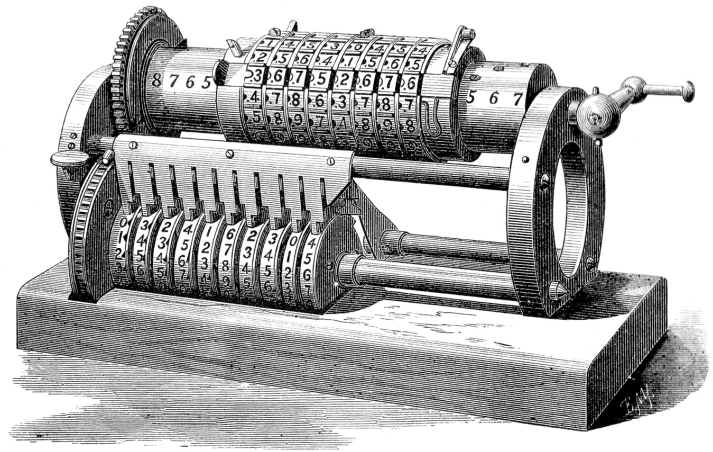
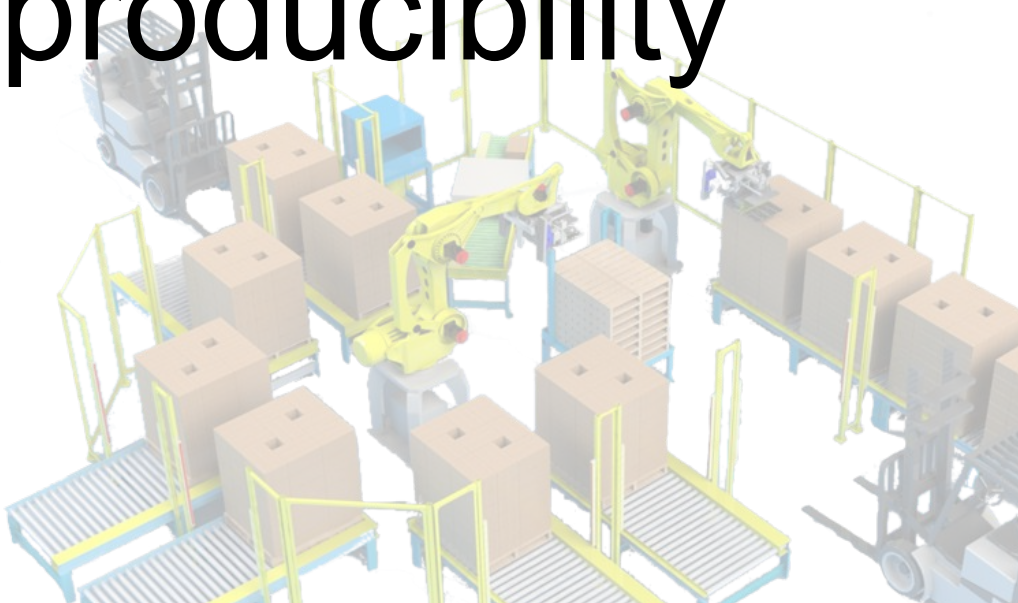
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programming in R



Unit 8: Reproducibility



Reproducibility

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "Dev" Track: Reproducibility
- "R" Track: Random Numbers

Dev Track
Reproducibility

Reproducibility

- Definition of "Reproducibility" varies a little and may mean different things depending on context. Here I talk about the concept in (data-)science. (Besides, I will use this chapter to also talk about related but different things)
- Strictest sense: One can use the same code and the same data and get to the same result. This is harder than it sounds:
  - If you did some analysis in an interactive R session without a script file there may be no "code"
  - If you used methods based on randomly sampled values you could get different results in different runs
  - If you run the code on a different computer with different package versions, results could differ
- More broad sense: One can use the same code with different data and the code does the same analysis. This is harder:
  - Number of samples could change, so use nrow(data) instead of writing it as a number in the code
  - Maybe other assumptions about the data should be inferred from the data as well
- Even broader sense: Others can use the code and data
  - Can easily be set up (e.g. all required dependencies are documented)
  - code is well documented and understandable (or: the method is simple and described well enough that someone could do the same analysis with little effort and come to the same results with the same raw data)
- Cf. "Replicability": When *new* empirical data is obtained, one will draw the same scientific conclusions. (This one is more about the validity of a result and not so much about code.)

