Package 'xlr'

March 7, 2025

Title Create Table Summaries and Export Neat Tables to 'Excel'

Version 1.0.3

Description A high-level interface for creating and exporting summary tables to 'Excel'. Built on 'dplyr' and 'openxlsx', it provides tools for generating one-way to n-way tables, and summarizing multiple response questions and question blocks. Tables are exported with native 'Excel' formatting, including titles, footnotes, and basic styling options.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Collate as_base_r.R xlr_table.R xlr_to_workbook.R build_multiple_response_table.R build_table.R write_xlsx.R xlr_numeric.R xlr_integer.R xlr_vector.R xlr_percent.R xlr_format.R openxlsx_utils.R xlr_doc.R error_utils.R create_table_of_contents.R build_question_block_table.R table_utils.R data.R is_xlr_type.R

Suggests knitr, rmarkdown, testthat (>= 3.0.0), data.table, lubridate

Config/testthat/edition 3

Imports rlang, vctrs (>= 0.6.0), haven, openxlsx, methods, cli, dplyr, tibble, pillar, tidyr, tidyselect

Depends R (>= 4.1.0)

LazyData true

VignetteBuilder knitr

URL https://nhilder.github.io/xlr/, https://github.com/NHilder/xlr

BugReports https://github.com/NHilder/xlr/issues

NeedsCompilation no

Author Nicholas Hilderson [aut, cre, cph]

Maintainer Nicholas Hilderson <nhilderson.code@gmail.com>

Repository CRAN

Date/Publication 2025-03-07 11:40:02 UTC

Contents

`as_base_r` *Convert xlr types to their base R type*

Description

`as_base_r` converts xlr objects, xlr_table, xlr_numeric, xlr_integer, xlr_percent, and xlr_format to their base R type.

Usage

```
as_base_r(x)
```

Arguments

x	a xlr object

Details

as_base_r is a generic. It is a wrapper around vec_data but will convert every object to its base type.

Value

The base type of the base R object.

Examples

```
library(xlr)

# We create a xlr objects
a <- xlr_numeric(1:100)
b <- xlr_percent(1:100/100)
tab <- xlr_table(mtcars,"a title","a footnote")

# now lets convert them back to their base types
as_base_r(a)
as_base_r(b)
as_base_r(tab)
```

`build_mtable` *Summarise a multiple response table*

Description

This function can take one or two multiple response responses and generate a summary table with them. You can also cut these columns by other categorical columns by specify the cols parameter.

Usage

```
build_mtable(
  x,
  mcols,
  cols = NULL,
  table_title = "",
  use_questions = FALSE,
  use_NA = FALSE,
  wt = NULL,
  footnote = ""
)
```

Arguments

`x`	a data frame or tidy object.
`mcols`	the column(s) that are multiple response questions. See the `Details` for more details of how these columns should be structured.
`cols`	the column(s) that we want to calculate the sum/percentage of and the multiple response question.
`table_title`	the title of the table sheet
`use_questions`	if the data has column labels (was a imported .sav) file, convert the column label to a footnote with the question.
`use_NA`	logical. whether to include `NA` values in the table. For more complicated `NA` processing post creation, we recommend using filter.

wt	Specify a weighting variable, if NULL no weight is applied.
footnote	optional parameter to pass a custom footnote to the question, this parameter overwrites use_questions.

Details

A multiple response response is a series of columns with a single unique response that stores survey data where a respondent may have chosen multiple options. This function works if this data is stored in a **wide** format. To have a valid multiple response column all the columns should start with the same text, and each contain a unique value. That is it has the form:

```
 data.frame(multi_col_1 = c(1,NA,1),
            multi_col_2 = c(1,1,1),
            multi_col_3 = c(NA,NA,1)
 )
#>   multi_col_1 multi_col_2 multi_col_3
#> 1           1           1          NA
#> 2          NA           1          NA
#> 3           1           1           1
```

This is how popular survey platforms such as Qualtrics output this data type. If your data is long, you will need to pivot the data before hand, we recommend using pivot_wider.

By default this function converts labelled to a xlr_vector by default (and underlying it is a `character()` type).

This function and its family (build_table, build_qtable) is designed to work with data with columns of type `haven::labelled`, which is the default format of data read with `haven::read_sav`/has the format of `.sav`. `.sav` is the default file function type of data from SPSS and can be exported from popular survey providers such as Qualtrics. When you read in data with `haven::read_sav` it imports data with the questions, labels for the response options etc.

