The `openxlsx2` book

Jan Marvin Garbuszus (JanMarvin) and `openxlsx2` authors

Table of contents

Preface

This is a work in progress book describing the features of openxlsx2 (Barbone and Garbuszus 2024). Having written a book before, I never imagined to ever write one again and therefore I shall not do it. But still I consider it a nice addition to have something more flexible as our `vignettes`.

The `openxlsx2` book is a comprehensive guide to using the R package openxlsx2 for working with xlsx files. It covers core functionalities such as reading, writing, and editing office open xml (OOXML) spreadsheet files, alongside advanced features like styling worksheets, handling conditional formatting, creating charts, managing pivot tables, and adding data validation. The book also discusses extending the package's functionality and upgrading from the original openxlsx package. It was created in the hopes that it's a useful resource for R users needing spreadsheet file manipulation.

This manual was compiled using:

```
R.version
```

```
               _
platform       x86_64-pc-linux-gnu
arch           x86_64
os             linux-gnu
system         x86_64, linux-gnu
status
major          4
minor          5.1
year           2025
month          06
day            13
svn rev        88306
language       R
version.string R version 4.5.1 (2025-06-13)
nickname       Great Square Root
```

and

```
packageVersion("openxlsx2")
```

```
[1] '1.17.0.9000'
```

Graphics might reflect earlier states and are not constantly updated. If you find any irregularities where our code produces different output than expected, please let us know in the issue tracker at https://github.com/JanMarvin/openxlsx2/.

Additional examples

For many more examples of what `openxlsx2` can do, have a look at the `Show and tell` section of the `openxlsx2` discussion board: https://github.com/JanMarvin/openxlsx2/discussions/categories/show-and-tell

Where to get help

For all things `openxlsx2` consult our discussion board at https://github.com/JanMarvin/openxlsx2/discussions/categories/q-a

1 Introduction

Unfortunately the entire business world is still built almost entirely on Microsoft Office tools and whenever data is involved, this means that is is largely built on the spreadsheet software Excel. R users that want to interact with this previously closed source file format had to rely on various packages (the following is not necessarily a complete list of all packages). Packages that create workbook objects like `xlsx` (Dragulescu and Arendt 2023) and `openxlsx` (Schauberger and Walker 2023) and packages for special tasks namely `readxl` (Wickham and Bryan 2023), `readxlsb` (Allen 2023)[1], `tidyxl` (Garmonsway 2022), `writexl` (Ooms 2023) and `WriteXLS` (Schwartz 2022), some are Windows exclusive interacting with Excel via a DCOM server `RDCOMClient` and `RExcel` [2], some are not, `XLconnect`. [3]

In Excel 2007 a new open standard called OOXML(short for office open xml)[4] which we will refer to as *openxml* was introduced. In December 2006 this standard was accepted by the ECMA and it subsequently replaced the previously used `xls` files wherever people are working with spreadsheet software (after all we are all aware that accounting does not really care whatever file format they are using as long as it opens up in their favorite spreadsheet software). The openxml standard introduced the so called Excel 2007 workbook format `xlsx`. These files are a collection of zipped XML-files. This makes is easy to import the files to R, because all you need is a tool to unzip the files and an XML-parser to import the files as data frames. Still, since there are various tasks available to interact with spreadsheet file, there are also various tools required. If all you want to do is read from files `readxl` is probably enough, if all you want to do is write xlsx files `writexl` is probably the fastest choice available. Yet there are a plethora of other tasks available and this book is about them.

The predecessor to `openxlsx2` (Barbone and Garbuszus 2024) called `openxlsx` (originally founded by Andrew Walker) was inspired by the `rJava` based `xlsx` package, but dropped the `rJava` dependency, and the support for the old `xls` files and wrote a custom XML parser in `Rcpp` (Eddelbuettel and François 2011). Later Phillip Schauberger picked up the abandoned `openxlsx` package and continues to maintain it. Finally `openxlsx2` was forked from `openxlsx`

[1] Since the original creation of this section `readxlsb` has been archived on CRAN. A release that includes a bug fix for overflowing values can be found at `https://github.com/JanMarvin/readxlsb` (but this is not actively developed or maintained).

[2] See https://github.com/omegahat/RDCOMClient.

[3] And there are many other packages on CRAN for working with `xls`/`xlsx` spreadsheet files. Without a guarantee for completeness: `SheetReader` (Henze 2024), `tablexlsx` (Dotta and Blasco 2024), `xlsx2dfs` (Kim 2019), `tablaxlsx` (Rodríguez 2023), `xlr` (Hilderson 2025), `xlcutter` (Gruson 2023), `knitxl` (Dreano 2023), `xlcharts` (Luginbuhl 2024), `joinXL` (Glanville 2016).

[4] See https://wikipedia.org/wiki/Office_Open_XML.

to include (1) the `pugixml` (Kapoulkine 2006-2023) library to address shortcomings of the `openxlsx` XML parser and (2) to switch from `methods` to the `R6` (Chang 2021) package to introduce modern programming flows. Since then `openxlsx2` has evolved a lot, includes many new features and is in a semi-stable API state since release `1.0`.[5] This manual is supposed to bundle and extend the existing vignettes and to document the changes.

1.1 Installation

You can install the stable version of `openxlsx2` with:

```
install.packages('openxlsx2')
```

You can install the development version of `openxlsx2` from GitHub with:

```
# install.packages("remotes")
remotes::install_github("JanMarvin/openxlsx2")
```

Or from r-universe with:

```
# Enable repository from janmarvin
options(repos = c(
  janmarvin = 'https://janmarvin.r-universe.dev',
  CRAN = 'https://cloud.r-project.org'))
# Download and install openxlsx2 in R
install.packages('openxlsx2')
```

1.2 Working with the package

We offer two different variants how to work with `openxlsx2`.

- The first one is to simply work with R objects. It is possible to read (`read_xlsx()`) and write (`write_xlsx()`) data from and to files. We offer a number of options in the commands to support various features of the openxml format, including reading and writing named ranges and tables. Furthermore, there are several ways to read certain information of an openxml spreadsheet without having opened it in a spreadsheet software before, e.g. to get the contained sheet names or tables.

[5] With 'semi-stable' we promise not to break the API unless we come across a bug that forces us to. All breaking changes are mentioned in the changelog.

- As a second variant `openxlsx2` offers the work with so called `wbWorkbook` objects. Here an openxml file is read into a corresponding `wbWorkbook` object (`wb_load()`) or a new one is created (`wb_workbook()`). Afterwards the object can be further modified using various functions. For example, worksheets can be added or removed, the layout of cells or entire worksheets can be changed, and cells can be modified (overwritten or rewritten). Afterwards the `wbWorkbook` objects can be written as openxml files and processed by suitable spreadsheet software.

1.3 Example

This is a basic example which shows you how to solve a common problem:

```
library(openxlsx2)
# read xlsx or xlsm files
path <- system.file("extdata/openxlsx2_example.xlsx", package = "openxlsx2")
read_xlsx(path)
```

```
    Var1 Var2 <NA>  Var3  Var4       Var5         Var6    Var7     Var8
3   TRUE    1   NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
4   TRUE   NA   NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
5   TRUE    2   NA  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
6  FALSE    2   NA  <NA> #NUM!       <NA>         <NA>       2 17:24:53
7  FALSE    3   NA  1.56     e       <NA>         <NA>    <NA>     <NA>
8  FALSE    1   NA   1.7     f 2023-03-02         <NA>     2.7 08:45:58
9     NA   NA   NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
10 FALSE    2   NA    23     h 2023-12-24         <NA>      25     <NA>
11 FALSE    3   NA  67.3     i 2023-12-25         <NA>       3     <NA>
12    NA    1   NA   123  <NA> 2023-07-31         <NA>     122     <NA>
```

```
# or import workbooks
wb <- wb_load(path)
wb
```

```
A Workbook object.

Worksheets:
 Sheets: Sheet1, Sheet2
 Write order: 1, 2
```

```
# read a data frame
wb_to_df(wb)
```

```
    Var1 Var2 <NA>  Var3  Var4       Var5          Var6    Var7     Var8
3   TRUE    1   NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
4   TRUE   NA   NA #NUM!     b 2023-05-23          <NA>       0 14:02:57
5   TRUE    2   NA  1.34     c 2023-02-01          <NA> #VALUE! 23:01:02
6  FALSE    2   NA  <NA> #NUM!       <NA>          <NA>       2 17:24:53
7  FALSE    3   NA  1.56     e       <NA>          <NA>    <NA>     <NA>
8  FALSE    1   NA   1.7     f 2023-03-02          <NA>     2.7 08:45:58
9     NA   NA   NA  <NA>  <NA>       <NA>          <NA>    <NA>     <NA>
10 FALSE    2   NA    23     h 2023-12-24          <NA>      25     <NA>
11 FALSE    3   NA  67.3     i 2023-12-25          <NA>       3     <NA>
12    NA    1   NA   123  <NA> 2023-07-31          <NA>     122     <NA>
```

```
# and save
temp <- temp_xlsx()
if (interactive()) wb_save(wb, temp)

## or create one yourself
wb <- wb_workbook()
# add a worksheet
wb$add_worksheet("sheet")
# add some data
wb$add_data("sheet", cars)
# open it in your default spreadsheet software
if (interactive()) wb$open()
```

1.4 Authors and contributions

For a full list of all authors that have made this package possible and for whom we are grateful, please see:

```
system.file("AUTHORS", package = "openxlsx2")
```

If you feel like you should be included on this list, please let us know. If you have something to contribute, you are welcome. If something is not working as expected, open issues or if you have solved an issue, open a pull request. Please be respectful and be aware that we are volunteers doing this for fun in our unpaid free time. We will work on problems when we have time or need.

1.5 License

The `openxlsx2` package is licensed under the MIT license and is based on `openxlsx` (by Alexander Walker and Philipp Schauberger; COPYRIGHT 2014-2022) and `pugixml` (by Arseny Kapoulkine; COPYRIGHT 2006-2023). Both released under the MIT license.

1.6 A note on speed and memory usage

The current state of `openxlsx2` is that it is reasonably fast. That is, it works well with reasonably large input data when reading or writing. It may not work well with data that tests the limits of the `openxml` specification. Things may slow down on the R side of things, and performance and usability will depend on the speed and size of the local operating system's CPU and memory.

Note that there are at least two cases where `openxlsx2` constructs potentially large data frames (i) when loading, `openxlsx2` usually needs to read the entire input file into pugixml and convert it into long data frame(s), and `wb_to_df()` converts one long data frame into two data frames that construct the output object and (ii) when adding data to the workbook, `openxlsx2` reshapes the input data frame into a long data frame and stores it in the workbook, and writes the entire worksheet into a pugixml file that is written when it is complete. Applying cell styles, date conversions etc. will further slow down the process and finally the sheets will be zipped to provide the xlsx output.

Therefore, if you are faced with an unreasonably large dataset, either give yourself enough time, use another package to write the xlsx output (`openxlsx2` was not written with the intention of working with maximum memory efficiency), and by all means use other ways to store data (binary file formats or a database). However, we are always happy to improve, so if you have found a way to improve what we are currently doing, please let us know and open an issue or a pull request.

1.7 Invitation to contribute

We have put a lot of work into `openxls2` to make it useful for our needs, improving what we found useful about `openxlsx` and removing what we didn't need. We do not claim to be omniscient about all the things you can do with spreadsheet software, nor do we claim to be omniscient about all the things you can do in `openxlsx2`. Nevertheless, we are quite fond of our little package and invite others to try it out and comment on what they like and of course what they think we are missing or if something doesn't work. `openxlsx2` is a complex piece of software that certainly does not work bug-free, even if we did our best. If you want to contribute to the development of `openxlsx2`, please be our guest on our Github. Join or open a discussion, post or fix issues or write us a mail.

2 Basics

Welcome to the basic manual to `openxlsx2`. In this manual you will learn how to use `openxlsx2` to import data from xlsx-files to R as well as how to export data from R to xlsx, and how to import and modify these openxml workbooks in R. This package is based on the work of many contributors to `openxlsx`. It was mostly rewritten using `pugixml` and `R6` making use of modern technology, providing a fresh and easy to use R package.

Over the years many people have worked on the tricky task to handle xls and xlsx files. Notably `openxlsx`, but there are countless other R-packages as well as third party libraries or calculation software capable of handling such files. Please feel free to use and test your files with other software and or let us know about your experience. Open an issue on github or write us a mail.

2.1 First steps

First let's assume that you have a working installation of `openxlsx2` otherwise run the lines below to install the latest CRAN release:

```
install.packages("openxlsx2")
```

Now we load the library:

```
library(openxlsx2)
```

2.2 Handling workbooks

The foundation of `openxlsx2` is a workbook object. You can think of this object as a workbook loaded in a spreadsheet software. We import the entire thing. Every sheet, every chart, image, column, formula style, conditional formatting, pivot table and whatever else a spreadsheet file is allowed to carry. Therefore if you have a file that you want to work with, you can load it with:

```
wb <- wb_load("your_file.xlsx")
```

We usually name workbook objects `wb` in our documentation, but this is no obligation, you can name your workbook object whatever you like to call them.

If you do not have a workbook yet, it is possible to create one. In the next line we will use three wrapper functions `wb_workbook()`, `wb_add_worksheet()`, and `wb_add_data()`. The wrapper functions are piped together using `R`'s native pipe operator `|>`, but similarly you can use the classic `magrittr` pipe operator `%>%`. [1] We assume that you have a dataset `your_data`, either a vector, a matrix or a data frame and want to write this in a worksheet:

```
wb <- wb_workbook() |> wb_add_worksheet() |> wb_add_data(x = your_data)
```

Okay, now you have a workbook object, but what have we actually done? Let's work along the pipe syntax: (1) first we have created the workbook object `wb_worbkook()`, (2) next we have assigned it a worksheet `wb_add_worksheet()`, and (3) we have written data onto the worksheet.

Let's try this with actual data. We use the `mtcars` dataset. In the code we switch the fictional `your_data` with `mtcars`:

```
wb <- wb_workbook() |> wb_add_worksheet() |> wb_add_data(x = mtcars)
```

Let's see what the output looks like:

```
wb
#> A Workbook object.
#>
#> Worksheets:
#>  Sheets: Sheet 1
#>  Write order: 1
```

The output looks a little cryptic, it simply tells the name of the worksheet: `wb_add_worksheet()` created a default worksheet name `"Sheet 1"`. In the code above you can see that we do not use `sheet` to tell `wb_add_data()` where it should write the data. This is because internally we use a waiver `current_sheet()` so that we do not have to write `sheet = "Sheet 1"` whenever we work on the same worksheet. Basically the current sheet is updated whenever a new worksheet is added to the workbook.

[1] Basically a pipe operator allows to write code from left to right. Without pipes the code would look like this:

```
wb <- wb_add_data(wb_add_worksheet(wb_workbook()), x = your_data)
```

```
wb <- wb_workbook() |>
  wb_add_worksheet() |>
  wb_add_worksheet() |>
  wb_add_data(x = mtcars)
```

This will create two sheets `"Sheet 1"` and `"Sheet 2"` and the data will be written to the second sheet.

```
wb
#> A Workbook object.
#>
#> Worksheets:
#>  Sheets: Sheet 1, Sheet 2
#>  Write order: 1, 2
```

So how can we access the data on the sheet? Either with `wb_to_df()` our internal handler to read from workbooks (this is the underlying function for `wb_read()` and `read_xlsx()` which are mere aliases for `wb_to_df()`). So lets have a look at the top of the output:

```
wb |> wb_to_df() |> head()
#> sheet found, but contains no data
#> NULL
```

Ah! The output is on the second sheet. We need either `sheet = 2` or `sheet = "Sheet 2"`. We go with the second variant, because the sheet index position and their name might differ.

```
wb |> wb_to_df(sheet = "Sheet 2") |> head()
#>    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> 2 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> 3 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> 4 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#> 5 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> 6 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
#> 7 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

This looks like the head of the `mtcars` dataset. So we have successfully read from the workbook. Now you want to export the workbook to a file:

```
wb |> wb_save(file = "my_first_worksheet.xlsx")
```

Alternatively you can directly open it in a spreadsheet software (if you have one installed):

```
wb |> wb_open()
```

Once again, lets try this with the `USPersonalExpenditure` dataset:

```
wb <- wb_workbook()
wb_add_worksheet(wb, sheet = "USexp")
wb_add_data(wb, "USexp", USPersonalExpenditure)
```

```
#> Error : Can't add data to a workbook with no worksheet.
#> Did you forget to add a worksheet with `wb_add_worksheet()`?
```

Dang! What did we do? We've added a worksheet, but wait, did we? No, you have to assign wrapper functions to an object for them to have an effect. Wrapper functions do not alter the workbook objects they are executed on. You can check that the workbook has no worksheets:

```
wb |> wb_get_sheet_names()
#> named character(0)
```

Once we assign a sheet, this changes, and the data was correctly written:

```
wb <- wb_workbook()
wb <- wb_add_worksheet(wb, sheet = "USexp")
wb <- wb_add_data(wb, "USexp", USPersonalExpenditure)
wb_get_sheet_names(wb)
#>   USexp
#> "USexp"
wb_to_df(wb)
#>     1940   1945  1950 1955  1960
#> 2 22.200 44.500 59.60 73.2 86.80
#> 3 10.500 15.500 29.00 36.5 46.20
#> 4  3.530  5.760  9.71 14.0 21.10
#> 5  1.040  1.980  2.45  3.4  5.40
#> 6  0.341  0.974  1.80  2.6  3.64
```

Now you're probably thinking, I don't want to assign the workbook object all the time and all the `wb_` functions are a little tedious to type. There is an alternative for you and it is called chaining. Since the workbook is a `R6` object internally, you can make use of chains. Basically every function that starts with `wb_` should have a underlying function of the same name without the prefix. So our data writing example from above can be written as:

```
wb <- wb_workbook()$add_worksheet("USexp")$add_data(x = USPersonalExpenditure)
wb$to_df()
#>     1940   1945  1950 1955  1960
#> 2 22.200 44.500 59.60 73.2 86.80
#> 3 10.500 15.500 29.00 36.5 46.20
#> 4  3.530  5.760  9.71 14.0 21.10
#> 5  1.040  1.980  2.45  3.4  5.40
#> 6  0.341  0.974  1.80  2.6  3.64
```

Whether you use wrapper functions or chain functions is up to you and personal preference. There is just one thing to remember, the documentation is exclusively written for the wrapper function. So if you want to know the arguments for the `wb$add_data()` part, you have to lookup the wrapper functions man page `?wb_add_data`.

2.3 Importing as workbook

In addition to importing directly from xlsx or xlsm files, `openxlsx2` provides the `wbWorkbook` class used for importing and modifying entire the openxml files in `R`. This `workbook` class is the heart of `openxlsx2` and probably the reason why you are reading this manual in the first place.

Importing a file into a workbook looks like this:

```
# the file we are going to load
file <- system.file("extdata", "openxlsx2_example.xlsx", package = "openxlsx2")
# loading the file into the workbook
wb <- wb_load(file = file)
```

The additional options `wb_load()` provides are for internal use: `sheet` loads only a selected sheet from the workbook and `data_only` reads only the data parts from a workbook and ignores any additional graphics or pivot tables. Both functions create workbook objects that can only be used to read data, and we do not recommend end users to use them. Especially not if they intend to re-export the workbook afterwards.

Once a workbook is imported, we provide several functions to interact with and modify it (the `wb_to_df()` function mentioned above works the same way for an imported workbook). It is possible to add new sheets and remove sheets, as well as to add or remove data. R-plots can be inserted and also the style of the workbook can be changed, new fonts, background colors and number formats. There is a wealth of options explained in the man pages and the additional style vignette (more vignettes to follow).

3 Exporting data

3.1 Exporting data frames or vectors

If you want to export a data frame from R, you can use `write_xlsx()` which will create an xlsx file. This file can be tweaked further. See `?write_xlsx` to see all the options. (further explanation and examples will follow).

```
write_xlsx(x = mtcars, file = "mtcars.xlsx")
```

3.2 Exporting a `wbWorkbook`

Imported workbooks can be saved as xlsx or xlsm files with the wrapper `wb_save()` or with `wb$save()`. Both functions take the filename and an optional `overwrite` option. If the latter is set, an optional guard is provided to check if the file you want to write already exists. But be careful, this is optional. The default is to save the file and replace an existing file. Of course, on Windows, files that are locked (for example, if they were opened by another process) will not be replaced.

```
# replace the existing file
wb$save("mtcars.xlsx")

# do not overwrite the existing file
try(wb$save("mtcars.xlsx", overwrite = FALSE))
```

3.3 `dims`/ `wb_dims()`

In `openxlsx2` functions that interact with worksheet cells are using `dims` as argument and require the users to provide these. `dims` are cells or cell ranges in A1 notation. The single argument `dims` hereby replaces `col`/`row`, `cols`/`rows` and `xy`. Since A1 notation is rather simple in the first few columns it might get confusing after the 26. Therefore we provide a wrapper to construct it:

```r
# various options
wb_dims(from_row = 4)
#> [1] "A4"

wb_dims(rows = 4, cols = 4)
#> [1] "D4"
wb_dims(rows = 4, cols = "D")
#> [1] "D4"

wb_dims(rows = 4:10, cols = 5:9)
#> [1] "E4:I10"

wb_dims(rows = 4:10, cols = "A:D") # same as below
#> [1] "A4:D10"
wb_dims(rows = seq_len(7), cols = seq_len(4), from_row = 4)
#> [1] "A4:D10"
# 10 rows and 15 columns from indice B2.
wb_dims(rows = 1:10, cols = 1:15, from_col = "B", from_row = 2)
#> [1] "B2:P11"

# data + col names
wb_dims(x = mtcars)
#> [1] "A1:K33"
# only data
wb_dims(x = mtcars, select = "data")
#> [1] "A2:K33"

# The dims of the values of a column in `x`
wb_dims(x = mtcars, cols = "cyl")
#> [1] "B2:B33"
# a column in `x` with the column name
wb_dims(x = mtcars, cols = "cyl", select = "x")
#> [1] "B1:B33"
# rows in `x`
wb_dims(x = mtcars)
#> [1] "A1:K33"

# in a wb chain
wb <- wb_workbook()$
  add_worksheet()$
  add_data(x = mtcars)$
  add_fill(
```

```
    dims = wb_dims(x = mtcars, rows = 1:5), # only 1st 5 rows of x data
    color = wb_color("yellow")
  )$
  add_fill(
    dims = wb_dims(x = mtcars, select = "col_names"), # only column names
    color = wb_color("cyan2")
  )

# or if the data's first coord needs to be located in B2.

wb_dims_custom <- function(...) {
  wb_dims(x = mtcars, from_col = "B", from_row = 2, ...)
}
wb <- wb_workbook()$
  add_worksheet()$
  add_data(x = mtcars, dims = wb_dims_custom())$
  add_fill(
    dims = wb_dims_custom(rows = 1:5),
    color = wb_color("yellow")
  )$
  add_fill(
    dims = wb_dims_custom(select = "col_names"),
    color = wb_color("cyan2")
  )
```

4 Reading to data frames

4.1 Importing data

Coming from `openxlsx` you might know about `read.xlsx()` (two functions, one for files and one for workbooks) and `readWorkbook()`. Functions that do different things, but mostly the same. In `openxlsx2` we tried our best to reduce the complexity under the hood and for the user as well. In `openxlsx2` they are replaced with `read_xlsx()`, `wb_read()` and they share the same underlying function `wb_to_df()`.

For this example we will use example data provided by the package. You can locate it in our "inst/extdata" folder. The files are included with the package source and you can open them in any calculation software as well.

4.1.1 Basic import

We begin with the `openxlsx2_example.xlsx` file by telling R where to find this file on our system

```
xl <- system.file("extdata", "openxlsx2_example.xlsx", package = "openxlsx2")
```

The object contains a path to the xlsx file and we pass this file to our function to read the workbook into R

```
# import workbook
wb_to_df(xl)
#>     Var1 Var2 <NA>  Var3  Var4       Var5         Var6    Var7     Var8
#> 3   TRUE    1   NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE   NA   NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2   NA  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2   NA  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3   NA  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1   NA   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9     NA   NA   NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
#> 10 FALSE    2   NA    23     h 2023-12-24         <NA>      25     <NA>
```

```
#> 11 FALSE    3   NA  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12    NA    1   NA   123  <NA> 2023-07-31         <NA>     122     <NA>
```

The output is created as a data frame and contains data types date, logical, numeric and character. The function to import the file to R, `wb_to_df()` provides similar options as the `openxlsx` functions `read.xlsx()` and `readWorkbook()` and a few new functions we will go through the options. As you might have noticed, we return the column of the xlsx file as the row name of the data frame returned. Per default the first sheet in the workbook is imported. If you want to switch this, either provide the `sheet` parameter with the correct index or provide the sheet name.

4.1.2 `col_names` - first row as column name

In the previous example the first imported row was used as column name for the data frame. This is the default behavior, but not always wanted or expected. Therefore this behavior can be disabled by the user.

```
# do not convert first row to column names
wb_to_df(xl, col_names = FALSE)
#>        B    C  D     E     F          G            H       I        J
#> 2   Var1 Var2 NA  Var3  Var4       Var5         Var6    Var7     Var8
#> 3   TRUE    1 NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE <NA> NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2 NA  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2 NA  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3 NA  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1 NA   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9   <NA> <NA> NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
#> 10 FALSE    2 NA    23     h 2023-12-24         <NA>      25     <NA>
#> 11 FALSE    3 NA  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12  <NA>    1 NA   123  <NA> 2023-07-31         <NA>     122     <NA>
```

4.1.3 `detect_dates` - convert cells to R dates

The creators of the openxml standard are well known for mistakenly treating something as a date and `openxlsx2` has built in ways to identify a cell as a date and will try to convert the value for you, but unfortunately this is not always a trivial task and might fail. In such a case we provide an option to disable the date conversion entirely. In this case the underlying numerical value will be returned.

```
# do not try to identify dates in the data
wb_to_df(xl, detect_dates = FALSE)
#>     Var1 Var2 <NA>  Var3  Var4  Var5         Var6    Var7       Var8
#> 3   TRUE    1   NA     1     a 45075 3209324 This #DIV/0! 0.06059028
#> 4   TRUE   NA   NA #NUM!     b 45069         <NA>       0 0.58538194
#> 5   TRUE    2   NA  1.34     c 44958         <NA> #VALUE! 0.95905093
#> 6  FALSE    2   NA  <NA> #NUM!    NA         <NA>       2 0.72561343
#> 7  FALSE    3   NA  1.56     e    NA         <NA>    <NA>         NA
#> 8  FALSE    1   NA   1.7     f 44987         <NA>     2.7 0.36525463
#> 9     NA   NA   NA  <NA>  <NA>    NA         <NA>    <NA>         NA
#> 10 FALSE    2   NA    23     h 45284         <NA>      25         NA
#> 11 FALSE    3   NA  67.3     i 45285         <NA>       3         NA
#> 12    NA    1   NA   123  <NA> 45138         <NA>     122         NA
```

4.1.4 `show_formula` - show formulas instead of results

Sometimes things might feel off. This can be because the openxml files are not updating formula results in the sheets unless they are opened in software that provides such functionality as certain tabular calculation software. Therefore the user might be interested in the underlying functions to see what is going on in the sheet. Using `show_formula` this is possible

```
# return the underlying Excel formula instead of their values
wb_to_df(xl, show_formula = TRUE)
#>     Var1 Var2 <NA>  Var3  Var4       Var5         Var6             Var7     Var8
#> 3   TRUE    1   NA     1     a 2023-05-29 3209324 This            E3/0 01:27:15
#> 4   TRUE   NA   NA #NUM!     b 2023-05-23         <NA>              C4 14:02:57
#> 5   TRUE    2   NA  1.34     c 2023-02-01         <NA>         #VALUE! 23:01:02
#> 6  FALSE    2   NA  <NA> #NUM!       <NA>         <NA>           C6+E6 17:24:53
#> 7  FALSE    3   NA  1.56     e       <NA>         <NA>            <NA>     <NA>
#> 8  FALSE    1   NA   1.7     f 2023-03-02         <NA>           C8+E8 08:45:58
#> 9     NA   NA   NA  <NA>  <NA>       <NA>         <NA>            <NA>     <NA>
#> 10 FALSE    2   NA    23     h 2023-12-24         <NA>    SUM(C10,E10)     <NA>
#> 11 FALSE    3   NA  67.3     i 2023-12-25         <NA> PRODUCT(C11,E3)     <NA>
#> 12    NA    1   NA   123  <NA> 2023-07-31         <NA>         E12-C12     <NA>
```

4.1.5 `dims` - read specific dimension

Sometimes the entire worksheet contains to much data, in such case we provide functions to read only a selected dimension range. Such a range consists of either a specific cell like "A1" or a cell range in the notion used in the `openxml` standard

```
# read dimension without column names
wb_to_df(xl, dims = "A2:C5", col_names = FALSE)
#>    A    B    C
#> 2 NA Var1 Var2
#> 3 NA TRUE    1
#> 4 NA TRUE <NA>
#> 5 NA TRUE    2
```

Alternatively, if you don't know the Excel sheet's address, you can use `wb_dims()` to specify the dimension. See below or in`?wb_dims` for more details.

```
# read dimension without column names with `wb_dims()`
wb_to_df(xl, dims = wb_dims(rows = 2:5, cols = 1:3), col_names = FALSE)
#>    A    B    C
#> 2 NA Var1 Var2
#> 3 NA TRUE    1
#> 4 NA TRUE <NA>
#> 5 NA TRUE    2
```

4.1.6 `cols` - read selected columns

If you do not want to read a specific cell, but a cell range you can use the column attribute. This attribute takes a numeric vector as argument

```
# read selected cols
wb_to_df(xl, cols = c("A:B", "G"))
#>    <NA>  Var1       Var5
#> 3    NA  TRUE 2023-05-29
#> 4    NA  TRUE 2023-05-23
#> 5    NA  TRUE 2023-02-01
#> 6    NA FALSE       <NA>
#> 7    NA FALSE       <NA>
#> 8    NA FALSE 2023-03-02
#> 9    NA    NA       <NA>
#> 10   NA FALSE 2023-12-24
#> 11   NA FALSE 2023-12-25
#> 12   NA    NA 2023-07-31
```

4.1.7 `rows` - read selected rows

The same goes with rows. You can select them using numeric vectors

```
# read selected rows
wb_to_df(xl, rows = c(2, 4, 6))
#>    Var1 Var2 <NA>  Var3  Var4       Var5 Var6 Var7     Var8
#> 4  TRUE   NA   NA #NUM!     b 2023-05-23   NA    0 14:02:57
#> 6 FALSE    2   NA  <NA> #NUM!       <NA>   NA    2 17:24:53
```

4.1.8 `convert` - convert input to guessed type

In xml exists no difference between value types. All values are per default characters. To provide these as numerics, logicals or dates, `openxlsx2` and every other software dealing with xlsx files has to make assumptions about the cell type. This is especially tricky due to the notion of worksheets. Unlike in a data frame, a worksheet can have a wild mix of all types of data. Even though the conversion process from character to date or numeric is rather solid, sometimes the user might want to see the data without any conversion applied. This might be useful in cases where something unexpected happened or the import created warnings. In such a case you can look at the raw input data. If you want to disable date detection as well, please see the entry above.

