A Short Introduction to Working With Data in R

Jonathan Whiteley

2023-09-07

Prerequisites

- Access to a copy of the R software
  - i.e., a "binary executable"
  - Go to *www.r-project.org* to get a copy, or ask your system administrator.
- Tidyverse packages installed on the same system as `R`
  - Please run this command in `R` *before* the workshop:

```
install.packages("tidyverse")
```

- Knowledge of common mathematical operations: arithmetic, logarithms, etc.
- Knowledge of basic `R` concepts, such as *variables*, *objects*, *operators*, *functions*, *packages*, etc.
  - This is covered in the first workshop: "A Gentle Introduction to R"

Section 1

Welcome

Pop Quiz

We will review these *at the end*, so you can see how much you have learned.

- If multiple packages have functions with the same name, how can you specify which one to use?
- Does `R` store data in memory or temporary files?
- What is the limit to the size of objects and datasets that can be loaded into `R`?
- TRUE or FALSE: `R` has rules and conventions for naming functions
- TRUE or FALSE: if you use one package from the `tidyverse`, you have to use all of them.

Answer in the chat:

What is your favourite emoji? Why do you like to use it so much?

Introductions

- Name
- Pronouns
- Job title, role
- *optional*: a favourite childhood treat or candy?

- What are you hoping to learn most in today's workshop?

Learning Objectives

- Load tabular data into R
- Explore data to check that it was loaded correctly
- Export data from R to external files
- Data frames
- Clean data
  - re-arrange & modify rows
  - Add & change columns
  - Edit values systematically
  - Change data types
- Tidy data
  - Change the *shape* of a data frame
- Re-use code, reproducible results, automated reports
  - Scripts
  - R Markdown, R Notebooks

Disclaimer

- There is often more than one way to achieve a desired result in `R`
- Some are faster in certain situations
- Some require less code, or are easier to write as code
- Some are more portable (work on multiple systems)
- But there is rarely as single 'best way'.

This workshop focuses on a coherent approach, that can be learned more easily and extended as needed to tackle bigger problems.

Feel free to take what you learn here and experiment, or explore alternatives. Find what works for *you*.

Section 2

Loading Data into R

The Working Directory

- When working with external files, it helps to know the current *working directory*
  - Any paths supplied to `R` functions will be relative to this path.

```
getwd()
```

- You can change the working directory with this command:

```
setwd('path/to/a/directory')
```

File paths

- A *file path* is a *character string* that represents the location of a file in your system (computer and OS)
- The format of paths can depend on the operating system (OS)
  - Some use `"/"` to separate directories
    e.g., "`/dir/subdir`"
  - Windows uses `"\"`
    e.g., `"C:\\dir\subdir"`
    `R` uses this as an escape character in strings, and must be escaped itself in paths (`"\\"`)
    e.g., `"C:\\\\dir\\subdir"`

Paths in R

- R generally uses and understands `"/"` in paths, even on Windows.[1]
  - e.g., `"C:/dir/subdir"`
  - on Windows, it also understands Windows-style paths:
    e.g, `"C:\\\\dir\\subdir"`
- R also has platform-independent functions for manipulating paths, such as `file.path()`, which I will use in examples to make them as reproducible as possible.

[1] For the gory details, see section 14.2 "Filepaths" in "An Introduction to R" (`help.start()`), `?file.path`, and documentation for related functions.

My paths are not like yours

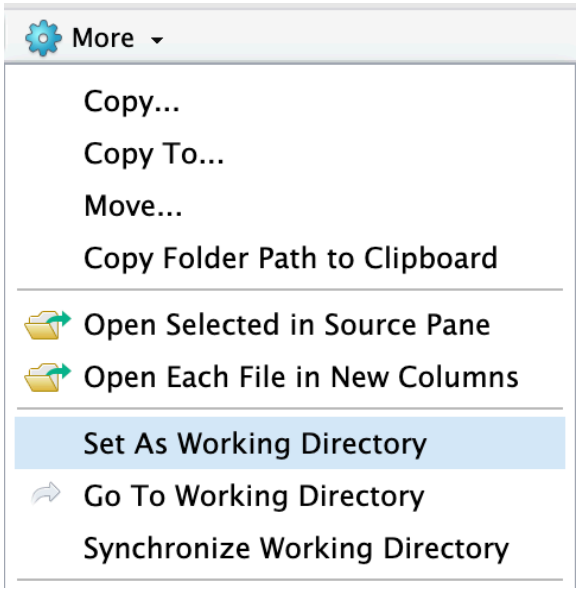
- Directory (folder) names can also vary from one computer to another — it's difficult to show a path in this document that will also work on your computer!
- Once you set a working directory *on your computer* based on the structure of the files in this project, we can use *relative paths* that should also work on your computer (assuming your downloaded the workshop files in the same structure as provided).