See labelled and read_sav if you would like more details on the importing type.

Value

a `xlr_table` object. Use write_xlsx to write to an `Excel` file. See xlr_table for more information.

Examples

```
library(xlr)
library(dplyr)

# You can use this function to calculate the number of people that have
# responded to the question `What is your favourite colour`
build_mtable(clothes_opinions,
             "Q2",
             table_title = "What is your favourite colour?")

# The function also lets you to see the number of NA questions (this is
# where someone doesn't answer any option)
build_mtable(clothes_opinions,
```

```
              "Q2",
              table_title = "What is your favourite colour?",
              use_NA = TRUE)

# You can also cut all questions in the multiple response functions by another
# column
build_mtable(clothes_opinions,
              "Q2",
              gender2,
              table_title = "Your favourite colour by gender")

# By setting `use_questions=TRUE` then the footnote will be the questions
# labels. This is useful to see what the question is.
# The function will try to pull out this based on the question label, and
# will manipulate try and get the correct label.
build_mtable(clothes_opinions,
              "Q2",
              gender2,
              table_title = "Your favourite colour by gender",
              use_questions = TRUE)

# It is common for your data to include 'other' responses in a multiple
# response column. You should remove the column before running build_mtable
clothes_opinions |>
  select(-Q3_other) |>
  build_mtable("Q3")

# You can also specify up to a maxium of two different multiple response
# columns.
clothes_opinions |>
  select(-Q3_other) |>
  build_mtable(c("Q2", "Q3"))

# These cam also be cut by other columns.
clothes_opinions |>
  select(-Q3_other) |>
  build_mtable(c("Q2", "Q3"),
               gender2)

# This function also supports weights and manual footnotes
clothes_opinions |>
  select(-Q3_other) |>
  build_mtable(c("Q2", "Q3"),
               gender2,
               wt = weight,
               footnote = "This is an example footnote.")
```

---

build_qtable *Summarise a question block*

Description

This function helps analyse a block of questions or matrix questions into a single table. It also lets the user cut these questions by other questions in the data. The block of questions mush have the same response options.

Usage

```
build_qtable(
  x,
  block_cols,
  cols = NULL,
  table_title = "",
  use_questions = FALSE,
  use_NA = FALSE,
  wt = NULL,
  footnote = ""
)
```

Arguments

x	a data frame or tidy object
block_cols	<tidyr_tidy_select> statement. These are the columns that make up the question block, they must have the same response option. Most question block columns start with the same piece of text, so you should use `starts_with('column_text')`. See the Examples below.
cols	<tidyr_tidy_select> statement. These are the column(s) that we want to cut the questions in the question block by.
table_title	a string. The title of the table sheet
use_questions	a logical. If the data has column labels (was a imported .sav) file, convert the column label to a footnote with the question.
use_NA	a logical. Whether to include `NA` values in the table. For more complicated `NA` processing post creation, we recommend using filter.
wt	a quoted or unquote column name. Specify a weighting variable, if `NULL` no weight is applied.
footnote	a character vector. Optional parameter to pass a custom footnote to the question, this parameter overwrites `use_questions`.

Details

This function and its family (build_table, build_qtable) is designed to work with data with columns of type `haven::labelled`, which is the default format of data read with `haven::read_sav`/has the format of `.sav`. `.sav` is the default file function type of data from `SPSS` and can be exported from popular survey providers such as Qualtrics. When you read in data with `haven::read_sav` it imports data with the questions, labels for the response options etc.

By default this function converts labelled to a xlr_vector by default (and underlying it is a `character()` type).

See labelled and read_sav if you would like more details on the importing type.

Value

a `xlr_table` object. Use write_xlsx to write to an `Excel` file. See xlr_table for more information.

Examples

```
library(xlr)

# You can use this function to get a block of questions
build_qtable(
  clothes_opinions,
  starts_with("Q1"),
  table_title = "This is an example table")

# Another way you could select the same columns
build_qtable(
  clothes_opinions,
  c(Q1_1,Q1_2,Q1_3,Q1_4),
  table_title = "This is an example table")

# Yet another way to select the same columns
build_qtable(
  clothes_opinions,
  all_of(c("Q1_1","Q1_2","Q1_3","Q1_4")),
  table_title = "This is an example table")
# You can also cut all questions in the block by a single column
build_qtable(
  clothes_opinions,
  starts_with("Q1"),
  gender2,
  table_title = "This is the second example table")

# You can also cut all questions in the block by a multiple columns
# By setting `use_questions=TRUE` then the footnote will be the questions
# labels, for the cut questions
build_qtable(
  clothes_opinions,
  starts_with("Q1"),
  c(gender2,age_group),
  table_title = "This is the third example table",
  use_questions = TRUE)

# You can also use weights, these weights can be either doubles or integers
# based weights
# You can also set a footnote
build_qtable(
  clothes_opinions,
  starts_with("Q1"),
  age_group,
  table_title = "This is the fourth example table",
  wt = weight,
  footnote = paste0("This is a footnote, you can use it if you want ",
                    "more detail in your table."))
```