```
# convert characters to numerics and date (logical too?)
wb_to_df(xl, convert = FALSE)
#>     Var1 Var2 <NA>  Var3  Var4       Var5         Var6    Var7     Var8
#> 3   TRUE    1 <NA>     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE <NA> <NA> #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2 <NA>  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2 <NA>  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3 <NA>  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1 <NA>   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9   <NA> <NA> <NA>  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
#> 10 FALSE    2 <NA>    23     h 2023-12-24         <NA>      25     <NA>
#> 11 FALSE    3 <NA>  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12  <NA>    1 <NA>   123  <NA> 2023-07-31         <NA>     122     <NA>
```

4.1.9 `skip_empty_rows` - remove empty rows

Even though `openxlsx2` imports everything as requested, sometimes it might be helpful to remove empty lines from the data. These might be either left empty intentional or empty because they are were formatted, but the cell value was removed afterwards. This was added mostly for backward comparability, but the default has been changed to `FALSE`. The behavior has changed a bit as well. Previously empty cells were removed prior to the conversion to R data frames, now they are removed after the conversion and are removed only if they are completely empty

```
# erase empty rows from dataset
wb_to_df(xl, sheet = 1, skip_empty_rows = TRUE) |> tail()
#>     Var1 Var2 <NA> Var3  Var4       Var5 Var6 Var7     Var8
#> 6  FALSE    2   NA <NA> #NUM!       <NA> <NA>    2 17:24:53
#> 7  FALSE    3   NA 1.56     e       <NA> <NA> <NA>     <NA>
#> 8  FALSE    1   NA  1.7     f 2023-03-02 <NA>  2.7 08:45:58
#> 10 FALSE    2   NA   23     h 2023-12-24 <NA>   25     <NA>
#> 11 FALSE    3   NA 67.3     i 2023-12-25 <NA>    3     <NA>
#> 12    NA    1   NA  123  <NA> 2023-07-31 <NA>  122     <NA>
```

4.1.10 `skip_empty_cols` - remove empty columns

The same for columns

```
# erase empty columns from dataset
wb_to_df(xl, skip_empty_cols = TRUE)
#>     Var1 Var2  Var3  Var4       Var5         Var6    Var7     Var8
#> 3   TRUE    1     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE   NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9     NA   NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
#> 10 FALSE    2    23     h 2023-12-24         <NA>      25     <NA>
#> 11 FALSE    3  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12    NA    1   123  <NA> 2023-07-31         <NA>     122     <NA>
```

4.1.11 `row_names` - keep rownames from input

Sometimes the data source might provide rownames as well. In such a case you can `openxlsx2` to treat the first column as rowname

```
# convert first row to rownames
wb_to_df(xl, sheet = 2, dims = "C6:G9", row_names = TRUE)
#>                mpg cyl disp  hp
#> Mazda RX4     21.0   6  160 110
#> Mazda RX4 Wag 21.0   6  160 110
#> Datsun 710    22.8   4  108  93
```

4.1.12 `types` - convert column to specific type

If the user know better than the software what type to expect in a worksheet, this can be provided via types. This parameter takes a named numeric. `0` is character, `1` is numeric and `2` is date

```
# define type of the data.frame
wb_to_df(xl, cols = c(2, 5), types = c("Var1" = 0, "Var3" = 1))
#>     Var1   Var3
#> 3   TRUE   1.00
#> 4   TRUE    NaN
#> 5   TRUE   1.34
#> 6  FALSE     NA
#> 7  FALSE   1.56
#> 8  FALSE   1.70
#> 9   <NA>     NA
#> 10 FALSE  23.00
#> 11 FALSE  67.30
#> 12  <NA> 123.00
```

4.1.13 `start_row` - where to begin

Often the creator of the worksheet has used a lot of creativity and the data does not begin in the first row, instead it begins somewhere else. To define the row where to begin reading, define it via the `start_row` parameter

```
# start in row 5
wb_to_df(xl, start_row = 5, col_names = FALSE)
#>        B  C  D      E     F          G  H       I        J
#> 5   TRUE  2 NA   1.34     c 2023-02-01 NA #VALUE! 23:01:02
#> 6  FALSE  2 NA     NA #NUM!       <NA> NA       2 17:24:53
#> 7  FALSE  3 NA   1.56     e       <NA> NA    <NA>     <NA>
#> 8  FALSE  1 NA   1.70     f 2023-03-02 NA     2.7 08:45:58
#> 9     NA NA NA     NA  <NA>       <NA> NA    <NA>     <NA>
#> 10 FALSE  2 NA  23.00     h 2023-12-24 NA      25     <NA>
#> 11 FALSE  3 NA  67.30     i 2023-12-25 NA       3     <NA>
#> 12    NA  1 NA 123.00  <NA> 2023-07-31 NA     122     <NA>
```

4.1.14 `na.strings` - define missing values

There is the "#N/A" string, but often the user will be faced with custom missing values and other values we are not interested. Such strings can be passed as character vector via

`na.strings`

```
# na strings
wb_to_df(xl, na.strings = "")
#>     Var1 Var2 <NA>  Var3  Var4       Var5          Var6    Var7     Var8
#> 3   TRUE    1   NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE   NA   NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2   NA  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2   NA  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3   NA  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1   NA   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9     NA   NA   NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
#> 10 FALSE    2   NA    23     h 2023-12-24         <NA>      25     <NA>
#> 11 FALSE    3   NA  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12    NA    1   NA   123  <NA> 2023-07-31         <NA>     122     <NA>
```

4.1.15 Importing as workbook

In addition to importing directly from xlsx or xlsm files, `openxlsx2` provides the `wbWorkbook` class used for importing and modifying entire the openxml files in `R`. This `workbook` class is the heart of `openxlsx2` and probably the reason why you are reading this manual in the first place.

Importing a file into a workbook looks like this:

```
# the file we are going to load
xl <- system.file("extdata", "openxlsx2_example.xlsx", package = "openxlsx2")
# loading the file into the workbook
wb <- wb_load(file = xl)
```

The additional options `wb_load()` provides are for internal use: `sheet` loads only a selected sheet from the workbook and `data_only` reads only the data parts from a workbook and ignores any additional graphics or pivot tables. Both functions create workbook objects that can only be used to read data, and we do not recommend end users to use them. Especially not if they intend to re-export the workbook afterwards.

Once a workbook is imported, we provide several functions to interact with and modify it (the `wb_to_df()` function mentioned above works the same way for an imported workbook). It is possible to add new sheets and remove sheets, as well as to add or remove data. R-plots can be inserted and also the style of the workbook can be changed, new fonts, background colors and number formats. There is a wealth of options explained in the man pages and the additional style vignette (more vignettes to follow).

4.2 Example: Reading real world data

In the lines above we have seen various ways how to read data. There is just one downside, actual real world data is usually not as nice and simple as the data we have seen above. Real world data has often features that help us humans to understand and interpret tables, like headlines that span across multiple rows and columns, or descriptions before the data, and footnotes after the data. In addition it is often mixed with totals and subtotals, so even if the data is imported, it still requires a lot of data cleaning. There are ways how `openxlsx2` can help in this regard. And while not necessarily required, actually looking at the data in a spreadsheet software can help with its understanding.

The file we use is part of the publications from the US Census. "Table 1. Full-Time, Year-Round Workers by Education, Sex, and Detailed Occupation: ACS 2022." At the time available at: https://www.census.gov/data/tables/2022/demo/acs-2022.html. The table is rather large with multiple groups in columns and rows. [1].

4.2.1 Reading the data table

In a first step we import the entire workbook

```
a <- "https://www2.census.gov/programs-surveys/demo/tables/industry-occupation"
b <- "2022/Detailed_occupation_by_sex_and_education_ACS_2022_tab1.xlsx"
fl <- file.path(a, b)

wb <- wb_load(fl)
```

Once the workbook is loaded, we read the entire worksheet and try to get an understanding how it looks like. For this we fill merged cells and remote the column name. After that we can inspect it with `View()`.

```
df <- wb_to_df(wb, fill_merged_cells = TRUE, col_names = FALSE,
               skip_empty_cols = TRUE)
# View(df)
```

Using this we realize that the table has a few description rows ahead and a few footnotes below. Numeric data starts in row `8` and ends in `605`. The table spans columns `A` to `BG`. Therefore our dimensions will look like this `A7:BG605`. We start one row earlier than the data we want to read, because we expect a column name. In addition there are a few missing values that we want to remove.

[1] A backup of the file can be found here https://janmarvin.github.io/openxlsx-data/Detailed_occupation_by_sex_and_education_ACS_2022_tab1.xlsx.

```
dims <- "A7:BG605"
df <- wb_to_df(wb, dims = dims, na.strings = c("-", "**", "#N/A"),
               fill_merged_cells = TRUE)
```

4.2.2 Cleaning the indents

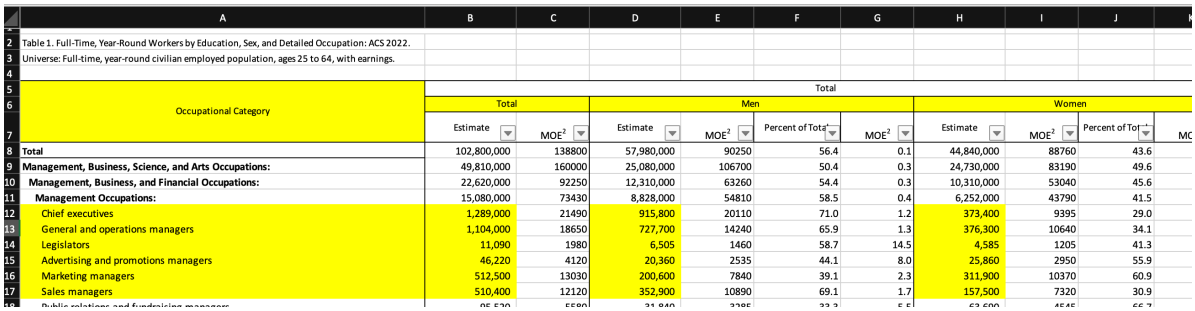
Not for every feature there is a custom function, but a lot of things can be done with `openxlsx2`. The rows are grouped by occupation. This occupation is not visisble in the data frame above. But since the information is available in the data, we can access it.

```
## adaption of https://github.com/JanMarvin/openxlsx2/discussions/710
description_dims <- wb_dims(rows = as.integer(rownames(df)), cols = 1)
text <- wb_to_df(wb, dims = description_dims, col_names = FALSE)[[1]]
want <- wb$get_cell_style(dims = description_dims)

# Get the styles for the range
styles <- wb$styles_mgr$styles$cellXfs[as.integer(want) + 1]

# function to replace "" with "0"
zeros   <- function(x) replace(x, x == "", "0")

# now get the indentation alignment from the style
indents <- openxlsx2:::read_xf(read_xml(styles))$indent |>
  zeros() |>
  as.integer()

# indent the text
itext <- NULL
for (i in seq_along(indents)) {
  if (!is.na(indents[i])) {
    itmp <- paste0(c(rep("__", indents[i]), text[i]), collapse = "")
    itext <- c(itext, itmp)
  } else {
    itext <- c(itext, text[i])
  }
}

# return it
message(paste(head(itext, 10), collapse = "\n"))
#> Total
#> Management, Business, Science, and Arts Occupations:
```

```
#> __Management, Business, and Financial Occupations:
#> ____Management Occupations:
#> ______Chief executives
#> ______General and operations managers
#> ______Legislators
#> ______Advertising and promotions managers
#> ______Marketing managers
#> ______Sales managers

rownames(df) <- itext
df$indents   <- indents

# quite a long list and I am not sure every item has the correct indentation
# in the spreadsheet
df_ind1 <- df[df$indents == 1, c("Estimate", "MOE2")]
head(df_ind1)
#>                                                                    Estimate
#> __Management, Business, and Financial Occupations:                22620000
#> __Computer, Engineering, and Science Occupations:                   9304000
#> __Education, Legal, Community Service, Arts, and Media Occupations: 11030000
#> __Chiropractors                                                       43980
#> __Dentists                                                            96470
#> __Dietitians and nutritionists                                        70760
#>                                                                     MOE2
#> __Management, Business, and Financial Occupations:                92250
#> __Computer, Engineering, and Science Occupations:                  74300
#> __Education, Legal, Community Service, Arts, and Media Occupations: 63890
#> __Chiropractors                                                     4205
#> __Dentists                                                          5520
#> __Dietitians and nutritionists                                      5135
```

4.2.3 Read selected dims

Lets say you have opened the file in a spreadsheet software and identified a few cells that you want to read. You don't want to read every cell, only a few occupations, and total estimates for man and woman. Basically you have decided, that you want to import the cells with yellow highlighting in the following screenshot.

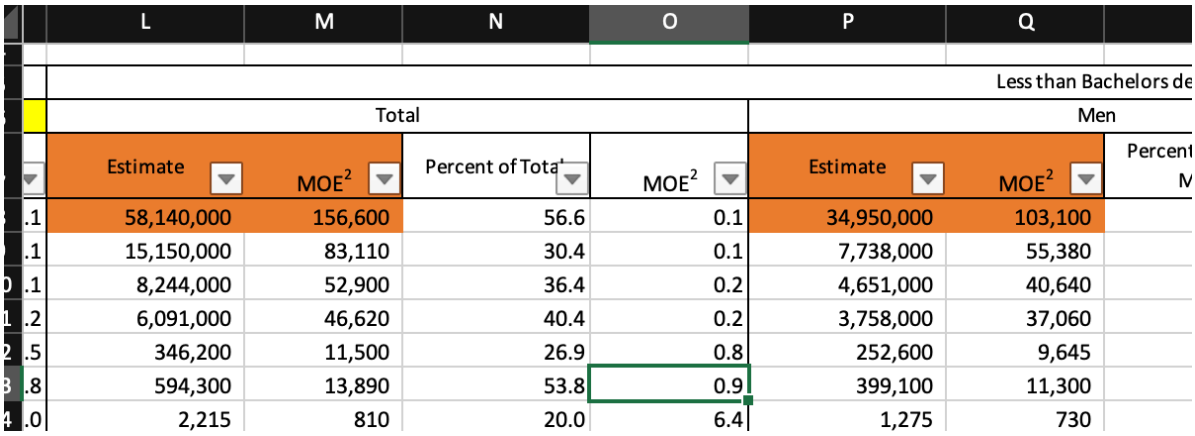


	A	B	C	D	E	F	G	H	I	J
2	Table 1. Full-Time, Year-Round Workers by Education, Sex, and Detailed Occupation: ACS 2022.									
3	Universe: Full-time, year-round civilian employed population, ages 25 to 64, with earnings.									
4										
5		Total								
6	Occupational Category	Total		Men				Women		
7		Estimate	MOE²	Estimate	MOE²	Percent of Total	MOE²	Estimate	MOE²	Percent of Tot
8	Total	102,800,000	138800	57,980,000	90250	56.4	0.1	44,840,000	88760	43.6
9	Management, Business, Science, and Arts Occupations:	49,810,000	160000	25,080,000	106700	50.4	0.3	24,730,000	83190	49.6
10	Management, Business, and Financial Occupations:	22,620,000	92250	12,310,000	63260	54.4	0.3	10,310,000	53040	45.6
11	Management Occupations:	15,080,000	73430	8,828,000	54810	58.5	0.4	6,252,000	43790	41.5
12	Chief executives	1,289,000	21490	915,800	20110	71.0	1.2	373,400	9395	29.0
13	General and operations managers	1,104,000	18650	727,700	14240	65.9	1.3	376,300	10640	34.1
14	Legislators	11,090	1980	6,505	1460	58.7	14.5	4,585	1205	41.3
15	Advertising and promotions managers	46,220	4120	20,360	2535	44.1	8.0	25,860	2950	55.9
16	Marketing managers	512,500	13030	200,600	7840	39.1	2.3	311,900	10370	60.9
17	Sales managers	510,400	12120	352,900	10890	69.1	1.7	157,500	7320	30.9

Since your output contains non consecutive cells, which are basically a square, you can pass them as a single `dims` string. In the `dims` string we treat header and column differently. We can check with `dims_to_dataframe("A6:B6,D6,H6,A12:B17,D12:D17,H12:H17", fill = TRUE, empty_rm = TRUE)` if our `dims` object works. Since we see no blanks, every cell is matched.

```
wb_to_df(
  wb,
  dims = "A6:B6,D6,H6,A12:B17,D12:D17,H12:H17",
  fill_merged_cells = TRUE
)
#>                   Occupational Category   Total   Men   Women
#> 12                     Chief executives 1289000 915800 373400
#> 13     General and operations managers 1104000 727700 376300
#> 14                           Legislators   11090   6505   4585
#> 15 Advertising and promotions managers   46220  20360  25860
#> 16                   Marketing managers  512500 200600 311900
#> 17                       Sales managers  510400 352900 157500
```

And another table, this time without a separate header row.



	L	M	N	O	P	Q	
							Less than Bachelors de
		Total			Men		
	Estimate	MOE²	Percent of Total	MOE²	Estimate	MOE²	Percent M
	.1 58,140,000	156,600	56.6	0.1	34,950,000	103,100	
	.1 15,150,000	83,110	30.4	0.1	7,738,000	55,380	
0	.1 8,244,000	52,900	36.4	0.2	4,651,000	40,640	
1	.2 6,091,000	46,620	40.4	0.2	3,758,000	37,060	
2	.5 346,200	11,500	26.9	0.8	252,600	9,645	
3	.8 594,300	13,890	53.8	0.9	399,100	11,300	
4	.0 2,215	810	20.0	6.4	1,275	730	

To read the orange cells, the following command can be used:

```
wb_to_df(
  wb,
  dims = "A7:A8,L7:M8,P7:Q8",
  fill_merged_cells = TRUE
)
#>   Occupational Category Estimate   MOE2 Estimate   MOE2
#> 8                 Total 58140000 156600 34950000 103100
```

4.2.4 Read data header and body in parts

Using only the last row right above the data, can result in many duplicated column names. It is not always possible to avoid this, but sometimes it is possible to create a unique name combining the multiple header rows. In the code below, we read two `df` objects, the `df_head` and the `df_body`. Once the data is imported, it is straightforward to modify the `df_head` object to create a unique column name.

```
# read body
dims <- "A8:BG605"
df_body <- wb_to_df(wb, dims = dims, na.strings = c("-", "**", "#N/A"),
                    fill_merged_cells = TRUE, col_names = FALSE)

# read header
dims <- "A5:BG7"
df_head <- wb_to_df(wb, dims = dims, na.strings = c("-", "**", "#N/A"),
                    fill_merged_cells = TRUE, col_names = FALSE)

# create single header string. remove all spaces, unique values. collapse on dot
nams <- vapply(names(df_head), function(x) {
  paste0(gsub("[ \t\r\n]", "_", unique(trimws(df_head[[x]]))), collapse = ".")
  }, NA_character_)

# check that names in body and head match
stopifnot(all(names(df_body) %in% names(nams)))

# assign names and create output object, avoid duplicates
df <- setNames(df_body, make.names(nams, unique = TRUE))

# a few names of the workbook
head(names(df), 10)
#>  [1] "Occupational_Category"        "Total.Estimate"
```

```
#>  [3] "Total.MOE2"                   "Total.Men.Estimate"
#>  [5] "Total.Men.MOE2"               "Total.Men.Percent_of_Total"
#>  [7] "Total.Men.MOE2.1"             "Total.Women.Estimate"
#>  [9] "Total.Women.MOE2"             "Total.Women.Percent_of_Total"

# a glimpse of the new object
head(df[seq_len(5)])
#>                                 Occupational_Category Total.Estimate
#> 8                                                Total      102800000
#> 9  Management, Business, Science, and Arts Occupations:       49810000
#> 10     Management, Business, and Financial Occupations:       22620000
#> 11                               Management Occupations:       15080000
#> 12                                     Chief executives        1289000
#> 13                     General and operations managers        1104000
#>    Total.MOE2 Total.Men.Estimate Total.Men.MOE2
#> 8      138800           57980000          90250
#> 9      160000           25080000         106700
#> 10      92250           12310000          63260
#> 11      73430            8828000          54810
#> 12      21490             915800          20110
#> 13      18650             727700          14240
```

Given enough knowledge about certain data files, it is often possible to identify cells, similar to `VLOOKUP()` in spreadsheets. In our case, we could maybe make use of e.g. `which(df$A == "Total")` or `int2col(which(df[6, ] == "Men"))`. But such cases require a bit more hand tailored solutions. From experience the most important thing is to remain doubtful about the data imported. There are many things that can go wrong, like picking the wrong column, or the wrong spreadsheet. Don't be shy to check your work against spreadsheet software. Again and again.

4.2.5 Bonus: clean up this xlsx table

Obviously something is wrong in the xlsx file. We have already worked with the data, so lets see if we can clean it up.

```
# fix some broken indentation in the file - this is only to please my OCD
sel <- seq.int(
  which(text == "Healthcare Practitioners and Technical Occupations:"),
  which(text == "Other healthcare practitioners and technical occupations")
)
indents[sel] <- indents[sel] + 1L
```

```r
sel <- seq.int(
  which(text == "Healthcare Practitioners and Technical Occupations:") + 1L,
  which(text == "Other production workers")
)
indents[sel] <- indents[sel] + 1L

create_groups <- function(sequence) {

  # Create a data frame
  df <- data.frame(Index = seq_along(sequence), Value = sequence)

  # Calculate Supergroup, Group, and Subgroup identifiers
  df$Supergroup <- cumsum(df$Value == 0)
  df$Group      <- cumsum(df$Value == 1)
  df$Subgroup   <- cumsum(df$Value == 2)

  # Fill NA values for non-group entries
  df$Supergroup <- ifelse(df$Value == 0, df$Supergroup, NA)
  df$Group      <- ifelse(df$Value == 1, df$Group, NA)
  df$Subroup    <- ifelse(df$Value == 2, df$Group, NA)

  #nolint start
  as.data.frame(
    tidyr::fill(df, Supergroup, Group, Subgroup, .direction = "down")
  )
  #nolint end
}

df$Index <- seq_len(nrow(df))
df$Value <- indents

# each duplicated MOE2 is a percent value
df <- merge(x = df, y = create_groups(indents), by = c("Index", "Value"),
            sort = FALSE)

vars <- c("Occupational_Category", "Total.Estimate", "Total.MOE2",
          "Supergroup", "Group", "Subgroup")
tab <- df[df$Value == 3, vars]
rownames(tab) <- NULL

aggregate(Total.Estimate ~ Supergroup, data = tab, sum)
#>   Supergroup Total.Estimate
```

```
#> 1          2      102819245
aggregate(Total.Estimate ~ Group, data = tab, sum)
#>   Group Total.Estimate
#> 1     1       22622315
#> 2     2        9304335
#> 3     3       11029725
#> 4     4        6853765
#> 5     5       11841230
#> 6     6       18861045
#> 7     7        9240130
#> 8     8        6044450
#> 9     9        7022250
```

5 Of strings and numbers

Contrary to R, spreadsheets do not require identical data types. While in R a column always consists of a unique type (the base types supported by openxlsx2 are character, integer, numeric, Date, and POSIXct/POSIXlt), spreadsheets might consist of arbitrary mixes of data types. E.g. it is not uncommon, to have tables consisting of multiple rows. In addition some spreadsheet software has issues identifying certain date types and a well known issue of spreadsheets is the number stored as text error. Below we will describe ways to write data with openxlsx2 and how to handle the most common types characters and numerics. Though in addition openxlsx2 also supports dates, date formats and makes use of the hms date class.

```
wb <- wb_workbook()
```

5.1 Default numeric data frame

Using a few rows of the cars data frame we show how to write numerics. The strings are left aligned and the numbers right aligned.

```
# default data frame
dat <- data.frame(
  speed = c(4, 4, 7, 7, 8, 9),
  dist = c(2, 10, 4, 22, 16, 10)
)

# Consisting only of numerics
str(dat)
#> 'data.frame':    6 obs. of  2 variables:
#>  $ speed: num  4 4 7 7 8 9
#>  $ dist : num  2 10 4 22 16 10

wb$add_worksheet("dat")$add_data(x = dat)
```

5.2 Writing missing values

Writing missing values to a spreadsheet (`NA`, `NA_character_`, `NA_integer_`, and `NA_real_`) results in the missing value to appear as the `#N/A` expression in spreadsheet software. Still there are multiple ways to create missing values, below are the three common solutions. If the default is unwanted `na.strings = NULL`, creates a blank cell and `na.strings = "N/A"` creates a character string `"N/A"`. There is a subtle difference between `na.strings = NULL` and `na.strings = ""`. The latter creates a string `""` whereas the former leaves the cell mostly untouched, aka there is no cell type attached to it. Unless some form of styling is attached to such a cell, it will be omitted when saving the file as `xlsx`. This reduces the file size of these sparse matrices significantly, because only cells that contain some kind of information will be written to the output.

```
# example matrix
mm <- matrix(seq_len(9), 3, 3)
diag(mm) <- NA

dims_1 <- wb_dims(x = mm)
dims_2 <- wb_dims(x = mm, from_dims = dims_1, right = 2)
dims_3 <- wb_dims(x = mm, from_dims = dims_2, right = 2)

wb$add_worksheet("missings")
# the default writes the expression #NA
wb$add_data(dims = dims_1, x = mm)$add_border(dims = dims_1)
# writes nothing, keeps the cell blank
wb$add_data(dims = dims_2, x = mm, na.strings = NULL)$add_border(dims = dims_2)
# writes the string N/A
wb$add_data(dims = dims_3, x = mm, na.strings = "N/A")$add_border(dims = dims_3)
```

	A	B	C	D	E	F	G	H	I	J	K
1	V1	V2	V3		V1	V2	V3		V1	V2	V3
2	#N/A	4	7			4	7		N/A	4	7
3	2	#N/A	8		2		8		2	N/A	8
4	3	6	#N/A		3	6			3	6	N/A
5											

5.3 Writing vectors

When writing vectors, the default direction is vertically. But this can be changed. It is possible to write vectors horizontally, if this is indicated via `dims`. In addition it is possible to `enforce` non-consecutive dimensions.

```
wb$add_worksheet("vectors")

# vertical
wb$add_data(x = 1:4)

# horizontal
wb$add_data(x = 1:4, dims = "C2:F2")

# mixed
wb$add_data(x = 1:4, dims = "C5,D4,E5,F4", enforce = TRUE)
```

5.4 Data frame with multiple row header

Now we alter the data frame with a second row adding the column label. Since `R` does not know mixed column types the entire data frame is converted to characters.

```
# add subtitle to the data
dat_w_subtitle <- data.frame(
  speed = c("Speed (mph)", 4, 4, 7, 7, 8, 9),
  dis = c("Stopping distance (ft)", 2, 10, 4, 22, 16, 10)
)
# Check that both columns are character
str(dat_w_subtitle)
#> 'data.frame':    7 obs. of  2 variables:
#>  $ speed: chr  "Speed (mph)" "4" "4" "7" ...
#>  $ dis  : chr  "Stopping distance (ft)" "2" "10" "4" ...

# write data as is. this creates number stored as text error
# this can be surpressed with: wb_add_ignore_error(number_stored_as_text)
wb$add_worksheet("dat_w_subtitle")$add_data(x = dat_w_subtitle)
```

Now the data is written as strings. Therefore the numbers are not written as `4`, but as `"4"`. In the openxml format characters are treated differently as numbers and are stored as inline strings (`openxlsx2` default) or as shared string. The file loads fine, but now all cells are right aligned and the previous numeric cells are all showing the number stored as text error. Spreadsheet software will treat these cells independently of the data type, so it does not matter other that the error is thrown and that number formats are not applied.

Since conversions to character are sometimes not wanted, we provide a way to detect these numbers stored as text and will convert them when the data is written into the workbook.

```
# write character string, but write string numbers as numerics
options("openxlsx2.string_nums" = TRUE)
wb$add_worksheet("string_nums")$add_data(x = dat_w_subtitle)
options("openxlsx2.string_nums" = NULL)
```

This way the data is written as numerics, but still right aligned. This is due to the cell style, otherwise it looks entirely identical to previous attempt. Since this conversion is not generally wanted this option needs to be enabled explicitly. Generally `openxlsx2` assumes that the users are mature and want what they request.

5.5 How to write multiple header rows?

The better approach to avoid the entire conversion is to write the column headers and the column data separately. The recommended approach to this would be something like this:

```
wb$add_worksheet("characters and numbers")$
  add_data(x = dat_w_subtitle[1, ])$
  add_data(dims = wb_dims(x = dat, col_names = FALSE, from_row = 3),
           x = dat, col_names = FALSE)
```

5.6 Labelled data

In addition to pure `numbers` and `characters` it is also possible to write `labelled` vectors such as factors or columns modified with the `labelled` package.

```
# Factors
x <- c("Man", "Male", "Man", "Lady", "Female")
xf <- factor(x, levels = c("Male", "Man", "Lady", "Female"),
             labels = c("Male", "Male", "Female", "Female"))

wb$add_worksheet("factors")$add_data(x = data.frame(x, xf))

# Labelled
v <- labelled::labelled(
  c(1, 2, 2, 2, 3, 9, 1, 3, 2, NA),
  c(yes = 1, no = 3, "don't know" = 8, refused = 9)
)

wb$add_worksheet("labelled")$add_data(x = v)
```

5.7 Hour - Minute - Second

If the `hms` package is loaded `openxlsx2` makes use of this as well. Otherwise the data would be returned as

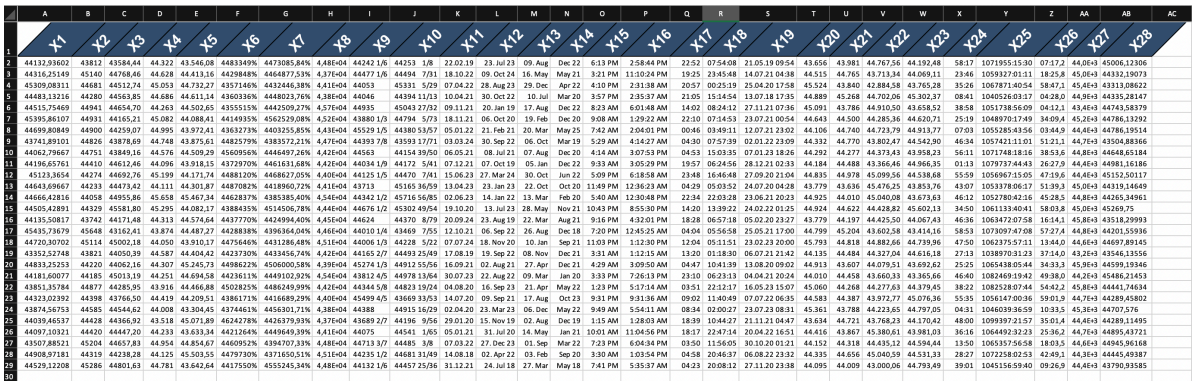
```
set.seed(123)
wb$add_worksheet("hms")$add_data(x = hms::hms(sample(1:100000, 5, TRUE)))

df <- wb_to_df(wb, sheet = "hms")
str(df)
#> 'data.frame':    4 obs. of  1 variable:
#>  $ 14:21:03: 'hms' num  16:04:30 00:49:46 08:18:45 26:27:26
#>   ..- attr(*, "tzone")= chr "UTC"