Set the working directory

- For this workshop, set the working directory to location where you downloaded this presentation and accompanying files.
  - the directory that *contains* the folder named 'data' that you downloaded along with the files for this workshop.
- Base R on Mac / Linux:
  - Menu item: "Misc > Change Working Directory..."
  - CMD+D on Mac; CTL+D on Linux (or Windows)
- In R Studio, you can use the *Files* pane (default bottom-right) to navigate to a directory in your system, and click on "*More > Set As Working Directory*"
  - or "*Session > Set Working Directory > To Files Pane Location*" in the R Studio menu.

- Base R on Windows:

```
setwd( choose.dir() )
```



Check your working directory

- Check to see that the working directory is in the right place, by checking to see if a known file exists (from R's perspective):

```
DF_path <- file.path("data", "data_example.csv")
file.exists(DF_path)
```

```
# [1] TRUE
```

- If the result of the statement above is not "TRUE" in your session, try one of the other approaches to change your working directory, and try again.

csv files

- 'csv' = **C**omma **S**eparated **V**alues
  - files in this format have a '`.csv`' file extension.
- They are:
  - plain text files
  - used to represent tabular data, with each *row* on a line, and values in each *column* separated by commas (`,`)
  - readable by a wide variety of analysis software (highly portable)
  - simple—no embedded metadata
- We'll try to load this file into `R`:

  `example_data.csv`
  - *optional: you can try opening it in a text editor, or spreadsheet software, to see what's in the file.*

Load a csv file into R (basic)

```
?read.csv
read.csv(DF_path)
```

```
# Error in read.table(file = file, header = header, sep = sep, quote
#   more columns than column names
```

Load a csv file into R (basic)

```
?read.csv
read.csv(DF_path)
```

```
# Error in read.table(file = file, header = header, sep = sep, quote
#   more columns than column names
```

- Uh oh! Something's not right.

Check the file contents

- Let's take a peek at the first few lines and see if we can identify the problem:

```
readLines(DF_path, n = 4)
```

```
# [1] "Data from an experiment on the cold tolerance of the grass sp
# [2] " Modified from `data(CO2)`.  See `?CO2`."
# [3] "Type,Treatment,PlantNum,95,175,250,350,500,675,1000"
# [4] "Quebec,nonchilled,1,16,30.4,34.8,37.2,35.3,39.2,39.7"
```

Check the file contents

- Let's take a peek at the first few lines and see if we can identify the problem:

```
readLines(DF_path, n = 4)
```

```
# [1] "Data from an experiment on the cold tolerance of the grass sp
# [2] " Modified from `data(CO2)`.  See `?CO2`."
# [3] "Type,Treatment,PlantNum,95,175,250,350,500,675,1000"
# [4] "Quebec,nonchilled,1,16,30.4,34.8,37.2,35.3,39.2,39.7"
```

- The first **2** lines don't look like comma-separated values!
- They look like extra information that is not part of the data table *structure*.

Load a csv file into R

- We can tell R to skip the lines with no data:
  - and we'll *assign* the result to a variable so we can work on it

```
DF <- read.csv(DF_path, skip = 2)
```

- Just because there were no Errors from R, doesn't mean there's nothing wrong with the data!