Reproducibility -- Why do you want it?

- Save work: Run your analysis again
  - when you found a mistake in the middle
  - when new data arrives
  - when the goals of the project change slightly
  - when you have another very similar project
- (Professional and ethical) responsibility towards stakeholders (industry: your boss, coworkers; academia: other scientists, the public)
  - Make it possible to understand how you arrived at a conclusion (and check whether you make any incorrect implicit assumptions)
  - Give others the opportunity to easily build upon your work
  - If running your code with the data gives the *exact same* result then others can be sure they have the correct setup (and can use it to make other experiments)

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    |- data_prep.R
 |    |- models.R
 |    `- utils.R
 |- data/
 |    |- raw/
 |    |    |- gdp_2018.csv
 |    |    |- geomap.csv
 |    |    `- oecd_survey.csv
 |    `- intermediate/
 |         |- model_rf.rds
 |         `- model_gp.rds
 |- plots/
 |    |- growth_vs_gini.pdf
 |    `- map_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

(Created by git, for git repos)

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    |- data_prep.R
 |    |- models.R
 |    `- utils.R
 |- data/
 |    |- raw/
 |    |    |- gdp_2018.csv
 |    |    |- geomap.csv
 |    |    `- oecd_survey.csv
 |    `- intermediate/
 |         |- model_rf.rds
 |         `- model_gp.rds
 |- plots/
 |    |- growth_vs_gini.pdf
 |    `- map_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

Folder with relevant functions for your analysis. Ideally *only* functions: 'source()'-ing these files should not *do* anything besides exporting functions. This way, you can 'source()' these files after changing something, without having to wait or being uncertain about what is happening.

file names indicative of their content

general small utility functions that don't really have a common theme

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    |- data_prep.R
 |    |- models.R
 |    `- utils.R
 |- data/
 |    |- raw/
 |    |    |- gdp_2018.csv
 |    |    |- geomap.csv
 |    |    `- oecd_survey.csv
 |    `- intermediate/
 |         |- model_rf.rds
 |         `- model_gp.rds
 |- plots/
 |    |- growth_vs_gini.pdf
 |    `- map_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

data folder containing data, both raw data (given to you, or downloaded somewhere) and processed data / intermediate results

.rds is a good format for storing data in R; create with saveRDS and load with readRDS. Do *not* use .csv for intermediate results within R, since they may lose precision and format info.

Good idea to save plots as .pdf, since they can easily be used in latex. .png or .jpg files are worse: they have visible pixels when zooming in or printing.

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    `-          ediate/
 |                odel_rf.rds
 |                odel_gp.rds
 |- plot
 |    |-  wth_vs_gini.pdf
 |    `-  p_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

README.md file (capital letters are not *necessary* but are the convention) is shown by GitHub when opening the repository (or folder within the repository). Use this to document the repository: tell people what is in it, how to set things up (install dependencies), how to use things, give relevant links.

.Rmd-file that runs the analysis and creates a nice-looking html that contains all the plots and short explanations of what is happening. Bigger projects may have more than one of these, or have a normal R-script that creates expensive intermediate results

File that contains global settings, it does nothing except create variables. code like:

```
config.map.resolution <- 64
```

Learn about `.gitignore`!

git should only keep track of code and text (things you have done *manually*) but not of data or result objects (things that your code creates *automatically*). Otherwise:

- the repo can get very big
- you could accidentally share confidential data

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    |- data_prep.R
 |    |- models.R
 |    `- utils.R
 |- data/
 |    |- raw/
 |    |    |- gdp_2018.csv
 |    |    |- geomap.csv
 |    |    `- oecd_survey.csv
 |    `- intermediate/
 |         |- model_rf.rds
 |         `- model_gp.rds
 |- plots/
 |    |- growth_vs_gini.pdf
 |    `- map_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

Code that sources all files in the 'code' folder, as well as settings.R (and possibly other utility files):

```
lapply(list.files("code",
    pattern = "\\.R$",
    ignore.case = TRUE,
    full.names = TRUE,
    recursive = TRUE),
  source
)
source("settings.R")
```

Reproducibility: Folder Setup

- All files pertaining to a project should be contained in a folder
- Use `git` for version control and backups (e.g. on GitHub)
- Use a folder structure for the project to separate code and presentation from raw data, intermediate results, plots, produced reports etc.

