# Building Reproducible Analytical Pipelines

**Master of Data Science, University of Luxembourg - 2024**

Bruno Rodrigues

2024-11-04

# Table of contents

# Introduction

*This is the 2024 edition of the course. If you're looking for the 2023 edition, you can click here*

What's new:

- The book is now built using Quarto
- Updated links to newer materials
- Longer chatper on Github Actions
- New chapter on reproducibility with Nix

*This course is based on my book titled Building Reproducible Analytical Pipelines with R. This course focuses only on certain aspects that are discussed in greater detail in the book.*

## Schedule

- 2024/11/04 - 4 hours, Introduction to reproducibility and functional programming
- 2024/11/11 - 4 hours, Version control with Git, Package development and unit testing
- 2024/11/14 - 4 hours, Build automation
- 2024/12/02 - 4 hours, Literate programming and Shiny
- 2024/12/05 - 2 hours, Self-contained pipelines with Docker
- 2024/12/09 - 5 hours, CI/CD with Github Actions
- 2024/12/16 - 4 hours, Reproducibility with Nix

- 2024/12/19 - 3 hours, Wrap-up

## Reproducible analytical pipelines?

This course is my take on setting up code that results in some *data product*. This code has to be reproducible, documented and production ready. Not my original idea, but introduced by the UK's Analysis Function.

The basic idea of a reproducible analytical pipeline (RAP) is to have code that always produces the same result when run, whatever this result might be. This is obviously crucial in research and science, but this is also the case in businesses that deal with data science/data-driven decision making etc.

A well documented RAP avoids a lot of headache and is usually re-usable for other projects as well.

## Data products?

In this course each of you will develop a *data product*. A data product is anything that requires data as an input. This can be a very simple report in PDF or Word format or a complex web app. This website is actually also a data product, which I made using the R programming language. In this course we will not focus too much on how to create automated reports or web apps (but I'll give an introduction to these, don't worry) but our focus will be on how to set up a pipeline that results in these data products in a reproducible way.

# Machine learning?

No, being a master in machine learning is not enough to become a data scientist. Actually, the older I get, the more I think that machine learning is almost optional. What is not optional is knowing how:

- to write, test, and properly document code;
- to acquire (reading in data can be tricky!) and clean data;
- to work inside the Linux terminal/command line interface;
- to use Git, Docker for Dev(Git)Ops;
- the Internet works (what's a firewall? what's a reverse proxy? what's a domain name? etc, etc…);

But what about machine learning? Well, depending what you'll end up doing, you might indeed focus a lot on machine learning and/or statistical modeling. That being said, in practice, it is very often much more efficient to let some automl algorithm figure out the best hyperparameters of a XGBoost model and simply use that, at least as a starting point (but good luck improving upon automl…). What matters, is that the data you're feeding to your model is clean, that your analysis is sensible, and most importantly, that it could be understood by someone taking over (imagine you get sick) and rerun with minimal effort in the future. The model here should simply be a piece that could be replaced by another model without much impact. The model is rarely central… but of course there are exceptions to this, especially in research, but every other point I've made still stands. It's just that not only do you have to care about your model a lot, you also have to care about everything else.

So in this course we're going to learn a bit of all of this. We're going to learn how to write reusable code, learn some basics of the Linux command line, Git and Docker.

# What actually is reproducibility?

A reproducible project means that this project can be rerun by anyone at 0 (or very minimal) cost. But there are different levels of reproducibility, and I will discuss this in the next section. Let's first discuss some requirements that a project must have to be considered a RAP.

## The requirements of a RAP

For something to be truly reproducible, it has to respect the following bullet points:

- Source code must obviously be available and thoroughly tested and documented (which is why we will be using Git and Github);
- All the dependencies must be easy to find and install (we are going to deal with this using dependency management tools);
- To be written with an open source programming language (nocode tools like Excel are by default non-reproducible because they can't be used non-interactively, and which is why we are going to use the R programming language);
- The project needs to be run on an open source operating system (thankfully, we can deal with this without having to install and learn to use a new operating system, thanks to Docker);
- Data and the paper/report need obviously to be accessible as well, if not publicly as is the case for research, then within your company.

Also, reproducibility is on a continuum, and depending on the constraints you face your project can be "not very reproducible"

to "totally reproducible". Let's consider the following list of anything that can influence how reproducible your project truly is:

- Version of the programming language used;
- Versions of the packages/libraries of said programming language used;
- Operating System, and its version;
- Versions of the underlying system libraries (which often go hand in hand with OS version, but not necessarily).
- And even the hardware architecture that you run all that software stack on.

So by "reproducibility is on a continuum", what I mean is that you could set up your project in a way that none, one, two, three, four or all of the preceding items are taken into consideration when making your project reproducible.

This is not a novel, or new idea. Peng (2011) already discussed this concept but named it the *reproducibility spectrum*.

## Why R? Why not [insert your favourite programming language]

In my absolutely objective opinion R is currently the most interesting and simple language you can use to create such data products. If you learn R you have access to almost 20'000 packages (as of October 2023) to:

- clean data (see: `{dplyr}`, `{tidyr}`, `{data.table}`…);
- work with medium and big data (see: `{arrow}`, `{sparklyr}`…);
- visualize data (see: `{ggplot2}`, `{plotly}`, `{echarts4r}`…);

- do literate programming (using Rmarkdown or Quarto, you can write books, documents even create a website);
- do functional programming (see: `{purrr}`…);
- call other languages from R (see: `{reticulate}` to call Python from R);
- do machine learning and AI (see: `{tidymodels}`, `{tensorflow}`, `{keras}`…)
- create webapps (see: `{shiny}`…)
- domain specific statistics/machine learning (see CRAN Task Views for an exhaustive list);
- and more

It's not just about what the packages provide: installing R and its packages and dependencies is rarely frustrating, which is not the case with Python (Python 2 vs Python 3, `pip` vs `conda`, `pyenv` vs `venv` vs `uv`, …, dependency hell is a real place full of snakes)

That doesn't mean that R does not have any issues. Quite

the contrary, R sometimes behaves in seemingly truly bizarre ways (as an example, try running `nchar("1000000000")` and then `nchar(1000000000)` and try to make sense of it). To know more about such bizarre behaviour, I recommend you read *The R Inferno* (linked at the end of this chapter). So, yes, R is far from perfect, but it sucks less than the alternatives (again, in my absolutely objective opinion).

```
nchar("1000000000")
```

## Pre-requisites

I will assume basic programming knowledge, and not much more. If you need to set up R on your computer you can read the intro to my other book Modern R with the tidyverse. Follow the pre-requisites there: install R, RStudio and these packages:

```
install.packages(c("Ecdat", "devtools", "janitor",
 ↪  "plm", "pwt9",
    "quarto", "renv", "rio", "shiny", "targets",
     ↪  "tarchetypes",
    "testthat", "tidyverse", "usethis"))
```

The course will be very, very hands-on. I'll give general hints and steps, and ask you to do stuff. It will not always be 100% simple and obvious, and you will need to also think a bit by yourself. I'll help of course, so don't worry. The idea is to put you in the shoes of a real data scientist that gets asked at 9 in the morning to come up with a solution to a problem by COB. In 99% of the cases, you will never have encountered that problem ever, as it will be very specific to the company you're working

at. Google and Stackoverflow will be your only friends in these moments.

The beginning of this course will likely be the toughest part, especially if you're not familiar with R. I will need to bring you up to speed in 6 hours. Only after can we actually start talking about RAPs. What's important is to never give up and work together with me.

## Grading

The way grading works in this course is as follows: during lecture hours you will follow along. At home, you'll be working on setting up your own pipeline. For this, choose a dataset that ideally would need some cleaning and/or tweaking to be usable. We are going first to learn how to package this dataset alongside some functions to make it clean. If time allows, I'll leave some time during lecture hours for you to work on it and ask me and your colleagues for help. At the end of the semester, I will need to download your code and get it running. The less effort this takes me, the better your score. Here is a tentative breakdown:

- Code is on github.com and the repository is documented with a Readme.md file: 5 points;
- Data and functions to run pipeline are documented and tested: 5 points;
- Every software dependency is easily installed: 5 points;
- Pipeline can be executed in one command: 5 points;
- Bonus points: pipeline is dockerized, or uses Nix, and/or uses Github Actions to run? 5 points

The way to fail this class is to write an undocumented script that only runs on your machine and expect me to debug it to get it to run.

## Jargon

There's some jargon that is helpful to know when working with R. Here's a non-exhaustive list to get you started:

- CRAN: the Comprehensive R Archive Network. This is a curated online repository of packages and R installers. When you type `install.packages("package_name")` in an R console, the package gets downloaded from there;
- Library: the collection of R packages installed on your machine;
- R console: the program where the R interpreter runs;
- Posit/RStudio: Posit (named RStudio in the past) are the makers of the RStudio IDE and of the *tidyverse* collection of packages;
- tidyverse: a collection of packages created by Posit that offer a common language and syntax to perform any task required for data science — from reading in data, to cleaning data, up to machine learning and visualisation;
- base R: refers to a vanilla installation (and vanilla capabilities) of R. Often used to contrast a *tidyverse* specific approach to a problem (for example, using base R's `lapply()` in constrast to the *tidyverse* `purrr::map()`).
- `package::function()`: Functions can be accessed in several ways in R, either by loading an entire package at the start of a script with `library(dplyr)` or by using `dplyr::select()`.

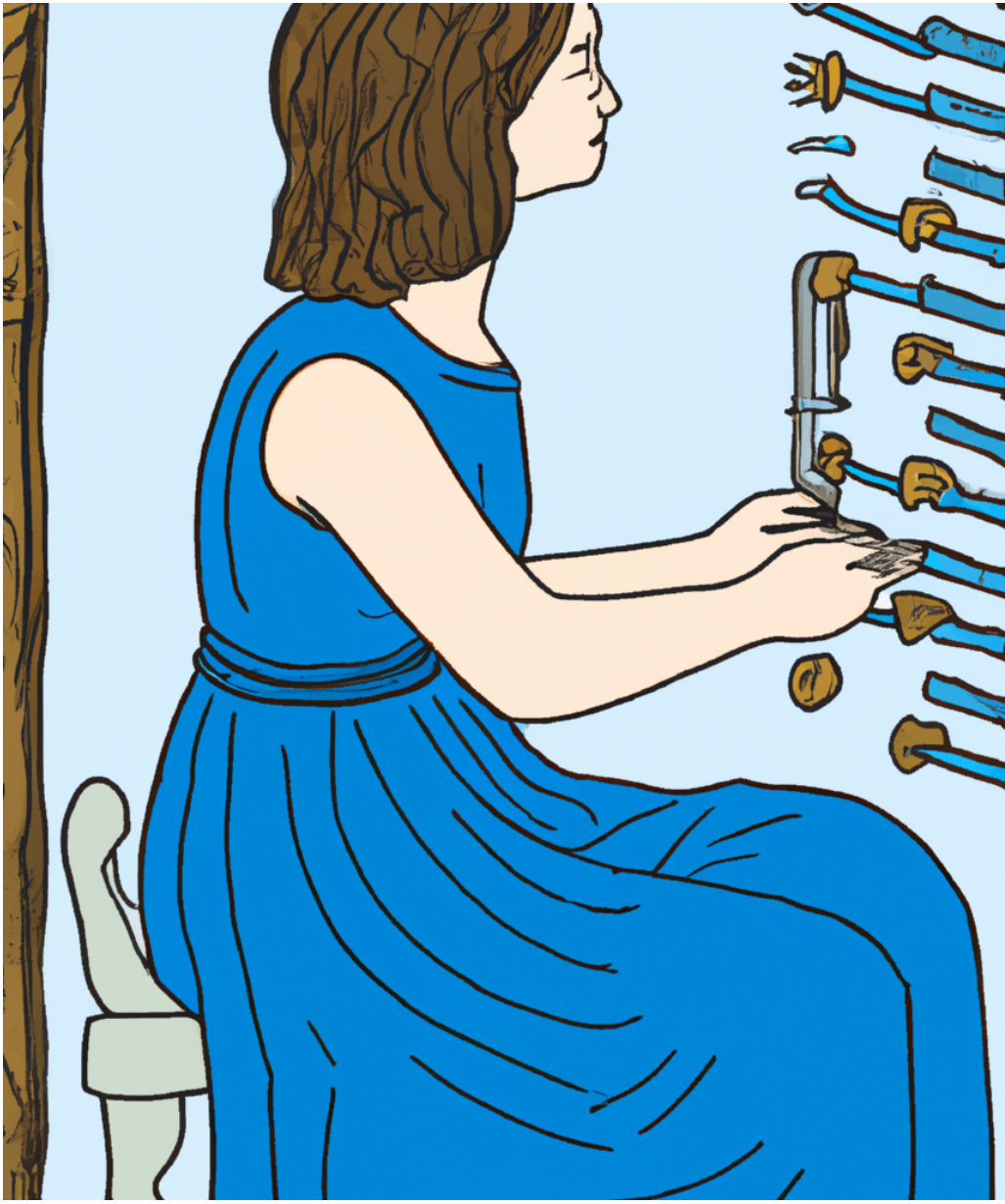
- Function factory (sometimes adverb): a function that returns a function.
- Variable: the variable of a function (as in `x` in `f(x)`) or the variable from statistical modeling (synonym of feature)
- `<-` vs `=`: in practice, you can use `<-` and `=` interchangeably. I prefer `<-`, but feel free to use `=` if you wish.

## Further reading

- An Introduction to R (from the R team themselves)
- What is CRAN?
- The R Inferno
- Building Reproducible Analytical Pipelines with R
- Reproducible Analytical Pipelines (RAP)

## License

# 1 Introduction to R



What you'll have learned by the end of the chapter: reading and writing, exploring (and optionally visualising) data.

## 1.1 Reading in data with R

Your first job is to actually get the following datasets into an R session.

First install the `{rio}` package (if you don't have it already), then download the following datasets:

- mtcars.csv
- mtcars.dta
- mtcars.sas7bdat
- multi.xlsx

Also download the following 4 `csv` files and put them in a directory called `unemployment`:

- unemp_2013.csv
- unemp_2014.csv
- unemp_2015.csv
- unemp_2016.csv

Finally, download this one as well, but put it in a folder called `problem`:

- mtcars.csv

and take a look at chapter 3 of my other book, Modern R with the {tidyverse} and follow along. This will teach you to import and export data.

`{rio}` is some kind of wrapper around many packages. You can keep using `{rio}`, but it is also a good idea to know which

packages are used under the hood by `{rio}`. For this, you can take a look at this vignette.

If you need to import very large datasets (potentially several GBs), you might want to look at packages like `{vroom}` (this benchmark shows a 1.5G csv file getting imported in seconds by `{vroom}`. For even larger files, take a look at `{arrow}` here. This package is able to efficiently read very large files (`csv`, `json`, `parquet` and `feather` formats).

## 1.2 A little aside on pipes

Since R version 4.1, a forward pipe `|>` is included in the standard library of the language. It allows to do this:

```
4 |>
  sqrt()
```

Before R version 4.1, there was already a forward pipe, introduced with the `{magrittr}` package (and automatically loaded by many other packages from the *tidyverse*, like `{dplyr}`):

```
library(dplyr)

4 %>%
  sqrt()
```

Both expressions above are equivalent to `sqrt(4)`. You will see why this is useful very soon. For now, just know this exists and try to get used to it.

## 1.3 Exploring and cleaning data with R

Take a look at chapter 4 of my other book, ideally you should study the entirety of the chapter, but for our purposes you should really focus on sections 4.3, 4.4, 4.5.3, 4.5.4, (optionally 4.7) and 4.8.

You should be able to read and understand expressions like the one below after having read the chapters above.

```
starwars %>%
  group_by(sex) %>%
  summarise(mean_height = mean(height, na.rm =
↪   TRUE))
```

While optional, the concept of list-columns is quite powerful and I wanted to say a few words about it. Take a look at the types of columns of the `starwars` dataset:

```
str(head(starwars, 9))
```

Each of the elements of the column `films` is a list. For example:

```
starwars %>%
  filter(name == "Luke Skywalker") %>%
  .$films
```

Because lists are very flexible and can contain any data type, it is possible to have a list-column of data frames. This is extremely useful to operate on groups without having to use loops:

```
starwars %>%
  group_nest(sex)
```

It is now possible to apply any function that takes a data frame as an input to each data frame from the `data` list-column:

```
starwars %>%
  group_nest(sex) %>%
  mutate(regression = lapply(data, \(x)(lm(height ~
 ↪  mass, data = x))),
         summary = lapply(regression, summary))
```

## 1.4 Data visualization

We're not going to focus on visualization due to lack of time. If you need to create graphs, read chapter 5.

## 1.5 Further reading

R for Data Science

# 2 A primer on functional programming

What you'll have learned by the end of the chapter: writing your own functions, functional programming basics (map, reduce, anonymous functions and higher-order functions).

## 2.1 Introduction

Functional programming is a way of writing programs that relies exclusively on the evalutation of functions. Mathematical functions have a very neat property: for any given input, they ALWAYS return exactly the same output. This is what we want to achieve with the functions that we will write. Functions that always return the same result are called pure, and a language that only allows writing pure functions is called a pure functional programming language. R is not a pure functional programming language, so we have to be careful not to write impure functions that manipulate the global state.

But what is state? Run the following code in your console:

```
ls()
```

This will list every object defined in the global environment. Now run the following line:

```
x <- 1
```

and then `ls()` again. `x` should now be listed alongside the other objects. You just manipulated the state of your current R session. Now if you run something like:

```
x + 1
```

This will produce 2. We want to avoid pipelines that depend on some definition of some global variable somewhere, which could be subject to change, because this could mean that 2 different runs of the same pipeline could produce 2 different results. Notice that I used the verb *avoid* in the sentence before. This is sometimes not possible to avoid. Such situations have to be carefully documented and controlled.

As a more realistic example, imagine that within the pipeline you set up, some random numbers are generated. For example, to generate 10 random draws from a normal distribution:

```
rnorm(n = 10)
```

Each time you run this line, you will get another set of 10 random numbers. This is obviously a good thing in interactive data analysis, but much less so when running a pipeline programmatically. R provides a way to fix the random seed, which will make sure you always get the same random numbers:

```
set.seed(1234)
rnorm(n = 10)
```

But `set.seed()` only works for one call, so you must call it again if you need the random numbers again:

```
set.seed(1234)
rnorm(10)
```

```
rnorm(10)
```

```
set.seed(1234)
rnorm(10)
```

The problem with `set.seed()` is that you only partially solve the problem of `rnorm()` not being pure; this is because while `rnorm()` now does return the same output for the same input, this only works if you manipulate the state of your program to change the seed beforehand. Ideally, we would like to have a pure version of `rnorm()`, which would be self-contained and not depend on the value of the seed defined in the global environment. There is a package developped by Posit (the makers of RStudio and the packages from the *tidyverse*), called `{withr}` which allows to rewrite our functions in a pure way. `{withr}` has several functions, all starting with `with_` that allow users to run code with some temporary defined variables, without altering the global environment. For example, it is possible to run a `rnorm()` with a seed, using `withr::with_seed()`:

```
library(withr)

with_seed(seed = 1234, {
  rnorm(10)
})
```

But ideally you'd want to go a step further and define a new function that is pure. To turn an impure function into a pure function, you usually only need to add some arguments to it. This is how we would create a `pure_rnorm()` function:

```
pure_rnorm <- function(..., seed){

  with_seed(seed, rnorm(...))
}

pure_rnorm(10, seed = 1234)
```

`pure_rnorm()` is now self-contained, and does not pollute the global environment. We're going to learn how to write functions in just a bit, so don't worry if the code above does not make sense yet.

A very practical consequence of using functional programming

is that loops are not used, because loops are imperative and imperative programming is all about manipulating state. However, there are situations where loops are more efficient than the alternative (in R at least). So we will still learn and use them, but only when absolutely necessary, and we will always encapsulate a loop inside a function. Just like with the example above, this ensures that we have a pure, self-contained function that we can reason about easily. What I mean by this, is that loops are not always very easy to decipher. The concept of loops is simple enough: take this instruction, and repeat it N times. But in practice, if you're reading code, it is not possible to understand what a loop is doing at first glance. There are only two solutions in this case:

- you're lucky and there are comments that explain what the loop is doing;
- you have to let the loop run either in your head or in a console with some examples to really understand whit is going on.

For example, consider the following code:

```
library(dplyr)

data(starwars)

sum_humans <- 0
sum_others <- 0
n_humans <- 0
n_others <- 0

for(i in seq_along(1:nrow(starwars))){

  if(!is.na(unlist(starwars[i, "species"])) &
```

```
     unlist(starwars[i, "species"]) == "Human"){
    if(!is.na(unlist(starwars[i, "height"]))){
      sum_humans <- sum_humans + unlist(starwars[i,
↪   "height"])
      n_humans <- n_humans + 1
    } else {

      0

    }

  } else {
    if(!is.na(unlist(starwars[i, "height"]))){
      sum_others <- sum_others + unlist(starwars[i,
↪   "height"])
      n_others <- n_others + 1
    } else {
      0
    }
  }
}

mean_height_humans <- sum_humans/n_humans
mean_height_others <- sum_others/n_others
```

What this does is not immediately obvious. The only hint you get are the two last lines, where you can read that we compute the average height for humans and non-humans in the sample. And this code could look a lot worse, because I am using functions like `is.na()` to test if a value is `NA` or not, and I'm using `unlist()` as well. If you compare this mess to a functional approach, I hope that I can stop my diatribe against imperative style programming here:

```
starwars %>%
  group_by(is_human = species == "Human") %>%
  summarise(mean_height = mean(height, na.rm =
 ↪  TRUE))
```

Not only is this shorter, it doesn't even need any comments to explain what's going on. If you're using functions with explicit names, the code becomes self-explanatory.

The other advantage of a functional (also called declarative) programming style is that you get function composition for free. Function composition is an operation that takes two functions $g$ and $f$ and returns a new function $h$ such that $h(x) = g(f(x))$. Formally:

```
h = g  f such that h(x) = g(f(x))
```

 is the composition operator. You can read `g  f` as *g after f*. When using functional programming, you can compose functions very easily, simply by using `|>` or `%>%`:

```
h <- f |> g
```

`f |> g` can be read as *f then g*, which is equivalent to *g after f*. Function composition might not seem like a big deal, but it actually is. If we structure our programs in this way, as a sequence of function calls, we get many benefits. Functions are easy to test, document, maintain, share and can be composed. This allows us to very succintly express complex workflows:

```
starwars %>%
  filter(skin_color == "light") %>%
  select(species, sex, mass) %>%
  group_by(sex, species) %>%
  summarise(
    total_individuals = n(),
    min_mass = min(mass, na.rm = TRUE),
    mean_mass = mean(mass, na.rm = TRUE),
    sd_mass = sd(mass, na.rm = TRUE),
    max_mass = max(mass, na.rm = TRUE),
    .groups = "drop"
  ) %>%
  select(-species) %>%
  tidyr::pivot_longer(-sex, names_to = "statistic",
↪   values_to = "value")
```

Needless to say, writing this in an imperative approach would be quite complicated.

Another consequence of using functional programming is that our code will live in plain text files, and not in Jupyter (or equivalent) notebooks. Not only does imperative code have state, but notebooks themselves have a (hidden) state. You should avoid notebooks at all costs, even for experimenting.

## 2.2 Defining your own functions

Let's first learn about actually writing functions. Read chapter 7 of my other book.

The most important concepts for this course are discussed in the following sections:

- functions that take functions as arguments (section 7.4)

```
my_func <- function(x, func, ...){
  func(x, ...)
}

my_func(c(1, 8, 1, NA, 8), mean, na.rm = TRUE)
```

- functions that take data (and the data's columns) as arguments (section 7.6);

```
simple_function <- function(dataset, col_name,
 ↪  var_name, fn_name){
  dataset %>%
    group_by(across({{col_name}})) %>%
    summarise("{{fn_name}}_{{var_name}}" :=
 ↪  {{fn_name}}({{var_name}}))
}

# Group by one column
simple_function(mtcars, vs, mpg, sd)

# Group by several columns
simple_function(mtcars, c(am, vs), mpg, sd)

# Even more general: group by any columns and apply
 ↪  any number of functions
simple_function <- function(dataset, cols, vars,
 ↪  fns){
  dataset %>%
    group_by(across({{cols}})) %>%
    summarise(across({{vars}}, fns, .names =
 ↪  "{.fn}_{.col}"))
```

```
}

simple_function(mtcars, c(am, vs), c(mpg, hp),
 ↪  list("mean" = mean, "sd" = sd))
```

Read more about it here and here.

## 2.3 Functional programming

You should ideally work through the whole of chapter 7, and then tackle chapter 8. What's important there are:

- `purrr::map()`, `purrr::reduce()` (sections 8.3.1 and 8.3.2)

Apply a function to each element of a vector or list:

```
map(seq(1:10), sqrt)
```

(`lapply()` is a base function that works similarly to `purrr::map()`)

Reduce a vector to a single element by iteratively applying a function:

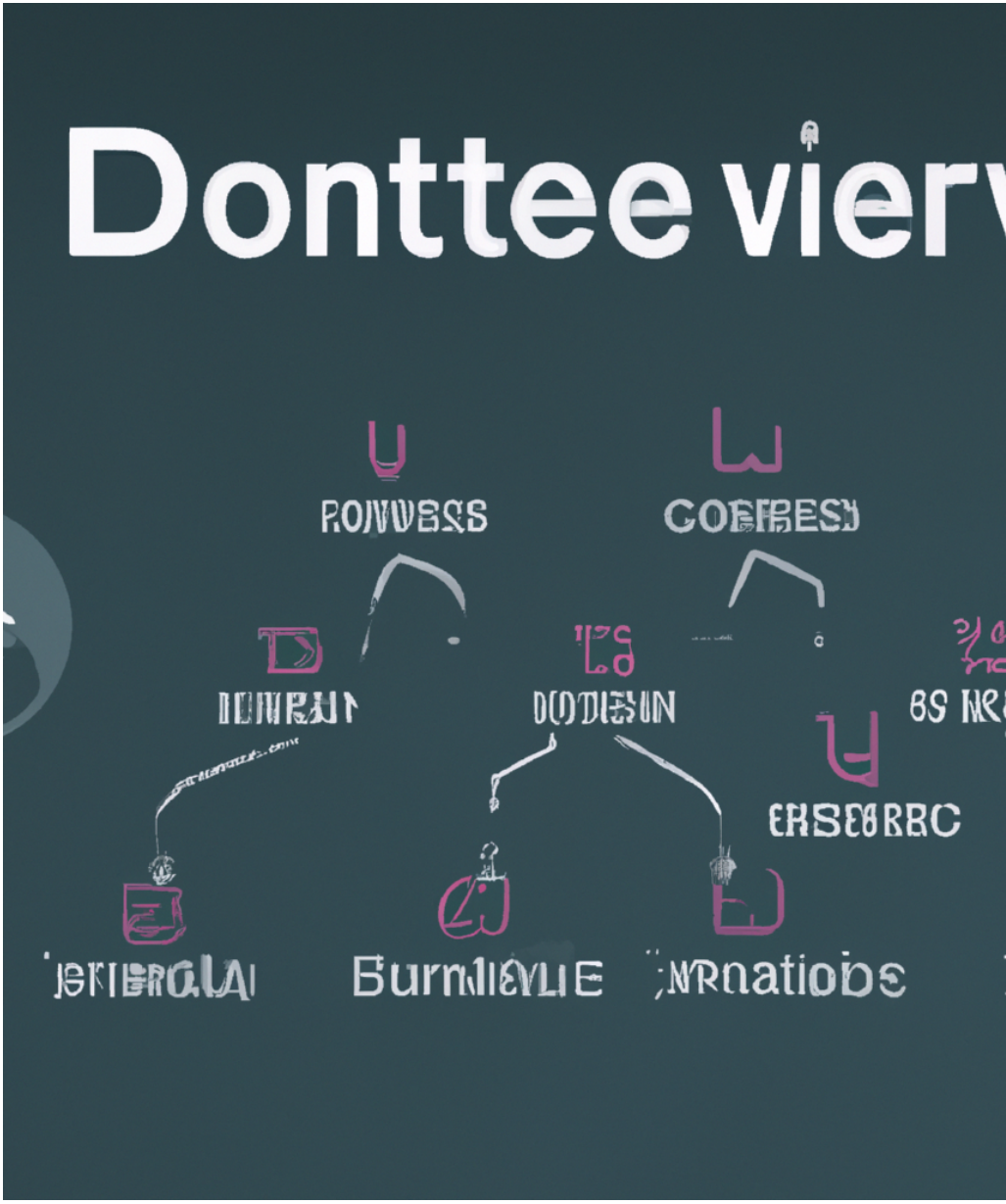
```
reduce(seq(1:10), `+`)
```

- And list based workflows (section 8.4)

## 2.4 Further reading

- Cleaner R Code with Functional Programming
- Functional Programming (Chapter from Advanced R)
- Why you should(n't) care about Monads if you're an R programmer
- Some learnings from functional programming you can use to write safer programs

# 3 Git



What you'll have learned by the end of the chapter: basics of working alone, and collaboration, using Git.

## 3.1 Introduction

Git is a software for version control. Version control is absolutely essential in software engineering, or when setting up a RAP. If you don't install a version control system such as Git, don't even start trying to set up a RAP. But what does a version control system like Git actually do? The basic workflow of Git is as follows: you start by setting up a repository for a project. On your computer, this is nothing more than a folder with your scripts in it. However, if you're using Git to keep track of what's inside that folder, there will be a hidden `.git` folder with a bunch of files in it. You can forget about that folder, this is for Git's own internal needs. What matters, is that when you make changes to your files, you can first *commit* these changes, and then push them back to a repository. Collaborators can copy this repository and synchronize their files saved on their computers with your changes. Your collaborators can then also work on the files, then commit and push the changes to the repository as well.

You can then pull back these changes onto your computer, add more code, commit, push, etc… Git makes it easy to collaborate on projects either with other people, or with future you. It is possible to roll back to previous versions of your code base, you can create new branches of your project to test new features (without affecting the main branch of your code), collaborators can submit patches that you can review and merge, and and and…

In my experience, learning git is one of the most difficult things there is for students. And this is because Git solves a complex problem, and there is no easy way to solve a complex problem. But I would however say that Git is not unnescessarily complex. So buckle up, because this chapter is not going to be easy.

Git is incredibly powerful, and absolutely essential in our line of work, it is simply not p ossible to not know at least some basics of Git. And this is what we're going to do, learn the basics, it'll keep us plenty busy already.

But for now, let's pause for a brief moment and watch this video that explains in 2 minutes the general idea of Git.

Let's get started.

You might have heard of github.com: this is a website that allows programmers to set up repositories on which they can host their code. The way to interact with github.com is via Git; but there are many other website like github.com, such as gitlab.com and bitbucket.com.

For this course, you should create an account on github.com. This should be easy enough. Then you should install Git on your computer.

## 3.2 Installing Git

Installing Git is not hard; it installs like any piece of software on your computer. If you're running a Linux distribution, chances are you already have Git installed. To check if it's already installed on a Linux system, open a terminal and type `which git`. If a path gets returned, like `usr/bin/gin`, congratulations, it's installed, if the command returns nothing you'll have to install

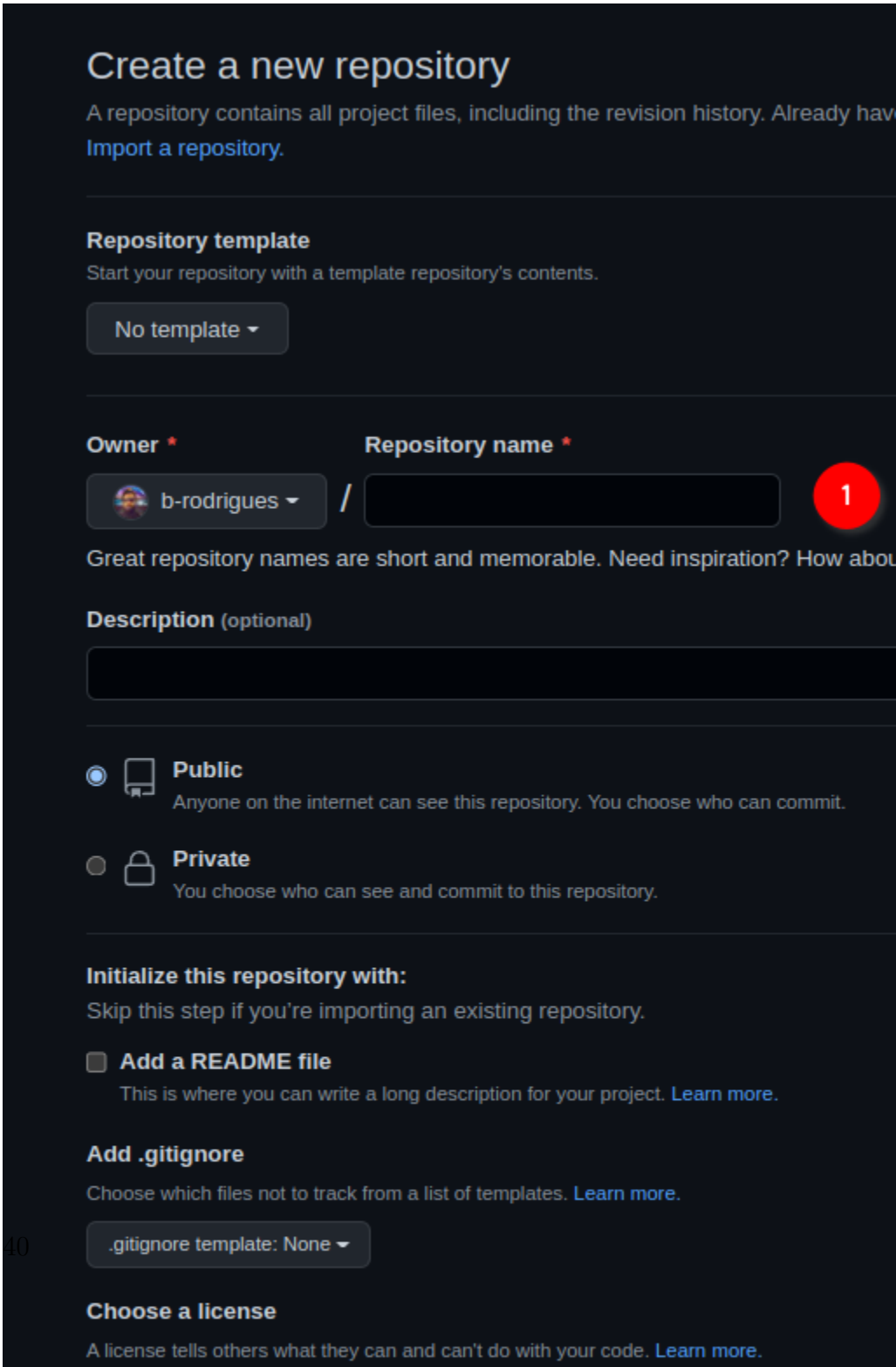
it. On Ubuntu, type `sudo apt-get install git` and just wait a bit. If you're using macOS or Windows, you will need to install it manually. For Windows, download the installer from here, and for macOS from here; you'll see that there are several ways of installing it on macOS, if you've never heard of homebrew or macports then install the binary package from https://sourceforge.net/projects/git-osx-installer/.

## 3.3 Setting up a repo

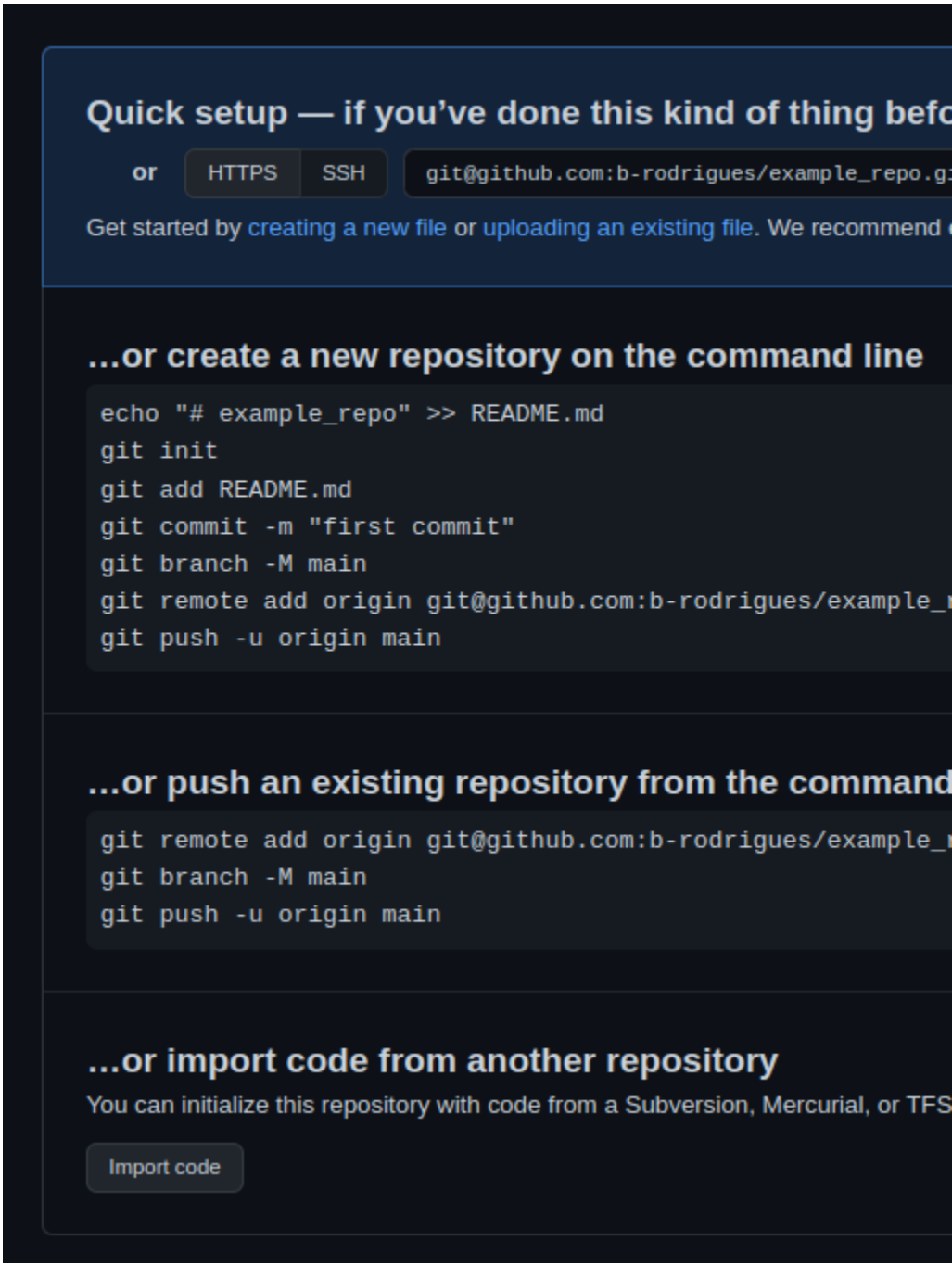
Ok so now that Git is installed, we can actually start using it. First, let's start by creating a new repository on github.com. As I've mentioned in the introductory paragraph, Git will allow you to interact with github.com, and you'll see in what ways soon enough. For now, login to your github.com account, and create a new repository by clicking on the 'plus' sign in the top right corner of your profile page and then choose 'New repository':


Search or jump to…
/
Pull requests
Issues
Mark
Overview
You unlocked
settings.
Popular reposi
chronicler
R
36
brotools
R
17
coolmlprojec
R
7
56 contribution
Oct
Mon
Wed
Fri
Bruno Rodrigues
b-rodrigues
Edit profile
192 followers · 12 following
MESR, Luxembourg
Luxembourg-City, Luxembourg
http://www.brodrigues.co/
Achievements

In the next screen, choose a nice name for your repository and ignore the other options, they're not important for now. Then click on 'Create repository':


Create a new repository
A repository contains all project files, including the revision history. Already hav
Import a repository.
Repository template
Start your repository with a template repository's contents.
No template
Owner *
Repository name *
b-rodrigues
/
1
Great repository names are short and memorable. Need inspiration? How abou
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Skip this step if you're importing an existing repository.
Add a README file
This is where you can write a long description for your project. Learn more.
Add .gitignore
Choose which files not to track from a list of templates. Learn more.
.gitignore template: None
Choose a license
A license tells others what they can and can't do with your code. Learn more.

