenate variables with leading zeros. Each variable will
# receive an individual padding length according to their
# longest value.
my_data[["id2"]] <- my_data |> concat(household_id, state, age,
                                      padding_char = "0")

# Concatenate variables with individual character and lengths.
my_data[["id2"]] <- my_data |> concat(household_id, state, age,
                                      padding_char   = "_",
                                      padding_length = c(5, 3, 4))

# Padding a single variable in place
my_data[["state"]] <- my_data |> concat(state, padding_char = "0")
```

---

content_report   *Get Detailed Summary About A Data Frame*

Description

Prints a summary of a data frames contents, including details such as variable names, types, unique
values, missings and min/max values. It also tells you the number of observations and variables
present in the data frame, memory usage and the number of duplicate observations.

Usage

```
content_report(data_frame, output = "console", monitor = FALSE)
```

Arguments

`data_frame`	The data frame to get the content information from.
`output`	The following output formats are available: console (default) or text.
`monitor`	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

content_report() is based on the 'SAS' procedure Proc Contents, which provides a summary of global information one one hand like number of observations and variables among many others and on the other hand shows per variable information like type and length.

'R' doesn't store the same information in a data frame like 'SAS', but there are many useful information to get a quick overview of a data frame. With this function you don't need to look at each variable individually. You can simply run it over a data frame and get values for: number of unique values, missing values (absolute and relative), min and max value as well as the top value.

Value

Returns a list containing the global information as well as a data table containing the per variable information.

Examples

```
# Example data frame
my_data <- dummy_data(100)

content_report(my_data)
```

---

convert_arguments   *Convert Function Arguments to Character Vector*

Description

args_to_char(): Converts any argument passed as a single character or symbol as well as character vectors or vector of symbols back as character vector.

dots_to_char(): When you define a function and want the user to be able to pass variable names without the need to have them stored in a vector c() or list() beforehand and without putting the names into quotation marks, you can convert this variable list passed as ... into a character vector.

Note: If the user passes a list of characters it is returned as given.

get_origin_as_char() is a wrapper that allows to retrieve the original contents of the provided variable, whether called directly or nested in multiple function calls, as a character vector.

Usage

```
args_to_char(argument)

dots_to_char(...)

get_origin_as_char(original, substituted)
```

Arguments

argument	Function argument to convert.
...	Used for variable names listed in ... without the need to put them in c() or list().
original	The data frame which contains the columns to be checked.
substituted	The grouping variables which potentially form unique combinations.

Value

Returns a character vector.

Examples

```
# Example function with function parameter
print_vnames <- function(parameter){
    var_names <- args_to_char(substitute(parameter))
    print(var_names)
}

print_vnames(age)
print_vnames("age")
print_vnames(c(age, sex, income, weight))
print_vnames(c("age", "sex", "income", "weight"))

# You can also pass in a character vector, if you have stored variable names elsewhere
var_names <- c("age", "sex", "income", "weight")
print_vnames(var_names)

# If you plan to use the function within other functions, better use get_origin_as_char()
print_vnames <- function(parameter){
    var_names <- get_origin_as_char(parameter, substitute(parameter))
    print(var_names)
}

another_function <- function(parameter){
    print_vnames(parameter)
}

another_function("age")
another_function(c("age", "sex", "income", "weight"))

# Example function with ellipsis
print_vnames <- function(...){
```

```
    var_names <- dots_to_char(...)
    print(var_names)
}

print_vnames(age)
print_vnames("age")
print_vnames(age, sex, income, weight)
print_vnames("age", "sex", "income", "weight")

# You can also pass in a character vector, if you have stored variable names elsewhere
var_names <- c("age", "sex", "income", "weight")
print_vnames(var_names)
```

---

convert_variables   *Convert Variables*

Description

[convert_numeric()](#) converts all given variables to numeric if possible. If a variable contains none numerical values (not including NAs), the variable will not be converted.

[convert_factor()](#) converts all given variables to factor.

Usage

```
convert_numeric(data_frame, variables)

convert_factor(data_frame, variables)
```

Arguments

data_frame	A data frame containing variables to convert.
variables	Variables from the data frame which should be converted.

Value

[convert_numeric()](#) returns the same data frame with converted variables where possible.

[convert_factor()](#) returns the same data frame with converted variables.

Examples

```
# Convert variables in a data frame to numeric where possible
test_df <- data.frame(var_a = c(1, 2, 3, NA, 4, 5),
                      var_b = c(1, 2, "Hello", NA, 4, 5))

convert_df <- test_df |> convert_numeric(c("var_a", "var_b"))

# Convert variables in a data frame to factor
```

```
test_df <- data.frame(var_a = c(1, 2, 3, 4, 5),
                      var_b = c("e", "c", "a", "d", "b"))

convert_df <- test_df |> convert_factor("var_b")
```

---

crosstabs   *Display Cross Table of Two Variables*

Description

[crosstabs()](#) produces a cross table of two variables. Statistics can be weighted sums, unweighted frequencies or different percentages.

Usage

```
crosstabs(
  data_frame,
  rows,
  columns,
  show_total = TRUE,
  statistics = "sum",
  formats = c(),
  by = c(),
  weight = NULL,
  titles = .qol_options[["titles"]],
  footnotes = .qol_options[["footnotes"]],
  style = .qol_options[["excel_style"]],
  output = .qol_options[["output"]],
  na.rm = .qol_options[["na.rm"]],
  print_miss = .qol_options[["print_miss"]],
  print = .qol_options[["print"]],
  monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frame	A data frame in which are the variables to tabulate.
rows	The variable that appears in the table rows.
columns	The variable that appears in the table columns.
show_total	TRUE by default. Whether to print row and column totals or not.
statistics	The user requested statistics.Available functions: • "sum" -> Weighted and unweighted sum • "freq" -> Unweighted frequency • "pct_row" -> Weighted and unweighted row percentages

- "pct_column" -> Weighted and unweighted column percentages
- "pct_total" -> Weighted and unweighted percentages compared to the grand total

formats	A list in which is specified which formats should be applied to which variables.
by	Compute tables stratified by the expressions of the provided variables.
weight	Put in a weight variable to compute weighted results.
titles	Specify one or more table titles.
footnotes	Specify one or more table footnotes.
style	A list of options can be passed to control the appearance of 'Excel' outputs. Styles can be created with excel_output_style().
output	The following output formats are available: console (default), text, excel and excel_nostyle.
na.rm	FALSE by default. If TRUE removes all NA values from the variables.
print_miss	FALSE by default. If TRUE outputs all possible categories of the grouping variables based on the provided formats, even if there are no observations for a combination.
print	TRUE by default. If TRUE prints the output, if FALSE doesn't print anything. Can be used if one only wants to catch the output data frame.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

crosstabs() is based on the 'SAS' procedure Proc Freq, which provides efficient and readable ways to perform cross tabulations.

To create a cross table you only need to provide a variable for the rows and columns. Nothing special about this. The real power comes into play, when you output your tables as a fully styled 'Excel' workbook. Setting up a custom, reusable style is as easy as setting up options like: provide a color for the table header, set the font size for the row header, should borders be drawn for the table cells yes/no, and so on.

You can not only output sums and frequencies, but also different percentages, all set up in separate, evenly designed tables. For just a quick overview, rather than fully designed tables, you can also just output the tables in ASCII style format.

Value

Returns a data tables containing the results for the cross table.

See Also

Creating a custom table style: excel_output_style(), modify_output_style(), number_format_style(), modify_number_formats().

Global style options: set_style_options(), set_variable_labels(), set_stat_labels().

Creating formats: discrete_format() and interval_format().

Functions that can handle formats and styles: frequencies(), any_table().

Additional functions that can handle styles: export_with_style()

Additional functions that can handle formats: summarise_plus(), recode(), recode_multi(), transpose_plus(), sort_plus()

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Define titles and footnotes. If you want to add hyperlinks you can do so by
# adding "link:" followed by the hyperlink to the main text.
set_titles("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")

set_footnotes("This is footnote number 1",
              "This is footnote number 2",
              "This is footnote number 3 link: https://cran.r-project.org/")

# Output cross tables
my_data |> crosstabs(age, sex)
my_data |> crosstabs(age, sex,
                     weight = "weight")

# Also works with characters
my_data |> crosstabs("age", "sex")
my_data |> crosstabs("age", "sex",
                     weight = "weight")

# Applying formats
age. <- discrete_format(
    "Total"         = 0:100,
    "under 18"      = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"   = 65:100)

sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

my_data |> crosstabs(age, sex,
                     formats   = list(age = age., sex = sex.))

# Split cross table by expressions of another variable
my_data |> crosstabs(age, sex, by = education)

# Compute different stats
my_data |> crosstabs(age, sex,
```

```
                        statistics = c("sum", "freq", "pct_row", "pct_column", "pct_total"))

# Get a list with two data tables for further usage
result_list <- my_data |> crosstabs(age, sex,
                                    formats = list(age = age., sex = sex.))

# Output in text file
my_data |> crosstabs(age, sex, output = "text")

# Output to Excel
my_data |> crosstabs(age, sex, output = "excel")

# Individual styling can also be passed directly
my_style <- excel_output_style(header_back_color = "0077B6",
                               font              = "Times New Roman")

my_data |> crosstabs(age, sex, output = "excel", style = my_style)

# To save a table as xlsx file you have to set the path and filename in the
# style element
# Example files paths
table_file <- tempfile(fileext = ".xlsx")

# Note: Normally you would directly input the path ("C:/MyPath/") and name ("MyFile.xlsx").
set_style_options(save_path  = dirname(table_file),
                  file       = basename(table_file),
                  sheet_name = "MyTable")

my_data |> crosstabs(age, sex, output = "excel")

# Manual cleanup for example
unlink(table_file)

# Global options are permanently active until the current R session is closed.
# There are also functions to reset the values manually.
reset_style_options()
reset_qol_options()
close_file()
```

---

drop_type_vars   *Drop automatically generated Variables*

Description

If [summarise_plus()](#) is used with the nested options "all" or "single", three variables are automatically generated: TYPE, TYPE_NR and DEPTH. With this functions these variables are dropped.

Usage

```
drop_type_vars(data_frame)
```

Arguments

data_frame   The data frame with automatically generated variables.

Value

Returns a data frame without the variables TYPE, TYPE_NR and DEPTH.

Examples