Section 3

Exploring Your Data

Object class: data frame

Before we explore our new data set, first a short review of the kind of *object* we're dealing with:

```
class(DF)
```

```
# [1] "data.frame"
```

```
typeof(DF)
```

```
# [1] "list"
```

Data frames

head(): peek at the first few rows

```
head(DF)
```

```
#     Type  Treatment PlantNum  X95 X175 X250 X350
# 1 Quebec nonchilled         1 16.0 30.4 34.8 37.2
# 2 Quebec nonchilled         2 13.6 27.3 37.1 41.8
# 3 Quebec nonchilled         3 16.2 32.4 40.3 42.1
# 4 Québec    chilled         1 14.2 24.1 30.3 34.6
# 5 Québec    chilled         2  9.3 27.3 35.0 38.8
# 6 Québec    chilled         3 15.1 21.0 38.1 34.0
#                    X500  X675 X1000
# 1                  35.3  39.2  39.7
# 2                  40.6 41.4   44.3
# 3                  42.9  43.9  45.5
# 4 32.5 (umol/m^2 sec)  35,4  38.7
# 5                  38.6  37,5  42.4
# 6                 +38.9  39,6  41.4
```

Dimensions (rows & columns)

```
dim(DF)
```

```
# [1] 13 10
```

```
nrow(DF)
```

```
# [1] 13
```

```
ncol(DF)
```

```
# [1] 10
```

Names of elements (columns)

```
names(DF)
```

```
#  [1] "Type"      "Treatment" "PlantNum"  "X95"
#  [5] "X175"      "X250"      "X350"      "X500"
#  [9] "X675"      "X1000"
```

```
colnames(DF)
```

```
#  [1] "Type"      "Treatment" "PlantNum"  "X95"
#  [5] "X175"      "X250"      "X350"      "X500"
#  [9] "X675"      "X1000"
```

Look at a column

Remember: you can refer to elements within a data frame by *name*.

```
DF[, "Treatment"]
```

```
#  [1] "nonchilled" "nonchilled" "nonchilled" "chilled"
#  [5] "chilled"    "chilled"    "nonchilled" "nonchilled"
#  [9] "nonchilled" "chilled"    "chilled"    "chilled"
# [13] "chilled"
```

```
unique(DF$Type)
```

```
# [1] "Quebec"      "Québec"      "Mississippi"
```

!

Looks like there might be some inconsistencies in the Type column. We'll learn how to fix those soon, but these simple functions are already helping us understand our data.