unloadNamespace("hms")
df <- wb_to_df(wb, sheet = "hms")
str(df)
#> 'data.frame':    4 obs. of  1 variable:
#>  $ 14:21:03: chr  "16:04:30" "00:49:46" "08:18:45" "02:27:26"
```

6 Styling of worksheets

Welcome to the styling manual for `openxlsx2`. In this manual you will learn how to use `openxlsx2` to style your worksheets. data from xlsx-files to R as well as how to export data from R to xlsx, and how to import and modify these openxml workbooks in R.

6.1 Colors, text rotation and number formats

Below we show you two ways how to create styled tables with `openxlsx2` one using the high level functions to style worksheet areas and one using the bare metal approach of creating the identical table. We show both ways to create styles in `openxlsx2` to show how you could build on our functions or create your very own functions.



	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19	X20	X21	X22	X23	X24	X25	X26	X27	X28
2	44132,93602	43812	43584,44	44.322	43.546,08	4483349%	4473085,84%	4,48E+04	44242 1/6	44253 1/8	22.02.19	23. Jul 23	09. Aug	Dec 22	6:13 PM	2:58:44 PM	22:52	07:54:08	21.05.19 09:54	43.656	43.981	44.767,56	44.192,48	58:17	1071955:15:30	07:17,2	44,0E+3	45006,12306
3	44316,25149	45140	44768,46	44.628	44.413,16	4429848%	4464877,53%	4,37E+04	44477 1/6	44494 7/31	18.10.22	09. Oct 24	16. May	May 21	3:21 PM	11:10:24 PM	19:25	23:45:48	14.07.21 04:38	44.515	44.765	43.713,34	44.069,11	23:46	1059327:01:11	18:25,8	45,0E+3	44332,19073
4	45309,08311	44681	44512,74	45.053	44.732,27	4357146%	4432446,38%	4,41E+04	44053	45331 5/29	07.04.22	28. Aug 23	29. Dec	Apr 22	4:10 PM	2:31:38 AM	20:57	00:25:19	25.04.20 17:58	45.524	43.840	42.884,58	43.765,28	35:26	1067871:40:54	58:47,1	45,4E+3	43313,08622
5	44483,13216	44280	44563,85	44.686	44.611,14	4360336%	4448023,76%	4,38E+04	44046	44394 11/13	10.04.21	30. Oct 22	10. Jul	Mar 20	3:57 PM	2:35:37 AM	21:05	15:14:54	13.07.18 17:35	44.889	45.268	44.702,06	45.302,37	08:41	1040526:03:17	04:28,0	44,9E+3	44335,28147
6	44515,75469	44941	44654,70	44.263	44.502,65	4355515%	4442509,27%	4,57E+04	44935	45043 27/32	09.11.21	20. Jan 19	17. Aug	Dec 22	8:23 AM	6:01:48 AM	14:02	08:24:12	27.11.21 07:36	45.091	43.786	44.910,50	43.658,52	38:58	1051738:56:09	04:12,1	43,4E+3	44743,58379
7	45395,86107	44931	44165,21	45.082	44.088,41	4414935%	4562529,08%	4,52E+04	43880 1/3	44794 5/73	18.11.21	06. Oct 20	19. Feb	Dec 20	9:08 AM	1:29:22 AM	22:10	07:14:53	23.07.21 00:54	44.643	44.500	44.285,36	44.620,71	25:19	1048970:17:49	34:09,4	45,2E+3	44786,13292
8	44699,80849	44900	44259,07	44.995	43.972,41	4363273%	4403255,85%	4,43E+04	45529 1/5	44380 53/57	05.01.22	21. Feb 21	20. Mar	May 25	7:42 AM	2:04:01 PM	00:46	03:49:11	12.07.21 23:02	44.106	44.740	44.723,79	44.913,77	07:03	1055285:43:56	03:44,9	44,4E+3	44786,19514
9	43741,89101	44826	43878,69	44.748	43.875,61	4482579%	4383572,21%	4,47E+04	44393 7/8	43593 17/71	03.03.24	30. Sep 22	06. Oct	Mar 19	5:29 AM	4:14:27 AM	04:30	07:57:39	02.01.22 23:09	44.332	44.770	43.802,47	44.542,90	46:34	1057421:11:01	51:21,1	44,7E+3	43504,88366
10	44062,79667	44751	43849,16	44.576	44.509,29	4560956%	4446497,26%	4,42E+04	44563	44154 39/50	06.05.21	08. Jul 21	07. Aug	Dec 20	4:14 AM	3:07:53 PM	04:53	15:03:35	07.01.23 18:26	44.292	44.277	44.373,43	43.958,23	56:11	1071748:18:16	38:53,6	44,8E+3	44648,65184
11	44196,65761	44410	44612,46	44.096	43.918,15	4372970%	4461631,68%	4,42E+04	44034 1/9	44172 5/41	07.12.21	07. Oct 19	05. Jan	Dec 22	9:33 AM	3:05:29 PM	19:57	06:24:56	28.12.21 02:33	44.184	44.488	43.366,46	44.966,35	01:13	1079737:44:43	26:27,9	44,4E+3	44981,16186
12	45123,3654	44274	44692,76	45.199	44.171,74	4488120%	4468627,05%	4,40E+04	44125 1/5	44470 7/41	15.06.23	27. Mar 24	30. Oct	Jun 22	5:09 PM	6:18:58 AM	23:48	16:46:48	27.09.20 21:04	44.835	44.978	45.099,56	44.538,68	55:59	1056967:15:05	47:19,6	44,4E+3	45152,50117
13	44643,69667	44233	44473,42	44.111	44.301,87	4487082%	4418960,72%	4,41E+04	43713	45165 36/59	13.04.23	23. Jan 23	22. Oct	Oct 20	11:49 PM	12:36:23 AM	04:29	05:03:52	24.07.20 04:28	43.779	43.636	45.476,25	43.853,76	43:07	1053378:06:17	51:39,3	45,0E+3	44319,14649
14	44666,42816	44058	44955,86	45.658	45.467,34	4462837%	4385385,40%	4,54E+04	44342 1/2	45716 56/85	02.06.23	14. Jan 22	13. Mar	Feb 20	5:40 AM	12:30:48 PM	22:34	22:03:28	23.06.21 20:23	44.925	44.010	45.040,08	43.673,63	46:12	1052780:42:16	45:28,5	44,8E+3	44265,34961
15	44505,42891	44329	45581,80	45.295	44.082,17	4388435%	4514506,78%	4,44E+04	44676 1/2	45302 49/54	19.10.20	13. Jul 23	28. May	Nov 21	10:43 PM	8:55:30 PM	14:20	13:39:22	24.02.22 01:25	44.924	44.622	44.428,82	45.602,13	34:50	1061133:40:41	58:03,8	45,0E+3	45269,75
16	44135,50817	43742	44171,48	44.313	44.574,64	4437770%	4424994,40%	4,45E+04	44624	44370 8/79	20.09.24	23. Aug 19	22. Mar	Aug 21	9:16 PM	4:32:01 PM	18:28	06:57:18	05.02.20 23:27	43.779	44.197	44.425,50	44.067,43	46:36	1063472:07:58	16:14,1	45,8E+3	43518,29993
17	45435,73679	45648	43162,41	43.874	44.487,27	4428838%	4396364,04%	4,46E+04	44010 1/4	43469 7/55	12.10.21	06. Sep 22	26. Aug	Dec 18	7:20 PM	12:45:25 AM	04:04	05:56:58	25.05.21 17:00	44.799	45.204	43.602,58	43.414,16	58:53	1073097:47:08	57:27,4	44,8E+3	44201,55936
18	44720,30702	45114	45002,18	44.050	43.910,17	4475646%	4431286,48%	4,51E+04	44006 1/3	44228 5/22	07.07.24	18. Nov 20	10. Jan	Sep 21	11:03 PM	1:12:30 PM	12:04	05:11:51	23.02.23 20:00	45.793	44.818	44.882,66	44.739,96	47:50	1062375:57:11	13:44,0	44,6E+3	44697,89145
19	43352,52748	43821	44050,39	44.587	44.404,42	4423730%	4433456,74%	4,42E+04	44165 2/7	44493 25/49	17.08.19	19. Sep 22	08. Nov	Dec 21	3:31 AM	1:12:15 AM	13:20	01:18:30	06.07.21 21:42	44.135	44.484	44.327,04	44.616,18	27:13	1038970:31:23	37:14,0	43,2E+3	43546,13556
20	44833,25253	44220	44062,16	44.307	45.245,73	4498622%	4506000,58%	4,39E+04	45274 1/3	44912 55/56	16.09.21	02. Aug 21	27. Apr	Dec 21	4:29 AM	3:09:50 AM	04:47	10:41:39	13.08.20 09:02	44.913	43.607	44.079,51	43.692,62	25:25	1065438:05:44	34:33,3	45,9E+3	44599,19346
21	44181,60077	44185	45013,19	44.251	44.694,58	4423611%	4449102,92%	4,54E+04	43812 4/5	44978 13/64	30.07.23	22. Aug 22	09. Mar	Jan 20	3:33 PM	7:26:13 PM	23:10	06:23:13	04.04.21 20:24	44.010	44.458	43.660,33	43.365,66	46:40	1082469:19:42	49:38,0	44,2E+3	45486,21453
22	43851,35784	44877	44285,95	43.916	44.466,88	4502825%	4486249,99%	4,42E+04	44344 5/8	44823 19/24	04.08.20	16. Sep 23	21. Apr	May 22	1:23 PM	5:17:14 AM	03:51	22:12:17	16.05.23 15:07	45.060	44.268	44.277,63	44.379,45	38:22	1082528:07:44	54:42,2	45,8E+3	44441,74634
23	44323,02392	44398	43766,50	44.419	44.209,51	4386171%	4416689,29%	4,40E+04	45499 4/5	43669 33/53	14.07.20	09. Sep 21	17. Aug	Oct 23	9:31 PM	9:31:36 AM	09:02	11:40:49	07.07.22 06:35	44.583	44.387	43.972,77	45.076,36	55:35	1056147:00:36	59:01,9	44,7E+3	44289,45802
24	43874,56753	44585	44544,62	44.008	43.304,45	4374461%	4456301,71%	4,38E+04	44388	44915 16/29	02.04.20	23. Mar 23	06. Dec	May 22	9:49 AM	5:54:11 AM	08:34	02:00:27	23.07.23 08:31	45.361	43.788	44.223,65	44.797,05	04:31	1046039:36:59	10:33,5	45,3E+3	44707,576
25	44039,46537	44428	44366,92	43.518	45.071,89	4624278%	4426379,93%	4,37E+04	43689 2/7	44196 9/56	29.01.20	15. Nov 19	02. Aug	Dec 19	1:15 AM	1:28:03 AM	18:39	10:44:27	21.11.21 04:47	43.634	44.721	43.768,23	44.170,42	48:00	1099397:21:57	35:01,4	44,4E+3	44289,11495
26	44097,10321	44420	44447,20	44.233	43.633,34	4421264%	4449649,39%	4,41E+04	44075	44541 1/65	05.01.21	31. Jul 20	14. May	Jan 21	10:01 AM	11:04:56 PM	18:17	22:47:14	20.04.22 16:51	44.416	43.867	45.380,61	43.981,03	36:16	1064492:32:23	25:36,2	44,7E+3	44895,43721
27	43507,88521	45204	44657,83	44.954	44.854,67	4460952%	4394707,33%	4,48E+04	44713 3/7	44485 3/8	07.03.22	27. Dec 23	01. Sep	Mar 22	7:23 PM	6:04:34 PM	03:50	11:56:05	30.10.20 01:21	44.152	44.318	44.435,12	44.594,44	13:50	1065357:56:58	18:03,5	44,6E+3	44945,96168
28	44908,97181	44319	44238,28	44.125	45.503,55	4479730%	4371650,51%	4,51E+04	44235 1/2	44681 31/49	14.08.18	02. Apr 22	03. Feb	Sep 20	3:30 AM	1:03:54 PM	04:58	20:46:37	06.08.22 23:32	44.335	44.656	45.040,59	44.531,33	28:27	1072258:02:53	42:49,1	44,3E+3	44445,49387
29	44529,12208	45286	44801,63	44.781	43.642,64	4417550%	4555245,34%	4,48E+04	44132 1/6	44457 25/36	31.12.21	24. Jul 18	27. Mar	May 18	7:41 PM	5:35:37 AM	04:23	20:08:12	27.11.20 23:38	44.095	44.009	43.000,06	44.793,49	39:01	1045156:59:40	09:26,9	44,4E+3	43790,93585

Figure 6.1: The example below, with increased column width.

6.1.1 the quick way: using high level functions

```
# add some dummy data
set.seed(123)
mat <- matrix(rnorm(28 * 28, mean = 44444, sd = 555), ncol = 28)
colnames(mat) <- make.names(seq_len(ncol(mat)))
border_col <- wb_color(theme = 1)
```

```r
border_sty <- "thin"

# prepare workbook with data and formated first row
wb <- wb_workbook() |>
  wb_add_worksheet("test") |>
  wb_add_data(x = mat) |>
  wb_add_border(dims = "A1:AB1",
    top_color = border_col, top_border = border_sty,
    bottom_color = border_col, bottom_border = border_sty,
    left_color = border_col, left_border = border_sty,
    right_color = border_col, right_border = border_sty,
    inner_hcolor = border_col, inner_hgrid = border_sty
  ) |>
  wb_add_fill(dims = "A1:AB1", color = wb_color(hex = "FF334E6F")) |>
  wb_add_font(dims = "A1:AB1", name = "Arial", bold = TRUE,
              color = wb_color(hex = "FFFFFFFF"), size = 20) |>
  wb_add_cell_style(dims = "A1:AB1", horizontal = "center", text_rotation = 45)

# create various number formats
x <- c(
  0, 1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
  37, 38, 39, 40, 45, 46, 47, 48, 49
)

# apply the styles
for (i in seq_along(x)) {
  cell <- sprintf("%s2:%s29", int2col(i), int2col(i))
  wb <- wb |> wb_add_numfmt(dims = cell, numfmt = x[i])
}

# wb$open()
```

6.1.2 the long way: using bare metal functions

```r
# create workbook
wb <- wb_workbook() |> wb_add_worksheet("test")

# add some dummy data to the worksheet
set.seed(123)
mat <- matrix(rnorm(28 * 28, mean = 44444, sd = 555), ncol = 28)
```

```r
colnames(mat) <- make.names(seq_len(ncol(mat)))
wb$add_data(x = mat, col_names = TRUE)

# create a border style and assign it to the workbook
black <- wb_color(hex = "FF000000")
new_border <- create_border(
  bottom = "thin", bottom_color = black,
  top = "thin", top_color = black,
  left = "thin", left_color = black,
  right = "thin", right_color = black
)
wb$styles_mgr$add(new_border, "new_border")


# create a fill style and assign it to the workbook
new_fill <- create_fill(patternType = "solid",
                        fgColor = wb_color(hex = "FF334E6F"))
wb$styles_mgr$add(new_fill, "new_fill")

# create a font style and assign it to the workbook
new_font <- create_font(sz = 20, name = "Arial", b = TRUE,
                        color = wb_color(hex = "FFFFFFFF"))
wb$styles_mgr$add(new_font, "new_font")

# create a new cell style, that uses the fill, the font and the border style
new_cellxfs <- create_cell_style(
  num_fmt_id    = 0,
  horizontal    = "center",
  text_rotation = 45,
  fill_id       = wb$styles_mgr$get_fill_id("new_fill"),
  font_id       = wb$styles_mgr$get_font_id("new_font"),
  border_id     = wb$styles_mgr$get_border_id("new_border")
)
# assign this style to the workbook
wb$styles_mgr$add(new_cellxfs, "new_styles")

# assign the new cell style to the header row of our data set
cell <- sprintf("A1:%s1", int2col(nrow(mat)))
wb <- wb |> wb_set_cell_style(
  dims = cell,
  style = wb$styles_mgr$get_xf_id("new_styles")
)
```

```r
## style the cells with some builtin format codes (no new numFmt entry is
## needed). add builtin style ids
x <- c(
  1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
  37, 38, 39, 40, 45, 46, 47, 48, 49
)

# create styles
new_cellxfs <- create_cell_style(num_fmt_id = x, horizontal = "center")

# assign the styles to the workbook
for (i in seq_along(x)) {
  wb$styles_mgr$add(new_cellxfs[i], paste0("new_style", i))
}

# new styles are 1:28
new_styles <- wb$styles_mgr$get_xf()
for (i in as.integer(new_styles$id[new_styles$name %in%
                                   paste0("new_style", seq_along(x))])) {
  cell <- sprintf("%s2:%s29", int2col(i), int2col(i))
  wb <- wb |> wb_set_cell_style(dims = cell, style = i)
}

# assign a custom tabColor
wb$worksheets[[1]]$sheetPr <- xml_node_create(
  "sheetPr",
  xml_children = xml_node_create(
    "tabColor",
    xml_attributes = wb_color(hex = "FF00FF00")
  )
)

# # look at the beauty you've created
# wb_open(wb)
```

6.2 Working with number formats

6.2.1 numfmts

Per default `openxlsx2` will pick up number formats for selected R classes.

```r
## Create Workbook object and add worksheets
wb <- wb_workbook()
wb$add_worksheet("S1")
wb$add_worksheet("S2")

df <- data.frame(
  "Date" = Sys.Date() - 0:19,
  "T" = TRUE, "F" = FALSE,
  "Time" = Sys.time() - 0:19 * 60 * 60,
  "Cash" = 1:20, "Cash2" = 31:50,
  "hLink" = "https://CRAN.R-project.org/",
  "Percentage" = seq(0, 1, length.out = 20),
  "TinyNumbers" = runif(20) / 1E9, stringsAsFactors = FALSE
)

## openxlsx will apply default Excel styling for these classes
class(df$Cash) <- c(class(df$Cash), "currency")
class(df$Cash2) <- c(class(df$Cash2), "accounting")
class(df$hLink) <- "hyperlink"
class(df$Percentage) <- c(class(df$Percentage), "percentage")
class(df$TinyNumbers) <- c(class(df$TinyNumbers), "scientific")

wb$add_data("S1", x = df, start_row = 4, row_names = FALSE)
wb$add_data_table("S2", x = df, start_row = 4, row_names = FALSE)
```

4	Date	T	F	Time	Cash	Cash2	hLink	Percentage	TinyNumbers
5	13.01.24	TRUE	FALSE	13.01.24 13:04	1.00 €	31.00	https://CRA	0.00%	93.7E-12
6	12.01.24	TRUE	FALSE	13.01.24 12:04	2.00 €	32.00	https://CRA	5.26%	865.1E-12
7	11.01.24	TRUE	FALSE	13.01.24 11:04	3.00 €	33.00	https://CRA	10.53%	276.6E-12
8	10.01.24	TRUE	FALSE	13.01.24 10:04	4.00 €	34.00	https://CRA	15.79%	27.3E-12
9	09.01.24	TRUE	FALSE	13.01.24 09:04	5.00 €	35.00	https://CRA	21.05%	787.4E-12
10	08.01.24	TRUE	FALSE	13.01.24 08:04	6.00 €	36.00	https://CRA	26.32%	358.7E-12
11	07.01.24	TRUE	FALSE	13.01.24 07:04	7.00 €	37.00	https://CRA	31.58%	25.1E-12
12	06.01.24	TRUE	FALSE	13.01.24 06:04	8.00 €	38.00	https://CRA	36.84%	746.1E-12
13	05.01.24	TRUE	FALSE	13.01.24 05:04	9.00 €	39.00	https://CRA	42.11%	29.6E-12
14	04.01.24	TRUE	FALSE	13.01.24 04:04	10.00 €	40.00	https://CRA	47.37%	433.9E-12
15	03.01.24	TRUE	FALSE	13.01.24 03:04	11.00 €	41.00	https://CRA	52.63%	256.6E-12
16	02.01.24	TRUE	FALSE	13.01.24 02:04	12.00 €	42.00	https://CRA	57.89%	498.0E-12

6.2.2 numfmts2

In addition, you can set the style to be picked up using `openxlsx2` options.

```
wb <- wb_workbook()
wb <- wb_add_worksheet(wb, "test")

options("openxlsx2.dateFormat" = "yyyy")
options("openxlsx2.datetimeFormat" = "yyyy-mm-dd")
options("openxlsx2.numFmt" = "€ #.0")

df <- data.frame(
  "Date" = Sys.Date() - 0:19,
  "T" = TRUE, "F" = FALSE,
  "Time" = Sys.time() - 0:19 * 60 * 60,
  "Cash" = 1:20, "Cash2" = 31:50,
  "hLink" = "https://CRAN.R-project.org/",
  "Percentage" = seq(0, 1, length.out = 20),
  "TinyNumbers" = runif(20) / 1E9, stringsAsFactors = FALSE,
  "numeric" = 1
)

## openxlsx will apply default Excel styling for these classes
class(df$Cash) <- c(class(df$Cash), "currency")
class(df$Cash2) <- c(class(df$Cash2), "accounting")
class(df$hLink) <- "hyperlink"
class(df$Percentage) <- c(class(df$Percentage), "percentage")
class(df$TinyNumbers) <- c(class(df$TinyNumbers), "scientific")

wb$add_data("test", df)
```

	A	B	C	D	E	F	G	H	I	J
1	Date	T	F	Time	Cash	Cash2	hLink	Percentage	TinyNumbe	numeric
2	2024	TRUE	FALSE	2024-01-13	1.00 €	31.00	https://CRA	0.00%	432.3E-12	€ 1.0
3	2024	TRUE	FALSE	2024-01-13	2.00 €	32.00	https://CRA	5.26%	348.6E-12	€ 1.0
4	2024	TRUE	FALSE	2024-01-13	3.00 €	33.00	https://CRA	10.53%	793.6E-12	€ 1.0
5	2024	TRUE	FALSE	2024-01-13	4.00 €	34.00	https://CRA	15.79%	886.9E-12	€ 1.0
6	2024	TRUE	FALSE	2024-01-13	5.00 €	35.00	https://CRA	21.05%	651.1E-12	€ 1.0
7	2024	TRUE	FALSE	2024-01-13	6.00 €	36.00	https://CRA	26.32%	592.9E-12	€ 1.0
8	2024	TRUE	FALSE	2024-01-13	7.00 €	37.00	https://CRA	31.58%	327.8E-12	€ 1.0
9	2024	TRUE	FALSE	2024-01-13	8.00 €	38.00	https://CRA	36.84%	75.7E-12	€ 1.0
10	2024	TRUE	FALSE	2024-01-13	9.00 €	39.00	https://CRA	42.11%	591.4E-12	€ 1.0
11	2024	TRUE	FALSE	2024-01-13	10.00 €	40.00	https://CRA	47.37%	178.3E-12	€ 1.0
12	2024	TRUE	FALSE	2024-01-13	11.00 €	41.00	https://CRA	52.63%	741.4E-12	€ 1.0

```
# disable styles via options
options("openxlsx2.dateFormat" = NULL)
```

```r
options("openxlsx2.datetimeFormat" = NULL)
options("openxlsx2.numFmt" = NULL)
```

6.3 Modifying the column and row widths

6.3.1 wb_set_col_widths

```r
wb <- wb_workbook() |>
  wb_add_worksheet() |>
  wb_add_data(x = mtcars, row_names = TRUE)

cols_1 <- 1:6
cols_2 <- "G:L"
wb <- wb |>
  wb_set_col_widths(cols = cols_1, widths = "auto") |>
  wb_set_col_widths(cols = cols_2, widths = 10)
```

6.3.2 wb_set_row_heigths

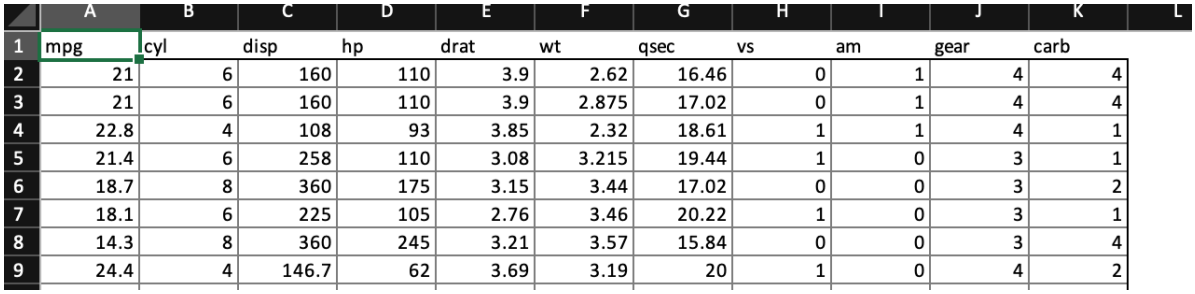
```r
wb <- wb |>
  wb_set_row_heights(rows = 1:10, heights = 10)
```

6.4 Adding borders

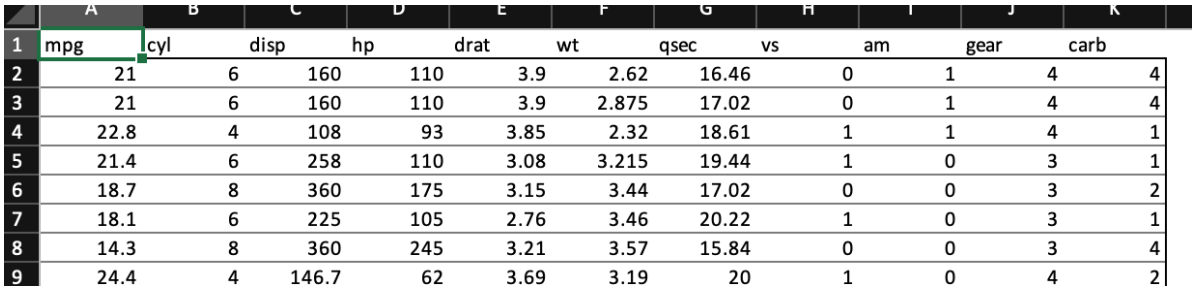
6.4.1 add borders

```r
wb <- wb_workbook()
# full inner grid
wb$add_worksheet("S1", grid_lines = FALSE)$add_data(x = mtcars)
wb$add_border(
  dims = "A2:K33",
  inner_hgrid = "thin", inner_hcolor = wb_color(hex = "FF808080"),
  inner_vgrid = "thin", inner_vcolor = wb_color(hex = "FF808080")
)
# only horizontal grid
```

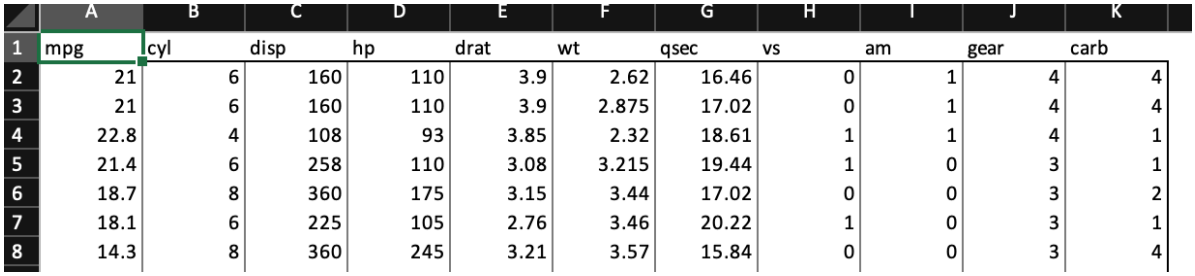
```
wb$add_worksheet("S2", grid_lines = FALSE)$add_data(x = mtcars)
wb$add_border(dims = wb_dims(x = mtcars, select = "data"), inner_hgrid = "thin",
              inner_hcolor = wb_color(hex = "FF808080"))
# only vertical grid
wb$add_worksheet("S3", grid_lines = FALSE)$add_data(x = mtcars)
wb$add_border(dims =  wb_dims(x = mtcars, select = "data"),
              inner_vgrid = "thin", inner_vcolor = wb_color(hex = "FF808080"))
# no inner grid
wb$add_worksheet("S4", grid_lines = FALSE)$add_data(x = mtcars)
wb$add_border("S4", dims = wb_dims(x = mtcars, select = "data"))
```



	A	B	C	D	E	F	G	H	I	J	K	L
1	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	
2	21	6	160	110	3.9	2.62	16.46	0	1	4	4	
3	21	6	160	110	3.9	2.875	17.02	0	1	4	4	
4	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1	
5	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	
6	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2	
7	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1	
8	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4	
9	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2	



	A	B	C	D	E	F	G	H	I	J	K
1	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
2	21	6	160	110	3.9	2.62	16.46	0	1	4	4
3	21	6	160	110	3.9	2.875	17.02	0	1	4	4
4	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
5	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
6	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
7	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
8	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
9	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2



	A	B	C	D	E	F	G	H	I	J	K
1	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
2	21	6	160	110	3.9	2.62	16.46	0	1	4	4
3	21	6	160	110	3.9	2.875	17.02	0	1	4	4
4	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
5	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
6	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
7	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
8	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4

	A	B	C	D	E	F	G	H	I	J	K
1	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
2	21	6	160	110	3.9	2.62	16.46	0	1	4	4
3	21	6	160	110	3.9	2.875	17.02	0	1	4	4
4	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
5	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
6	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
7	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
8	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4

6.4.2 styled table

Below we show you two ways how to create styled tables with `openxlsx2` one using the high level functions to style worksheet areas and one using the bare metal approach of creating the identical table.

X1	X2
1	3
2	4

6.4.2.1 the quick way: using high level functions

```
# add some dummy data to the worksheet
mat <- matrix(1:4, ncol = 2, nrow = 2)
colnames(mat) <- make.names(seq_len(ncol(mat)))

dims_head <- wb_dims(x = mat, from_col = 2, from_row = 2, select = "col_names")
dims_data <- wb_dims(x = mat, from_col = 2, from_row = 2, select = "data")

wb <- wb_workbook() |>
  wb_add_worksheet("test") |>
  wb_add_data(x = mat, col_names = TRUE, start_col = 2, start_row = 2) |>
  # center first row
  wb_add_cell_style(dims = dims_head, horizontal = "center") |>
  # add border for first row
  wb_add_border(
    dims = dims_head,
    bottom_color = wb_color(theme = 1), bottom_border = "thin",
    top_color = wb_color(theme = 1), top_border = "double",
    left_border = NULL, right_border = NULL
```

```
  ) |>
  # add border for last row
  wb_add_border(
    dims = dims_data,
    bottom_color = wb_color(theme = 1), bottom_border = "double",
    top_border = NULL, left_border = NULL, right_border = NULL
  )
```

6.4.2.2 the long way: creating everything from the bone

```
# add some dummy data to the worksheet
mat <- matrix(1:4, ncol = 2, nrow = 2)
colnames(mat) <- make.names(seq_len(ncol(mat)))

wb <- wb_workbook() |>
  wb_add_worksheet("test") |>
  wb_add_data(x = mat, start_col = 2, start_row = 2)

# create a border style and assign it to the workbook
black <- wb_color(hex = "FF000000")
top_border <- create_border(
  top = "double", top_color = black,
  bottom = "thin", bottom_color = black
)

bottom_border <- create_border(bottom = "double", bottom_color = black)

wb$styles_mgr$add(top_border, "top_border")
wb$styles_mgr$add(bottom_border, "bottom_border")

# create a new cell style, that uses the fill, the font and the border style
top_cellxfs <- create_cell_style(
  numFmtId = 0,
  horizontal = "center",
  borderId = wb$styles_mgr$get_border_id("top_border")
)
bottom_cellxfs <- create_cell_style(
  numFmtId = 0,
  borderId = wb$styles_mgr$get_border_id("bottom_border")
)
```

```
# assign this style to the workbook
wb$styles_mgr$add(top_cellxfs, "top_styles")
wb$styles_mgr$add(bottom_cellxfs, "bottom_styles")

# assign the new cell style to the header row of our data set
cell <- "B2:C2"
wb <- wb |> wb_set_cell_style(dims = cell,
                              style = wb$styles_mgr$get_xf_id("top_styles"))
cell <- "B4:C4"
wb <- wb |> wb_set_cell_style(dims = cell,
                              style = wb$styles_mgr$get_xf_id("bottom_styles"))
```

6.5 Use workbook colors and modify them

The loop below will apply the tint attribute to the fill color

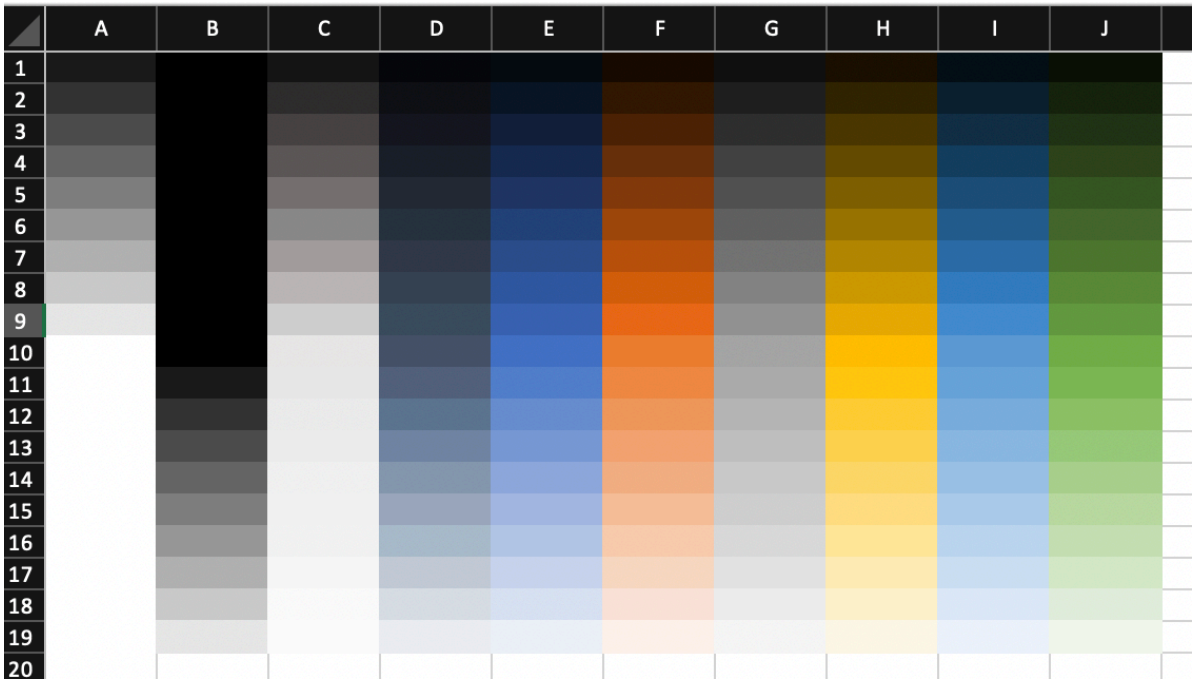


Figure 6.2: Tint variations of the theme colors.

```
wb <- wb_workbook() |> wb_add_worksheet("S1")
```

```
tints <- seq(-0.9, 0.9, by = 0.1)

for (i in 0:9) {
  for (tnt in tints) {
    col <- paste0(int2col(i + 1), which(tints %in% tnt))

    if (tnt == 0) {
      wb <- wb |> wb_add_fill(dims = col,
                              color = wb_color(theme = i))
    } else {
      wb <- wb |> wb_add_fill(dims = col,
                              color = wb_color(theme = i, tint = tnt))
    }
  }
}
```

6.6 Copy cell styles

It is possible to copy the styles of several cells at once. In the following example, the styles of some cells from a formatted workbook are applied to a previously empty cell range. Be careful though, `wb_get_cell_style()` returns only some styles, so you have to make sure that the copy-from and copy-to dimensions match in a meaningful way.

```
xl <- system.file("extdata", "oxlsx2_sheet.xlsx", package = "openxlsx2")
wb <- wb_load(xl)
wb$set_cell_style(1, "A30:G35", wb$get_cell_style(1, "A10:G15"))
# wb_open(wb)
```

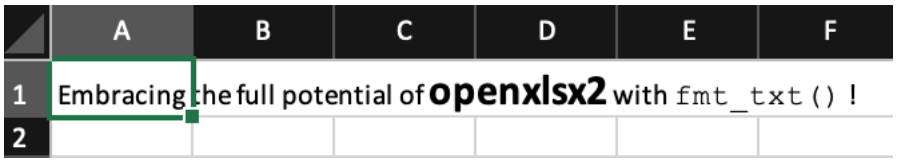
	A	B	C	D	E	F	H	I
1								
2					***Header***			
3		**Date**	**Value1**		**Value2**		**Value3**	
4			**€**	**%**	**€**	**%**	**€**	**%**
6		Jan-21	1,000		431		29	
7		Feb-21	264	26 %	777	180.28 %	28	96.55 %
8		Mar-21	4	1 %	4567	587.77 %	27	96.43 %
9		Apr-21	4,393	120492 %	464	10.16 %	26	96.30 %
10		May-21	53	1 %	433	93.32 %	25	96.15 %
11		Jun-21	63	119 %	356	82.22 %	24	96.00 %
12		Jul-21	838	1324 %	354	99.44 %	23	95.83 %
13		Aug-21	23,131	2760 %	3355	947.74 %	22	95.65 %
14		Sep-21	2,323	10 %	334	9.96 %	21	95.45 %
15		Oct-21	3,323	143 %	541	161.98 %	20	95.24 %
16		Nov-21	35	1 %	555	102.59 %	20	100.00 %
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32								
33								
34								
35								
36								
37								

6.7 Style strings

Using `fmt_txt()` is possible to style strings independently of the cell containing the string.