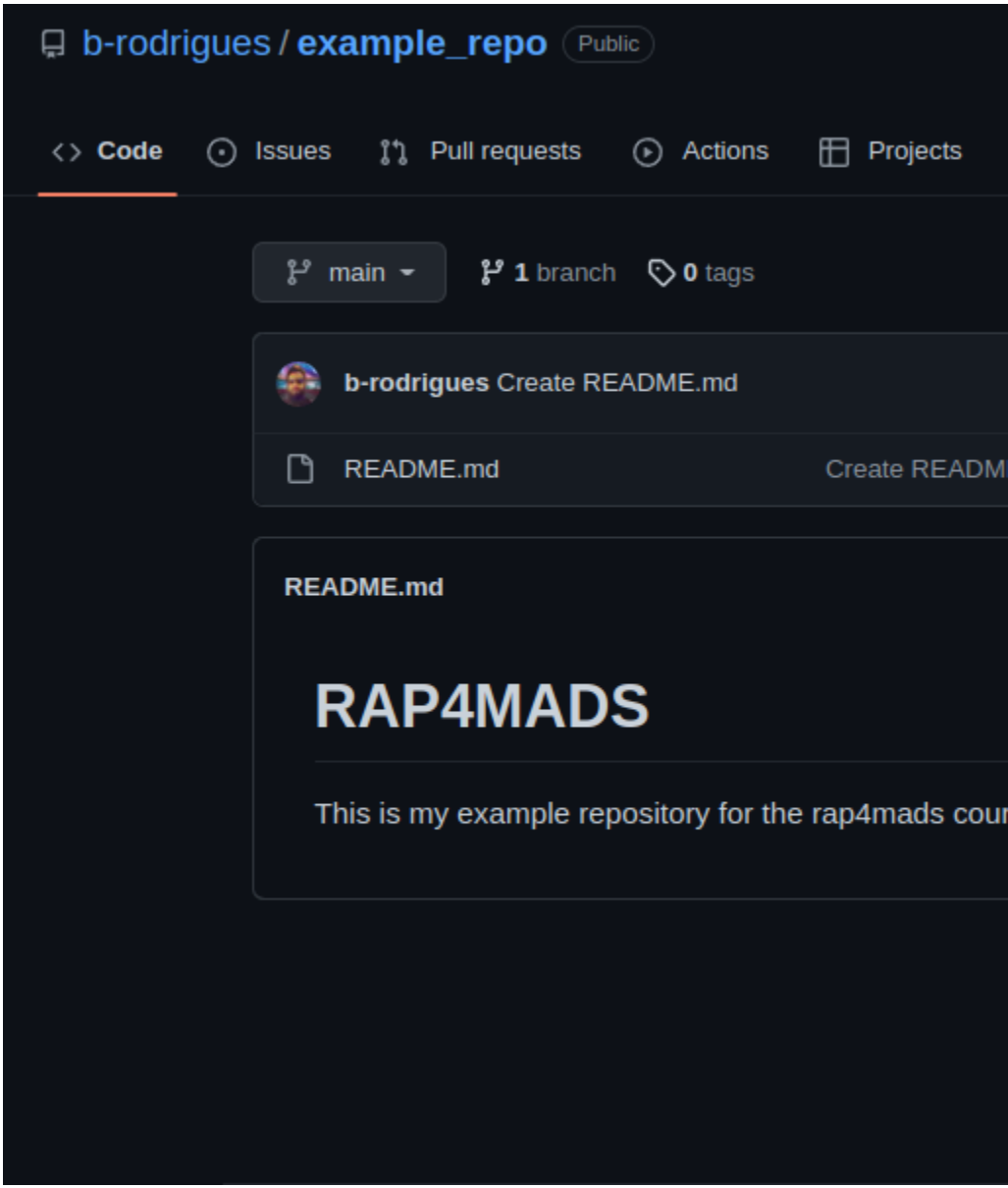
Ok, we're almost done with the easy part. The next screen tells us we can start interacting with the repository. For this, we're first going to click on 'README':


Quick setup — if you've done this kind of thing befo
or
HTTPS
SSH
git@github.com:b-rodrigues/example_repo.g
Get started by creating a new file or uploading an existing file. We recommend
...or create a new repository on the command line
echo "# example_repo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:b-rodrigues/example_
git push -u origin main
...or push an existing repository from the command
git remote add origin git@github.com:b-rodrigues/example_
git branch -M main
git push -u origin main
...or import code from another repository
You can initialize this repository with code from a Subversion, Mercurial, or TFS
Import code

This will add a `README` file that we can also edit from github.com directly:


Edit new file
Preview
1 # RAP4MADS
2
3 This is my example repository for the rap4mads course.
Attach files by dragging & dropping, selecting or pasting them.
Commit new file

Add some lines to the file, and then click on 'Commit new file'. You'll end up on the main page of your freshly created repository. We are now done with setting up the repository on github.com. We can now *clone* the repository onto our machines. For this, click on 'Code', then 'SSH' and then on the copy icon:

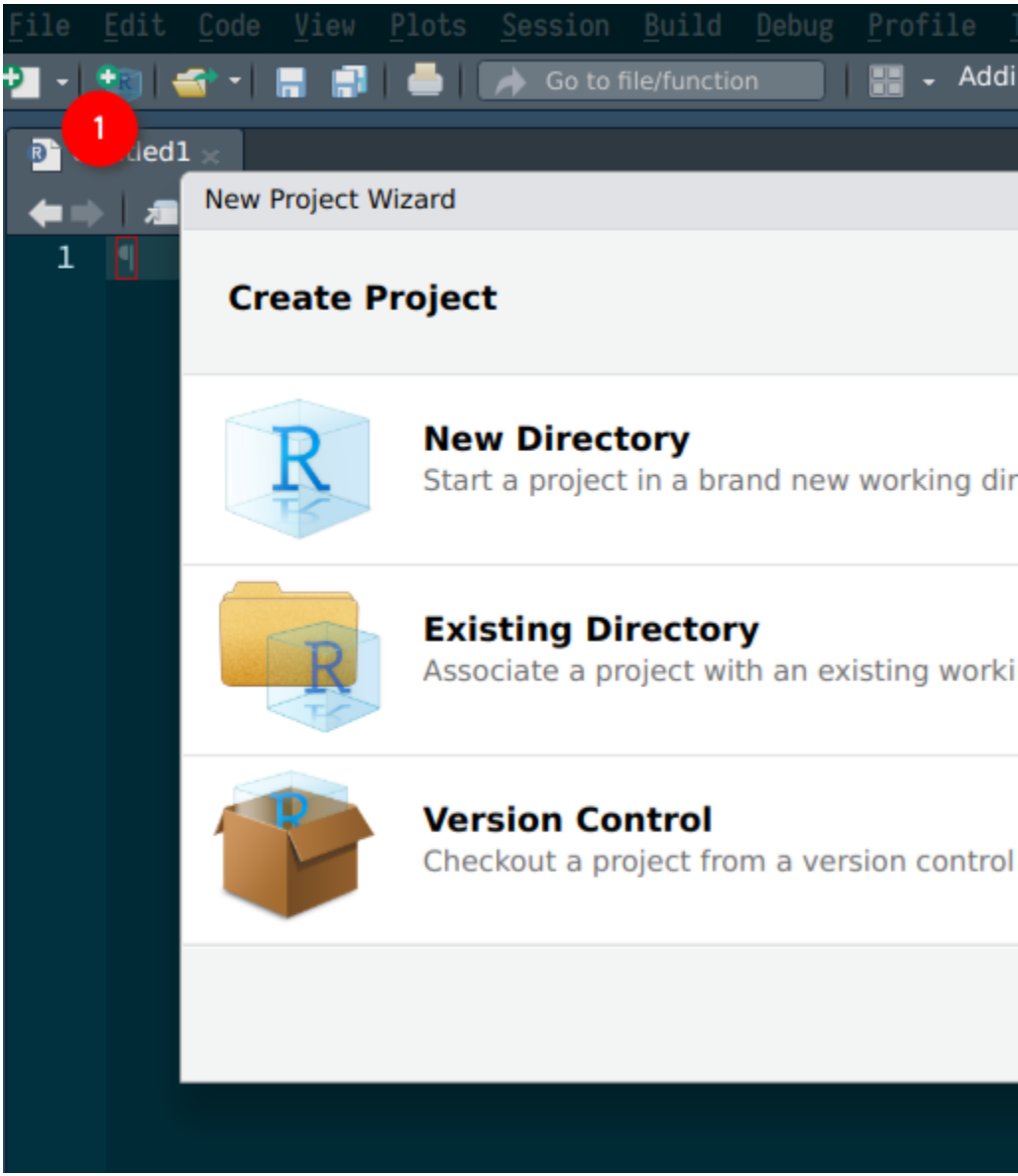



Now, to make things easier on you, we're going to use Rstudio as an interface for Git. But you should know that Git can be used

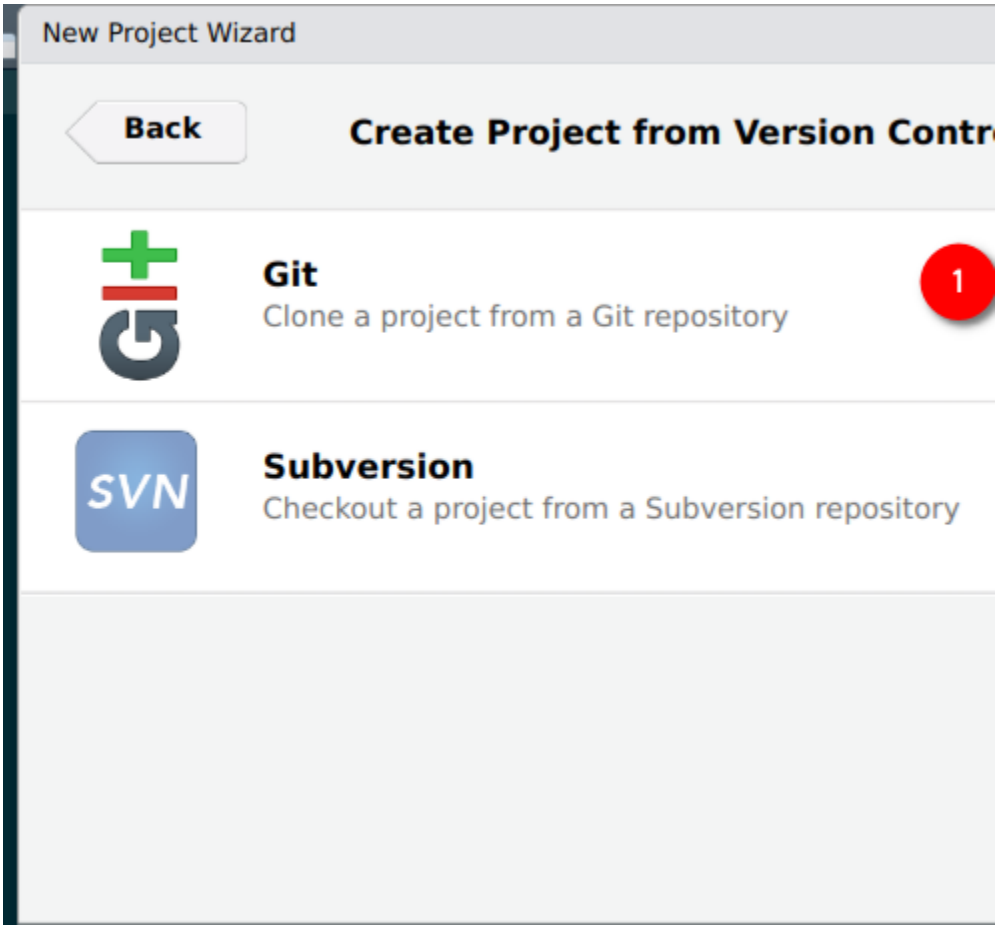
independently from a terminal application on Linux or macOS, or from Git Bash on Windows, and you should definitely get familiar with the Linux/macOS command line at some point if you wish to become a data scientist. This is because most servers, if not all, that you are going to interact with in your career are running some flavour of Linux. But since the Linux command line is outside the scope of this course, we'll use Rstudio instead (well, we'll use it as much as we can, because at some point it won't be enough and have to use the terminal instead anyways…).

## 3.4 Cloning the repository onto your computer

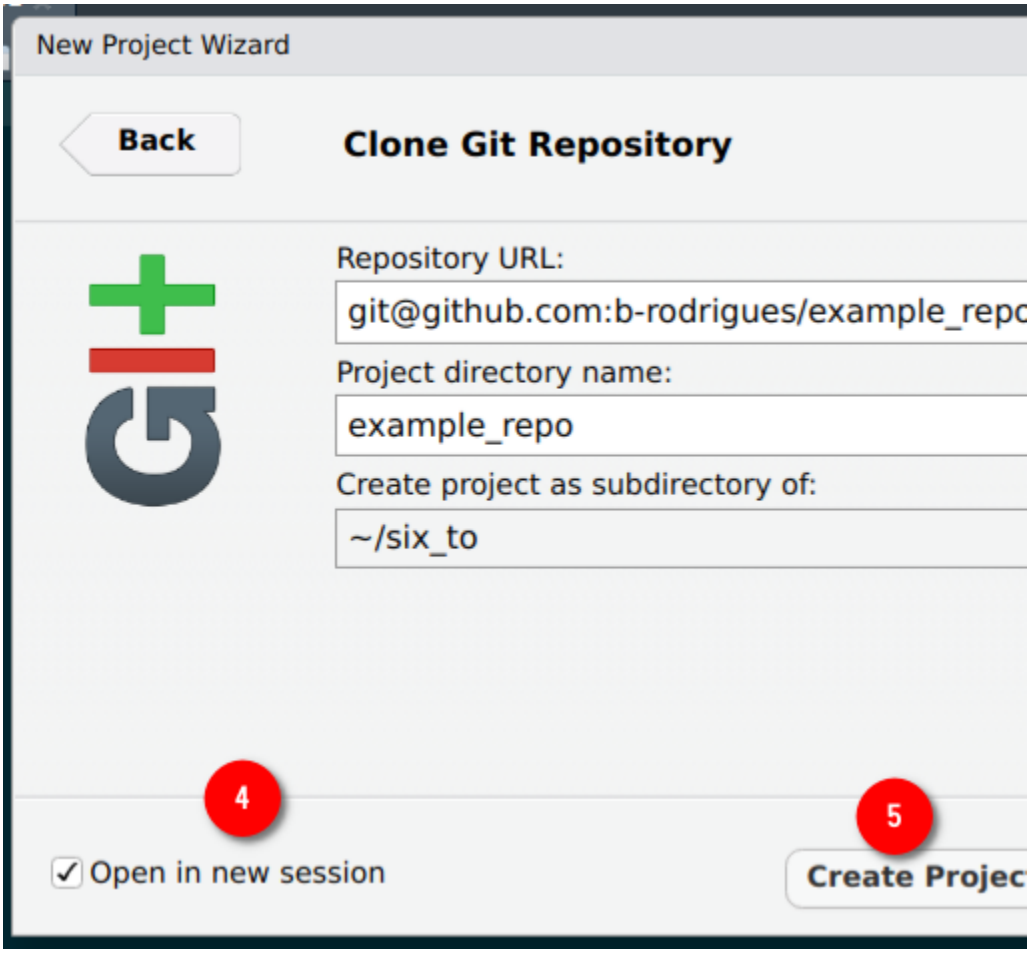
Start Rstudio and click on 'new project' and then 'Version Control':



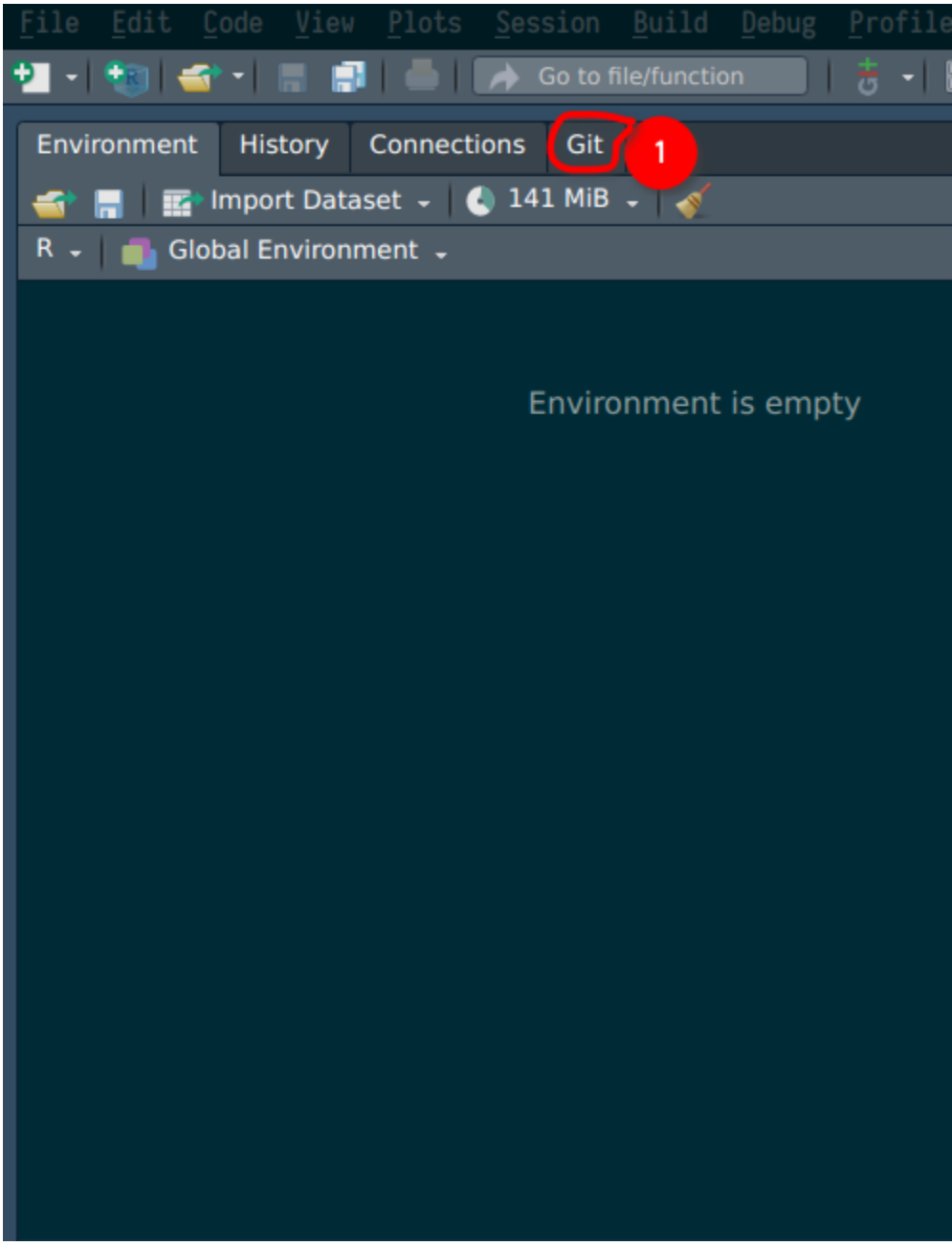

Then choose ‘Git’:




Then paste the link from before into the 'Repository URL' field, the 'project directory name' will fill out automatically, choose where to save the repository in your computer, click on 'Open in new session' and then on 'Create Project':



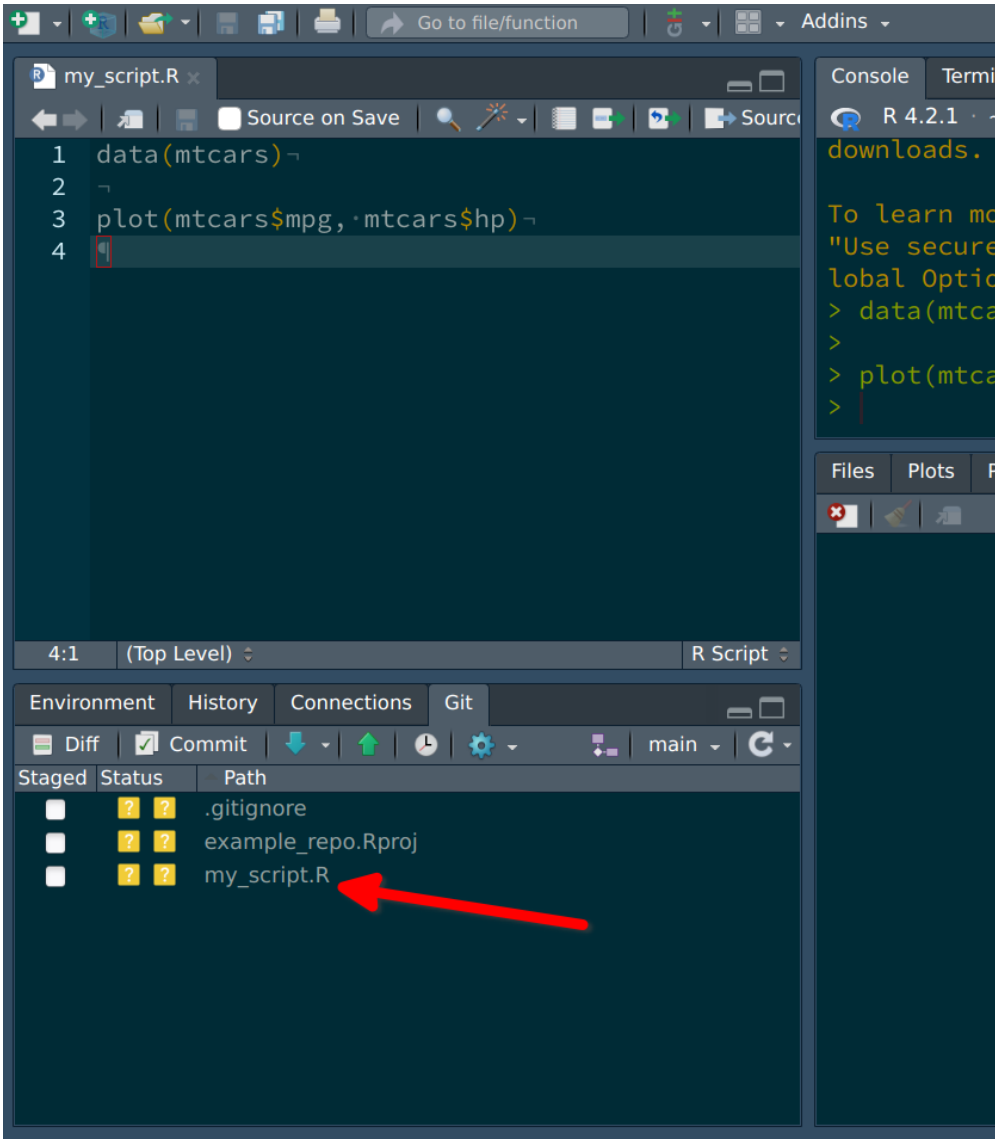

A new Rstudio window should open. There are several things that you should pay attention to now:


File Edit Code View Plots Session Build Debug Profile
Go to file/function
Environment
History
Connections
Git
1
Import Dataset
141 MiB
R
Global Environment
Environment is empty

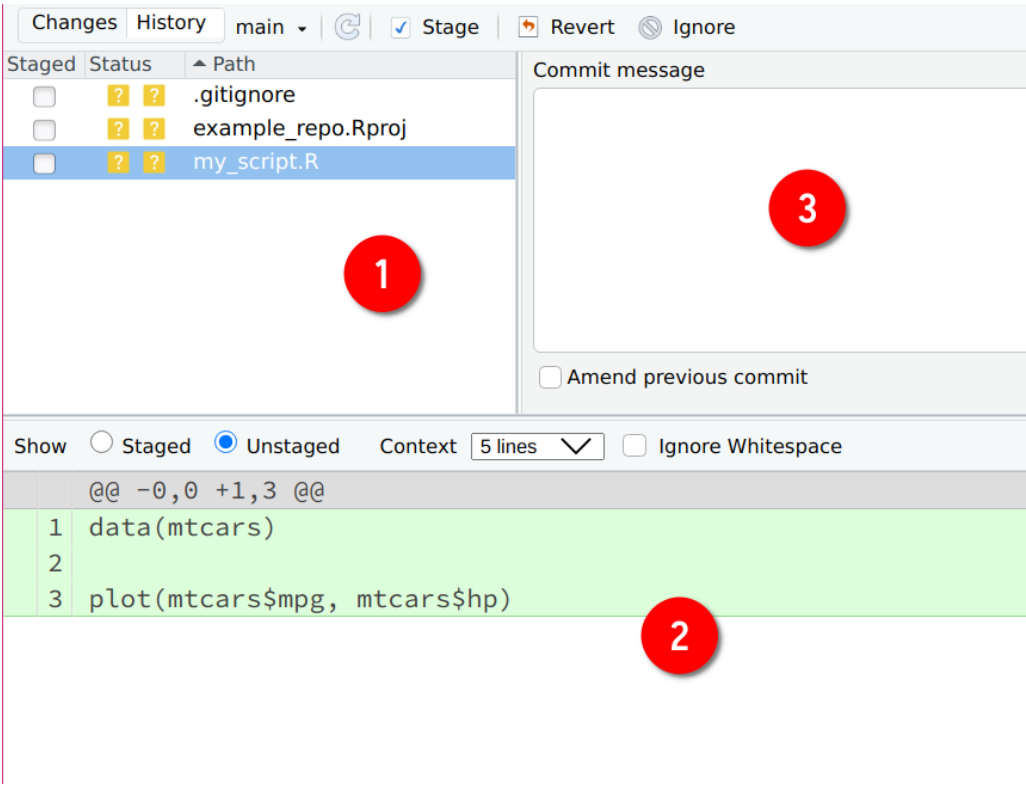
Icon (1) indicates that this project is *git-enabled* so to speak. (2) shows you that Rstudio is open inside the `example_repo` (or whatever you named your repo to) project, and (3) shows you the actual repository that was downloaded from github.com at the path you chose before. You will also see the `README` file that we created before.

## 3.5 Your first commit

Let's now create a simple script and add some lines of code to it, and save it. Check out the `Git` tab now, you should see your script there, alongside a `?` icon:

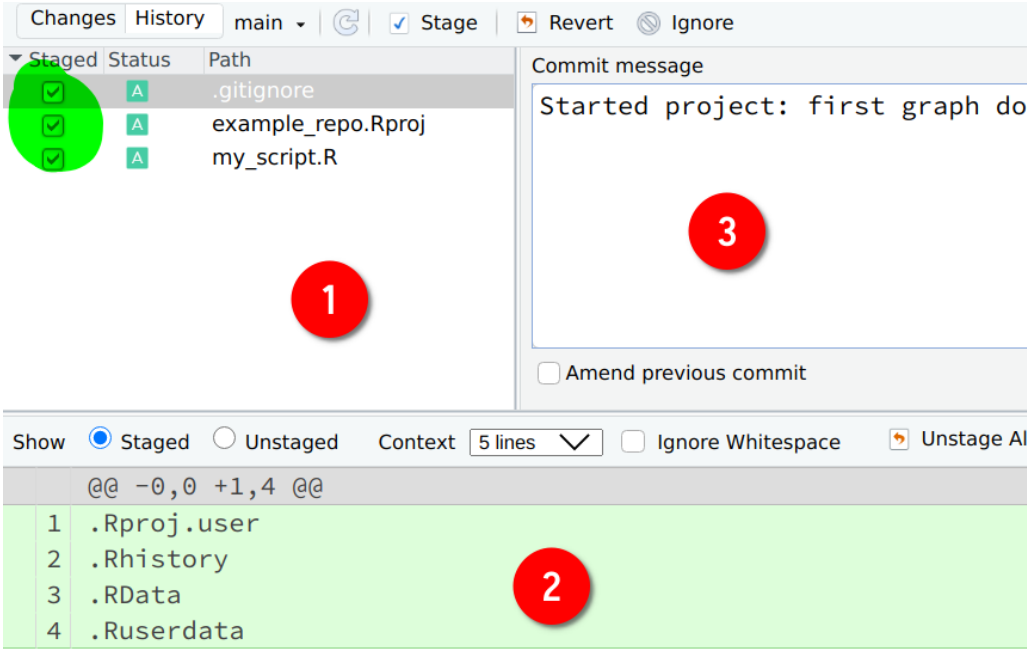



We are now ready to commit the file, but first let's check out what actually changed. If you click on `Diff`, a new window will open with the different files that changed since last time:



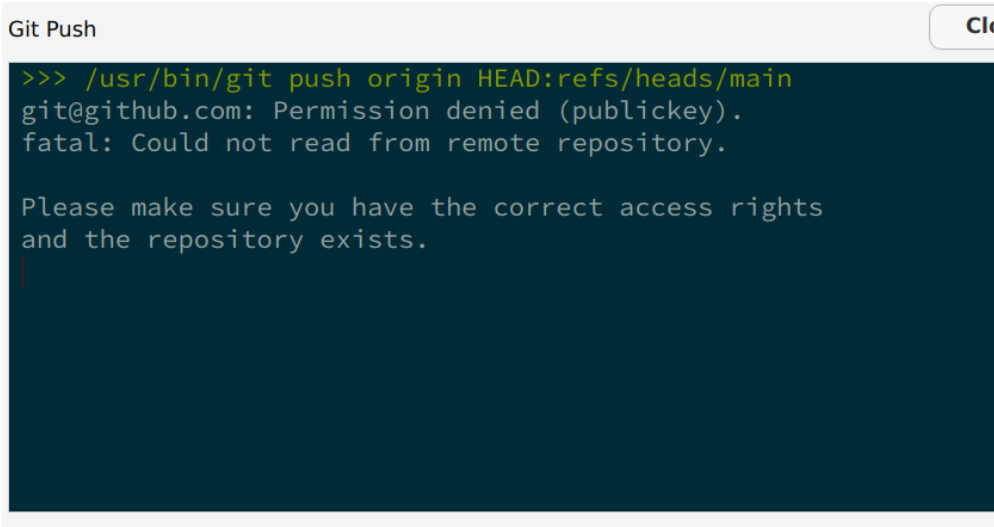

Icon (1) shows you the list of files that changed. We only created the file called `my_script.R`, but two other files are listed as well. These files are automatically generated when starting a new project. `.gitignore` lists files and folders that Git should not track, meaning, any change that will affect these files will be ignored by Git. This means that these files will also not be uploaded to github.com when committing. The file ending with the `.Rproj` extension is a RStudio specific file, which simply defines some variables that help RStudio start your project. What matters here is that the files you changed are listed, and that you saved them. You can double check that you actually correctly saved your files by looking at (2), which lists the lines that were

added (added lines will be highlighted in green, deleted lines in red). In (3) you can write a commit message. This message should be informative enough that a coworker, or future you, can read through them and have a rough idea of what changed. Best practice is to commit often and early, and try to have one commit per change (per file for example, or per function within that file) that you make. Let's write something like: "Started project: first graph done" as the commit message. We're almost done: now let's stage the files for this commit. This means that we can choose which files should actually be included in this commit. You can only stage one file, several files, or all files. Since this is our first commit, let's stage everything we've got, by simply clicking on the checkboxes below the column `Staged` in (1).




The status of the files now changed: they've been added for this commit. We can now click on the `Commit` button. Now these changes have been committed there are no unstaged files anymore. We have two options at this point: we can continue working, and then do another commit, or we can push our changes to github.com. Committing without pushing does not make our changes available to our colleagues, but because we committed them, we can recover our changes. For example, if I continue working on my file and remove some lines by mistake, I can recover them (I'll show you how to do this later on). But it is a much better idea to push our commit now. This makes our changes available to colleagues (who need to pull the changes from github.com) and should our computer spontaneously com-

bust, at least or work is now securely saved on github.com. So let's `Push`:

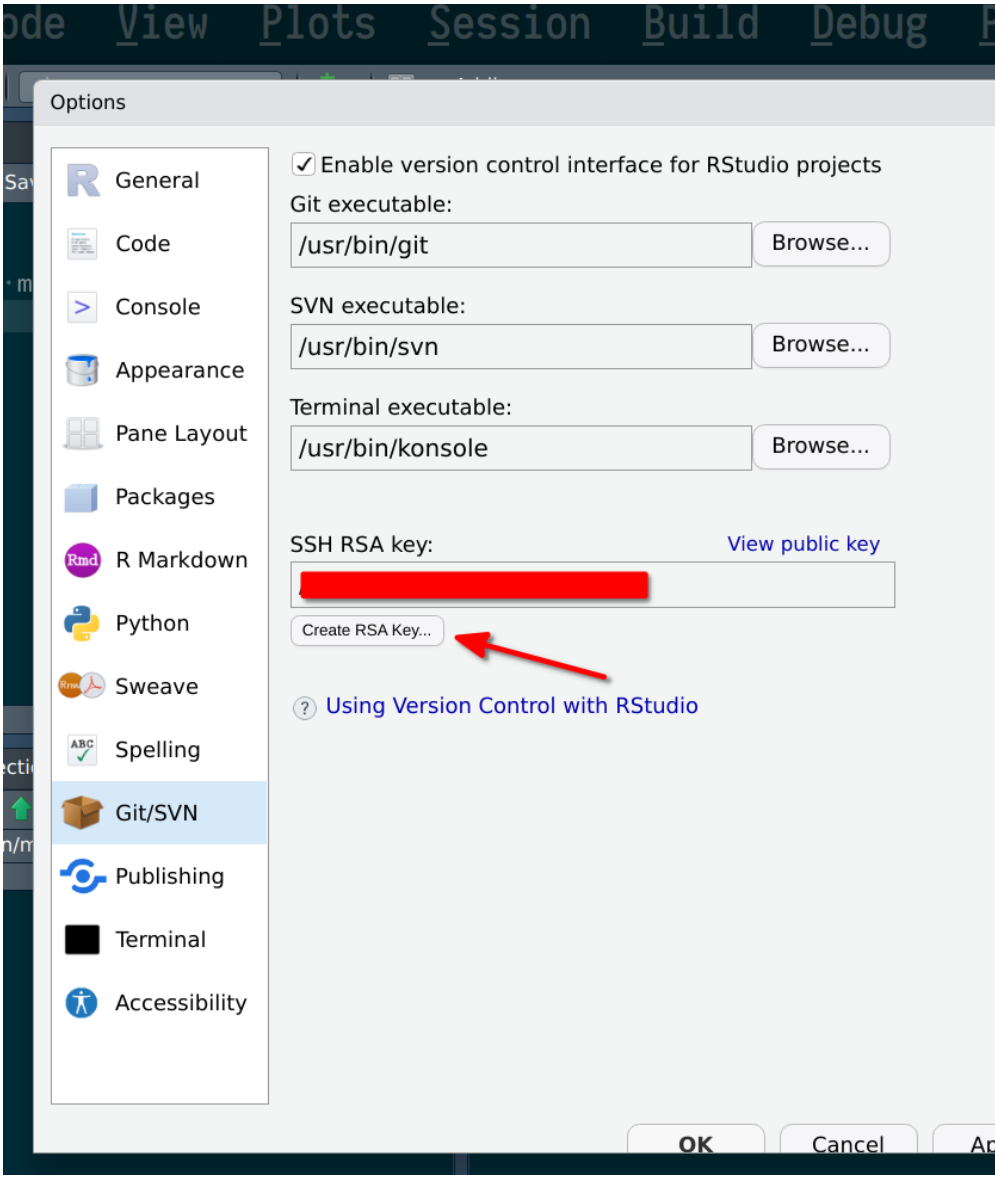



Ooooooops! Something's wrong! Apparently, we do not have access rights to the repo? This can sound weird, because after all, we created the repo with our account and then cloned it. So what's going on? Well, remember that anyone can clone a public repository, but only authorized people can push changes to it. So at this stage, the Git software (that we're using through RStudio) has no clue who you are. Git simply doesn't know that your the *admin* of the repository. You need to provide a way for Git to know by logging in. And the way you login is through a so-called ssh key.

Now if you thought that Git was confusing, I'm sorry to say that what's coming confuses students in general even more. Ok so what's a ssh key, and why does Git need it? *An* ssh key is actually a misnomer, because we should really be talking about a pair of keys. The idea is that you generated two files on the computer that you need to access github.com from. One of these

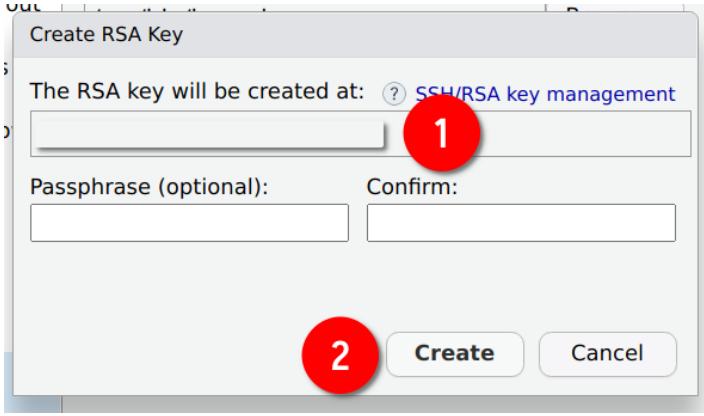
keys will be a public key, the other a private key. The private key will be a file usually called `id_rsa` without any extension, while the public key will be called the same, but with a `.pub` extension, so `id_rsa.pub` (we will generate these two files using RStudio in a bit). What you do is that you give the public key to github.com, but you keep your private key on your machine. Never, ever, upload or share your private key with anyone! It's called private for a reason. Once github.com has your public key, each time you want to push to github.com, what happens is that the public key is checked against your private key. If they match, github.com knows that you are the person you claim to be, and will allow you to push to the repository. If not you will get the error from before.

So let's now generate an ssh key pair. For this, go to `Tools > Global Options > Git/Svn`, and then click on the `Create RSA Key...`

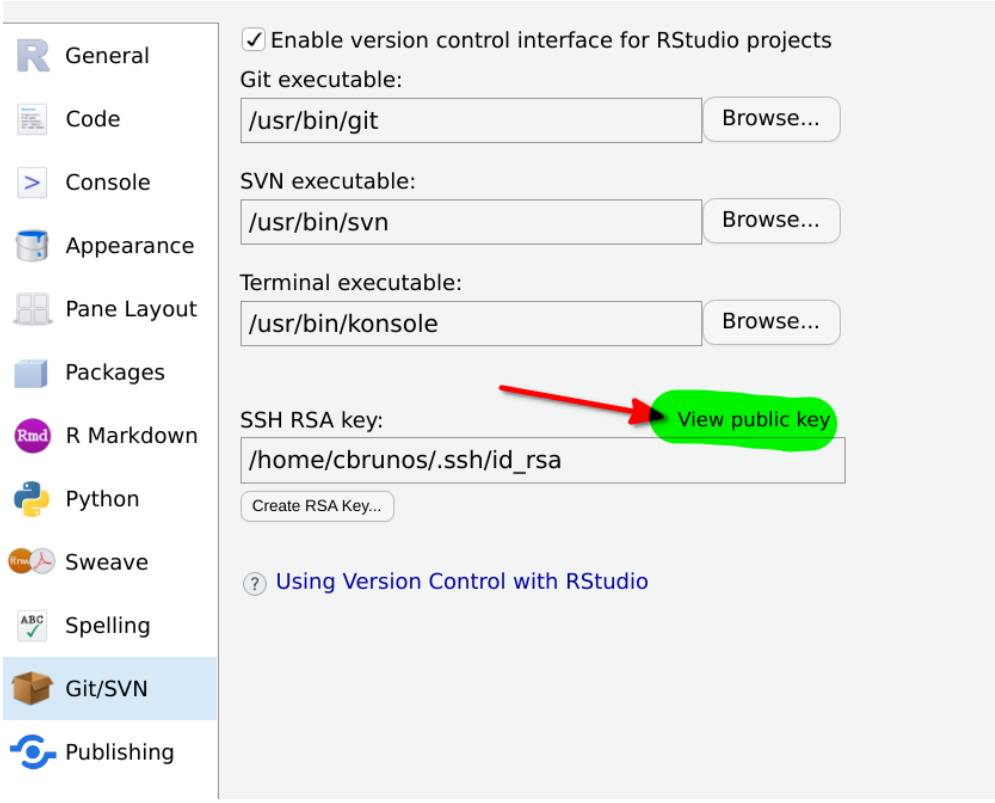



Icon (1) shows you the path where the keys will be saved. This is only useful if you have reasons to worry that your private

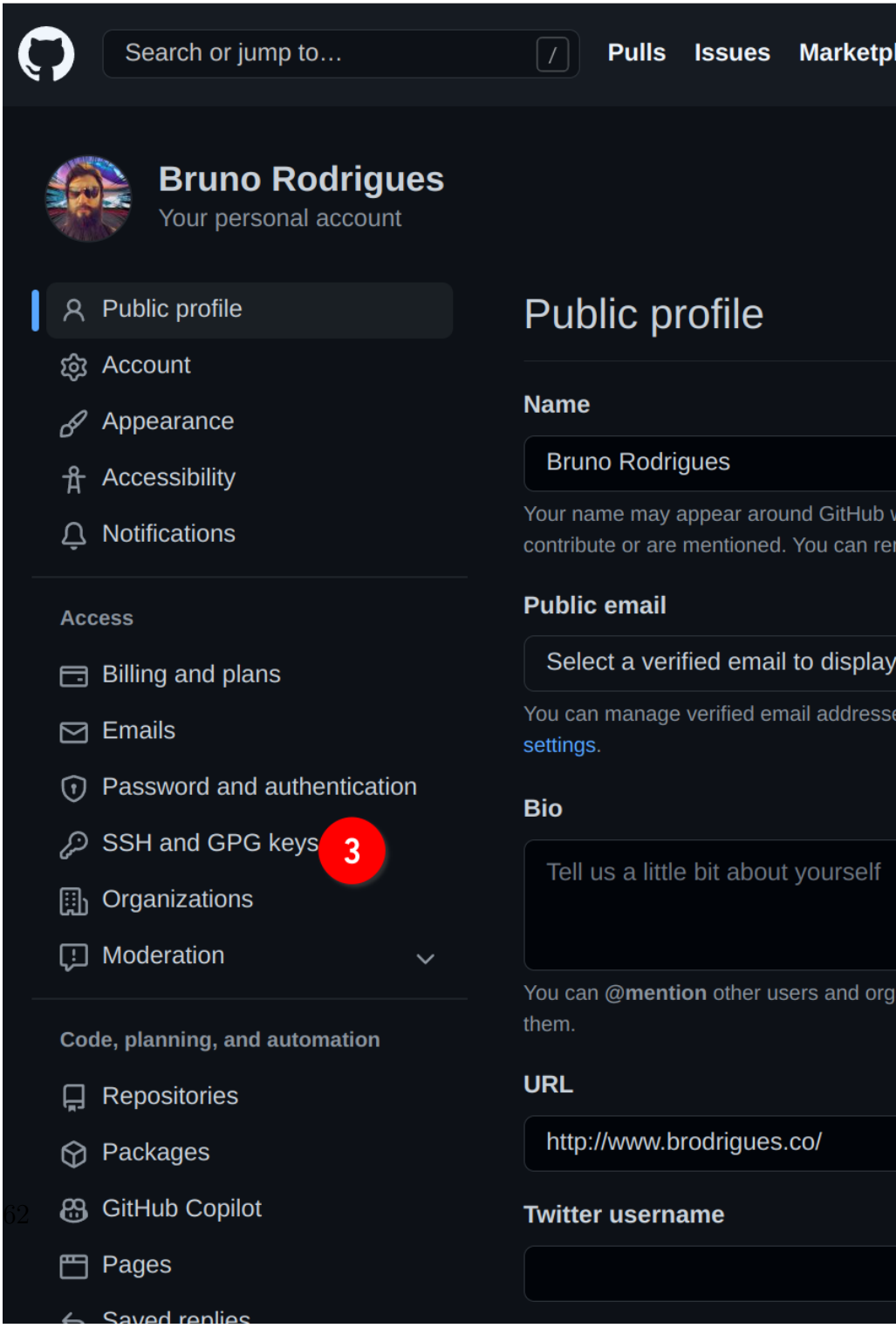
key might be compromised, but without physical access to your machine, an attacker would have a lot of trouble retrieving it (if you keep your OS updated…). Finally click on `Create`:



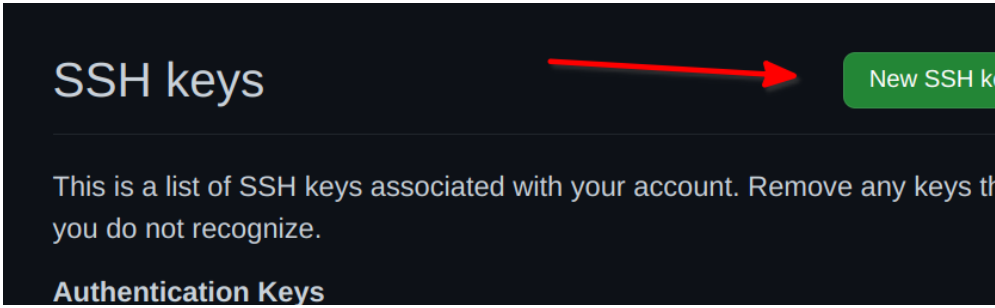

Ok so now that you have generated these keys, let's copy the public key in our clipboard (because we need to paste the key into github.com). You should be able to find this key from RStudio. Go back to `Tools > Global Options > Git/Svn`, and then click on `View public key`:



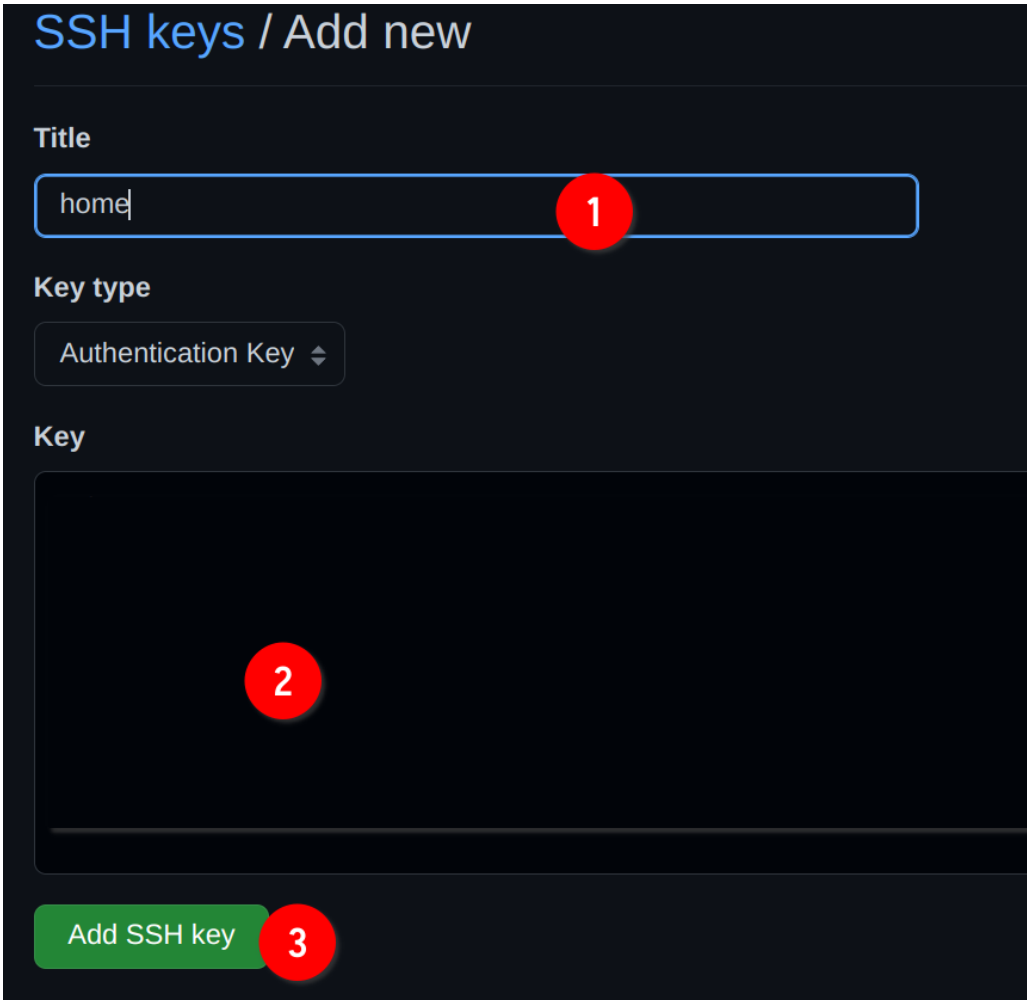

A new window will open showing you your public key. You can now copy and paste it into github.com. For this, first go to your profile, then `Settings` then `SSH and GPG keys`:


Search or jump to…
/
Pulls
Issues
Marketpl
Bruno Rodrigues
Your personal account
Public profile
Account
Appearance
Accessibility
Notifications
Access
Billing and plans
Emails
Password and authentication
SSH and GPG keys
3
Organizations
Moderation
Code, planning, and automation
Repositories
Packages
GitHub Copilot
Pages
Public profile
Name
Bruno Rodrigues
Your name may appear around GitHub
contribute or are mentioned. You can re
Public email
Select a verified email to display
You can manage verified email addresse
settings.
Bio
Tell us a little bit about yourself
You can @mention other users and org
them.
URL
http://www.brodrigues.co/
Twitter username

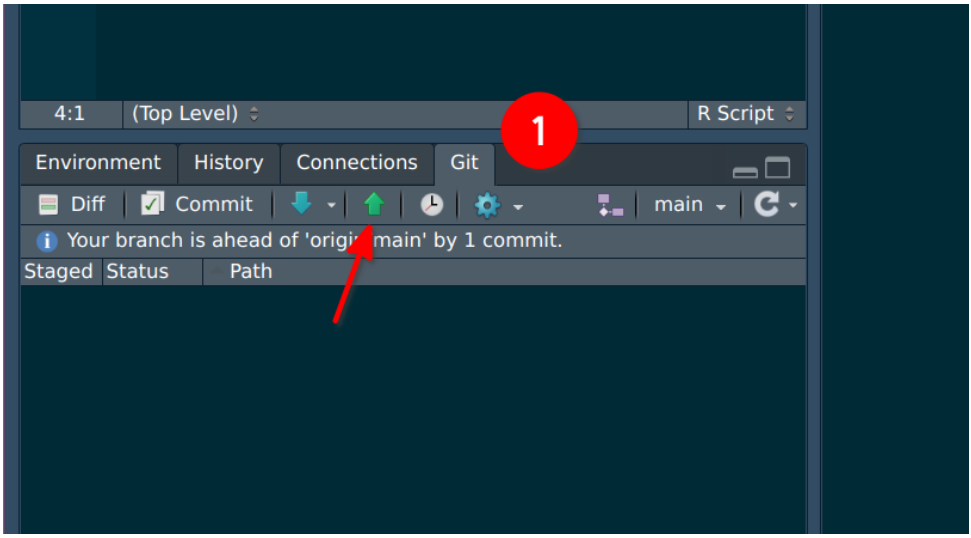
Then, on the new screen click on `New SSH key`:



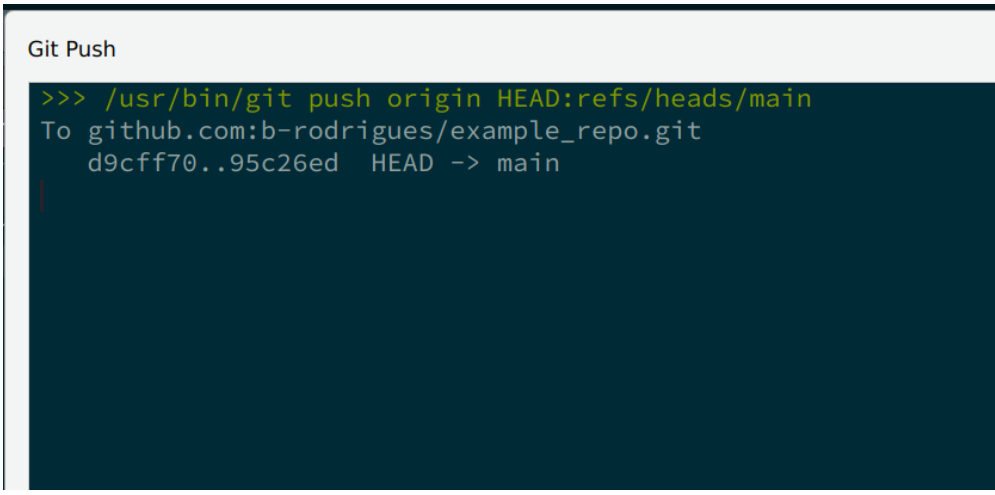

You can now add your key. Add a title, for example `home` for your home computer, or `work` for your work laptop. Paste the key from RStudio into the field (2), and then click on `Add SSH key`:




Ok, now that github.com has your public key, you can now push your commits without any error. Go back to RStudio, to the `Git` tab and click on `Push`:



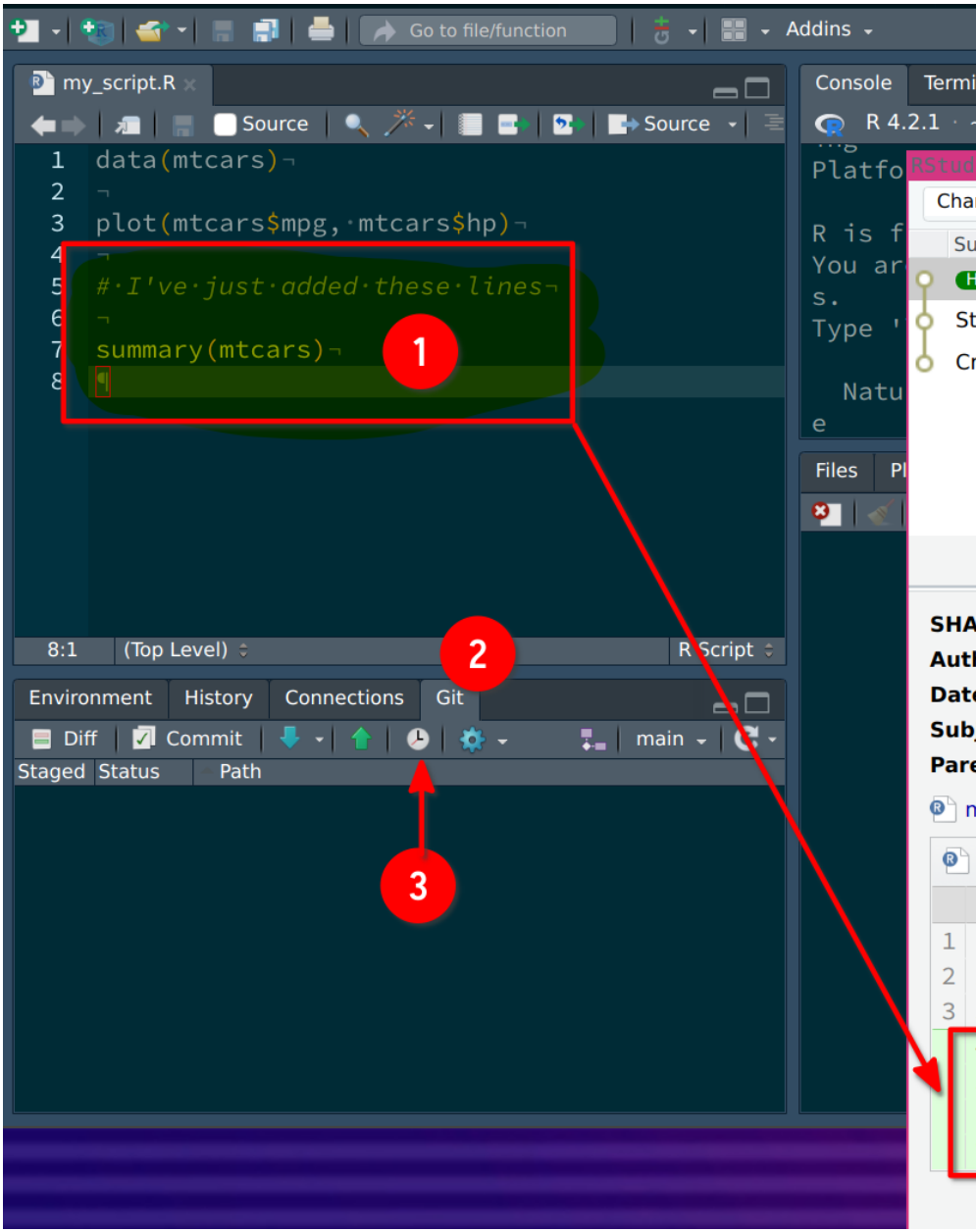

A new window will open, this time showing you that the upload went through:



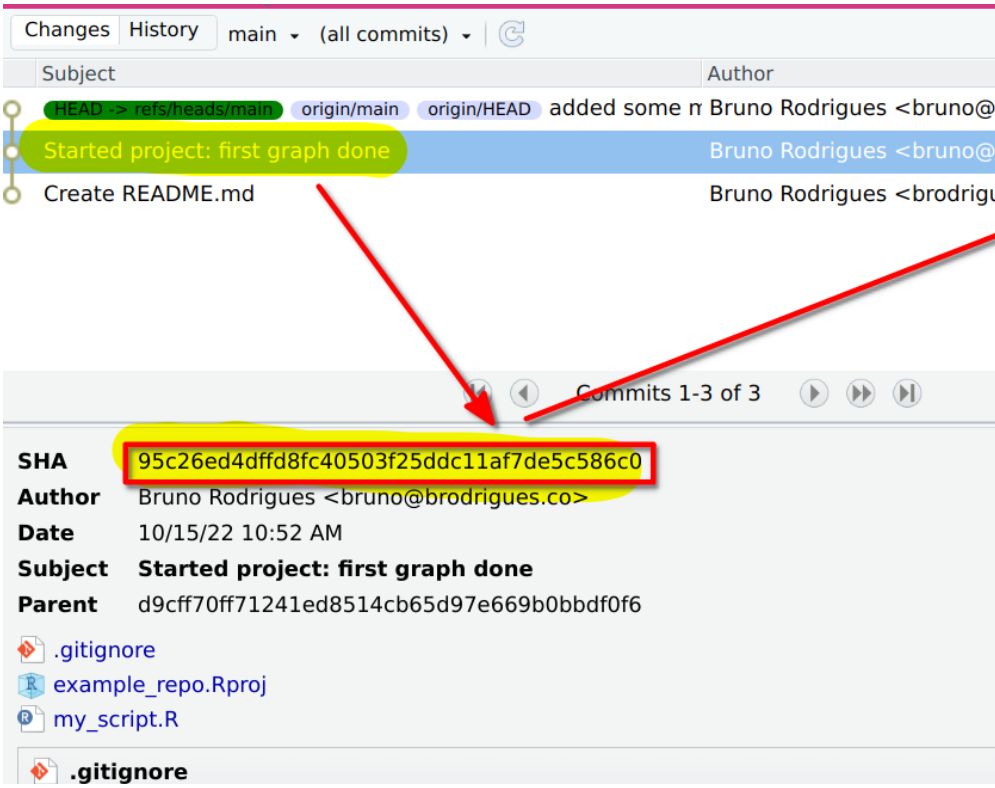

You will need to add one public key per computer you use on github.com. In the past, it was possible to push your commits by providing a password each time. This was not secure enough

however, so now the only way to to push commits is via ssh key pairs. This concept is quite important: whatever service you use, even if your company has a private Git server instance, you will need to provide the public key to the central server. All of this ssh key pair business IS NOT specific to github.com, so make sure that you understand this well, because sooner rather later, you will need to provide another public key, either because you work from several computers or because the your first job will have it's own Git instance.

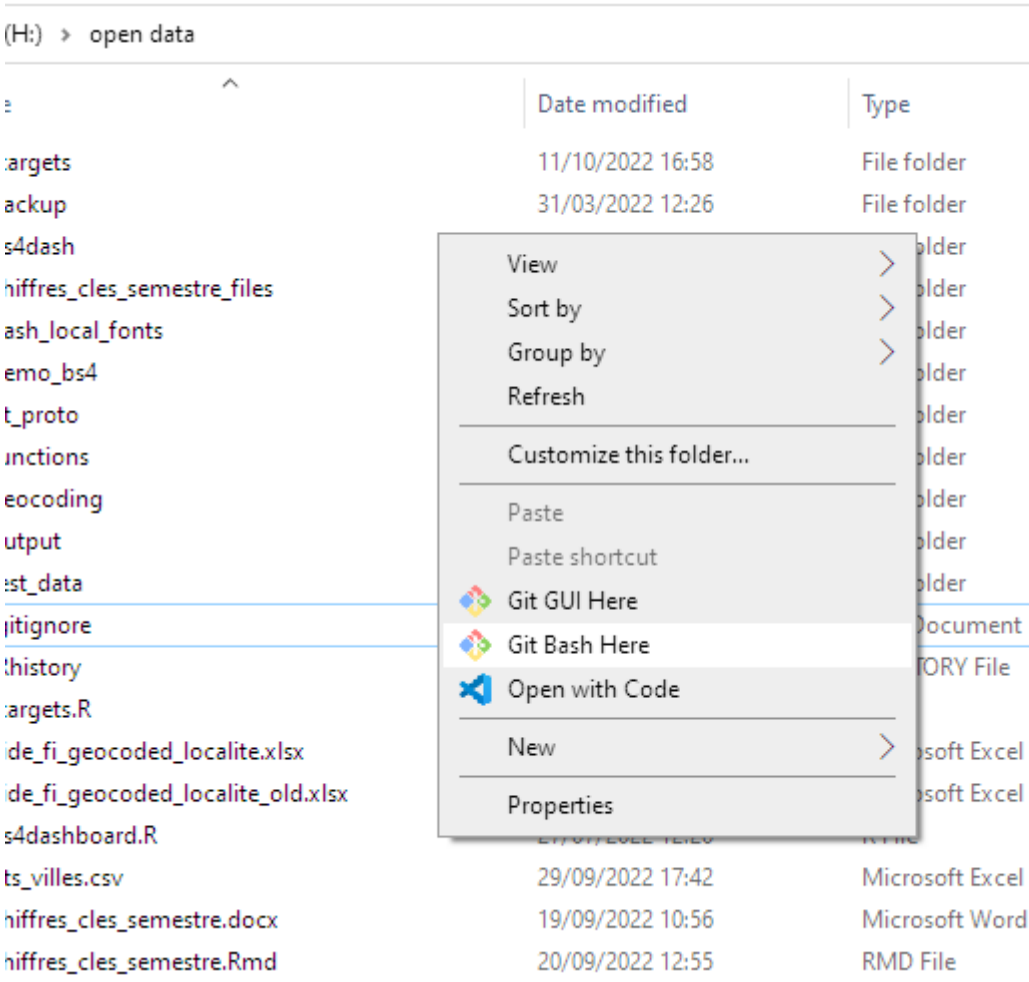
Ok so now you have an account on github.com, and know how to set up a repo and push code to it. This is already quite useful, because it allows you and future you to collaborate. What I mean by this is that if in two or three months you need to go back to some previous version of your code this is now possible. Let's try it out; change the file by adding some lines to it, commit your changes and push again. Remember to use a commit message that explain what you did. Once you're done, go back to the `Git` tab of Rstudio, and click on the `History` button (the icon is a clock):


Go to file/function
Addins
my_script.R
Source
Source
1 data(mtcars)
2
3 plot(mtcars$mpg, mtcars$hp)
4
5 # I've just added these lines
6
7 summary(mtcars)
8
1
8:1
(Top Level)
R Script
2
Environment
History
Connections
Git
Diff
Commit
main
Staged
Status
Path
3
Console
R 4.2.1

As you can see from the picture above, clicking on `History` shows every commit since the beginning of the repo. It also shows you who pushed that particular commit, and when. For now, you will only see your name. At (1) you see the lines I've added. These are reflected, in green, in the `History` window. If I had removed some lines, these would have been highlighted in red in the same window. (4) shows you the only commit history. There's not much for now, but for projects that have been ongoing for some time, this can get quite long! Finally, (5) shows many interesting details. As before, who pushed the commit, when, the commit message (under **Subject**), and finally the **SHA**. This is a unique sequence of characters that identifies the commit. If you select another commit, you will notice that its SHA is different:




The SHA identifier (called a *hash*) is what we're going to use to revert to a previous state of the code base. But because this is a bit advanced, there is no way of doing it from RStudio. You will need to open a terminal and use Git from there. On Windows, go to the folder of your project, right-click on some white space and select `Git Bash Here`:




A similar approach can be used for most Linux distributions (but simply open a terminal, Git Bash is Windows only), and you can apparently do something similar on macOS, but first need to active the required service as explained here. You can also simply open a terminal and navigate to the right folder using `cd`.[1]

[1]Remember the introduction to this book, where I discussed everything

Once the terminal is opened, follow along but by adapting the paths to your computer:

```
# The first line changes the working directory to my
 ↪  github repo on my computer
# If you did not open the terminal inside the folder
 ↪  as explained above, you need
# adapt the path.

cd ~/six_to/example_repo  # example_repo is the
 ↪  folder where I cloned the repo
ls # List all the files in the directory
```

Listing the files inside the folder confirms that I'm in the right spot. Something else you could do here is try out some git commands, for example, `git log`:

```
git log
```

```
## commit bd7daf0dafb12c0a19ba65f85b54834a02f7d150
## Author: Bruno Rodrigues <bruno@brodrigues.co>
## Date:   Mon Oct 17 14:38:59 2022 +0200
##
##     added some more lines
##
## commit 95c26ed4dffd8fc40503f25ddc11af7de5c586c0
## Author: Bruno Rodrigues <bruno@brodrigues.co>
## Date:   Sat Oct 15 12:52:43 2022 +0200
##
##     Started project: first graph done
##
```

else that you should know…

```
## commit d9cff70ff71241ed8514cb65d97e669b0bbdf0f6
## Author: Bruno Rodrigues
<brodriguesco@protonmail.com>
## Date:   Thu Oct 13 22:12:06 2022 +0200
##
##     Create README.md
```

`git log` returns the same stuff as the `History` button of the `Git` pane inside RStudio. You see the commit hash, the name of the author and when the commit was pushed. At this stage, we have two options. We could "go back in time", but just look around, and then go back to where the repository stands currently. Or we could essentially go back in time, and stay there, meaning, we actually revert the code base back. Let's try the first option, let's just take a look around at the code base at a particular point in time. Copy the hash of a previous commit. With the hash in your clipboard, use the `git checkout` command to go back to this commit:

```
git checkout 95c26ed4dffd8f
```

You will see an output similar to this:

```
Note: switching to '95c26ed4dffd8f'.

You are in 'detached HEAD' state. You can look
around, make experimental changes
and commit them, and you can discard any commits you
make in this state without
impacting any branches by switching back to a
branch.

If you want to create a new branch to retain commits
you create, you may do so
```

```
(now or later) by using -c with the switch command.
Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable
advice.detachedHead to false

HEAD is now at 95c26ed Started project: first graph
done
```

When checking out a commit, you are in *detached HEAD* state. I won't go into specifics, but what this means is that anything you do here, won't get saved, unless you specifically create a new branch for it. A Git repository is composed of branches. The branche you're currently working on should be called *main* or *master*. You can create new branches, and continue working on these other branches, without affecting the *master* branch. This allows to explore new ideas and experiment. If this turns out to be fruitful, you can merge the experimental branch back into *master*. We are not going to explore branches in this course, so you'll have to read about it on your own. But don't worry, branches are not that difficult to grok.

Take a look at the script file now, you will see that the lines you added are now missing (the following line only works on Linux, macOS, or inside a Git Bash terminal on Windows. `cat` is a command line program that prints the contents of a text file to a terminal):

```
cat my_script.R
```

Once you're done taking your tour, go back to the main (or master) branch by running:

```
git checkout main
```

Ok, so how do we actually go back to a previous state? For this, use `git revert`. But unlike `git checkout`, you don't use the hash of the commit you want to go back to. Instead, you need to use the hash of the commit you want to "cancel". For example, imagine that my commit history looks like this:

```
## commit bd7daf0dafb12c0a19ba65f85b54834a02f7d150
## Author: Bruno Rodrigues <bruno@brodrigues.co>
## Date:   Mon Oct 17 14:38:59 2022 +0200
##
##     added some more lines
##
## commit 95c26ed4dffd8fc40503f25ddc11af7de5c586c0
## Author: Bruno Rodrigues <bruno@brodrigues.co>
## Date:   Sat Oct 15 12:52:43 2022 +0200
##
##     Started project: first graph done
##
## commit d9cff70ff71241ed8514cb65d97e669b0bbdf0f6
## Author: Bruno Rodrigues
<brodriguesco@protonmail.com>
## Date:   Thu Oct 13 22:12:06 2022 +0200
##
##     Create README.md
```

and let's suppose I want to go back to commit `95c26ed4dffd8fc` (so my second commit). What I need to do is essentially cancel commit `bd7daf0dafb1`, which comes after commit `95c26ed4dffd8fc` (look at the dates: commit `95c26ed4dffd8fc` was made on October 15th and commit `bd7daf0dafb1` was made on October 17th). So I need to revert commit `bd7daf0dafb1`. And that's what we're going to do:

```
git revert bd7daf0dafb1
```

This opens a text editor inside your terminal. Here you can add a commit message or just keep the one that was added by default. Let's just keep it and quit the text editor. Unfortunately, this is not very use friendly, but to quit the editor type `:q`. (The editor that was opened is vim, a very powerful terminal editor, but with a very steep learning curve.) Now you're back inside your terminal. Type `git log` and you will see a new commit (that you have yet to push), which essentially cancels the commit `bd7daf0dafb1`. You can now push this; for pushing this one, let's stay inside the terminal and use the following command:

```
git push origin main
```

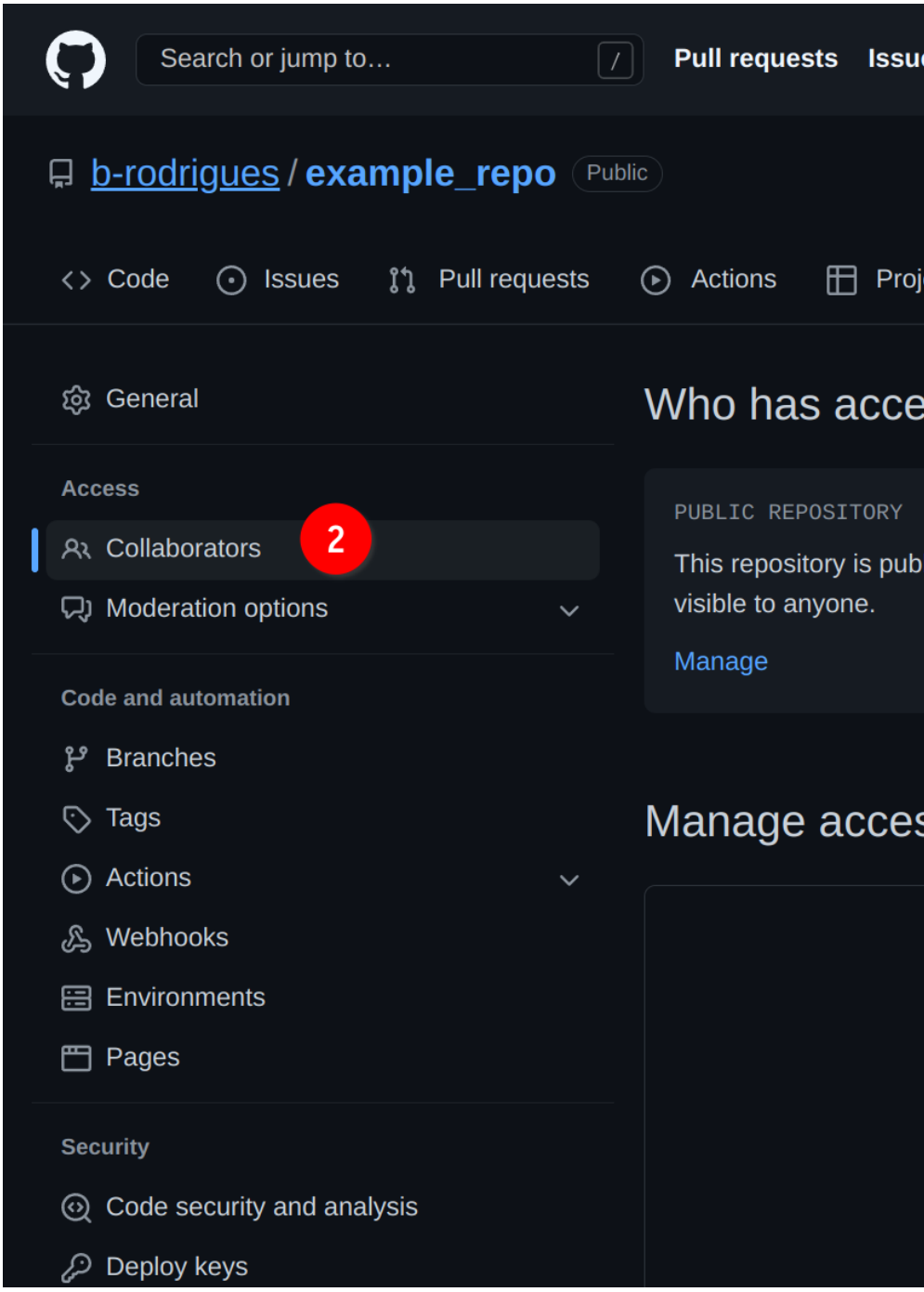
`origin main`: `origin` here refers to the remote repository, so to github.com, and `main` to the main branch.

Ok, we're doing with the basics. Let's now see how we can contribute to some repository.

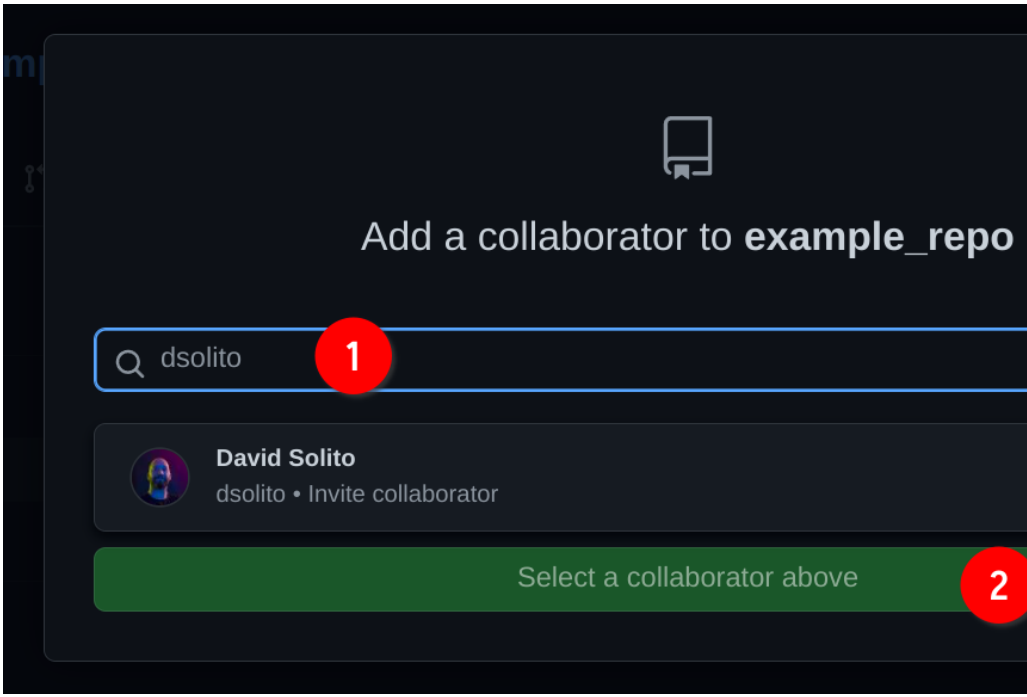
## 3.6 Collaborating

Github (and similar services) allow you to collaborate with people. There are two ways of achieving this. You can invite people to work with you on the same project, by giving them writing rights to the repository. This is what we are going to cover in this section. The other way to collaborate is to let strangers fork your repository (make a copy of it on github.com); they can then work on their copy of the project independently from you. If they want to submit patches to you, they can do so by doing a so-called *pull request*. This workflow is quite different from what we'll see here and will be discussed in the next section.

So for this section you will need to form teams of at least 2 people. One of you will invite the other to collaborate by going on github.com and then following the instructions in the picture below:


Search or jump to…
/
Pull requests
b-rodrigues / example_repo
Public
Code
Issues
Pull requests
Actions
General
Access
Collaborators
2
Moderation options
Code and automation
Branches
Tags
Actions
Webhooks
Environments
Pages
Security
Code security and analysis
Deploy keys
Who has acce
PUBLIC REPOSITORY
This repository is pub
visible to anyone.
Manage
Manage acces

Type the username of your colleague to find him/her. In my case I'm inviting my good friend David Solito:



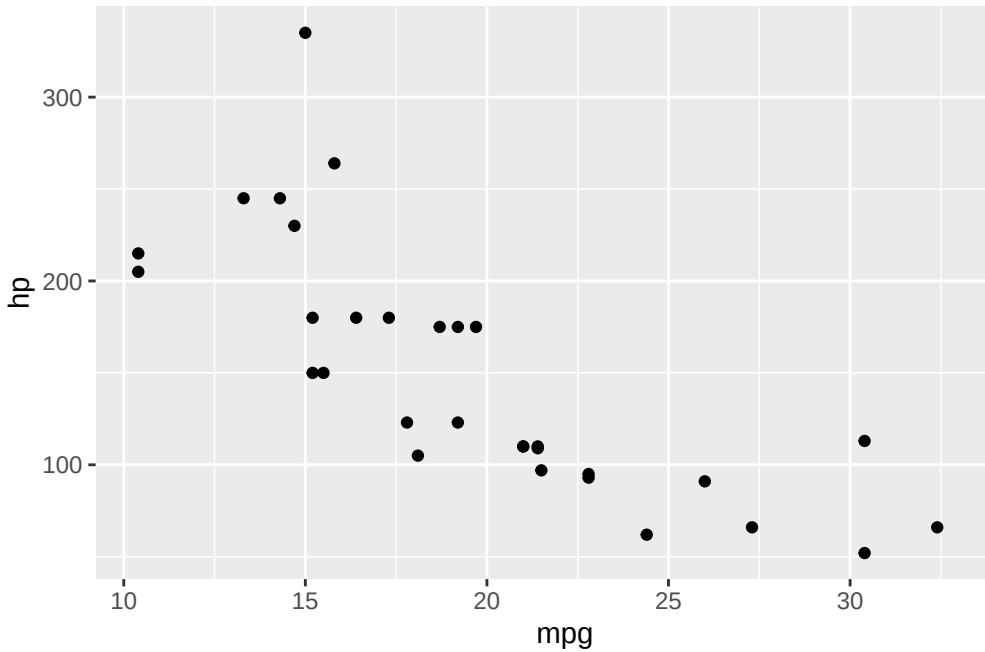

David now essentially owns the repository as well! So he can contribute to it, just like me. Now, let's suppose that I continue working on my end, and don't coordinate with David. After all, this is a post-covid world, so David might be working asynchronously from home, and maybe he lives in an entire different time zone completely! What's important to realize, is that unlike other ways of collaborating online (for example with an office suite), you do not need to coordinate to collaborate with Git.

The file should look like this (yours might be different, it doesn't matter):

```
data(mtcars)

plot(mtcars$mpg, mtcars$hp)
```

I'm going to change it to this:

```
library(ggplot2)

data(mtcars)

ggplot(data = mtcars) +
  geom_point(aes(y = hp, x = mpg))
```



The only thing I did was change from the base plotting functions to {ggplot2}. Since you guys formed groups, please work independently on the repository. Go crazy, change some lines, add lines, remove lines, or add new files with new things. Just work as normal, and commit and push your changes and see what happens.

So let's commit and push. You can do it from RStudio or from the command line/Git Bash. This is what I'll be doing from now on, but feel free to continue using Git through RStudio:

```
git add . # This adds every file I've changed to
 ↪  this next commit
git commit -am "Remade plot with ggplot2" # git
 ↪  commit is the command to create the commit. The
 ↪  -am flag means: 'a' stands for all, as in
 ↪  'adding all files to the commit', so it's
 ↪  actually redundant with the previous line, but I
 ↪  use it out of habit, and 'm' specifies that we
 ↪  want to add a message
git push origin main # This pushes the commit to the
 ↪  repository on github.com
```

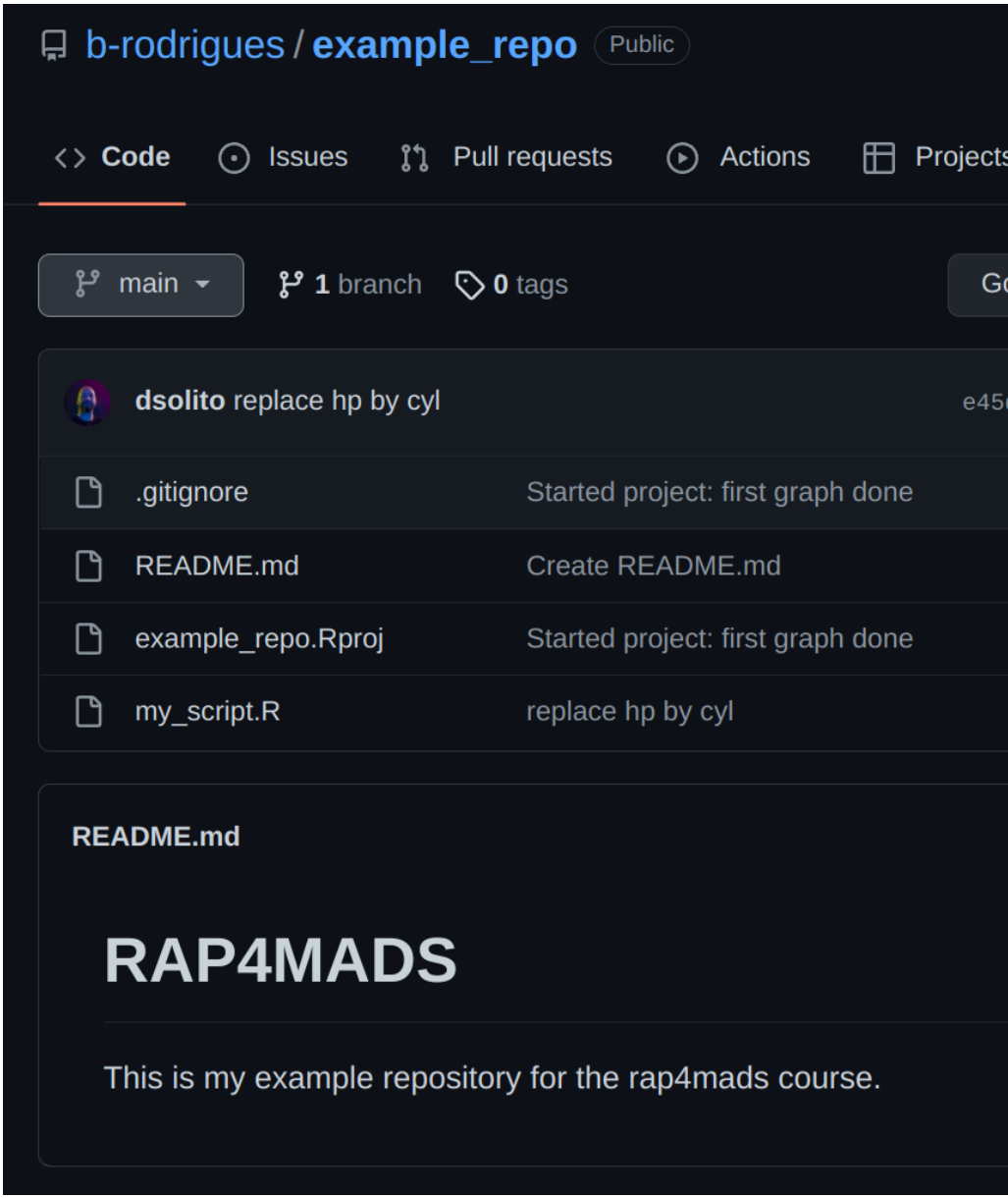
And this is what happens:

```
  git push origin main

    To github.com:b-rodrigues/example_repo.git
     ! [rejected]        main -> main (fetch first)
    error: failed to push some refs to
    'github.com:b-rodrigues/example_repo.git'
    hint: Updates were rejected because the remote
    contains work that you do
```

```
hint: not have locally. This is usually caused by
another repository pushing
hint: to the same ref. You may want to first
integrate the remote changes
hint: (e.g., 'git pull ...') before pushing
again.
hint: See the 'Note about fast-forwards' in 'git
push --help' for details.
```

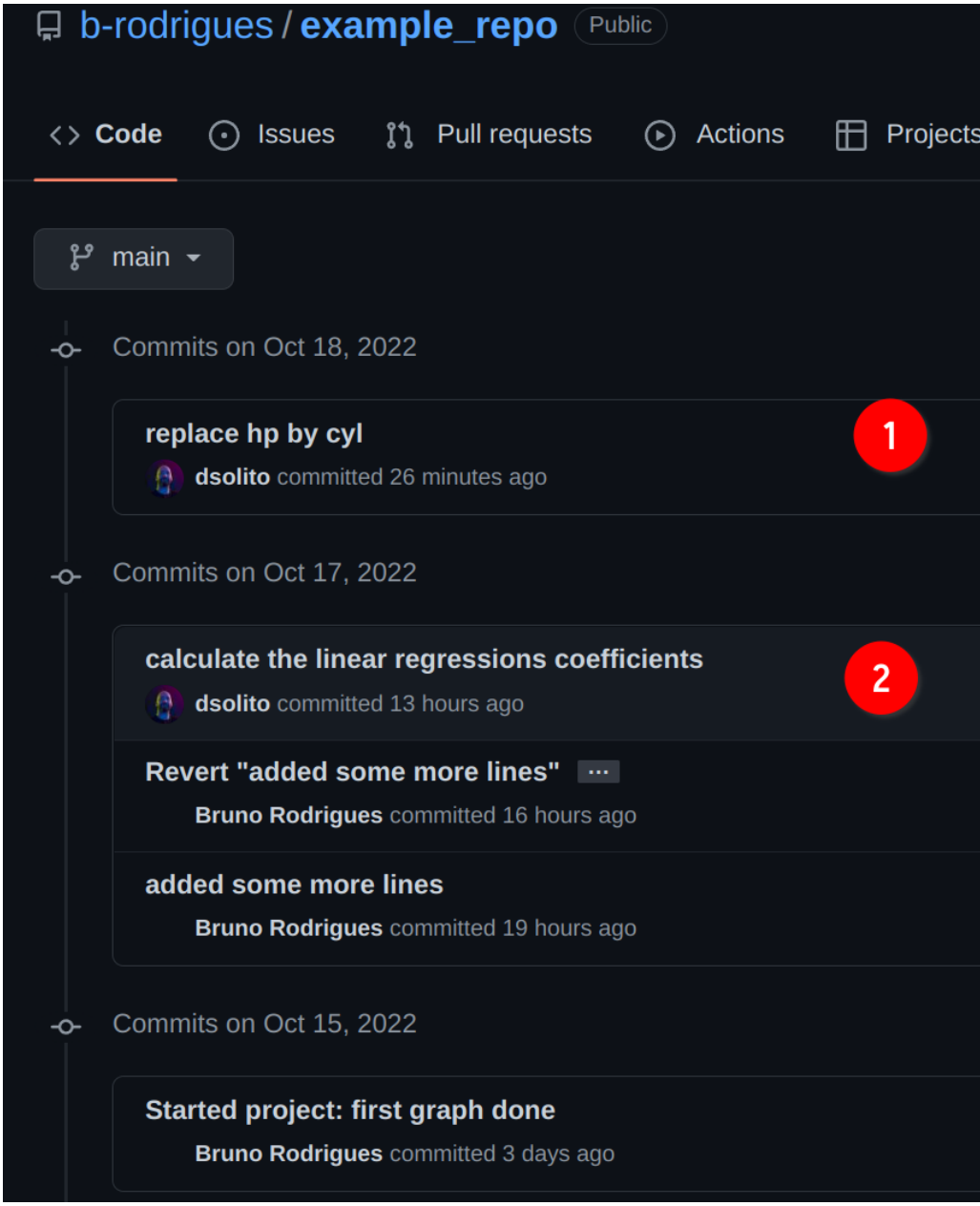
What this all means is that David already pushed some changes while I was working on the project as well. It says so very cleary *Updates were rejected because the remote contains work that you do not have locally.* Git tells us that we first need to pull (download, if you will) the changes to our own computer to integrate the changes, and then we can push again.