```
# Example format
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

# Example data frame
my_data <- dummy_data(1000)

# Call function
all_possible <- my_data |>
    summarise_plus(class      = c(year, sex),
                   values     = c(income, probability),
                   statistics = c("sum", "mean", "freq"),
                   formats    = list(sex = "sex."),
                   weight     = weight,
                   nesting    = "all",
                   na.rm      = TRUE) |>
    drop_type_vars()
```

---

dummy_data   *Dummy Data*

Description

The dummy data frame contains a few randomly generated variables like year, sex, age, income and weight to test out functionalities. It can be generated with the desired number of observations.

Usage

```
dummy_data(
  no_obs = 25000,
  insert_na = TRUE,
  monitor = .qol_options[["monitor"]]
)
```

Arguments

`no_obs`	Number of observations.
`insert_na`	TRUE by default. Inserts random NA values into variables.
`monitor`	FALSE by default. If TRUE outputs two charts to visualize the functions time consumption.

Value

Returns a dummy data table.

Examples

```
my_data <- dummy_data(1000)
```

---

duplicates   *Check For Duplicate Variable Names*

Description

Checks for duplicate variable names in a data frame, e.g. AGE, age and Age.

Counts the number of duplicated variables in a data frame. If a variable appears three times, e.g. AGE, age and Age, the variable count will be one.

Usage

```
get_duplicate_var_names(data_frame)

get_duplicate_var_count(data_frame)
```

Arguments

`data_frame`	The data frame which variable names to check for duplicates.

Value

`get_duplicate_var_names()`: Returns a vector of duplicate variable names.

`get_duplicate_var_count()`: Returns the count of duplicated variables as a numeric value.

Examples

```
# Example data frame
my_data <- data.frame(age    = 1,
                      AGE    = 2,
                      Age    = 3,
                      sex    = 1)

dup_var_names <- my_data |> get_duplicate_var_names()

dup_var_count <- my_data |> get_duplicate_var_count()
```

---

error_handling   *Error Handling*

Description

`resolve_intersection()`: Compares if two vectors have intersecting values. If TRUE, removes the intersection values from the base vector

`part_of_df()`: Check if variable names are part of a data frame. If not, remove them from the given vector.

`remove_doubled_values()`: Remove values from a vector that appear more than once.

`check_weight()`: Check if a weight variable was provided. If TRUE, check whether it can be used else add a temporary weight variable.

Usage

```
resolve_intersection(base, vector_to_check, check_only = FALSE)

part_of_df(data_frame, var_names, check_only = FALSE)

remove_doubled_values(var_names)

check_weight(data_frame, var_names)
```

Arguments

base	The base vector from which to remove any intersecting values.
vector_to_check	
	The vector for which intersections should be checked.
check_only	Returns a list of invalid entries instead of a vector. Additionally it doesn't throw a warning.
data_frame	A data frame in which to look up variable names.
var_names	A character vector of variable names.

Value

Returns a vector or list.

Examples

```
# Resolve intersection between two vectors
vec1 <- c("a", "b", "c", "d")
vec2 <- c("e", "f", "a", "g")

vec1 <- resolve_intersection(vec1, vec2)

# Check if variables are part of a data frame
my_data   <- dummy_data(100)
var_names <- c("year", "state", "age", "test")

var_names <- my_data |> part_of_df(var_names)

# Remove doubled values
var_names <- c("year", "state", "state", "age")

var_names <- remove_doubled_values(var_names)

# Check the provided weight variable
var_names <- my_data |> check_weight("weight")
```

---

excel_output_style   *Style for 'Excel' Table Outputs*

Description

Set different options which define the visual output of 'Excel' tables produced by frequencies(),
crosstabs() and any_table().

Usage

```
excel_output_style(
  save_path = NULL,
  file = NULL,
  sheet_name = "Table",
  font = "Arial",
  column_widths = "auto",
  row_heights = "auto",
  title_heights = NULL,
  header_heights = NULL,
  subheader_heights = NULL,
  table_heights = NULL,
  footnote_heights = NULL,
```

```
start_row = 2,
start_column = 2,
freeze_col_header = FALSE,
freeze_row_header = FALSE,
filters = TRUE,
grid_lines = TRUE,
by_as_subheaders = FALSE,
header_back_color = "FFFFFF",
header_font_color = "000000",
header_font_size = 10,
header_font_bold = TRUE,
header_alignment = "center",
header_wrap = "1",
header_indent = 0,
header_borders = TRUE,
header_border_color = "000000",
subheader_back_color = "FFFFFF",
subheader_font_color = "000000",
subheader_font_size = 10,
subheader_font_bold = TRUE,
subheader_alignment = "center",
subheader_wrap = "1",
subheader_indent = 0,
subheader_borders = TRUE,
subheader_border_color = "000000",
cat_col_back_color = "FFFFFF",
cat_col_font_color = "000000",
cat_col_font_size = 10,
cat_col_font_bold = FALSE,
cat_col_alignment = "left",
cat_col_wrap = "1",
cat_col_indent = 1,
cat_col_borders = TRUE,
cat_col_border_color = "000000",
table_back_color = "FFFFFF",
table_font_color = "000000",
table_font_size = 10,
table_font_bold = FALSE,
table_alignment = "right",
table_indent = 1,
table_borders = FALSE,
table_border_color = "000000",
as_heatmap = FALSE,
heatmap_low_color = "F8696B",
heatmap_middle_color = "FFFFFF",
heatmap_high_color = "63BE7B",
box_back_color = "FFFFFF",
box_font_color = "000000",
```

```
    box_font_size = 10,
    box_font_bold = TRUE,
    box_alignment = "center",
    box_wrap = "1",
    box_indent = 0,
    box_borders = TRUE,
    box_border_color = "000000",
    number_formats = number_format_style(),
    title_font_color = "000000",
    title_font_size = 10,
    title_font_bold = TRUE,
    title_alignment = "left",
    footnote_font_color = "000000",
    footnote_font_size = 8,
    footnote_font_bold = FALSE,
    footnote_alignment = "left",
    na_symbol = "."
)
```

Arguments

save_path	If NULL, opens the output as temporary file. Otherwise specify an output path.
file	If NULL, opens the output as temporary file. Otherwise specify a filename with extension.
sheet_name	Name of the sheet inside the workbook to which the output shall be written. If multiple outputs are produced in one go, the sheet name additionally receives a running number.
font	Set the font to be used for the entire output.
column_widths	Specify whether column widths should be set automatically and individually or if a numeric vector is passed each column width can be specified manually. If a table has more columns than column widths are provided, the last given column width will be repeated until the end of the table.
row_heights	Specify whether row heights should be set automatically and individually or if a numeric vector is passed each row height can be specified manually. If a table has more rows than row heights are provided, the last given row height will be repeated until the end of the table.
title_heights	Set individual row heights for the titles only.
header_heights	Set individual row heights for the table header only.
subheader_heights	
	Set individual row heights for the table subheader only.
table_heights	Set individual row heights for the table body only.
footnote_heights	
	Set individual row heights for the footnotes only.
start_row	The row in which the table starts.
start_column	The column in which the table starts.

freeze_col_header

    Whether to freeze the column header so that it is always visible while scrolling down the document.

freeze_row_header

    Whether to freeze the row header so that it is always visible while scrolling sideways in the document.

filters   Whether to set filters in the column header, when exporting a data frame.

grid_lines   Whether to show grid lines or not.

by_as_subheaders

    Whether to format by variables as subheaders in one table instead of single tables on multiple sheets.

header_back_color

    Background cell color of the table header.

header_font_color

    Font color of the table header.

header_font_size

    Font size of the table header.

header_font_bold

    Whether to print the table header in bold letters.

header_alignment

    Set the text alignment of the table header.

header_wrap   Whether to wrap the texts in the table header.

header_indent Indentation level of the table header.

header_borders Whether to draw borders around the table header cells.

header_border_color

    Borders colors of the table header cells.

subheader_back_color

    Background cell color of the table subheader.

subheader_font_color

    Font color of the table subheader.

subheader_font_size

    Font size of the table subheader.

subheader_font_bold

    Whether to print the table subheader in bold letters.

subheader_alignment

    Set the text alignment of the table subheader.

subheader_wrap Whether to wrap the texts in the table subheader.

subheader_indent

    Indentation level of the table subheader.

subheader_borders

    Whether to draw borders around the table subheader cells.

subheader_border_color

    Borders colors of the table subheader cells.

cat_col_back_color

    Background cell color of the category columns inside the table.

cat_col_font_color
> Font color of the category columns inside the table.

cat_col_font_size
> Font size of the category columns inside the table.

cat_col_font_bold
> Whether to print the category columns inside the table in bold letters.

cat_col_alignment
> Set the text alignment of the category columns inside the table.

cat_col_wrap   Whether to wrap the texts in the category columns inside the table.

cat_col_indent   Indentation level of the category columns inside the table.

cat_col_borders
> Whether to draw borders around the category columns inside the table.

cat_col_border_color
> Borders colors of the category columns inside the table.

table_back_color
> Background color of the inner table cells.

table_font_color
> Font color of the inner table cells.

table_font_size
> Font size of the inner table cells.

table_font_bold
> Whether to print the inner table cells in bold numbers

table_alignment
> Set the text alignment of the inner table cells.

table_indent   Indentation level of the inner table cells.

table_borders   Whether to draw borders around the inner table cells.

table_border_color
> Borders colors of the inner table cells.

as_heatmap   Whether to lay a conditional formatting over the values.

heatmap_low_color
> The color for lower values in the conditional formatting.

heatmap_middle_color
> The color for middle values in the conditional formatting.

heatmap_high_color
> The color for high values in the conditional formatting.

box_back_color   Background color of the left box in table header.

box_font_color   Font color of the left box in table header.

box_font_size   Font size of the left box in table header.

box_font_bold   Whether to print the left box in table header in bold letters.

box_alignment   Set the text alignment of the left box in table header.

box_wrap   Whether to wrap the texts in the left box in table header.

box_indent   Indentation level of the left box in table header.

box_borders   Whether to draw borders around the left box in table header.

box_border_color

Borders colors of the left box in table header.

number_formats Put in a list of number formats which should be assigned to the different stats. Number formats can be created with [number_format_style()](#).

title_font_color

Font color of the titles.

title_font_size

Font size of the tables titles.

title_font_bold

Whether to print the tables titles in bold letters.

title_alignment

Set the text alignment of the titles.

footnote_font_color

Font color of the footnotes

footnote_font_size

Font size of the tables footnotes

footnote_font_bold

Whether to print the tables footnotes in bold letters.

footnote_alignment

Set the text alignment of the footnotes.

na_symbol   Define the symbol that should be used for NA values.

Details

[excel_output_style()](#) is based on the Output Delivery System (ODS) in 'SAS', which provides efficient and readable ways to set up different table styles.

With the output style you have full control over the table design. There is no need to think about calculating the right place to input a background color or a border of a certain type and how to do this in a loop for multiple cells. Just input colors, borders, font styles, etc. for the different table parts and everything else is handled by the functions capable of using styles.

The concept basically is: design over complex calculations.

Value

Returns a list of named style options.

See Also

Creating a custom table style: [modify_output_style()](#), [number_format_style()](#), [modify_number_formats()](#).

Global style options: [set_style_options()](#), [set_variable_labels()](#), [set_stat_labels()](#).

Functions that can handle styles: [frequencies()](#), [crosstabs()](#), [any_table()](#), [export_with_style()](#)

Examples