```
txt <-
  fmt_txt("Embracing the full potential of ") +
  fmt_txt("openxlsx2", bold = TRUE, size = 16) +
```

```
  fmt_txt(" with ") +
  fmt_txt("fmt_txt()", font = "Courier") +
  fmt_txt(" !")

wb <- wb_workbook()$add_worksheet()$add_data(x = txt, col_names = FALSE)
```



As shown above it is possible to combine multiple styles together into a longer string. It is even possible to use `fmt_txt()` as `na.strings`:

```
df <- mtcars
df[df < 4] <- NA

na_red <- fmt_txt("N/A", color = wb_color("red"), italic = TRUE, bold = TRUE)

wb <- wb_workbook()$add_worksheet()$add_data(x = df, na.strings = na_red)
```

D	E	F	G	H	I	J	K	
p	drat	wt	qsec	vs	am	gear	carb	
110	*N/A*	*N/A*	16.46	*N/A*	*N/A*	4	4	
110	*N/A*	*N/A*	17.02	*N/A*	*N/A*	4	4	
93	*N/A*	*N/A*	18.61	*N/A*	*N/A*	4	*N/A*	
110	*N/A*	*N/A*	19.44	*N/A*	*N/A*	*N/A*	*N/A*	
175	*N/A*	*N/A*	17.02	*N/A*	*N/A*	*N/A*	*N/A*	
105	*N/A*	*N/A*	20.22	*N/A*	*N/A*	*N/A*	*N/A*	
245	*N/A*	*N/A*	15.84	*N/A*	*N/A*	*N/A*	4	
62	*N/A*	*N/A*	20	*N/A*	*N/A*	4	*N/A*	
95	*N/A*	*N/A*	22.9	*N/A*	*N/A*	4	*N/A*	
123	*N/A*	*N/A*	18.3	*N/A*	*N/A*	4	4	
123	*N/A*	*N/A*	18.9	*N/A*	*N/A*	4	4	
180	*N/A*	4.07	17.4	*N/A*	*N/A*	*N/A*	*N/A*	
180	*N/A*	*N/A*	17.6	*N/A*	*N/A*	*N/A*	*N/A*	

6.8 Create custom table styles

With `create_tablestyle()` it is possible to create your own table styles. This function uses `create_dxfs_style()` (just like your spreadsheet software does). Therefore, it is not quite as user-friendly. The following example shows how the function creates a red table style. The various dxfs styles must be created and assigned to the workbook (similar styles are used in conditional formatting). In `create_tablestyle()` these styles are assigned to the table style elements. Once the table style is created, it must also be assigned to the workbook. After that you can use it in the workbook like any other table style.

```
# a red table style
dx0 <- create_dxfs_style(
  border = TRUE,
  left_color = wb_color("red"),
  right_color = NULL, right_style = NULL,
  top_color = NULL, top_style = NULL,
  bottom_color = NULL, bottom_style = NULL
)

dx1 <- create_dxfs_style(
  border = TRUE,
  left_color = wb_color("red"),
  right_color = NULL, right_style = NULL,
  top_color = NULL, top_style = NULL,
  bottom_color = NULL, bottom_style = NULL
)

dx2 <- create_dxfs_style(
  border = TRUE,
  top_color = wb_color("red"),
  left_color = NULL, left_style = NULL,
  right_color = NULL, right_style = NULL,
  bottom_color = NULL, bottom_style = NULL
)

dx3 <- create_dxfs_style(
  border = TRUE,
  top_color = wb_color("red"),
  left_color = NULL, left_style = NULL,
  right_color = NULL, right_style = NULL,
  bottom_color = NULL, bottom_style = NULL
)
```

```r
dx4 <- create_dxfs_style(
  text_bold = TRUE
)

dx5 <- create_dxfs_style(
  text_bold = TRUE
)

dx6 <- create_dxfs_style(
  font_color = wb_color("red"),
  text_bold = TRUE,
  border = TRUE,
  top_style = "double",
  left_color = NULL, left_style = NULL,
  right_color = NULL, right_style = NULL,
  bottom_color = NULL, bottom_style = NULL
)

dx7 <- create_dxfs_style(
  font_color = wb_color("white"),
  text_bold = TRUE,
  bgFill = wb_color("red"),
  fgColor = wb_color("red")
)

dx8 <- create_dxfs_style(
  border = TRUE,
  left_color = wb_color("red"),
  top_color = wb_color("red"),
  right_color = wb_color("red"),
  bottom_color = wb_color("red")
)


wb <- wb_workbook() |>
  wb_add_worksheet(grid_lines = FALSE)

wb$add_style(dx0)
wb$add_style(dx1)
wb$add_style(dx2)
wb$add_style(dx3)
wb$add_style(dx4)
```

```
wb$add_style(dx5)
wb$add_style(dx6)
wb$add_style(dx7)
wb$add_style(dx8)

# finally create the table
xml <- create_tablestyle(
  name                 = "red_table",
  whole_table          = wb$styles_mgr$get_dxf_id("dx8"),
  header_row           = wb$styles_mgr$get_dxf_id("dx7"),
  total_row            = wb$styles_mgr$get_dxf_id("dx6"),
  first_column         = wb$styles_mgr$get_dxf_id("dx5"),
  last_column          = wb$styles_mgr$get_dxf_id("dx4"),
  first_row_stripe     = wb$styles_mgr$get_dxf_id("dx3"),
  second_row_stripe    = wb$styles_mgr$get_dxf_id("dx2"),
  first_column_stripe  = wb$styles_mgr$get_dxf_id("dx1"),
  second_column_stripe = wb$styles_mgr$get_dxf_id("dx0")
)


wb$add_style(xml)

# create a table and apply the custom style
wb <- wb |>
  wb_add_data_table(x = mtcars, table_style = "red_table")
```

	A	B	C	D	
1	mpg	cyl	disp	hp	drat
2	21	6	160	110	
3	21	6	160	110	
4	22.8	4	108	93	
5	21.4	6	258	110	
6	18.7	8	360	175	
7	18.1	6	225	105	

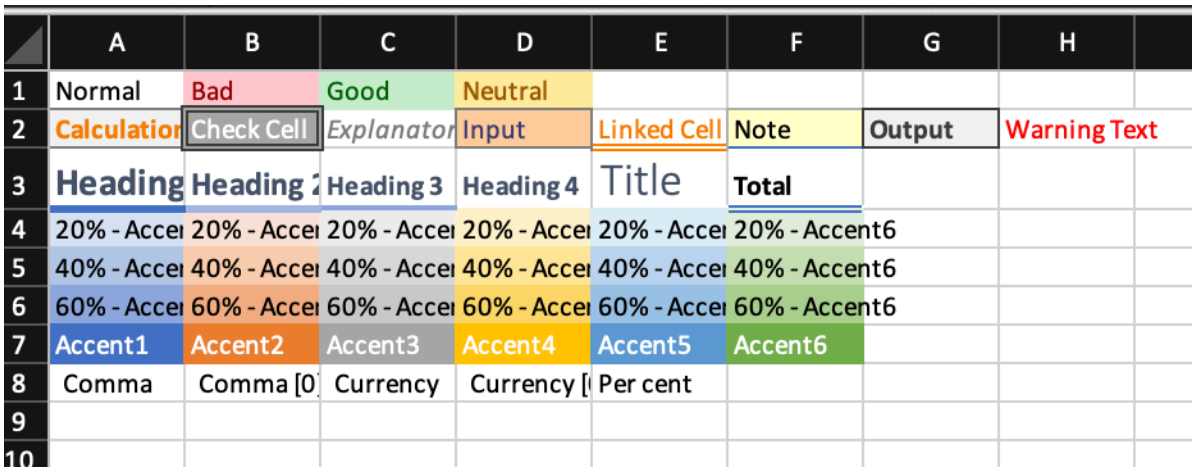
6.9 Named styles

```
wb <- wb_workbook()$add_worksheet()

name <- "Normal"
dims <- "A1"
wb$add_data(dims = dims, x = name)

name <- "Bad"
dims <- "B1"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Good"
dims <- "C1"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Neutral"
dims <- "D1"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Calculation"
dims <- "A2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Check Cell"
dims <- "B2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Explanatory Text"
dims <- "C2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Input"
dims <- "D2"
wb$add_named_style(dims = dims, name = name)
```

```
wb$add_data(dims = dims, x = name)

name <- "Linked Cell"
dims <- "E2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Note"
dims <- "F2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Output"
dims <- "G2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Warning Text"
dims <- "H2"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Heading 1"
dims <- "A3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Heading 2"
dims <- "B3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Heading 3"
dims <- "C3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Heading 4"
dims <- "D3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)
```

```
name <- "Title"
dims <- "E3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Total"
dims <- "F3"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

for (i in seq_len(6)) {

  name <- paste0("20% - Accent", i)
  dims <- paste0(int2col(i), "4")
  wb$add_named_style(dims = dims, name = name)
  wb$add_data(dims = dims, x = name)

  name <- paste0("40% - Accent", i)
  dims <- paste0(int2col(i), "5")
  wb$add_named_style(dims = dims, name = name)
  wb$add_data(dims = dims, x = name)

  name <- paste0("60% - Accent", i)
  dims <- paste0(int2col(i), "6")
  wb$add_named_style(dims = dims, name = name)
  wb$add_data(dims = dims, x = name)

  name <- paste0("Accent", i)
  dims <- paste0(int2col(i), "7")
  wb$add_named_style(dims = dims, name = name)
  wb$add_data(dims = dims, x = name)

}

name <- "Comma"
dims <- "A8"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Comma [0]"
dims <- "B8"
wb$add_named_style(dims = dims, name = name)
```

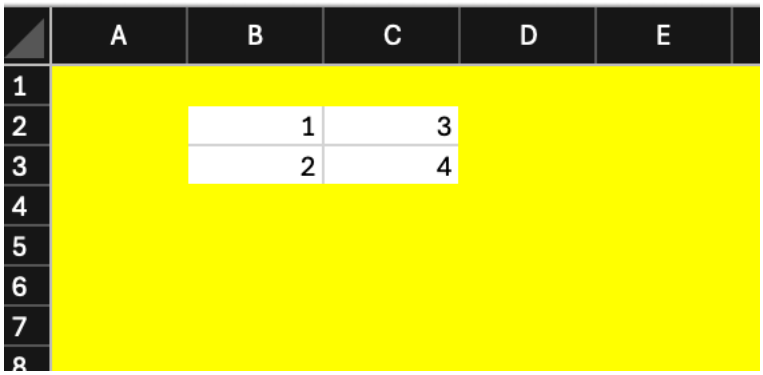
```
wb$add_data(dims = dims, x = name)

name <- "Currency"
dims <- "C8"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Currency [0]"
dims <- "D8"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

name <- "Per cent"
dims <- "E8"
wb$add_named_style(dims = dims, name = name)
wb$add_data(dims = dims, x = name)

# wb$open()
```



6.10 Styled columns / rows

In addition to individually styled cells, spreadsheets can also have styled columns and rows. Though, these column and row styles are overridden by cell styles. Therefore, if a row is filled with yellow, initialized cells in this row are not impacted by the yellow color. Therefore it is required to individually style these cells too.

```
wb <- wb_workbook()

# make the entire thing yellow (pick the style from an exisisting cell)
wb$add_worksheet("cols")$add_fill(color = wb_color("yellow"))
wb$set_cell_style_across(cols = "A:XFD", style = wb$get_cell_style(dims = "A1"))

# create an orange cell style
f1 <- create_fill(patternType = "solid", fgColor = wb_color("orange"))
wb$add_style(f1, "f1")

s1 <- create_cell_style(fill_id = wb$styles_mgr$get_fill_id("f1"))
wb$add_style(s1, "s1")

# fill all rows with this orange cell style
wb$add_worksheet("rows")
wb$set_cell_style_across(rows = seq_len(1048576),
                         style = wb$styles_mgr$get_xf_id("s1"))

# show what happens
mm   <- matrix(1:4, 2, 2)
dims <- wb_dims(x = mm, from_dims = "B2", x = mm, col_names = FALSE)

wb$add_data(sheet = "cols", dims = dims, x = mm, col_names = FALSE)
wb$add_data(sheet = "rows", dims = dims, x = mm, col_names = FALSE)
```



Styling columns and rows is quicker and memory efficient. In the first example all columns are modified, but the entire modification can be boiled down to a single XML string.

A neat example of this is the following: a user wanted to lock certain cells on a worksheet. To achieve this, we have to apply an unlocked style to the entire worksheet. Afterwards we can select a few cells that we want to lock.

```
wb <- wb_workbook()$add_worksheet()

# create an unlocked cell style
s1 <- create_cell_style(locked = FALSE)
wb$add_style(s1, "s1")

# apply this to the entire worksheet
wb$set_cell_style_across(cols = "A:XFD", style = "s1")

# locked a few cells
dims_list <- c(
  wb_dims(1:3, 3:10),
  wb_dims(5:10, 8:12),
  wb_dims(100:105, 300:408)
)

# lock a few ranges and highlight these in red
for (dims in dims_list) {
  message("locking: ", dims)
  wb$add_fill(dims = dims, color = wb_color("red"))
  wb$add_cell_style(dims = dims, locked = TRUE)
}
#> locking: C1:J3
#> locking: H5:L10
#> locking: KN100:OR105

# protect the worksheet and the workbook
wb$protect_worksheet()
wb$protect()
```

6.11 Styling with `dims`

It is possible to style multiple cells at once using **`dims`**. This is way faster than looping over rows, columns or both as would be required in the example below.

```
set.seed(123)
mm <- matrix(sample(0:1, 2500, TRUE), 50, 50)

zeros <- as.data.frame(which(mm == 0, arr.ind = TRUE))
ones  <- as.data.frame(which(mm == 1, arr.ind = TRUE))
```

```
dims_z <- paste0(mapply(rowcol_to_dims, zeros$row, zeros$col), collapse = ",")
dims_o <- paste0(mapply(rowcol_to_dims, ones$row, ones$col), collapse = ",")

wb <- wb_workbook()$add_worksheet()$
  add_data(x = mm, col_names = FALSE)$
  add_fill(dims = dims_z, color = wb_color("lightgray"))$
  add_fill(dims = dims_o, color = wb_color("darkgray"))
```

	A	B	C	D	E	F	G
1	0	1	0	0	1	0	
2	0	0	1	1	1	1	
3	0	0	1	0	1	1	
4	1	0	0	1	0	0	
5	0	0	1	0	1	0	
6	1	1	1	0	1	0	
7	1	1	0	0	1	0	
8	1	0	0	0	0	0	
9	0	1	0	0	1	1	
10	0	0	1	0	1	0	
11	1	0	1	0	1	1	
12	1	1	1	1	0	1	
13	1	1	1	1	0	1	
14	0	0	0	1	1	0	

7 Conditional Formatting, Databars, and Sparklines

With `openxlsx2` it is possible to add conditional formatting, databars, and sparklines to the spreadsheet as a dynamic layer to data visualization, enhancing the interpretation and analysis of information. They help with presenting complex data sets, providing a visual representation that goes beyond raw numbers. It is possible to modify each with various style options.

1. Conditional Formatting: Conditional formatting enables users to apply a cell style overlay based on predefined rules. This can highlight patterns, trends, and anomalies within the data. The rules make it possible to provide a visual highlighting without having to style every cell individually.

2. Databars: Databars are a specific type of conditional formatting that adds horizontal bars within cells to represent the values they contain. Similar to a `barplot` just spreading as an overlay across multiple rows. The length of the bar corresponds to the magnitude of the data, allowing for a quick and intuitive comparison between different values.

3. Sparklines: Sparklines are compact, miniature charts embedded within a single cell, offering a condensed visual representation of trends or variations in a dataset. These tiny graphs, such as line charts, bar charts, or win/loss charts, provide a quick overview of the data's trajectory without the need for a separate chart. Sparklines are especially valuable when it is required to maintain a compact layout while still conveying the overall patterns in the data.

7.1 Conditional Formatting

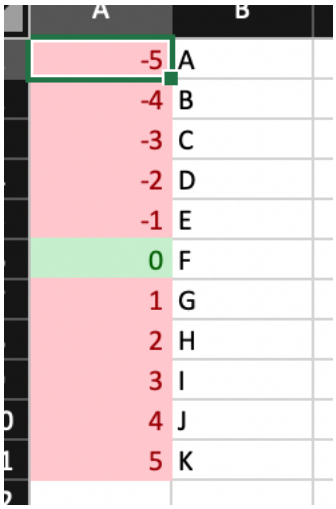
Conditional formatting is helpful to visually emphasize trends, outliers, or other important aspects of the data it is applied to. In `openxlsx2` conditional formatting is applied as follows:

1. Select dimension range: First select, the range of cells to which the conditional formatting is applied to.
2. Define a rule: Define a rule or condition that will be used for the formatting.
3. Define a style: (optional) the style used by conditional formatting of various cells can differ. This styles can include for example changes to the font, background, borders.

We will use the following workbook and the two styles to differentiate in negative and positive values.

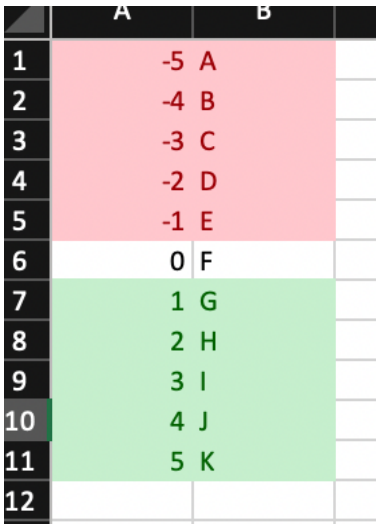
```
wb <- wb_workbook()
wb$add_dxfs_style(name = "negStyle", font_color = wb_color(hex = "FF9C0006"),
                  bg_fill = wb_color(hex = "FFFFC7CE"))
wb$add_dxfs_style(name = "posStyle", font_color = wb_color(hex = "FF006100"),
                  bg_fill = wb_color(hex = "FFC6EFCE"))
```

7.1.1 Rule applies to all each cell in range




```
wb$add_worksheet("cellIs")
wb$add_data("cellIs", -5:5)
wb$add_data("cellIs", LETTERS[1:11], start_col = 2)
wb$add_conditional_formatting(
  "cellIs",
  dims = "A1:A11",
  rule = "!=0",
  style = "negStyle"
)
wb$add_conditional_formatting(
  "cellIs",
  dims = "A1:A11",
  rule = "==0",
  style = "posStyle"
)
```

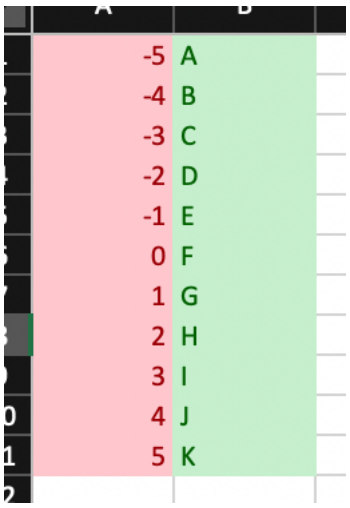
7.1.2 Highlight row dependent on first cell in row



	A	B
1	-5	A
2	-4	B
3	-3	C
4	-2	D
5	-1	E
6	0	F
7	1	G
8	2	H
9	3	I
10	4	J
11	5	K
12		

```
wb$add_worksheet("Moving Row")
wb$add_data("Moving Row", -5:5)
wb$add_data("Moving Row", LETTERS[1:11], start_col = 2)
wb$add_conditional_formatting(
  "Moving Row",
  dims = "A1:B11",
  rule = "$A1<0",
  style = "negStyle"
)
wb$add_conditional_formatting(
  "Moving Row",
  dims = "A1:B11",
  rule = "$A1>0",
  style = "posStyle"
)
```

7.1.3 Highlight column dependent on first cell in column



	A	B
1	-5	A
2	-4	B
3	-3	C
4	-2	D
5	-1	E
6	0	F
7	1	G
8	2	H
9	3	I
10	4	J
11	5	K
12		

```
wb$add_worksheet("Moving Col")
wb$add_data("Moving Col", -5:5)
wb$add_data("Moving Col", LETTERS[1:11], start_col = 2)
wb$add_conditional_formatting(
  "Moving Col",
  dims = "A1:B11",
  rule = "A$1<0",
  style = "negStyle"
)
wb$add_conditional_formatting(
  "Moving Col",
  dims = "A1:B11",
  rule = "A$1>0",
  style = "posStyle"
)
```

7.1.4 Highlight entire range cols X rows dependent only on cell A1

1	-5	A
2	-4	B
3	-3	C
4	-2	D
5	-1	E
6	0	F
7	1	G
8	2	H
9	3	I
10	4	J
11	5	K
12		
13		
14		
15	x	y
16	1	0,287578
17	2	0,788305
18	3	0,408977
19	4	0,883017
20	5	0,940467
21	6	0,045556
22	7	0,528105
23	8	0,892419
24	9	0,551435
25	10	0,456615
26		

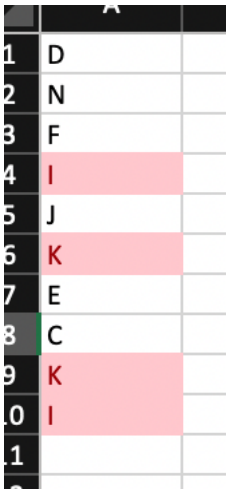
```
wb$add_worksheet("Dependent on")
wb$add_data("Dependent on", -5:5)
wb$add_data("Dependent on", LETTERS[1:11], start_col = 2)
wb$add_conditional_formatting(
  "Dependent on",
  dims = "A1:B11",
  rule = "$A$1 < 0",
  style = "negStyle"
)
wb$add_conditional_formatting(
```

```
  "Dependent on",
  dims = "A1:B11",
  rule = "$A$1>0",
  style = "posStyle"
)
```

7.1.5 Highlight cells in column 1 based on value in column 2

```
wb$add_data("Dependent on", data.frame(x = 1:10, y = runif(10)), startRow = 15)
wb$add_conditional_formatting(
  "Dependent on",
  dims = "A16:A25",
  rule = "B16<0.5",
  style = "negStyle"
)
wb$add_conditional_formatting(
  "Dependent on",
  dims = "A16:A25",
  rule = "B16>=0.5",
  style = "posStyle"
)
```

7.1.6 Highlight duplicates using default style



```
wb$add_worksheet("Duplicates")
wb$add_data("Duplicates", sample(LETTERS[1:15], size = 10, replace = TRUE))
wb$add_conditional_formatting(
  "Duplicates",
  dims = "A1:A10",
  type = "duplicatedValues"
)
```

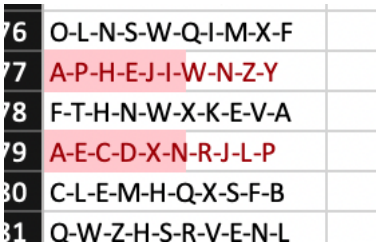
7.1.7 Cells containing text


```
fn <- function(x) paste(sample(LETTERS, 10), collapse = "-")
wb$add_worksheet("containsText")
wb$add_data("containsText", sapply(1:10, fn))
wb$add_conditional_formatting(
  "containsText",
  dim = "A1:A10",
  type = "containsText",
  rule = "A"
)
wb$add_worksheet("notcontainsText")
```

7.1.8 Cells not containing text


```
fn <- function(x) paste(sample(LETTERS, 10), collapse = "-")
wb$add_data("notcontainsText", x = sapply(1:10, fn))
wb$add_conditional_formatting(
  "notcontainsText",
  dim = "A1:A10",
  type = "notContainsText",
  rule = "A"
)
```

7.1.9 Cells begins with text




```
fn <- function(x) paste(sample(LETTERS, 10), collapse = "-")
wb$add_worksheet("beginsWith")
wb$add_data("beginsWith", x = sapply(1:100, fn))
wb$add_conditional_formatting(
  "beginsWith",
  dim = "A1:A100",
```

```
  type = "beginsWith",
  rule = "A"
)
```

7.1.10 Cells ends with text


```
fn <- function(x) paste(sample(LETTERS, 10), collapse = "-")
wb$add_worksheet("endsWith")
wb$add_data("endsWith", x = sapply(1:100, fn))
wb$add_conditional_formatting(
  "endsWith",
  dim = "A1:A100",
  type = "endsWith",
  rule = "A"
)
```

7.1.11 Colorscale colors cells based on cell value

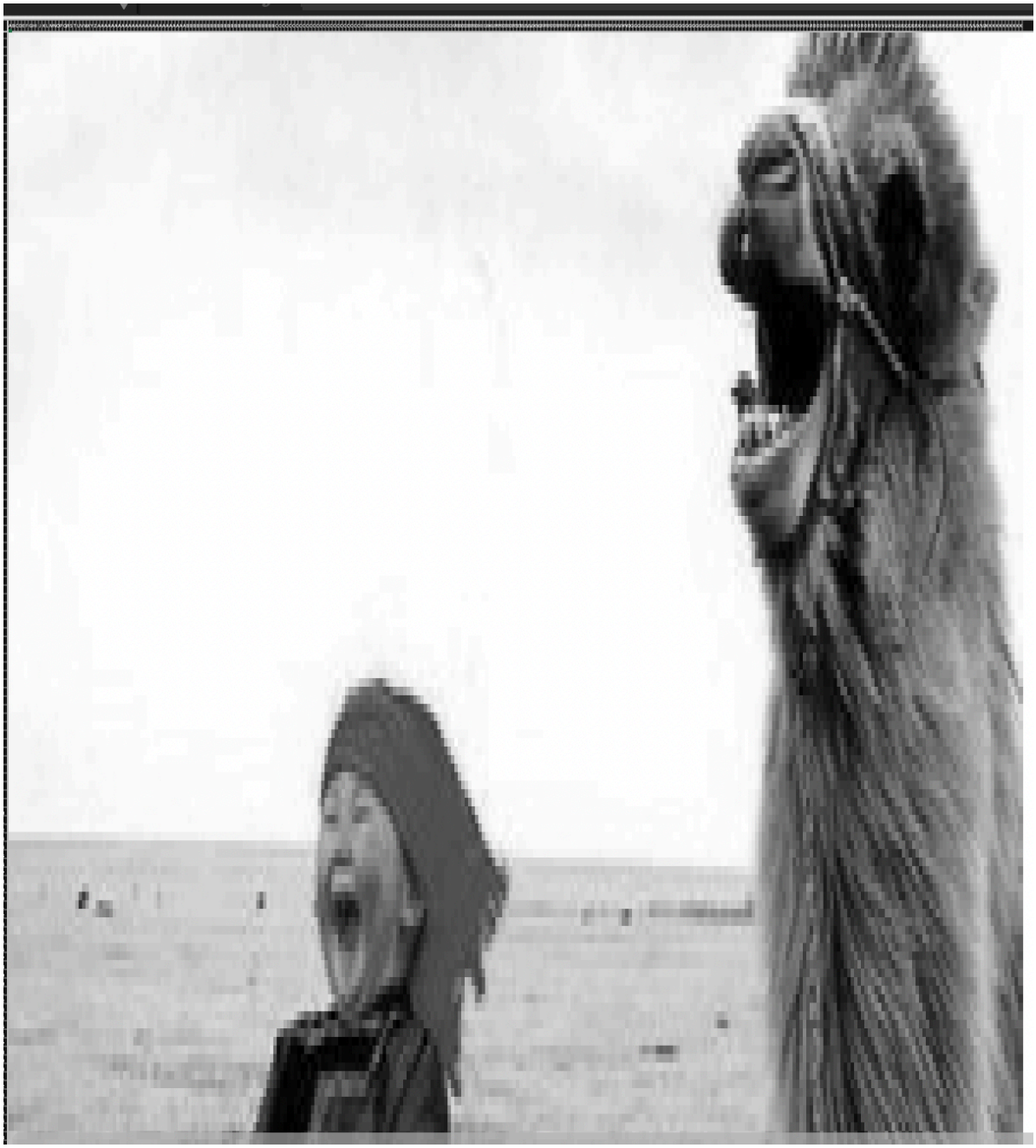


Figure 7.1: Yep, that is a color scale image.

```
fl <- "https://github.com/JanMarvin/openxlsx-data/raw/main/readTest.xlsx"
df <- read_xlsx(fl, sheet = 5)
wb$add_worksheet("colorScale", zoom = 30)
wb$add_data("colorScale", x = df, col_names = FALSE) ### write data.frame
```

Rule is a vector or colors of length 2 or 3 (any hex color or any of `colors()`). If rule is `NULL`, min and max of cells is used. Rule must be the same length as style or L.