Folder structure (suggestion):

```
project/
 |- .git/.....
 |- code/
 |    |- data_download.R
 |    |- data_prep.R
 |    |- models.R
 |    `- utils.R
 |- data/
 |    |- raw/
 |    |    |- gdp_2018.csv
 |    |    |- geomap.csv
 |    |    `- oecd_survey.csv
 |    `- intermediate/
 |         |- model_rf.rds
 |         `- model_gp.rds
 |- plots/
 |    |- growth_vs_gini.pdf
 |    `- map_growth.pdf
 |- .gitignore
 |- README.md
 |- report.Rmd
 |- settings.R
 `- source_all.R
```

This is a suggestion for a small project; depending on the needs of your project you should adapt it.

Reproducibility: Documentation

Documentation should come first:

1. what does the project actually do?
2. (if not using docker-files or renv, see later) what packages and requirements should be installed?
3. What commands should be run to get all the results?
4. What do the specific functions, commands etc. in the project actually do?

1.-3. should be documented in a "README" file. Because GitHub actually renders them as a nice webpage, it is now common to use a **README.md** file where these things are documented. It can contain headings, links, tables, pictures etc.

4. Should actually be documented inside the code and report files.

Reproducibility: Scripts

- You should definitely play around with your data in the interactive R session, try different things etc., this can be a very productive way of working
- You could even have some lines of code that you use a lot in the interactive session in some R-file in RStudio and ctrl-Enter these lines when you need them
- But what you do interactively is fundamentally not reproducible:
  - The history of commands that you ran contains many things that did not lead to your results, and may possibly not even get saved
  - The *state* of your R session (e.g. loaded data, values of variables) may be impossible to reconstruct

➔ Whenever you have an insight about your data, or constructed some useful code, write it down as either a function or as some lines of code in a *script file* or *.Rmd-file* (see next slide).

➔ Running your scripts top to bottom should give you all relevant results, without manual intervention!

➔ If your R command history was deleted tomorrow, you shouldn't have lost anything!

Reproducibility: R Markdown (.Rmd)-Files

R-Markdown (knitr under the hood) files are very useful for a reproducible workflow

- "Literate programming" -- Rmd puts documentation first and encourages you to write organised scripts
- Individual chunks can be executed and re-executed at will (but mind that the running them out-of-order is also dangerous and leads to non-reproducible states)
- Knitr allows *caching* of expensive results (but learn about cache invalidation: how to make knitr re-execute a cached result when dependencies changed?)
- Rmd is rendered to html or pdf **reports** that you can present to others directly
    - Changing the underlying data will directly generate an updated report!

Reproducibility: Simple Workflow

For a simple project, I would recommend a workflow like the following:

- `code/`-folder contains R-files that define useful functions
- optional `settings.R` file that contains global options (don't hard-code things such as URLs inside functions)
- `source_all.R` file that loads all scripts from code folder
- One (or several) `.Rmd`-files that do the actual analysis, with additional text and organized as a report that describes what is happening and shows plots and tables
  - should run `source("source_all.R")` in the beginning
  - should run `set.seed()` in the beginning (see following chapter!)
  - ... and then call functions from `code/`-folder
  - ... probably shouldn't contain too much low-level code (put that in `code/`)

Reproducibility: Running R

Some Notes:

- By default, R offers to save the content of the session and load it again next run. This is not good for reproducibility, because you could accidentally use values that were not created in a reproducible way.
  - If you run R (e.g. in a script), use the `--no-restore-data` flag to avoid loading data
  - Disable loading .RData in RStudio in Tools->Project options->Restore .RData at startup
  - The state of your R-session is not only determined by the available variables, but also the options() you set and the packages you load. Therefore `rm(list = ls(all.names = TRUE))` does *not* completely reset your state. For that, restart R with `--no-restore-data`
- Don't hard-code anything specific to your computer into your scripts; in particular, **don't write out the path of the project in the code**.
  - Always use relative paths, relative to the project's base directory (where the .git-folder is also)
  - Make sure you always run R from this path as well
  - don't use setwd()
  - You can use the "here" package to help you with this (although it is another package dependency for something relatively basic)

Reproducibility: Automation

- Everything that you do by hand is fundamentally not reproducible!
  - do not edit downloaded datasets by hand
  - do not copy-paste tables into the report file
  - do not edit plots with a graphics editor to shift around legends or axis labels
  - --- if your Master thesis supervisor gives you a new dataset at the end of your thesis you may not remember what fixes you made the first time you got the data!
- Automate things outside of R (e.g. folder creation, file deletion (careful here!))
  - using R itself (system2(), dir.create(), file.copy(), file.rename(), file.remove() (careful!), ...)
  - using **shell scripts**
    - Bourne shell for linux, Mac. Should also work on Windows if you have 'git bash', which you probably installed for this course during '01_git'.
    - Shell scripting is incredibly useful for many things but unfortunately not part of this course.