build_table *Create a one, two, three,..., n-way table*

Description

build_table creates a one, two, three, ..., n-way table. It should be used to calculate the count and percentage of different categorical variables. It gives the data back in a long format. The percentages calculated are the 'row' percentages.

Usage

```
build_table(
  x,
  cols,
  table_title = "",
  use_questions = FALSE,
  use_NA = FALSE,
  wt = NULL,
  footnote = ""
)
```

Arguments

x	a data frame or tidy object.
cols	<tidyr_tidy_select> These are the column(s) that we want to calculate the count and percentage of.
table_title	a string. The title of the table sheet.
use_questions	a logical. If the data has column labels convert the column label to a footnote with the question. See details for more information.
use_NA	a logical. Whether to include NA values in the table. For more complicated NA processing post creation, we recommend using filter.
wt	a quoted or unquote column name. Specify a weighting variable, if NULL no weight is applied.
footnote	a character vector. Optional parameter to pass a custom footnote to the question, this parameter overwrites use_questions.

Details

This function and its family (build_mtable, build_qtable) is designed to work with data with columns of type haven::labelled, which is the default format of data read with haven::read_sav/has the format of .sav. .sav is the default file function type of data from SPSS and can be exported from popular survey providers such as Qualtrics. When you read in data with haven::read_sav it imports data with the questions, labels for the response options etc.

By default this function converts labelled to a xlr_vector by default (and underlying it is a character() type).

See labelled and read_sav if you would like more details on the importing type.

Value

a xlr_table object. Use write_xlsx to write to an Excel file. See xlr_table for more information.

Examples

```
library(xlr)

# You can use this function to calculate the number count and percentage
# of a categorical variable
build_table(
  clothes_opinions,
  gender,
  table_title = "The count of the gender groups")

# You must use a `tidyselect` statement, to select the columns that you wish to
# calculate the count, and group percentage.
# This will calculate the number of observations in each group of age and
# gender.
# The percentage will be the percentage of each age_group in each gender
# group (the row percentage).
build_table(
  clothes_opinions,
  c(gender,age_group),
  table_title = "This is the second example table")

# You can use more complicated tidy select statements if you have a large number
# of columns, but this is probably not recommended
#
# Using use_questions, if you have labelled data, it will take the label and
# include it as a footnote.
# This is useful for when you have exported data from survey platforms
# as a .sav, use `haven::read_sav` to load it into your R environment.
build_table(
  clothes_opinions,
  c(group:gender,Q1_1),
  table_title = "This is the third example table",
  use_questions = TRUE)

# You can also use weights, these weights can be either doubles or integers
# based weights
# You can also set a footnote manually
build_table(
  clothes_opinions,
  age_group,
  table_title = "This is the fourth example table",
  wt = weight,
  footnote = paste0("This is a footnote, you can use it if you want",
                    "more detail in your table."))
```

clothes_opinions *Clothes opinions data*

Description

This is a fake data set used to show how to work with the xlr package.

Usage

```
clothes_opinions
```

Format

`clothes_opinions`**:**

A data frame with 1000 rows and 20 variables.

weight Fake survey weights

group A grouping variable

gender A character vector for gender

gender2 A haven labelled vector for gender

age A continuous age variable

age_group A character vector for grouped age, generated from `age`

Q1_1 The first column in a question block asking whether pants are good to wear. Likert scale.

Q1_2 The second column in a question block asking whether shirts are good to wear. Likert scale.

Q1_3 The third column in a question block asking whether shoes are good to wear. Likert scale.

Q1_4 The forth column in a question block asking whether pants are good to wear. Likert scale. This column is intentionally has no label.

Q2_1,2,3,4,5,6 Multiple response columns. Question asking what is your favourite colour to wear.