```
# For default values
excel_style <- excel_output_style()

# Set specific options, the rest will be set to default values
excel_style <- excel_output_style(font      = "Calibri",
                                  sheet_name = "My_Output")

# For cells with no background color pass an empty string
excel_style <- excel_output_style(table_back_color = "")
```

---

expand_formats	*Expand A List Of Formats To Generate All Possible Combinations*

Description

Generates a data frame which contains all nested combinations of the provided format labels.

Usage

```
expand_formats(..., names = NULL)
```

Arguments

...	A list containing format data frames.
names	Custom variable names for the newly generated data frame.

Value

Returns a data frames with all format label combinations.

Examples

```
# Example formats
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

age. <- discrete_format(
    "Total"         = 0:100,
    "under 18"      = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"  = 65:100)

# Expand formats
```

```
expand_df <- expand_formats(sex., age.,
                            names = c("sex", "age"))
```

---

export_with_style   *Export Data Frame With Style*

Description

export_with_style() prints a data frame as an individually styled 'Excel' table. Titles, footnotes and labels for variable names can optionally be added.

Usage

```
export_with_style(
  data_frame,
  titles = .qol_options[["titles"]],
  footnotes = .qol_options[["footnotes"]],
  var_labels = .qol_options[["var_labels"]],
  workbook = NULL,
  style = .qol_options[["excel_style"]],
  output = .qol_options[["output"]],
  print = .qol_options[["print"]],
  monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frame	A data frame to print.
titles	Specify one or more table titles.
footnotes	Specify one or more table footnotes.
var_labels	A list in which is specified which label should be printed for which variable instead of the variable name.
workbook	Insert a previously created workbook to expand the sheets instead of creating a new file.
style	A list of options can be passed to control the appearance of 'Excel' outputs. Styles can be created with excel_output_style().
output	The following output formats are available: excel and excel_nostyle.
print	TRUE by default. If TRUE prints the output, if FALSE doesn't print anything. Can be used if one only wants to catch the output workbook.
monitor	FALSE by default. If TRUE outputs two charts to visualize the functions time consumption.

Details

export_with_style() is based on the 'SAS' procedure Proc Print, which outputs the data frame as is into a styled table.

Value

Returns a formatted 'Excel' workbook.

See Also

Creating a custom table style: excel_output_style(), modify_output_style(), number_format_style(), modify_number_formats().

Global style options: set_style_options(), set_variable_labels(), set_stat_labels().

Functions that can handle styles: frequencies(), crosstabs(), any_table().

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Define style
set_style_options(column_widths = c(2, 15, 15, 15, 9))

# Define titles and footnotes. If you want to add hyperlinks you can do so by
# adding "link:" followed by the hyperlink to the main text.
set_titles("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")

set_footnotes("This is footnote number 1",
              "This is footnote number 2",
              "This is footnote number 3 link: https://cran.r-project.org/")

# Print styled data frame
my_data |> export_with_style()

# Retrieve formatted workbook for further usage
wb <- my_data |> export_with_style()

# To save a table as xlsx file you have to set the path and filename in the
# style element
# Example files paths
table_file <- tempfile(fileext = ".xlsx")

# Note: Normally you would directly input the path ("C:/MyPath/") and name ("MyFile.xlsx").
set_style_options(save_path  = dirname(table_file),
                  file       = basename(table_file),
                  sheet_name = "MyTable")

my_data |> export_with_style()
```

```
# Manual cleanup for example
unlink(table_file)

# Global options are permanently active until the current R session is closed.
# There are also functions to reset the values manually.
reset_style_options()
reset_qol_options()
close_file()
```

---

first_row_as_names   *Set First Data Frame Row As Variable Names*

Description

Sets the first row of a data frame as variable names and deletes it. In case of NA, numeric values or empty characters in the first row, the old names are kept.

Usage

```
first_row_as_names(data_frame)
```

Arguments

data_frame   A data frame for which to set new variable names.

Value

Returns a data frame with renamed variables.

Examples

```
# Example data frame
my_data <- data.frame(
              var1 = c("id", 1, 2, 3),
              var2 = c(NA, "a", "b", "c"),
              var3 = c("value", 1, 2, 3),
              var4 = c("", "a", "b", "c"),
              var5 = c(1, 2, 3, 4))

my_data <- my_data |> first_row_as_names()
```

---

free_memory

Remove Objects From Memory By Name And Type

Description

Remove objects by name or type to free up memory. Uses the base rm() function but provides more flexible ways to remove objects.

Usage

```
free_memory(names = NULL, types = NULL, envir = .GlobalEnv)
```

Arguments

names	Object names to be removed.
types	Object types to be removed.
envir	The environment to remove the objects from.

Details

free_memory() is based on the 'SAS' function Proc Datasets. Among other things this procedure is able to remove datasets from memory. But not only by writing out the full file name, it is capable of looking for datasets starting with, ending with or containing a certain text. Additionally certain object types can be removed.

Value

Returns removed objects.

Examples

```
# Example data frames
my_data1 <- dummy_data(10)
my_data2 <- dummy_data(10)
data     <- dummy_data(10)
file     <- dummy_data(10)

# Free memory by name
free_memory("my_:")
free_memory(c("data", "file"))

# Free by type
my_data1 <- dummy_data(10)
my_data2 <- dummy_data(10)
data     <- dummy_data(10)
file     <- dummy_data(10)

free_memory(type = "data.frame")
```

---

frequencies   *Display Frequency Tables of Single Variables*

Description

[frequencies()](#) produces two kinds of tables for a quick overview of single variables. The first table is for a broader overview and contains mean, sd, min, max, freq and missings. The second table is the actual frequency table which shows the weighted sums, percentages and unweighted frequencies per expression.

Usage

```
frequencies(
  data_frame,
  variables,
  formats = c(),
  by = c(),
  weight = NULL,
  means = FALSE,
  titles = .qol_options[["titles"]],
  footnotes = .qol_options[["footnotes"]],
  style = .qol_options[["excel_style"]],
  output = .qol_options[["output"]],
  na.rm = .qol_options[["na.rm"]],
  print_miss = .qol_options[["print_miss"]],
  print = .qol_options[["print"]],
  monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frame	A data frame in which are the variables to tabulate.
variables	A vector of single variables to create frequency tables for.
formats	A list in which is specified which formats should be applied to which variables.
by	Compute tables stratified by the expressions of the provided variables.
weight	Put in a weight variable to compute weighted results.
means	FALSE by default. If TRUE prints a small summarising table which contains mean, sd, min, max, total freq and missing values.
titles	Specify one or more table titles.
footnotes	Specify one or more table footnotes.
style	A list of options can be passed to control the appearance of 'Excel' outputs. Styles can be created with [excel_output_style()](#).
output	The following output formats are available: console (default), text, excel and excel_nostyle.

na.rm	FALSE by default. If TRUE removes all NA values from the variables.
print_miss	FALSE by default. If TRUE outputs all possible categories of the grouping variables based on the provided formats, even if there are no observations for a combination.
print	TRUE by default. If TRUE prints the output, if FALSE doesn't print anything. Can be used if one only wants to catch the output data frame.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

[frequencies()](#) is based on the 'SAS' procedure Proc Freq, which provides efficient and readable ways to output frequency tables.

To create a frequency table you only need to provide a single variable. Nothing special about this. The real power comes into play, when you output your tables as a fully styled 'Excel' workbook. Setting up a custom, reusable style is as easy as setting up options like: provide a color for the table header, set the font size for the row header, should borders be drawn for the table cells yes/no, and so on.

You also can provide multiple single variables to generate multiple, evenly designed tables, all at once. For just a quick overview, rather than fully designed tables, you can also just output the tables in ASCII style format.

Value

Returns a list of two data tables containing the results for the frequency tables.

See Also

Creating a custom table style: [excel_output_style()](#), [modify_output_style()](#), [number_format_style()](#), [modify_number_formats()](#).

Global style options: [set_style_options()](#), [set_variable_labels()](#), [set_stat_labels()](#).

Creating formats: [discrete_format()](#) and [interval_format()](#).

Functions that can handle formats and styles: [crosstabs()](#), [any_table()](#).

Additional functions that can handle styles: [export_with_style()](#)

Additional functions that can handle formats: [summarise_plus()](#), [recode()](#), [recode_multi()](#), [transpose_plus()](#), [sort_plus()](#)

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Define titles and footnotes. If you want to add hyperlinks you can do so by
# adding "link:" followed by the hyperlink to the main text.
set_titles("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")
```

```
set_footnotes("This is footnote number 1",
              "This is footnote number 2",
              "This is footnote number 3 link: https://cran.r-project.org/")

# Output frequencies tables
my_data |> frequencies(sex)
my_data |> frequencies(c(age, education),
                       weight = weight)

# Also works with characters
my_data |> frequencies("sex")
my_data |> frequencies(c("age", "education"),
                       weight = "weight")

# Applying
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

my_data |> frequencies(sex, formats = list(sex = sex.))

# Split frequencies by expressions of another variable
my_data |> frequencies(sex, by = education)

# Get a list with two data tables for further usage
result_list <- my_data |> frequencies(sex, formats = list(sex = sex.))