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs:

```
if (file.exists(resultfile)) {
  result <- readRDS(resultfile)
} else {
  result <- computeResult(...)
  saveRDS(result, resultfile)
}
```

Pros:

- Easy to write
- You know exactly what is happening
- No package dependency

Cons:

- Have to delete the file when computeResult() changes, or when input changes; If you forget to do that, you get broken results!

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs
  - Use workflow pipelining R package, like `drake` or `targets`

Pros:

- Everything happens in R
- Automatically manages cache and only does necessary work
- Can use multiple CPUs in parallel

Cons:

- Requires learning to use the package
- Excuse me ranting here for a while: I actually planned to include 'drake' as a teaching unit in this course, only to learn that it was "superseded" / deprecated by another package ('targets') a few months ago (beginning 2021). Wow, so good I didn't teach it last year, I totally would have wasted everyone's time! So this space seems to be "in motion" right now and who knows what package will be used in a few years. You are free to learn more about these packages, but consider putting that energy into learning to use **`Make`** (next slide) -- It has been around since the beginning of time (more specifically the 70's) and will still be around when the sun burns out.

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs
  - Use workflow pipelining R package, like `drake` or `targets`
  - Use `Make` (documentation, small tutorial, another tutorial, hints for Make on windows)

Pros:

- Is a bit like 'git' in that it is useful outside of R. You could e.g. use it to efficiently recompile your LaTeX files
- Easily mix different programming languages in your workflow
- Can use multiple CPUs in parallel

Cons:

- File format needs some getting used to
- Need special setup on windows
- Doesn't work with files that have spaces in them

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs
  - Use workflow pipelining R package, like `drake` or `targets`
  - Use `Make` (documentation, small tutorial, another tutorial, hints for Make on windows)
  - Use Snakemake or Nextflow

Pros:

- More robust & easier to use than `Make`
- Support distributed computing and containerization
- Pre-existing workflows

Cons:

- Possibly heavy non-R dependency
- Another thing to learn

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs
  - Use workflow pipelining R package, like `drake` or `targets`
  - Use `Make` (documentation, small tutorial, another tutorial, hints for Make on windows)
  - Use Snakemake or Nextflow
  - Use knitr's caching mechanism

Pros:

- Used inside .Rmd-files
- Automatically recognizes that cache should be cleared in many cases

Cons:

- You are bound to knitr-files with this
- Cache has to be cleared manually when underlying files change

Reproducibility: Automation in Pipelines

- For every intermediate result, you should always have the code that produced that intermediate result, so in principle you shouldn't even have to save that result
- However, some results take a long time to compute
- There are some mechanisms of saving or caching intermediate results
  - Use some R constructs
  - Use workflow pipelining R package, like `drake` or `targets`
  - Use `Make` (documentation, small tutorial, another tutorial, hints for Make on windows)
  - Use Snakemake or Nextflow
  - Use knitr's caching mechanism
- Unfortunately we will not have the time to go deeper into this
- My recommendation: one of Snakemake or Nextflow
- Also, learn basics of `Make` in your own time because it is also widely used.