At this point, if we want, we can first go to github.com and see the commit history there to see what David did. Go to your repo, and click on the commit history icon:




Doing so will list the commit history, as currently on

github.com:


b-rodrigues / example_repo
Public
Code
Issues
Pull requests
Actions
Projects
main
Commits on Oct 18, 2022
replace hp by cyl
dsolito committed 26 minutes ago
1
Commits on Oct 17, 2022
calculate the linear regressions coefficients
dsolito committed 13 hours ago
2
Revert "added some more lines"
Bruno Rodrigues committed 16 hours ago
added some more lines
Bruno Rodrigues committed 19 hours ago
Commits on Oct 15, 2022
Started project: first graph done
Bruno Rodrigues committed 3 days ago

While I was working, David pushed 2 commits to the repository. If you compare to your local history, using `git log` you will see that these commits are not there, but instead, however many commits you did (this will not be the case for all of you; whoever of you pushed first will not see any difference between the local and remote repository). Let's see how it looks for me:

```
git log
```

```
commit d2ab909fc679a5661fc3c49c7ac549a2764c539e
(HEAD -> main)
Author: Bruno Rodrigues <bruno@brodrigues.co>
Date:   Tue Oct 18 09:28:10 2022 +0200

    Remade plot with ggplot2

commit e66c68cc8b58831004d1c9433b2223503d718e1c
(origin/main, origin/HEAD)
Author: Bruno Rodrigues <bruno@brodrigues.co>
Date:   Mon Oct 17 17:33:33 2022 +0200

    Revert "added some more lines"

    This reverts commit
    bd7daf0dafb12c0a19ba65f85b54834a02f7d150.

commit bd7daf0dafb12c0a19ba65f85b54834a02f7d150
Author: Bruno Rodrigues <bruno@brodrigues.co>
Date:   Mon Oct 17 14:38:59 2022 +0200

    added some more lines

commit 95c26ed4dffd8fc40503f25ddc11af7de5c586c0
```

```
Author: Bruno Rodrigues <bruno@brodrigues.co>
Date:   Sat Oct 15 12:52:43 2022 +0200

    Started project: first graph done

commit d9cff70ff71241ed8514cb65d97e669b0bbdf0f6
Author: Bruno Rodrigues
<brodriguesco@protonmail.com>
Date:   Thu Oct 13 22:12:06 2022 +0200

    Create README.md
```

Yep, so none of David's commits in sight. Let me do what Git told me to do: let's pull, or download, David's commits locally:

```
git pull --rebase
```

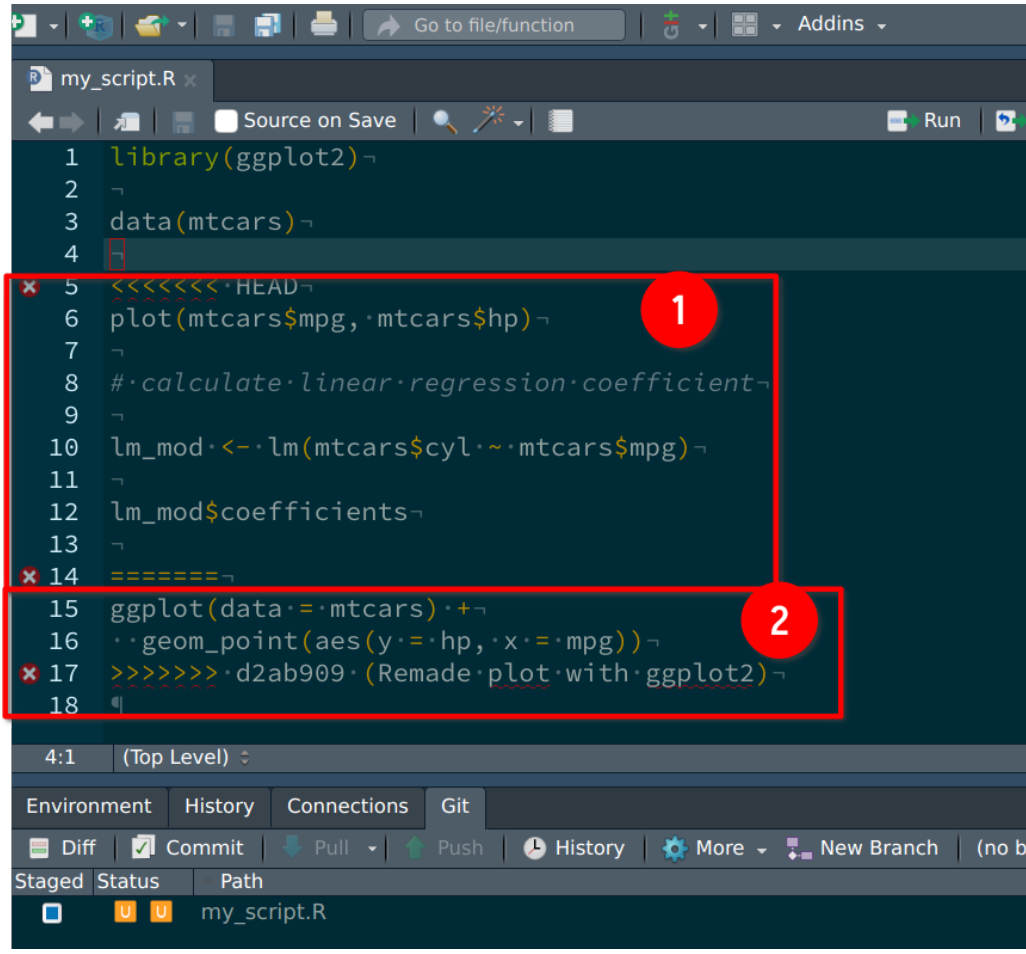
`--rebase` is a flag that keeps the commit history linear. There are many different ways you can pull changes, but for our purposes we can focus on `--rebase`. The other strategies are more advanced, and you might want at some point to take a look at them.

Once `git pull --rebase` is done, we get the following message:

```
Auto-merging my_script.R
CONFLICT (content): Merge conflict in my_script.R
error: could not apply d2ab909... Remade plot with
ggplot2
hint: Resolve all conflicts manually, mark them as
resolved with
```

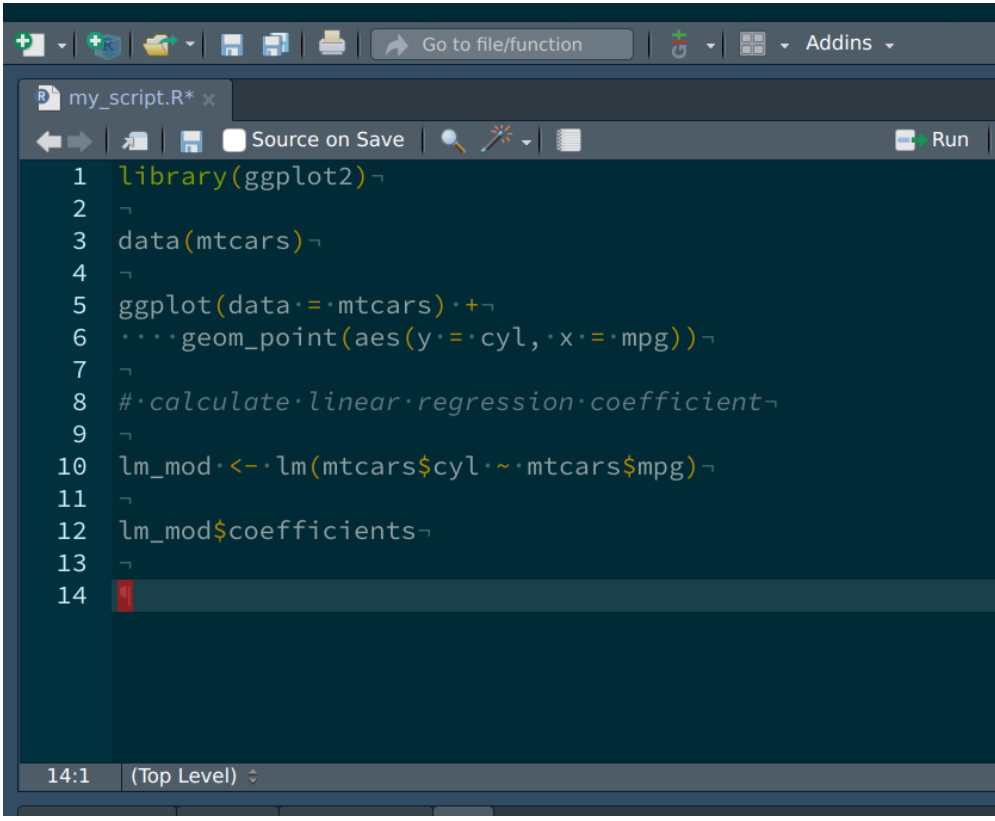
```
hint: "git add/rm <conflicted_files>", then run "git
rebase --continue".
hint: You can instead skip this commit: run "git
rebase --skip".
hint: To abort and get back to the state before "git
rebase", run "git rebase --abort".
Could not apply d2ab909... Remade plot with ggplot2
```

Once again, it is important to read what Git is telling us. There is a merge conflict in the `my_script.R` file. Let's open it, and see what's going on:




We can see two things: the lines that David changed in (1), and the lines I've added in (2). This happened because we changed the same lines. Had I added lines instead of changing lines that were already there, the merge would have happened automatically, because there would not have been any conflict. In this case however, Git does not know how to solve the issue: do we keep David's changes, or mine? Actually, we need to keep both. I'll keep my version of plot that uses `{ggplot2}`, but will also

keep what David added: he replaced the `hp` variable by `cyl`, and added a linear regression as well. Since this seems sensible to me, I will adapt the script in a way that gracefully merges both contributions. So the file looks like this now:



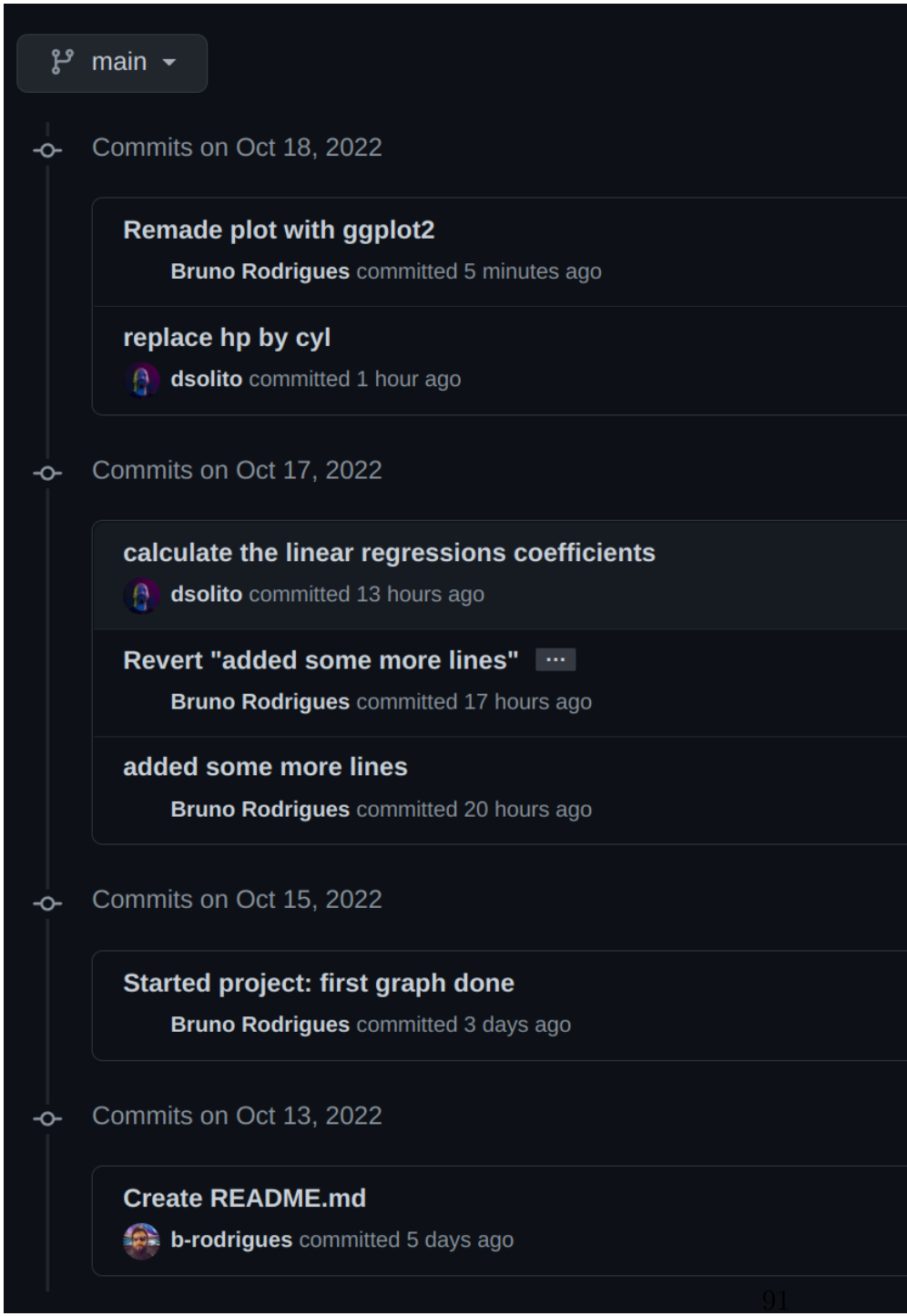

We can now save, and continue following the hints from Git, namely, adding the changed file to the next commit and then use `git rebase --continue`:

```
git add my_script.R
git rebase --continue
```

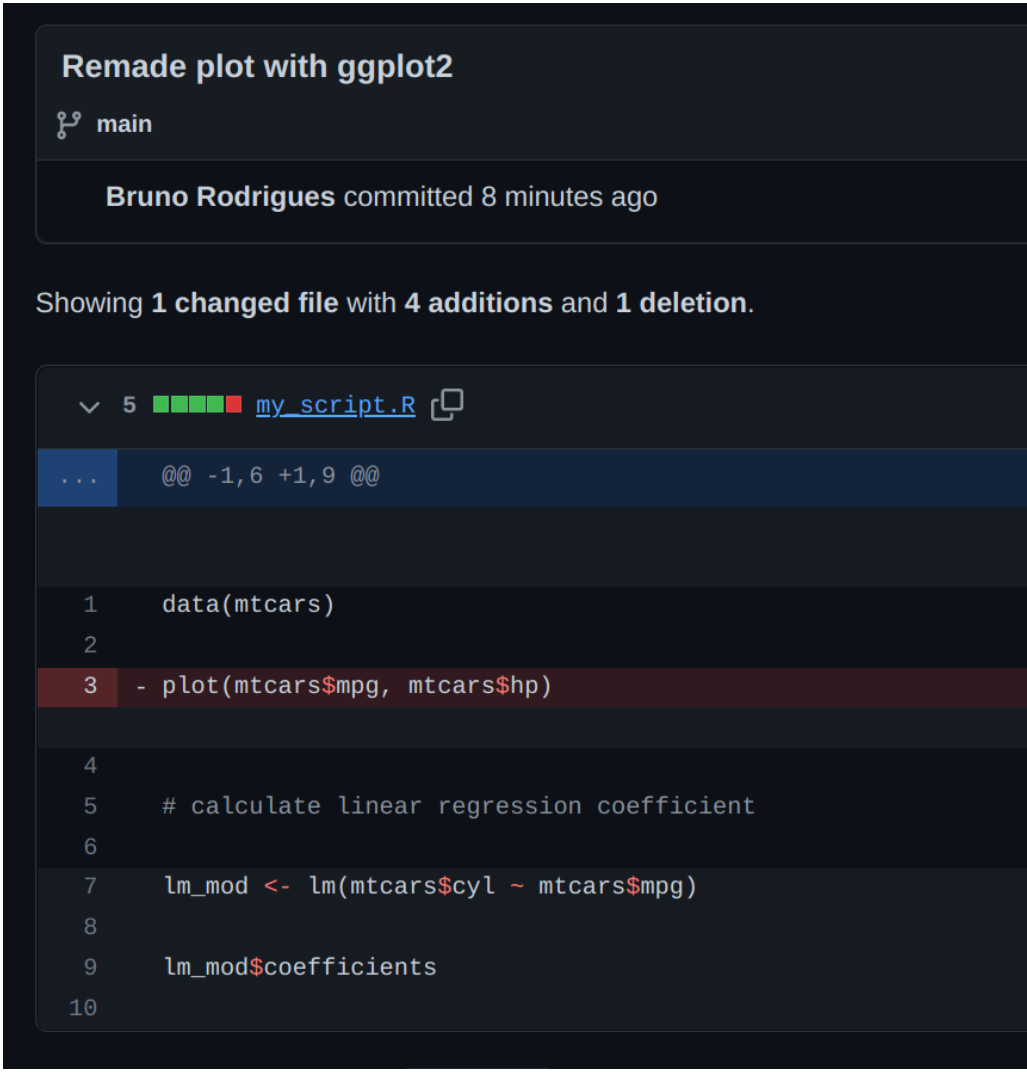
This will once again open the editor in your terminal. Simply close it with `:q`. Let's now push:

```
git push origin main
```

and we're done! Let's go back to github.com to see the commit history. You can click on the hash to see the details of how the file changed (you can do so from RStudio as well):


main
Commits on Oct 18, 2022
Remade plot with ggplot2
Bruno Rodrigues committed 5 minutes ago
replace hp by cyl
dsolito committed 1 hour ago
Commits on Oct 17, 2022
calculate the linear regressions coefficients
dsolito committed 13 hours ago
Revert "added some more lines"
Bruno Rodrigues committed 17 hours ago
added some more lines
Bruno Rodrigues committed 20 hours ago
Commits on Oct 15, 2022
Started project: first graph done
Bruno Rodrigues committed 3 days ago
Commits on Oct 13, 2022
Create README.md
b-rodrigues committed 5 days ago

In green, you see lines that were added, and in red, lines that were removed. The lines where the linear model was defined are not impacted, because David wrote them at the bottom of the script, and I did not write anything there:



## 3.7 Branches

It is possible to create new branches and continue working on these branches without impacting the code in the main branch. This is useful if you want to experiment and explore new ideas. The `main` or `master` branch can thus be used only to have code that is ready to get shipped and distributed, while you can keep working on a development branch. Let's create a branch called `dev` by using the `git checkout` command, and let's also add the `-b` flag to immediately switch to it:

```
  git checkout -b dev
Switched to a new branch 'dev'
```

It is possible to list the existing branches using `git branch`:

```
  git branch
* dev
main
```

As a little aside, if you're working inside a terminal instead of RStudio or another GUI application, it might be a good idea to configure your terminal a little bit to do two things:

- change the branch you're currently on
- show if some files got changed.

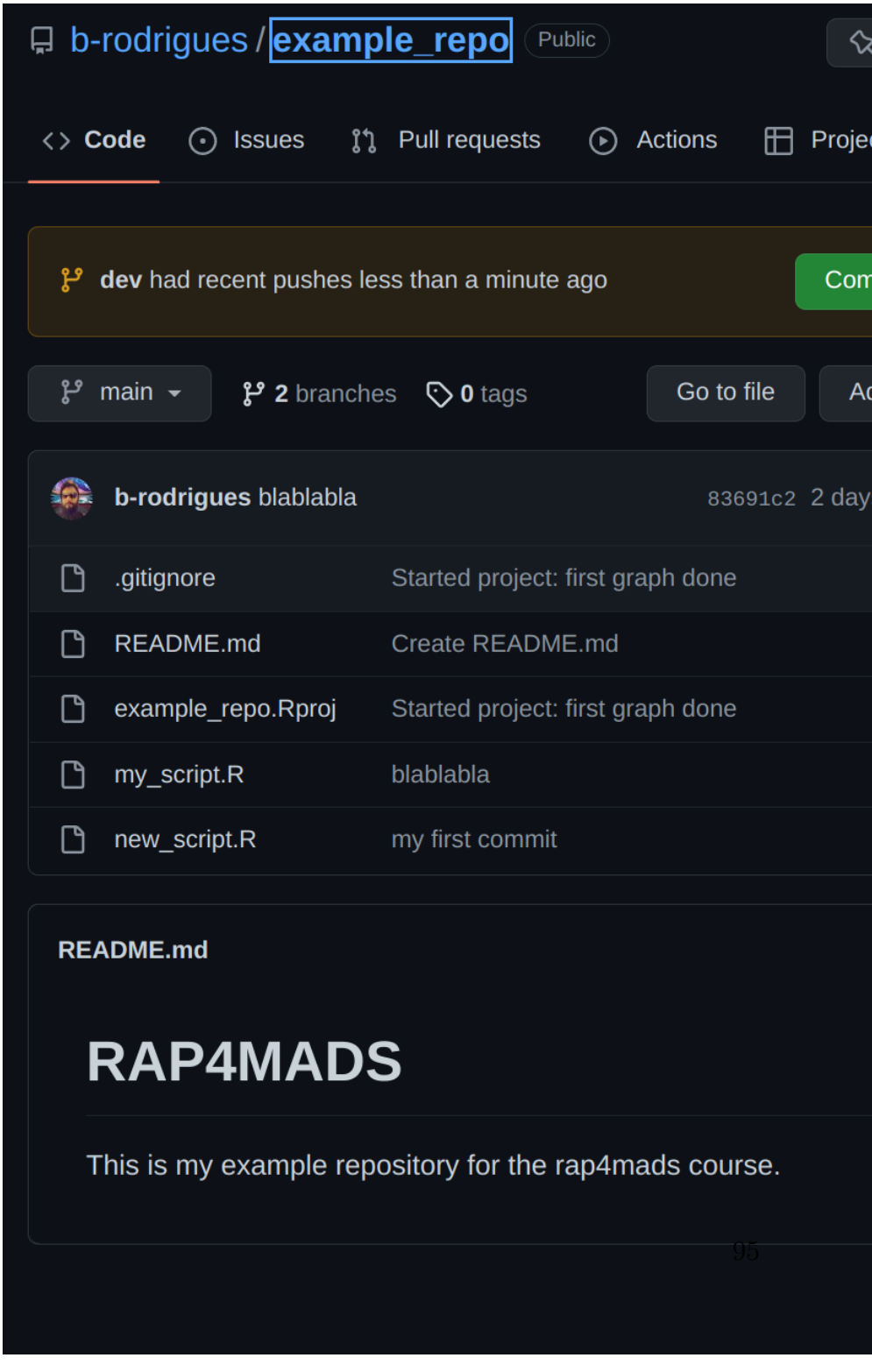
If you want to keep it simple, following this tutorial should be enough. If you want something more fancy, use this other tutorial. I have not followed either, so I don't know if they work, but by the looks of it they should, and it should work on both Linux and macOS I believe. If these don't work, just google for

"showing git branch in terminal". This is entirely optional, and you can use `git branch` to check which branch you're currently working on.

Ok so now that we are on the `dev` branch, let's change the files a little bit. Change some lines, then commit, then add some new files and commit again. Then push to `dev` using:

```
git push origin dev
```

This is what you should see on github.com after all is done:


b-rodrigues / example_repo
Public
Code
Issues
Pull requests
Actions
dev had recent pushes less than a minute ago
main
2 branches
0 tags
Go to file
b-rodrigues blablabla
83691c2
2 day
.gitignore
Started project: first graph done
README.md
Create README.md
example_repo.Rproj
Started project: first graph done
my_script.R
blablabla
new_script.R
my first commit
README.md
RAP4MADS
This is my example repository for the rap4mads course.

The video below shows you how you can switch between branches and check the commit history of both:

Let's suppose that we are happy with our experiments on the `dev` branch, and are ready to add them to the `master` or `main` branch. For this, checkout the main branch:

```
 git checkout main
```

You can now pull from `dev`. This will update your local `main` branch with the changes from `dev`. Depending on what changes you introduced, you might need to solve some conflicts. Try to use the `rebase` strategy, and then solve the conflict. In my case, the merge didn't cause an issue:

```
 git pull origin dev
From github.com:b-rodrigues/example_repo
 * branch            dev        -> FETCH_HEAD
Updating a9a417f..8b2f04f
Fast-forward
 my_script.R  | 8 ++++-----
 new_script.R | 1 +
 2 files changed, 4 insertions(+), 5 deletions(-)
 create mode 100644 new_script.R
```

Now if you run git status, this is what you'll see:

```
 git status
On branch main
Your branch and 'origin/main' have diverged,
and have 2 and 2 different commits each,
 ↪  respectively.
  (use "git pull" to merge the remote branch into
   ↪  yours)
```

Now, remember that I've pulled from `dev` into `main`. But `git status` complains that the remote `main` and local `main` branches have diverged. In these situations, git suggests to pull. This time we're pulling from `main`:

```
git pull
```

This will likely result in the following message:

```
hint: You have divergent branches and need to
 ↪  specify how to reconcile them.
hint: You can do so by running one of the following
 ↪  commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward
 ↪  only
hint:
hint: You can replace "git config" with "git config
 ↪  --global" to set a default
hint: preference for all repositories. You can also
 ↪  pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override
 ↪  the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent
 ↪  branches.
```

Because there are conflicts, I need to specify how the pulling should be done. For this, I'm using once again the `rebase` flag:

```
 git pull --rebase
Auto-merging my_script.R
CONFLICT (content): Merge conflict in my_script.R
error: could not apply b240566... lm -> rf
hint: Resolve all conflicts manually, mark them as
 ↪  resolved with
hint: "git add/rm <conflicted_files>", then run "git
 ↪  rebase --continue".
hint: You can instead skip this commit: run "git
 ↪  rebase --skip".
hint: To abort and get back to the state before "git
 ↪  rebase", run "git rebase --abort".
Could not apply b240566... lm -> rf
```

So now I have conflicts. This is how the `my_script.R` file looks like:

```
library(ggplot2)
library(randomForest)

data(mtcars)

ggplot(data = mtcars) +
  geom_point(aes(y = cyl, x = mpg))

rf <- randomForest(hp ~ mpg, data = mtcars)

<<<<<<< HEAD
data(iris)
```

```
head(iris)
=======
plot(rf)
>>>>>>> b240566 (lm -> rf)
```

I need to solve the conflicts, and will do so by keeping the following lines:

```
library(ggplot2)
library(randomForest)

data(mtcars)

ggplot(data = mtcars) +
  geom_point(aes(y = cyl, x = mpg))

rf <- randomForest(hp ~ mpg, data = mtcars)

plot(rf)
```

Let's save the script, and call `git rebase --continue`. You might see something like this:

```
 git rebase --continue
[detached HEAD 929f4ab] lm -> rf
 1 file changed, 4 insertions(+), 9 deletions(-)
Auto-merging new_script.R
CONFLICT (add/add): Merge conflict in new_script.R
error: could not apply 8b2f04f... new file
```

There's another conflict: this time, this is because of the commit `8b2f04f`, where I added a new file. This one is easy to solve: I

simply want to keep this file, so I simply keep track of it with `git add new_script.R` and then, once again, call `git rebase --continue`:

```
 git rebase --continue
[detached HEAD 20c04f8] new file
 1 file changed, 4 insertions(+)
Successfully rebased and updated refs/heads/main.
```

I'm now done and can push to main:

```
 git push origin main
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 12 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 660 bytes | 660.00
 ↪  KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with
 ↪  1 local object.
To github.com:b-rodrigues/example_repo.git
   83691c2..20c04f8  main -> main
```

There are other ways to achieve this. So let's go back to `dev` and continue working:

```
 git checkout dev
```

Add some lines to `my_script.R` and then commit and push:

```
 git add .
 git commit -am "more models"
[dev a0fa9fa] more models
 1 file changed, 4 insertions(+)

 git push origin dev
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 329 bytes | 329.00
 ↪  KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with
 ↪  2 local objects.
To github.com:b-rodrigues/example_repo.git
   8b2f04f..a0fa9fa  dev -> dev
```

Let's suppose we're done with adding features to `dev`. Let's checkout `main`:

```
 git checkout main
```

and now, let's not pull from `dev`, but merge:

```
 git merge dev
Auto-merging my_script.R
CONFLICT (content): Merge conflict in my_script.R
Auto-merging new_script.R
CONFLICT (add/add): Merge conflict in new_script.R
Automatic merge failed; fix conflicts and then
 ↪  commit the result.
```

Some conflicts are in the file. Let's take a look (because I'm in the terminal, I use `cat` to print the file to the terminal, but you can open it in RStudio):

```
  cat my_script.R
library(ggplot2)
library(randomForest)

data(mtcars)

ggplot(data = mtcars) +
  geom_point(aes(y = cyl, x = mpg))

rf <- randomForest(hp ~ mpg, data = mtcars)
<<<<<<< HEAD

plot(rf)
=======

plot(rf)

rf2 <- randomForest(hp ~ mpg + am + cyl, data =
 ↪  mtcars)

plot(rf2)
>>>>>>> dev
```

Looks like I somehow added some newline somewhere and this caused the conflict. This is quite easy to solve, let's make the script look like this:

```
library(ggplot2)
library(randomForest)

data(mtcars)

ggplot(data = mtcars) +
  geom_point(aes(y = cyl, x = mpg))

rf <- randomForest(hp ~ mpg, data = mtcars)

plot(rf)

rf2 <- randomForest(hp ~ mpg + am + cyl, data =
 ↪  mtcars)

plot(rf2)
```

We can now simply commit and push. Merging can be simpler than pulling and rebasing, especially if you exclusively worked on `dev` and master has not seen any activity.

## 3.8 Contributing to someone else's repository

It is also possible to contribute to someone else's repository; by this I mean someone who is not a colleague, and who did not invite you to his or her repository. So this means that you do not have writing rights to the repository and cannot push to it.

This is outside the scope of this course, but it is crucial that you understand this as well. For this reason, I highly recommend reading this link.

Ok, so this wraps up this chapter. Git is incredibly feature rich and complex, but as already discussed, it is NOT optional to know about Git in our trade. So now that you have some understanding of how it works, I suggest that you read the manual here. W3Schools has a great tutorial as well.

# 4 Package development

What you'll have learned by the end of the chapter: building and documenting your own package.

## 4.1 Introduction

In this chapter we're going to develop our own package. This package will contain some functions that we will write to analyze some data. Don't focus too much on what these functions do or don't do, that's not really important. What matters is that you understand how to build your own package, and why that's useful.

R, as you know, has many many packages. When you type something like

```
install.packages("dplyr")
```

This installs the `{dplyr}` package. The package gets downloaded from a repository called CRAN - The Comprehensive R Archive Network (or from one of its mirrors). Developers thus work on their packages and once they feel the package is ready for production they submit it to CRAN. There are very strict rules to respect to publish on CRAN; but if the developers respects these rules, and the package does something *non-trivial* (non-trivial is not really defined but the idea is that your package cannot simply be a collection of your own implementation of common mathematical functions for example), it'll get published.

CRAN is actually quite a miracle; it works really well, and it's been working well for decades, since CRAN was founded in 1997. Installing packages on R is rarely frustrating, and when it is, it

is rarely, if ever, CRAN's fault (there are some packages that require your operating system to have certain libraries or programs installed beforehand, and these can be frustrating to install, like `java` or certain libraries used for geospatial statistics).

But while from the point of view from the user, CRAN is great, there are sometimes some frictions between package developers and CRAN maintainers. I'll spare you the drama, but just know that contributing to CRAN can be sometimes frustrating.

This does not concern us however, because we are going to learn how to develop a package but we are not going to publish it on CRAN. Instead, we will be using github.com as a replacement for CRAN. This has the advantage that we do not have to be so strict and disciplined when writing our package, and other users can install the package almost just as easily from github.com, with the following command:

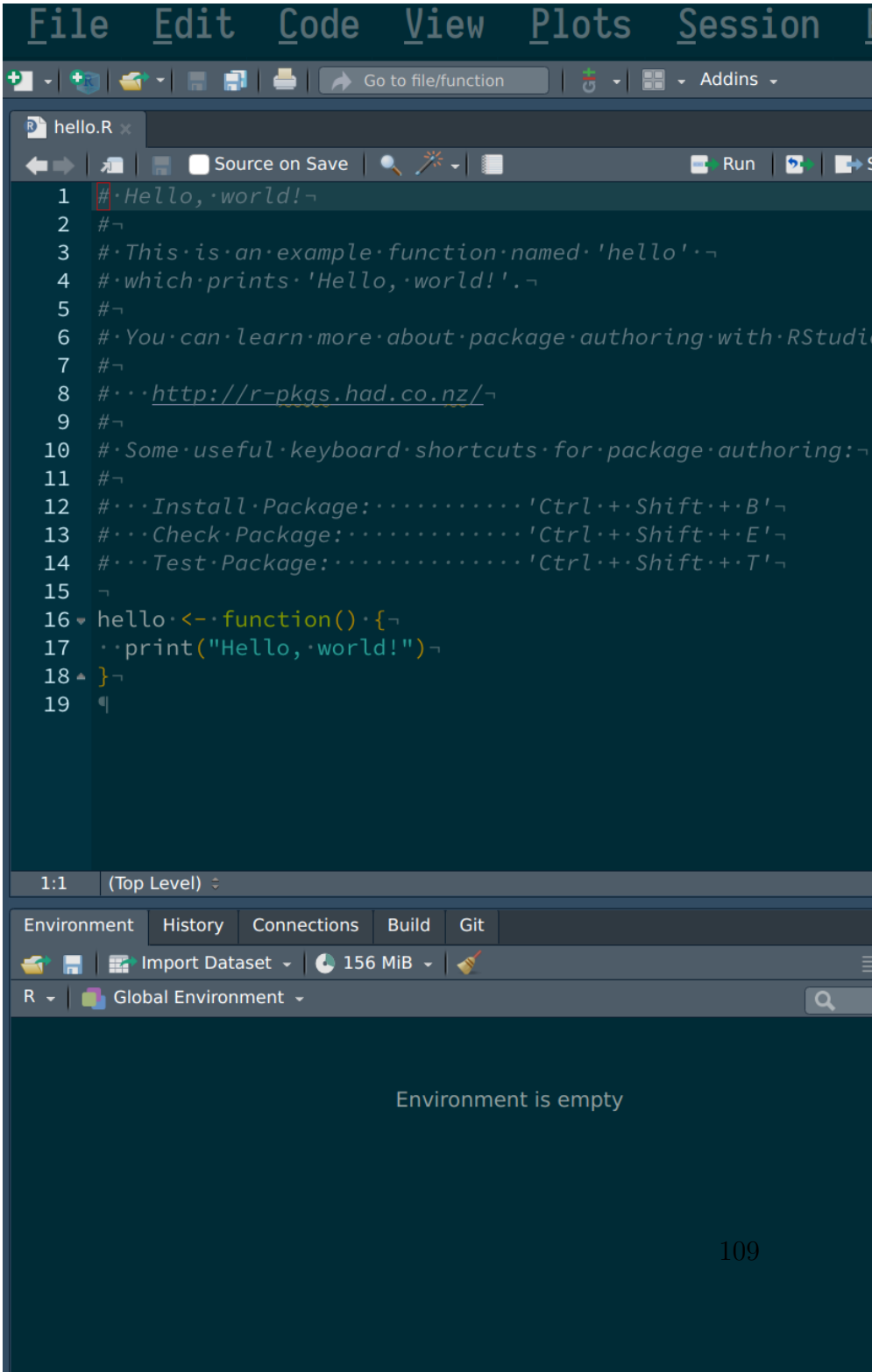
```
remotes::install_github("github_username/some_package")
```

It is also possible to build the package and send it as a file per email for example, and then install a local copy. This is more cumbersome, that's why we're going to use github.com as a repository.

## 4.2 Getting started

Let's first start by opening RStudio, and start a new project:

Following these steps creates a folder in the specified path that already contains some scaffolding for our package. This also opens a new RStudio session with the default script `hello.R` opened:

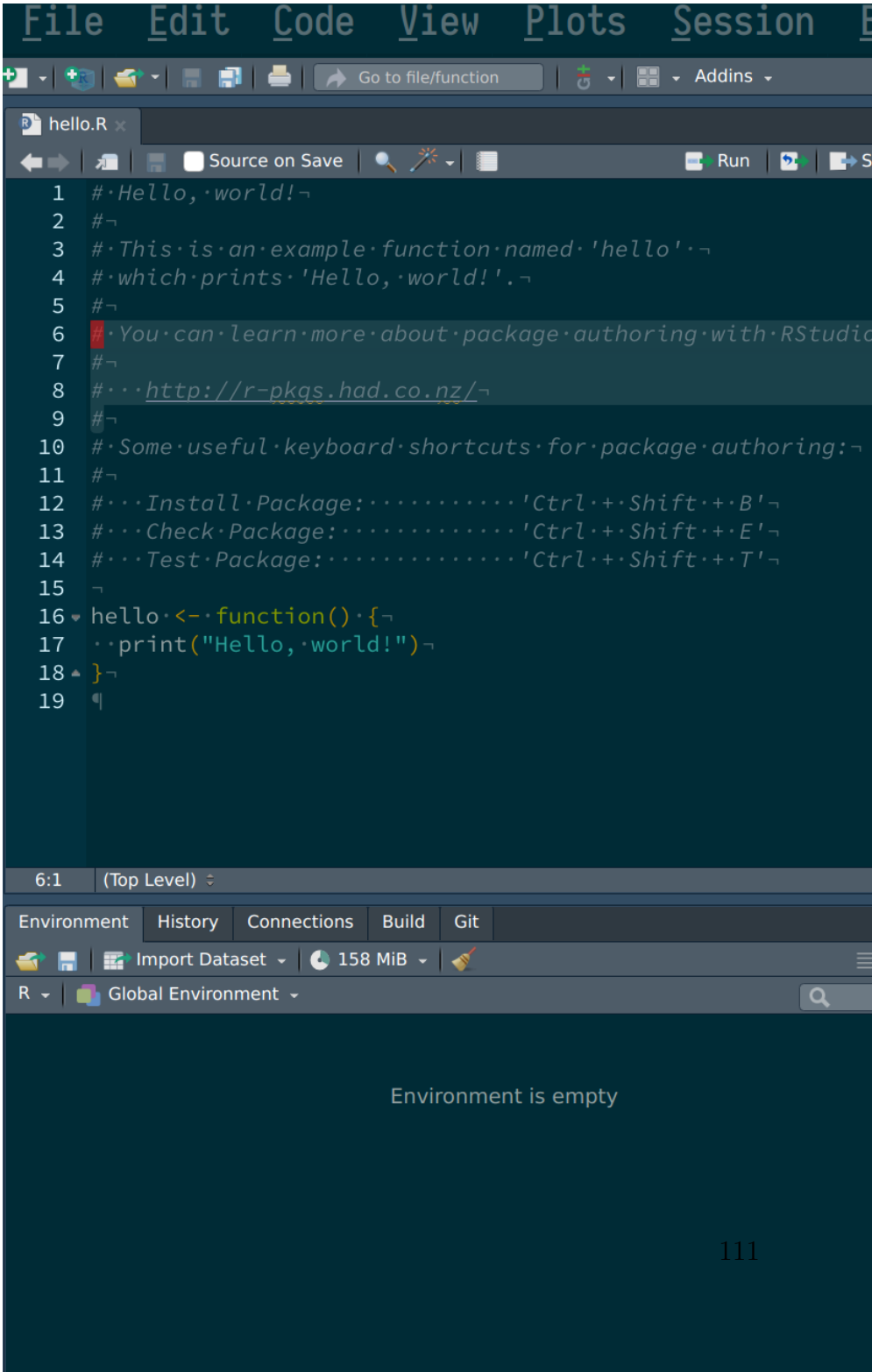

File Edit Code View Plots Session
Go to file/function
Addins
hello.R
Source on Save
Run
1 #·Hello,·world!¬
2 #¬
3 #·This·is·an·example·function·named·'hello'·¬
4 #·which·prints·'Hello,·world!'.¬
5 #¬
6 #·You·can·learn·more·about·package·authoring·with·RStudi
7 #¬
8 #···http://r-pkgs.had.co.nz/¬
9 #¬
10 #·Some·useful·keyboard·shortcuts·for·package·authoring:¬
11 #¬
12 #···Install·Package:···········'Ctrl·+·Shift·+·B'¬
13 #···Check·Package:·············'Ctrl·+·Shift·+·E'¬
14 #···Test·Package:··············'Ctrl·+·Shift·+·T'¬
15 ¬
16 hello·<-·function()·{¬
17 ··print("Hello,·world!")¬
18 }¬
19 ¶
1:1
(Top Level)
Environment
History
Connections
Build
Git
Import Dataset
156 MiB
R
Global Environment
Environment is empty

We can remove this script, but do take note of the following sentence:

```
# You can learn more about package authoring with
RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
```

If this course succeeded in turning you into an avid R programmer, you might want to contribute to the language by submitting some nice packages one day. You could at that point refer to this link to learn the many, many subtleties of package development. But for our purposes, this chapter will suffice.