# Output in text file
my_data |> frequencies(sex, output = "text")

# Output to Excel
my_data |> frequencies(sex, output = "excel")

# Individual styling can also be passed directly
my_style <- excel_output_style(header_back_color = "0077B6",
                               font              = "Times New Roman")

my_data |> frequencies(sex, output = "excel", style = my_style)

# To save a table as xlsx file you have to set the path and filename in the
# style element
# Example files paths
table_file <- tempfile(fileext = ".xlsx")

# Note: Normally you would directly input the path ("C:/MyPath/") and name ("MyFile.xlsx").
set_style_options(save_path  = dirname(table_file),
                  file       = basename(table_file),
                  sheet_name = "MyTable")

my_data |> frequencies(sex, output = "excel")
```

```
# Manual cleanup for example
unlink(table_file)

# Global options are permanently active until the current R session is closed.
# There are also functions to reset the values manually.
reset_style_options()
reset_qol_options()
close_file()
```

---

fuse_variables   *Fuse Multiple Variables*

Description

When you have a situation where you have multiple variables with different NA values that happen
to be in different places (where one variable has a value the other is NA and vice versa) you can
fuse these together to a single variable.

Usage

```
fuse_variables(
  data_frame,
  new_variable_name,
  variables_to_fuse,
  drop_original_vars = TRUE
)
```

Arguments

data_frame   A data frame with variables to fuse.

new_variable_name
                  The name of the new fused variable.

variables_to_fuse
                  A vector with the variables that should be fused together.

drop_original_vars
                  Whether to drop or keep the original values. TRUE by default.

Value

Returns a data frame without the variables TYPE, TYPE_NR and DEPTH.

Examples

```
# Example format
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
```

```
      "Female" = 2)

# Example data frame
my_data <- dummy_data(1000)

# Call function
all_possible <- my_data |>
    summarise_plus(class     = c(year, sex),
                   values    = c(income, probability),
                   statistics = c("sum", "mean", "freq"),
                   formats   = list(sex = "sex."),
                   weight    = weight,
                   nesting   = "all",
                   na.rm     = TRUE)

all_possible <- all_possible[DEPTH <= 1] |>
    fuse_variables("fusion", c("year", "sex"))

# NOTE: You can generally use this function to fuse variables. What is done in
#       multiple steps above can be achieved by just using nested = "single" in
#       summarise_plus.
single <- my_data |>
    summarise_plus(class     = c(year, sex),
                   values    = c(income, probability),
                   statistics = c("sum", "mean", "freq"),
                   formats   = list(sex = "sex."),
                   weight    = weight,
                   nesting   = "single",
                   na.rm     = TRUE)
```

---

get_excel_range   *Converts Numbers into 'Excel' Ranges*

Description

Converts a column number into the according letter to form a cell reference like it is used in 'Excel' (e.g "A1"). Also can compute a range from cell to cell (e.g. "A1:BY22").

Usage

```
get_excel_range(
  row = NULL,
  column = NULL,
  from_row = NULL,
  from_column = NULL,
  to_row = NULL,
  to_column = NULL
)
```

Arguments

`row`	Single row number.
`column`	Single column number.
`from_row`	Range start row.
`from_column`	Range start column.
`to_row`	Range end row.
`to_column`	Range end column.

Value

Returns a character with an 'Excel' range.

Examples

```
single_cell <- get_excel_range(row = 1, column = 6)
range       <- get_excel_range(from_row = 1, from_column = 6,
                               to_row = 5,   to_column = 35)
```

`get_integer_length`   *Get Integer Length*

Description

Get the number of digits of an integer variable.

Usage

```
get_integer_length(variable)
```

Arguments

`variable`	The integer variable from which to get the length.

Value

Returns a vector with the number of digits places.

Examples

```
# Example data frame
my_data <- dummy_data(100)

my_data[["age_length"]] <- get_integer_length(my_data[["age"]])
```

---

import_export   *High Level Import From And Export To CSV And XLSX*

Description

import_data(): A wrapper for data.table::fread() and openxlsx2::wb_to_df(), providing basic import functionality with minimal code.

import_multi(): Runs multiple imports on all provided files. Is able to import all sheets from xlsx files.

export_data(): A wrapper for data.table::fwrite() and openxlsx2::wb_save(), providing basic export functionality with minimal code.

export_multi(): Runs multiple exports on a list of data frames. Is able to export to a single xlsx file with multiple sheets.

Usage

```
import_data(
  infile,
  sheet = 1,
  region = NULL,
  separator = "auto",
  decimal = "auto",
  var_names = TRUE
)

import_multi(
  file_list,
  sheet = "all",
  region = NULL,
  separator = "auto",
  decimal = "auto",
  var_names = TRUE
)

export_data(
  data_frame,
  outfile,
  separator = ";",
  decimal = ",",
  var_names = TRUE
)

export_multi(
  file_list,
  out_path,
  into_sheets = TRUE,
```

```
    separator = NULL,
    decimal = ",",
    var_names = TRUE
)
```

Arguments

`infile`	Full file path with extension to a csv or xlsx file to be imported.
`sheet`	Only used in xlsx import. Which sheet of the workbook to import.
`region`	Only used in xlsx import. Can either be an 'Excel' range like 'A1:BY27' or the name of a named region.
`separator`	Only used in CSV-export. Defines the single character value separator.
`decimal`	Only used in CSV-export. Defines the single character decimal character.
`var_names`	TRUE by default. Whether to export variable names or not.
`file_list`	`import_multi()`: A character vector containing full file paths.
	`export_multi()`: A list of data frames.
`data_frame`	A data frame to export.
`outfile`	Full file path with extension. Allowed extensions are ".csv" and ".xlsx".
`out_path`	The file path where to save the exported files.
`into_sheets`	TRUE by default. Whether to export all data frames into multiple sheets of a single xlsx file or into separate xlsx files.

Details

`import_data()` and `export_data()` are based on the 'SAS' procedures Proc Import and Proc Export, which provide a very straight forward syntax. While 'SAS' can import many different formats with these procedures, these 'R' versions concentrate on importing CSV and XLSX files.

The main goal here is to just provide as few as possible parameters to tackle most of the imports and exports. These error handling also tries to let an import and export happen, even though a parameter wasn't provided in the correct way.

Value

Returns a data frame.

Multi functions: Returns a list of data frames.

See Also

Functions that can export with style: `frequencies()`, `crosstabs()`, `any_table()`, `export_with_style()`.

Creating a custom table style: `excel_output_style()`, `modify_output_style()`, `number_format_style()`, `modify_number_formats()`.

Global style options: `set_style_options()`, `set_variable_labels()`, `set_stat_labels()`.

Examples

```
# Example files
csv_file  <- system.file("extdata", "qol_example_data.csv",  package = "qol")
xlsx_file <- system.file("extdata", "qol_example_data.xlsx", package = "qol")

# Import: Provide full file path
my_csv  <- import_data(csv_file)
my_xlsx <- import_data(xlsx_file)

# Import specific regions
range_import <- import_data(xlsx_file, region = "B4:H32")
name_import  <- import_data(xlsx_file, region = "test_region")

# Import from another sheet
sheet_import <- import_data(xlsx_file, sheet = "Sheet 2")

# Import multiple files at once
all_files <- import_multi(c(csv_file, xlsx_file))

# Example data frame
my_data <- dummy_data(100)

# Example export file paths
export_csv  <- tempfile(fileext = ".csv")
export_xlsx <- tempfile(fileext = ".xlsx")

# Export: Provide full file path
my_data |> export_data(export_csv)
my_data |> export_data(export_xlsx)

# Example data frame list
my_list <- list(first  = dummy_data(10),
                second = dummy_data(10))

# Export multiple data frames into one xlsx file
# with multiple sheets
export_multi(my_list, tempdir())

# Export multiple data frames into multiple xlsx files
export_multi(my_list, tempdir(), into_sheets = FALSE)

# Export multiple data frames into multiple csv files
export_multi(my_list, tempdir(), separator = ";")

# Manual cleanup for example
file1 <- file.path(tempdir(), "my_list.xlsx")
file2 <- file.path(tempdir(), "first.xlsx")
file3 <- file.path(tempdir(), "second.xlsx")
file4 <- file.path(tempdir(), "first.csv")
file5 <- file.path(tempdir(), "second.csv")

unlink(c(export_csv, export_xlsx,
```

```
        file1, file2, file3, file4, file5))
```

---

inverse   *Get Variable Names Which Are Not Part Of The Given Vector*

Description

If you have stored variable names inside a character vector, this function gives you the inverse
variable name vector.

Usage

```
inverse(data_frame, var_names)
```

Arguments

data_frame	The data frame from which to take the variable names.
var_names	A character vector of variable names.

Value

Returns the inverse vector of variable names compared to the given vector.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Get variable names
var_names <- c("year", "age", "sex")
other_names <- my_data |> inverse(var_names)

# Can also be used to just get all variable names
all_names <- my_data |> inverse()
```

---

libname   *Check If Path Exists And Retrieve Files*

Description

[libname()](#) checks if a given path exists and writes a message in the console accordingly. Optional all files from the given path can be retrieved as a named character vector.

Usage

```
libname(path, get_files = FALSE)
```

Arguments

path	A folder path.
get_files	FALSE by default. If TRUE returns a named character vector containing file paths.

Value

Returns the given file path or a named character vector containing file paths.

Examples

```
my_path   <- libname("C:/My_Path/")
file_list <- libname("C:/My_Path/", get_files = TRUE)
```

---

macro   *Resolve Macro Variables In A Text*

Description

Variables written with a leading "&" inside a text will be resolved into the character or numeric expression the corresponding object is holding.

Usage

```
macro(text)

apply_macro(character_vector)
```

Arguments

text	A text containing macro variables to resolve.
character_vector	
	A vector containing multiple character expressions in which to resolve macro variables.

Details

Macro variables in 'SAS' can be set up with %Let and can act as global accessible variables. This in itself is nothing special to 'R' because basically every object created outside a function is a global variable.

To use these global objects within a text one has to use e.g. paste0("The current year is: ", year). With the macro function one can write it like this: macro("The current year is &year").

So where is the benefit? If implemented within a function, a parameter like "title" or "footnote" in the tabulation functions, can resolve these variables without the need of another function. You can just pass the character expression "The current year is &year" and the implementation inside the function resolves the macro variable directly in place.

Value

[macro()](): Returns a character.

[apply_macro()](): Returns a character vector.

See Also

Functions that can make use of macro variables: [any_table()](), [frequencies()](), [crosstabs()](), [export_with_style()](), [summarise_plus()](), [transpose_plus()]()

Within the tabulation functions titles, footnotes, var_labels and stat_labels can resolve macros. Additionally they can be used in the [any_table()]() rows and columns parameter, in the [summarise_plus()]() types parameter and in the [transpose_plus()]() pivot parameter.

Examples

```
# Resolving macro variable in single character
year <- 2026

text <- macro("The current year is &year")

# You can also combine multiple macro variables
some_variable <- "current"
current2026   <- "The current year is"

text_combi <- macro("&&some_variable&year &year")

# Resolving macro variable in character vector
char_vector <- c("The current year is &year",
                 "The &some_variable year is &year",
                 "&&some_variable&year &year")

text_vector <- apply_macro(char_vector)
```

---

modify_number_formats *Modify Number Formats Used by* any_table()

Description

Modify previously created number formats with number_format_style().