Reproducibility: Working Environment

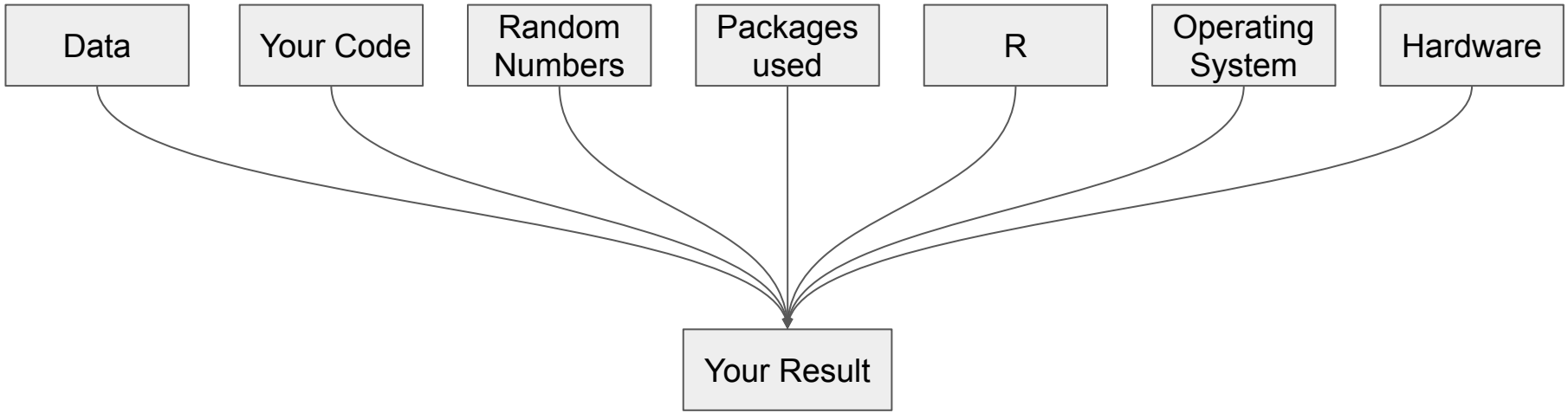
Consider some possibilities:

- Your script works well on your laptop, but for the new bigger dataset you have to use your institute's compute cluster -- will everything work as it should?
- Your supervisor gave you a script that you should extend, but when you run the script with the demo data you get a different result than he got. Did he send you the wrong script?

Thankfully you and your supervisor made sure to have a reproducible workflow, so you won't have to retrace whether you had to make manual fixes to your datafiles. But what could still go wrong?

