Package 'qol'

November 20, 2025

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

Version 1.1.0

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 7.3.2

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 2025-11-20 21:20:09 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(1000)

# 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")
```

---

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 = c("sum"),
  pct_group = c(),
  pct_value = list(),
  formats = list(),
  by = c(),
  weight = NULL,
```

```
  order_by = "stats",
  titles = c(),
  footnotes = c(),
  var_labels = list(),
  stat_labels = list(),
  box = "",
  workbook = NULL,
  style = excel_output_style(),
  output = "excel",
  pre_summed = FALSE,
  na.rm = FALSE,
  print = TRUE,
  monitor = FALSE
)
```

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 • "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.

`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.

`pre_summed` FALSE by default. If TRUE this function works with pre summarised data. This can be used, if not all the needed results can be calculated by `any_table()` and need to be prepared in advance. Enabling you to still make use of the styled tabulation. For this to work, the values have to carry the statistic extension (e.g. "_sum", "_pct") in the variable name.

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

`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()`.

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()`

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
my_style <- excel_output_style(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.
titles <- c("This is title number 1 link: https://cran.r-project.org/",
            "This is title number 2",
            "This is title number 3")
footnotes <- c("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", "education", "year"),
                     formats    = list(sex = sex., age = age.,
                                       education = education.),
                     style      = my_style,
                     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.),
                     style      = my_style,
                     na.rm      = TRUE)

# Percentages based on value variables instead of categories
my_data |> any_table(rows       = c("age + year"),
                     columns    = c("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)
```

```
# Customize the visual appearance by adding titles, footnotes and variable
# and statistic labels.
# 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     = c("sex"),
                     values      = weight,
                     statistics  = c("sum", "pct_group"),
                     order_by    = "interleaved",
                     formats     = list(sex = sex., age = age.),
                     titles      = titles,
                     footnotes   = footnotes,
                     var_labels  = list(age = "Age categories",
                                        sex = "", weight = ""),
                     stat_labels = list(pct = "%"),
                     style       = my_style,
                     na.rm       = TRUE)

# With individual styling
my_style <- my_style |> modify_output_style(header_back_color = "0077B6",
                                            font              = "Times New Roman")

my_data |> any_table(rows       = c("age + year"),
                     columns    = c("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       = c("age"),
                     columns    = c("sex"),
                     values     = weight,
                     statistics = c("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       = c("education"),
                     columns    = c("year"),
                     values     = weight,
```

```
                    statistics = c("pct_group"),
                    formats    = list(education = education.),
                    style      = my_style,
                    na.rm      = TRUE)

# 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    = c("education + year"),
                    values     = weight,
                    by         = state,
                    statistics = c("sum", "pct_group"),
                    pct_group  = c("education"),
                    formats    = list(sex = sex., age = age., state = state.,
                                      education = education.),
                    titles     = titles,
                    footnotes  = footnotes,
                    style      = my_style,
                    na.rm      = TRUE)
```

`args_to_char` *Convert Ellipsis to Character Vector*

Description

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.

Usage

```
args_to_char(...)
```

Arguments

...	Used for variable names listed in ... without the need to put them in c() or list()

Value

Returns a character vector

Examples

```
# Example function
print_vnames <- function(...){
    var_names <- args_to_char(...)
    print(var_names)
}

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)
```

`build_master` *Build a Master Script From Folder*

Description

`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",
  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.
`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

```
build_master(dir = "C:/My Projects/Code", master_name = "Master Script")
```

`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 function
build_rstheme("C:/My Path/", "My theme",
              editor_background = "#417291",
              editor_headline   = "#602BCA",
              editor_font       = "#C75C48")
```

`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
my_style <- excel_output_style(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.
titles <- c("This is title number 1 link: https://cran.r-project.org/",
            "This is title number 2",
            "This is title number 3")
footnotes <- c("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.
my_style <- my_style |> modify_output_style(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.),
                             style      = my_style,
                             na.rm      = TRUE,
```

```
                              print      = FALSE,
                              output     = "excel_nostyle")

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

tab2 <- my_data |> any_table(rows       = c("age"),
                              columns    = c("sex"),
                              values     = weight,
                              statistics = c("sum"),
                              formats    = list(sex = sex., age = age.),
                              style      = my_style,
                              na.rm      = TRUE,
                              print      = FALSE,
                              output     = "excel_nostyle")

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

tab3 <- my_data |> any_table(rows       = c("education"),
                              columns    = c("year"),
                              values     = weight,
                              statistics = c("pct_group"),
                              formats    = list(education = education.),
                              style      = my_style,
                              na.rm      = TRUE,
                              print      = FALSE,
                              output     = "excel_nostyle")

# 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)
```

`convert_numeric` *Check and Convert to Numeric*

Description

`is_numeric()` checks whether all values of the given variable, that are not NA, are numerical.

`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.

Usage