Usage

```
modify_number_formats(formats_to_modify, ...)
```

Arguments

formats_to_modify

Pre created number formats where only certain elements should be modified while the rest is kept as is.

... Pass in names and corresponding new values for existing number formats.

Details

modify_number_formats() is based on 'SAS' number formats and the Output Delivery System (ODS), which provides efficient and readable ways to set up different table styles.

With the number format style you have full control over formatting numbers according to the different statistics. There is no need to think about calculating the right place to input the number formats and how to do this in a loop for multiple cells. Just input the different number formats and decimals for the different statistics and everything else is handled by the functions capable of using number styles.

The concept basically is: design over complex calculations.

Value

Returns a modified list of number format options.

See Also

Creating a custom table style: excel_output_style(), modify_output_style(), number_format_style().

Global style options: set_style_options(), set_variable_labels(), set_stat_labels().

Functions that can handle styles: frequencies(), crosstabs(), any_table(), export_with_style().

Examples

```
# For default values
format_list <- number_format_style(pct_excel    = "0.00000000",
                                    pct_decimals = 8)

# Set specific options, the rest will be kept as is
format_list <- format_list |> modify_number_formats(sum_excel = "#,###,##0.000")
```

```
# IMPORTANT: Don't forget to add individual formats to an excel style, otherwise
# they won't come into affect.
excel_style <- excel_output_style(number_formats = format_list)
```

---

modify_output_style   *Modify Style for 'Excel' Table Outputs*

Description

Modify a previously created style with [excel_output_style()](#).

Usage

```
modify_output_style(style_to_modify, ...)
```

Arguments

style_to_modify

> A pre created style where only certain elements should be modified while the rest is kept as is.

...   Pass in names and corresponding new values for existing style elements.

Details

[modify_output_style()](#) is based on the Output Delivery System (ODS) in 'SAS', which provides efficient and readable ways to set up different table styles.

With the output style you have full control over the table design. There is no need to think about calculating the right place to input a background color or a border of a certain type and how to do this in a loop for multiple cells. Just input colors, borders, font styles, etc. for the different table parts and everything else is handled by the functions capable of using styles.

The concept basically is: design over complex calculations.

Value

Returns a modified list of named style options.

See Also

Creating a custom table style: [excel_output_style()](#), [number_format_style()](#), [modify_number_formats()](#).

Global style options: [set_style_options()](#), [set_variable_labels()](#), [set_stat_labels()](#).

Functions that can handle styles: [frequencies()](#), [crosstabs()](#), [any_table()](#), [export_with_style()](#)

Examples

```
# For default values
excel_style <- excel_output_style()

# Set specific options, the rest will be kept as is
excel_style <- excel_style |> modify_output_style(sheet_name      = "Sheet",
                                                  title_font_bold = FALSE)

# For cells with no background color pass an empty string
excel_style <- excel_style |> modify_output_style(table_back_color = "")
```

---

multi_join	*Join Multiple Data Frames In One Go*

Description

Join two or more data frames together in one operation. `multi_join()` can handle multiple different join methods and can join on differently named variables.

Usage

```
multi_join(
  data_frames,
  on,
  how = "left",
  keep_indicators = FALSE,
  monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frames	A list of data frames to join together. The second and all following data frames will be joined on the first one.
on	The key variables on which the data frames should be joined. If a character vector is provided, the function assumes all the variables are in every data frame. To join on different variable names a list of character vectors has to be provided.
how	A character vector containing the join method names. Available methods are: left, right, inner, full, outer, left_inner and right_inner.
keep_indicators	
	FALSE by default. If TRUE, a variable for each data frame is created, which indicates whether a data frame provides values.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

[multi_join()](#) is based on the 'SAS' Data-Step function Merge. Merge is capable of joining multiple data sets together at once, with a very basic syntax.

Provide the dataset names, the variables, on which they should be joined and after a full join is complete, the user can decide which parts of the joins should remain in the final dataset.

[multi_join()](#) tries to keep the simplicity, while giving the user the power, to do more joins at the same time. Additionally to what Merge can do, this function also makes use of the Proc SQL possibility to join datasets on different variable names.

Value

Returns a single data frame with joined variables from all given data frames.

Examples

```
# Example data frames
df1 <- data.frame(key = c(1, 1, 1, 2, 2, 2),
                  a   = c("a", "a", "a", "a", "a", "a"))

df2 <- data.frame(key = c(2, 3),
                  b   = c("b", "b"))

# See all different joins in action
join_methods <- c("left", "right", "inner", "full", "outer", "left_inner", "right_inner")
joined_data  <- list()

for (method in seq_along(join_methods)){
    joined_data[[method]] <- multi_join(list(df1, df2),
                                        on  = "key",
                                        how = join_methods[[method]])
}

# Left join on more than one key
df1b <- data.frame(key1 = c(1, 1, 1, 2, 2, 2),
                   key2 = c("a", "a", "a", "a", "a", "a"),
                   a    = c("a", "a", "a", "a", "a", "a"))

df2b <- data.frame(key1 = c(2, 3),
                   key2 = c("a", "a"),
                   b    = c("b", "b"))

left_joined <- multi_join(list(df1b, df2b), on = c("key1", "key2"))

# Join more than two data frames
df3 <- data.frame(key = c(1, 2),
                  c   = c("c", "c"))

multiple_joined <- multi_join(list(df1, df2, df3), on = "key")

# You can also use different methods for each join
multiple_joined2 <- multi_join(list(df1, df3, df2),
```

```
                                          on  = "key",
                                          how = c("left", "right"))

       # Joining on different variable names
       df1c <- data.frame(key1 = c(1, 1, 1, 2, 2, 2),
                          key2 = c("a", "a", "a", "a", "a", "a"),
                          a    = c("a", "a", "a", "a", "a", "a"))

       df2c <- data.frame(var1 = c(2, 3),
                          var2 = c("a", "a"),
                          b    = c("b", "b"))

       df3c <- data.frame(any  = c(1, 2),
                          name = c("a", "a"),
                          c    = c("c", "c"))

       multiple_joined3 <- multi_join(list(df1c, df2c, df3c),
                                      on = list(df1c = c("key1", "key2"),
                                                df2c = c("var1", "var2"),
                                                df3c = c("any", "name")))
```

---

number_format_style   *Number Formats Used by* [any_table()](#)

Description

Set individual number formats for the different statistics in tables produced with [any_table()](#).

Usage

```
number_format_style(
  pct_excel = "0.0",
  freq_excel = "#,###,##0",
  freq.g0_excel = "#,###,##0",
  sum_excel = "#,###,##0",
  sum.wgt_excel = "#,###,##0",
  mean_excel = "#,###,##0",
  median_excel = "#,###,##0",
  mode_excel = "#,###,##0",
  min_excel = "#,###,##0",
  max_excel = "#,###,##0",
  sd_excel = "#,###,##0.000",
  variance_excel = "#,###,##0.000",
  first_excel = "#,###,##0",
  last_excel = "#,###,##0",
  p_excel = "#,###,##0",
  missing_excel = "#,###,##0",
  pct_decimals = 1,
```

```
    freq_decimals = 0,
    freq.g0_decimals = 0,
    sum_decimals = 3,
    sum.wgt_decimals = 3,
    mean_decimals = 2,
    median_decimals = 2,
    mode_decimals = 2,
    min_decimals = 2,
    max_decimals = 2,
    sd_decimals = 3,
    variance_decimals = 3,
    first_decimals = 0,
    last_decimals = 0,
    p_decimals = 2,
    missing_decimals = 0
)
```

Arguments

`pct_excel`	Number format for percentage applied in Excel workbook.
`freq_excel`	Number format for frequency applied in Excel workbook.
`freq.g0_excel`	Number format for frequency greater zero applied in Excel workbook.
`sum_excel`	Number format for sum applied in Excel workbook.
`sum.wgt_excel`	Number format for sum of weights applied in Excel workbook.
`mean_excel`	Number format for mean applied in Excel workbook.
`median_excel`	Number format for median applied in Excel workbook.
`mode_excel`	Number format for mode applied in Excel workbook.
`min_excel`	Number format for min applied in Excel workbook.
`max_excel`	Number format for max applied in Excel workbook.
`sd_excel`	Number format for sd applied in Excel workbook.
`variance_excel`	Number format for variance applied in Excel workbook.
`first_excel`	Number format for first applied in Excel workbook.
`last_excel`	Number format for last applied in Excel workbook.
`p_excel`	Number format for percentile applied in Excel workbook.
`missing_excel`	Number format for missing applied in Excel workbook.
`pct_decimals`	Number of decimals for percentage.
`freq_decimals`	Number of decimals for frequency.
`freq.g0_decimals`	
	Number of decimals for frequency greater zero.
`sum_decimals`	Number of decimals for sum.
`sum.wgt_decimals`	
	Number of decimals for sum of weights.
`mean_decimals`	Number of decimals for mean.