Q3_1,2,3 Multiple response columns. Question asking what is your favourite jewellery to wear.

Q3_other The other column for question 3

create_table_of_contents

Adds a table of contents to an .xlsx (`Excel`) file

Description

This function adds a table of contents to an `Excel` file by reading the information from the `Excel` sheet in, and then using that data to create the table of contents. It guesses what the information is, see details below.

Usage

```
create_table_of_contents(
  file,
  title = NA_character_,
  overwrite = TRUE,
  pull_titles = TRUE,
  TOC_sheet_name = "Table of Contents"
)
```

Arguments

file	the file name.
title	the title for the table.
overwrite	logical. When TRUE overwrite the file, if FALSE it will not overwrite the file.
pull_titles	when TRUE take the titles from the Excel sheets, and add them to the description in the TOC_sheet_name.
TOC_sheet_name	string. the sheet name for the table of contents.

Details

This function uses the sheet names to create the table of contents. For the titles it pulls the text that is the position A1 in each of the sheets. It chooses this as this is the default location of titles when you write a xlr_table with write_xlsx.

Value

Returns a logical or error if writing the file succeeded.

Examples

```
library(xlr)
library(openxlsx)
table_list <- list("Sheet name 1" = mtcars,
                   "Sheet name 2" = mtcars)

output_file <- "example_file.xlsx"

# using write xlsx we create an `Excel` document
# You could use xlr::write_xlsx to create a table of
# contents automatically.
write.xlsx(table_list,
           output_file)

# Now add the table of contents to the existing file
create_table_of_contents(output_file,
                         "A workbook with example tables",
                         # it only makes sense to pull titles when
                         # the first cell has a text description
                         pull_titles = FALSE)
```

is_xlr_format *Test if an object is a* xlr_format

Description

Test if an object is a xlr_format

Usage

```
is_xlr_format(x)
```

Arguments

x	An object to test

Value

a logical.

Examples

```
# Test if an object is a xlr_format
is_xlr_format(1)
bf <- xlr_format(font_size = 14)
is_xlr_format(bf)
```

is_xlr_type *Check if a variable is an xlr type This function tests whether an R variable has a xlr type.*

Description

Check if a variable is an xlr type This function tests whether an R variable has a xlr type.

Usage

```
is_xlr_type(x)
```

Arguments

x	a variable you wish to test

Value

a logical.

update_theme *Update the* xlr_table *theme*

Description

This function allows you to update the underlying styling for your xlr_table. This changes how the titles, footnotes, columns, and body objects look when you write you xlr_table to Excel with write_xlsx().

Usage

```
update_theme(
  x,
  title_format = xlr_format(font_size = 12, text_style = "bold"),
  footnote_format = xlr_format(font_size = 9, text_style = "italic"),
 column_heading_format = xlr_format(font_size = 11, text_style = "bold", border =
   c("top", "bottom"), halign = "center", wrap_text = TRUE),
  table_body_format = xlr_format(border = c("top", "left", "right", "bottom"))
)
```

Arguments

x	a xlr_table
title_format	a xlr_format object to format the title
footnote_format	a xlr_format object to format the footnote
column_heading_format	a xlr_format object to format the column heading
table_body_format	a xlr_format object to format the body

Details

If you want to change the style of the *columns* in the data, you should convert them to a xlr_vector, xlr_numeric, xlr_integer or xlr_percent type if they are not already, and then update the xlr_format attribute, by setting the style parameter.

Value

Returns a xlr_table object.

Examples

```
library(xlr)
# set up a basic table
bt <- xlr_table(mtcars,
                "A title",
```

```
                "A footnote")
# now we want to update the title
# This changes what it look likes when we print it to `Excel`
bt <- update_theme(bt,
                   xlr_format(font_size = 12,
                              text_style = c("bold","underline")))
# To see the change you must write to an Excel file
write_xlsx(bt,
           "example.xlsx",
           "Test")
```

`write_xlsx` *Write a* `xlr_table`*,* `data.frame`*, or* `tibble` *to an .xlsx (*`Excel`*) file*

Description

This function writes `xlr_table`, `data.frame`, or `tibble` to an .xlsx (`Excel` file). Like write.xlsx you can also write a `list` of `xlr_table`'s, `data.frame`'s, and `tibbles`'s to the one file. The main use of this function is that it uses the formatting in a `xlr_table` when it writes to the `Excel` sheet. See xlr_table for more information.