Ok, so now let's take a look inside the folder you just created and take a look at the package's structure. You can do so easily from within RStudio:

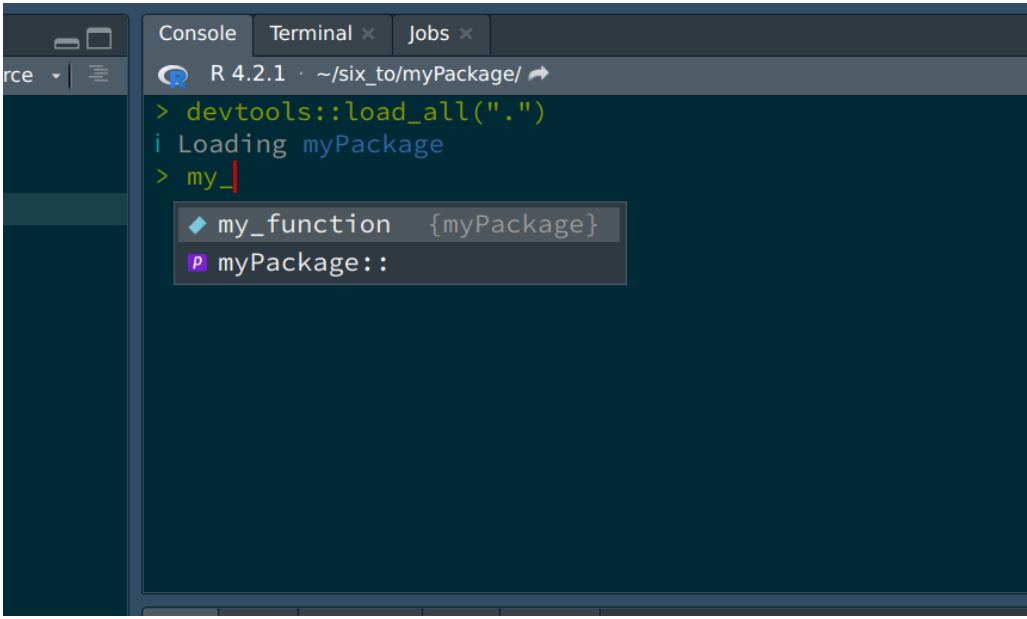

File Edit Code View Plots Session
Go to file/function
Addins
hello.R
Source on Save
Run
#·Hello,·world!¬
#¬
#·This·is·an·example·function·named·'hello'·¬
#·which·prints·'Hello,·world!'.¬
#¬
#·You·can·learn·more·about·package·authoring·with·RStudio
#¬
#···http://r-pkgs.had.co.nz/¬
#¬
#·Some·useful·keyboard·shortcuts·for·package·authoring:¬
#¬
#···Install·Package:·····'Ctrl·+·Shift·+·B'¬
#···Check·Package:·····'Ctrl·+·Shift·+·E'¬
#···Test·Package:·····'Ctrl·+·Shift·+·T'¬
¬
hello·<-·function()·{¬
··print("Hello,·world!")¬
}¬
¶
6:1
(Top Level)
Environment
History
Connections
Build
Git
Import Dataset
158 MiB
R
Global Environment
Environment is empty

But you can also navigate to the folder from inside a file explorer. The folder that will matter to us the most for now is the `R` folder. This folder will contain your scripts, which will contain your package's functions. Let's start by adding a new script.

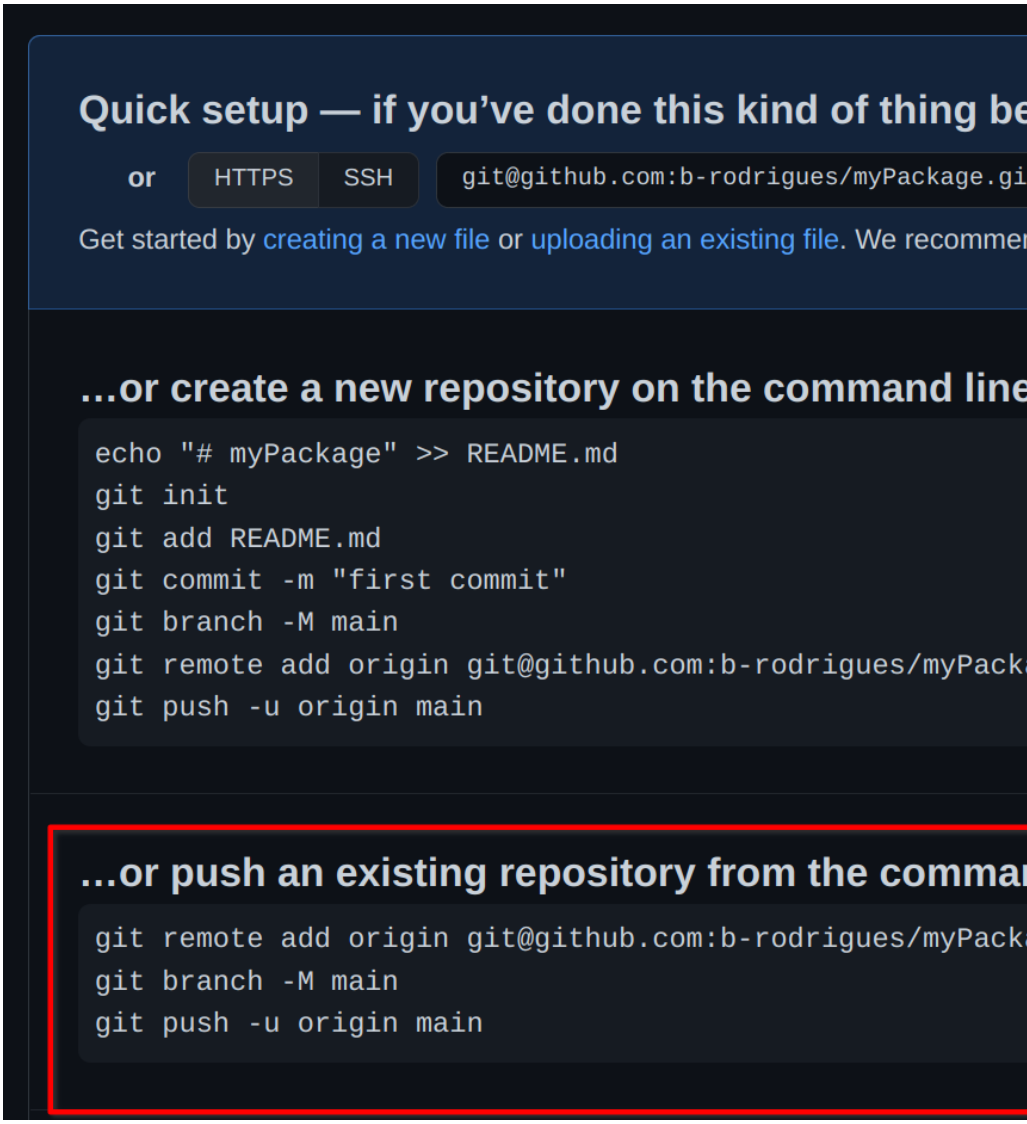
## 4.3 Adding functions

To add a new script, simply create a new script, and while we're at it, let's add some code to it:

and that's it! Well, this example is incredibly easy; there will be more subtleties later on, but these are the basics: simply write your script as usual. Now let's load the package with `CTRL-SHIFT-L`. Loading the package makes it available in your current R session:



As you can see, the package is loaded, and RStudio's autocomplete even suggest the function's name already. So now that we have already a function, let's push our code to github.com (you remember that we checked the box `Create a git repository` when we started the project?). For this, let's go back to github.com and create a new repository. Give it the same name as your package on your computer, just to avoid confusion. Once the repo is created, you will see this familiar screen:




We will start from an existing repository, because our repository already exists. So we can use the terminal to enter the commands suggested here. We can also use the terminal from RStudio:

The steps above are a way to link your local repository to the remote repository living on github.com. Without these initial steps, there is no way to link your package project to github.com!

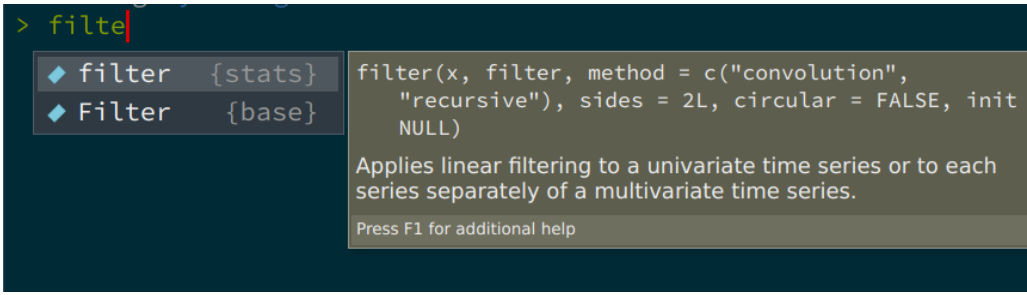
Let's now write another function, which will depend on functions from other packages.

### 4.3.1 Functions dependencies

In the same script, add the following code:

```
only_automatics <- function(dataset){
  dataset |>
    filter(am == 1)
}
```

This creates a function that takes a dataset as an argument, and filters the `am` variable. This function is not great: it is not documented, so the user might not know that the dataset that is meant here is the `mtcars` dataset (which is included with R by default). So we will need to document this. Also, the variable `am` is hardcoded, that's not good either. What if the user wants to filter another variable with another value? We will solve these issues later on. But there is a worse problem here. The `filter()` function that the developer intended to use here is `dplyr::filter()`, so the one from the `{dplyr}` package. However, there are several functions called `filter()`. If you start typing `filter` inside a fresh R session, this is what autocomplete suggests:




So there's a `filter()` function from the `{stats}` package (which gets loaded automatically with every new R session), and there's a capital F `Filter()` function from the `{base}` package (R is case sensitive, so `filter()` and `Filter()` are different functions). So how can the developer specify the correct `filter()` function? Simply by using the following notation: `dplyr::filter()` (which we have already encountered). So let's rewrite the function correctly:

```
only_automatics <- function(dataset){
  dataset |>
    dplyr::filter(am == 1)
}
```

Great, so now `only_automatics()` at least knows which `filter` function to use, but this function could be improved a lot more. In general, what you want is to have a function that is general enough that it could work with any variable (if the dataset is supposed to be fixed), or that could work with any combination of dataset and variable. Let's make our function a bit more general, by making it work on any variable from any dataset:

```
my_filter <- function(dataset, condition){
  dataset |>
```

```
    dplyr::filter(condition)
}
```

I renamed the function to `my_filter()` because now this function can work on any dataset and with any predicate condition (of course this function is not really useful, since it's only a wrapper around `filter()`. But that's not important). Let's save the script and reload the package with `CRTL-SHIFT-L` and try out the function:

```
my_filter(mtcars, am == 1)
```

You will get this output:

```
Error in `dplyr::filter()` at
myPackage/R/functions.R:6:4:
! Problem while computing `..1 = condition`.
Caused by error in `mask$eval_all_filter()`:
! object 'am' not found
Run `rlang::last_error()` to see where the error
occurred.
```

so what's going on? R complains that it cannot find `am`. What is wrong with our function? After all, if I call the following, it works:

```
mtcars |>
  dplyr::filter(am == 1)
```

```
                mpg cyl  disp  hp drat    wt  qsec
                vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46
0  1    4    4
```

```
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02
0  1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61
1  1    4    1
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47
1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52
1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90
1  1    4    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90
1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70
0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90
1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50
0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50
0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60
0  1    5    8
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60
1  1    4    2
```

So what gives? What's going on here, is that R doesn't know that it has look for `am` inside the `mtcars` dataset. R is looking for a variable called `am` in the global environment, which does not exist. `dplyr::filter()` is programmed in a way that tells R to look for `am` inside `mtcars` and not in the global environment (or whatever parent environment the function gets called from). We need to program our function in the same way. Remember in chapter 3, where we learned about functions that take columns of data frames as arguments? This is exactly the same situta-

tion here. So, let's simply enclose references to columns of data frames inside `{{}}`, like so:

```
my_filter <- function(dataset, condition){
  dataset |>
    dplyr::filter({{condition}})
}
```

Now, R knows where to look. So reload the package with `CTRL-SHIFT-L` and try again:

```
my_filter(mtcars, am == 1)
```

```
                mpg cyl  disp  hp drat    wt  qsec
                vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46
0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02
0  1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61
1  1    4    1
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47
1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52
1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90
1  1    4    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90
1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70
0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90
1  1    5    2
```

```
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50
0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50
0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60
0  1    5    8
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60
1  1    4    2
```

And it's working!

`{{}}` is not a feature available in a base installation of R, but is provided by packages from the `tidyverse` (like `{dplyr}`, `{tidyr}`, etc). If you write functions that depend on `{dplyr}` functions like `filter()`, `select()` etc, you'll have to know to keep using `{{}}`.

Let's now write a more useful function. Remember the datasets about unemployment in Luxembourg? I'm thinking about the ones here, `unemp_2013.csv`, `unemp_2014.csv`, etc.

Let's write a function that does some basic transformations on these files:

```
clean_unemp <- function(unemp_data, level,
 ↪  col_of_interest){

  unemp_data |>
    janitor::clean_names() |>
    dplyr::filter({{level}}) |>
    dplyr::select(year, commune,
 ↪  {{col_of_interest}})
}
```

This function does 3 things:

- using `janitor::clean_names()`, it cleans the column names;
- it filters on a user supplied level. This is because the csv file contains three "regional" levels so to speak: the whole country: first row, where commune equals `Grand-Duché de Luxembourg`, canton level: where commune contains the string `Canton` and the last level: the actual communes. Welcome to the real world, where data is dirty and does not always make sense.
- it selects the columns that interest us (with year and commune hardcoded, because we always want those)

So save the script, and reload your package using `CTRL-SHIFT-L`, and try with the following lines:

```
unemp_2013 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodr
```

```
Rows: 118 Columns: 8
-- Column specification
--------------------------------------------------------
Delimiter: ","
chr (1): Commune
dbl (7): Total employed population, of which:
Wage-earners, of which: Non-wa...

i Use `spec()` to retrieve the full column
specification for this data.
i Specify the column types or set `show_col_types =
FALSE` to quiet this message.
```

```
clean_unemp(unemp_2013,
            grepl("Grand-D.*", commune),
            active_population)
```

```
# A tibble: 1 x 3
   year commune                    active_population
  <dbl> <chr>                                  <dbl>
1  2013 Grand-Duche de Luxembourg             242694
```

This selects the columns for the whole country. Let's try for cantons:

```
clean_unemp(unemp_2013,
            grepl("Canton", commune),
            active_population)
```

```
# A tibble: 12 x 3
    year commune             active_population
   <dbl> <chr>                           <dbl>
 1  2013 Canton Capellen                18873
 2  2013 Canton Esch                    73063
 3  2013 Canton Luxembourg              68368
 4  2013 Canton Mersch                  13774
 5  2013 Canton Clervaux                 7936
 6  2013 Canton Diekirch                14056
 7  2013 Canton Redange                  7902
 8  2013 Canton Vianden                  2280
 9  2013 Canton Wiltz                    6670
10  2013 Canton Echternach               7967
11  2013 Canton Grevenmacher            12254
12  2013 Canton Remich                   9551
```

And to select for communes, we need to not select cantons nor the whole country:

```
clean_unemp(unemp_2013,
            !grepl("(Canton|Grand-D.*)", commune),
            active_population)
```

```
# A tibble: 105 x 3
    year commune       active_population
   <dbl> <chr>                     <dbl>
 1  2013 Dippach                    1817
 2  2013 Garnich                     869
 3  2013 Hobscheid                  1505
 4  2013 Kaerjeng                   4355
 5  2013 Kehlen                     2244
 6  2013 Koerich                    1016
 7  2013 Kopstal                    1284
 8  2013 Mamer                      3209
 9  2013 Septfontaines               399
10  2013 Steinfort                  2175
# i 95 more rows
```

This seems to be working well (in one of the next sections we will learn how to systematize these tests, instead of running them by hand each time we change the function). Before continuing, let's commit and push our changes.

## 4.4 Documentation

It is time to start documenting our functions, and then our package. Documentation in R is not just about telling users how to use the package and its functions, but it also serves a functional role. There are several files that must be edited to completely document a package, and these files also help define the dependencies of the package. Let's start with the simplest thing we can do, which is documenting functions.

### 4.4.1 Documenting functions

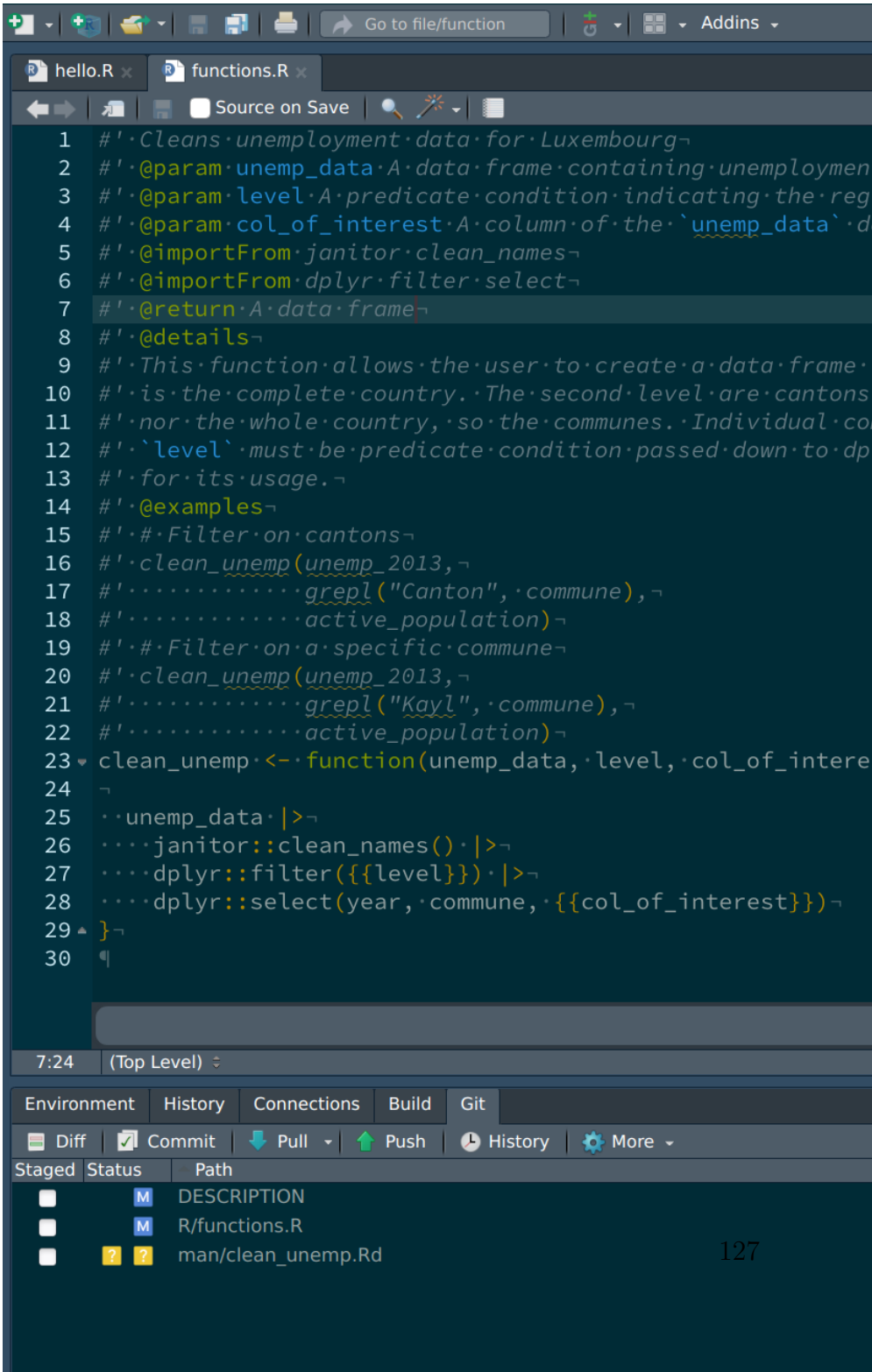
As you'll know, comments in R start with `#`. Documenting functions consists in commenting them with a special kind of comments that start with `#'`. Let's try on our `clean_unemp()` function:

```
#' Easily filter unemployment data for Luxembourg
#' @param unemp_data A data frame containing
 ↪  unemployment data for Luxembourg.
#' @param level A predicate condition indicating the
 ↪  regional level of interest. See details for
 ↪  more.
#' @param col_of_interest A column of the
 ↪  `unemp_data` data frame that you wish to select.
#' @importFrom janitor clean_names
#' @importFrom dplyr filter select
#' @export
#' @return A data frame
#' @details
#' This function allows the user to create a data
 ↪  frame for several regional levels. The first
 ↪  level
#' is the complete country. The second level are
 ↪  cantons, and the third level are neither cantons
#' nor the whole country, so the communes.
 ↪  Individual communes can be selected as well.
#' `level` must be predicate condition passed down
 ↪  to dplyr::filter. See the examples below
#' for its usage.
#' @examples
#' # Filter on cantons
#' clean_unemp(unemp_2013,
```

```
#'             grepl("Canton", commune),
#'             active_population)
#' # Filter on a specific commune
#' clean_unemp(unemp_2013,
#'             grepl("Kayl", commune),
#'             active_population)
clean_unemp <- function(unemp_data, level,
 ↪  col_of_interest){

  unemp_data |>
    janitor::clean_names() |>
    dplyr::filter({{level}}) |>
    dplyr::select(year, commune,
 ↪  {{col_of_interest}})
}
```

The special comments that start with `#'` will be compiled into a nice looking document that users can then read. You can add sections to the documentation by using keywords that start with `@`. The example above shows essentially everything you need to know to properly document your functions. An important keyword, that will not appear in the documentation itself, is `@importFrom`. This will be useful later, when we document the package, as it helps define the dependencies of your package. For now, let's simply remember to write this. The other thing that might not be obvious is the `@export` line. This simply tells R that this function should be public, available to the users. If you need to define private functions, you can omit this keyword and the function won't be visible to users (this is only partially true however, users can always reach deep into the package and use private functions by using `:::`, as in `package:::my_private_function()`).

You can now save the script and press `CRTL-SHIFT-D`. This will generate the help file for your function:


Go to file/function
Addins
hello.R
functions.R
Source on Save
#' Cleans unemployment data for Luxembourg
#' @param unemp_data A data frame containing unemploymen
#' @param level A predicate condition indicating the reg
#' @param col_of_interest A column of the `unemp_data` d
#' @importFrom janitor clean_names
#' @importFrom dplyr filter select
#' @return A data frame
#' @details
#' This function allows the user to create a data frame
#' is the complete country. The second level are cantons
#' nor the whole country, so the communes. Individual co
#' `level` must be predicate condition passed down to dp
#' for its usage.
#' @examples
#' #Filter on cantons
#' clean_unemp(unemp_2013,
#' grepl("Canton", commune),
#' active_population)
#' #Filter on a specific commune
#' clean_unemp(unemp_2013,
#' grepl("Kayl", commune),
#' active_population)
clean_unemp <- function(unemp_data, level, col_of_intere
unemp_data |>
janitor::clean_names() |>
dplyr::filter({{level}}) |>
dplyr::select(year, commune, {{col_of_interest}})
}
7:24
(Top Level)
Environment
History
Connections
Build
Git
Diff
Commit
Pull
Push
History
More
Staged
Status
Path
DESCRIPTION
R/functions.R
man/clean_unemp.Rd

Now, writing `?clean_unemp` in the console shows the documentation for `clean_unemp`:

Now, let's document the package.

## 4.4.2 Documenting the package

### 4.4.2.1 The NAMESPACE file

There are several files you need to edit to properly document your package. Some are optional like *vignettes*, and some are not optional, like the DESCRIPTION and the NAMESPACE. Let's start with the NAMESPACE. This file gets generated automatically, but sometimes it can happen that it gets stuck in a state where it doesn't get generated anymore. In these cases, you should simply delete it, and then document your package again:

As you can see from the video, the NAMESPACE defines some interesting stuff. First, it says which of our functions are exported, and should be available to the users. Then, it defines the imports. This is possible because of the `@importFrom` keywords from before. What we can now do, is go back to our function and remove all the references to the packages and simply use the functions, meaning that we can use `filter(blabla)` instead of `dplyr::filter()`. But in my opinion, it is best to keep them the script as it is. There already there, and by having them there, even if they're redundant with the NAMESPACE, someone reading the source code will know immediately where the functions come from. But if you want, you can remove the references to the packages, it'll work.

#### 4.4.2.2 The DESCRIPTION file

The DESCRIPTION file requires more manual work. Let's take a look at the file as it stands:

```
Package: myPackage
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer
<yourself@somewhere.net>
Description: More about what it does (maybe more
than one line)
    Use four spaces when indenting paragraphs within
    the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.2.1
```

Let's change this to:

```
Package: myPackage
Type: Package
Title: Clean Lux Unemployment Data
Version: 0.1.0
Author: Bruno Rodrigues
Maintainer: Bruno Rodrigues
Description: This package allows users to easily get
unemployent data for Luxembourg
    from raw csv files
License: GPL (>=3)
Encoding: UTF-8
```

```
LazyData: true
RoxygenNote: 7.2.1
```

All of this is not really important if you're not releasing your package on CRAN, but still important to think about. If it's a package you're keeping private to your company, none of it matters much, but if it's on github.com, you might want to still fill out these fields. What could be important is the license you're releasing the package under (again, only important if you release on CRAN or keep it on github.com). What's really important in this file, is what's missing. We are going to add some more lines to this file, which are quite important:

```
Package: myPackage
Type: Package
Title: Clean Lux Unemployment Data
Version: 0.1.0
Author: Bruno Rodrigues
Maintainer: Bruno Rodrigues
Description: This package allows users to easily get
unemployent data for Luxembourg
    from raw csv files
License: GPL (>=3)
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.2.1
RemoteType: github
Depends:
    R (>= 4.1),
Imports:
    dplyr,
    janitor
Suggests:
    knitr,
```

```
rmarkdown,
testthat
```

We added three fields:

- RemoteType: we need to specify here that this package lives on github.com. This will become important for reproducibility purposes.
- Depends: we can define hard dependencies here. Because I'm using the base pipe `|>` in my examples, my package needs at least R version 4.1.
- Imports: these is where we list packages that our package needs in order to run. If these packages are not available, they will be installed when users install our package.
- Suggests: these packages are not required to run, but can unlock further capabilities. This is also where we can list packages that are required to build *vignettes* (which we'll discover shortly), or for unit testing.

There is another field we could add, Remotes, which is where we could define the usage of a package only released on github.com. To know more about this, read this section of R packages.

#### 4.4.2.3 Vignettes

Vignettes are long form documentation that explain some of the use-cases of your package. They are written in the RMarkdown format, which we will learn about in chapter 8. To see an example of a vignette, you can take a look at this vignette titled A non-mathematician's introduction to monads from my `{chronicler}` package, or you could also type:

```
vignette(package = "dplyr")
```

to see the list of available vignettes for the `{dplyr}` package, and then write:

```
vignette("programming", "dplyr")
```

to open the vignette locally.

#### 4.4.2.4 Package's website

In the previous section I've linked to a vignette from my `{chronicler}` package. The website was automatically generated using the pkgdown package. We are not going to discuss how it works in detail here, but you should know this exists, and is actually quite easy to use.

### 4.4.3 Checking your package

Before sharing your package with the world, you might want to run `devtools::check()` to make sure everything is alright. `devtools::check()` will make sure that you didn't forget something crucial, like declaring a dependency in the DESCRIPTION file for example. The goal is to see something like this at the end of `devtools::check()`:

```
0 errors   | 0 warnings   | 3 notes
```

If you want to release your package on CRAN, it's a good idea to address the notes as well, but what you must absolutely deal with are errors and warnings, even if you're keeping this package for yourself.

### 4.4.4 Installing your package

Once you're done working on your package, you can install it with `CRTL-SHIFT-B`. This way you can start using your package from any R session. People that want to install your package can use `devtools::install_github()` to install it from github.com. You might want to communicate a specific commit hash to your users, so they install a fixed version of your package, and not the latest development version. For example, let's suppose that I have been working on my package, but would prefer my potential users to install the package as it stood at the commit with the hash `"e9d9129de3047c1ecce26d09dff429ec078d4dae"`. I can write this in the README of the package:

```
To install the package, please use the following
line

devtools::install_github("b-rodrigues/myPackage",
ref = "e9d9129de3047c1ecce26d09dff429ec078d4dae")
```

This will install the `{myPackage}` package as it looked like at this particular commit. You could also create a branch called `release` for example, and direct users to install from this branch:

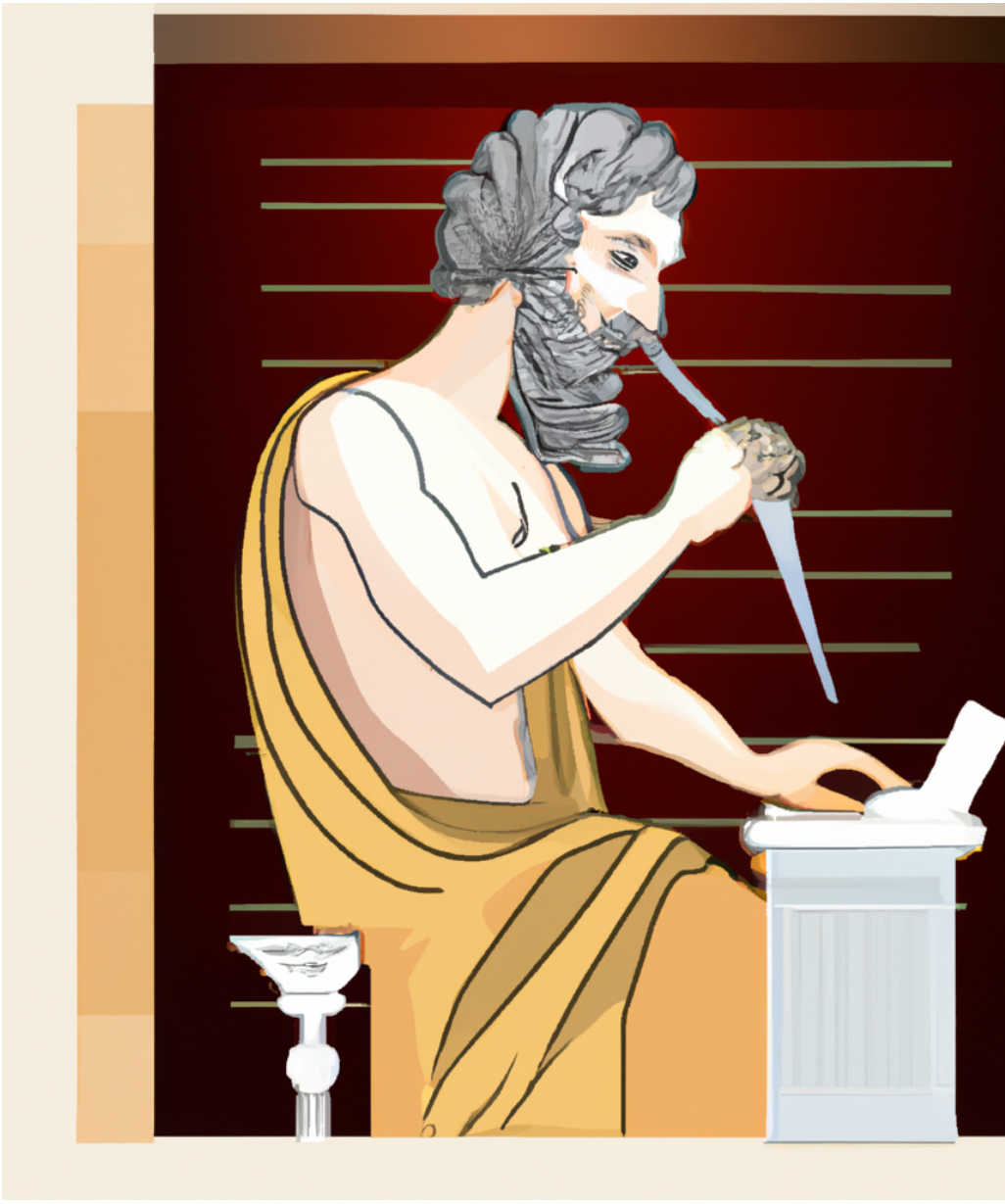
```
To install the package, please use the following
line

devtools::install_github("b-rodrigues/myPackage",
ref = "release")
```

But for this, you need to create a release branch, which will only contain release-ready code.

## 4.5 Further reading

- https://r-pkgs.org/

# 5 Unit tests



What you'll have learned by the end of the chapter: what unit tests are, how to write them, and how to test your package thoroughly.

## 5.1 Introduction

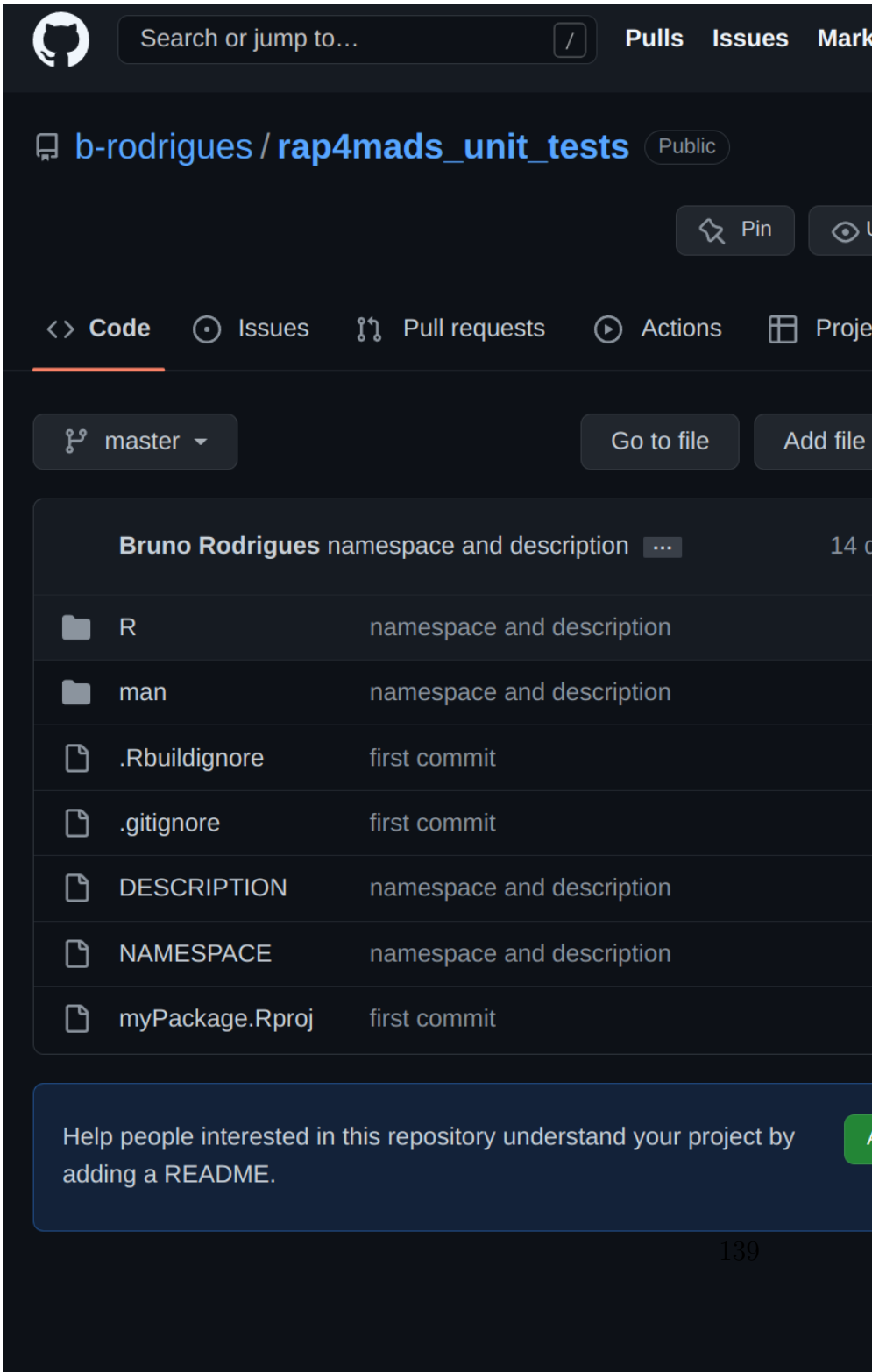
It might not have seemed like it, but developing our own package was actually the first step in writing reproducible code. Packaged code is easy to share, and much easier to run than code that lives inside scripts. When you share code, be it with future you or others, you have a responsibility to ship high quality code. Unit tests are one way to ensure that your code works as intented, but it is not a panacea. But if you write short, well-documented functions, and you package them, and test them thoroughly, you are on the right track for success.

But what are unit tests? Unit tests are pieces of code that test other pieces of code (called units in this context). It turns out that functions are units of code, and that makes testing them quite easy. I hope that you are starting to see the pieces coming all together: I introduced you to functional programming and insisted that you write your code as a sequence of functions calls, because it makes it easier to package and document everything. And now that your code lives inside a package, as a series of functions, it will be very easy to test these functions (or units of code).

## 5.2 Testing your package

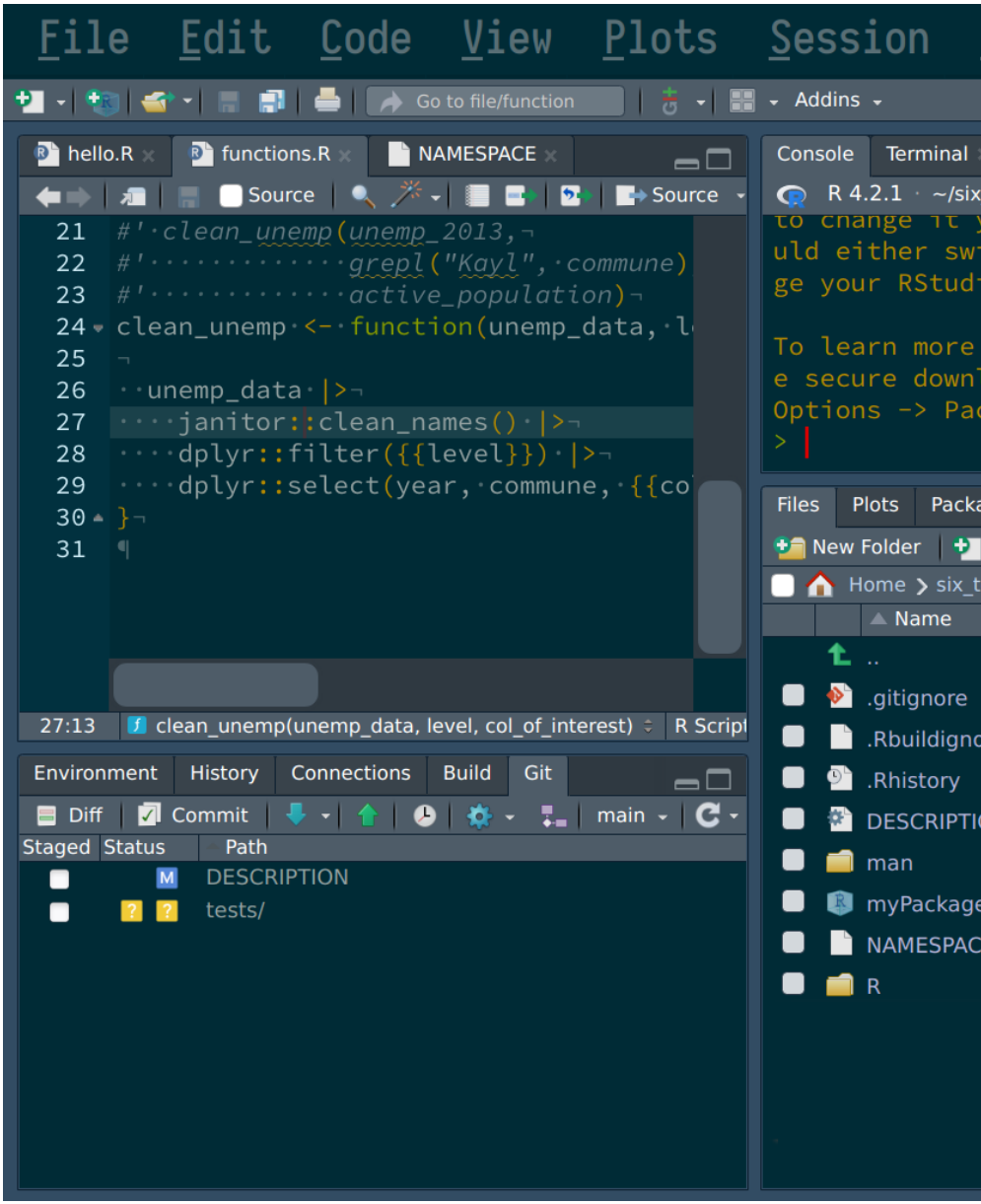
To make sure that each one of us starts with the exact same package and code, you will first of all fork the following repository

that you can find here.


Search or jump to…
/
Pulls
Issues
Mark
b-rodrigues / rap4mads_unit_tests
Public
Pin
Code
Issues
Pull requests
Actions
Proje
master
Go to file
Add file
Bruno Rodrigues namespace and description
14 c
R
namespace and description
man
namespace and description
.Rbuildignore
first commit
.gitignore
first commit
DESCRIPTION
namespace and description
NAMESPACE
namespace and description
myPackage.Rproj
first commit
Help people interested in this repository understand your project by adding a README.

Forking the repository will add a copy of the repository to your github account. You can now clone your fork of the repo (make sure you clone using the ssh link!) and start working!