```
is_numeric(variable)

convert_numeric(data_frame, variables)
```

Arguments

variable	A vector with values to check.
data_frame	A data frame containing variables to convert.
variables	Variables from the data frame which should be converted to numeric.

Value

is_numeric() returns TRUE if all none NA values are numerical, otherwise FALSE.

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

Examples

```
# Check if vectors contain only numeric values
test_vector1 <- c(1, 2, 3, NA, 4, 5)
test_vector2 <- c(1, 2, "Hello", NA, 4, 5)

numeric_check1 <- is_numeric(test_vector1)
numeric_check2 <- is_numeric(test_vector2)

# 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"))
```

---

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,
  statistics = c("sum"),
  formats = c(),
  by = c(),
  weight = NULL,
  titles = c(),
  footnotes = c(),
  style = excel_output_style(),
```

```
  output = "console",
  na.rm = FALSE,
  print = TRUE,
  monitor = FALSE
)
```

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.
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	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().

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()

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.
titles <- c("This is title number 1 link: https://cran.r-project.org/",
            "This is title number 2",
            "This is title number 3")
footnotes <- c("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 and titles
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.),
                     titles    = titles,
                     footnotes = footnotes)

# 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")

# With individual styling
my_style <- excel_output_style(header_back_color = "0077B6",
                               font              = "Times New Roman")

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

`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, monitor = FALSE)
```

Arguments

`no_obs`	Number of observations.
`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)
```

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(
  file = NULL,
  sheet_name = "Table",
  font = "Arial",
  column_widths = "auto",
  row_heights = "auto",
  title_heights = NULL,
  header_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,
  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",
  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",
  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

file	If NULL, opens the output as temporary file. If a filename with path is specified, saves the output to the specified path.
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.
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.

`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.

`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.

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().

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 = "")
```

---

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 = c(),
  footnotes = c(),
  var_labels = list(),
  workbook = NULL,
```

```
  style = excel_output_style(),
  output = "excel",
  print = TRUE,
  monitor = FALSE
)
```

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().

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

Examples

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

# Define style
my_style <- excel_output_style(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.
titles <- c("This is title number 1 link: https://cran.r-project.org/",
            "This is title number 2",
            "This is title number 3")
footnotes <- c("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(titles    = titles,
                             footnotes = footnotes,
                             style     = my_style)

# Retrieve formatted workbook for further usage
wb <- my_data |>
    export_with_style(titles    = titles,
                      footnotes = footnotes,
                      style     = my_style)
```

`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,
  titles = c(),
  footnotes = c(),
  style = excel_output_style(),
  output = "console",
  na.rm = FALSE,
  print = TRUE,
  monitor = FALSE
)
```

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.
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	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().

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()

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.
titles <- c("This is title number 1 link: https://cran.r-project.org/",
            "This is title number 2",
            "This is title number 3")
footnotes <- c("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 formats and titles
sex. <- discrete_format(
    "Total"  = 1:2,
    "Male"   = 1,
    "Female" = 2)

my_data |> frequencies(sex, formats(sex = sex.),
                       titles    = titles,
                       footnotes = footnotes)

# 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(sex = sex.))

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

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

# With individual styling
my_style <- excel_output_style(header_back_color = "0077B6",
                               font              = "Times New Roman")

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

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)
```

`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(NULL)
all_names <- my_data |> inverse(character(0))
```

`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()`.

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()`.

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 = "")
```

---

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().

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)
```

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, new_var, ...)

recode_multi(data_frame, ...)
```

Arguments

data_frame	A data frame which contains the the original variables to recode.
new_var	The name of the newly created and recoded variable.
...	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

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().

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 <- my_data |> recode("age_group1", age = age.)

# Formats can also be passed as characters
my_data <- my_data |> recode("age_group2", 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)

multi_data <- my_data |> recode_multi(sex = sex., income = income.)
```

---

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_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 hsould 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)
```

`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"))
```

---

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.

Usage

```
split_by_var(data_frame, variable)

split_by_condition(data_frame, ..., inverse = FALSE)
```

Arguments

data_frame	A data frame which should be split up into multiple data frames.
variable	In split_by_var() pass in a variable name which expressions are used for splitting up the data frame.
...	In split_by_condition() pass in one or multiple conditions on which the provided data frame should be splitted.
inverse	In split_by_condition() if only one condition is provided, the data frame can be split into two parts. The second returned data frame will be the inverse group of the first.

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

split_by_var(): Returns a list of data frames split by variable expressions. The lists names are the variable expressions.

split_by_condition(): Returns a list of data frames split conditionally. The lists names are the conditions.

Examples

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

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

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

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

---

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 = c(),
  types = c(),
  weight = NULL,
  nesting = "deepest",
  merge_back = FALSE,
  na.rm = FALSE,
  monitor = FALSE,
  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.

`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().

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)
```

Index