```
wb$add_conditional_formatting(
  "colorScale",
  dims = wb_dims(x = df, col_names = FALSE),
  style = c("black", "white"),
  rule = c(0, 255),
  type = "colorScale"
)
wb$set_col_widths("colorScale", cols = seq_along(df), widths = 1.07)
wb$set_row_heights("colorScale", rows = seq_len(nrow(df)), heights = 7.5)
```

7.1.12 Between

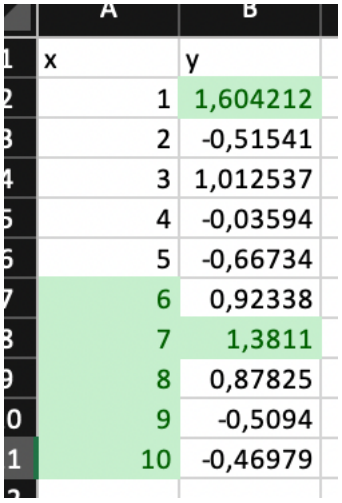
	A
1	-5
2	-4
3	-3
4	-2
5	-1
6	0
7	1
8	2
9	3
10	4
11	5
12	

Highlight cells in interval [-2, 2]

```
wb$add_worksheet("between")
wb$add_data("between", -5:5)
wb$add_conditional_formatting(
  "between",
```

```
  dims = "A1:A11",
  type = "between",
  rule = c(-2, 2)
)
wb$add_worksheet("topN")
```

7.1.13 Top N



	A	B
1	x	y
2	1	1,604212
3	2	-0,51541
4	3	1,012537
5	4	-0,03594
6	5	-0,66734
7	6	0,92338
8	7	1,3811
9	8	0,87825
10	9	-0,5094
11	10	-0,46979

```
wb$add_data("topN", data.frame(x = 1:10, y = rnorm(10)))
```

Highlight top 5 values in column `x`

```
wb$add_conditional_formatting(
  "topN",
  dims = "A2:A11",
  style = "posStyle",
  type = "topN",
  params = list(rank = 5)
)
```

Highlight top 20 percentage in column `y`

```
wb$add_conditional_formatting(
  "topN",
  dims = "B2:B11",
```

```
  style = "posStyle",
  type = "topN",
  params = list(rank = 20, percent = TRUE)
)
wb$add_worksheet("bottomN")
```

7.1.14 Bottom N

	x	y
2	1	1,377676
3	2	0,352826
4	3	0,829574
5	4	-0,3387
6	5	1,261035
7	6	-0,80876
8	7	0,625352
9	8	-0,81717
10	9	-2,46258
11	10	-1,34296
12		

```
wb$add_data("bottomN", data.frame(x = 1:10, y = rnorm(10)))
```

Highlight bottom 5 values in column `x`

```
wb$add_conditional_formatting(
  "bottomN",
  dims = "A2:A11",
  style = "negStyle",
  type = "bottomN",
  params = list(rank = 5)
)
```

Highlight bottom 20 percentage in column `y`

```
wb$add_conditional_formatting(
  "bottomN",
  dims = "B2:B11",
  style = "negStyle",
```

```
  type = "bottomN",
  params = list(rank = 20, percent = TRUE)
)
wb$add_worksheet("logical operators")
```

7.1.15 Logical Operators

	A
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	

You can use Excels logical Operators

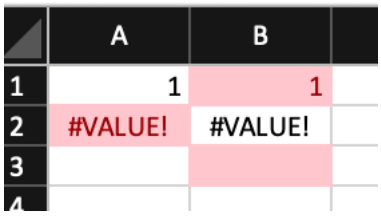
```
wb$add_data("logical operators", 1:10)
wb$add_conditional_formatting(
  "logical operators",
  dims = "A1:A10",
  rule = "OR($A1=1,$A1=3,$A1=5,$A1=7)"
)
```

7.1.16 (Not) Contains Blanks

	A	B
1		
2	1	1
3	2	2
4		
5		
6		

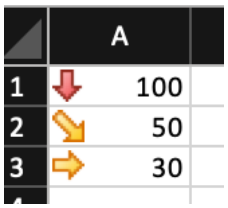
```
wb$add_worksheet("contains blanks")
wb$add_data(x = c(NA, 1, 2, ''), col_names = FALSE, na.strings = NULL)
wb$add_data(x = c(NA, 1, 2, ''), col_names = FALSE, na.strings = NULL,
            start_col = 2)
wb$add_conditional_formatting(dims = "A1:A4", type = "containsBlanks")
wb$add_conditional_formatting(dims = "B1:B4", type = "notContainsBlanks")
```

7.1.17 (Not) Contains Errors




```
wb$add_worksheet("contains errors")
wb$add_data(x = c(1, NaN), colNames = FALSE)
wb$add_data(x = c(1, NaN), colNames = FALSE, start_col = 2)
wb$add_conditional_formatting(dims = "A1:A3", type = "containsErrors")
wb$add_conditional_formatting(dims = "A1:A3", type = "notContainsErrors")
```

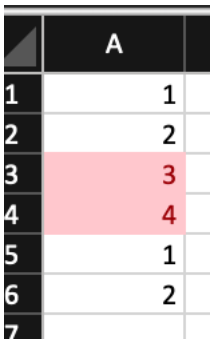
7.1.18 Iconset




```
wb$add_worksheet("iconset")
wb$add_data(x = c(100, 50, 30), colNames = FALSE)
wb$add_conditional_formatting(
  dims = "A1:A6",
  rule = c(-67, -33, 0, 33, 67),
  type = "iconSet",
  params = list(
    percent = FALSE,
    iconSet = "5Arrows",
```

```
    reverse = TRUE)
  )
```

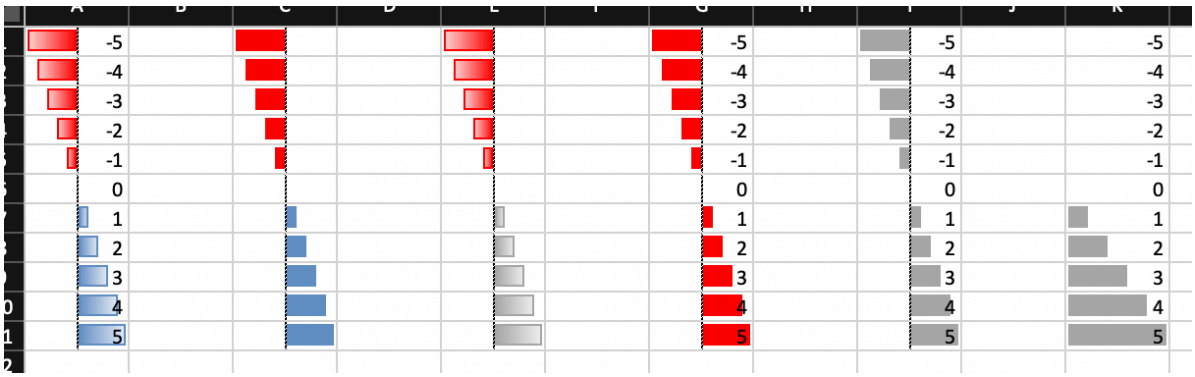
7.1.19 Unique Values



	A
1	1
2	2
3	3
4	4
5	1
6	2
7	

```
wb$add_worksheet("unique values")
wb$add_data(x = c(1:4, 1:2), colNames = FALSE)
wb$add_conditional_formatting(dims = "A1:A6", type = "uniqueValues")
```

7.2 Databars



	A	K
1	-5	-5
2	-4	-4
3	-3	-3
4	-2	-2
5	-1	-1
6	0	0
7	1	1
8	2	2
9	3	3
10	4	4
11	5	5
12		

```
wb$add_worksheet("databar")
### Databars
wb$add_data("databar", -5:5, start_col = 1)
wb <- wb_add_conditional_formatting(
  wb,
  "databar",
  dims = "A1:A11",
```

```
  type = "dataBar"
) ### Default colors

wb$add_data("databar", -5:5, start_col = 3)
wb <- wb_add_conditional_formatting(
  wb,
  "databar",
  dims = "C1:C11",
  type = "dataBar",
  params = list(
    showValue = FALSE,
    gradient = FALSE
  )
) ### Default colors

wb$add_data("databar", -5:5, start_col = 5)
wb <- wb_add_conditional_formatting(
  wb,
  "databar",
  dims = "E1:E11",
  type = "dataBar",
  style = c("#a6a6a6"),
  params = list(showValue = FALSE)
)

wb$add_data("databar", -5:5, start_col = 7)
wb <- wb_add_conditional_formatting(
  wb,
  "databar",
  dims = "G1:G11",
  type = "dataBar",
  style = c("red"),
  params = list(
    showValue = TRUE,
    gradient = FALSE
  )
)

# custom color
wb$add_data("databar", -5:5, start_col = 9)
wb <- wb_add_conditional_formatting(
  wb,
```

```
  "databar",
  dims = wb_dims(cols = 9, rows = 1:11),
  type = "dataBar",
  style = c("#a6a6a6", "#a6a6a6"),
  params = list(showValue = TRUE, gradient = FALSE)
)

# with rule
wb$add_data(x = -5:5, start_col = 11)
wb <- wb_add_conditional_formatting(
  wb,
  "databar",
  dims = wb_dims(cols = 11, rows = 1:11),
  type = "dataBar",
  rule = c(0, 5),
  style = c("#a6a6a6", "#a6a6a6"),
  params = list(showValue = TRUE, gradient = FALSE)
)
```

7.3 Sparklines

	A	B	C	D	E	F	G	H	I	J	K	L
1	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	
2	21	6	160	110	3.9	2.62	16.46	0	1	4	4	
3	21	6	160	110	3.9	2.875	17.02	0	1	4	4	
4	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1	

```
sl <- create_sparklines("Sheet 1", "A3:K3", "L3")
wb <- wb_workbook() |>
   wb_add_worksheet() |>
   wb_add_data(x = mtcars) |>
   wb_add_sparklines(sparklines = sl)
```

8 Charts

The following manual will present various ways to add plots and charts to `openxlsx2` worksheets and even chartsheets. This assumes that you have basic knowledge how to handle `openxlsx2` and are familiar with either the default `R graphics` functions like `plot()` or `barplot()` and `grDevices`, or with the packages {ggplot2}, {rvg} or {mschart}. There are plenty of other manuals that cover using these packages better than we could ever tell you to.

```
## create a workbook
wb <- wb_workbook()
```

8.1 Adding a chart as an image to a workbook

You can include any image in PNG or JPEG format. Simply open a device and save the output and pass it to the worksheet with `wb_add_image()`.

```
myplot <- tempfile(fileext = ".jpg")
jpeg(myplot)
plot(AirPassengers)
invisible(dev.off())

# Add basic plots to the workbook
wb$add_worksheet("add_image")$add_image(file = myplot)
```

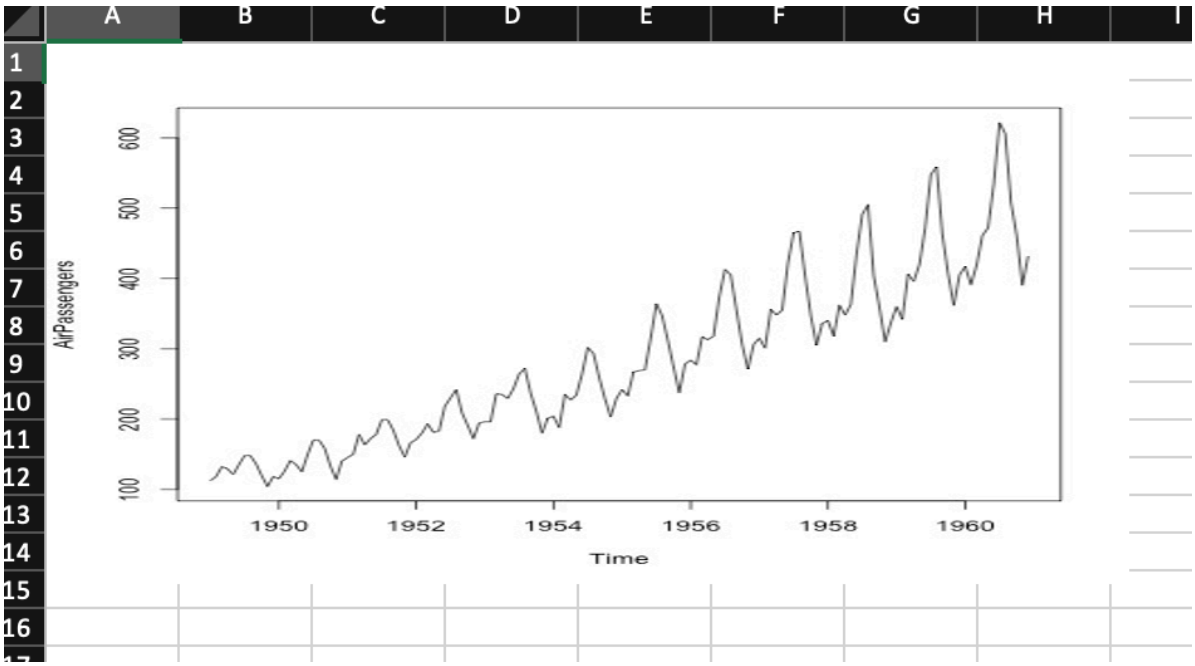


Figure 8.1: The plot output added as image

It is possible to use `{ragg}` to create the png files to add to the worksheet:

```
library(ragg)
ragg_file <- tempfile(fileext = ".png")
agg_png(ragg_file, width = 1000, height = 500, res = 144)
plot(x = mtcars$mpg, y = mtcars$disp)
invisible(dev.off())

wb$add_worksheet("add_image2")$add_image(file = ragg_file)
```

8.2 Adding `{ggplot2}` plots to a workbook

You can include `{ggplot2}` plots similar to how you would include them with `openxlsx`. Call the plot first and afterwards use `wb_add_plot()`.

```
library(ggplot2)

ggplot(mtcars, aes(x = mpg, fill = as.factor(gear))) +
  ggtitle("Distribution of Gas Mileage") +
```

```
  geom_density(alpha = 0.5)

# Add ggplot to the workbook
wb$add_worksheet("add_plot")$
  add_plot(width = 5, height = 3.5, fileType = "png", units = "in")
```

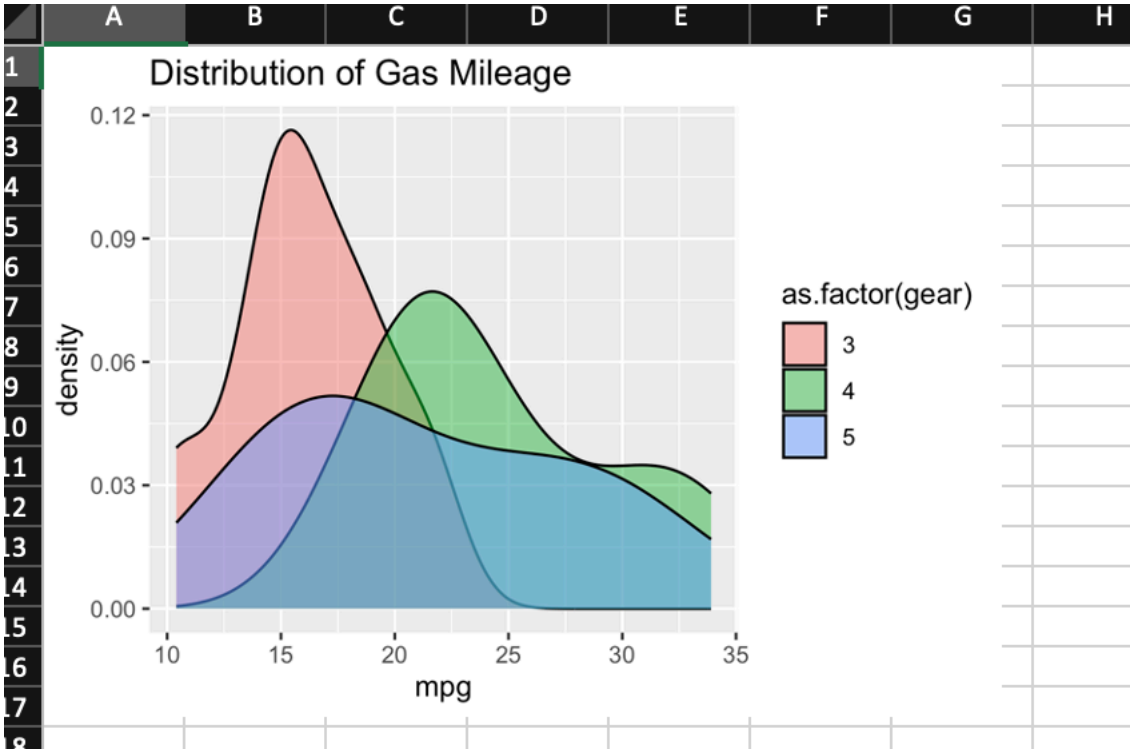


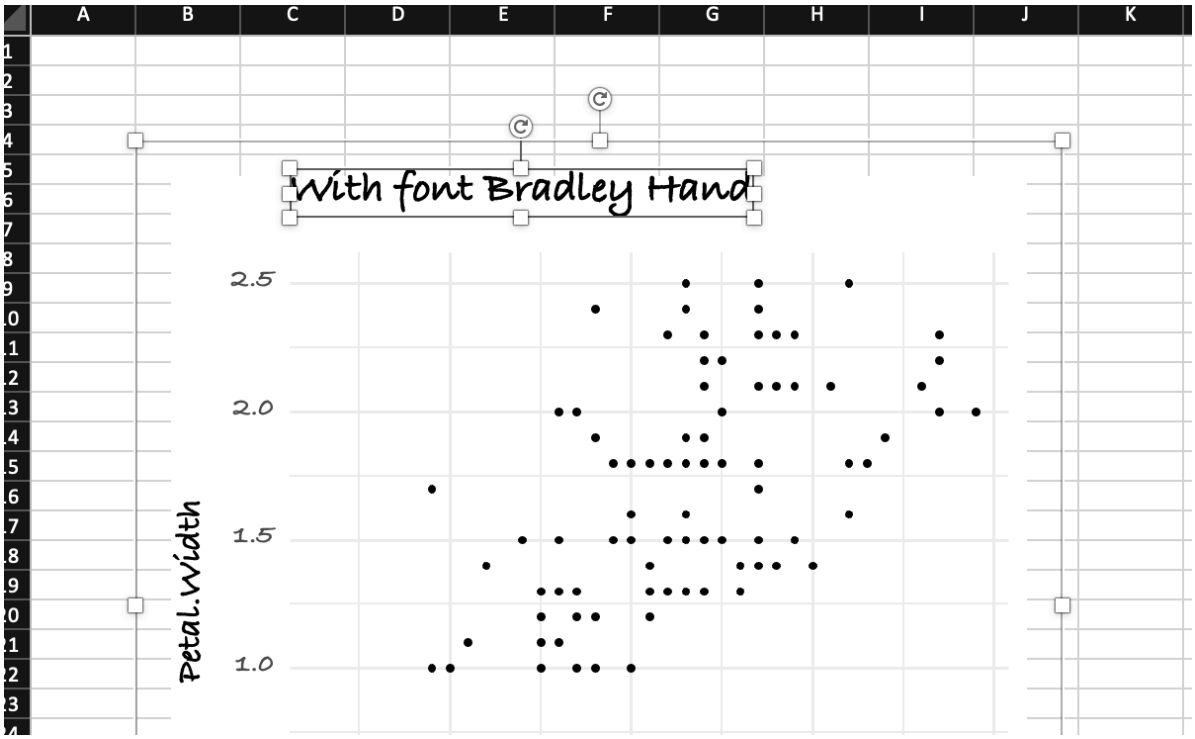
Figure 8.2: The ggplot2 output

8.3 Adding plots via {rvg} or {devEMF}

If you want vector graphics that can be modified in spreadsheet software the `dml_xlsx()` device comes in handy. You can pass the output via `wb_add_drawing()`.

```
library(rvg)

## create rvg example
tmp <- tempfile(fileext = ".xml")
dml_xlsx(file =  tmp, fonts = list(sans = "Bradley Hand"))
ggplot(data = iris,
```

```
       mapping = aes(x = Sepal.Length, y = Petal.Width)) +
  geom_point() + labs(title = "With font Bradley Hand") +
  theme_minimal(base_family = "sans", base_size = 18)
invisible(dev.off())

# Add rvg to the workbook
wb$add_worksheet("add_drawing")$
  add_drawing(xml = tmp)$
  add_drawing(xml = tmp, dims = NULL)
```




```
library(devEMF)

tmp_emf <- tempfile(fileext = ".emf")
devEMF::emf(file = tmp_emf)
ggplot(data = iris,
       mapping = aes(x = Sepal.Length, y = Petal.Width)) +
  geom_point()
#> Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): devEMF: your
#> system substituted font family 'Nimbus Sans' when you requested 'Helvetica'
#> Warning in grid.Call(C_stringMetric, as.graphicsAnnot(x$label)): devEMF: your
#> system substituted font family 'Nimbus Sans' when you requested 'Helvetica'
```

```
dev.off()
#> pdf
#>   2

# Add rvg to the workbook
wb$add_worksheet("add_emf")$
  add_drawing(dims = "A1:D4", xml = tmp)$
  add_image(dims = "E1:H4", file = tmp_emf)
```

8.4 Adding {mschart} plots

If you want native openxml charts, have a look at {mschart}. Create one of the chart files and pass it to the workbook with wb_add_mschart().

There are two options possible.

1. Either the default {mschart} output identical to the one in {officer}. Passing a data object and let {mschart} prepare the data. In this case wb_add_mschart() will add a new data region.
2. Passing a wb_data() object to {mschart}. This object contains references to the data on the worksheet and allows using data "as is".

8.4.1 Add chart and data

```
library(mschart)

## create chart from mschart object (this creates new input data)
mylc <- ms_linechart(
  data = browser_ts,
  x = "date",
  y = "freq",
  group = "browser"
)

wb$add_worksheet("add_mschart")$add_mschart(dims = "A10:G25", graph = mylc)
```

	A	B	C	D	E	F	G	H
1	date	Chrome	IE	Firefox	Safari	Opera	Android	
2	30.09.15	0.514141	0.187879	0.172727	0.10202	0.009091	0.014141	
3	31.12.15	0.497065	0.189824	0.168297	0.109589	0.023483	0.011742	
4	31.03.16	0.515121	0.180962	0.182945	0.089737	0.014378	0.016857	
5	30.06.16	0.508526	0.172518	0.178536	0.110331	0.023069	0.007021	
6	30.09.16	0.521212	0.171717	0.168687	0.10303	0.022222	0.013131	
7	31.12.16	0.506341	0.185366	0.182439	0.091707	0.020488	0.013659	
8								
9								

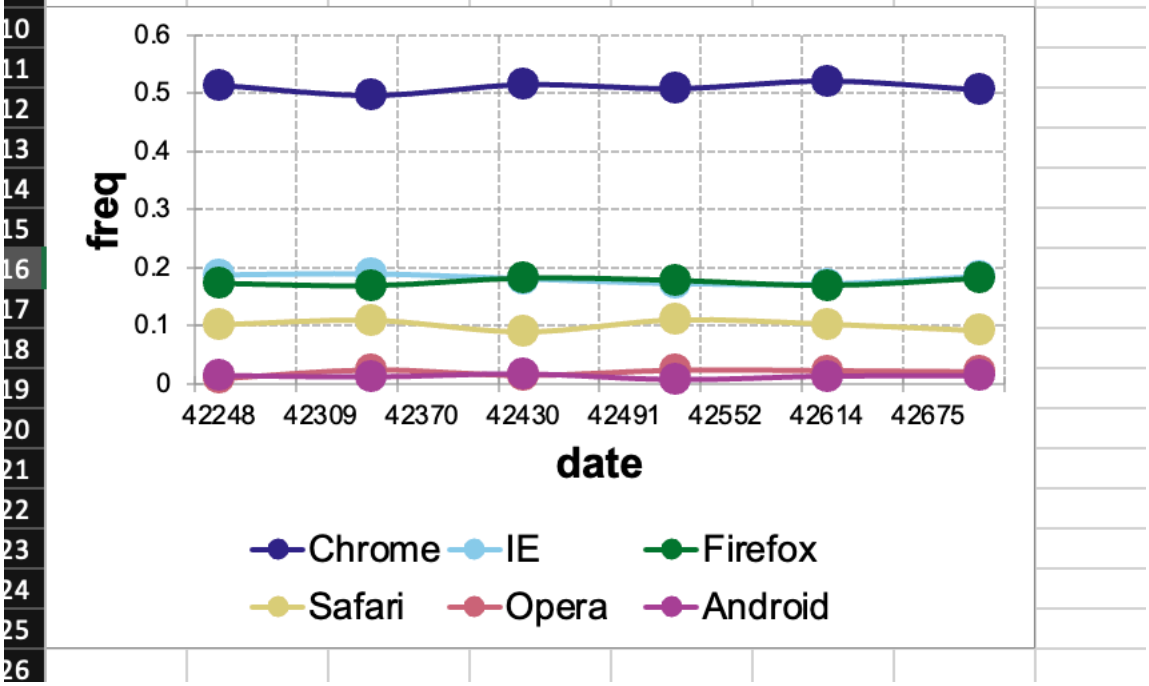


Figure 8.3: An mschart graph

8.4.2 Add chart using `wb_data()`

These are native spreadsheet charts that are dynamic in terms of the data visible on the sheet. It is therefore possible to hide columns or rows of the data, e.g. with groups, so that the chart shows more data when a group is expanded.

```
## create chart referencing worksheet cells as input
# write data starting at B2
wb$add_worksheet("add_mschart - wb_data")$
```

```
  add_data(x = mtcars, dims = "B2")$
  add_data(x = data.frame(name = rownames(mtcars)), dims = "A2")

# create wb_data object this will tell this mschart
# from this PR to create a file corresponding to openxlsx2
dat <- wb_data(wb, dims = "A2:G10")

# create a few mscharts
scatter_plot <- ms_scatterchart(
  data = dat,
  x = "mpg",
  y = c("disp", "hp")
)

bar_plot <- ms_barchart(
  data = dat,
  x = "name",
  y = c("disp", "hp")
)

area_plot <- ms_areachart(
  data = dat,
  x = "name",
  y = c("disp", "hp")
)

line_plot <- ms_linechart(
  data = dat,
  x = "name",
  y = c("disp", "hp"),
  labels = c("disp", "hp")
)

# add the charts to the data
wb <- wb |>
  wb_add_mschart(dims = "F4:L20", graph = scatter_plot) |>
  wb_add_mschart(dims = "F21:L37", graph = bar_plot) |>
  wb_add_mschart(dims = "M4:S20", graph = area_plot) |>
  wb_add_mschart(dims = "M21:S37", graph = line_plot)
```

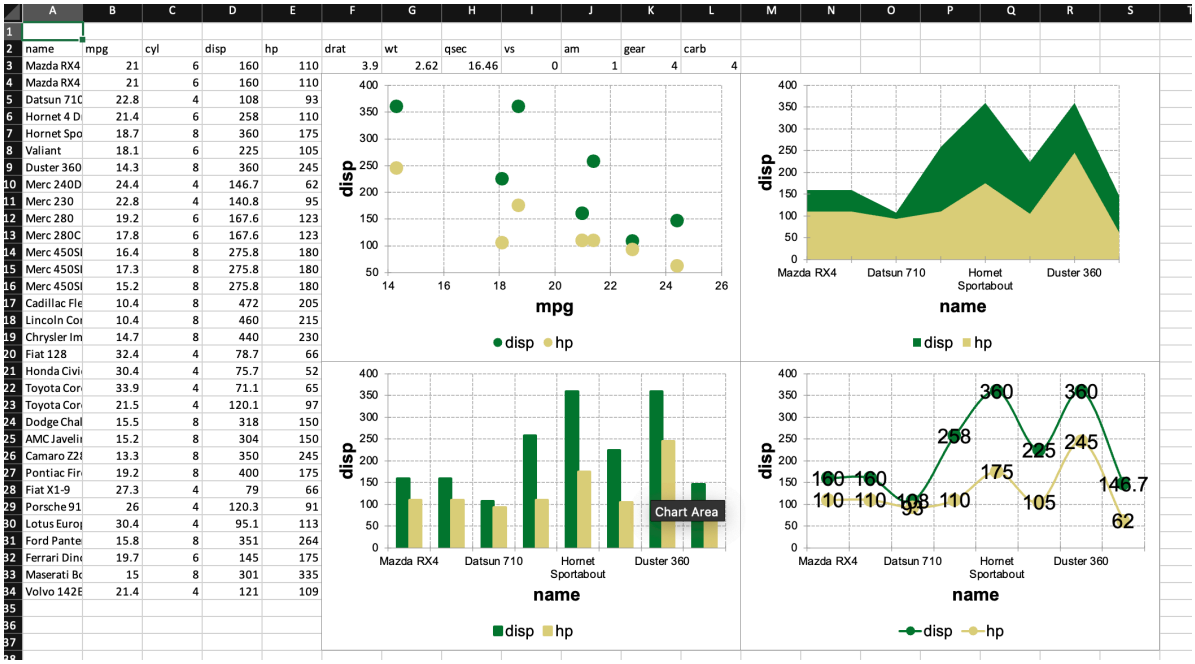


Figure 8.4: Multiple mschart graphs in a single spreadsheet

8.4.3 Add and fill a chartsheet

Finally it is possible to add `mschart` charts on a so called chartsheet. These are special sheets that contain only a chart object, referencing data from another sheet.

```
# add chartsheet
wb <- wb |>
  wb_add_chartsheet() |>
  wb_add_mschart(graph = scatter_plot)
```

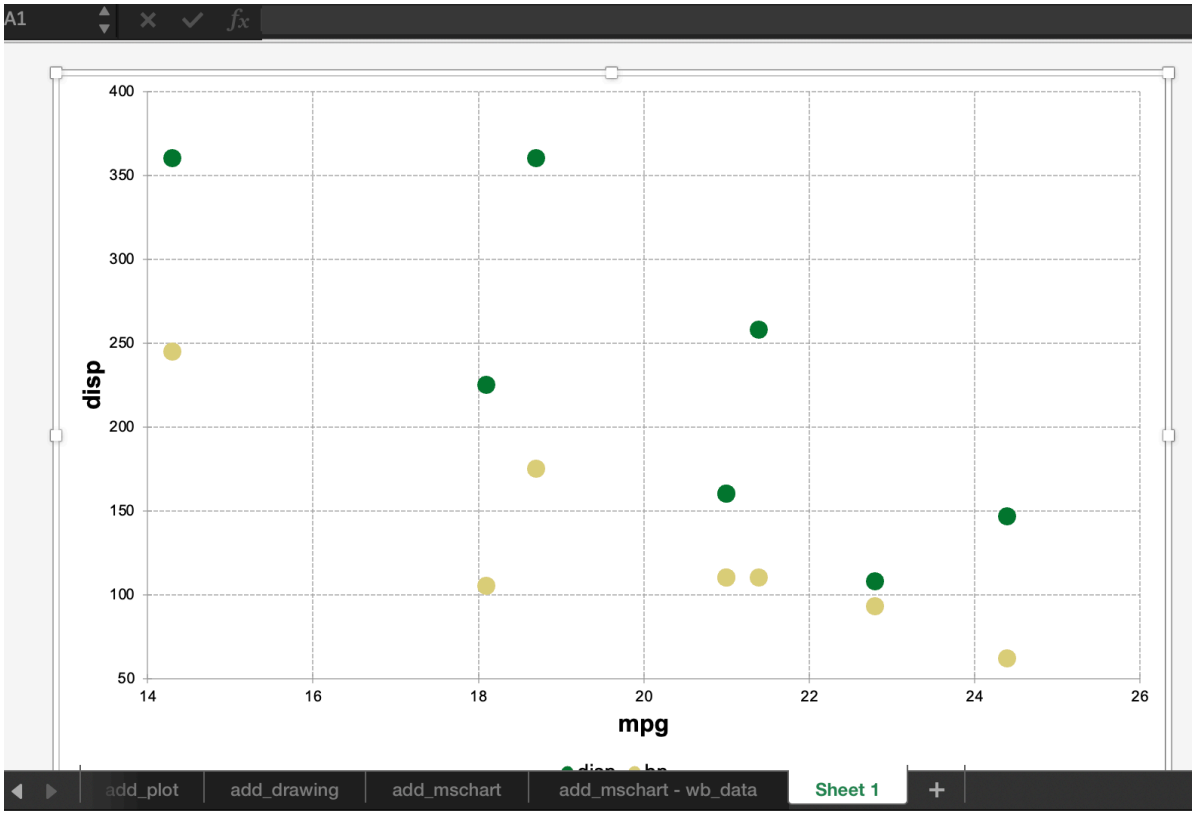


Figure 8.5: A mschart graph on a chartsheet

9 Spreadsheet formulas

Below you find various examples how to create formulas with `openxlsx2`. Though, before we start with the examples, let us begin with a word of warning. Please be aware, while it is possible to create all these formulas, they are not evaluated unless they are opened in spreadsheet software. Even worse, if there are cells containing the result of some formula, it can not be trusted unless the formula is evaluated in spreadsheet software.

This can be shown in a simple example: We have a spreadsheet with a formula `A1 + B1`. This formula was evaluated with spreadsheet software as `A1 + B1 = 2`. Therefore if we read the cell, we see the value 2. Lets recreate this output in `openxlsx2`

```
# Create artificial xlsx file
wb <- wb_workbook()$add_worksheet()$add_data(x = t(c(1, 1)), col_names = FALSE)$
  add_formula(dims = "C1", x = "A1 + B1")
# Users should never modify cc as shown here
wb$worksheets[[1]]$sheet_data$cc$v[3] <- 2

# we expect a value of 2
wb_to_df(wb, col_names = FALSE)
#>   A B C
#> 1 1 1 2
```

Now, lets assume we modify the data in cell `A1`.

```
wb$add_data(x = 2)

# we expect 3
wb_to_df(wb, col_names = FALSE)
#>   A B C
#> 1 2 1 2
```

What happened? Even though we see cells `A1` and `B1` show a value of `2` and `1` our formula in `C1` was not updated. It still shows a value of `2`. This is because `openxlsx2` does not evaluate formulas and workbooks on a more general scale. In the open xml style the cell looks something like this:

```
<c r="C1">
  <f>A1 + B1</f>
  <v>2</v>
</c>
```

And when we read from this cell, we always return the value of `v`. In this case it is obvious, but still wrong and it is a good idea to check if underlying fields contain formulas.

```
wb_to_df(wb, col_names = FALSE, show_formula = TRUE)
#>   A B       C
#> 1 2 1 A1 + B1
```

If `openxlsx2` writes formulas, as shown in the examples below, the fields will be entirely blank. These fields will only be evaluated and filled, once the output file is opened in spreadsheet software.

The only way to avoid surprises is to be aware of this all the time and similar, checking for similar things all the time.

9.1 Simple formulas

Generally speaking it is possible to use all valid formulas allowed in spreadsheet software. This can be functions, arithmetic operators or a mix of both. It's possible to create functions for all by spreadsheet software supported functions, including custom vml ones (though this requires a workbook that was loaded with the required macros).

```
wb <- wb_workbook()$add_worksheet()$
  add_data(x = head(cars))$
  add_formula(x = "SUM(A2, B2)", dims = "D2")$
  add_formula(x = "A2 + B2", dims = "D3")
```

9.2 Array formulas

Array formulas in openxml spreadsheets allow performing multiple calculations on a data vector or 'array' instead of a single cell. An array is similar to a vector in R. Unlike regular formulas that operate on a single value, array formulas can process multiple values simultaneously. An important distinction is that that you need array formulas, whenever the formula evaluates an array, even if the output creates only a single cell. So something like this `SUM(ABS(A2:A11))`

would require an array formula, because the `SUM()` function is called on a function that returns an array `ABS(A2:A11)`. If the previous formula in written as basic formula, spreadsheet software is likely to mess it up and tries to insert `@` characters in the formula.

```
wb <- wb_workbook()$add_worksheet()$
  add_data(x = head(cars))$
  add_formula(x = "A2:A7 * B2:B7", dims = "C2:C7", array = TRUE)
```

9.3 Array formulas creating multiple fields

In the example below we want to use `MMULT()` which creates a matrix multiplication. This requires us to write an array formula and to specify the region where the output will be written to.

```
m1 <- matrix(1:6, ncol = 2)
m2 <- matrix(7:12, nrow = 2)

wb <- wb_workbook()$add_worksheet()$
  add_data(x = m1, startCol = 1)$
  add_data(x = m2, startCol = 4)$
  add_formula(x = "MMULT(A2:B4, D2:F3)", dims = "H2:J4", array = TRUE)
# wb$open()
```

Similar a the coefficients of a linear regression

```
# we expect to find this in D1:E1
coef(lm(head(cars)))
#> (Intercept)        dist
#>   5.2692308   0.1153846
wb <- wb_workbook()$add_worksheet()$
  add_data(x = head(cars))$
  add_formula(x = "LINEST(A2:A7, B2:B7, TRUE)", dims = "D2:E2", array = TRUE)
# wb$open()
```

9.4 Modern spreadsheet functions

Spreadsheet functions are constantly evolving and similarly extended. Several formulas introduced in the MS365 Excel versions require `_xlfn.` as prefix for the function name. Such formulas will only be evaluated with compatible spreadsheet software. In case of doubt, see

Excel functions (alphabetical) for a list of all functions and an indicator for the software version in which they were introduced.