Because our code is packaged, starting to write unit tests will be very easy. For this, open RStudio and make sure your package's project is opened:




In order to set up the required files and folders for unit testing,

run the following line in the R console:

```
usethis::use_test("clean_unemp")
```

You should see a folder called `tests` appear inside the package. Inside `tests`, there is another folder called `testthat`, and inside this folder you should find a file called `test-clean_unemp.R`. This file should contain an example:

```
test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

This is quite self-explanatory; `test_that()` is the function that we are going to use to write tests. It takes a string as an argument, and a test. For the string write an explanatory name. This will make it easier to find the test if it fails. `expect_equal()` is a function that tests the equality between its arguments. On one side we have `2 * 2`, and on the other, `4`. All our tests will look somewhat like this. There are many `expect_` functions, that allow you to test for many conditions. You can take a look at `{testthat}`'s function reference for a complete list.

So, what should we test? Well, here are several ideas:

- Is the function returning an expected value for a given input?
- Can the function deal with all kinds of input? What happens if an unexpected input is provided?
- Is the function failing as expected for certain inputs?
- Is the function dealing with corner cases as intended?

Let's try to write some tests for our `clean_unemp()` function now, and start to consider each of these questions.

### 5.2.1 Is the function returning an expected value for a given input?

Let's start by testing if our function actually returns data for the Grand-Duchy of Luxembourg if the user provides a correct regular expression. Add these lines to the script (and remove the example test while you're at it):

```
unemp_2013 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodr

test_that("selecting the grand duchy works", {

  returned_value <- clean_unemp(
                       unemp_2013,
                       grepl("Grand-D.*", commune),
                       active_population)

expected_value <- tibble::as_tibble(
list("year" = 2013,
"commune" = "Grand-Duche de Luxembourg",
"active_population" = 242694))

  expect_equal(returned_value, expected_value)
})
```

So what's going on here? First, I need to get the data. I load the data outside of the test, so it'll be available to every test afterwards as well. Then, inside the test, I need to define two more variables: the actual value returned by the function, and the value that we expect. I need to create this value by hand, and I do so using the `tibble::as_tibble()` function. This function takes a list as an argument and converts it to a tibble. I did not

explain what tibbles are yet: tibbles are basically the same as a data frame, but have a nicer print method, and other niceties. In practice, you don't need to think about tibbles too much, but here you need to be careful: `clean_unemp()` returns a tibble, because that's what `{dplyr}` functions return by default. So if in your test you compare a tibble to a data.frame, your test will fail, because their classes are not equal. So I need to define my expected value as a tibble for the test to pass.

You can now save the script, and press `CTRL-SHIFT-T` to run the test. The test should pass, if not, there's either something wrong with your function, with the inputs you provided to it, or with the expected value. You can keep adding tests to this script, to cover every possible use case:

```
test_that("selecting cantons works", {

  returned_value <- clean_unemp(
                       unemp_2013,
                       grepl("Canton", commune),
                       active_population)

expected_value <-
 ↪  readr::read_csv("test_data_cantons.csv")

  expect_equal(returned_value, expected_value)
})
```

In the test above, I cannot write the expected value by hand. So what I did instead was run my function in a terminal, and save the output in a csv file. I used the following code for this:

```
clean_unemp(unemp_2013,
            grepl("Canton", commune),
            active_population) %>%

 ↪  readr::write_csv("tests/testthat/test_data_cantons.csv")
```

I inspected this output to make sure everything was correct. I can now keep this csv file and test my function against it. Should my function fail when tested against it, I know that something is wrong. We can do the same for communes. First, save the "ground truth" in a csv file:

```
clean_unemp(unemp_2013,
            !grepl("(Canton|Grand-D.*)", commune),
            active_population) %>%

 ↪  readr::write_csv("tests/testthat/test_data_communes.csv")
```

Then, we can use this csv file in our tests:

```
test_that("selecting communes works", {

  returned_value <- clean_unemp(
                      unemp_2013,
                      !grepl("(Canton|Grand-D.*)",
                       ↪  commune),
                      active_population)

  expected_value <-
 ↪  readr::read_csv("test_data_communes.csv")
```

```
  expect_equal(returned_value, expected_value)
})
```

We could even add a test for a specific commune:

```
test_that("selecting one commune works", {

  returned_value <- clean_unemp(
                        unemp_2013,
                        grepl("Kayl", commune),
                        active_population)

expected_value <- tibble::as_tibble(
                              list("year" = 2013,
                                   "commune" = "Kayl",
                                    ↪
                                   "active_population"
                                    ↪  = 3863))

  expect_equal(returned_value, expected_value)
})
```

So your final script would look something like this:

```
unemp_2013 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodrig
 ↪  show_col_types = FALSE)

test_that("selecting the grand duchy works", {

  returned_value <- clean_unemp(
```

```
    unemp_2013,
    grepl("Grand-D.*", commune),
    active_population)

  expected_value <- tibble::as_tibble(
                               list("year" = 2013,
                                    "commune" =
                                     ↪  "Grand-Duche
                                     ↪  de
                                     ↪  Luxembourg",

                                     ↪  "active_population"
                                     ↪  = 242694))

  expect_equal(returned_value, expected_value)

})

test_that("selecting cantons work", {

  returned_value <- clean_unemp(
    unemp_2013,
    grepl("Canton", commune),
    active_population)

  expected_value <-
 ↪  readr::read_csv("test_data_cantons.csv",
 ↪  show_col_types = FALSE)

  expect_equal(returned_value, expected_value)

})
```

```
test_that("selecting communes works", {

  returned_value <- clean_unemp(
    unemp_2013,
    !grepl("(Canton|Grand-D.*)", commune),
    active_population)

  expected_value <-
 ↪  readr::read_csv("test_data_communes.csv",
 ↪  show_col_types = FALSE)

  expect_equal(returned_value, expected_value)

})

test_that("selecting one commune works", {

  returned_value <- clean_unemp(
    unemp_2013,
    grepl("Kayl", commune),
    active_population)

  expected_value <- tibble::as_tibble(
                              list("year" = 2013,
                                   "commune" =
                                    ↪  "Kayl",

                                    ↪  "active_population"
                                    ↪  = 3863))

  expect_equal(returned_value, expected_value)

})
```

### 5.2.2 Can the function deal with all kinds of input?

What *should* happen if your function gets an unexpected input? Let's write a unit test and then see if it passes. For example, what if the user enters a commune name that is not in Luxembourg? We expect the data frame to be empty, so let's write a test for that

```
test_that("wrong commune name", {

returned_value <- clean_unemp(
                      unemp_2013,
                      grepl("Paris", commune),
                      active_population)

expected_value <- tibble::as_tibble(
list("year" = numeric(0),
"commune" = character(0),
"active_population" = numeric(0)))


  expect_equal(returned_value, expected_value)

})
```

This test reveals something interesting: your function returns an empty data frame, but the user might not understand what's wrong. Maybe we could add a message to inform the user? We could write something like:

```
clean_unemp <- function(unemp_data, level,
 ↪  col_of_interest){

  result <- unemp_data |>
    janitor::clean_names() |>
    dplyr::filter({{level}}) |>
    dplyr::select(year, commune,
 ↪  {{col_of_interest}})

  if(nrow(result) == 0) {
    warning("The returned data frame is empty. This
     ↪  is likely because the `level` argument
     ↪  supplied does not match any rows in the
     ↪  original data.")
  }
  result
}
```

Replace the `clean_unemp()` function from your package with this one, and rerun the tests. The test should still pass, but a warning will be shown. We can test for this as well; is the warning thrown? Let's write the required test for it:

```
test_that("wrong commune name: warning is thrown", {

  expect_warning({
    clean_unemp(
      unemp_2013,
      grepl("Paris", commune),
      active_population)
  }, "This is likely")
```

```
})
```

`expect_warning()` needs the expression that should raise the warning, and a regular expression. I've used the string "This is likely", which appears in the warning. This is to make sure that the correct warning is raised. Should another warning be thrown, the test will fail, and I'll know that something's wrong (try to change the regular expression and rerun the test, you see that it'll fail).

## 5.3 Back to developing again

Now might be a good time to stop writing tests and think a little bit. While writing these tests, and filling the shoes of your users, you might have realized that your function might not be that great. We are asking users to enter a regular expression to filter data, which is really not great nor user-friendly. And this is because the data we're dealing with is actually not clean, because the same column mixes three different regional levels. For example, what if the users wants to take a look at the commune "Luxembourg"?

```
clean_unemp(
  unemp_2013,
  grepl("Luxembourg", commune),
  active_population)
```

```
# A tibble: 3 × 3
 year commune                    active_population
<dbl> <chr>                                 <dbl>
```

```
2013 Grand-Duche de Luxembourg          242694
2013 Canton Luxembourg                   68368
2013 Luxembourg                          43368
```

So the user gets back three rows; that's because there's the country, the canton and the commune of Luxembourg. Of course the user can now filter again to just get the commune. But this is not a good interface.

What we should do instead is clean the input data. And while we're at it, we could also provide the data directly inside the package. This way users get the data "for free" once they install the package. Let's do exactly that. To package data, we first need to create the `data-raw` folder. This can be done with the following call:

```
usethis::use_data_raw()
```

There's a script called `DATASET.R` inside the `data-raw` folder. This is the script that we should edit to clean the data. Let's write the following lines in it:

```
## code to prepare `DATASET` dataset goes here

unemp_2013 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodrig
unemp_2014 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodrig
unemp_2015 <-
 ↪  readr::read_csv("https://raw.githubusercontent.com/b-rodrig

library(dplyr)
```

```
clean_data <- function(x){
  x %>%
    janitor::clean_names() %>%
    mutate(level = case_when(
              grepl("Grand-D.*", commune) ~
               ↪  "Country",
              grepl("Canton", commune) ~ "Canton",
              !grepl("(Canton|Grand-D.*)", commune) ~
               ↪  "Commune"
            ),
            commune = ifelse(grepl("Canton",
             ↪  commune),

 ↪  stringr::str_remove_all(commune, "Canton "),
                               commune),
            commune = ifelse(grepl("Grand-D.*",
             ↪  commune),

 ↪  stringr::str_remove_all(commune, "Grand-Duche de
 ↪  "),
                               commune),
            ) %>%
    select(year,
            place_name = commune,
            level,
            everything())
}


my_datasets <- list(
  unemp_2013,
  unemp_2014,
  unemp_2015
```

```
)

unemp <- purrr::map_dfr(my_datasets, clean_data)

usethis::use_data(unemp, overwrite = TRUE)
```

Running this code creates a dataset called `unemp`, which users of your package will be able to load using `data("unemp")` (after having loaded your package). The now contains a new column called `level` which will make filtering much easier. After `usethis::use_data()` is done, we can read following message in the R console:

```
  Saving 'unemp' to 'data/unemp.rda'
• Document your data (see
'https://r-pkgs.org/data.html')
```

We are invited to document our data. To do so, create and edit a file called `data.R` in the `R` directory:

```
#' Unemployement in Luxembourg data
#'
#' A tidy dataset of unemployment data in
 ↪  Luxembourg.
#'
#' @format ## `who`
#' A data frame with 7,240 rows and 60 columns:
#' \describe{
#'   \item{year}{Year}
#'   \item{place_name}{Name of commune, canton or
 ↪  country}
#'   \item{level}{Country, Canton, or Commune}
#'   \item{total_employed_population}{Total employed
 ↪  population living in `place_name`}
```

```
#'   \item{of_which_wage_earners}{... of which are
 ↪  wage earners living in `place_name`}
#'   \item{of_which_non_wage_earners}{... of which
 ↪  are non-wage earners living in `place_name`}
#'   \item{unemployed}{Total unemployed population
 ↪  living in `place_name`}
#'   \item{active_population}{Total active
 ↪  population living in `place_name`}
#'
 ↪  \item{unemployement_rate_in_percent}{Unemployement
 ↪  rate in `place_name`}
#'   ...
#' }
#' @source <https://is.gd/e6wKRk>
"unemp"
```

You can now rebuild the document using `CTRL-SHIFT-D` and reload the package using `CRTL-SHIFT-L`. You should now be able to load the data into your session using `data("unemp")`.

We can now change our function to accommodate this new data format. Let's edit our function like this:

```
#' Easily filter unemployment data for Luxembourg
#' @param unemp_data A data frame containing
 ↪  unemployment data for Luxembourg.
#' @param year_of_interest Optional: The year that
 ↪  should be kept. Leave empty to select every
 ↪  year.
#' @param place_name_of_interest Optional: The name
 ↪  of the place of interest: leave empty to select
 ↪  every place in `level_of_interest`.
#' @param level_of_interest Optional: The level of
 ↪  interest: one of `Country`, `Canton`, `Commune`.
 ↪  Leave empty to select every level with the same
 ↪  place name.
```

```
#' @param col_of_interest A column of the `unemp`
 ↪  data frame that you wish to select.
#' @importFrom janitor clean_names
#' @importFrom dplyr filter select
#' @importFrom rlang quo `!!`
#' @return A data frame
#' @export
#' @details
#' Users can filter data on two variables: the name
 ↪  of the place of interest, and the level of
 ↪  interest.
#' By leaving the argument `place_name_of_interest`
 ↪  empty
#' @examples
#' # Filter on cantons
#' clean_unemp(unemp,
#'             level_of_interest = "Canton",
#'             col_of_interest = active_population)
#' # Filter on a specific commune
#' clean_unemp(unemp,
#'             place_name_of_interest =
 ↪  "Luxembourg",
#'             level_of_interest = "Commune",
#'             col_of_interest = active_population)
#' # Filter on every level called Luxembourg
#' clean_unemp(unemp,
#'             place_name_of_interest =
 ↪  "Luxembourg",
#'             col_of_interest = active_population)
clean_unemp <- function(unemp_data,
                        year_of_interest = NULL,
                        place_name_of_interest =
                         ↪  NULL,
```

```
                        level_of_interest = NULL,
                        col_of_interest){

  if(is.null(year_of_interest)){

    year_of_interest <- quo(year)

  }

  if(is.null(place_name_of_interest)){

    place_name_of_interest <- quo(place_name)

  }

  if(is.null(level_of_interest)){

    level_of_interest <- quo(level)

  }

  result <- unemp_data |>
    janitor::clean_names() |>
    dplyr::filter(year %in% !!year_of_interest,
                  place_name %in%
↪   !!place_name_of_interest,
                  level %in% !!level_of_interest) |>
    dplyr::select(year, place_name, level,
↪   {{col_of_interest}})

  if(nrow(result) == 0) {
    warning("The returned data frame is empty. This
     ↪  is likely because the
     ↪  `place_name_of_interest` or
     ↪  `level_of_interest` argument supplied does
     ↪  not match any rows in the original data.")
```

```
  }
  result
}
```

There's a lot more going on now: if you don't get everything that's going on in this function, don't worry, it is not that important for what follows. But do try to understand what's happening, especially the part about the optional arguments.

## 5.4 And back to testing

Running our tests now will obviously fail:

```
  devtools::test('.')

  Testing myPackage
  | F W S  OK | Context
  | 6       0 | clean_unemp [0.3s]

Error (test-clean_unemp.R:5:3): selecting the grand
duchy works
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
 1. myPackage::clean_unemp(...)
      at test-clean_unemp.R:5:2
 2. base::grepl("Grand-D.*", commune)
      at myPackage/R/functions.R:29:2
 3. base::is.factor(x)

Error (test-clean_unemp.R:21:3): selecting cantons
work
```

```
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
 1. myPackage::clean_unemp(...)
      at test-clean_unemp.R:21:2
 2. base::grepl("Canton", commune)
      at myPackage/R/functions.R:29:2
 3. base::is.factor(x)

Error (test-clean_unemp.R:34:3): selecting communes
works
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
 1. myPackage::clean_unemp(...)
      at test-clean_unemp.R:34:2
 2. base::grepl("(Canton|Grand-D.*)", commune)
      at myPackage/R/functions.R:29:2
 3. base::is.factor(x)

Error (test-clean_unemp.R:47:3): selecting one
commune works
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
 1. myPackage::clean_unemp(unemp_2013, grepl("Kayl",
 commune), active_population)
      at test-clean_unemp.R:47:2
 2. base::grepl("Kayl", commune)
      at myPackage/R/functions.R:29:2
 3. base::is.factor(x)

Error (test-clean_unemp.R:63:3): wrong commune name
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
```

```
 1. myPackage::clean_unemp(unemp_2013,
 grepl("Paris", commune), active_population)
      at test-clean_unemp.R:63:2
 2. base::grepl("Paris", commune)
      at myPackage/R/functions.R:29:2
 3. base::is.factor(x)

Error (test-clean_unemp.R:80:3): wrong commune name:
warning is thrown
Error in `is.factor(x)`: object 'commune' not found
Backtrace:
 1. testthat::expect_warning(...)
      at test-clean_unemp.R:80:2
 8. base::grepl("Paris", commune)
      at myPackage/R/functions.R:29:2
 9. base::is.factor(x)


  Results

Duration: 0.4 s

[ FAIL 6 | WARN 0 | SKIP 0 | PASS 0 ]
Warning message:
  Conflicts

myPackage conflicts

  `clean_unemp` masks `myPackage::clean_unemp()`.
  Did you accidentally source a file rather than
using `load_all()`?
  Run `rm(list = c("clean_unemp"))` to remove the
  conflicts.
```

>

At this stage, it might be a good idea to at least commit. Maybe let's not push yet, and only push once the tests have been rewritten to pass. Commit from RStudio or from a terminal, the choice is yours. We now have to rewrite the tests, to make them pass again. We also need to recreate the csv files for some of the tests, and will probably need to create others. This is what the script containing the tests could look like once you're done:

```
test_that("selecting the grand duchy works", {

  returned_value <- clean_unemp(
    unemp,
    year_of_interest = 2013,
    level_of_interest = "Country",
    col_of_interest = active_population) |>
      as.data.frame()

  expected_value <- as.data.frame(
                                list("year" = 2013,
                                     "place_name" =
                                      ↪  "Luxembourg",
                                     "level" =
                                      ↪  "Country",

                                      ↪  "active_population"
                                      ↪  = 242694)
                                )

  expect_equal(returned_value, expected_value)

})
```

```
test_that("selecting cantons work", {

  returned_value <- clean_unemp(
    unemp,
    year_of_interest = 2013,
    level_of_interest = "Canton",
    col_of_interest = active_population) |>
      as.data.frame()

  expected_value <-
↪   read.csv("test_data_cantons.csv")

  expect_equal(returned_value, expected_value)

})

test_that("selecting communes works", {

  returned_value <- clean_unemp(
    unemp,
    year_of_interest = 2013,
    level_of_interest = "Commune",
    col_of_interest = active_population) |>
      as.data.frame()

  expected_value <-
↪   read.csv("test_data_communes.csv")

  expect_equal(returned_value, expected_value)

})

test_that("selecting one commune works", {
```

```
  returned_value <- clean_unemp(
    unemp,
    year_of_interest = 2013,
    place_name_of_interest = "Kayl",
    col_of_interest = active_population) |>
      as.data.frame()

  expected_value <- as.data.frame(
                               list("year" = 2013,
                                    "place_name" =
                                     ↪  "Kayl",
                                    "level" =
                                     ↪  "Commune",

                                     ↪  "active_population"
                                     ↪  = 3863))

  expect_equal(returned_value, expected_value)

})

test_that("wrong commune name", {

  returned_value <- clean_unemp(
    unemp,
    year_of_interest = 2013,
    place_name_of_interest = "Paris",
    col_of_interest = active_population) |>
      as.data.frame()

  expected_value <- as.data.frame(
                               list("year" =
                                ↪  numeric(0),
```

```
                                    "place_name" =
                                     ↪  character(0),
                                    "level" =
                                     ↪  character(0),

                                     ↪  "active_population"
                                     ↪  =
                                     ↪  numeric(0)))


  expect_equal(returned_value, expected_value)

})

test_that("wrong commune name: warning is thrown", {

  expect_warning({
    clean_unemp(
      unemp,
      year_of_interest = 2013,
      place_name_of_interest = "Paris",
      col_of_interest = active_population)
  }, "This is likely")

})
```

Once you're done, commit and push your changes.

You should now have a pretty good intuition about unit tests. As you can see, unit tests are not just useful to make sure that changes that get introduced in our functions don't result in regressions in our code, but also to actually improve our code.

Writing unit tests allows us to fill the shoes of our users and rethink our code.

A little sidenote before continuing; you might want to look into *code coverage* using the `{covr}` package. This package helps you identify code from your package that is not tested yet. The goal of course being to improve the coverage as much as possible! Take a look at `{cover}`'s website to learn more.

Ok, one final thing; let's say that we're happy with our package. To actually use it in other projects we have to install it to our library. To do so, make sure RStudio is inside the right project, and press `CTRL-SHIFT-B`. This will install the package to our library.

# 6 Setting up pipelines with {targets}

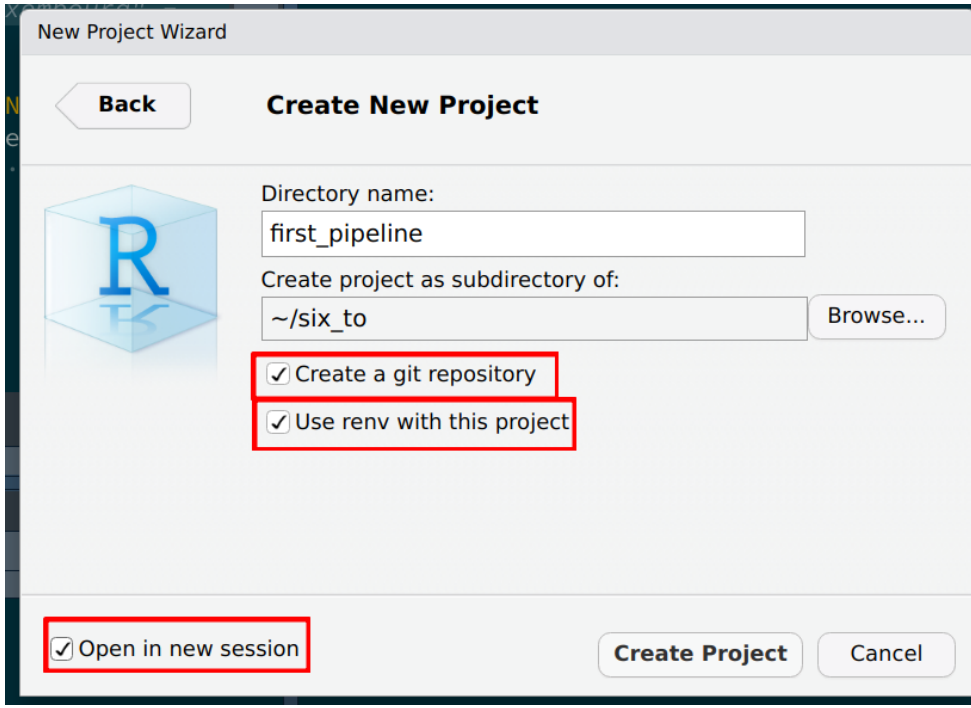
What you'll have learned by the end of the chapter: how to set up an (almost) reproducible pipeline.

## 6.1 Introduction

`{targets}` is a build automation tool for the R programming language. *Build*, in the context of this course means the creation of a data product. As mentioned in the introduction, this data product can be anything from predictions from a model to interactive web applications. *Automation* means that this build automation tool will take the burden off our shoulders when it'll be time to run the build pipeline. Using such a tool, programmers don't need to think about which parts of the code to rerun if they introduce a change somewhere. Only the parts affected by the change will run. These tools also run the pipeline in parallel, because they identify independent parts of the pipeline which are then run simultaneously. Build automation tools have many benefits, and because they work in a certain way, they also force you to work in a more structured way.

## 6.2 Build automation with R

As an introduction, there really is not a better source than the `{targets}` manual itself, and in particular the walkthrough section. After reading this section, we have the basic ledge to build our first pipeline. The goal of this pipeline will be to simply create a plot using the unemployment data we've been working on. Let's create a new project in RStudio, but make sure that you check the following boxes:




You should see the following message in the R console:

```
* Initializing project ...
* Discovering package dependencies ... Done!
* Copying packages into the cache ... Done!
The following package(s) will be updated in the
lockfile:

# CRAN ===============================
- renv   [* -> 0.16.0]

The version of R recorded in the lockfile will be
updated:
- R      [*] -> [4.2.1]
```

```
* Lockfile written to
'~/six_to/first_pipeline/renv.lock'.
```

This is something that we have not discussed yet, so before we move on, let's have a little aside on what {renv} is.

## 6.3 An aside on {renv}

Whether they're simple scripts to analyze some data or more complex reproducible analytical pipelines, all of your projects depend on the packages that you use for the analysis. And these packages evolve and change. It can very well happen that a function that you use from package {xyz} version 1 won't be available anymore in version 2. Or maybe it's still available but it works slightly differently. It can be something as trivial as the arguments of the function have been renamed. The consequence is that when you'll try to rerun your code, it won't work at all, or worse, it'll work, but produce a result that is not comparable to old results anymore, because the function got changed and the underlying algorithm isn't the same anymore. So we need a certain stability, and ideally keep reusing the same packages for the same project. If you want to update a project to use new packages version, this can of course also be done, but it has to be conscious choice and you will have to make sure that the updated pipeline (using the updated packages) is able to reproduce old results before putting it into production.

It must be noted however that in my experience, it is usually possible to rerun old R code without much hassle. But that's what makes it worse; you get so used to this stability that you don't think about a way to keep your projects reproducible, because issues rarely happen with R. It is thus best to get into the

habit to use a tool like {renv}, which offers a certain stability to your projects.

{renv} creates separate package libraries, one per project. The idea is quite simple; start your project with {renv} enabled (if you're using RStudio, you can check the box *Use renv with this project* when starting a new project, if you're not using RStudio, you can run `renv::init()`, on the root of the folder's project). This will create a file in the root of your project called `renv.lock`. You can open this file in a text editor, and you should see that the R version you're currently using is recorded (and the version of {renv} itself). It should look something like this:

```
{
  "R": {
    "Version": "4.2.1",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "http://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "renv": {
      "Package": "renv",
      "Version": "0.16.0",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "c9e8442ab69bc21c9697ecf856c1e6c7",
      "Requirements": []
    }
  }
}
```

You can then work as usual. It doesn't matter if you're simply writing a script to perform a "simple" analysis, or doing something more complex like a RAP. You will likely need to re-install packages though; remember, `{renv}` sets up a library per project!

Once you're done and satisfied, run `renv::snapshot()`. As you might have guessed from the name this will take a snapshot of the project and write the current status to the `renv.lock` file:

```
> renv::snapshot()
The following package(s) will be updated in the
lockfile:

# CRAN ===============================
- R6           [* -> 2.5.1]
- cli          [* -> 3.4.1]
- dplyr        [* -> 1.0.10]
- fansi        [* -> 1.0.3]
- generics     [* -> 0.1.3]
- glue         [* -> 1.6.2]
- lifecycle    [* -> 1.0.3]
- magrittr     [* -> 2.0.3]
- pillar       [* -> 1.8.1]
- pkgconfig    [* -> 2.0.3]
- rlang        [* -> 1.0.6]
- tibble       [* -> 3.1.8]
- tidyselect   [* -> 1.2.0]
- utf8         [* -> 1.2.2]
- vctrs        [* -> 0.4.2]
- withr        [* -> 2.5.0]

Do you want to proceed? [y/N]:
```

In it, you will see that the libraries needed to run the project are also recorded. The `renv.lock` will now look like this:

```
{
  "R": {
    "Version": "4.2.1",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "http://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "R6": {
      "Package": "R6",
      "Version": "2.5.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "470851b6d5d0ac559e9d01bb352b4021",
      "Requirements": []
    },
    "cli": {
      "Package": "cli",
      "Version": "3.4.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "0d297d01734d2bcea40197bd4971a764",
      "Requirements": []
    },
    "dplyr": {
      "Package": "dplyr",
      "Version": "1.0.10",
      "Source": "Repository",
```

```
    "Repository": "CRAN",
    "Hash": "539412282059f7f0c07295723d23f987",
    "Requirements": [
      "R6",
      "generics",
      "glue",
      "lifecycle",
      "magrittr",
      "pillar",
      "rlang",
      "tibble",
      "tidyselect",
      "vctrs"
    ]
  },
  "fansi": {
    "Package": "fansi",
    "Version": "1.0.3",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "83a8afdbe71839506baa9f90eebad7ec",
    "Requirements": []
  },
  "generics": {
    "Package": "generics",
    "Version": "0.1.3",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "15e9634c0fcd294799e9b2e929ed1b86",
    "Requirements": []
  },
  "glue": {
    "Package": "glue",
```

```
    "Version": "1.6.2",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "4f2596dfb05dac67b9dc558e5c6fba2e",
    "Requirements": []
  },
  "lifecycle": {
    "Package": "lifecycle",
    "Version": "1.0.3",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "001cecbeac1cff9301bdc3775ee46a86",
    "Requirements": [
      "cli",
      "glue",
      "rlang"
    ]
  },
  "magrittr": {
    "Package": "magrittr",
    "Version": "2.0.3",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "7ce2733a9826b3aeb1775d56fd305472",
    "Requirements": []
  },
  "pillar": {
    "Package": "pillar",
    "Version": "1.8.1",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "f2316df30902c81729ae9de95ad5a608",
    "Requirements": [
```

```
      "cli",
      "fansi",
      "glue",
      "lifecycle",
      "rlang",
      "utf8",
      "vctrs"
    ]
  },
  "pkgconfig": {
    "Package": "pkgconfig",
    "Version": "2.0.3",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "01f28d4278f15c76cddbea05899c5d6f",
    "Requirements": []
  },
  "renv": {
    "Package": "renv",
    "Version": "0.16.0",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "c9e8442ab69bc21c9697ecf856c1e6c7",
    "Requirements": []
  },
  "rlang": {
    "Package": "rlang",
    "Version": "1.0.6",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "4ed1f8336c8d52c3e750adcdc57228a7",
    "Requirements": []
  },
```

```
  "tibble": {
    "Package": "tibble",
    "Version": "3.1.8",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "56b6934ef0f8c68225949a8672fe1a8f",
    "Requirements": [
      "fansi",
      "lifecycle",
      "magrittr",
      "pillar",
      "pkgconfig",
      "rlang",
      "vctrs"
    ]
  },
  "tidyselect": {
    "Package": "tidyselect",
    "Version": "1.2.0",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "79540e5fcd9e0435af547d885f184fd5",
    "Requirements": [
      "cli",
      "glue",
      "lifecycle",
      "rlang",
      "vctrs",
      "withr"
    ]
  },
  "utf8": {
    "Package": "utf8",
```

```
      "Version": "1.2.2",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "c9c462b759a5cc844ae25b5942654d13",
      "Requirements": []
    },
    "vctrs": {
      "Package": "vctrs",
      "Version": "0.4.2",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "0e3dfc070b2a8f0478fcdf86fb33355d",
      "Requirements": [
        "cli",
        "glue",
        "rlang"
      ]
    },
    "withr": {
      "Package": "withr",
      "Version": "2.5.0",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "c0e49a9760983e81e55cdd9be92e7182",
      "Requirements": []
    }
  }
}
```

On your end, you're done. You can push this project to github.com for instance, and someone else who wishes to run this project will have to simply:

- Clone the repository;

- Run `renv::restore()` to install all the required libraries.

And that's it! This person, who might be future you, will now be able to re-run the project with the required libraries and the right versions.

In a coming section we are actually going to do just that, but for now, let's go back to `{targets}`.

## 6.4 Our actual first pipeline

Let's start by building our very first pipeline. Our goal is the following: start with the unemployment data for Luxembourg, and build a series of graphs. We want one graph for Luxembourg, one graph for cantons, and one graph for selected communes. This series of graphs will be our data product; let's not focus too much on the data product itself, the focus here is on building the pipeline. In the next chapter we are going to build more interesting data products.

I've been mentioning pipelines for some time now, but what is it actually? Nothing more than a script. Let's go back to the project we started at the beginning of the chapter. We can now create a `_targets.R` file on the root of the project. Insert the following lines in the script, we will then go through each of them:

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)

list(
  tar_target(unemp_data, get_data())
)
```

The first line simply loads the `{targets}` package, the second to fourth lines load the packages required for the pipeline to actually run. Then comes a list. Inside this list is where we will define the targets, or the (intermediary) outputs of the pipeline. We defined `unemp_data` as being the output of the function `get_data()`… but where does this function come from? Well, we need to create another script called `functions.R` where we will define every function the we need for this pipeline. Let's create an empty script and put the following lines in it:

```
get_data <- function(){
  myPackage::unemp
}
```

This function is as simple as it gets; it doesn't take any arguments and returns the data that we added to our package. This function could of course just as well read data from your computer, or from the Internet, and return it. The fact our data is inside our package is just for convenience. We can now go back to `_targets.R` and first of all, add a line to source this file, and then define new targets:

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)
source("functions.R")

list(
  tar_target(unemp_data, get_data())
)
```

You can now run the pipeline using `tar_make()`... and you should get immediately an error:

```
> targets::tar_make()
• start target unemp_data
  error target unemp_data
• end pipeline [0.198 seconds]
```

This is because our package is not in the `renv.lock` file. Remember that {renv} creates a new library per project, and as

such we now need to install `{myPackage}` from github.com into our project. For this, run the following line (you might need to install `{devtools}` beforehand):

```
devtools::install_github("b-rodrigues/myPackage",
                         ref =
                          ↪  "e9d9129de3047c1ecce26d09dff429e
```

The provided hash will make sure that the right version of the package gets installed for this project. This way, if I continue to work on the package, users will be able to still install the correct version. Since we've installed some new packages, run `renv::snapshot()` to rewrite the `renv.lock` file:

```
> renv::snapshot()
The following package(s) will be updated in the
lockfile:

# CRAN ===============================
- Matrix       [* -> 1.4-1]
- backports    [* -> 1.4.1]
- base64url    [* -> 1.4]
- callr        [* -> 3.7.2]
- codetools    [* -> 0.2-18]
- data.table   [* -> 1.14.4]
- digest       [* -> 0.6.30]
- evaluate     [* -> 0.17]
- highr        [* -> 0.9]
- igraph       [* -> 1.3.5]
- knitr        [* -> 1.40]
- lattice      [* -> 0.20-45]
- processx     [* -> 3.7.0]
- ps           [* -> 1.7.1]
```

```
- stringi      [* -> 1.7.8]
- stringr      [* -> 1.4.1]
- targets      [* -> 0.13.5]
- xfun         [* -> 0.34]
- yaml         [* -> 2.3.6]

# GitHub ============================
- myPackage    [* ->
b-rodrigues/myPackage@e9d9129de3047c1ecce26d09dff429ec078d4dae]

Do you want to proceed? [y/N]:
```

It might be a good idea to take a look at the lock file, and in particular the `{myPackage}` entry. If everything went alright, you should see something like this:

```
"myPackage": {
  "Package": "myPackage",
  "Version": "0.1.0",
  "Source": "GitHub",
  "RemoteType": "github",
  "RemoteHost": "api.github.com",
  "RemoteRepo": "myPackage",
  "RemoteUsername": "b-rodrigues",
  "RemoteRef":
  "e9d9129de3047c1ecce26d09dff429ec078d4dae",
  "RemoteSha":
  "e9d9129de3047c1ecce26d09dff429ec078d4dae",
  "Hash": "4740b43847e10e012bad2b8a1a533433",
  "Requirements": [
    "dplyr",
    "janitor",
    "rlang"
```

```
  ]
},
```

It can happen that every entry starting with "Remote" is missing. This depends how this package was installed, and if the `DESCRIPTION` file of this package contains the required info. If you installed it using `devtools::install_github()`, it should be fine. But it is always better to check. In case these are missing, you should add them by hand. Adding these fields will ensure that the package will always get installed from github.com, and that the correct commit will be used. You can then save the lock file and commit it alongside the rest of your project.

Ok, now we should be able to run our pipeline with `targets::tar_make()`.

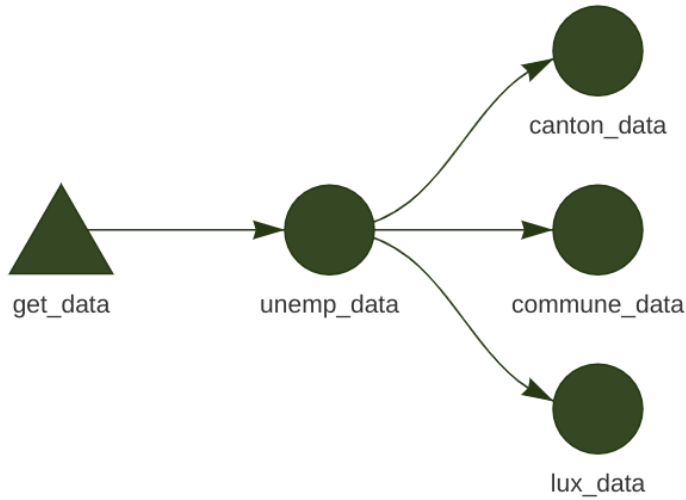
This will not create any output, but this way you can at least test that it's running. Also, remember `tar_load()` and `tar_read()` from the walkthrough? First try to run `tar_read(unemp_data)`, this should print the data in your console. You can then run `tar_load(unemp_data)`, which this time loads the data in your global environment. You can now access it interactively. This is quite useful if you need to inspect intermediary outputs.

Let's now add intermediary outputs; we need data for Luxembourg, data for the cantons and data for some communes. This is where the function `clean_unemp()` from our package will come into play:

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)
source("functions.R")
```

```
list(
  tar_target(
    unemp_data,
    get_data()
  ),

  tar_target(
    lux_data,
    clean_unemp(unemp_data,
                place_name_of_interest =
                 ↪  "Luxembourg",
                level_of_interest = "Country",
                col_of_interest = active_population)
  ),

  tar_target(
    canton_data,
    clean_unemp(unemp_data,
                level_of_interest = "Canton",
                col_of_interest = active_population)
  ),

  tar_target(
    commune_data,
    clean_unemp(unemp_data,
                place_name_of_interest =
                 ↪  c("Luxembourg", "Dippach",
                 ↪  "Wiltz", "Esch/Alzette",
                 ↪  "Mersch"),
                col_of_interest = active_population)
  )
)
```

At this stage it might be interesting to take a look at the network. Call `tar_visnetwork()` (you might get prompted to install yet another package) and take a look at the pipeline:




All that's missing now is to write a function to create plots. Since we didn't learn how to make them using `{ggplot2}`, simply copy the code below into the `functions.R` script:

```
make_plot <- function(data){
  ggplot(data) +
    geom_col(
      aes(
        y = active_population,
```

```
        x = year,
        fill = place_name
      )
    ) +
    theme(legend.position = "bottom",
          legend.title = element_blank())
}
```

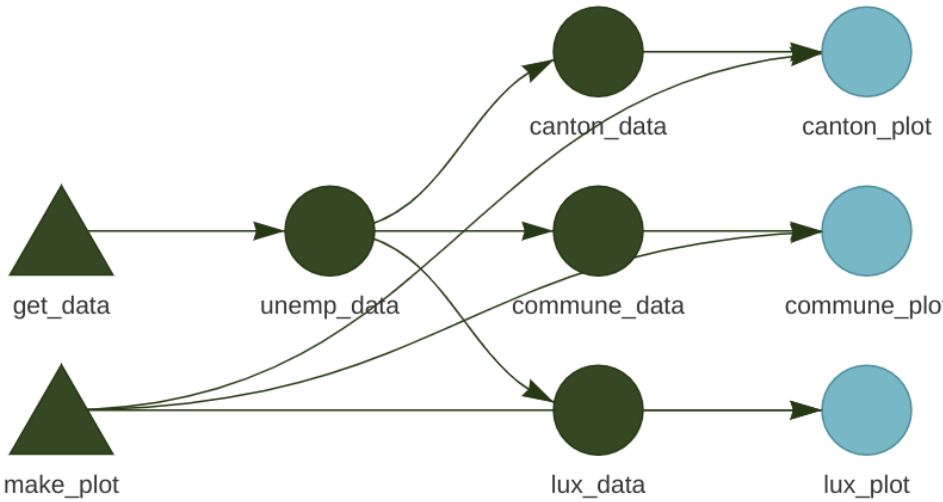
We can now use this function to define new targets in our `_targets.R` file:

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)
source("functions.R")


list(
    tar_target(
        unemp_data,
        get_data()
    ),

    tar_target(
        lux_data,
        clean_unemp(unemp_data,
                    place_name_of_interest =
                     ↪  "Luxembourg",
                    level_of_interest = "Country",
                    col_of_interest =
                     ↪  active_population)
```

```
    ),

    tar_target(
        canton_data,
        clean_unemp(unemp_data,
                    level_of_interest = "Canton",
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        commune_data,
        clean_unemp(unemp_data,
                    place_name_of_interest =
                     ↪  c("Luxembourg", "Dippach",
                     ↪  "Wiltz", "Esch/Alzette",
                     ↪  "Mersch"),
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        lux_plot,
        make_plot(lux_data)
    ),

    tar_target(
        canton_plot,
        make_plot(canton_data)
    ),

    tar_target(
        commune_plot,
```

```
        make_plot(commune_data)
    )

)
```

Let's now take a look at the pipeline again with `tar_visnetwork()`:




We see that new targets are outdated, and we need to run the pipeline to build them, so run the pipeline using `tar_make()`. If everything went well, we can now take a look at our plots using `tar_read(luxembourg_plot)`.

Finally, let's write these plots to disk. The way to save a `ggplot` to disk is to use the `ggsave()` function. But targets have to return something, so side-effects like writing to disk must be handled in a specific way. What we're going to do is write a wrapper around `ggsave()` that will take the path where the plot should be save to disk, save the plot to the specified path, and then return the plot. This way, we have a function that does not only have a side-effect, but also a return value. Let's go back to `functions.R` and add the following lines:

```
save_plot <- function(save_path, plot){
  ggsave(save_path, plot)
  save_path
}
```

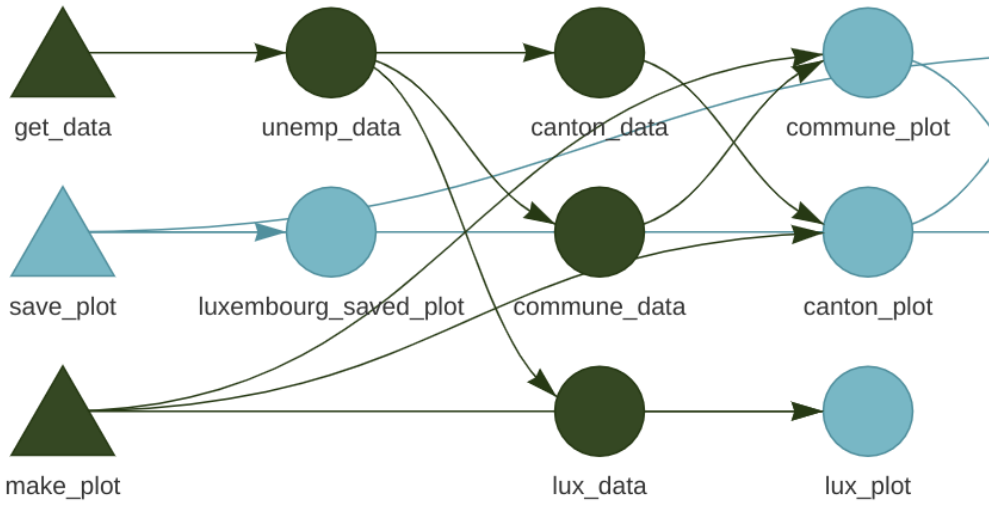
We can now define these additional targets:

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)
source("functions.R")


list(
    tar_target(
        unemp_data,
        get_data()
    ),

    tar_target(
        lux_data,
        clean_unemp(unemp_data,
```

```
                    place_name_of_interest =
                     ↪  "Luxembourg",
                    level_of_interest = "Country",
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        canton_data,
        clean_unemp(unemp_data,
                    level_of_interest = "Canton",
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        commune_data,
        clean_unemp(unemp_data,
                    place_name_of_interest =
                     ↪  c("Luxembourg", "Dippach",
                     ↪  "Wiltz", "Esch/Alzette",
                     ↪  "Mersch"),
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        lux_plot,
        make_plot(lux_data)
    ),

    tar_target(
        canton_plot,
```

```
        make_plot(canton_data)
    ),

    tar_target(
        commune_plot,
        make_plot(commune_data)
    ),

    tar_target(
        luxembourg_saved_plot,
        save_plot("fig/luxembourg.png", lux_plot),
        format = "file"
    ),

    tar_target(
        canton_saved_plot,
        save_plot("fig/canton.png", canton_plot),
        format = "file"
    ),

    tar_target(
        commune_saved_plot,
        save_plot("fig/commune.png", commune_plot),
        format = "file"
    )


)
```

Let's take a look at the network again:



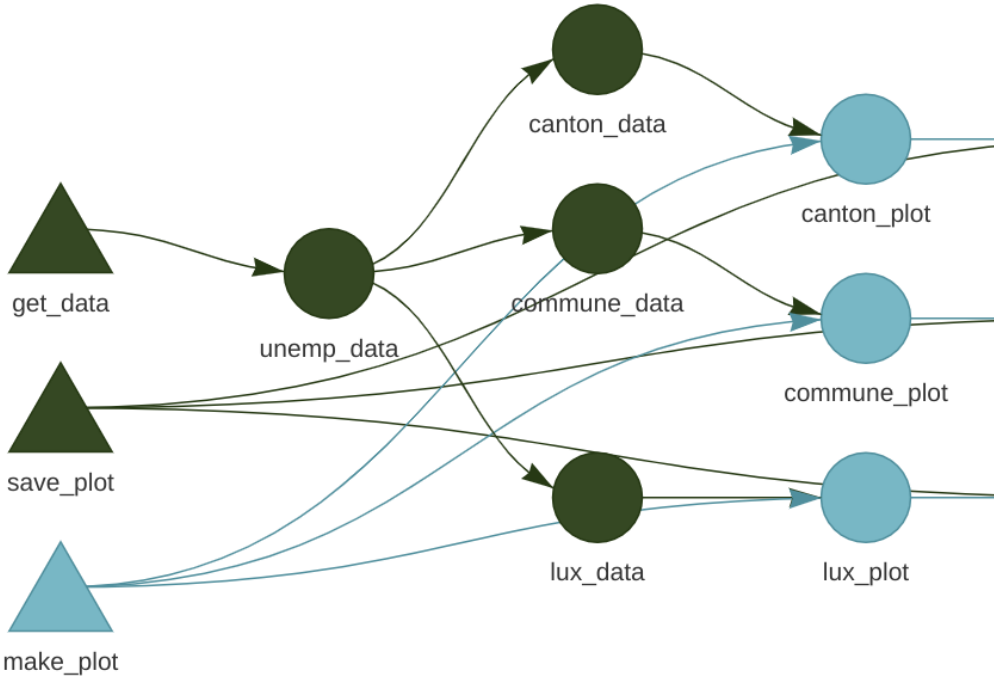

Because we are saving a file to disk, we must add the `format = "file"` argument to the target definiton. This way, `{targets}` watches these files for changes as well, and reruns the pipeline if a change is detected. Run the pipeline now with `tar_make()` and watch the plots appear in the `fig` folder (which you may have to create before running the pipeline).