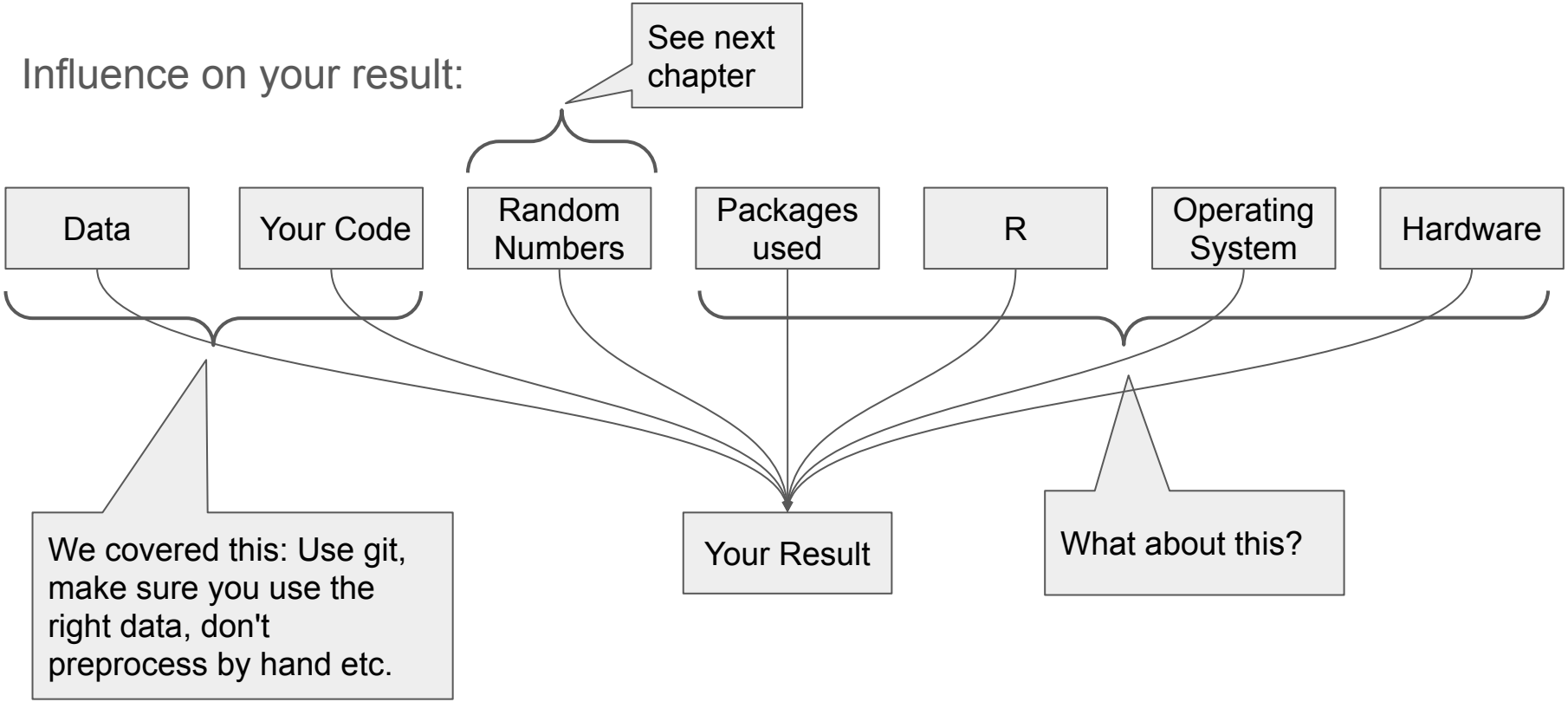
Reproducibility: Working Environment

Influence on your result:



Reproducibility: Working Environment

Influence on your result:



Reproducibility: Working Environment -- Packages

The R-packages that you use influence your result

- ➔ Changing package versions can change your result
- ➔ Make sure you use the correct package version when trying to reproduce a result
- ➔ Especially helpful when a package changes and suddenly your script doesn't run at all

Solutions:

- Install packages from Posit package manager snapshots from a *specific date*: `install.packages(<packages>, repos="https://packagemanager.posit.co/cran/YYYY-MM-DD/")`
- Use a package that manages a specific package library for your project, such `renv` (new).

I would personally classify this as "less important" than file versioning and an automated setup, but it is still good to have and can be very useful.

Reproducibility: Working Environment -- R and the OS

The rest of the software stack on your computer can influence results

- E.g. changes in the R version. Specific example: The default of 'stringsAsFactors' for many functions was changed from TRUE (in R ≤ 3.6) to FALSE (in R ≥ 4.0), so if your old scripts don't work on R 4, you may have to set `options(stringsAsFactors = FALSE)`
- Behavior of other software that you rely on (libraries used by R, programs that you call from R) could have changed
- Whether your program runs on Linux, Mac, Windows (and their versions) can have an influence on results (and some R programs may not work on some of these)

Solutions for this would be either to make sure to use the same R-version / apply some workarounds for specific changes and bugs, or to use system images in "containers", most notably **Docker containers**. These are basically files that contain a large part of an operating system, together with a specific R executable and all libraries being used. The container is then run in a virtualized environment, similar to a virtual machine / virtual computer

- Be aware of the possibility of influence of R versions and other things on your result
- (Docker-)containers is what you would do if you are *serious* about reproducibility
- but it goes beyond the scope of this course by far
- many people would probably consider this overkill for everyday work

However, I want to be the last person to be standing in the way between you and learning things, so here is a small intro tutorial for Docker for reproducibility, if you are interested. Docker is another one of these things that are useful for many different purposes outside of R.

Reproducibility: Working Environment -- Hardware

The hardware that you use can influence your results

- In theory you could be using a CPU that has a bug and behaves differently than someone else's CPU, but that is unlikely
- Your script could fail if you have **not enough RAM**; under some circumstances it might fail without even giving you an error message (and instead produce bogus results)
  - ➔ Be aware (and communicate) how much memory your script is likely to need
  - ➔ Always make sure you have enough RAM
- If you do computation (e.g. for neural networks) on a GPU, it could give different results than on a CPU
- If your results depend on the *timing* of things, they will obviously not be reproducible, but that effect could be amplified by running on a different platform.
  - ➔ Just don't do things that depend on timing and expect it to be reproducible!

What We Expect You to Know

- Adopting a **reproducible workflow** for your projects can save you time and stress in the long run.
- Set up your **project folders** so that data, code, results etc. are kept separate.
- Document what you are doing and how to run things, both in the **README.md** file and in the script / report files.
- Use **git**, and set up your **.gitignore** so that data, intermediate results etc. are not added to the repository.
- Use RMarkdown (**.Rmd**) files to create reports.
- Make sure your result can be reached from raw data **without any manual editing** of files
- Make sure that all steps needed to get the results are actually in script files, and that **interactive commands in the R console are not necessary**.
- Make your code independent of your computer; in particular do not use absolute paths and setwd(), instead always use paths relative to the project's folder.
- Consider using **shell scripts** to do some data preparation or result post-processing.
- Consider using an automation and dependency management tool such as **Snakemake** or **Nextflow**, especially when you have expensive intermediate results.