```
median_decimals
                Number of decimals for median.
mode_decimals   Number of decimals for mode.
min_decimals    Number of decimals for min.
max_decimals    Number of decimals for max.
sd_decimals     Number of decimals for sd.
variance_decimals
                Number of decimals for variance.
first_decimals  Number of decimals for first.
last_decimals   Number of decimals for last.
p_decimals      Number of decimals for percentile.
missing_decimals
                Number of decimals for missing.
```

Details

[number_format_style()](#) is based on 'SAS' number formats and the Output Delivery System (ODS), which provides efficient and readable ways to set up different table styles.

With the number format style you have full control over formatting numbers according to the different statistics. There is no need to think about calculating the right place to input the number formats and how to do this in a loop for multiple cells. Just input the different number formats and decimals for the different statistics and everything else is handled by the functions capable of using number styles.

The concept basically is: design over complex calculations.

Value

Returns a list of named number format options.

See Also

Creating a custom table style: [excel_output_style()](#), [modify_output_style()](#), [modify_number_formats()](#).

Global style options: [set_style_options()](#), [set_variable_labels()](#), [set_stat_labels()](#).

Functions that can handle styles: [frequencies()](#), [crosstabs()](#), [any_table()](#), [export_with_style()](#)

Examples

```
# For default values
format_list <- number_format_style()

# Set specific options, the rest will be set to default values
format_list <- number_format_style(pct_excel    = "0.00000000",
                                    pct_decimals = 8)

# IMPORTANT: Don't forget to add individual formats to an excel style, otherwise
# they won't come into affect.
excel_style <- excel_output_style(number_formats = format_list)
```

---

qol_news   *Go To GitHub NEWS Page*

Description

Opens browser and goes to the Github NEWS page

Usage

```
qol_news()
```

Value

URL.

---

qol_options   *Set Global Print Option*

Description

set_print(): Set the print option globally for the tabulation and export to Excel functions.

get_print(): Get the globally stored print option.

set_monitor(): Set the monitor option globally for the heavier functions which are able to show how they work internally.

get_monitor(): Get the globally stored monitor option.

set_na.rm(): Set the na.rm option globally for each function which can remove NA values.

get_na.rm(): Get the globally stored na.rm option.

set_print_miss(): Set the print_miss option globally for each function which can display missing categories.

get_print_miss(): Get the globally stored print_miss option.

set_output(): Set the output option globally for each function that can output results to "console", "text", "excel" or "excel_nostyle".

get_output(): Get the globally stored output option.

set_titles(): Set the titles globally for each function that can print titles above the output table.

get_titles(): Get the globally stored titles.

set_footnotes(): Set the footnotes globally for each function that can print footnotes above the output table.

get_footnotes(): Get the globally stored footnotes.

Usage

```
set_print(...)

get_print()

set_monitor(...)

get_monitor()

set_na.rm(...)

get_na.rm()

set_print_miss(...)

get_print_miss()

set_output(...)

get_output()

set_titles(...)

get_titles()

set_footnotes(...)

get_footnotes()
```

Arguments

...   [set_footnotes()](): Put in the footnotes that should appear below tables.

Value

[set_print()](): Changed global print option.

[get_print()](): TRUE or FALSE.

[set_monitor()](): Changed global monitor option.

[get_monitor()](): TRUE or FALSE.

[set_na.rm()](): Changed global na.rm option.

[get_na.rm()](): TRUE or FALSE.

[set_print_miss()](): Changed global print_miss option.

[get_print_miss()](): TRUE or FALSE.

[set_output()](): Changed global output option.

[get_output()](): Current output option as character.

set_titles(): Changed global titles.

get_titles(): Current titles as character.

set_footnotes(): Changed global footnotes.

get_footnotes(): Current footnotes as character.

Examples

```
set_print(FALSE)
set_print(TRUE)

get_print()

set_monitor(TRUE)
set_monitor(FALSE)

get_monitor()

set_na.rm(TRUE)
set_na.rm(FALSE)

get_na.rm()

set_print_miss(TRUE)
set_print_miss(FALSE)

get_print_miss()

set_output("excel")

get_output()

set_titles("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")

get_titles()

set_footnotes("This is title number 1 link: https://cran.r-project.org/",
           "This is title number 2",
           "This is title number 3")

get_footnotes()
```

---

recode   *Recode New Variables With Formats*

Description

Instead of writing multiple if-clauses to recode values into a new variable, you can use formats to recode a variable into a new one.

Usage

```
recode(data_frame, ...)

recode_multi(data_frame, ...)
```

Arguments

data_frame   A data frame which contains the the original variables to recode.

...   recode() Pass in the original variable name that should be recoded along with
               the corresponding format container in the form: variable = format.

               In recode_multi() multiple variables can be recoded in one go and multilabels
               can be applied. This overwrites the original variables and duplicates rows if
               multilabels are applied. In occasions were you want to use format containers
               to afterwards perform operations with other packages, you can make use of this
               principle with this function.

Details

recode() is based on the 'SAS' function put(), which provides an efficient and readable way, to
generate new variables with the help of formats.

When creating a format you can basically write code like you think: This new category consists of
these original values. And after that you just apply these new categories to the original values to
create a new variable. No need for multiple if_else statements.

Value

recode(): Returns a vector with recoded values.

recode_multi(): Returns a data frame with the newly recoded variable.

See Also

Creating formats: discrete_format() and interval_format().

Functions that also make use of formats: frequencies(), crosstabs(), any_table(), summarise_plus(),
transpose_plus(), sort_plus()

Examples

```
# Example formats
age. <- discrete_format(
    "under 18"      = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
```

```
        "65 and older"   = 65:100)

# Example data frame
my_data <- dummy_data(1000)

# Call function
my_data[["age_group1"]] <- my_data |> recode(age = age.)

# Formats can also be passed as characters
my_data[["age_group2"]] <- my_data |> recode(age = "age.")

# Multilabel recode
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

income. <- interval_format(
    "Total"              = 0:99999,
    "below 500"          = 0:499,
    "500 to under 1000"  = 500:999,
    "1000 to under 2000" = 1000:1999,
    "2000 and more"      = 2000:99999)

# recode_multi() can not only apply multiple recodings, but it can also
# apply multilabels.
# NOTE: Recoding will always be in place. When applying multilabels the
#       result data frame will have more observations than before.
multi_data <- my_data |> recode_multi(sex = sex., income = income.)
```

---

remove_blanks   *Remove Blanks*

Description

Removes leading and trailing blanks or both. Can also remove all blanks from a character or normalize multiple blanks to single ones.

Usage

```
remove_blanks(data_frame, variable, which = "all")
```

Arguments

data_frame	A data frame which contains the character variables from which blanks should be removed.
variable	Variable name of the one from which to remove blanks.
which	"all" by default. Can be "leading", "trailing", "trim", "normalize" or "all". Determines which blanks should be removed

Value

Returns a character vector with removed blanks.

See Also

Other character manipulating functions: concat(), sub_string()

Examples

```
# Example data frame
my_data <- dummy_data(100)
my_data[["blanks"]] <- " This  is  a  test "

# Remove blanks
my_data[["leading"]]   <- my_data |> remove_blanks(blanks, which = "leading")
my_data[["trailing"]] <- my_data |> remove_blanks(blanks, which = "trailing")
my_data[["trim"]]      <- my_data |> remove_blanks(blanks, which = "trim")
my_data[["all"]]       <- my_data |> remove_blanks(blanks, which = "all")
my_data[["normalize"]] <- my_data |> remove_blanks(blanks, which = "normalize")
```

---

remove_stat_extension   *Replace Statistic From Variable Names*

Description

Remove the statistic name from variable names, so that they get back their old names without extension.

Usage

```
remove_stat_extension(data_frame, statistics)
```

Arguments

data_frame	The data frame in which there are variables to be renamed.
statistics	Statistic extensions that should be removed from the variable names.

Value

Returns a data frame with renamed variables.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Summarise data
all_nested <- my_data |>
    summarise_plus(class      = c(year, sex),
                   values     = c(weight, income),
                   statistics = c("sum", "pct_group", "pct_total", "sum_wgt", "freq"),
                   weight     = weight,
                   nesting    = "deepest",
                   na.rm      = TRUE)

# Remove statistic extension
new_names <- all_nested |> remove_stat_extension("sum")
```

---

rename_multi   *Rename One Or More Variables*

Description

Can rename one or more existing variable names into the corresponding new variable names in one
go.

Usage

```
rename_multi(data_frame, ...)
```

Arguments

data_frame   The data frame which contains the variable names to be renamed.

...   Pass in variables to be renamed in the form: "old_var" = "new_var".

Value

Returns a data_frame with renamed variables.

Examples

```
# Example data frame
my_data <- dummy_data(10)

# Rename multiple variables at once
new_names_df <- my_data |> rename_multi(sex   = var1,
                                        age   = var2,
                                        state = var3)

# Also works with variable names in quotation marks
```

```
new_names_df <- my_data |> rename_multi("sex"   = "var1",
                                        "age"   = "var2",
                                        "state" = "var3")
```

---

rename_pattern   *Replace Patterns Inside Variable Names*

Description

Replace a certain pattern inside a variable name with a new one. This can be used if there are multiple different variable names which have a pattern in common (e.g. all end in "_sum" but start different), so that there don't have to be multiple rename variable calls.

Usage

```
rename_pattern(data_frame, old_pattern, new_pattern)
```

Arguments

data_frame	The data frame in which there are variables to be renamed.
old_pattern	The pattern which should be replaced in the variable names.
new_pattern	The pattern which should be set in place for the old one.

Value

Returns a data frame with renamed variables.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Summarise data
all_nested <- my_data |>
    summarise_plus(class      = c(year, sex),
                   values     = c(weight, income),
                   statistics = c("sum", "pct_group", "pct_total", "sum_wgt", "freq"),
                   weight     = weight,
                   nesting    = "deepest",
                   na.rm      = TRUE)

# Rename variables by repacing patterns
new_names <- all_nested |>
    rename_pattern("pct", "percent") |>
    rename_pattern("_sum", "")
```

---

replace_except   *Replace Patterns While Protecting Exceptions*

Description

Replaces a provided pattern with another, while protecting exceptions. Exceptions can contain the given pattern, but won't be changed during replacement.

Usage

```
replace_except(vector, pattern, replacement, exceptions = NULL)
```

Arguments

vector   A vector containing the texts, where a pattern should be replaced.

pattern   The pattern that should be replaced.

replacement   The new pattern, which replaces the old one.

exceptions   A character vector containing exceptions, which should not be altered.

Value

Returns a vector with replaced pattern.

Examples

```
# Vector, where underscores should be replaced
underscores <- c("my_variable", "var_with_underscores", "var_sum", "var_pct_total")

# Extensions, where underscores shouldn't be replaced
extensions <- c("_sum", "_pct_group", "_pct_total", "_pct_value", "_pct", "_freq_g0",
                "_freq", "_mean", "_median", "_mode", "_min", "_max", "_first",
                "_last", "_p1", "_p2", "_p3", "_p4", "_p5", "_p6", "_p7", "_p8", "_p9",
                "sum_wgt", "_sd", "_variance", "_missing")

# Replace
new_vector <- underscores |> replace_except("_", ".", extensions)
```

---

round_values   *Round Values With Half Rounded Up*

Description

This function rounds values according to DIN 1333 (round half up).

Usage

```
round_values(values, digits = 0)
```

Arguments

values	Numeric values to round.
digits	The number of decimal places the values should be rounded to.

Value

Returns rounded values

Examples

```
round_numbers1 <- round_values(c(-0.5, -0.4, 0.1, 0.49, 0.5, 1.5, 2.5, 3.2))
round_numbers2 <- round_values(c(-0.5, -0.49, 0.17, 0.499, 0.51, 1.549, 2.51, 3.25),
                               digits = 1)
```

---

set   *Stack Multiple Data Frames*

Description

Stacks multiple data frames and matches column names.

Usage