```
wb <- wb_workbook()$add_worksheet()$
  add_data(x = cars)$
  add_data(dims = "D1", x = "Unique Values of Speed")$
  add_formula(
    dims = wb_dims(x = unique(cars$speed), from_col = "D", from_row = 2),
    x = paste0("_xlfn.UNIQUE(", wb_dims(x = cars, cols = "speed"), ")"),
    array = TRUE
  )
```

9.5 Shared formulas

A neat feature in spreadsheet software is that you can drag cells around to fill cells with content of other cells. Whenever you are dragging a cell containing a formula, this formula will be extended onto other cell regions. This is called a shared formula. In `openxlsx2` you can use shared formulas starting with release `1.9`.

```
df <- data.frame(
  x = 1:5,
  y = 1:5 * 2
)

wb <-  wb_workbook()$add_worksheet()$add_data(x = df)$
  add_formula(x = "=A2/B2", dims = "C2:C6", shared = TRUE)$
  add_formula(x = "=A$2/B$2", dims   = "D2:D6", shared = TRUE)

wb_to_df(wb, show_formula = TRUE)
#>   x  y   <NA>     <NA>
#> 2 1  2 =A2/B2 =A$2/B$2
#> 3 2  4 =A3/B3 =A$2/B$2
#> 4 3  6 =A4/B4 =A$2/B$2
#> 5 4  8 =A5/B5 =A$2/B$2
#> 6 5 10 =A6/B6 =A$2/B$2
```

9.6 Cell error handling

Spreadsheet users will be familiar with various errors thrown once formulas are used. These are not always useful in spreadsheet software and can be removed using `wb_add_ignore_error()`.

This function allows to fine tune the errors that are returned per cell.

```
wb <- wb_workbook()$add_worksheet()$
  add_data(dims = "B1", x = t(c(1, 2, 3)), colNames = FALSE)$
  add_formula(dims = "A1", x = "SUM(B1:C1)")$
  add_ignore_error(dims = "A1", formulaRange = TRUE)
```

9.7 cells metadata (cm) formulas

Similar to array formulas, these cell metadata (cm) formulas hide to the user that they are array formulas. Using these is implemented in `openxlsx2` > 0.6.1:

```
wb <- wb_workbook()$add_worksheet()$
  add_data(x = head(cars))$
  add_formula(x = 'SUM(ABS(A2:A7))', dims = "D2", cm = TRUE)
#> Warning in write_data2(wb = wb, sheet = sheet, data = x, name = name, colNames
#> = colNames, : modifications with cm formulas are experimental. use at own risk
# wb$open()
```

9.8 `dataTable` formulas[1]

	A	B	C
1	sales_price	COGS	sales_quantity
2	20	5	1
3	30	11	2
4	40	13	3

Given a basic table like the above, a similarly basic formula for `total_sales` would be "= A2 * C2" with the row value changing at each row.

An implementation for this formula using `wb_add_formula()` would look this (taken from current documentation) lets say we've read in the data and assigned it to the table `company_sales`

[1] this example was originally provided by @zykezero for `openxlsx`.

```r
## creating example data
company_sales <- data.frame(
    sales_price = c(20, 30, 40),
    COGS = c(5, 11, 13),
    sales_quantity = c(1, 2, 3)
)

## write in the formula
company_sales$total_sales  <- paste(paste0("A", 1:3 + 1L),
                                    paste0("C", 1:3 + 1L), sep = " * ")
## add the formula class
class(company_sales$total_sales) <- c(class(company_sales$total_sales),
                                      "formula")

## write a workbook
wb <- wb_workbook()$
  add_worksheet("Total Sales")$
  add_data_table(x = company_sales)
```

Then we create the workbook, worksheet, and use `wb_add_data_table()`.

One of the advantages of the open xml `dataTable` syntax is that we don't have to specify row numbers or columns as letters. The table also grows dynamically, adding new rows as new data is appended and extending formulas to the new rows. These `dataTable` have named columns that we can use instead of letters. When writing the formulas within the `dataTable` we would use the following syntax `[@[column_name]]` to reference the current row. So the `total_sales` formula written in open xml in `dataTable` would look like this; `=[@[sales_price]] * [@[sales_quantity]]`

If we are writing the formula outside of the `dataTable` we have to reference the table name. In this case lets say the table name is 'daily_sales' `=daily_sales[@[sales_price]] * daily_sales[@[sales_quantity]]`

However, if we were to pass this as the text for the formula to be written it would cause an error because the syntax that open xml requires for selecting the current row is different.

In open xml the `dataTable` formula looks like this:

```
<calculatedColumnFormula>
  daily_sales[[#This Row],[sales_price]]*daily_sales[[#ThisRow],[sales_quantity]]
</calculatedColumnFormula>
```

Now we can see that open xml replaces `[@[sales_price]]` with `daily_sales[[#This Row],[sales_price]]` We must then use this syntax when writing formulas for `dataTable`

```
## Because we want the `dataTable` formula to propagate down the entire column
## of the data we can assign the formula by itself to any column and allow that
## single string to be repeated for each row.

## creating example data
example_data <-
  data.frame(
    sales_price = c(20, 30, 40),
    COGS = c(5, 11, 13),
    sales_quantity = c(1, 2, 3)
  )

## base R method
example_data$gross_profit       <- "daily_sales[[#This Row],[sales_price]] -
daily_sales[[#This Row],[COGS]]"
example_data$total_COGS         <- "daily_sales[[#This Row],[COGS]] *
daily_sales[[#This Row],[sales_quantity]]"
example_data$total_sales        <- "daily_sales[[#This Row],[sales_price]] *
daily_sales[[#This Row],[sales_quantity]]"
example_data$total_gross_profit <- "daily_sales[[#This Row],[total_sales]] -
daily_sales[[#This Row],[total_COGS]]"
class(example_data$gross_profit)       <- c(class(example_data$gross_profit),
                                            "formula")
class(example_data$total_COGS)         <- c(class(example_data$total_COGS),
                                            "formula")
class(example_data$total_sales)        <- c(class(example_data$total_sales),
                                            "formula")
class(example_data$total_gross_profit) <- c(
  class(example_data$total_gross_profit), "formula")
```

```
wb$
  add_worksheet("Daily Sales")$
  add_data_table(
    x           = example_data,
    table_style = "TableStyleMedium2",
    table_name  = "daily_sales"
  )
```

And if we open the workbook to view the table we created we can see that the formula has worked.

	A	B	C	D	E	F	G
1	sales_price	COGS	sales_quantity	gross_profit	total_COGS	total_sales	total_gross_profit
2	20	5	1	15	5	20	15
3	30	11	2	19	22	60	38
4	40	13	3	27	39	120	81

We can also see that it has replaced `[#This Row]` with `@`.

	A	B	C	D	E	F	G
1	sales_price	COGS	sales_quantity	gross_profit	total_COGS	total_sales	total_gross_profit
2	20	5	1	=[@sales_price] - [@COGS]	=[@COGS] * [@sales_quantity]	=[@sales_price] * [@sales_quantity]	=[@[total_sales]] - [@[total_COGS]]
3	30	11	2	=[@sales_price] - [@COGS]	=[@COGS] * [@sales_quantity]	=[@sales_price] * [@sales_quantity]	=[@[total_sales]] - [@[total_COGS]]
4	40	13	3	=[@sales_price] - [@COGS]	=[@COGS] * [@sales_quantity]	=[@sales_price] * [@sales_quantity]	=[@[total_sales]] - [@[total_COGS]]

For completion, the formula as we wrote it appears as;

D	E	F	G
gross_profit	total_COGS	total_sales	total_gross_profit
=gross_profit[[#This Row],[sales_price]] - gross_profit[[#This Row],[COGS]]	=gross_profit[[#This Row],[COGS]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[sales_price]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[total_sales]] - gross_profit[[#This Row],[total_COGS]]
=gross_profit[[#This Row],[sales_price]] - gross_profit[[#This Row],[COGS]]	=gross_profit[[#This Row],[COGS]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[sales_price]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[total_sales]] - gross_profit[[#This Row],[total_COGS]]
=gross_profit[[#This Row],[sales_price]] - gross_profit[[#This Row],[COGS]]	=gross_profit[[#This Row],[COGS]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[sales_price]] * gross_profit[[#This Row],[sales_quantity]]	=gross_profit[[#This Row],[total_sales]] - gross_profit[[#This Row],[total_COGS]]

```r
#### sum dataTable examples
wb$add_worksheet("sum_examples")

### Note: dataTable formula do not need to be used inside of dataTables.
### dataTable formula are for referencing the data within the dataTable.
sum_examples <- data.frame(
  description = c("sum_sales_price", "sum_product_Price_Quantity"),
  formula = c("", "")
)

wb$add_data(x = sum_examples)

# add formulas
wb$add_formula(x = "sum(daily_sales[[#Data],[sales_price]])", dims = "B2")
wb$add_formula(x = "sum(daily_sales[[#Data],[sales_price]] *
               daily_sales[[#Data],[sales_quantity]])", dims = "B3",
               array = TRUE)

#### dataTable referencing
wb$add_worksheet("dt_references")

### Adding the headers by themselves.
wb$add_formula(
  x = "daily_sales[[#Headers],[sales_price]:[total_gross_profit]]",
  dims = "A1:G1",
  array = TRUE
)

### Adding the raw data by reference and selecting them directly.
wb$add_formula(
  x = "daily_sales[[#Data],[sales_price]:[total_gross_profit]]",
  start_row = 2,
  dims = "A2:G4",
  array = TRUE
)
# wb$open()
```

10 Pivot tables

Pivot tables are a feature of spreadsheet software dating back to Lotus Improv. They allow creating interactive tables to aggregate data that still allows the user to modify the table, by changing the aggregation function or variables. Pivot tables are frequently used in reports to create something like a dashboard.

Even though they are a long requested feature, it took a while until support was added to `openxlsx2`. Since release 0.5 users are able to use `wb_add_pivot_table()` and since then support was further improved and now it is also possible to add slicers to pivot tables. Slicers further increase the dashboard character of pivot tables, as they provide a button interface to filter the pivot table.

The state of pivot tables is now that they work quite well, though they bring a few features users should be aware of. Most importantly, our function only provides the spreadsheet with an instruction set how to create the pivot table, while the actual sheet where the table is supposed to appear remains empty until it is evaluated by the spreadsheet software. This is similar to our approach with formulas.

Please, though, be a little careful if you start experimenting with pivot table params as there are actual cases, where the instruction set results into spreadsheet software crashes. Make copies and try to prevent some headaches afterwards. Also it is a good idea to check the expected outcome of a pivot table. We will make use of the `{pivottabler}` package for this in this chapter, but there are obviously other ways in base R, or the many data wrangling packages like `{data.table}`, `{dplyr}` or `{polars}`.

10.1 Adding pivot tables

```
wb <- wb_workbook()$
  add_worksheet()$
  add_data(x = esoph)

df <- wb_data(wb)

wb$add_pivot_table(df, rows = "agegp", cols = "tobgp", data = c("ncontrols"))
```

```r
# for visual comparison
pt <- pivottabler::PivotTable$new()
pt$addData(esoph)
pt$addColumnDataGroups("tobgp")
pt$addRowDataGroups("agegp")
pt$defineCalculation(calculationName = "ncontrols",
                     summariseExpression = "sum(ncontrols)")
pt$evaluatePivot()
pt
#>        0-9g/day  10-19  20-29  30+  Total
#> 25-34        70     18     11   16    115
#> 35-44       107     42     24   17    190
#> 45-54        90     44     25    8    167
#> 55-64        92     42     26    6    166
#> 65-74        68     26     10    2    106
#> 75+          20      6      3    2     31
#> Total       447    178     99   51    775

wb$add_data_table(dims = "A14", x = pt$asDataFrame(), row_names = TRUE)

if (interactive()) wb$open()
```

	A	B	C	D	E	F
1						
2						
3	Sum of ncontrols	Column Labels				
4	Row Labels	0-9g/day	10-19	20-29	30+	Grand Total
5	25-34	70	18	11	16	115
6	35-44	107	42	24	17	190
7	45-54	90	44	25	8	167
8	55-64	92	42	26	6	166
9	65-74	68	26	10	2	106
10	75+	20	6	3	2	31
11	Grand Total	447	178	99	51	775
12						
13						
14	_rowNames_	0-9g/day	10-	20-	3	Total
15	25-34	70	18	11	16	115
16	35-44	107	42	24	17	190
17	45-54	90	44	25	8	167
18	55-64	92	42	26	6	166
19	65-74	68	26	10	2	106
20	75+	20	6	3	2	31
21	Total	447	178	99	51	775
22						

Unlike `pivottabler` the pivot tables in `openxlsx2` are not evaluated. Therefore there is nothing in the sheet region `A3:F11` and if you write something here, spreadsheet software will complain.[1]

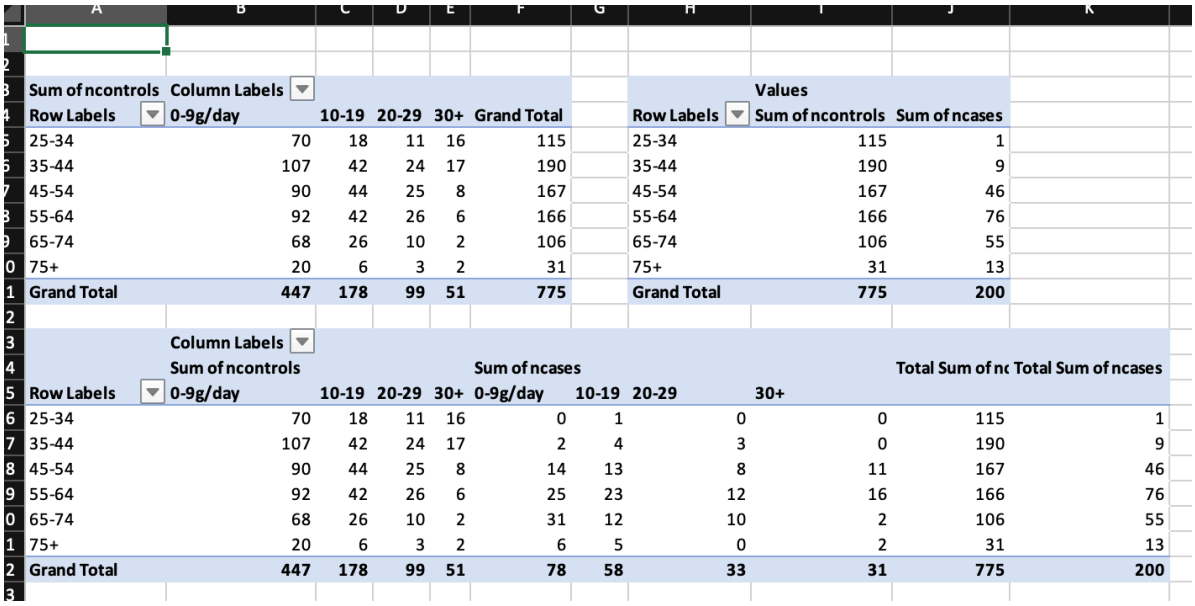
10.1.1 Filter, row, column, and data

Similar to pivot tables in Excel, it is possible to assign variables to the table dimensions filter, row, column, and data. It is not required to have all dimensions filled. You can assign each variable only once per dimension, but can combine multiple variables.

```
wb <- wb_workbook()$
  add_worksheet()$
  add_data(x = esoph)
```

[1] It should be possible to integrate results similar to `pivottabler` into `wb_add_pivot_table()` so that you should be able to have evaluated pivot tables straight ahead. Pull requests are welcome.

```
df <- wb_data(wb)

wb$add_pivot_table(df, dims = "A3", rows = "agegp", cols = "tobgp",
                   data = c("ncontrols"))
wb$add_pivot_table(df, dims = "A13", rows = "agegp",
                   data = c("ncontrols", "ncases"))
wb$add_pivot_table(df, dims = "A18", rows = "agegp", cols = "tobgp",
                   data = c("ncontrols", "ncases"))
```



Sum of ncontrols	Column Labels				
Row Labels	0-9g/day	10-19	20-29	30+	Grand Total
25-34	70	18	11	16	115
35-44	107	42	24	17	190
45-54	90	44	25	8	167
55-64	92	42	26	6	166
65-74	68	26	10	2	106
75+	20	6	3	2	31
Grand Total	447	178	99	51	775

	Values	
Row Labels	Sum of ncontrols	Sum of ncases
25-34	115	1
35-44	190	9
45-54	167	46
55-64	166	76
65-74	106	55
75+	31	13
Grand Total	775	200

	Column Labels									
	Sum of ncontrols				Sum of ncases				Total Sum of nc	Total Sum of ncases
Row Labels	0-9g/day	10-19	20-29	30+	0-9g/day	10-19	20-29	30+		
25-34	70	18	11	16	0	1	0	0	115	1
35-44	107	42	24	17	2	4	3	0	190	9
45-54	90	44	25	8	14	13	8	11	167	46
55-64	92	42	26	6	25	23	12	16	166	76
65-74	68	26	10	2	31	12	10	2	106	55
75+	20	6	3	2	6	5	0	2	31	13
Grand Total	447	178	99	51	78	58	33	31	775	200

10.1.2 Sorting

Using `sort_item` it is possible to order the pivot table. `sort_item` can take either integers or characters, the latter is beneficial in cases as below, where the variable you want to sort is a factor. Though, be aware that pivot table uses a different approach to distinct unique elements and that `Berlin` and `BERLIN` are identical to it. You can check for distinct cases with `openxlsx2:::distinct()`.

```
tbl_prueba_2 <- data.frame(
  var_1 = as.Date(rep(
    c(
      "2023-02-01", "2023-03-01", "2023-04-01", "2023-05-01", "2023-06-01",
      "2023-07-01", "2023-08-01", "2023-09-01", "2023-10-01", "2023-11-01",
      "2023-12-01", "2024-01-01", "2024-02-01", "2024-03-01"
```

```
    ),
    each = 2L
  )),
  var_2 = rep(2:15, each = 2L),
  year = rep(c(2023, 2024), c(22L, 6L)),
  month = ordered(
    rep(
      c(
        "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov",
        "Dec", "Jan", "Feb", "Mar"
      ),
      each = 2L
    ),
    levels = c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
               "Oct", "Nov", "Dec")
  )
)


wb_1 <- wb_workbook() |>
  wb_add_worksheet() |>
  wb_add_data(x = tbl_prueba_2)

df <- wb_data(wb_1)

wb_1 <- wb_1 |>
  wb_add_pivot_table(
    x = df,
    cols = c("year", "month"),
    data = "var_2",
    fun = "sum",
    params = list(
      sort_item = list(month = rev(levels(tbl_prueba_2$month)))
    )
  )

if (interactive()) wb_1$open()
```

10.1.3 Aggregation functions

The default aggregation function is `SUM`, but others are possible as well: `AVERAGE`, `COUNT`, `COUNTA`, `MAX`, `MIN`, `PRODUCT`, `STDEV`, `STDEVP`, `SUM`, `VAR`, `VARP`. This is limited to functions available in the openxml specification. Each data variable can use a different function.

```
wb <- wb_workbook()$
  add_worksheet()$
  add_data(x = mtcars)

df <- wb_data(wb)

wb$add_pivot_table(df, dims = "A1", rows = "cyl", cols = "gear",
                   data = c("disp", "hp"))
wb$add_pivot_table(df, dims = "A10", sheet = 2, rows = "cyl", cols = "gear",
                   data = c("disp", "hp"), fun = "count")
wb$add_pivot_table(df, dims = "A20", sheet = 2, rows = "cyl", cols = "gear",
                   data = c("disp", "hp"), fun = "average")
wb$add_pivot_table(df, dims = "A30", sheet = 2, rows = "cyl", cols = "gear",
                   data = c("disp", "hp"), fun = c("sum", "average"))
```

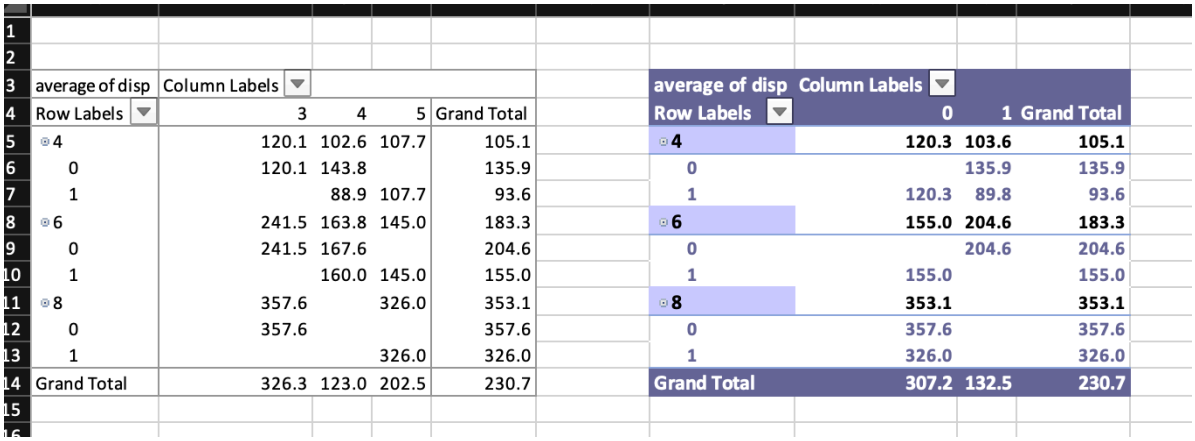
	A	B	C	D	E	F	G	H	I	J
1		Column Labels								
2		Sum of disp			Sum of hp			Total Sum of disp	Total Sum of hp	
3	Row Labels	3	4	5	3	4	5			
4	4	120.1	821	215.4	97	608	204	1156.5	909	
5	6	483	655.2	145	215	466	175	1283.2	856	
6	8	4291.4		652	2330		599	4943.4	2929	
7	Grand Total	4894.5	1476.2	1012.4	2642	1074	978	7383.1	4694	
8										
9										
10		Column Labels								
11		count of disp			count of hp			Total count of disp	Total count of hp	
12	Row Labels	3	4	5	3	4	5			
13	4	1	8	2	1	8	2	11	11	
14	6	2	4	1	2	4	1	7	7	
15	8	12		2	12		2	14	14	
16	Grand Total	15	12	5	15	12	5	32	32	
17										
18										
19										
20		Column Labels								
21		average of disp			average of hp			Total average of di	Total average of hp	
22	Row Labels	3	4	5	3	4	5			
23	4	120.1	102.63	107.7	97	76	102	105.1363636	82.63636364	
24	6	241.5	163.8	145	107.5	117	175	183.3142857	122.2857143	
25	8	357.6166667		326	194.167		300	353.1	209.2142857	
26	Grand Total	326.3	123.02	202.48	176.133	89.5	196	230.721875	146.6875	
27										
28										
29										
30		Column Labels								
31		sum of disp			average of hp			Total sum of disp	Total average of hp	
32	Row Labels	3	4	5	3	4	5			
33	4	120.1	821	215.4	97	76	102	1156.5	82.63636364	
34	6	483	655.2	145	107.5	117	175	1283.2	122.2857143	
35	8	4291.4		652	194.167		300	4943.4	209.2142857	
36	Grand Total	4894.5	1476.2	1012.4	176.133	89.5	196	7383.1	146.6875	
37										

10.1.4 Styling pivot tables

There is no real support for individual pivot table styles. Aside from the default style, it is possible to disable the style and to apply auto format styles (for various styles see annex `G.3 - Built-in PivotTable AutoFormats` of ECMA-376-1 (2016)). In the example below style id `4099` is applied, ids range from `4096` to `4117`.

```
wb <- wb_workbook() |>
  wb_add_worksheet("table") |>
```

```
  wb_add_worksheet("data") |>
  wb_add_data(x = mtcars)

df <- wb_data(wb)

wb <- wb |>

  # pivot table without style
  wb_add_pivot_table(
    df, dims = "A3", sheet = "table",
    rows = c("cyl", "am"), cols = "gear", data = "disp",
    fun = "average",
    params = list(no_style = TRUE, numfmt = c(formatCode = "##0.0"))
  ) |>

  # Applied a few params and use auto_format_id
  wb_add_pivot_table(
    df, dims = "G3", sheet = "table",
    rows = c("cyl", "am"), cols = "vs", data = "disp",
    fun = "average",
    params = list(
      apply_alignment_formats    = TRUE,
      apply_number_formats       = TRUE,
      apply_border_formats       = TRUE,
      apply_font_formats         = TRUE,
      apply_pattern_formats      = TRUE,
      apply_width_height_formats = TRUE,
      auto_format_id             = 4099,
      numfmt = c(formatCode = "##0.0")
    )
  )

if (interactive()) wb$open()
```



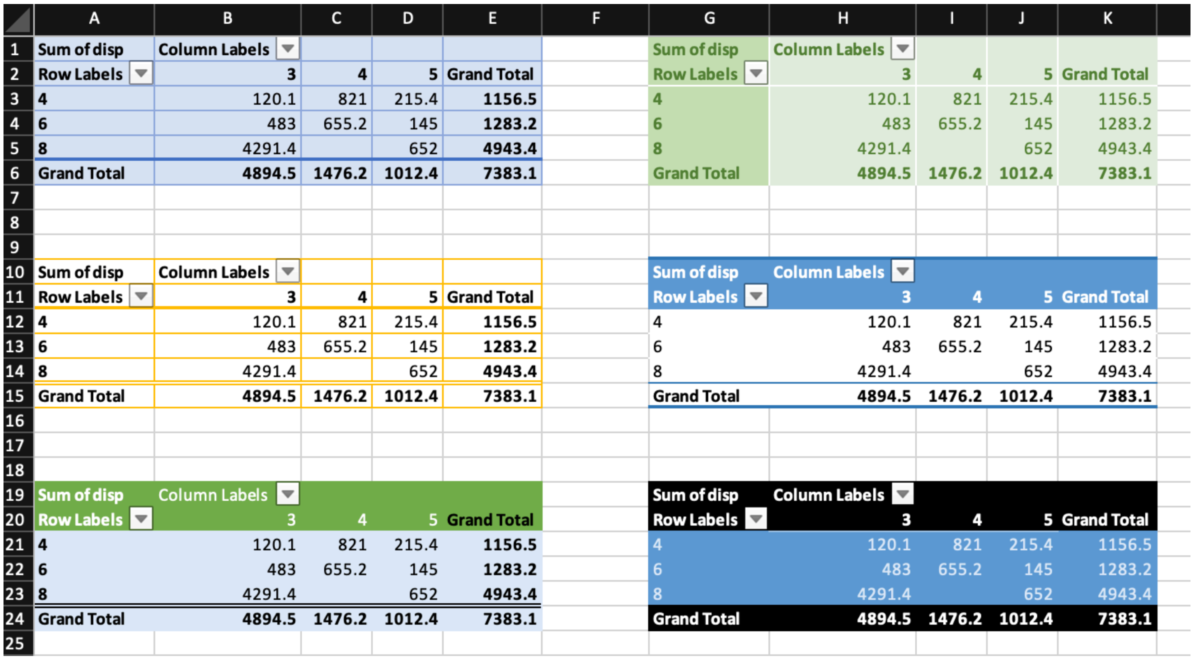
average of disp	Column Labels			
Row Labels	3	4	5	Grand Total
4	120.1	102.6	107.7	105.1
0	120.1	143.8		135.9
1		88.9	107.7	93.6
6	241.5	163.8	145.0	183.3
0	241.5	167.6		204.6
1		160.0	145.0	155.0
8	357.6		326.0	353.1
0	357.6			357.6
1			326.0	326.0
Grand Total	326.3	123.0	202.5	230.7

average of disp	Column Labels		
Row Labels	0	1	Grand Total
4	**120.3**	**103.6**	**105.1**
0		135.9	135.9
1	120.3	89.8	93.6
6	**155.0**	**204.6**	**183.3**
0		204.6	204.6
1	155.0		155.0
8	**353.1**		**353.1**
0	357.6		357.6
1	326.0		326.0
Grand Total	307.2	132.5	230.7

With `params` it is possible to tweak many pivot table arguments such as `params = list(col_header_caption = "test caption")`. This way it is also possible to apply built in pivot table styles. The default is `PivotStyleLight16` (for more built in styles see `G.1 Built-in Table Styles` of ECMA-376-1 (2016)).

```
wb <- wb_workbook()$
  add_worksheet("table")$
  add_worksheet("data")$add_data(x = mtcars)

df <- wb_data(wb)

wb$add_pivot_table(df, sheet = "table", dims = "A1", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "TableStyleMedium23"))
wb$add_pivot_table(df, sheet = "table", dims = "A10", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "TableStyleLight19"))
wb$add_pivot_table(df, sheet = "table", dims = "A19", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "TableStyleDark11"))

wb$add_pivot_table(df, sheet = "table", dims = "G1", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "PivotStyleMedium28"))
wb$add_pivot_table(df, sheet = "table", dims = "G10", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "PivotStyleMedium13"))
wb$add_pivot_table(df, sheet = "table", dims = "G19", rows = "cyl",
                   cols = "gear", data = "disp", params = list(
                     table_style = "PivotStyleDark20"))
```

```r
if (interactive()) wb$open()
```



	A	B	C	D	E	F	G	H	I	J	K
1	Sum of disp	Column Labels					Sum of disp	Column Labels			
2	Row Labels	3	4	5	Grand Total		Row Labels	3	4	5	Grand Total
3	4	120.1	821	215.4	1156.5		4	120.1	821	215.4	1156.5
4	6	483	655.2	145	1283.2		6	483	655.2	145	1283.2
5	8	4291.4		652	4943.4		8	4291.4		652	4943.4
6	Grand Total	4894.5	1476.2	1012.4	7383.1		Grand Total	4894.5	1476.2	1012.4	7383.1
7											
8											
9											
10	Sum of disp	Column Labels					Sum of disp	Column Labels			
11	Row Labels	3	4	5	Grand Total		Row Labels	3	4	5	Grand Total
12	4	120.1	821	215.4	1156.5		4	120.1	821	215.4	1156.5
13	6	483	655.2	145	1283.2		6	483	655.2	145	1283.2
14	8	4291.4		652	4943.4		8	4291.4		652	4943.4
15	Grand Total	4894.5	1476.2	1012.4	7383.1		Grand Total	4894.5	1476.2	1012.4	7383.1
16											
17											
18											
19	Sum of disp	Column Labels					Sum of disp	Column Labels			
20	Row Labels	3	4	5	Grand Total		Row Labels	3	4	5	Grand Total
21	4	120.1	821	215.4	1156.5		4	120.1	821	215.4	1156.5
22	6	483	655.2	145	1283.2		6	483	655.2	145	1283.2
23	8	4291.4		652	4943.4		8	4291.4		652	4943.4
24	Grand Total	4894.5	1476.2	1012.4	7383.1		Grand Total	4894.5	1476.2	1012.4	7383.1
25											

10.1.5 Pivot table `dims`

It is possible to use dims without end row. This way the entire column is used as input. This obviously is slower than using a fixed range, because the `wb_data()` object will contain each possible row. This is

```r
# original pivot table as reference
pt <- pivottabler::PivotTable$new()
pt$addData(pivottabler::bhmtrains)
pt$addColumnDataGroups("TrainCategory")
pt$addRowDataGroups("TOC",
                    outlineBefore = list(isEmpty = FALSE,
                                         groupStyleDeclarations = list(
                                           color = "blue")),
                    outlineTotal = list(isEmpty = FALSE,
                                        groupStyleDeclarations = list(
                                          color = "blue")))
pt$addRowDataGroups("PowerType", addTotal = FALSE)
pt$defineCalculation(calculationName = "TotalTrains",
                     summariseExpression = "n()")
```

		Express Passenger	Ordinary Passenger	Total
Arriva Trains Wales		**3079**	**830**	**3909**
	DMU	3079	830	3909
CrossCountry		**22865**	**63**	**22928**
	DMU	22133	63	22196
	HST	732		732
London Midland		**14487**	**33792**	**48279**
	DMU	5638	5591	11229
	EMU	8849	28201	37050
Virgin Trains		**8594**		**8594**
	DMU	2137		2137
	EMU	6457		6457
Total		**49025**	**34685**	**83710**

```
# use A:P
wb <- wb_workbook()$add_worksheet()$add_data(x = pivottabler::bhmtrains,
                                             na.strings = NULL)
df <- wb_data(wb, dims = "A:P")

# use TrainCategory on column and data
wb$add_pivot_table(
  df,
  rows = c("TOC", "PowerType"),
  cols = "TrainCategory",
  data = "TrainCategory",
  fun = "count"
)

if (interactive()) wb$open()
```

	A	B	C	D	E
1					
2					
3	count of TrainCategory	Column Labels			
4	Row Labels	Express Passenger	Ordinary Passenger	(blank)	Grand Total
5	**Arriva Trains Wales**	**3079**	**830**		**3909**
6	DMU	3079	830		3909
7	**CrossCountry**	**22865**	**63**		**22928**
8	DMU	22133	63		22196
9	HST	732			732
10	**London Midland**	**14487**	**33792**		**48279**
11	DMU	5638	5591		11229
12	EMU	8849	28201		37050
13	**Virgin Trains**	**8594**			**8594**
14	DMU	2137			2137
15	EMU	6457			6457
16	**(blank)**				
17	(blank)				
18	**Grand Total**	**49025**	**34685**		**83710**
19					

10.1.6 Using number formats

```
## Pivot table example 1
wb <- wb_workbook() |>
  wb_add_worksheet() |>
  wb_add_data(x = mtcars, inline_strings = FALSE)

wb$add_numfmt(dims = wb_dims(x = mtcars, cols = "disp"), numfmt = "$ #,###")

df <- wb_data(wb)

# basic pivot table with filter, rows, cols and data
wb$add_pivot_table(
  df,
  rows = "cyl", cols = "gear",
  data = c("disp", "hp"),
  fun = c("sum", "count"),
```

```
  params = list(
    numfmt = c(formatCode = "$ ###", formatCode = "#")
  ))
```

10.2 Adding slicers to pivot tables

Since `openxlsx2` release 1.1 it is possible to add slicers to pivot tables created with `wb_add_pivot_tables()`. For this to work you have to provide a name for a pivot table name you are going to add and make sure that the slicer variable is actually 'activated' in the pivot table. Adding slicers to loaded pivot tables is not possible and the creation of slicers needs to go hand in hand with a pivot table.

It is possible to apply slicer styles with `params = list(style = "SlicerStyleLight2")`

```
wb <- wb_workbook() |>
  wb_add_worksheet() |>
  wb_add_data(x = mtcars)

df <- wb_data(wb, sheet = 1)

wb$
  add_pivot_table(
    df, dims = "A3", slicer = "vs", rows = "cyl", cols = "gear", data = "disp",
    pivot_table = "mtcars"
  )$
  add_slicer(x = df, dims = "B7:D9", slicer = "vs", pivot_table = "mtcars",
             params = list(edit_as = "twoCell", style = "SlicerStyleLight2"))

if (interactive()) wb$open()
```

It is possible to tweak the number of columns in a slicer using `columnCount` and to add a `caption` and change the sorting order to `descending`.