Usage

```
write_xlsx(
  x,
  file,
  sheet_name = NULL,
  overwrite = FALSE,
  append = TRUE,
  TOC = FALSE,
  TOC_title = NA_character_,
  overwrite_sheets = TRUE,
  excel_data_table = TRUE
)
```

Arguments

`x`	a single or `list` of types `xlr_table`, `data.frame`, or `tibble`.
`file`	character. A valid file path.
`sheet_name`	a sheet name (optional). Only valid for when you pass a single object to `x`.
`overwrite`	logical. Whether to overwrite the file/worksheet or not.
`append`	logical. Whether or not to append a worksheet to an existing file.
`TOC`	logical. Whether to create a table of contents with the document. Works only when you pass a `list` to `x`. To add a table of contents to an existing file, use `create_table_of_contents()`.

`TOC_title` character. To specify the table of contents title (optional).

`overwrite_sheets`
logical. Whether to overwrite existing sheets in a file.

`excel_data_table`
logical. Whether to save the data as an `Excel` table in the worksheet. These are more accessible than data in the sheet.

Value

None

Examples

```
library(xlr)
library(tibble)
# we can write a data.frame or tibble with write_xlsx
example_tibble <- tibble(example = c(1:100))

write_xlsx(mtcars,
           "example_file.xlsx",
           sheet_name = "Example sheet")

# you must specify a sheet name
write_xlsx(example_tibble,
           "example_file.xlsx",
           sheet_name = "Example sheet")

# You can write a xlr_table.
# When you write a xlr_table you can specify the formatting as well as titles
# and footnotes.
example_xlr_table <- xlr_table(mtcars,
                               "This is a title",
                               "This is a footnote")

write_xlsx(example_xlr_table,
           "example_file.xlsx",
           "Example sheet")

# like openxlsx, you can also pass a list
table_list <- list("Sheet name 1" = xlr_table(mtcars,
                                              "This is a title",
                                              "This is a footnote"),
                   "Sheet name 2" = xlr_table(mtcars,
                                              "This is a title too",
                                              "This is a footnote as well"))

write_xlsx(table_list,
           "example_file.xlsx")
```

xlr_and_dplyr *xlr and dplyr*

Description

`xlr_table()` is designed to work with dplyr verbs by default. This is so you `mutate`, `summarise`, `arrange` etc. your data without losing your `xlr_table` information. Particularly if you have used `build_table` first on your data, which outputs data as a `xlr_table`.

The list of currently supported dplyrs verbs are: `arrange`, `distinct`, `filter`, `mutate`, `relocate`, `rename`, `rename_with`, `rowwise`, `select`, `slice`, `slice_head`, `slice_max`, `slice_min`, `slice_sample`, `slice_tail`, `summarise`.

xlr_format *Specify formatting options for* `xlr_*` *types*

Description

This function is a utility to work with `openxlxs`'s createStyle, and work with styles between them. `xlr_format_numeric()` is an alias for `xlr_format()` but with different default values.

Usage

```
xlr_format(
  font_size = 11,
  font_colour = "black",
  font = "calibri",
  text_style = NULL,
  border = NULL,
  border_colour = "black",
  border_style = "thin",
  background_colour = NULL,
  halign = "left",
  valign = "top",
  wrap_text = FALSE,
  text_rotation = 0L,
  indent = 0L
)

xlr_format_numeric(
  font_size = 11,
  font_colour = "black",
  font = "calibri",
  text_style = NULL,
  border = NULL,
```

```
  border_colour = "black",
  border_style = "thin",
  background_colour = NULL,
  halign = "right",
  valign = "bottom",
  wrap_text = FALSE,
  text_rotation = 0L,
  indent = 0L
)
```

Arguments

font_size	A numeric. The font size, must be greater than 0.
font_colour	String. The colour of text in the cell. Must be one of colours() or a valid hex colour beginning with "#".
font	String. The name of a font. This is not validated.
text_style	the text styling. You can pass a vector of text decorations or a single string. The options for text style are "bold", "strikeout", "italic", "underline","underline2" (double underline), "accounting" (accounting underline), "accounting2" (double accounting underline). See Details.
border	the cell border. You can pass a vector of "top", "bottom", "left", "right" or a single string to set the borders that you want.
border_colour	Character. The colour of border. Must be the same length as the number of sides specified in border. Each element must be one of colours() or a valid hex colour beginning with "#".
border_style	Border line style vector the same length as the number of sides specified in border. The list of styles are "none", "thin", "medium", "dashed", "dotted", "thick", "double", "hair", "mediumDashed", "dashDot", "mediumDashDot", "dashDotDot", "mediumDashDot", "dastDotDot", "mediumDashDotDot", "slantDashDosh". See createStyle for more details.
background_colour	Character. Set the background colour for the cell. Must be one of colours() or a valid hex colour beginning with "#".
halign	the horizontal alignment of cell contents. Must be either "left", "right", "center" or "justify".
valign	the vertical alignment of cell contents. Must be either "top", "center", or "bottom".
wrap_text	Logical. If TRUE cell contents will rap to fit in the column.
text_rotation	Integer. Rotation of text in degrees. Must be an integer between -90 and 90.
indent	Integer. The number of indent positions, must be an integer between 0 and 250.