str(): structure of an object

```
str(DF)
```

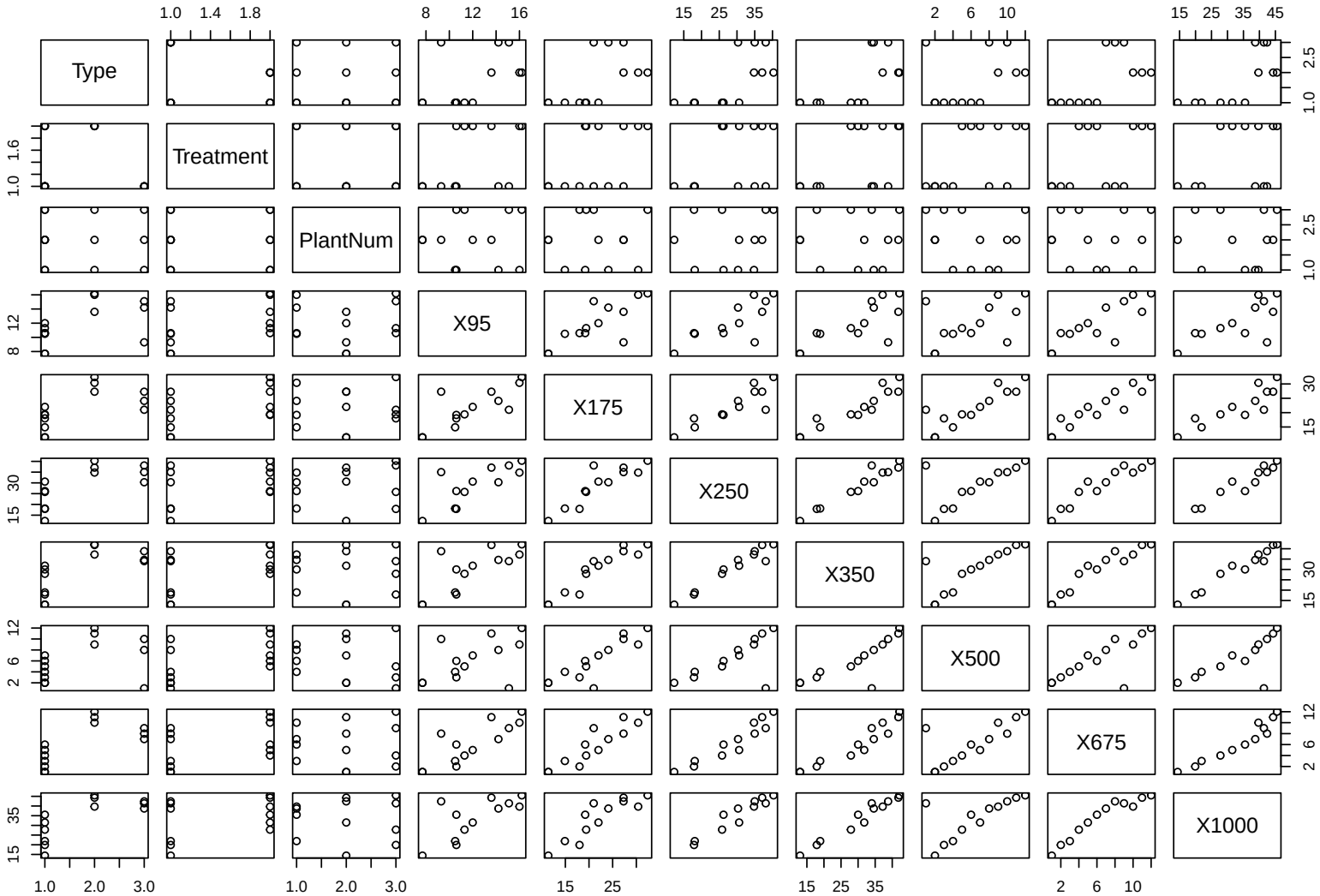
```
# 'data.frame': 13 obs. of  10 variables:
#  $ Type     : chr  "Quebec" "Quebec" "Quebec" "Québec" ...
#  $ Treatment: chr  "nonchilled" "nonchilled" "nonchilled" "chille
#  $ PlantNum : int  1 2 3 1 2 3 1 2 3 1 ...
#  $ X95      : num  16 13.6 16.2 14.2 9.3 15.1 10.6 12 11.3 10.5 .
#  $ X175     : num  30.4 27.3 32.4 24.1 27.3 21 19.2 22 19.4 14.9
#  $ X250     : num  34.8 37.1 40.3 30.3 35 38.1 26.2 30.6 25.8 18.
#  $ X350     : num  37.2 41.8 42.1 34.6 38.8 34 30 31.8 27.9 18.9
#  $ X500     : chr  "35.3" "40.6" "42.9" "32.5 (umol/m^2 sec)" ...
#  $ X675     : chr  "39.2" "41.4 " "43.9" "35,4" ...
#  $ X1000    : num  39.7 44.3 45.5 38.7 42.4 41.4 35.5 31.5 27.8 2
```

Tip

*The `str()` and `names()` functions can be used with **any** object.*

Simple plots

```
plot(DF)
```



Spreadsheet-like `View()`

```
View(DF)
```

- This command opens a data frame in a spreadsheet-like view, which can be easier to navigate.
- In R Studio, you can achieve the same thing by double-clicking on an object name in the '*Environment*' pane (default upper-right)
  - The `View()` pane in R Studio (default upper-left; '*Source*') also allows for sorting and filtering, but these do not change the viewed object, only the view.

Know Your Data

- These functions are useful for exploring different aspects of a loaded data set
- But they won't tell you if these are *correct*.
- Ideally, you should always "Know Your Data", and use these functions to verify that the data was loaded correctly.
  - Are the number of rows and columns what you expected?
  - Are the different columns of the expected type (numeric, character, etc.)?
  - Are the values in the expected range and format?
  - Is anything missing, or different than expected?