Let's now change `make_plot()` function like this:

```
make_plot <- function(data){
  ggplot(data) +
    geom_col(
      aes(
        y = active_population,
        x = year,
```

```
        fill = place_name
      )
    ) +
    theme(legend.position = "bottom",
          legend.title = element_blank()) +
    labs(title = paste0("Unemployement for ",
     ↪  paste(unique(data$place_name), collapse = ",
     ↪  ")))
}
```

If you now save the script, and check the pipeline with `tar_visnetwork()`, you will see that some targets are not out of date:




Every targets that gets made by `make_plot()` must be recomputed, and every targets that depends on the intemediary outputs of `make_plot()` as well. Reloading the data is not necessary, since the edits on `make_plot()` do not affect these targets. This is an incredible cognitive load that is taken off the shoulders of data scientists; no need to keep track of outputs that are outdated, and no need to re-run everything either, saving lots of time and processing power.

You now know the basics of setting up a reproducible (well, almost, as you'll see) analytical pipeline. Let's now move to running someone else's pipeline.

## 6.5 Running someone else's pipeline

Let's now suppose that we want to run someone else's pipeline, and let's assume that that person did a good job and used `{renv}` to lock the dependencies of the pipeline, and also made the pipeline available on github.com.

As an example, we are going to use this repository from the author of `{targets}`. To start clone this repository:

```
git clone git@github.com:wlandau/targets-minimal.git
```

and open an R session in the root of the folder (or open the `targets-minimal.rproj` file in the folder you just cloned to open the project in RStudio), then call `renv::restore()`. You should see this:

```
> renv::restore()

This project has not yet been activated.
Activating this project will ensure the project
library is used during restore.
Please see `?renv::activate` for more details.

Would you like to activate this project before
restore? [Y/n]:
```

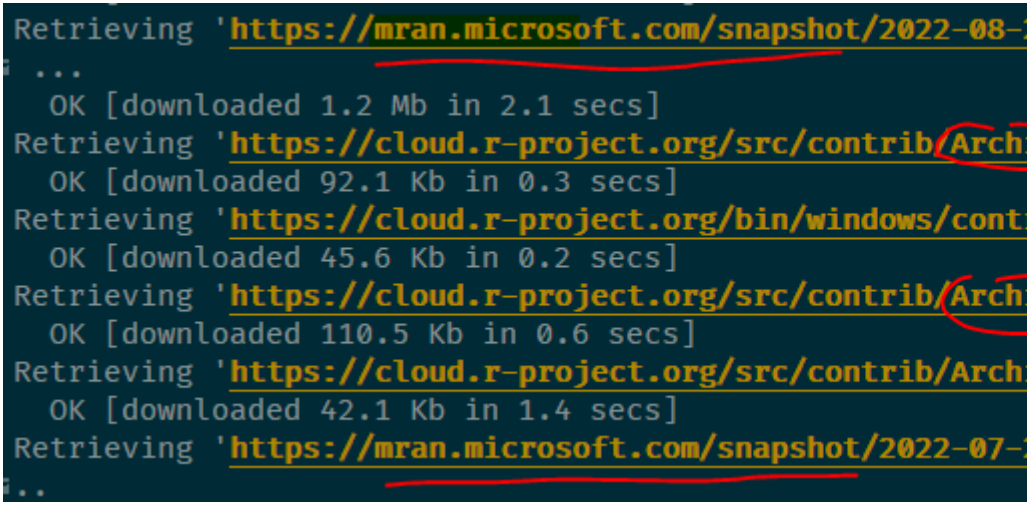
Press the `y` key on your keyboard to continue. The packages to run this pipeline will get installed in a new library, separate from the default library as usual with `{renv}`:

```
  Using R 4.2.1 (lockfile was generated with R 4.1.0)
* Project '~/targets-minimal' loaded. [renv 0.16.0]
```

```
* The project library is out of sync with the
lockfile.
* Use `renv::restore()` to install packages recorded
in the lockfile.
The following package(s) will be updated:

# CRAN ===============================
...very long list of packages I'm not showing
here...
Do you want to proceed? [y/N]:
```

Press Y to install the packages in the separate library.

As you can see from the screenshot below, because the packages used for this pipeline are not old(ish), they get either download from MRAN or from CRAN's archive (MRAN is the Microsoft R Archive Network, a mirror of CRAN maintained by Microsoft, which gets snapshotted every day. It is thus possible to download old packages from there):




You should now restart your R session for good measure (go to

`Session -> Restart Session`). You can now run the pipeline simply with:

```
targets::tar_make()
```

The data product (or output) from this pipeline is the `index.html` file that appeared on the root folder of your project.

## 6.6 Why we need more

While `{renv}` is a huge step towards the right direction, there are at least four problems with it:

- `{renv}` doesn't do anything about R itself: a pipeline made to run on R version 3 (for example) could still produce different results when run on R 4, even if the packages are the same. In practice, however, R is quite stable, and breaking changes between versions are very rare; most code is retrocompatible for many versions.
- `{renv}` doesn't do anything about the operating system the pipeline is running on. Results can even be different between different versions of the same operating system, but in practice, that should only affect you in very specific businesses were very high precision floating point arithmetic is required.
- `{renv}` can sometimes fail to install packages. I tried running William Landau's demo pipeline on two computers, one running OpenSuse Linux and one running Windows 10. It ran successfully on Linux, but not on Windows.

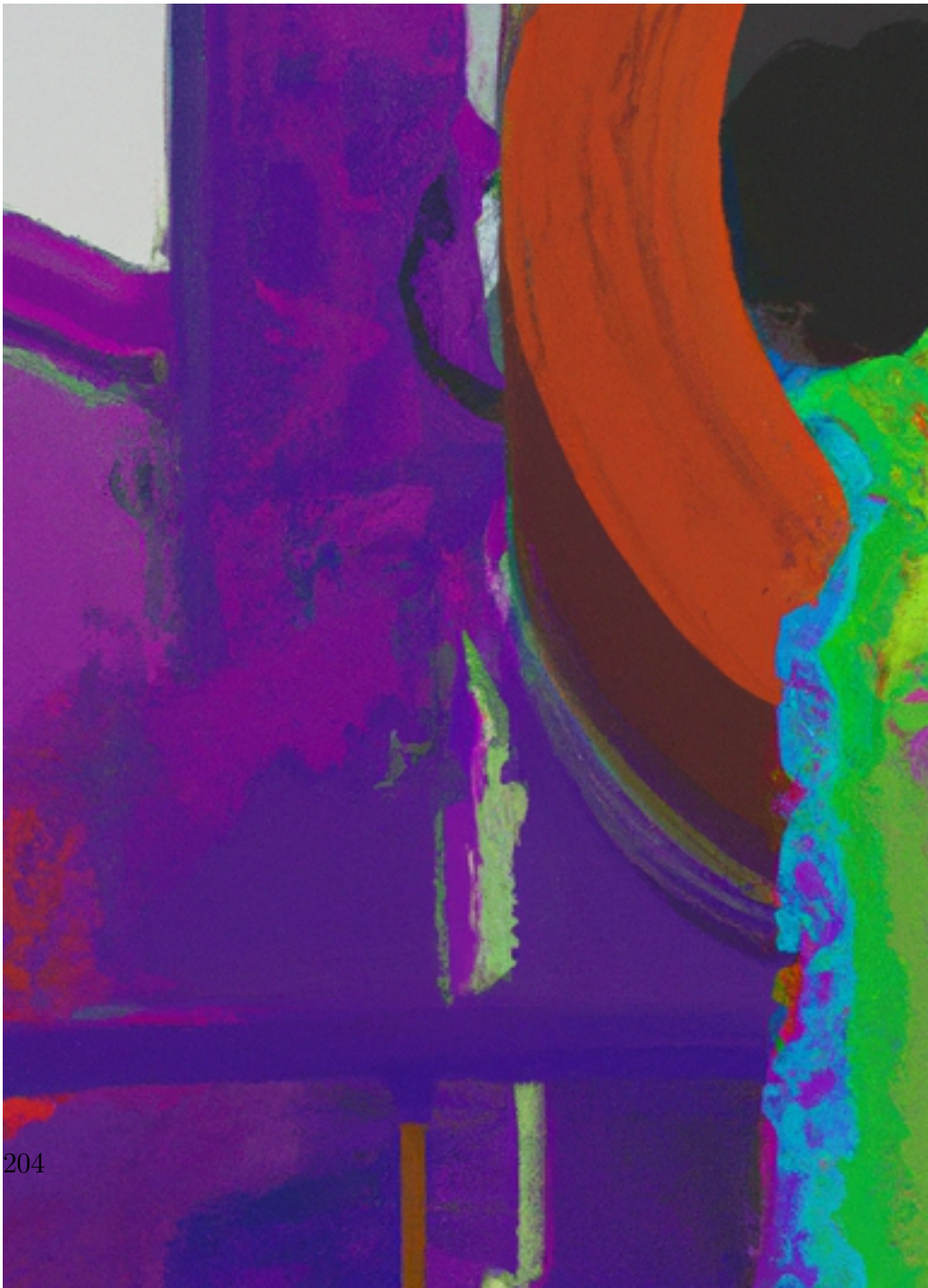
- {renv} relies on CRAN archives staying online. While it is very unlikely that CRAN will ever be offline, as there are many, many mirrors around the world, and it could be argued that the longer CRAN is online, the likelier it is it'll stay online, for very mission critical projects it might be needed to host your own mirror.

We are going to solve these issues in chapter 9, but for now, let's be grateful for {renv}, {targets} and CRAN, for they allow us to quite easily build (almost) reproducible pipelines quite easily! In the next chapter, we will continue building pipelines and build our very first data products.

## 6.7 Further reading

- The {targets} manual
- The {renv} website

# 7 Data products



What you'll have learned by the end of the chapter: you'll know how to build data products using Quarto and Shiny.

## 7.1 Introduction

We are going to start by building data products using Quarto.

Quarto is a tool created by Posit, the company behind RStudio and Shiny. Quarto leverages pandoc to convert between many document formats (for example, from `.md` to `.docx`) and makes it possible to embed R, Python, Julia and Observable JS code into documents. It is not an R-specific tool, so it is a program that you must install on your computer. So go to this link and download and install Quarto.

We are going to start simple, with "static" data products. By static I mean products without any sort of interactivity, so the user can look at them, read them, but not change them in any way. These products are essentially going to be documents in the `.docx`, `.pptx` and `.pdf` formats, but also `.html`. Thanks to Quarto, it is thus possible to programmatically create documents.

## 7.2 A first taste of Quarto

A Quarto file looks very close to a standard Markdown file. So if you know Markdown, you will not have many problems to switch to Quarto. If you don't know Markdown, no worries, its syntax is quite simple and can be very quickly picked up.

Let's start with a basic Quarto source. Open your favorite text editor (doesn't have to be RStudio) and create a file called `example.qmd` and copy the following lines in it:

````
---
title: "My Title"
author: "My name"
date: today
---

## This is a simple quarto document
```{r}
n <- 10
rnorm(n)
```
````

This is the output.

The first few lines of the document is where you can define the title of the document, the name of the author and the date. For the date, I've use the `today` keyword to get today's date but you could use a string to set the date to a specific day (for example, "2022-10-28"). The content in the document consists of a level 2 title (`## This is a simple quarto document`) and of an R code chunk. Code chunks is were you will write code that gets then evaluated at render (compile) time. To compile this file, run the following inside a terminal:

```
quarto render example.qmd
```

If you're inside RStudio, you can also render the document by pressing `CTRL-SHIFT-K` or run the command:

```
quarto::quarto_render("example.qmd")
```

There are various ways to integrate Quarto with different editors:

- VS Code
- RStudio
- Jupyter
- (Neo)Vim, Emacs, Sublime

Once the file is done rendering, you should find an `html` file in the same folder. Open this html file inside a web browser and see the output. It is possible to run arbitrary R code inside the code chunks:

````
---
title: "My Title"
author: "My name"
date: today
---

## This is a simple quarto document

```{r}
library(dplyr)
library(tidyr)
library(purrr)
library(ggplot2)
library(myPackage)

data("unemp")

unemp %>%
````
````

```
  janitor::clean_names() %>%
  filter(level == "Commune",
         place_name %in% c("Luxembourg",
 ↪  "Esch-sur-Alzette", "Wiltz")) %>%
  group_by(place_name) %>%
  nest() %>%
  mutate(plots = map2(.x = data, .y = place_name,
 ↪  ~ggplot(data = .x) +

 ↪  theme_minimal() +

 ↪  geom_line(aes(year,
 ↪  unemployment_rate_in_percent, group = 1)) +

 ↪  labs(title = paste("Unemployment in", .y)))) %>%
  pull(plots)
```

This is what the output looks like.

As you can see, it is quite easy to create a document with potentially hundreds of plots using what we've learned until now. However, our document does not look great; for starters, we see the source code there, which we would like to hide. People that will read this document might not be interested in the source code, but only in the plots. The other issue is that when loading the `{dplyr}` package, users get some message informing them about some functions that get masked. We would like to hide all of this. It turns out that code chunks have options, and we can use them to hide source code and warning messages:

````
---
title: "My Title"
author: "My name"
date: today
---

## This is a simple quarto document

```{r}
#| echo: false
#| warning: false

library(dplyr)
library(tidyr)
library(purrr)
library(ggplot2)
library(myPackage)

data("unemp")

unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune",
         place_name %in% c("Luxembourg",
 ↪  "Esch-sur-Alzette", "Wiltz")) %>%
  group_by(place_name) %>%
  nest() %>%
  mutate(plots = map2(.x = data, .y = place_name,
 ↪  ~ggplot(data = .x) +

 ↪  theme_minimal() +

 ↪  geom_line(aes(year,
 ↪  unemployment_rate_in_percent, group = 1)) +
````
````

```
 ↪  labs(title = paste("Unemployment in", .y)))) %>%
  pull(plots)
```

This is what the output looks like.

Rendering this document will result in something nicer. We could also fold the code instead of completely removing it. This is useful if we need to send the document to collaborators who might be interested in the source code as well. However, code folding is something that only works in `html` outputs, and thus we need to specify the output format in the header of the document (look at the three new lines after we define the data), and also remove the `echo: false` option from the R chunk:

````
---
title: "My Title"
author: "My name"
date: today
format:
  html:
    code-fold: true
---

## This is a simple quarto document

```{r}
#| warning: false

library(dplyr)
library(tidyr)
library(purrr)
````
````

```
library(ggplot2)
library(myPackage)

data("unemp")

unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune",
         place_name %in% c("Luxembourg",
 ↪  "Esch-sur-Alzette", "Wiltz")) %>%
  group_by(place_name) %>%
  nest() %>%
  mutate(plots = map2(.x = data, .y = place_name,
 ↪  ~ggplot(data = .x) +

 ↪  theme_minimal() +

 ↪  geom_line(aes(year,
 ↪  unemployment_rate_in_percent, group = 1)) +

 ↪  labs(title = paste("Unemployment in", .y)))) %>%
  pull(plots)
```

This is what the output looks like.

It is of course possible to write several R chunks:

```
---
title: "My Title"
author: "My name"
date: today
format:
```

```
  html:
    code-fold: true
---

## This is a simple quarto document

```{r}
#| warning: false

library(dplyr)
library(tidyr)
library(purrr)
library(ggplot2)
library(myPackage)

data("unemp")

unemp <- unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune")
```

There are `r length(unique(unemp$place_name))`
 ↪  communes in the dataset.
Below we plot the unemployment rate for 3 communes:

```{r}
unemp %>%
  filter(place_name %in% c("Luxembourg",
 ↪  "Esch-sur-Alzette", "Wiltz")) %>%
  group_by(place_name) %>%
  nest() %>%
  mutate(plots = map2(.x = data, .y = place_name,
 ↪  ~ggplot(data = .x) +
```
```

```
 ↪  theme_minimal() +

 ↪  geom_line(aes(year,
 ↪  unemployment_rate_in_percent, group = 1)) +

 ↪  labs(title = paste("Unemployment in", .y)))) %>%
  pull(plots)
```

This is what the output looks like.

### 7.2.1 Python and Julia code chunks

It is possible, inside the same Quarto document, to define code chunks that run Python (or even Julia) code. Put the following lines inside a file called `example2.qmd` (to run the example below, you will need to have Python installed):

````
---
title: "R and Python"
author: "Bruno Rodrigues"
date: today
---

## This is a simple quarto document

```{r}
#| warning: false

library(dplyr)
````
````

````
library(tidyr)
library(purrr)
library(ggplot2)
library(myPackage)

data("unemp")

unemp <- unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune")
```

```{python}
print("hello from Python")
import sys
print(sys.version)
```
````

This is what the output looks like.

If you have trouble rendering this line, make sure that you have the `jupyter` and `jupyterlab` modules installed.

It is also possible to pass objects from R to Python (and vice-versa):

```
---
title: "R and Python"
author: "Bruno Rodrigues"
date: today
---

## This is a simple quarto document
```
```

````
```{r}
#| warning: false

library(dplyr)
library(tidyr)
library(purrr)
library(ggplot2)
library(myPackage)

data("unemp")

unemp <- unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune")
```

The data that was loaded and cleaned from R can be
 ↪  accessed from Python using `r.unemp`:

```{python}
import pandas as pd
unemp_pd = pd.DataFrame(r.unemp)
unemp_pd.head
```
````

This is what the output looks like.

The HTML output is quite flexible, as it is possible to also integrate JS libraries. The following example uses the `{g2r}` library (an R wrapper around the g2 javascript library) for creating visualisations. To run the following code, make sure that you have the `{g2r}` package installed (can only be install from github):

```
devtools::install_github("devOpifex/g2r")
```

The source file looks like this:

````
---
title: "Quarto and JS libraries"
author: "My name"
date: today
format:
  html:
    code-fold: true
---

## This is a simple quarto document showing basic
 ↪  plot interactivity using {g2r}

```{r}
#| warning: false

library(dplyr)
library(tidyr)
library(purrr)
library(g2r)
library(myPackage)

data("unemp")

unemp <- unemp %>%
  janitor::clean_names() %>%
  filter(level == "Commune")
```
````

````
There are `r length(unique(unemp$place_name))`
 ↪  communes in the dataset. Below we plot the
 ↪  unemployment rate for 3 communes:

```{r}
unemp %>%
  filter(place_name %in% c("Luxembourg",
 ↪  "Esch-sur-Alzette", "Wiltz")) %>%
  g2(data = .) %>%
  fig_line(asp(year, unemployment_rate_in_percent,
 ↪  color = place_name))
```
````

This is what the output looks like.

It is possible to use other JS libraries, like here `DataTables`, wrapped inside the `{DT}` package:

````
---
title: "Quarto and JS libraries"
author: "My name"
date: today
format:
  html:
    toc: true
    code-fold: true
---

## Basic plot interactivity using {g2r}

```{r}
#| warning: false
````
````

```
library(dplyr)
library(tidyr)
library(purrr)
library(g2r)
library(DT)
library(myPackage)

data("unemp")

unemp <- unemp %>%
  mutate(year = as.character(year)) %>%
  janitor::clean_names() %>%
  filter(level == "Commune")
```

There are `r length(unique(unemp$place_name))` communes in the dataset. Below we plot the unemployment rate for 3 communes:

```{r}
unemp %>%
  filter(place_name %in% c("Luxembourg", "Esch-sur-Alzette", "Wiltz")) %>%
  g2(data = .) %>%
  fig_line(asp(year, unemployment_rate_in_percent, color = place_name))
```

## Interactive tables with {DT}

```{r}
unemp %>%
  DT::datatable(filter = "top")
```

```
```

## Others

You can find more widgets over
 ↪  [here](http://gallery.htmlwidgets.org/).
```

This is what the output looks like.

The final example illustrates templating. It is possible to write code that generates qmd code:

```
---
title: "Templating with Quarto"
author: "Bruno Rodrigues"
date: today
format:
  html:
    toc: true
---


# Set up
The goal is to have a frequency table for each
 ↪  question in a survey. But we
do not want to have to do it by hand, so we define a
 ↪  function to create a
table, and then, using the templating capabilities
 ↪  of Quarto, write some
code to generate valid qmarkdown code. In the
 ↪  example below our survey only
has 4 questions, but the solution described
 ↪  trivially scales to an infinity
```
```

```
of questions. This is not the case if you're solving
 ↪  this problem by hand.

Start by loading the data and defining some needed
 ↪  variables:

```{r}
#| warning: false
library(lubridate)
library(dplyr)
library(purrr)
library(rlang)
library(DT)
survey_data <- read.csv(
"https://gist.githubusercontent.com/b-rodrigues/0c2249dec5a9c94
)

```

Let's take a look at the data:

```{r}
datatable(survey_data)
```

The column names are actually questions, so we save
 ↪  those in a variable:

```{r}
questions <- colnames(survey_data)

questions
```
```

```
Now we define question codes:

```{r}
codes <- paste0("var_", seq(1, 4))

codes
```

We create a lookup table that links questions to
 ↪  their codes:

```{r}
lookup <- bind_cols("codes" = codes, "questions" =
 ↪  questions)

datatable(lookup)
```

Finally, we replace the question names in the
 ↪  dataset by the code:

```{r}
colnames(survey_data) <- codes

datatable(survey_data)
```

Now, we define a function that creates a frequency
 ↪  table. This function has
two arguments: `dataset` and `var`. It uses the
 ↪  `dplyr::count()` function to
count each instance of the levels of `var` in
 ↪  `dataset`. Then it uses the
```

```
`knitr::kable()` function. This functions takes a
 ↪  data frame as an argument
and returns a table formatted in markdown code:

```{r}
create_table <- function(dataset, var){
  dataset %>%
    count(!!var) %>%
    knitr::kable()
}
```

The next function is the one that does the magic: it
 ↪  takes only one argument
as an input, and generates valid markdown code using
 ↪  the `knitr::knit_expand()`
function. Any variable between `{{}}` gets replaced
 ↪  by its value (so
`{{question}}` gets replaced by the question that
 ↪  gets fetched from the
lookup table defined above). Using this function, we
 ↪  can now loop over
question codes, and what we get in return is valid
 ↪  markdown code that defines
a section with the question as the title, and our
 ↪  table.

```{r}
return_section <- function(var){
  a <- knitr::knit_expand(text = c("##
 ↪  {{question}}",   create_table(survey_data,
 ↪  var)),
                          question =
 ↪  lookup$questions[grepl(quo_name(var),
 ↪  lookup$codes)])
```
```

```
  cat(a, sep = "\n")
}
```

Our codes are strings, so to be able to use them inside of `dplyr::count()`
we need to define them as bare string, or symbols. This can be done using the
`rlang::sym()` function. If this is confusing, try running `count(mtcars, "am")`
and you will see that it will not return what you want (compare to `count(mtcars, am)`).
This is also why we needed `rlang::quo_name()` in the function above, to convert
the symbol back to a string, which is what `grepl()` requires:

```{r}
sym_codes <- map(codes, sym)
```

Finally, we can create the sections. The line below uses `purrr::walk()`, which
is equivalent to `purrr::map()`, the difference being that we use `purrr::walk()`
when we are interested in the side effects of a function:

```{r, results="asis"}
walk(sym_codes, return_section)
```

This is what the output looks like..

## 7.3 Other output formats

### 7.3.1 Word

Let's now generate a Word document using Quarto. As you will see, this will be quite easy; but keep in mind that the basic interactivity that we have seen with HTML outputs won't be possible here (but templating will work). Render the following source file to get back a `.docx` document (you don't even need to have MS Word installed for it to work), and take of what we changed from the previous file:

- Output changed from `html` to `docx`;
- No more `{DT}`, but `{pander}` instead to generated `.docx` tables

Here is the file:

```
---
title: "Templating with Quarto"
author: "Bruno Rodrigues"
date: today
format: docx
---


# Set up
The goal is to have a frequency table for each
 ↪  question in a survey. But we
do not want to have to do it by hand, so we define a
 ↪  function to create a
table, and then, using the templating capabilities
 ↪  of Quarto, write some
```

```
code to generate valid qmarkdown code. In the
 ↪  example below our survey only
has 4 questions, but the solution described
 ↪  trivially scales to an infinity
of questions. This is not the case if you're solving
 ↪  this problem by hand.

Start by loading the data and defining some needed
 ↪  variables:

```{r}
#| warning: false
library(lubridate)
library(dplyr)
library(purrr)
library(pander)
library(rlang)
survey_data <- read.csv(
"https://gist.githubusercontent.com/b-rodrigues/0c2249dec5a9c
)

```

Let's take a look at the data:

```{r}
pander(head(survey_data))
```

The column names are actually questions, so we save
 ↪  those in a variable:

```{r}
```
```

```
questions <- colnames(survey_data)

questions
```

Now we define question codes:

```{r}
codes <- paste0("var_", seq(1, 4))

codes
```

We create a lookup table that links questions to
↪ their codes:

```{r}
lookup <- bind_cols("codes" = codes, "questions" =
↪ questions)

pander(lookup)
```

Finally, we replace the question names in the
↪ dataset by the code:

```{r}
colnames(survey_data) <- codes

pander(survey_data)
```

Now, we define a function that creates a frequency
↪ table. This function has

```
two arguments: `dataset` and `var`. It uses the
 ↪  `dplyr::count()` function to
count each instance of the levels of `var` in
 ↪  `dataset`. Then it uses the
`knitr::kable()` function. This functions takes a
 ↪  data frame as an argument
and returns a table formatted in markdown code:

```{r}
create_table <- function(dataset, var){
  dataset %>%
    count(!!var) %>%
    knitr::kable()
}
```

The next function is the one that does the magic: it
 ↪  takes only one argument
as an input, and generates valid markdown code using
 ↪  the `knitr::knit_expand()`
function. Any variable between `{{}}` gets replaced
 ↪  by its value (so
`{{question}}` gets replaced by the question that
 ↪  gets fetched from the
lookup table defined above). Using this function, we
 ↪  can now loop over
question codes, and what we get in return is valid
 ↪  markdown code that defines
a section with the question as the title, and our
 ↪  table.

```{r}
return_section <- function(var){
```
```

```
  a <- knitr::knit_expand(text = c("##
 ↪  {{question}}",   create_table(survey_data,
 ↪  var)),
                           question =
 ↪  lookup$questions[grepl(quo_name(var),
 ↪  lookup$codes)])
  cat(a, sep = "\n")
}
```

```
Our codes are strings, so to be able to use them
 ↪  inside of `dplyr::count()`
we need to define them as bare string, or symbols.
 ↪  This can be done using the
`rlang::sym()` function. If this is confusing, try
 ↪  running `count(mtcars, "am")`
and you will see that it will not return what you
 ↪  want (compare to `count(mtcars, am)`).
This is also why we needed `rlang::quo_name()` in
 ↪  the function above, to convert
the symbol back to a string, which is what `grepl()`
 ↪  requires:
```

```{r}
sym_codes <- map(codes, sym)
```

```
Finally, we can create the sections. The line below
 ↪  uses `purrr::walk()`, which
is equivalent to `purrr::map()`, the difference
 ↪  being that we use `purrr::walk()`
when we are interested in the side effects of a
 ↪  function:
```

````
```{r, results="asis"}
walk(sym_codes, return_section)
```
````

You can download the output here.

Unlike with HTML outputs, it is also not possible to enable code folding, but you could hide the code completely using the "#| echo = false" chunk option. If you wan to hide all the code without having to specify "#| echo = false" on each chunk you can also add the `execute` option to the document header:

```
---
title: "Templating with Quarto"
author: "Bruno Rodrigues"
date: today
format: docx
execute:
  echo: false
---
```

You can use a document as a template for Word documents generated with Quarto. For this, you must create a new Word file, and update the styles. This document, with the updated styles, can then be referenced in the header to act as a template:

```
---
title: "Using a custom Word style"
author: "Bruno Rodrigues"
date: today
format:
```

```
  docx:
    reference-doc: fancy_template.docx
execute:
  echo: false
---

# Introduction

## MS Word is great (lol)

This is normal text that is unreadable.
```

Just put `fancy_template.docx` in the same folder as your source qmd file. You can download the template I've used from here to test things out.

For more details, visit this page.

### 7.3.2 Presentations

It is also possible to create presentations using Quarto. There are output formats as well: HTML, PDF and Powerpoint. I will not discuss this here, because it is quite easy to get started, simply follow along.

### 7.3.3 PDF

I do need to discuss the PDF output a little bit. In order to generate PDF files, Quarto uses the `pdflatex` compiler (or rather pandoc, called by Quarto, uses `pdflatex`). `pdflatex` compiles `.tex` source files to PDF, so what Quarto does (by leveraging

pandoc) is first converting a `.qmd` file to a `.tex` file, and then call `pdflatex` to compile it. `.tex` files are the file extension of the Latex typesetting language, extensively used in science. It makes it easy to write complex mathematical formulas, like this one:

$$
\begin{aligned}
S(\omega) &= \frac{\alpha g^2}{\omega^5} e^{[-0.74\left\{\frac{\omega U_\omega 19.5}{g}\right\}^{-4}]} \\
&= \frac{\alpha g^2}{\omega^5} \exp\left[-0.74\left\{\frac{\omega U_\omega 19.5}{g}\right\}^{-4}\right]
\end{aligned}
$$

Latex is a bit unwieldly, so using Markdown to write scientific documents is becoming more and more popular. However, Latex still has an edge when it comes to tables. But thankfully, it is possible to simply embed the Latex code that produces these tables in Markdown, and there are packages that export regression table directly to PDF. In any case, in order to compile to PDF, you need to install Texlive. Installing Texlive is frankly a mess, but thankfully there is a very simple alternative called TinyTex. TinyTex is both available as an R package or as a standalone installation, and was put together by the author of RMarkdown (in a sense, Quarto is a spiritual successor to RMarkdown). This package installs a self-contained Texlive installation locally, which can then be used to compile PDF documents (from, or outside of R/RStudio). I highly recommend you use Tinytex. Instructions can be found here. Once you've installed TinyTex, you can try to compile the following example document (the first time you run this, it might take some time, as the required packages get installed):

```
---
title: "PDF example with table"
```

```
format: pdf
---

## A PDF document using Quarto

In the code below, we fit several models and then
 ↪  use the `{modelsummary}`
package to print a nicely formatted table with
 ↪  minimal effort:

```{r}
library(modelsummary)

url <-
 ↪  'https://vincentarelbundock.github.io/Rdatasets/csv/HistDat
dat <- read.csv(url)

models <- list(
  "OLS 1"     = lm(Donations ~ Literacy + Clergy,
 ↪  data = dat),
  "Poisson 1" = glm(Donations ~ Literacy + Commerce,
 ↪  family = poisson, data = dat),
  "OLS 2"     = lm(Crime_pers ~ Literacy + Clergy,
 ↪  data = dat),
  "Poisson 2" = glm(Crime_pers ~ Literacy +
 ↪  Commerce, family = poisson, data = dat),
  "OLS 3"     = lm(Crime_prop ~ Literacy + Clergy,
 ↪  data = dat)
)

modelsummary(models)
```
And an equation, for good measure:
```

```
\begin{align*}
S(\omega)
&= \frac{\alpha g^2}{\omega^5} e^{[
 ↪  -0.74\bigl\{\frac{\omega U_\omega
 ↪  19.5}{g}\bigr\}^{\!-4}\,]} \\
&= \frac{\alpha g^2}{\omega^5} \exp\Bigl[
 ↪  -0.74\Bigl\{\frac{\omega U_\omega
 ↪  19.5}{g}\Bigr\}^{\!-4}\,\Bigr]
\end{align*}
```

This is what the output looks like (scroll down to page 2)..

It is possible to author, many, many, different types of documents using Quarto. For more formats, consult this page. Quarto is still very new – it was officially anounced in July of 2022 by Posit– so much more content will arrive. There are still many features of Quarto that we have not explored, so take your time to read its documentation in detail.

## 7.4 Interactive web applications with {shiny}

`{shiny}` is a package developed by Posit to build interactive web applications. These apps can be quite "simple" (for example, an app that shows a graph but in which the user can choose the variable to plot), but can be arbitrarily complex. Some people even go as far as make games with {shiny}. A version for Python is also in alpha, and you can already experiment with it.

In this section, I will give a very, very short introduction to `{shiny}`. This is because `{shiny}` is so feature-rich, that I could spend 20 hours teaching you and even then we would not have seen everything. That being said, we can with only some cursory knowledge build some useful apps. These apps can run locally on your machine, but they're really only useful if deploy them on a server, so that users can then use these web apps on their browsers.

### 7.4.1 The basic structure of a Shiny app

Shiny apps are always made of at least 2 parts: a *server* and a *ui*. In general, each of these parts are in separate scripts called `server.R` and `ui.R`. It is possible to have another script, called `global.R`, where you can define variables that you want to be available for both the server and the ui, and to every user of your app.

Let's start by building a very basic app. This app will allow users to visualize unemployment data for Luxembourg. For now, let's say that we want users only to be able to select communes, but not variables. The example code below is based on this official example (this is how I recommend you learn by the way. Take a look at the different example there are and adapt them to suit your needs! You can find the examples here). Create a folder called something like `my_app` and then create three scripts in it:

- `global.R`
- `server.R`
- `ui.R`

Let's start with `global.R`:

```
library(myPackage)
library(dplyr)
library(ggplot2)

data("unemp")
```

In the `global.R` file, we load the required packages and data. This is now available everywhere. Let's continue with the `server.R` script:

```
server <- function(session, input, output) {

  filtered_data <- reactive(
    unemp %>%
    filter(place_name %in%
     ↪  input$place_name_selected)
  )

  output$unemp_plot <- renderPlot({

    ggplot(data = filtered_data()) +
      theme_minimal() +
      geom_line(aes(year,
       ↪  unemployment_rate_in_percent, color =
       ↪  place_name)) +
      labs(title = paste("Unemployment in",
       ↪  paste(input$place_name_selected, collapse
       ↪  = ", ")))


  })
}
```

Several things need to be commented here: first, the script contains a single function, called `server()`. This function take three arguments, `session`, `input` and `output`. I won't go into details here, but you should know that you will never call the `server()` function yourself, and that these arguments are required so the function can… function. I will leave a reference at the end of this section with more details. The next important thing is that we defined an object called `filtered_data`. This is a reactive object. What this means is that this object should get recomputed every time the user interacts with it. But how does the user interact with it? By choosing the `place_name` he or she wants to see! The predicate inside `filter()` is `place_name %in% input$place_name_selected`. Where does that `input$place_name_selected` come from? This comes from the `ui` (that we have not written yet). But the idea is that the user will be able to chose place names from a list, and this list will be called `place_name_selected` and will contain the place names that the user wants to see.

Finally, we define a new object called `output$unemp_plot`. The goal of the `server()` function is to compute things that will be part of the `output` list. This list, and the objects it contains, get then rendered in the `ui`. `unemp_plot` is a ggplot graph that uses the reactive data set we defined first. Notice the `()` after `filtered_data` inside the ggplot call. These are required; this is how we say that the reactive object must be recomputed. If the plot does not get rendered, the reactive data set does not get computed, since it never gets called.

Ok so now to the ui. Let's take inspiration from the same example again:

```
ui <- function(request){
  fluidPage(
```

```
    titlePanel("Unemployment in Luxembourg"),

    sidebarLayout(

      sidebarPanel(
        selectizeInput("place_name_selected",
         ↪  "Select place:",

                        ↪  choices=unique(unemp$place_name),
                       multiple = TRUE,
                       selected = c("Rumelange",
                        ↪  "Dudelange"),
                       options = list(
                         plugins =
                          ↪  list("remove_button"),
                         create = TRUE,
                         persist = FALSE # keep created
                          ↪  choices in dropdown
                       )
                       ),
        hr(),
        helpText("Original data from STATEC")
      ),

      mainPanel(
        plotOutput("unemp_plot")
      )
    )
  )

}
```

I've added some useful things to the ui. First of all, I made it a function of an argument, `request`. This is useful for bookmarking the state of the variable. We'll add a bookmark button later. The ui is divided into two parts, a sidebar panel, and a main panel. The sidebar panel is where you will typically add dropdown menus, checkboxes, radio buttons, etc, for the users to make various selections. In the main panel, you will show the result of their selections. In the sidebar panel I add a `selectizeInput()` to create a dynamic dropdown list using the `selectize` JS library, included with `{shiny}`. The available choices are all the unique place names contained in our data, I allow users to select multiple place names, by default two communes are selected and using the `options` argument I need little "remove" buttons in the selected commune names. Finally, in the main panel I use the `plotOutput()` function to render the plot. Notice that I use the name of the plot defined in the server, "unemp_plot". Finally, to run this, add a new script, called `app.R` and add the following line in it:

```
shiny::runApp(".")
```

You can now run this script in RStudio, or from any R console, and this should open a web browser with your app.

Believe it or not, but this app contains almost every ingredient you need to know to build shiny apps. But of course, there are many, many other widgets that you can use to give your users even more ways to interact with applications.