Details

Text styling:

For text styling you can pass either one of the options or options in a vector. For example if you would like to have text that is **bold** and *italised* then set:

```
fmt <- xlr_format(text_style = c("bold", "italic"))
```

If you would like to the text to be only **bold** then:

```
fmt <- xlr_format(text_style = "bold")
```

Border styling:

The three arguments to create border styling are `border`, `border_colour`, and `border_style`. They each take either a vector, where you specify to change what borders to have in each cell and what they look like. To specify that you want a border around a cell, use `border`, you need to pass a vector of what sides you want to have a border (or a single element if it's only one side). For example:

- `"top"` the top border
- `"left"` the left border
- `c("bottom", "right")` the top and bottom border
- `c("left", "right", "bottom")` the left, right and bottom borders
- `c("top","right","bottom","left")` the borders for all sides of the cells

Based on this you can use `border_colour` to set the border colours. If you want all the same border colour, just pass a character representing the colour you want (e.g. set `border_colour = "blue"` if you'd like all borders to be blue). Alternatively you can pass a vector the same length as the vector that you passed to `border`, with the location specifying the colour. For example, if you set:

```
fmt <- xlr_format(border = c("left", "top"),
                    border_colour = c("blue","red"))
```

the top border will be red, and the left border will be blue. You set the pattern in the same way for `border_style`. Alternatively if you only wanted it to be dashed with default colours. You'd set:

```
fmt <- xlr_format(border = c("left", "top"),
                  border_style = "dashed")
```

Value

a `xlr_format` S3 class.

See Also

- `is_xlr_format()` to test if an R object is a `xlr_format`
- `xlr_table()` to use xlr formats

Examples

```
library(xlr)
# You can initialise a xlr_format, it comes with a list of defaults
bf <- xlr_format()
# It outputs what the style looks like
bf
# You can update the format by defining a new format
bf <- xlr_format(font_size = 11,
```

```
                  # not that font is not validated
                  font = "helvetica")
# The main use of xlr_format is to change the format of a vector of
# a xlr type
bd <- xlr_numeric(1:200,
                  dp = 1,
                  style = bf)
# You can also use it to change the styles of an xlr_table, this only
# affect the format in `Excel`
bt <- xlr_table(mtcars, "A clever title", "A useful footnote")
bt <- bt |>
      update_theme(footnote_format = xlr_format(font_size = 7))
```

`xlr_integer` `xlr_integer` *vector*

Description

This creates an integer vector that will be printed neatly and can easily be exported to `Excel` using it's native format.You can convert a vector back to its base type with `as_base_r()`.

Usage

```
xlr_integer(x = integer(), style = xlr_format_numeric())

is_xlr_integer(x)

as_xlr_integer(x, style = xlr_format_numeric())
```

Arguments

`x`	A numeric vector • For `xlr_integer()`: A numeric vector • For `is_xlr_integer()`: An object to test • For `as_xlr_integer()` : a vector
`style`	Additional styling options for the vector. See xlr_format_numeric for more details.

Details

Internally, `xlr_integer` uses `vec_cast` to convert numeric types to integers. Anything that `vec_cast` can handle so can `xlr_integer`. Read more about casting at vec_cast.