The CO2 dataset: background

The example data file is based on the 'CO2' dataset available in R (?CO2), with a few changes added to make things interesting.

From the documentation:

> *The CO2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.*

Exercise: what's wrong with this data?

The original dataset has the following properties (`str(CO2)`):

- 84 rows and 5 columns

Column Name	Description
`Plant`	factor with 12 levels: Qn1, Qn2, ... Mc3, Mc1
`Type`	factor with 2 levels: "Quebec" and "Mississippi"
`Treatment`	factor with 2 levels: "nonchilled" and "chilled"
`conc`	numeric: ambient carbon dioxide concentrations (mL/L)
`uptake`	numeric: carbon dioxide uptake rates (mol/m^2 sec)

Your turn

Using the functions described in this section, can you identify some possible issues and differences with the data set you loaded?

Spoiler alert: *suggested answers on the next slide.*

Exercise 1: what's wrong with this data

- The data we loaded has different dimensions!
  - Values from the `conc` column are shown as *column names*
  - `uptake` values are the values of these columns
  - This isn't necessarily *bad*: such a structure can be useful for presentation and interpretation by people, but it is not *tidy* and less convenient for analysis & visualization (more on this later).
- Some of the `uptake` values are *character*, but should be *numeric*
- One of the `Type` values is spelled inconsistently: "Quebec"/"Québec"
- The `PlantNum` column does not contain the original unique identifier in `Plant`
  - The values are no longer *unique*, without also considering the `Type` and `Treatment` columns.

There are other differences you may have noticed: we'll look at other ways to identify these automatically later.

Section 4

Downloading Data From the Internet

Downloading Data From the Internet

Section 5

Saving Data Outside R

Saving Data Outside R

Section 6

Re-using your code: scripts and other files

Re-using your code: scripts and other files

Section 7

The `tidyverse` collection of packages

The tidyverse

```
install.packages("tidyverse")
help(package="tidyverse")
```

- The tidyverse is an "opinionated" collection of packages that are designed to work together.
- All packages share an underlying design philosophy, grammar, and data structures.
  - *Unlike base R*
  - Shared naming conventions (e.g., '_' instead of '.' in function names)
  - Emphasis on functions that do one thing well
  - Designed to be combined together to achieve complex operations
- tidyverse is under active development.
  - New functions and features sometimes replace or supersede old ones.
  - No guarantee that functions will continue to work the same way in future versions.

Core `tidyverse` packages

Today, we will focus on a few of the core `tidyverse` packages for loading, cleaning, and manipulating data:

- readr, readxl for **loading** data
- dplyr for **manipulating** data (values)
- tidyr for **rearranging** data
- stringr for working with **strings**

`dplyr`: grammar of data manipulation

- `dplyr` provides many functions, within a coherent framework or *grammar*
- They are intended to help you focus on *what* you want to do, and translate your thoughts into code.
- High-level functions have active names and called "**verbs**" — they describe what they do.
- `dplyr` and `tidyr` provide many "**helper functions**" that work *inside* verbs and other functions to make many tasks easier to translate into code.
  - These functions may not work on their own, outside of `dplyr` verbs and `tidyr` functions.

dplyr verbs

Verbs can be grouped based on the component of the dataset that they work with[2]:

- Rows:
  - `filter()` chooses rows based on column values.
  - `slice()` chooses rows based on location.
  - `arrange()` changes the order of the rows.
- Columns:
  - `select()` changes whether or not a column is included.
  - `rename()` changes the name of columns.
  - `mutate()` changes the *values* of columns and creates new columns.
  - `relocate()` changes the order of the columns.
- Groups of rows:
  - `group_by()` defines groups of rows.
  - `summarise()` collapses a group into a single row.

[2] https://dplyr.tidyverse.org/articles/dplyr.html#single-table-verbs

`dplyr` *semantics*

`dplyr` verbs and helper functions let you refer to column names of the data frame directly in their arguments as regular variables — without having to quote them. But these names have different meanings (semantics) in different verbs.