```
wb <- wb_workbook() |>
  ### Sheet 1
  wb_add_worksheet() |>
  wb_add_data(x = mtcars)

df <- wb_data(wb, sheet = 1)
```

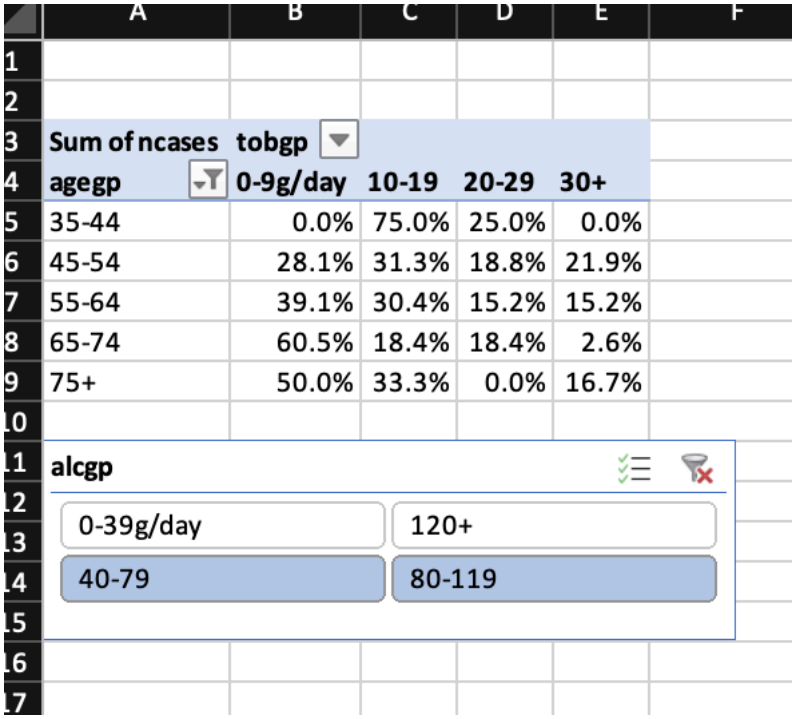
```r
varname <- c("vs", "drat")

### Sheet 2
wb$
  # first pivot
  add_pivot_table(
    df, dims = "A3", slicer = varname, rows = "cyl", cols = "gear",
    data = "disp", pivot_table = "mtcars"
  )$
  add_slicer(x = df, sheet = current_sheet(), slicer = "vs",
             pivot_table = "mtcars")$
  add_slicer(x = df, dims = "B18:D24", sheet = current_sheet(), slicer = "drat",
             pivot_table = "mtcars", params = list(columnCount = 5))$
  # second pivot
  add_pivot_table(
    df, dims = "G3", sheet = current_sheet(), slicer = varname, rows = "gear",
    cols = "carb", data = "mpg", pivot_table = "mtcars2"
  )$
  add_slicer(x = df, dims = "G12:I16", slicer = "vs", pivot_table = "mtcars2",
             params = list(sortOrder = "descending", caption = "Wow!"))

### Sheet 3
wb$
  add_pivot_table(
    df, dims = "A3", slicer = varname, rows = "gear", cols = "carb",
    data = "mpg", pivot_table = "mtcars3"
  )$
  add_slicer(x = df, dims = "A12:D16", slicer = "vs", pivot_table = "mtcars3")

if (interactive()) wb$open()
```

10.3 Choosing variable filters

Using the `choose` param argument it is possible to select subsets of the data. The code looks like this: `choose = c(agegp = 'x > "25-34"')`. The variable name as seen in the `wb_data()` object, `x` is mandatory and some expression that R understands. This can be something like `%in%`, `==`, `<`, `>`, or `!=`.

```r
wb <- wb_workbook() |>
  wb_add_worksheet("table") |>
```

```r
  wb_add_worksheet("data") |>
  wb_add_data(x = datasets::esoph)

df <- wb_data(wb)

# add a pivot table and a slicer and preselect
# a few cases and style it a bit
wb <- wb |>
  wb_add_pivot_table(
    df, dims = "A3", sheet = "table",
    rows = "agegp", cols = "tobgp", data = "ncases",
    slicer = "alcgp", pivot_table = "pt1",
    param = list(
      show_data_as = c("percentOfRow"),
      numfmt = c(formatCode = "0.0%"),
      compact = FALSE, outline = FALSE, compact_data = FALSE,
      row_grand_totals = FALSE, col_grand_totals = FALSE,
      choose = c(agegp = 'x > "25-34"')
    )
  ) |>
  wb_add_slicer(
    x = df, dims = "B14:D18",
    slicer = "alcgp", pivot_table = "pt1",
    param = list(
      columnCount = 2,
      choose = c(alcgp = 'x %in% c("40-79", "80-119")')
    )
  )

if (interactive()) wb$open()
```



Sum of ncases	tobgp			
agegp	0-9g/day	10-19	20-29	30+
35-44	0.0%	75.0%	25.0%	0.0%
45-54	28.1%	31.3%	18.8%	21.9%
55-64	39.1%	30.4%	15.2%	15.2%
65-74	60.5%	18.4%	18.4%	2.6%
75+	50.0%	33.3%	0.0%	16.7%

10.4 Final remarks

As of now it is not possible to add charts to pivot tables. This would require pivot table evaluation to construct the `wb_data()` object to use for and access to the area where the pivot table is stored on the sheet.

It is always a good idea to check that the constructed pivot table and the expected pivot table match. Either construct the pivot table manually or as shown here via `{pivottabler}` or maybe with either `{data.table}` or `{dplyr}`. It is a little tricky for `openxlsx2` to check if the pivot table works, when we have no real way to validate that it does.

11 Data Validation

Contrary to R objects like vectors or data frames, spreadsheets can contain various types of data in any type of order. A string followed by a date and a formula is not uncommon in spreadsheets. Thankfully even spreadsheets provide a tool to validate some input. This is called data validation. A tool that enhances data integrity and accuracy. By setting specific criteria and constraints for data entry, users can ensure that the data entered into cells meets predefined standards and rules. These rules apply to cells that expect data entry as well as cells that already contain data. Using data validation can help to prevent errors, maintain consistency, and streamline data even in a flexible environment such as a spreadsheet. Key aspects of data validation include creating drop-down lists for easy selection, applying date and number constraints to ensure appropriate data ranges, and using custom formulas to enforce complex validation rules. Understanding and implementing data validation can protect the user from otherwise hard to spot mistakes.

We begin with a small dataset that we want to test with data validation.

```
df <- data.frame(
  "d" = as.Date("2016-01-01") + -5:5,
  "t" = as.POSIXct("2016-01-01") + -5:5 * 10000
)
```

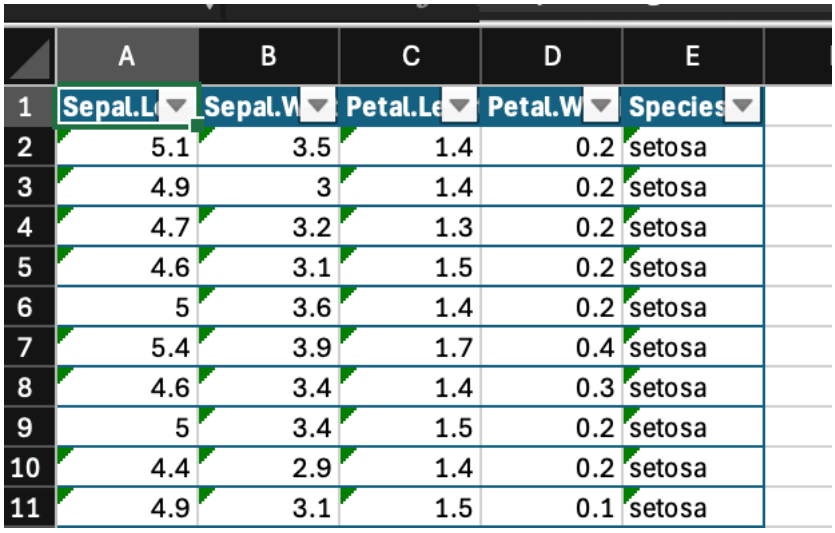
11.1 Checking numeric ranges and text lengths

In the next two code snippets we are going to check for a specific `type` of data, if a condition defined by `operator` is met for a selection or range of `values`. We construct a workbook that will be filled with four sheets of data tables and data validation for the workbook.

```
wb <- wb_workbook()$
  add_worksheet("Sheet 1")$
  add_data_table(x = iris)$
  # whole numbers are fine
  add_data_validation(dims = "A2:C151", type = "whole",
                      operator = "between", value = c(1, 9)
  )$
```

```
  # text width 7-9 is fine
  add_data_validation(dims = "E2:E151", type = "textLength",
                      operator = "between", value = c(7, 9)
  )
```

In the screenshot below, the green flag in the top left corner indicates a warning thrown by the data validation rule implemented.



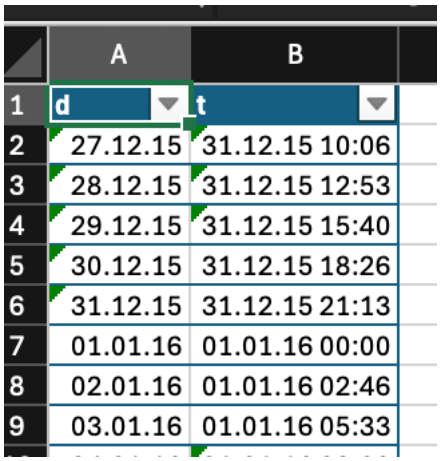
	A	B	C	D	E
1	Sepal.L	Sepal.W	Petal.Le	Petal.W	Species
2	5.1	3.5	1.4	0.2	setosa
3	4.9	3	1.4	0.2	setosa
4	4.7	3.2	1.3	0.2	setosa
5	4.6	3.1	1.5	0.2	setosa
6	5	3.6	1.4	0.2	setosa
7	5.4	3.9	1.7	0.4	setosa
8	4.6	3.4	1.4	0.3	setosa
9	5	3.4	1.5	0.2	setosa
10	4.4	2.9	1.4	0.2	setosa
11	4.9	3.1	1.5	0.1	setosa

11.2 Date and Time cell validation

In the code below we use a new data operator `greaterThanOrEqual`, all operators can be found in the documentation for `wb_add_data_validation()`. Here we add checks for a specific date and a range of timestamps that are allowed.

```
wb$
  add_worksheet("Sheet 2")$
  add_data_table(x = df)$
  # date >= 2016-01-01 is fine
  add_data_validation(dims = "A2:A12", type = "date",
                      operator = "greaterThanOrEqual",
                      value = as.Date("2016-01-01")
  )$
  # a few timestamps are fine
  add_data_validation(dims = "B2:B12", type = "time",
                      operator = "between", value = df$t[c(4, 8)]
  )
```

There are many warnings in here too.



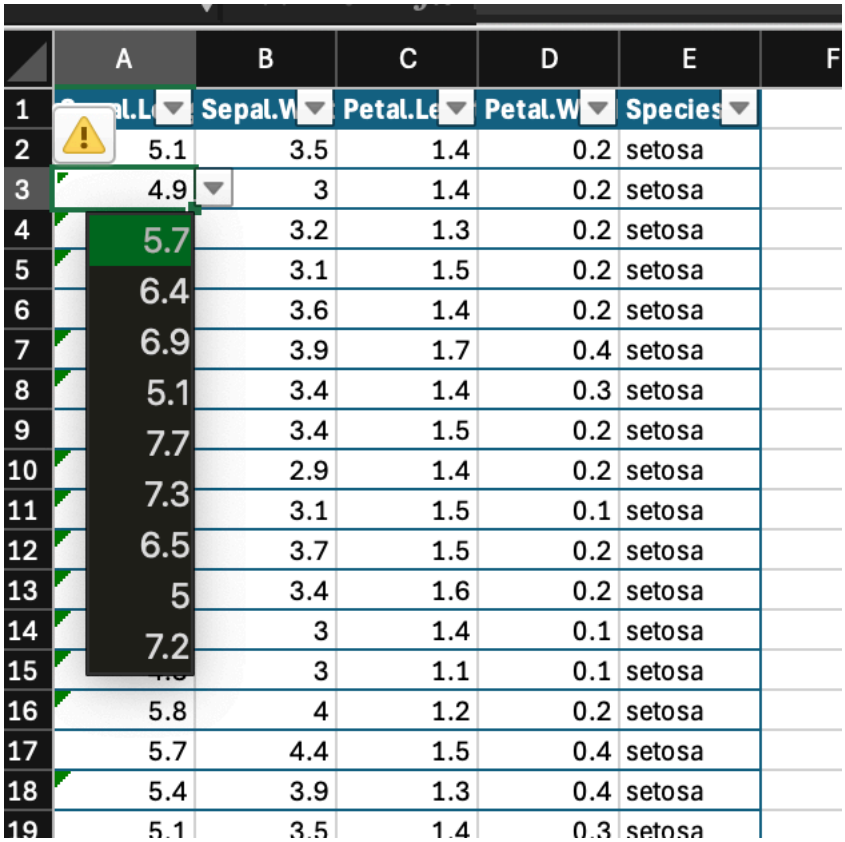
	A	B
1	d	t
2	27.12.15	31.12.15 10:06
3	28.12.15	31.12.15 12:53
4	29.12.15	31.12.15 15:40
5	30.12.15	31.12.15 18:26
6	31.12.15	31.12.15 21:13
7	01.01.16	01.01.16 00:00
8	02.01.16	01.01.16 02:46
9	03.01.16	01.01.16 05:33

11.3 validate list: validate inputs on one sheet with another

In the code below we create a sample list from the `iris` dataset on `Sheet 4` and reference this a list options for column `A` on `Sheet 3`. Our references do not have to be from the same dataset, it can be anything else. This helps, if you do not want to store the values in the `wb_add_data_validation()` step and or want to be able to quickly adjust the possible values.

```
wb$
  add_worksheet("Sheet 3")$
  add_data_table(x = iris[1:30, ])$
  add_worksheet("Sheet 4")$
  add_data(x = sample(iris$Sepal.Length, 10))$
  add_data_validation("Sheet 3", dims = "A2:A31", type = "list",
                      value = "'Sheet 4'!$A$1:$A$10")
```

Below is the drop down list and the input used to populate it.



	A	B	C	D	E	F
1	[illegible]	Sepal.W	Petal.Le	Petal.W	Species	
2	5.1	3.5	1.4	0.2	setosa	
3	4.9	3	1.4	0.2	setosa	
4		3.2	1.3	0.2	setosa	
5		3.1	1.5	0.2	setosa	
6		3.6	1.4	0.2	setosa	
7		3.9	1.7	0.4	setosa	
8		3.4	1.4	0.3	setosa	
9		3.4	1.5	0.2	setosa	
10		2.9	1.4	0.2	setosa	
11		3.1	1.5	0.1	setosa	
12		3.7	1.5	0.2	setosa	
13		3.4	1.6	0.2	setosa	
14		3	1.4	0.1	setosa	
15		3	1.1	0.1	setosa	
16	5.8	4	1.2	0.2	setosa	
17	5.7	4.4	1.5	0.4	setosa	
18	5.4	3.9	1.3	0.4	setosa	
19	5.1	3.5	1.4	0.3	setosa	

5.7
6.4
6.9
5.1
7.7
7.3
6.5
5
7.2

	A	B	C
1	5.7		
2	6.4		
3	6.9		
4	5.1		
5	7.7		
6	7.3		
7	6.5		
8	5		
9	7.2		
10	5		
11			
12			

11.4 validate list: validate inputs with values

In the code below we create drop down lists for values directly passed to `wb_add_data_validation()`. In the upper cell range options `"O1"` and `"O2"` are available, in the lower cell range `"O2"` and `"O3"`. Using values directly is helpful if there are only a few values and it is not required to provide a list of values on a spreadsheet.

```
wb <- wb_workbook()$
  add_worksheet()$add_data(x = iris[1:30, ])$
  add_worksheet()$add_data(sheet = 2, x = sample(iris$Sepal.Length, 10))$
  add_data_validation(sheet = 1, dims = "A2:A11", type = "list",
                      value = '"O1,O2"')$
  add_data_validation(sheet = 1, dims = "A12:A21", type = "list",
                      value = '"O2,O3"')
```

11.5 Examples combining data validation and formulas

11.5.1 Example 1: hyperlink to selected value

```
formula_old <- '=HYPERLINK("#Tab_1!" &amp; CELL("address",
INDEX(C1:F1, MATCH(A1, C1:F1, 0))), "Go to the selected column")'
formula_new <- '=HYPERLINK("#Tab_1!" & CELL("address",
INDEX(C1:F1, MATCH(A1, C1:F1, 0))), "Go to the selected column")'

wb <- wb_workbook()$
  add_worksheet("Tab_1", zoom = 80, gridLines = FALSE)$
  add_data(x = rbind(2016:2019), dims = "C1:F1", colNames = FALSE)$
  add_data(x = 2017, dims = "A1", colNames = FALSE)$
  add_data_validation(dims = "A1", type = "list",
                      value = '"2016,2017,2018,2019"')$
  add_formula(dims = "B1", x = formula_old)$
  add_formula(dims = "B2", x = formula_new)
```

11.5.2 Example 2: create hyperlink to github

```
wb <- wb_workbook()$
  add_worksheet("Tab_1", zoom = 80, gridLines = FALSE)$
```

```
    add_data(dims = "C1:F1", x = rbind(2016:2019), colNames = FALSE)$
    add_data(x = 2017, startCol = 1, startRow = 1, colNames = FALSE)$
    add_data_validation(dims = "A1", type = "list",
                        value = '"2016,2017,2018,2019"')$
    add_formula(dims = "B1", x = '=HYPERLINK("#Tab_1!" &
                CELL("address", INDEX(C1:F1, MATCH(A1, C1:F1, 0))),
                "Go to the selected column")')$
    add_formula(dims = "B2", x = '=IF(2017 = VALUE(A1),
                HYPERLINK("github.com","github.com"), A1)')
```

12 Form control

This chapter delves into the `wb_add_form_control()` function, a versatile tool for embedding interactive elements directly into your workbook. It will show how to seamlessly integrate various form controls, including checkboxes[1], radio buttons, and dropdowns, to enhance user interaction and data input within your spreadsheets.

There are a few function's parameter, available, to set or retrieve the form control value. This allows the creation of dynamic and user-friendly workbooks that go beyond static data display, enabling more engaging and efficient data management.

12.1 What Are Form Controls?

Form controls in a spreadsheet environment are interactive graphical objects that allow users to input data, make selections, or trigger actions within a worksheet. Unlike directly typing into cells, form controls provide a more structured and often more intuitive way for users to interact with a workbook. They are commonly used to create interactive dashboards, data entry forms, and simple applications within spreadsheet software.

Common types of form controls include:

- Checkboxes: Used for binary choices (e.g., "Yes/No," "True/False," or to select multiple options from a list).
- Radio Buttons (Option Buttons): Used when the user must select only one option from a mutually exclusive set of choices.
- Dropdown Lists (Combo Boxes): Allow users to select an item from a predefined list, saving space on the worksheet and ensuring data consistency.

There are other form controls that are not yet implemented in `openxlsx2`, mostly due to the lack of interest. The entire `wb_add_form_control()` function dates back to a user request.

[1] In 2024 a new checkbox was added in Excel. This makes use of the feature property bag and works slightly different compared to the form control checkbox. Basically it is a logical value (0 or 1) that takes an overlay to display a checked or unchecked box. They have the benefit that they are rather lightweight and stick to the cell like any other embedded cell content, whereas the form control elements float over the spreadsheet. There is no API for this yet, but you can use the new checkboxes like this:

```
library(openxlsx2)

wb <- wb_workbook()$add_worksheet()

# add feature property bag
wb$featurePropertyBag <- '<FeaturePropertyBags xmlns=
"http://schemas.microsoft.com/office/spreadsheetml/2022/featurepropertybag">
 <bag type="Checkbox" />
 <bag type="XFControls">
  <bagId k="CellControl">0</bagId>
 </bag>
 <bag type="XFComplement">
  <bagId k="XFControls">1</bagId>
 </bag>
 <bag type="XFComplements" extRef="XFComplementsMapperExtRef">
  <a k="MappedFeaturePropertyBags">
   <bagId>2</bagId>
  </a>
 </bag>
</FeaturePropertyBags>'
wb$append("workbook.xml.rels",
  '<Relationship Id="rId5" Type=
  "http://schemas.microsoft.com/office/2022/11/relationships/FeaturePropertyBag"
  Target="featurePropertyBag/featurePropertyBag.xml"/>')
wb$append("Content_Types",
      '<Override PartName="/xl/featurePropertyBag/featurePropertyBag.xml"
      ContentType="application/vnd.ms-excel.featurepropertybag+xml"/>')

# add style
extLst <- '<extLst>
 <ext xmlns:xfpb=
 "http://schemas.microsoft.com/office/spreadsheetml/2022/featurepropertybag"
 uri="{C7286773-470A-42A8-94C5-96B5CB345126}">
  <xfpb:xfComplement i="0" />
 </ext>
</extLst>'
sty <- create_cell_style(ext_lst = extLst)
wb$styles_mgr$add(sty, "checkbox_sty")
xf_sty <- wb$styles_mgr$get_xf_id("checkbox_sty")

# add data and assign style
wb$add_data(x = matrix(sample(c(TRUE, FALSE), 5, TRUE), 5, 2))
wb$set_cell_style(dims = "A2:A6", style = xf_sty)

# wb$open()
```

12.2 Pros and Cons of Using Form Controls

12.2.1 Pros:

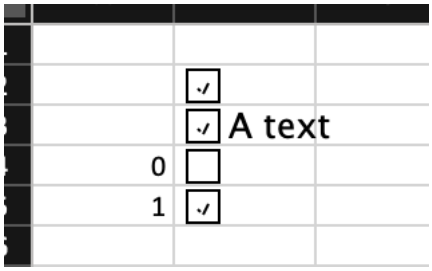
- Improved User Experience: Form controls make spreadsheets more intuitive and user-friendly, guiding users through data entry and selection processes. This can reduce errors and make complex workbooks more accessible.
- Data Validation and Consistency: By providing predefined options (e.g., dropdowns) or structured inputs (e.g., checkboxes), form controls help enforce data validation, ensuring that users enter data in a consistent and correct format.
- Reduced Manual Input Errors: By limiting choices or providing visual cues, form controls can minimize the likelihood of typos or incorrect entries that often occur with manual cell input.

12.2.2 Cons:

- Complexity in Setup: Implementing and linking form controls, especially with more advanced functionalities like dynamic lists or macro triggers, can be more complex and time-consuming than simple cell-based data entry.
- Accessibility Concerns: While generally improving usability, poorly implemented form controls might pose challenges for users with certain disabilities or those relying on screen readers, if not designed with accessibility in mind.
- Version Compatibility: The appearance and behavior of form controls can sometimes vary slightly across different versions of spreadsheet software, which might lead to minor display or functionality issues if shared across diverse user environments.

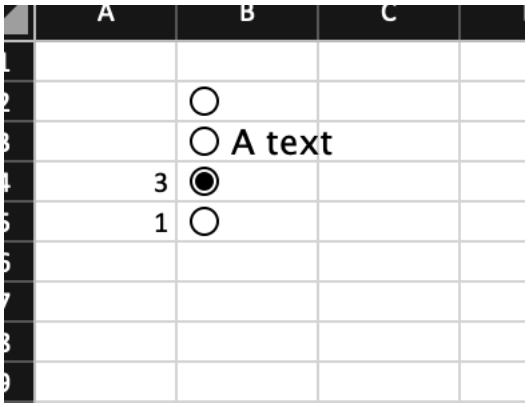
12.3 Checkboxes

```
wb <- wb_workbook()$
  add_worksheet()$
  add_form_control(dims = "B2")$
  add_form_control(dims = "B3", text = "A text")$
  add_data(dims = "A4", x = 0, colNames = FALSE)$
  add_form_control(dims = "B4", link = "A4")$
  add_data(dims = "A5", x = TRUE, colNames = FALSE)$
  add_form_control(dims = "B5", range = "'Sheet 1'!A5", link = "B5")
```



12.4 Radio Buttons

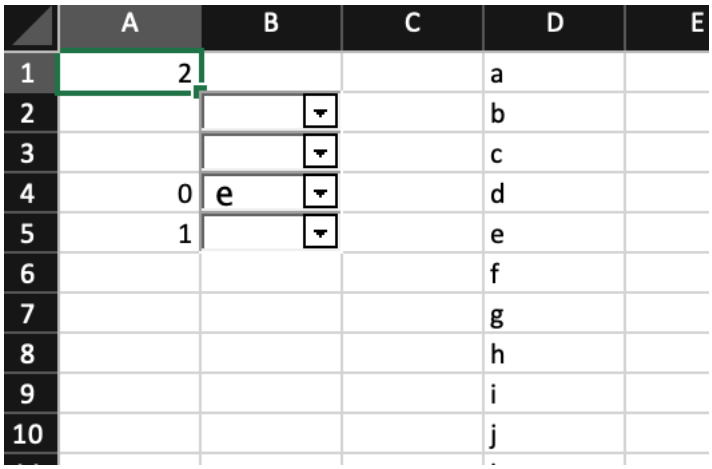
```
wb$
add_worksheet()$
add_form_control(dims = "B2", type = "Radio")$
add_form_control(dims = "B3", type = "Radio", text = "A text")$
add_data(dims = "A4", x = 0, colNames = FALSE)$
add_form_control(dims = "B4", type = "Radio", link = "A4")$
add_data(dims = "A5", x = 1, colNames = FALSE)$
add_form_control(dims = "B5", type = "Radio")
```



12.5 Dropdown lists

```
wb$
add_worksheet()$
add_form_control(dims = "B2", type = "Drop")$
add_form_control(dims = "B3", type = "Drop", text = "A text")$
```

```
    add_data(dims = "A4", x = 0, colNames = FALSE)$
    add_form_control(dims = "B4", type = "Drop", link = "A1", range = "D4:D15")$
    add_data(dims = "A5", x = 1, colNames = FALSE)$
    add_form_control(dims = "B5", type = "Drop", link = "'Sheet 3'!D1:D26",
                     range = "A1")$
    add_data(dims = "D1", x = letters)
```



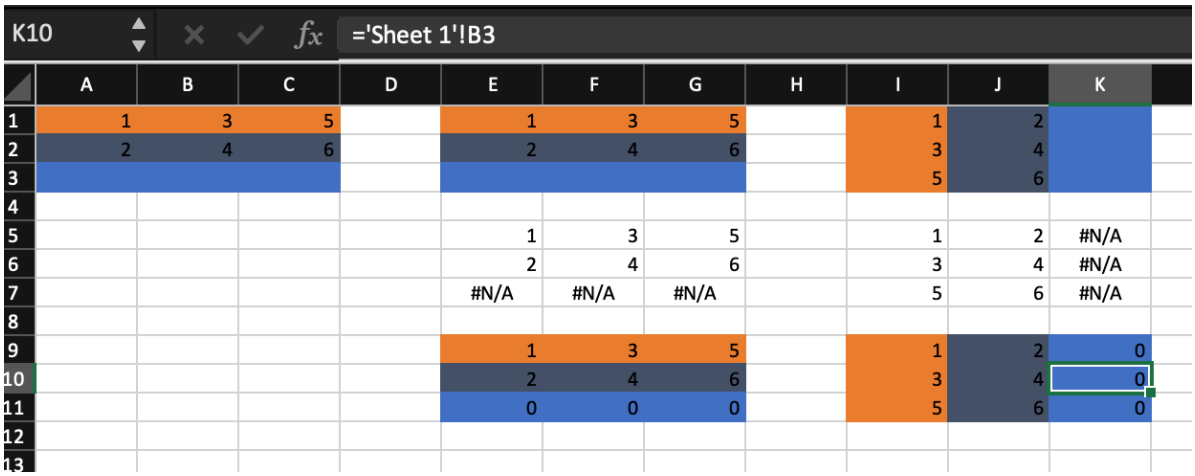
13 Cloning and copying

When using `openxlsx2` there are multiple ways to modify the workbook including various ways to copy and clone sheets, cells and styles.

13.1 Copying cells

It is possible to copy cells into different regions of the worksheet using `wb_copy_cells()`. There are three ways to copy cells: (1) as is, including styles, (2) as value replacing all formulas and (3) as reference to the cell origin. This can be seen in the following image, the transposed cell contains a formula pointing to the original cell.

```
mm <- matrix(1:6, 2)
wb <- wb_workbook()$add_worksheet()$
  add_data(x = mm, col_names = FALSE)$
  add_fill(dims = "A1:C1", color = wb_color(theme = 5))$
  add_fill(dims = "A2:C2", color = wb_color(theme = 3))$
  add_fill(dims = "A3:C3", color = wb_color(theme = 4))

dat <- wb_data(wb, dims = "A1:C3", col_names = FALSE)

wb$copy_cells(dims = "E1", data = dat)
wb$copy_cells(dims = "E5", data = dat, as_value = TRUE)
wb$copy_cells(dims = "E9", data = dat, as_ref   = TRUE)

wb$copy_cells(dims = "I1", data = dat, transpose = TRUE)
wb$copy_cells(dims = "I5", data = dat, transpose = TRUE, as_value = TRUE)
wb$copy_cells(dims = "I9", data = dat, transpose = TRUE, as_ref   = TRUE)

if (interactive()) wb$open()
```



	A	B	C	D	E	F	G	H	I	J	K
1	1	3	5		1	3	5		1	2	
2	2	4	6		2	4	6		3	4	
3									5	6	
4											
5					1	3	5		1	2	#N/A
6					2	4	6		3	4	#N/A
7					#N/A	#N/A	#N/A		5	6	#N/A
8											
9					1	3	5		1	2	0
10					2	4	6		3	4	0
11					0	0	0		5	6	0
12											
13											

13.2 Cloning worksheets

Sometimes it is not enough to copy a cell range, sometimes you need to copy entire worksheets. This can be done using `wb_clone_worksheet()`. You can clone a worksheet in a workbook, but also across workbooks, though the first option is simpler and might provide more features. Cloning worksheets around that contain (pivot) tables and slicers for instance might be impossible and some other features of the workbook might also not be present. In addition it is not guaranteed that a clone will look identical to the original worksheet if relative theme colors are used. As always, be careful if you use this feature and test that it works, before you start cloning production worksheets.