```
set(..., id = FALSE, compress = NULL, guessing_rows = 100)
```

Arguments

...	Put in multiple data frames to stack them in the provided order.
id	Adds an ID column to indicate the different data frames.
compress	No compression by default. If compression receives any value, set() will convert character variables to numeric or integer where possible. If set to "factor" all non numeric character variables will be converted to factors.
guessing_rows	100 by default. set() takes a sample of rows to determine of which type variables are.

Value

Returns a stacked data frame.

Examples

```
# Example data frames
my_data1 <- dummy_data(100)
my_data2 <- dummy_data(100)
my_data3 <- dummy_data(100)
my_data4 <- dummy_data(100)
my_data5 <- dummy_data(100)

# Stack data frames
stacked_df <- set(my_data1,
                  my_data2,
                  my_data3,
                  my_data4,
                  my_data5)
```

---

setcolorder_by_pattern
                              Order Columns by Variable Name Patterns

---

Description

Order variables in a data frame based on a pattern rather than whole variable names. E.g. grab every variable that contains "sum" in it's name and order them together so that they appear next to each other.

Usage

```
setcolorder_by_pattern(data_frame, pattern)
```

Arguments

data_frame	The data frame to be ordered.
pattern	The pattern which is used for ordering the data frame columns.

Value

Returns a reordered data frame with the ordered variables at the end.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Summarise data
all_nested <- my_data |>
    summarise_plus(class      = c(year, sex),
                   values     = c(weight, income),
                   statistics = c("sum", "pct_group", "pct_total", "sum_wgt", "freq"),
                   weight     = weight,
                   nesting    = "deepest",
                   na.rm      = TRUE)

# Set a different column order
new_order <- all_nested |> setcolorder_by_pattern(c("pct", "freq", "sum"))
```

---

sort_plus   *Sort Data Frame Rows With Some Additions*

Description

Sort data frame rows by the provided variables. sort_plus() is also able to preserve the current order of certain variables and only sort other variables within this order. As another option one can sort a variable with the help of formats, which can be used to e.g. sort a character variable in another than alphabetical order without creating a temporary variable just for sorting.

Usage

```
sort_plus(
  data_frame,
  by,
  preserve = NULL,
  order = "ascending",
  formats = c(),
  na.last = TRUE
)
```

Arguments

data_frame	A data frame to summarise.
by	A variable vector which contains the variables to sort by.
preserve	A vector containing all variables which current order should be preserved.
order	A vector containing the sorting order for each variable. 'ascending'/'a' or 'descending'/'d' can be used. If there are less orders given than by variables provided, the last given sorting order will be used for the additional by variables.

formats	A list in which is specified which formats should be used to sort certain vari-ables.
na.last	TRUE by default. Specifies whether NA values should come last or first.

Details

[sort_plus()](#) is just very loosely based on the 'SAS' procedure Proc Sort. It tries to keep the simplicity, but with some added features.

Value

Returns a sorted data table.

See Also

Creating formats: [discrete_format()](#) and [interval_format()](#).

Functions that also make use of formats: [frequencies()](#), [crosstabs()](#), [any_table()](#), [recode()](#), [recode_multi()](#), [transpose_plus()](#).

Examples

```
# Example formats
education. <- discrete_format(
    "1" = "low",
    "2" = "middle",
    "3" = "high")

# Example data frame
my_data <- dummy_data(1000)

# Simple sorting
sort_df1 <- my_data |> sort_plus(by = c(state, sex, age))
sort_df2 <- my_data |> sort_plus(by    = c(state, sex, age),
                                 order = c("ascending", "descending"))

# Character variables will normally be sorted alphabetically. With the help
# of a format this variable can be sorted in a completely different way.
sort_df3 <- my_data |> sort_plus(by      = education,
                                 formats = list(education = education.))

# Preserve the order of the character variable, otherwise it couldn't stay in
# it's current order.
sort_df4 <- sort_df3 |> sort_plus(by       = age,
                                  preserve = education)
```

---

split_by   *Split Data Frame By Variable Expressions Or Condition*

Description

Split up a data frame based on variable expressions or on conditions to receive multiple smaller data frames. Both possibilities can be used at the same time.

Usage

```
split_by(
  data_frame,
  ...,
  formats = list(),
  inverse = FALSE,
  monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frame	A data frame which should be split up into multiple data frames.
...	Pass in one or multiple variables and/or conditions on which the provided data frame should be splitted.
formats	A list in which is specified which formats should be applied to which variables.
inverse	Uses the inverse conditions to split up the data frame.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

split_by() is based on the explicit Output from 'SAS'. With the Output function one can - among other things - explicitly tell 'SAS' which observation to output into which data set. Which enables the user to output one observation into one or multiple data sets.

Instead of subsetting the same data frame multiple times manually, you can subset it multiple times at once with this function.

Value

Returns a list of data frames split by variable expressions and/or conditions. The lists names are the variable expressions or conditions.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Split by variable expressions
split_var_df <- my_data |> split_by(sex)

# Split by conditions
split_cond_df <- my_data |> split_by(sex == 1 & age <  18,
                                     sex == 2 & age >= 18)

# Split by condition with inverse group
split_inv_df <- my_data |> split_by(sex == 1, inverse = TRUE)

# Split by variables and conditions
split_combi_df <- my_data |> split_by(state, education,
                                      sex == 1, age < 18)

# Split by variable expressions using formats
state. <- discrete_format(
    ”Germany”                        = 1:16,
    ”Schleswig-Holstein”             = 1,
    ”Hamburg”                        = 2,
    ”Lower Saxony”                   = 3,
    ”Bremen”                         = 4,
    ”North Rhine-Westphalia”         = 5,
    ”Hesse”                          = 6,
    ”Rhineland-Palatinate”           = 7,
    ”Baden-Württemberg”              = 8,
    ”Bavaria”                        = 9,
    ”Saarland”                       = 10,
    ”West”                           = 1:10,
    ”Berlin”                         = 11,
    ”Brandenburg”                    = 12,
    ”Mecklenburg-Western Pomerania” = 13,
    ”Saxony”                         = 14,
    ”Saxony-Anhalt”                  = 15,
    ”Thuringia”                      = 16,
    ”East”                           = 11:16)

split_format_df <- my_data |> split_by(state,
                                       formats = list(state = state.))
```

---

style_options　　　　　　　　*Set Global Styling Options For Excel Workbooks*

Description

Modify Styling options for Excel workbooks. Available parameters can be seen in excel_output_style()
or number_format_style().

`set_style_options()` sets the styling options for Excel workbooks globally.These options are used by all tabulation and output functions, which are capable of exporting styled outputs.

`reset_style_options()` resets global style options to the default parameters.

`get_style_options()` prints out the currently set global styling options.

`close_file()` is a simple, more readable wrapper for setting file parameter to NULL.

`set_variable_labels()`: Can set variable labels globally so that they don't have to be provided in every output function separately.

`get_variable_labels()`: Get the globally stored variable labels.

`set_stat_labels()`: Can set statistic labels globally so that they don't have to be provided in every output function separately.

`get_stat_labels()`: Get the globally stored statistic labels.

`reset_qol_options()` resets global options to the default parameters.

Usage

```
set_style_options(...)

reset_style_options()

get_style_options()

close_file()

set_variable_labels(...)

get_variable_labels()

set_stat_labels(...)

get_stat_labels()

reset_qol_options()
```

Arguments

```
...                     set_stat_labels(): Put in the statistics and their respective labels.
```

Value

`set_style_options()`: Returns modified global styling options.

`reset_style_options()`: Returns default global styling options.

`get_style_options()`: List of global styling options.

`close_file()`: List of global styling options with file = NULL.

`set_variable_labels()`: List of variable labels.

`get_variable_labels()`: List of variable labels.

[set_stat_labels()](): List of statistic labels.

[get_stat_labels()](): List of statistic labels.

[reset_qol_options()](): Returns default global options.

See Also

Functions that use global styling options: [any_table()](), [frequencies()](), [crosstabs()]().

Functions that also use global variable labels: [export_with_style()]().

Functions that use global variable and statistic labels: [any_table()](), [frequencies()](), [crosstabs()]().

Functions that also use global variable labels: [export_with_style()]().

Examples

```
set_style_options(save_path    = "C:/My Projects/",
                  sum_decimals = 8)

reset_style_options()

get_style_options()

close_file()

set_variable_labels(age_gr = "Group of ages",
                    status = "Current status")

get_variable_labels()

set_stat_labels(pct  = "%",
                freq = "Count")

get_stat_labels()

reset_qol_options()
```

---

sub_string   *Retrieve A Substring From A Character*

Description

[sub_string()]() can extract parts of a character from the left side, right side or from the middle. It is also able to start or end at specific letter sequences instead of positions.

Usage

```
sub_string(data_frame, variable, from = NULL, to = NULL, case_sensitive = TRUE)
```

Arguments

`data_frame`	A data frame which contains the variables to concatenate.
`variable`	A character variable to extract parts from.
`from`	The names of the variables to concatenate.
`to`	A single character which will be used to fill up the empty places.
`case_sensitive`	TRUE by default. When a character expression is passed as from or to it makes a difference whether a letter is written in upper or lower case. Pass FALSE to handle upper and lower case equaly.

Value

Returns parts of a character vector.

See Also

Other character manipulating functions: [concat()](#), [remove_blanks()](#)

Examples

```
# Example data frame
my_data <- dummy_data(100)

# Extract text from the left
my_data[["left"]] <- my_data |> sub_string(education, to = 2)

# Extract text from the right
my_data[["right"]] <- my_data |> sub_string(education, from = 2)

# Extract text from the middle
my_data[["middle"]] <- my_data |> sub_string(education, from = 2, to = 3)

# Find text and extract from the left
my_data[["left2"]] <- my_data |> sub_string(education, to = "l")

# Find text and extract from the right
my_data[["right2"]] <- my_data |> sub_string(education, from = "l")

# Find text and extract from the middle
my_data[["middle2"]] <- my_data |> sub_string(education, from = "i", to = "l")
```

---

summarise_plus *Fast And Powerful Yet Simple To Use Summarise*

Description

summarise_plus() creates a new aggregated data table with the desired grouping. It can output only the deepest nested combination of the grouping variables (default) or you can also output every possible combination of the grouping variables at once, with just one small change. Besides the normal summary functions like sum, mean or median, you can also calculate their respective weighted version by just setting a weight variable.