- **"select semantics"**: in `select()` and similar functions, a column name refers to its *position* in the data frame.
  - you can refer to a column as a quoted string in `select()`, and it is interpreted as a reference to the column.
- **"mutate semantics"**: in `mutate()`, a column name refers to a *vector of values*.
  - you cannot supply a column name as a string in `mutate()`, because it is treated as a vector of length 1, rather than a reference to a column of values.

A 'pipe' operator

Figure 1: "La Trahison des Images" ("The Treachery of Images") or "Ceci n'est pas une pipe" ("This is not a pipe") by René Magritte.

- The `magrittr` package (included with `tidyverse`) provides a "forward-pipe operator":

```
%>%     # ?magrittr::`%>%`
```

- The `magrittr` package is automatically loaded when loading most `tidyverse` packages (e.g., `tidyr`, `dplyr`, `ggplot2`), as these packages all use this operator extensively.
  - It is often unnecessary to load `magrittr` separately, unless you are **not** using these other packages.

magrittr's 'forward-pipe' operator

- `%>%` allows you to pass results from an expression on the left-hand side (LHS) as an argument (usually the first) to a *function call* on the right-hand side (RHS).

This expression ...	is equivalent to:
`x %>% f()`	`f(x)`
`x %>% f(y)`	`f(x, y)`
`x %>% f(y, z = .)`	`f(y, z = x)`
`x %>% f %>% g %>% h`	`h(g(f(x)))`

- This can make code easier to read, as expressions are written and evaluated from *left to right*, rather than from *inside to outside* nested parentheses.

R now has a 'native' pipe operator

- A pipe operator was introduced in base R in v4.1 (May 2021)[3]:

```
|>     # ?pipeOp
```

- It was inspired by the "forward pipe operator" introduced by `magrittr`, but is more streamlined. See these links for details:
  - Differences between the base R and magrittr pipes
  - "Understanding the native R pipe |>"
- Because it is so new, most code examples online still use '%>%' from `magrittr`.
- But '|>' is always available *in R >= v4.1*, without having to load additional packages.
- This document will use '%>%' in the examples, for consistency and because many `tidyverse` functions were designed to work with it.

[3] https://cran.r-project.org/bin/windows/base/old/4.1.0/NEWS.R-4.1.0.html

Pipes: exercise

Section 8

Clean data

Clean data

Section 9

Tidy data

Tidy datasets

"Happy families are all alike; every unhappy family is unhappy in its own way."
— Leo Tolstoy

"Tidy datasets are all alike but every messy dataset is messy in its own way."
— Hadley Wickham (doi: 10.18637/jss.v059.i10)

- Tidy datasets provide a standardized way to link the *structure* of a dataset (its physical layout) with its *semantics* (its meaning).
  - `tidyr` vignette

Section 10

Review

Exercise

Quiz Review

Section 11

Backmatter

Other packages to look at

- `data.table`: a high-performance version of `data.frame` with few dependencies.

Other packages in the `tidyverse`:

- `lubridate` and `hms`: for date & time values.
- `purrr`: functional programming (FP) tools for working with functions and vectors.
  - Replace `for` loops with code that is more efficient and easier to read.

Writing to Microsoft Excel™ files

Packages that can write to Excel files:

- `xlsx`: read, write, format Excel 2007 (`.xlsx`) and Excel 97/2000/XP/2003 (`.xls`) files.
  - Depends on Java and the `rJava` package
- `XLConnect`: comprehensive and cross-platform R package for manipulating Microsoft Excel files (`.xlsx` & `.xls`) from within R.
  - Requires a Java Runtime Environment (JRE)
- `openxlsx`: simplified creation of Excel `.xlsx` files (**not** `.xls`).
  - *No dependency* on Java
- `writexl`: portable, light-weight data frame to **xlsx** exporter.
  - No Java or Excel required

!

I recommend *avoiding* exporting data to Excel files if possible. `csv` files are easier to read to & write from, and can be read by a wider variety of software (they are more portable).
Automated reports can be produced with R Markdown and output to a variety of more portable formats (pdf, HTML, etc.) instead.

References

Cheatsheets:

- readr/readxl
- Data transformation with dplyr
- Data tidying with tidyr