Value

An S3 vector of class `xlr_integer`

See Also

xlr_vector(), xlr_percent(), xlr_numeric()

Examples

```
library(xlr)
# Create a variable to represent an integer
x <- xlr_integer(2)
# This will print nicely
x
# You can change the styling, which affects how it looks when we save it as an
# `Excel` document
x <- xlr_integer(x, style = xlr_format(font_size = 9, font_colour = "red"))
x
# We can also define a vector of integers
y <- xlr_integer(c(1,2,3))
y
# You can convert existing data to a integer using dplyr verbs
# It formats large numbers nicely
df <- data.frame(col_1 = c(1:100*100))
df |>
  dplyr::mutate(col_pct = as_xlr_integer(col_1))
# You can use as_xlr_integer to convert a string in a integer
df <- data.frame(col_str = c("12","13","14"))
# now we can convert the string to a integer(), internally it uses the same
# logic as as.integer()
df |>
  dplyr::mutate(col_percent = as_xlr_integer(col_str))
```

---

xlr_numeric xlr_numeric *vector*

Description

This creates an numeric vector that will be printed neatly and can easily be exported to Excel using it's native format. You can convert a vector back to its base type with as_base_r().

Usage

```
xlr_numeric(
  x = numeric(),
  dp = 2L,
  scientific = FALSE,
  style = xlr_format_numeric()
)

is_xlr_numeric(x)

as_xlr_numeric(x, dp = 0L, scientific = FALSE, style = xlr_format_numeric())
```

Arguments

x	• For `xlr_numeric()`: A numeric vector • For `is_xlr_numeric()`: An object to test • For `as_xlr_numeric()` : a vector
dp	the number of decimal places to print
scientific	logical. Whether to format the numeric using scientific notation.
style	Additional styling options for the vector. See xlr_format_numeric for more details.

Details

Internally, `xlr_numeric` uses `vec_cast` to convert numeric types to integers. Anything that `vec_cast` can handle so can `xlr_numeric`. Read more about casting at vec_cast.

Value

An S3 vector of class `xlr_numeric`

See Also

`xlr_percent()`, `xlr_integer()`, `xlr_vector()`, `as_base_r()`

Examples

```
library(xlr)
# Create a variable to represent a double with two decimal places
# The decimal places must be a positive integer
x <- xlr_numeric(2.1134,dp = 2)
# This will print nicely
x
# You can change the styling, which affects how it looks when we print it
x <- xlr_numeric(x, dp = 3L, style = xlr_format(font_size = 9, font_colour = "red"))
x
# We can also define a vector of doubles
y <- xlr_numeric(c(22.1055,1.3333333,3.1234567), dp = 2)
y
# You can convert existing data to a double using dplyr verbs
df <- data.frame(col_1 = c(2,3.2,1.33,4.43251))
df |>
  dplyr::mutate(col_pct = as_xlr_numeric(col_1))
# You can use as_xlr_numeric to convert a string in a double
df <- data.frame(col_str = c("12.22","12.34567","100"))
# now we can convert the string to a double(), internally it uses the same
# logic as as.double()
df |>
  dplyr::mutate(col_double = as_xlr_numeric(col_str,2))
```

xlr_percent xlr_percent *vector*

Description

This creates a numeric vector that will be printed as a percentage and exported to Excel using it's native format.You can convert a vector back to its base type with as_base_r().

Usage

```
xlr_percent(x = double(), dp = 0L, style = xlr_format_numeric())

is_xlr_percent(x)

as_xlr_percent(x, dp = 0L, style = xlr_format_numeric())
```

Arguments

x	• For xlr_percent(): A numeric vector • For is_xlr_percent(): An object to test • For as_xlr_percent() : a numeric or character vector. For a character vector, the data must be in the format "XXX.YYY...%".
dp	the number of decimal places to print
style	Additional styling options for the vector. See xlr_format_numeric for more details.

Value

An S3 vector of class xlr_percent

See Also

xlr_vector(), xlr_integer(), xlr_numeric(), as_base_r()

Examples

```
library(xlr)
# lets define a xlr_percent, a xlr_percent is between a number between [0-1], not
# between 1-100
#
# Create a variable to represent 10%
x <- xlr_percent(0.1)
# This will print nicely
x
# Now we can increase the number of decimal places to display
# The decimal places must be a positive integer
x <- xlr_percent(x, dp = 3L)
x
```

```
# We can also define a vector of xlr_percents
y <- xlr_percent(c(0.1055,0.3333333,0.1234567), dp = 2)
y
# You can convert existing data to a xlr_percentage using dplyr verbs
df <- data.frame(col_1 = c(0,0.2,0.33,0.43251))
df |>
  dplyr::mutate(col_pct = as_xlr_percent(col_1))
# You can also change the styling of a xlr_percent column, this is only relevant
# if you print it to `Excel` with write_xlsx
df |>
  dplyr::mutate(col_pct = xlr_percent(col_1,
                                      dp = 2,
                                      style = xlr_format(font_size = 8)))
# You can use as_xlr_percent to convert a string in a xlr_percentage format to a
# xlr_percent
df <- data.frame(col_str = c("12.22%","12.34567%","100%"))
# now we can convert the string to a xlr_xlr_percent()
df |>
  dplyr::mutate(col_xlr_percent = as_xlr_percent(col_str,2))
```