```
fl <- system.file("extdata", "oxlsx2_sheet.xlsx", package = "openxlsx2")
wb_from <- wb_load(fl)

# clone worksheet from SUM to NOT_SUM
wb_from$clone_worksheet(old = "SUM", new = "NOT_SUM")

# clone worksheet across workbooks including styles and shared strings
wb$clone_worksheet(old = "SUM", new = "SUM", from = wb_from)
```

14 Comments and Working with Shapes in openxlsx2

14.1 Adding Comments

Comments in Excel are useful for annotating cells with additional information. Using `openxlsx2`, we can create comments and we can even create, reply to, and close threads programmatically. Threads are a feature introduced in MS365 and replace the "comment", while legacy comments are now called "note". We use comment and thread (their names in the XML code), but each their own.

14.1.1 Creating a Comment

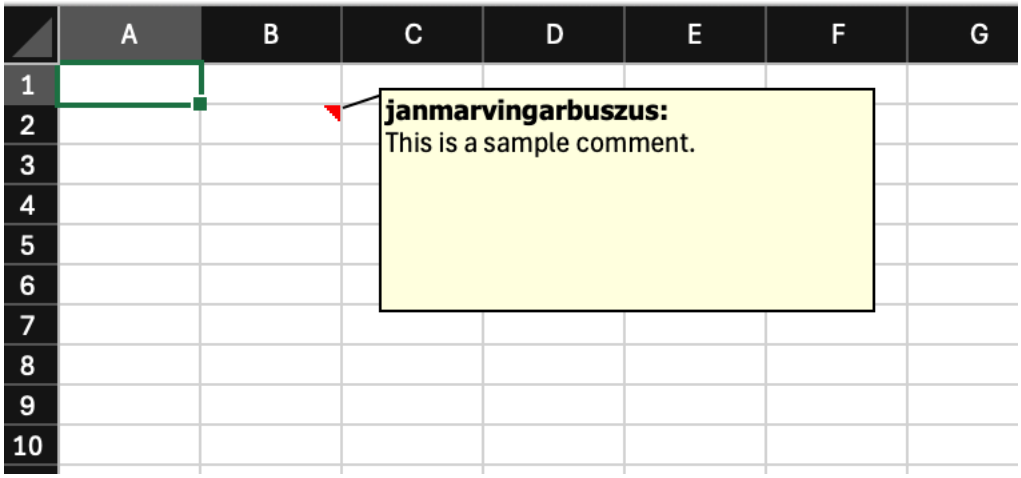
```
wb <- wb_workbook()$
  add_worksheet()

# Add a comment to cell A1
wb$add_comment(
  dims = "B2",
  comment = "This is a sample comment."
)
```

It is possible to style the comment, the manual page provides a few examples of this. `openxlsx2` provides additional niche features such as background images for comments.[1] For this we are going to remove this previous comment.

```
# eh what was it again?
wb$get_comment(dims = "B2")
#>   ref author                                comment cmmt_id
#> 1  B2 runner runner: \n This is a sample comment.       1
```

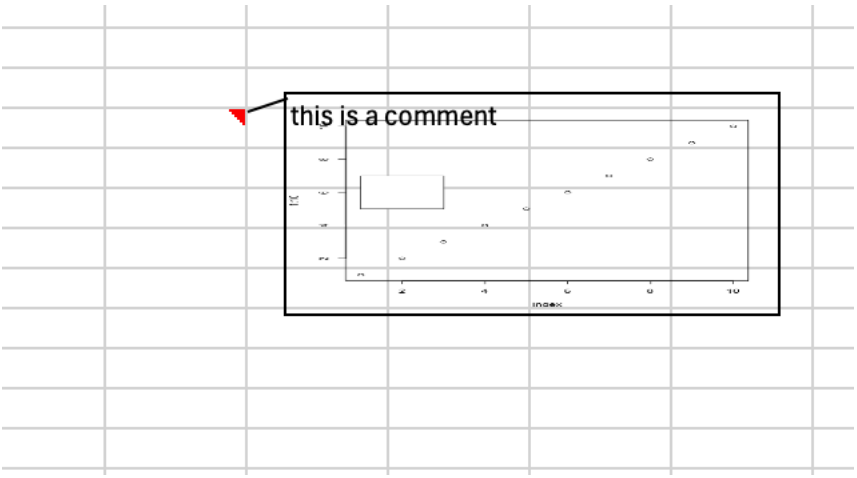
[1] Since there is no dialog option for this on MS365 for Mac, I was not even sure what the user requesting this feature, was even talking about.

```
# okay, sample comment. can be removed
wb$remove_comment(dims = "B2")
```



14.1.2 Comments with background images

```
tmp <- tempfile(fileext = ".png")
png(file = tmp, bg = "transparent")
plot(1:10)
rect(1, 5, 3, 7, col = "white")
dev.off()
#> pdf
#>   2

c1 <- wb_comment(text = "this is a comment", author = "", visible = TRUE)
wb$add_comment(dims = "B12", comment = c1, file = tmp)
```



14.2 Working with Threads

In its foundation a comment is just some text in a quadratic shape. Usually it contains some author information, but this is entirely optional, it could also be just some fictional text or the authors name can be removed entirely. It is also quite complex to reply to a comment. For this, threads were invented. A thread is something similar to a chat or mail chain. It is created chronologically and it has a person attached to it. It is possible to answer to a thread and to close it. As in, the question was solved, but it is left for everyone to see.

In spreadsheet software that does not support threads, a comment is shown with the information content of the thread and a hint that the comment should not be altered.

14.2.1 Persons, create one or become one

To create a comment, you need to be a person assigned with the worksheet. Persons could be corporate accounts with specific Ids (you need to import a worksheet with such an id). Afterwards you can get the persons attached to the worksheet with `wb_get_person()`. While there is an id attached to the person, it is not different compared to an email username. It can be spoofed, and basically, if you select your id from the list of available names, please consider if your company finds it as hilarious as you do, if you decide to create, open or answer threads as someone else. For now, we will create two persons.

```
wb <- wb_workbook()$add_worksheet()
# Add a person to the workbook.
wb$add_person(name = "somebody")
wb$add_person(name = "you")
```

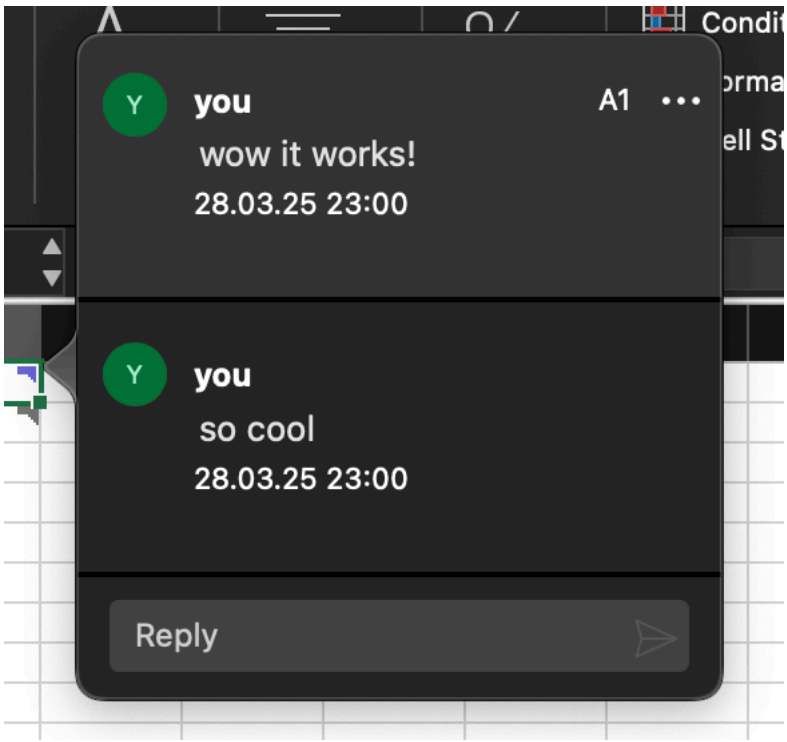
Now we want to create a thread as `"you"`. The id `pid` itself is rather uninteresting, it is a guid, similar to many others used in `openxlsx2`.

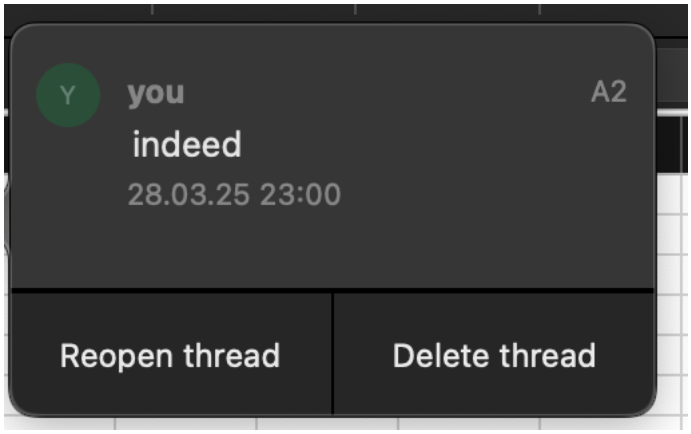
```
pid <- wb$get_person(name = "you")$id
```

14.2.2 Creating a Thread

And that's it. Now we can create a thread as you.

```
wb$add_thread(dims = "A1", comment = "wow it works!", person_id = pid)
wb$add_thread(dims = "A2", comment = "indeed", person_id = pid, resolve = TRUE)
wb$add_thread(dims = "A1", comment = "so cool", person_id = pid, reply = TRUE)
```





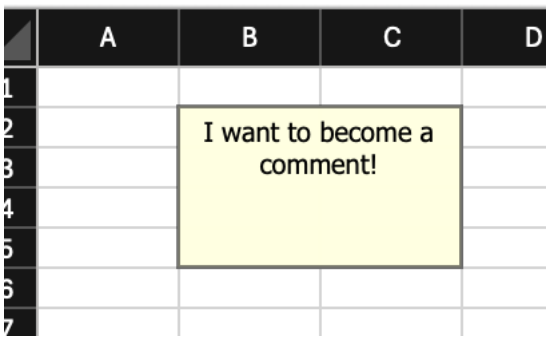
14.3 Working with Shape Objects

Besides comments and notes, `openxlsx2` allows for the addition of shapes, such as rectangles, circles, and other graphical elements, to a worksheet.

14.3.1 Adding a Rectangle Shape

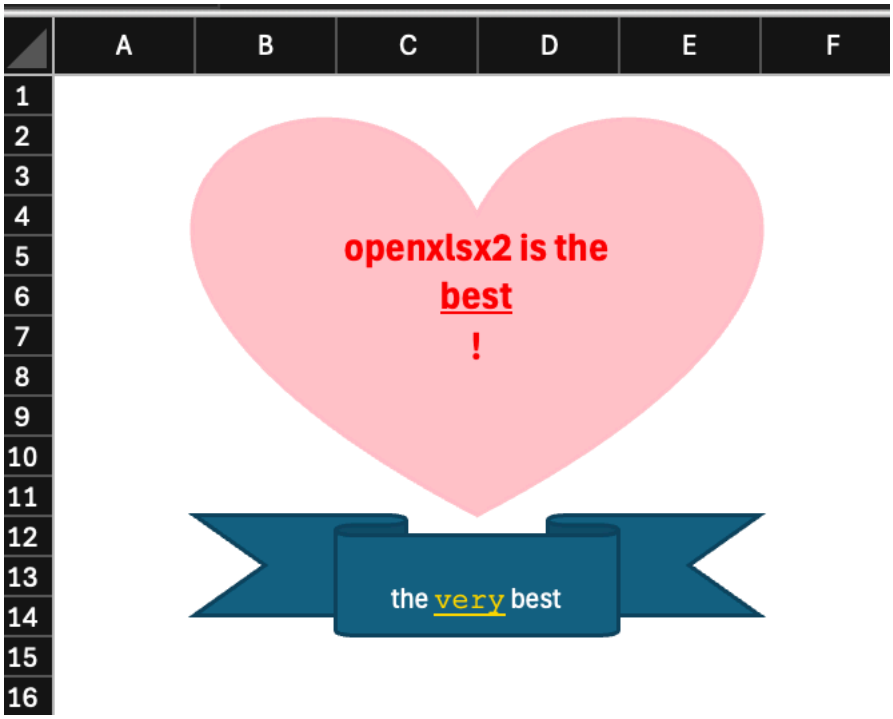
If you are wondering why the section about comment and threads is in the same section as shapes, after all comments are something like a rectangular shape with a text.

```
rect <- create_shape(
  shape = "rect", text_align = "center",
  text = fmt_txt("I want to become a comment!", font = "Tahoma", size = "10",
                 color = wb_color("black"), family = 2),
  fill_colour = wb_color(hex = "ffffe1"),
  line_color = wb_color("black"),
  line_transparency = 50
)

wb <- wb_workbook()$add_worksheet()$
   add_drawing(dims = "B2:C5", xml = rect)
```



As seen it is possible to assign `fmt_txt()` strings to shape objects. And given some trial and error it is even possible to create complex images with `create_shape()` objects.

```
## heart
txt <- fmt_txt("openxlsx2 is the \n", bold = TRUE, size = 15) +
  fmt_txt("best", underline = TRUE, bold = TRUE, size = 15) +
  fmt_txt("\n!", bold = TRUE, size = 15)

heart <- create_shape(
  shape = "heart", text = txt, text_align = "center",
  fill_colour = wb_color("pink"), text_colour = wb_color("red"))

## ribbon
txt <- fmt_txt("\nthe ") +
  fmt_txt("very", underline = TRUE, font = "Courier",
          color = wb_color("gold")) +
  fmt_txt(" best")

ribbon <- create_shape(shape = "ribbon", text = txt, text_align = "center")

wb <- wb_workbook()$add_worksheet(grid_lines = FALSE)$
  add_drawing(dims = "B2:E11", xml = heart)$
  add_drawing(dims = "B12:E14", xml = ribbon)$
  add_worksheet()$add_drawing(dims = "B2:E5",
                              xml = create_shape(
                                "rect", text = txt,
                                fill_color = wb_color(theme = 5),
                                fill_transparency =  50))
```


A
B
C
D
E
F
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
openxlsx2 is the
best
!
the very best

15 Upgrade from `openxlsx`

15.1 Basic read and write functions

Welcome to the `openxlsx2` update vignette. In this vignette we will take some common code examples from `openxlsx` and show you how similar results can be replicated in `openxlsx2`. Thank you for taking a look, and let's get started. While previous `openxlsx` functions used the `.` in function calls, as well as camelCase, we have tried to switch to snake_case (this is still a work in progress, there may still be function arguments that use camelCase).

15.1.1 Read xlsx or xlsm files

The basic read function changed from `read.xlsx` to `read_xlsx`. Using a default xlsx file included in the package:

```
file <- system.file("extdata", "openxlsx2_example.xlsx", package = "openxlsx2")
```

The old syntax looked like this:

```
# read in openxlsx
openxlsx::read.xlsx(xlsxFile = file)
```

This has changed to this:

```
# read in openxlsx2
openxlsx2::read_xlsx(file = file)
#>     Var1 Var2 <NA>  Var3  Var4       Var5         Var6    Var7     Var8
#> 3   TRUE    1   NA     1     a 2023-05-29 3209324 This #DIV/0! 01:27:15
#> 4   TRUE   NA   NA #NUM!     b 2023-05-23         <NA>       0 14:02:57
#> 5   TRUE    2   NA  1.34     c 2023-02-01         <NA> #VALUE! 23:01:02
#> 6  FALSE    2   NA  <NA> #NUM!       <NA>         <NA>       2 17:24:53
#> 7  FALSE    3   NA  1.56     e       <NA>         <NA>    <NA>     <NA>
#> 8  FALSE    1   NA   1.7     f 2023-03-02         <NA>     2.7 08:45:58
#> 9     NA   NA   NA  <NA>  <NA>       <NA>         <NA>    <NA>     <NA>
```

```
#> 10 FALSE    2   NA    23     h 2023-12-24         <NA>      25     <NA>
#> 11 FALSE    3   NA  67.3     i 2023-12-25         <NA>       3     <NA>
#> 12    NA    1   NA   123  <NA> 2023-07-31         <NA>     122     <NA>
```

As you can see, we return the spreadsheet return codes (e.g., `#NUM`) in openxlsx2. Another thing to see above, we return the cell row as rowname for the data frame returned. `openxlsx2` should return a data frame of the selected size, even if it empty. If you preferred `openxlsx::readWorkbook()` this has become `wb_read()`. All of these are wrappers for the newly introduced function `wb_to_df()` which provides the most options. `read_xlsx()` and `wb_read()` were created for backward comparability.

15.2 Write xlsx files

Basic writing in `openxlsx2` behaves identical to `openxlsx`. Though be aware that `overwrite` is an optional parameter in `openxlsx2` and just like in other functions like `base::write.csv()` if you write onto an existing file name, this file will be replaced.

Setting the output to some temporary xlsx file

```
output <- temp_xlsx()
```

The previous write function looks like this:

```
# write in openxlsx
openxlsx::write.xlsx(iris, file = output, colNames = TRUE)
```

The new function looks quite similar:

```
# write in openxlsx2
openxlsx2::write_xlsx(iris, file = output, col_names = TRUE)
```

15.3 Basic workbook functions

Workbook functions have been renamed to begin with `wb_` there are plenty of these in the package, therefore looking at the man pages seems to be the fastest way. Yet, it all begins with loading the workbook.

15.3.1 Loading a workbook

A major feature in `openxlsx` are workbooks. Obviously they remain a central piece in `openxlsx2`. Previous you would load them with:

```
wb <- openxlsx::loadWorkbook(file = file)
```

In `openxlsx2` loading was changed to:

```
wb <- wb_load(file = file)
```

There are plenty of functions to interact with workbooks and we will not describe every single one here. A detailed list can be found over at our references

15.3.2 Styles

One of the biggest user facing change was the removal of the `stylesObject`. In the following section we use code from `openxlsx::addStyle()`

```
# openxlsx
## Create a new workbook
wb <- createWorkbook(creator = "My name here")
addWorksheet(wb, "Expenditure", gridLines = FALSE)
writeData(wb, sheet = 1, USPersonalExpenditure, rowNames = TRUE)

## style for body
bodyStyle <- createStyle(border = "TopBottom", borderColor = "#4F81BD")
addStyle(wb, sheet = 1, bodyStyle, rows = 2:6, cols = 1:6, gridExpand = TRUE)

## set column width for row names column
setColWidths(wb, 1, cols = 1, widths = 21)
```

In `openxlsx2` the same code looks something like this:

```
# openxlsx2 chained
border_color <- wb_color(hex = "4F81BD")
wb <- wb_workbook(creator = "My name here")$
  add_worksheet("Expenditure", grid_lines = FALSE)$
  add_data(x = USPersonalExpenditure, row_names = TRUE)$
  add_border( # add the outer and inner border
```

```
    dims = "A1:F6",
    top_border = "thin", top_color = border_color,
    bottom_border = "thin", bottom_color = border_color,
    inner_hgrid = "thin", inner_hcolor = border_color,
    left_border = "", right_border = ""
  )$
  set_col_widths( # set column width
    cols = 1:6,
    widths = c(20, rep(10, 5))
  )$ # remove the value in A1
  add_data(dims = "A1", x = "")
```

The code above uses chaining. If you prefer piping, we provide the chained functions with the prefix `wb_` so `wb_add_worksheet()`, `wb_add_data()`, `wb_add_border()` and `wb_set_col_widths()` would be the functions to use with pipes `|>` or `|>`.

With pipes the code from above becomes

```
# openxlsx2 with pipes
border_color <- wb_color(hex = "4F81BD")
wb <- wb_workbook(creator = "My name here") |>
  wb_add_worksheet(sheet = "Expenditure", grid_lines = FALSE) |>
  wb_add_data(x = USPersonalExpenditure, row_names = TRUE) |>
  wb_add_border( # add the outer and inner border
    dims = "A1:F6",
    top_border = "thin", top_color = border_color,
    bottom_border = "thin", bottom_color = border_color,
    inner_hgrid = "thin", inner_hcolor = border_color,
    left_border = "", right_border = ""
  ) |>
  wb_set_col_widths( # set column width
    cols = 1:6,
    widths = c(20, rep(10, 5))
  ) |> # remove the value in A1
  wb_add_data(dims = "A1", x = "")
```

Be aware that chains modify an object in place and pipes do not.

```
# openxlsx2
wbp <- wb_workbook() |> wb_add_worksheet()
wbc <- wb_workbook()$add_worksheet()
```

```
# need to assign wbp
wbp <- wbp |> wb_add_data(x = iris)
wbc$add_data(x = iris)
```

You can re-use styles with `wb_get_cell_style()` and `wb_set_cell_style()`. Abandoning `stylesObject` in `openxlsx2` has the huge benefit that we can import and export a spreadsheet without changing any cell style. It is still possible to modify a cell style with `wb_add_border()`, `wb_add_fill()`, `wb_add_font()` and `wb_add_numfmt()`.

Additional examples regarding styles can be found in the styles vignette.

15.3.3 Conditional formatting

See `vignette("conditional-formatting")` for extended examples on formatting.

Here is a minimal example:

```
# openxlsx2 with chains
wb <- wb_workbook()$
  add_worksheet("a")$
  add_data(x = 1:4, col_names = FALSE)$
  add_conditional_formatting(dims = "A1:A4", rule = ">2")

# openxlsx2 with pipes
wb <- wb_workbook() |>
  wb_add_worksheet("a") |>
  wb_add_data(x = 1:4, col_names = FALSE) |>
  wb_add_conditional_formatting(dims = "A1:A4", rule = ">2")
```

15.3.4 Data validation

Similarly, data validation has been updated and improved. This `openxlsx` code for data validation

```
# openxlsx
wb <- createWorkbook()
addWorksheet(wb, "Sheet 1")
writeDataTable(wb, 1, x = iris[1:30, ])
dataValidation(wb, 1,
  col = 1:3, rows = 2:31, type = "whole",
  operator = "between", value = c(1, 9)
)
```

looks in `openxlsx2` something like this:

```
# openxlsx2 with chains
wb <- wb_workbook()$
  add_worksheet("Sheet 1")$
  add_data_table(1, x = iris[1:30, ])$
  add_data_validation(1,
    dims = wb_dims(rows = 2:31, cols = 1:3),
    # alternatively, dims can also be "A2:C31" if you know the span in your
    # spreadsheet
    type = "whole",
    operator = "between",
    value = c(1, 9)
  )

# openxlsx2 with pipes
wb <- wb_workbook() |>
  wb_add_worksheet("Sheet 1") |>
  wb_add_data_table(1, x = iris[1:30, ]) |>
  wb_add_data_validation(
    sheet = 1,
    dims = "A2:C31", # alternatively, dims = wb_dims(rows = 2:31, cols = 1:3)
    type = "whole",
    operator = "between",
    value = c(1, 9)
  )
```

15.3.5 Saving

Saving has been switched from `saveWorbook()` to `wb_save()` and opening a workbook has been switched from `openXL()` to `wb_open()`.

16 Extending `openxlsx2`

```
library(openxlsx2)
```

16.1 `msoc` - Encrypting / Decrypting workbooks

You might want to look at **msoc** (Garbuszus 2023) for openxml file level encryption/decryption.

```
library(msoc)

xlsx <- temp_xlsx()

# let us write some worksheet
wb_workbook()$add_worksheet()$add_data(x = mtcars)$save(xlsx)

# now we can encrypt it
encrypt(xlsx, xlsx, pass = "msoc")
#> [1] "/tmp/Rtmp8qj45c/temp_xlsx_2455487753c7.xlsx"

# the file is encrypted, we can not read it
try(wb <- wb_load(xlsx))
#> Error : Unable to open and load file:  /tmp/Rtmp8qj45c/temp_xlsx_2455487753c7.xlsx

# we have to decrypt it first
decrypt(xlsx, xlsx, pass = "msoc")
#> [1] "/tmp/Rtmp8qj45c/temp_xlsx_2455487753c7.xlsx"

# now we can load it again
wb_load(xlsx)$to_df() |> head()
#>    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> 2 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> 3 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> 4 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
#> 5 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> 6 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
#> 7 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

16.2 `flexlsx` - Exporting `flextable` to workbooks

Using `flexlsx` (Heidler 2024) you can extend `openxlsx2` to write `flextable` objects (Gohel and Skintzos 2023) to spreadsheets. Various styling options are supported. A detailed description how to create flextables is given in the flextable book (a link is in the bibliography).

```
library(flexlsx)

wb <- wb_workbook()$add_worksheet("mtcars", grid_lines = FALSE)

# Create a flextable and an openxlsx2 workbook
ft <- flextable::as_flextable(table(mtcars[2:5, 1:2]))
ft

# add the flextable ft to the workbook, sheet "mtcars"
# offset the table to cell 'C2'
wb <- flexlsx::wb_add_flextable(wb, "mtcars", ft, dims = "C2")

if (interactive()) wb$open()
```

		cyl			
mpg		4	6	8	Total
18.7	Count	0 (0.0%)	0 (0.0%)	1 (25.0%)	1 (25.0%)
	Mar. pct [1]	0.0% ; 0.0%	0.0% ; 0.0%	100.0% ; 100.0%	
21	Count	0 (0.0%)	1 (25.0%)	0 (0.0%)	1 (25.0%)
	Mar. pct	0.0% ; 0.0%	50.0% ; 100.0%	0.0% ; 0.0%	
21.4	Count	0 (0.0%)	1 (25.0%)	0 (0.0%)	1 (25.0%)
	Mar. pct	0.0% ; 0.0%	50.0% ; 100.0%	0.0% ; 0.0%	
22.8	Count	1 (25.0%)	0 (0.0%)	0 (0.0%)	1 (25.0%)
	Mar. pct	100.0% ; 100.0%	0.0% ; 0.0%	0.0% ; 0.0%	
Total	Count	1 (25.0%)	2 (50.0%)	1 (25.0%)	4 (100.0%)

[1] Columns and rows percentages

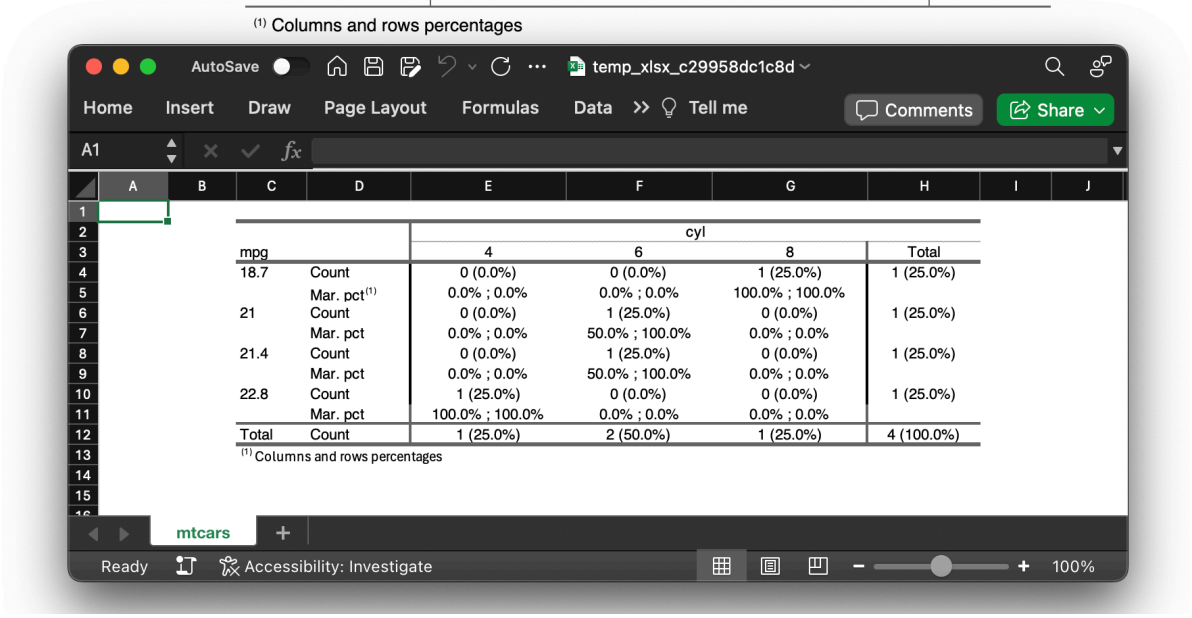


Figure 16.1: The flextable written as xlsx file and as image

16.3 `openxlsx2Extras` - Extending `openxlsx2`

Early in development, `openxlsx2Extras` (Pousson 2024) allows extending various functions for user convenience or for features, that are more focused on working along `openxlsx2` and therefore are not necessary a requirement for the package itself.

One example (more can be found on the project github and pkgdown pages) is the following.

```
library(openxlsx2)
library(openxlsx2Extras)
```

```
wb_new_workbook(
  title = "Workbook created with wb_new_workbook",
  sheet_names = c("First sheet", "Second sheet"),
  tab_color = c(wb_color("orange"), wb_color("yellow"))
)
#> A Workbook object.
#>
#> Worksheets:
#>  Sheets: First sheet, Second sheet
#>  Write order: 1, 2
```

16.4 `ovbars` - Reading the `vbaProject.bin`

Another niche package is `ovbars` (Garbuszus 2024). This package allows reading the binary blob that contains macros in `xlsm` and potentially `xlsb` files. The package allows extracting the VBA code.

```
url <- "https://github.com/JanMarvin/openxlsx-data/raw/refs/heads/main"
fl <- file.path(url, "gh_issue_416.xlsm")
wb <- openxlsx2::wb_load(fl)
vba <- wb$vbaProject

code <- ovbars::ovbar_out(name = vba)
message(code["Sheet1"])
#> Attribute VB_Name = "Sheet1"
#> Attribute VB_Base = "0{00020820-0000-0000-C000-000000000046}"
#> Attribute VB_GlobalNameSpace = False
#> Attribute VB_Creatable = False
#> Attribute VB_PredeclaredId = True
#> Attribute VB_Exposed = True
#> Attribute VB_TemplateDerived = False
#> Attribute VB_Customizable = True
#> Private Sub Worksheet_SelectionChange(ByVal Target As Range)
#>     #donothing
#> End Sub
```

References

Allen, Michael. 2023. *Readxlsb: Read 'Excel' Binary (.xlsb) Workbooks*. https://CRAN.R-project.org/package=readxlsb.

Barbone, Jordan Mark, and Jan Marvin Garbuszus. 2024. *Openxlsx2: Read, Write and Edit 'Xlsx' Files*. https://janmarvin.github.io/openxlsx2/.

Chang, Winston. 2021. *R6: Encapsulated Classes with Reference Semantics*. https://CRAN.R-project.org/package=R6.

Dotta, Damien, and Julien Blasco. 2024. *Tablexlsx: Export Data Frames to Excel Workbook*. https://doi.org/10.32614/CRAN.package.tablexlsx.

Dragulescu, Adrian, and Cole Arendt. 2023. *Xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*. https://CRAN.R-project.org/package=xlsx.

Dreano, Denis. 2023. *Knitxl: Generates a Spreadsheet Report from an 'Rmarkdown' File*. https://doi.org/10.32614/CRAN.package.knitxl.

ECMA-376-1. 2016. *Office Open XML File Formats — Fundamentals and Markup Language Reference*.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. https://doi.org/10.18637/jss.v040.i08.

Garbuszus, Jan Marvin. 2023. *Msoc: Encrypt and Decrypt of Office Open Xml Files*. https://janmarvin.github.io/msoc/.

———. 2024. *Ovbars: Accesses the 'Ovba' Rust Library to Extract Vba Binary Files*. https://janmarvin.github.io/ovbars/.

Garmonsway, Duncan. 2022. *Tidyxl: Read Untidy Excel Files*. https://CRAN.R-project.org/package=tidyxl.

Glanville, Yvonne. 2016. *joinXL: Perform Joins or Minus Queries on 'Excel' Files*. https://doi.org/10.32614/CRAN.package.joinXL.

Gohel, David, and Panagiotis Skintzos. 2023. *Flextable: Functions for Tabular Reporting*. https://ardata-fr.github.io/flextable-book/.

Gruson, Hugo. 2023. *Xlcutter: Parse Batches of 'Xlsx' Files Based on a Template*. https://doi.org/10.32614/CRAN.package.xlcutter.

Heidler, Tobias. 2024. *Flexlsx: Exporting 'Flextable' to 'Xlsx' Files*. https://github.com/pteridin/flexlsx.

Henze, Felix. 2024. *SheetReader: Parse Xlsx Files*. https://doi.org/10.32614/CRAN.package.SheetReader.

Hilderson, Nicholas. 2025. *Xlr: Create Table Summaries and Export Neat Tables to 'Excel'*. https://doi.org/10.32614/CRAN.package.xlr.

Kapoulkine, Arseny. 2006-2023. *Pugixml*. https://pugixml.org.

Kim, Gwang-Jin. 2019. *Xlsx2dfs: Read and Write 'Excel' Sheets into and from List of Data Frames*. https://doi.org/10.32614/CRAN.package.xlsx2dfs.

Luginbuhl, Felix. 2024. *Xlcharts: Create Native 'Excel' Charts and Work with Microsoft 'Excel' Files*. https://doi.org/10.32614/CRAN.package.xlcharts.

Ooms, Jeroen. 2023. *Writexl: Export Data Frames to Excel 'Xlsx' Format*. https://CRAN.R-project.org/package=writexl.

Pousson, Eli. 2024. *openxlsx2Extras: Extra Functions for the Openxlsx2 Package*. https://github.com/elipousson/openxlsx2Extras.

Rodríguez, Jesus Maria Rodríguez. 2023. *Tablaxlsx: Write Formatted Tables in Excel Workbooks*. https://doi.org/10.32614/CRAN.package.tablaxlsx.

Schauberger, Philipp, and Alexander Walker. 2023. *Openxlsx: Read, Write and Edit Xlsx Files*. https://CRAN.R-project.org/package=openxlsx.

Schwartz, Marc. 2022. *WriteXLS: Cross-Platform Perl Based r Function to Create Excel 2003 (XLS) and Excel 2007 (XLSX) Files*. https://CRAN.R-project.org/package=WriteXLS.

Wickham, Hadley, and Jennifer Bryan. 2023. *Readxl: Read Excel Files*. https://CRAN.R-project.org/package=readxl.