Usage

```
summarise_plus(
  data_frame,
  class = NULL,
  values,
  statistics = c("sum", "freq"),
  formats = list(),
  types = "",
  weight = NULL,
  nesting = "deepest",
  merge_back = FALSE,
  na.rm = .qol_options[["na.rm"]],
  print_miss = .qol_options[["print_miss"]],
  monitor = .qol_options[["monitor"]],
  notes = TRUE
)
```

Arguments

data_frame	A data frame to summarise.
class	A vector containing all grouping variables.
values	A vector containing all variables that should be summarised.
statistics	Available functions:

- "sum" -> Weighted and unweighted sum
- "sum_wgt" -> Sum of all weights
- "freq" -> Unweighted frequency
- "freq_g0" -> Unweighted frequency of all values greater than zero
- "pct_group" -> Weighted and unweighted percentages within the respective group
- "pct_total" -> Weighted and unweighted percentages compared to the grand total
- "mean" -> Weighted and unweighted mean
- "median" -> Weighted and unweighted median
- "mode" -> Weighted and unweighted mode
- "min" -> Minimum
- "max" -> Maximum
- "sd" -> Weighted and unweighted standard deviation

- "variance" -> Weighted and unweighted standard variance
- "first" -> First value
- "last" -> Last value
- "pn" -> Weighted and unweighted percentiles (any p1, p2, p3, ... possible)
- "missing" -> Missings generated by the value variables

formats
: A list in which is specified which formats should be applied to which class variables.

types
: A character vector specifying the different combinations of group variables which should be computed when using nesting = "all". If left empty all possible combinations will be computed.

weight
: Put in a weight variable to compute weighted results.

nesting
: The predefined value is "deepest" meaning that only the fully nested version of all class variables will be computed. If set to "all", all possible combinations will be computed in one data table. The option "single" only outputs the ungrouped summary of all class variables in one data table.

merge_back
: Newly summarised variables can be merged back to the original data frame if TRUE. Only works if nested = "deepest and no formats are defined.

na.rm
: FALSE by default. If TRUE removes all NA values from the class variables.

print_miss
: FALSE by default. If TRUE outputs all possible categories of the grouping variables based on the provided formats, even if there are no observations for a combination.

monitor
: FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

notes
: TRUE by default. Prints notifications about NA values produced by class variables during summarise.

Details

summarise_plus() is based on the 'SAS' procedure Proc Summary, which provides efficient and readable ways to perform complex aggregations.

Normally you would compute new categorical variables beforehand - probably even in different forms, if you wanted to have different categorizations - and bloat up the data set. After all this recoding footwork you could finally use multiple summaries to compute all the stats you need to then put them back together. With this function this is no more necessary.

In summarise_plus() you put in the original data frame and let the recoding happen via format containers. This is very efficient, since new variables and categories are only created just before the summarise happens.

Additionally you can specify whether you only want to produce the all nested version of all group variables or whether you want to produce every possible combination in one go. All with a single option.

The function is optimized to always take the fastest route, depending on the options specified.

Value

Returns a summarised data table.

See Also

Creating formats: discrete_format() and interval_format().

Functions that also make use of formats: frequencies(), crosstabs(), any_table(), recode(), recode_multi(), transpose_plus(), sort_plus().

Examples

```
# Example formats
age. <- discrete_format(
    "Total"          = 0:100,
    "under 18"       = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"   = 65:100)

sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

income. <- interval_format(
    "Total"             = 0:99999,
    "below 500"         = 0:499,
    "500 to under 1000"  = 500:999,
    "1000 to under 2000" = 1000:1999,
    "2000 and more"      = 2000:99999)

# Example data frame
my_data <- dummy_data(1000)

# Call function
all_nested <- my_data |>
    summarise_plus(class      = c(year, sex, age),
                   values     = income,
                   statistics = c("sum", "pct_group", "pct_total", "sum_wgt", "freq"),
                   formats    = list(sex = sex., age = age.),
                   weight     = weight,
                   nesting    = "deepest",
                   na.rm      = TRUE)

all_possible <- my_data |>
    summarise_plus(class      = c(year, sex, age, income),
                   values     = c(probability),
                   statistics = c("sum", "p1", "p99", "min", "max", "freq", "freq_g0"),
                   formats    = list(sex    = sex.,
                                     age    = age.,
                                     income = income.),
                   weight     = weight,
                   nesting    = "all",
                   na.rm      = TRUE)
```

```
# Formats can also be passed as characters
single <- my_data |>
    summarise_plus(class      = c(year, age, sex),
                   values     = weight,
                   statistics = c("sum", "mean"),
                   formats    = list(sex = "sex.", age = "age."),
                   nesting    = "single")

merge_back <- my_data |>
    summarise_plus(class      = c(year, age, sex),
                   values     = weight,
                   statistics = c("sum", "mean"),
                   nesting    = "deepest",
                   merge_back = TRUE)

certain_types <- my_data |>
    summarise_plus(class      = c(year, sex, age),
                   values     = c(probability),
                   statistics = c("sum", "mean", "freq"),
                   formats    = list(sex = sex.,
                                     age = age.),
                   types      = c("year", "year + age", "age + sex"),
                   weight     = weight,
                   nesting    = "all",
                   na.rm      = TRUE)
```

transpose_plus	*Fast And Powerful Yet Simple To Use Transpose*

Description

[transpose_plus()](#) is able to reshape a data frame from long to wide and from wide to long. In the long to wide transposition variables can be nested or placed side by side. With the wide to long transposition it is also possible to transpose multiple variables at once.

Additionally [transpose_plus()](#) is able to weight results before transposing them from long to wide.

The function also makes use of formats, which means you don't need to create variables storing the new variable names before transposition. You can just use formats to name the new variables and with multilabels you can even generate new variable expressions at the same time.

Usage

```
transpose_plus(
  data_frame,
  preserve = NULL,
  pivot,
  values = NULL,
```

```
    formats = c(),
    weight = NULL,
    na.rm = .qol_options[["na.rm"]],
    monitor = .qol_options[["monitor"]]
)
```

Arguments

data_frame	A data frame to transpose
preserve	Variables to keep and preserve in their current form.
pivot	A vector that provides the expressions of single variables or od variable combinations that should be transposed. To nest variables use the form: "var1 + var2 + var3 + ...".
values	A vector containing all value variables that should be transposed.
formats	A list in which is specified which formats should be applied to which variables.
weight	Put in a weight variable to compute weighted results.
na.rm	FALSE by default. If TRUE removes all NA values from the preserve and pivot variables.
monitor	FALSE by default. If TRUE, outputs two charts to visualize the functions time consumption.

Details

[transpose_plus()](#) is just very loosely based on the 'SAS' procedure Proc Transpose, and the possibilities of a Data-Step transposition using loops.

The transposition methods 'SAS' has to offer are actually fairly weak. Which is weird because all tools are there to have another powerful function. So [transpose_plus()](#) tries to create the function 'SAS' should have.

The function is able to interpret which transposition direction the user wants by just looking at what the user provided with the function parameters. For a long to wide transposition it is natural to just provide variables to transpose. While it is also just natural to provide new variable names when transposing from wide to long. That alone reduces the number of parameters the user has to enter to perform a simple transposition.

The real magic happens when formats come into play. With their help you can not only name new variables or their expressions, but you can also generate completely new expressions with no effort, just with the help of multilabels.

Value

Returns a transposed data table.

See Also

Creating formats: [discrete_format()](#) and [interval_format()](#).

Functions that also make use of formats: [frequencies()](#), [crosstabs()](#), [any_table()](#), [recode()](#), [recode_multi()](#), [sort_plus()](#).

Examples

```
# Example formats
age. <- discrete_format(
    "Total"         = 0:100,
    "under 18"      = 0:17,
    "18 to under 25" = 18:24,
    "25 to under 55" = 25:54,
    "55 to under 65" = 55:64,
    "65 and older"  = 65:100)

sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

sex2. <- discrete_format(
    "Total"  = c("Male", "Female"),
    "Male"   = "Male",
    "Female" = "Female")

income. <- interval_format(
    "Total"            = 0:99999,
    "below 500"        = 0:499,
    "500 to under 1000"  = 500:999,
    "1000 to under 2000" = 1000:1999,
    "2000 and more"    = 2000:99999)

# Example data frame
my_data <- dummy_data(1000)

# Transpose from long to wide and use a multilabel to generate additional categories
long_to_wide <- my_data |>
    transpose_plus(preserve = c(year, age),
                   pivot   = c("sex", "education"),
                   values  = income,
                   formats = list(sex = sex., age = age.),
                   weight  = weight,
                   na.rm   = TRUE)

# Transpose back from wide to long
wide_to_long <- long_to_wide |>
    transpose_plus(preserve = c(year, age),
                   pivot   = list(sex       = c("Total", "Male", "Female"),
                                  education = c("low", "middle", "high")))

# Nesting variables in long to wide transposition
nested <- my_data |>
    transpose_plus(preserve = c(year, age),
                   pivot   = "sex + education",
                   values  = income,
                   formats = list(sex = sex., age = age.),
                   weight  = weight,
```

```
                          na.rm   = TRUE)

# Or both, nested and un-nested, at the same time
both <- my_data |>
    transpose_plus(preserve = c(year, age),
                   pivot   = c("sex + education", "sex", "education"),
                   values  = income,
                   formats = list(sex = sex., age = age.),
                   weight  = weight,
                   na.rm   = TRUE)
```

---

vars_between   *Get All Variable Names Between Two Variables*

Description

Get all the variable names inside a data frame between two variables (including the provided ones) as a character vector

Usage

```
vars_between(data_frame, from, to)
```

Arguments

data_frame	The data frame from which to take the variable names.
from	Starting variable of variable range.
to	Ending variable of variable range.

Value

Returns a character vector of variable names.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# Get variable names
var_names <- my_data |> vars_between(state, income)

# Get variable names in reverse order
vars_reverse <- my_data |> vars_between(income, state)

# If you only provide "from" or "to" you get all variable names from a point to
# the end or from the beginning to a given point.
vars_from <- my_data |> vars_between(state)
vars_to   <- my_data |> vars_between(to = state)
```

```
# Or just get all variable names
vars_all <- my_data |> vars_between()
```

---

where.   *Filter Data Frame With Direct View*

Description

Filter observations and variables and directly view the result on screen.

Usage

```
where.(data_frame, condition = NULL, keep = NULL)
```

Arguments

data_frame	A data frame on which to apply filters.
condition	The condition on which to filter observations.
keep	The Variables to keep in the result data frame.

Value

Returns a filtered data frame.

Examples

```
# Example data frame
my_data <- dummy_data(1000)

# G       et a quick filtered view
my_data |> where.(sex == 1 & age < 25,
                  c(sex, age, household_id, education))
```

Index