`xlr_table` `xlr_table` *object*

Description

Create a `xlr_table` S3 object. This is used to create an object that stores formatting information, as well as a title and footnote. This objects makes it easy to convert to an `Excel` sheet, using `write_xlsx()`. To edit underlying formatting options use `update_theme()`.

A number of `dplyr` methods have been implemented for `xlr_table`, these include `mutate`, `summarise`, `select`, etc. This means you can use these functions on a `xlr_table`, without losing the `xlr_table` attributes. You can check if the `dplyr` function is supported by checking the documentation of the function. Currently, it is not possible to use `group_by` and a `xlr_table`, as this would require the implementation of a new class.

You can convert a table back to a data.frame with base type with `as_base_r()`.

Usage

```
xlr_table(x, title = character(), footnote = character())

is_xlr_table(x)

as_xlr_table(x, title = character(), footnote = character())
```

Arguments

x	a data object • for `xlr_table()` : a `data.frame`, or `tibble`. See notes for further details. • for `is_xlr_table()` : An object

- for `as_xlr_table()` a `data.frame`, or `tibble`.

`title`	a string that is the title
`footnote`	a string that is the footnote

Value

a `xlr_table` S3 class

See Also

`update_theme()`, `as_base_r()`

Examples

```
library(xlr)
library(dplyr)
# Create a xlr_table, we set the footnotes and the title
# It converts to the xlr types by default
x <- xlr_table(mtcars,
               title = "mtcars is a fun data set",
               footnote = "mtcars is a data set that comes with base R")
# The title and the footnote print to console
x
# You can use mutate and summarise with xlr_tables and they are preserved
x |>
  summarise(mean_mpg = sum(mpg))
# Rename a column
x |>
  rename(new_name = mpg)
# When you want to change how elements of the table look when written using
# write_xlsx, you can update it with update them
x <- x |>
  # make the font bigger
  update_theme(title_format = xlr_format(font_size = 14))
# you must write it in order to see the formatting changes
write_xlsx(x,
            "example.xlsx",
            "A example sheet",
            TOC = FALSE)
```

`xlr_vector` `xlr_vector` *vector*

Description

A general container for including additional styling options within a vector so that it can easily be exported to `Excel`. This vector type should be used for characters, factors, Booleans, complex numbers, etc. It does not support dates.

Usage

```
xlr_vector(x = vector(), excel_format = "GENERAL", style = xlr_format())

is_xlr_vector(x)

as_xlr_vector(x, excel_format = "GENERAL", style = xlr_format())
```

Arguments

x	A vector • For `xlr_vector()`: A vector • For `is_xlr_vector()`: An object to test • For `as_xlr_vector()` : a vector
excel_format	a character, the `Excel` cell format, not validated. See createStyle argument numFmt for more details on what you can specify.
style	Additional styling options for the vector. See xlr_format for more details.

Details

While you can use it with `integer`, and `double` types and specifying the associated `Excel` format, we recommend using xlr_integer, xlr_numeric, or xlr_percent types instead.

You can convert a vector back to its base type with `as_base_r()`.

Value

An S3 vector of class `xlr_vector`

See Also

`xlr_percent()`, `xlr_integer()`, `xlr_numeric()`, `as_base_r()`

Examples

```
library(xlr)
# Create a xlr_vector object, this is used so we can add styling to an existing
# vector so that it prints nicely in `Excel`
#
# Note currently the style will not change the style in the console
x <- xlr_vector(1:100,
                excel_format = "00.0##",
                style = xlr_format(font_size = 8))

# You can also use it so that dates are nicely printed in `Excel`
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
dates <- as.Date(dates, "%m/%d/%y")
x <- xlr_vector(dates,
                # Print it as a long date in `Excel`
                excel_format = "LONGDATE")
# You can convert existing data to a xlr_vectors using dplyr verbs
```

```
iris |>
  dplyr::mutate(iris_format = as_xlr_vector(Species,
                                            "TEXT",
                                            xlr_format(text_style = "italic")))
```

Index