### 7.4.2 Slightly more advanced shiny

Let's take a look at another, more complex example. Because this second example is much more complex, let's first take a look

at a video of the app in action:

The global file will be almost the same as before:

```
library(myPackage)
library(dplyr)
library(ggplot2)
library(g2r)

data("unemp")

enableBookmarking(store = "url")
```

The only difference is that I load the `{g2r}` package to create a nice interactive plot, and enable bookmarking of the state of the app using `enableBookmarking(store = "url")`. Let's move on to the ui:

```
ui <- function(request){
  fluidPage(

    titlePanel("Unemployment in Luxembourg"),

    sidebarLayout(

      sidebarPanel(
        selectizeInput("place_name_selected",
         ↪  "Select place:",

                        ↪  choices=unique(unemp$place_name),
                       multiple = TRUE,
                       selected = c("Rumelange",
                        ↪  "Dudelange"),
```

```
                    options = list(
                      plugins =
                       ↪ list("remove_button"),
                      create = TRUE,
                      persist = FALSE # keep
                       ↪ created choices in
                       ↪ dropdown
                    )
                    ),
      hr(),
      # To allow users to select the variable, we
       ↪ add a selectInput
      # (not selectizeInput, like above)
      # don't forget to use
       ↪ input$variable_selected in the
      # server function later!
      selectInput("variable_selected", "Select
       ↪ variable to plot:",
                  choices =
                   ↪ setdiff(unique(colnames(unemp)),
                   ↪ c("year", "place_name",
                   ↪ "level")),
                  multiple = FALSE,
                  selected =
                   ↪ "unemployment_rate_in_percent",
                  ),
      hr(),
      # Just for illustration purposes, these
       ↪ radioButtons will not be bound
      # to the actionButton.
      radioButtons(inputId = "legend_position",
                   label = "Choose legend
                    ↪ position",
```

```
                    choices = c("top", "bottom",
                     ↪  "left", "right"),
                    selected = "right",
                    inline = TRUE),
        hr(),
        actionButton(inputId = "render_plot",
                    label = "Click here to generate
                     ↪  plot"),
        hr(),
        helpText("Original data from STATEC"),
        hr(),
        bookmarkButton()
      ),

      mainPanel(
        # We add a tabsetPanel with two tabs. The
         ↪  first tab show
        # the plot made using ggplot the second tab
        # shows the plot using g2r
        tabsetPanel(
          tabPanel("ggplot version",
           ↪  plotOutput("unemp_plot")),
          tabPanel("g2r version",
           ↪  g2Output("unemp_plot_g2r"))
        )
      )
    )
  )

}
```

There are many new things. Everything is explained in the comments within the script itself so take a look at them. What's

important to notice, is that I now added two buttons, an action button, and a bookmark button. The action button will be used to draw the plots. This means that the user will choose the options for the plot, and then the plot will only appear once the user clicks on the button. This is quite useful in cases where computations take time to run, and you don't want the every reactive object to get recomputed as soon as the user interacts with the app. This way, only once every selection has been made can the user give the green light to the app to compute everything.

At the bottom of the ui you'll see that I've added a `tabsetPanel()` with some `tabPanel()`s. This is where the graphs "live". Let's move on to the server script:

```
server <- function(session, input, output) {

  # Because I want the plots to only render once the
   ↪ user clicks the
  # actionButton, I need to move every interactive,
   ↪ or reactive, element into
  # an eventReactive() function. eventReactive()
   ↪ waits for something to "happen"
  # in order to let the reactive variables run. If
   ↪ you don't do that, then
  # when the user interacts with app, these reactive
   ↪ variables will run
  # which we do not want.

  # Data only gets filtered once the user clicks on
   ↪ the actionButton
  filtered_data <- eventReactive(input$render_plot,
↪  {
    unemp %>%
```

```
    filter(place_name %in%
     ↪  input$place_name_selected)
})


# The variable the user selects gets passed down
 ↪  to the plot only once the user
# clicks on the actionButton.
# If you don't do this, what will happen is that
 ↪  the variable will then update the plot
# even when the user does not click on the
 ↪  actionButton
variable_selected <-
↪  eventReactive(input$render_plot, {
  input$variable_selected
})

# The plot title only gets generated once the user
 ↪  clicks on the actionButton
# If you don't do this, what will happen is that
 ↪  the title of the plot will get
# updated even when the user does not click on the
 ↪  actionButton
plot_title <- eventReactive(input$render_plot, {
  paste(variable_selected(), "for",
   ↪  paste(input$place_name_selected, collapse =
   ↪  ", "))
})

output$unemp_plot <- renderPlot({

  ggplot(data = filtered_data()) +
    theme_minimal() +
```

```
    # Because the selected variable is a string,
     ↪ we need to convert it to a symbol
    # using rlang::sym and evaluate it using !!.
     ↪ This is because the aes() function
    # expects bare variable names, and not
     ↪ strings.
    # Because this is something that developers
     ↪ have to use often in shiny apps,
    # there is a version of aes() that works with
     ↪ strings, called aes_string()
    # You can use both approaches interchangeably.
    #geom_line(aes(year,
     ↪ !!rlang::sym(variable_selected()), color =
     ↪ place_name)) +
    geom_line(aes_string("year",
     ↪ variable_selected(), color =
     ↪ "place_name")) +
    labs(title = plot_title()) +
    theme(legend.position = input$legend_position)


})

output$unemp_plot_g2r <- renderG2({

  g2(data = filtered_data()) %>%
    # g2r's asp() requires bare variable names
    fig_line(asp(year,
     ↪ !!rlang::sym(variable_selected()), color =
     ↪ place_name)) %>%
    # For some reason, the title does not show...
    subject(plot_title()) %>%
    legend_color(position = input$legend_position)
```

```
  })
}
```

What's new here, is that I now must redefine the reactive objects in such a way that they only get run once the user clicks the button. This is why every reactive object (but one, the position of the legend) is now wrapped by `eventReactive()`. `eventReactive()` waits for a trigger, in this case the clicking of the action button, to run the reactive object. `eventReactive()` takes the action button ID as an input. I've also defined the plot title as a reactive value, not only the dataset as before, because if I didn't do it, then the title of the plot would get updated as the user would choose other communes, but the contents of the plot, that depend on the data, would not get updated. To avoid the title and the plot to get desynched, I need to also wrap it around `eventReactive()`. You can see this behaviour by changing the legend position. The legend position gets updated without the user needing to click the button. This is because I have not wrapped the legend position inside `eventReactive()`.

Finally, I keep the `{ggplot2}` graph, but also remake it using `{g2r}`, to illustrate how it works inside a Shiny app.

To conclude this section, we will take a look at one last app. This app will allow users to do data aggregation on relatively large dataset, so computations will take some time. The app will illustrate how to best deal with this.

### 7.4.3 Basic optimization of Shiny apps

The app we will build now requires what is sometimes referred to *medium* size data. *Medium* size data is data that is far from

being big data, but already big enough that handling it requires some thought, especially in this scenario. What we want to do is build an app that will allow users to do some aggregations on this data. Because the size of the data is not trivial, these computations will take some time to run. So we need to think about certain strategies to avoid frustrating our users. The file we will be using can be downloaded from here. We're not going to use the exact same data set though, I have prepared a smaller version that will be more than enough for our purposes. But the strategies that we are going to implement here will also work for the original, much larger, dataset. You can get the smaller version here. Uncompressed it'll be a 2.4GB file. Not big data in any sense, but big enough to be annoying to handle without the use of some optimization strategies.

One such strategy is only letting the computations run once the user gives the green light by clicking on an action button. This is what we have seen in the previous example. The next obvious strategy is to use packages that are optimized for speed. It turns out that the functions we have seen until now, from packages like `{dplyr}` and the like, are not the fastest. Their ease of use and expressiveness come at a speed cost. So we will need to switch to something faster. We will do the same to read in the data.

This faster solution is the `{arrow}` package, which is an interface to the Arrow software developed by Apache.

The final strategy is to enable caching in the app.

So first, install the `{arrow}` package by running `install.packages("arrow"` This will compile `libarrow` from source on Linux and might take some time, so perhaps go grab a coffee.

Before building the app, let me perform a very simple benchmark. The script below reads in the data, then performs some

aggregations. This is done using standard `{tidyverse}` functions, but also using `{arrow}`:

```
start_tidy <- Sys.time()
  # {vroom} is able to read in larger files than
   ↪ {readr}
  # I could not get this file into R using
   ↪ readr::read_csv
  # my RAM would get maxed out
  air <- vroom::vroom("data/combined")

  mean_dep_delay <- air |>
    dplyr::group_by(Year, Month, DayofMonth) |>
    dplyr::summarise(mean_delay = mean(DepDelay,
 ↪  na.rm = TRUE))
end_tidy <- Sys.time()

time_tidy <- end_tidy - start_tidy


start_arrow <- Sys.time()
  air <- arrow::open_dataset("data/combined", format
 ↪  = "csv")

  mean_dep_delay <- air |>
    dplyr::group_by(Year, Month, DayofMonth) |>
    dplyr::summarise(mean_delay = mean(DepDelay,
 ↪  na.rm = TRUE))
end_arrow <- Sys.time()

end_tidy - start_tidy
end_arrow - start_arrow
```

The "tidy" approach took 17 seconds, while the arrow approach took 6 seconds. This is an impressive improvement, but put yourself in the shoes of a user who has to wait 6 seconds for each query. That would get very annoying, very quickly. So the other strategy that we will use is to provide some visual cue that computations are running, and then we will go one step further and use caching of results in the Shiny app.

But before we continue, you may be confused by the code above. After all, I told you before that functions from `{dplyr}` and the like were not the fastest, and yet, I am using them in the arrow approach as well, and they now run almost 3 times as fast. What's going on? What's happening here, is that the `air` object that we read using `arrow::open_dataset` is not a dataframe, but an `arrow` dataset. These are special, and work in a different way. But that's not what's important: what's important is that the `{dplyr}` api can be used to work with these `arrow` datasets. This means that functions from `{dplyr}` change the way they work depending on the type of the object their dealing with. If it's a good old regular data frame, some C++ code gets called to perform the computations. If it's an `arrow` dataset, `libarrow` and its black magic get called instead to perform the computations. If you're familiar with the concept of polymorphism this is it (think of `+` in Python: `1+1` returns `2`, `"a"+"b"` returns `"a+b"`. A different computation gets performed depending on the type of the function's inputs).

Let's now build a basic version of the app, only using `{arrow}` functions for speed. This is the global file:

```
library(arrow)
library(dplyr)
library(rlang)
library(DT)
```

```
air <- arrow::open_dataset("data/combined", format =
 ↪  "csv")
```

The ui will be quite simple:

```
ui <- function(request){
  fluidPage(

    titlePanel("Air On Time data"),

    sidebarLayout(

      sidebarPanel(
        selectizeInput("group_by_selected",
         ↪  "Variables to group by:",
                       choices = c("Year", "Month",
                        ↪  "DayofMonth", "Origin",
                        ↪  "Dest"),
                       multiple = TRUE,
                       selected = c("Year",
                        ↪  "Month"),
                       options = list(
                         plugins =
                          ↪  list("remove_button"),
                         create = TRUE,
                         persist = FALSE # keep
                          ↪  created choices in
                          ↪  dropdown
                       )
                       ),
        hr(),
```

```
        selectizeInput("var_to_average", "Select
         ↪  variable to average by groups:",
                        choices = c("ArrDelay",
                         ↪  "DepDelay", "Distance"),
                        multiple = FALSE,
                        selected = "DepDelay",
                        ),
        hr(),
        actionButton(inputId = "run_aggregation",
                     label = "Click here to run
                      ↪  aggregation"),
        hr(),
        bookmarkButton()
      ),

      mainPanel(
        DTOutput("result")
      )
    )
  )

}
```

And finally the server:

```
server <- function(session, input, output) {

  # Numbers get crunched only when the user clicks
   ↪  on the action button
  grouped_data <-
↪   eventReactive(input$run_aggregation, {
    air %>%
```

```
      group_by(!!!syms(input$group_by_selected)) %>%
      summarise(result =
       ↪  mean(!!sym(input$var_to_average),
                                 na.rm = TRUE)) %>%
      as.data.frame()
  })

  output$result <- renderDT({
    grouped_data()
  })

}
```

Because `group_by()` and `mean()` expect bare variable names, I convert them from strings to symbols using `rlang::syms()` and `rlang::sym()`. The difference between the two is that `rlang::syms()` is required when a list of strings gets passed down to the function (remember that the user must select several variables to group by), and this is also why `!!!` are needed (to unquote the list of symbols). Finally, the computed data must be converted back to a data frame using `as.data.frame()`. This is actually when the computations happen. `{arrow}` collects all the aggregations but does not perform anything until absolutely required. Let's see the app in action:

As you can see, in terms of User Experience (UX) this is quite poor. When the user clicks on the button nothing seems to be going on for several seconds, until the table appears. Then, when the user changes some options and clicks again on the action button, it looks like the app is crashing.

Let's add some visual cues to indicate to the user that something is happening when the button gets clicked. For this, we are going to use the `{shinycssloaders}` package:

```
install.packages("shinycssloaders")
```

and simply change the ui to this (and don't forget to load `{shinycssloaders}` in the global script!):

```
ui <- function(request){
  fluidPage(

    titlePanel("Air On Time data"),

    sidebarLayout(

      sidebarPanel(
        selectizeInput("group_by_selected",
         ↪  "Variables to group by:",
                       choices = c("Year", "Month",
                        ↪  "DayofMonth", "Origin",
                        ↪  "Dest"),
                       multiple = TRUE,
                       selected = c("Year",
                        ↪  "Month"),
                       options = list(
                         plugins =
                          ↪  list("remove_button"),
                         create = TRUE,
                         persist = FALSE # keep
                          ↪  created choices in
                          ↪  dropdown
                       )
                       ),
        hr(),
        selectizeInput("var_to_average", "Select
         ↪  variable to average by groups:",
```

```
                        choices = c("ArrDelay",
                         ↪  "DepDelay", "Distance"),
                        multiple = FALSE,
                        selected = "DepDelay",
                        ),
        hr(),
        actionButton(inputId = "run_aggregation",
                     label = "Click here to run
                      ↪  aggregation"),
        hr(),
        bookmarkButton()
      ),

      mainPanel(
        # We add a tabsetPanel with two tabs. The
         ↪  first tab show the plot made using
         ↪  ggplot
        # the second tab shows the plot using g2r
        DTOutput("result") |>
          withSpinner()
      )
    )
  )

}
```

The only difference with before is that now the `DTOutput()` right at the end gets passed down to `withSpinner()`. There are several spinners that you can choose, but let's simply use the default one. This is how the app looks now:

Now the user gets a visual cue that something is happening. This makes waiting more bearable, but even better than waiting with

a spinner is no waiting at all. For this, we are going to enable caching of results. There are several ways that you can cache results inside your app. You can enable the cache on a per-user and per-session basis, or only on a per-user basis. But I think that in our case here, the ideal caching strategy is to keep the cache persistent, and available across sessions. This means that each computation done by any user will get cached and available to any other user. In order to achieve this, you simply have to install the `{cachem}` packages add the following lines to the global script:

```
shinyOptions(cache =
 ↪  cachem::cache_disk("./app-cache",
                                                max_age =
                                                 ↪  Inf))
```

By setting the `max_age` argument to `Inf`, the cache will never get pruned. The maximum size of the cache, by default is 1GB. You can of course increase it.

Now, you must also edit the server file like so:

```
server <- function(session, input, output) {

  # Numbers get crunched only when the user clicks
   ↪  on the action button
  grouped_data <- reactive({
    air %>%
      group_by(!!!syms(input$group_by_selected)) %>%
      summarise(result =
       ↪  mean(!!sym(input$var_to_average),
                                 na.rm = TRUE)) %>%
      as.data.frame()
```

```
  }) %>%
    bindCache(input$group_by_selected,
              input$var_to_average) %>%
    bindEvent(input$run_aggregation)

  output$result <- renderDT({
    grouped_data()
  })

}
```

We've had to change `eventReactive()` to `reactive()`, just like in the app where we don't use an action button to run computations. Then, we pass the reactive object to `bindCache()`. `bindCache()` also takes the `inputs` as arguments. These are used to generate cache keys to retrieve the correct objects from cache. Finally, we pass all this to `bindEvent()`. This function takes the input referencing the action button. This is how we can now bind the computations to the button once again. Let's test our app now. You will notice that the first time we choose certain options, the computations will take time, as before. But if we perform the same computations again, then the results will be shown instantly:

As you can see, once I go back to a computation that was done in the past, the table appears instantly. At the end of the video I open a terminal and navigate to the directory of the app, and show you the cache. There are several `.Rds` objects, these are the final data frames that get computed by the app. If the user wants to rerun a previous computation, the correct data frame gets retrieved, making it look like the computation happened instantly, and with another added benefit: as discussed above, the cache is persistent between sessions, so even if the user closes

the browser and comes back later, the cache is still there, and other users will also benefit from the cache.

### 7.4.4 Deploying your shiny app

The easiest way is certainly to use shinyapps.io. I won't go into details, but you can read more about it here. You could also get a Virtual Private Server on a website like Vultr or DigitalOcean. When signing up with these services you get some free credit to test things out. If you use my Digital Ocean referral link you get 200USD to test the platform. This is more than enough to get a basic VPS with Ubuntu on it. You can then try to install everything needed to deploy Shiny apps from your VPS. You could follow this guide to deploy from DigitalOcean, which should generalize well to other services like Vultr. Doing this will teach you a lot, and I would highly recommend you do it.

### 7.4.5 References

- The server function
- Using caching in Shiny to maximize performance
- Engineering Production-Grade Shiny Apps

## 7.5 How to build data products using `{targets}`

We will now put everything together and create a `{targets}` pipeline to build a data product from start to finish. Let's go back to one of the pipelines we wrote in Chapter 7. If you're

using RStudio, start a new project and make it `renv`-enabled by checking the required checkbox. If you're using another editor, start with an empty folder and run `renv::init()`. Now create a new script with the following code (create the script `functions.R` and put the `get_data()` function in it, as described here):

```
library(targets)
library(myPackage)
library(dplyr)
library(ggplot2)
source("functions.R")

list(
    tar_target(
        unemp_data,
        get_data()
    ),

    tar_target(
        lux_data,
        clean_unemp(unemp_data,
                    place_name_of_interest =
                     ↪  "Luxembourg",
                    level_of_interest = "Country",
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        canton_data,
        clean_unemp(unemp_data,
                    level_of_interest = "Canton",
```

```
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        commune_data,
        clean_unemp(unemp_data,
                    place_name_of_interest =
                     ↪  c("Luxembourg", "Dippach",
                     ↪  "Wiltz", "Esch/Alzette",
                     ↪  "Mersch"),
                    col_of_interest =
                     ↪  active_population)
    ),

    tar_target(
        lux_plot,
        make_plot(lux_data)
    ),

    tar_target(
        canton_plot,
        make_plot(canton_data)
    ),

    tar_target(
        commune_plot,
        make_plot(commune_data)
    )

)
```

This pipeline reads in data, then filters data and produces some

plots. In another version of this pipeline we wrote the plots to disk. Now we will add them to a Quarto document, using the `tar_quarto()` function that can be found in the `{tarchetypes}` packages (so install it if this is not the case yet). `{tarchetypes}` provides functions to define further types of targets, such as `tar_quarto()` which makes it possible to render Quarto documents from a `{targets}` pipeline. But before rendering a document, we need to write this document. This is what the document could look like:

````
---
title: "Reading objects from a targets pipeline"
author: "Bruno Rodrigues"
date: today
---

This document loads three plots that were made using
 ↪  a `{targets}` pipeline.

```{r}
targets::tar_read(lux_plot)
```

```{r}
targets::tar_read(canton_plot)
```

```{r}
targets::tar_read(commune_plot)
```
````

Here is what the final pipeline would look like (notice that I've added `library(quarto)` to the list of packages getting

called):

```
library(targets)
library(tarchetypes)
library(myPackage)
library(dplyr)
library(ggplot2)
library(quarto)
source("functions.R")

list(

  tar_target(
    unemp_data,
    get_data()
  ),

  tar_target(
    lux_data,
    clean_unemp(unemp_data,
                place_name_of_interest =
                 ↪  "Luxembourg",
                level_of_interest = "Country",
                col_of_interest = active_population)
  ),

  tar_target(
    canton_data,
    clean_unemp(unemp_data,
                level_of_interest = "Canton",
                col_of_interest = active_population)
  ),
```

```
  tar_target(
    commune_data,
    clean_unemp(unemp_data,
                place_name_of_interest =
                 ↪  c("Luxembourg", "Dippach",
                 ↪  "Wiltz", "Esch/Alzette",
                 ↪  "Mersch"),
                col_of_interest = active_population)
  ),

  tar_target(
    lux_plot,
    make_plot(lux_data)
  ),

  tar_target(
    canton_plot,
    make_plot(canton_data)
  ),

  tar_target(
    commune_plot,
    make_plot(commune_data)
  ),

  tar_quarto(
    my_doc,
    path = "my_doc.qmd"
  )

)
```

Make sure that this pipeline runs using `tar_make()`. If yes, and

you're done with it, don't forget to run `renv::snapshot()` to save the projects dependencies in a lock file. Again, take a look at the lock file to make extra sure that your package is correctly being versioned. As a reminder, you should see something like this:

```
"myPackage": {
  "Package": "myPackage",
  "Version": "0.1.0",
  "Source": "GitHub",
  "RemoteType": "github",
  "RemoteHost": "api.github.com",
  "RemoteRepo": "myPackage",
  "RemoteUsername": "b-rodrigues",
  "RemoteRef":
  "e9d9129de3047c1ecce26d09dff429ec078d4dae",
  "RemoteSha":
  "e9d9129de3047c1ecce26d09dff429ec078d4dae",
  "Hash": "4740b43847e10e012bad2b8a1a533433",
  "Requirements": [
    "dplyr",
    "janitor",
    "rlang"
  ]
},
```

What's really important is that you find the "RemoteXXXX" fields. We are now ready to push this project to github.com. Don't forget to first edit the `.gitignore` file and add the `renv` folder in it. This is the folder that contains the downloaded packages, and it can get quite big. It is better to not push it. We are now done with building an almost 100% reproducible pipeline! If your product is a Shiny app, you may want to put as much calculations as possible in the `{targets}` pipelines. You

can then use `tar_load()` or `tar_read()` inside the `global.R` file.

# 8 Self-contained RAPs with Docker

What you'll have learned by the end of the chapter: build self-contained, truly reproducible analytical pipelines thanks to Docker.

## 8.1 Introduction

As discussed in section 7.7, while `{renv}` is a great tool that makes reproducibility quite simple, there are still some issues. In order to tackle these issues, we are going to learn about Docker. Docker is used to package software including all its dependencies, making it easy to run/deploy anywhere. The idea is to not only deliver the source code to our data products, but also include it inside a complete package that contains not only R and the required libraries to rebuild the data product, but also many components of the underlying operating system itself, which will usually be Ubuntu… which also means that if you're familiar with Ubuntu, you're at an advantage.

For rebuilding this data product, a single command can be used which will pull the Docker image from Docker Hub, start a container, build the data product, and stop.

If you've never heard of Docker before, this chapter should give you the very basic knowledge required to get started.

Let's start by watching this very short video that introduces Docker.

As a reminder, let's state again what problems `{renv}` does not allow to solve. Users will need to make sure themselves that they're running the pipeline with the same version of R, (as recorded in the `renv.lock` file), the same operating system, hope that `{renv}` will be able to install the packages (which can unfortunately sometimes fail) and hope that the underlying

infrastructure (MRAN and CRAN) are up and running. In the past, these issues were solved using virtual machines. The issue with virtual machines is that you cannot work with them programmatically. In a sense, Docker can be seen a lightweight virtual machine running a Linux distribution (usually Ubuntu) that you can interact with using the command line. This also means then that familiarity with Linux distributions (and in particular Ubuntu) will make using Docker easily. Thankfully, there is a very large community of Docker users who also use R. This community is organized as the Rocker Project and provides a very large collection of `Dockerfile`s to get easily started. As you saw in the video above, `Dockerfile`s are simple text files that define a Docker image, from which you can start a container.

## 8.2 Installing Docker

The first step is to install Docker. You'll find the instructions for Ubuntu here, for Windows here (read the system requirements section as well!) and for macOS here (make sure to choose the right version for the architecture of your Mac, if you have an M1 Mac use *Mac with Apple silicon*).

After installation, it might be a good idea to restart your computer, if the installation wizard does not invite you to do so. To check whether Docker was installed successfully, run the following command in a terminal (or on the desktop app on Windows):

```
docker run --rm hello-world
```

This should print the following message:

```
Hello from Docker!
This message shows that your installation appears to
be working correctly.

To generate this message, Docker took the following
steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image
 from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from
 that image which runs the
    executable that produces the output you are
    currently reading.
 4. The Docker daemon streamed that output to the
 Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an
Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a
free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

If you see this message, congratulations, you are ready to run Docker. If you see an error message about permissions, this means that something went wrong. If you're running Linux, make sure that your user is in the Docker group by running:

```
groups $USER
```

you should see your username and a list of groups that your user belongs to. If a group called `docker` is not listed, then you should add yourself to the group by following these steps.

## 8.3 The Rocker Project

The Rocker Project is instrumental for R users that want to use Docker. The project provides a large list of images that are ready to run with a single command. As an illustration, open a terminal and paste the following line:

```
docker run --rm -e PASSWORD=yourpassword -p
8787:8787 rocker/rstudio
```

Once this stops running, go to `http://localhost:8787/` and enter `rstudio` as the username and `yourpassword` as the password. You should login to a RStudio instance: this is the web interface of RStudio that allows you to work with R from a server. In this case, the *server* is the Docker container running the image. Yes, you've just pulled a Docker image containing Ubuntu with a fully working installation of RStudio web!

(If you cannot connect to `http://localhost:8787`, try with the following command:

```
docker run --rm -ti -d -e PASSWORD=yourpassword -p
8787:8787 --network="host" rocker/rstudio
```

)

Let's open a new script and run the following lines:

```
data(mtcars)

summary(mtcars)
```

You can now stop the container (by pressing `CTRL-C` in the terminal). Let's now rerun the container… (with the same command as before) you should realize that your script is gone! This is the first lesson: whatever you do inside a container will disappear once the container is stopped. This also means that if you install the R packages that you need while the container is running, you will need to reinstall them every time. Thankfully, the Rocker Project provides a list of images with many packages already available. For example to run R with the `{tidyverse}` collection of packages already pre-installed, run the following command:

```
docker run --rm -ti -e PASSWORD=yourpassword -p
8787:8787 rocker/tidyverse
```

If you compare it to the previous command, you see that we have replaced *rstudio* with *tidyverse*. This is because `rocker/tidyverse` references an image, hosted on Docker Hub, that provides the latest version of R, RStudio server and the packages from the `{tidyverse}`. You can find the image hosted on Docker Hub here. There are many different images, and we will be using the *versioned* images made specifically for reproducibility. For now, however, let's stick with the `tidyverse` image, and let's learn a bit more about some specifics.

## 8.4 Docker essentials

You already know about running containers using `docker run`. With the commands we ran before, your terminal will need to stay open, or else, the container will stop. Starting now, we will run Docker commands in the background. For this, we will use the `-d` flag (`d` as in *detach*), so let's stop the container one last time with `CTRL-C` and rerun it using:

```
docker run --rm -d -e PASSWORD=yourpassword -p
8787:8787 rocker/tidyverse
```

(notice `-d` just after `run`). You can run several containers in the background simultaneously. You can list running containers with `docker ps`:

```
docker ps
CONTAINER ID   IMAGE              COMMAND   CREATED
STATUS         PORTS
NAMES
c956fbeeebcb   rocker/tidyverse   "/init"   3
minutes ago   Up 3 minutes   0.0.0.0:8787->8787/tcp,
:::8787->8787/tcp   elastic_morse
```

The running container has the ID `c956fbeeebcb`. Also, the very last column, shows the name of the running container. This is a label that you can change. For now, take note of ID, because we are going to stop the container:

```
docker stop c956fbeeebcb
```

After Docker is done stopping the running container, you can check the running containers using `docker ps` again, but this time no containers should get listed. Let's also discuss the other flags `--rm`, `-e` and `-p`. `--rm` removes the container once it's

stopped. Without this flag, we can restart the container and all the data and preferences we saved will be restored. However, this is dangerous because if the container gets removed, then everything will get lost, forever. We are going to learn how to deal with that later. `-e` allows you to provide environment variables to the container, so in this case the `$PASSWORD` variable. `-p` is for setting the port at which your app is going to get served. Let's now rerun the container, but by giving it a name:

```
docker run -d --name my_r --rm -e
PASSWORD=yourpassword -p 8787:8787 rocker/tidyverse
```

Notice the `--name` flag followed by the name we want to use, `my_r`. We can now interact with this container using its name instead of its ID. For example, let's open an interactive bash session. Run the following command:

```
docker exec -ti my_r bash
```

You are now inside a terminal session, inside the running container! This can be useful for debugging purposes. It's also possible to start R in the terminal, simply replace `bash` by `R` in the command above.

Finally, let's solve the issue of our scripts disappearing. For this, create a folder somewhere on your computer (host). Then, rerun the container, but this time with this command:

```
docker run -d --name my_r --rm -e
PASSWORD=yourpassword -p 8787:8787 -v
/path/to/your/local/folder:/home/rstudio/scripts:rw
rocker/tidyverse
```

where `/path/to/your/local/folder` should be replaced to the folder you created. You should now be able to save the scripts inside the `scripts/` folder from RStudio and they will appear in the folder you created.

## 8.5 Making our own images and run some code

We now know how to save files to our computer from Docker. But as the container gets stopped (and removed because of –rm) if we install R packages, we would need to reinstall them each time. The solution is thus to create our own Docker image, and as you will see, it is quite simple to get started. Create a folder somewhere on your computer, and add a text file called `Dockerfile` (without any extension). In this file add, the following lines:

```
FROM rocker/tidyverse

RUN R -e
"devtools::install_github('b-rodrigues/myPackage',
ref = 'e9d9129de3047c1ecce26d09dff429ec078d4dae')"
```

Every `Dockerfile` starts with a `FROM` statement. This means that this `Dockerfile` will use the `rocker/tidyverse` image as a starting point. Then, we will simply download the package we've been developing together using a `RUN` statement, which you've guessed it, runs a command. Then we need to build the image. For this, run the following line:

```
docker build -t my_package .
```

This will build the image right in this folder and call it `my_package`.

```
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM rocker/tidyverse
 ---> a838ee142831
```

```
Step 2/2 : RUN R -e
"devtools::install_github('b-rodrigues/myPackage',
ref = 'e9d9129de3047c1ecce26d09dff429ec078d4dae')"
 ---> Using cache
 ---> 17d5d3179293
Successfully built 17d5d3179293
Successfully tagged my_package:latest
```

By running `docker images` you should see all the images that are on your PC (with running containers or not):

```
docker images
```

```
REPOSITORY          TAG       IMAGE ID       CREATED
SIZE
my_package          latest    17d5d3179293   13
minutes ago   2.16GB
rocker/tidyverse    latest    a838ee142831   11 days
ago      2.15GB
rocker/rstudio      latest    d110bab4d154   11 days
ago      1.79GB
hello-world         latest    feb5d9fea6a5   13
months ago    13.3kB
```

You should see that each image takes up a lot of space: but this is misleading. Each image that builds upon another does not duplicate the same layers. So this means that our image, `my_package`, only add the `{myPackage}` package to the `rocker/tidyverse` image, which in turn only adds the `{tidyverse}` packages to `rocker/rstudio`. This means unlike what is shown here, all the images to not need 6GB of space, but only 2.16GB in total. So let's now make sure that every other container is stopped (because we will run our container on the same port) and let's run our container using this command:

```
docker run --rm -d --name my_package_container -e
PASSWORD=yourpassword -p 8787:8787 my_package
```

You should now see `{myPackage}` available in the list of packages in the RStudio pane. Let's now go one step further. Let's create one plot from within Docker, and make it available to the person running it. Let's stop again our container:

```
docker stop my_package_container
```

Now, in the same folder where your `Dockerfile` resides, add the following `R` script (save this inside `my_graph.R`):

```
library(ggplot2)
library(myPackage)

data("unemp")

canton_data <- clean_unemp(unemp,
                           level_of_interest =
                           "Canton",
                           col_of_interest =
                           active_population)

my_plot <- ggplot(canton_data) +
  geom_col(
    aes(
      y = active_population,
      x = year,
      fill = place_name
    )
  ) +
  theme(legend.position = "bottom",
        legend.title = element_blank())
```

```
ggsave("/home/rstudio/scripts/my_plot.pdf", my_plot)
```

This script loads the data, and saves it to the scripts folder (as you see, this is a path inside of the Docker image). We will also need to update the `Dockerfile`. Edit it to look like this:

```
FROM rocker/tidyverse

RUN R -e
"devtools::install_github('b-rodrigues/myPackage',
ref = 'e9d9129de3047c1ecce26d09dff429ec078d4dae')"

RUN mkdir /home/rstudio/graphs

COPY my_graph.R /home/rstudio/graphs/my_graph.R

CMD R -e "source('/home/rstudio/graphs/my_graph.R')"
```

We added three commands at the end; one to create a folder (using `mkdir`) another to copy our script to this folder (so for this, remember that you should put the R script that creates the plot next to the `Dockerfile`) and finally an R command to source (or run) the script we've just copied. Save the `Dockerfile` and build it again:

```
docker build -t my_package .
```

Let's now run our container with the following command (notice that we do not use `-p` nor the `-e` flags anymore, because we're not interested in running RStudio in the browser anymore):

```
docker run --rm --name my_package_container -v
/path/to/your/local/folder:/home/rstudio/scripts:rw
my_package
```

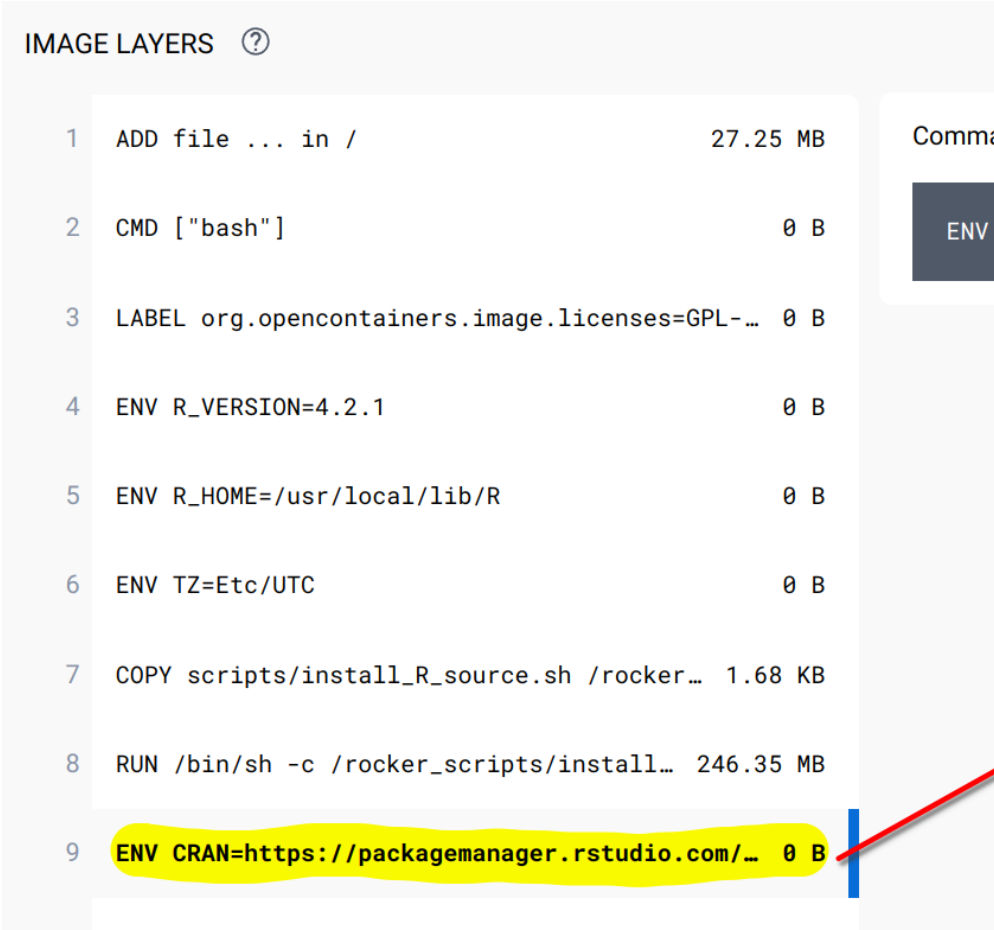
After some seconds, you should see a PDF in the folder that you set up. This is the output of the script! You probably see now where this is going: we are going to define a `{targets}` pipeline that will be run each time the container is run. But one problem remains.

## 8.6 Reproducibility with Docker

Our `Dockerfile`, as it is now, is not suited for reproducibility. This is because each time the image gets built, the latest version of R and package will get pulled from the Internet. We need to use a `Dockerfile` that builds exactly the same image, regardless of when it gets built. Thankfully, the Rocker Project is here to help. A series of `Dockerfiles` are available that:

- always use the exact same version of R;
- a frozen CRAN repository will be used to pull the packages;
- a long term support of Ubuntu is used as a base image.

You can read about it more here. As I'm writing this, the latest stable image uses R v4.2.1 on Ubuntu 20.04. The latest image, based on Ubuntu 22.04 and which uses the latest version of R (v4.2.2) still uses the default CRAN repository, not a frozen one. So for our purposes, we will be using the `rocker/r-ver:4.2.1` image, which you can find here. What's quite important, is to check that the CRAN mirror is frozen. Look for the line in the `Dockerfile` that starts with `ENV CRAN...` and you should see this:




As you can see in the screenshot, we see that the CRAN mirror is set to the 28 of October 2022. Let's now edit our `Dockerfile` like so:

```
FROM rocker/r-ver:4.2.1

RUN R -e "install.packages(c('devtools',
'ggplot2'))"
```

```
RUN R -e
"devtools::install_github('b-rodrigues/myPackage',
ref = 'e9d9129de3047c1ecce26d09dff429ec078d4dae')"

RUN mkdir /home/graphs

COPY my_graph.R /home/graphs/my_graph.R

CMD R -e "source('/home/graphs/my_graph.R')"
```

As you can see, we've changed to first line to `rocker/r-ver:4.2.1`, added a line to install the required packages, and we've removed `rstudio` from the paths in the other commands. This is because `r-ver` does not launch an RStudio session in browser, so there's no `rstudio` user. Before building the image, you should also update the script that creates the plot. This is because in the last line of our script, we save the plot to `"/home/rstudio/scripts/my_plot.pdf"`, but remember, there's no `rstudio` user. So remove this from the `ggsave()` function. Also, add another line to the script, right at the bottom:

```
writeLines(capture.output(sessionInfo()),
"/home/scripts/sessionInfo.txt")
```

so the script finally looks like this:

```
library(ggplot2)
library(myPackage)

data("unemp")

canton_data <- clean_unemp(unemp,
                           level_of_interest =
                           "Canton",
```

```
                              col_of_interest =
                              active_population)

my_plot <- ggplot(canton_data) +
  geom_col(
    aes(
      y = active_population,
      x = year,
      fill = place_name
    )
  ) +
  theme(legend.position = "bottom",
        legend.title = element_blank())

ggsave("/home/scripts/my_plot.pdf", my_plot)

writeLines(capture.output(sessionInfo()),
"/home/scripts/sessionInfo.txt")
```

Now, build this image using:

```
docker build -t my_package .
```

and this will run R and install the packages. This should take some time, because `r-ver` images do not come with any packages preinstalled. Once this is done, we can run a container from this image using:

```
docker run --rm --name my_package_container -v
/path/to/your/local/folder:/home/scripts:rw
my_package
```

You should see two files now: the plot, and a `sessionInfo.txt` file. Open this file, and you should see the following:

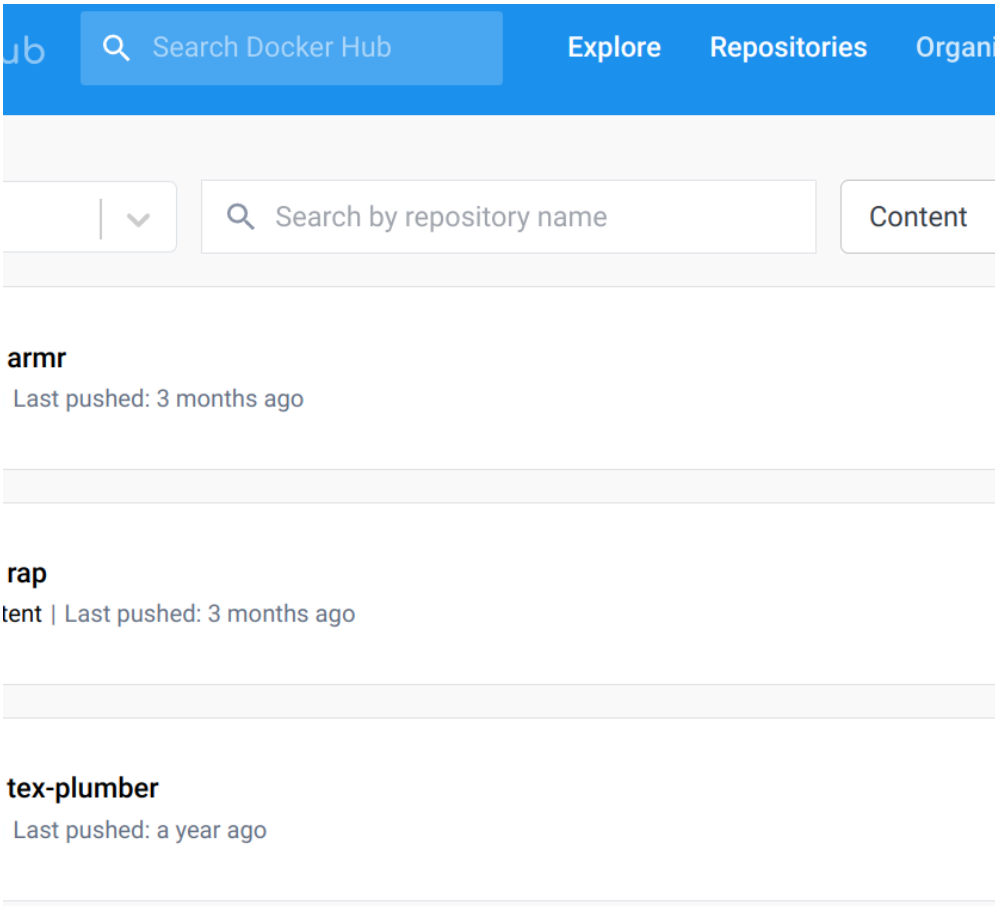
```
R version 4.2.1 (2022-06-23)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.5 LTS
```

This confirms that the code ran indeed on R 4.2.1 under Ubuntu 20.04.5 LTS. You should also see that the `{ggplot2}` version used is `{ggplot2}` version 3.3.6, which is older than the version you could get now (as of November 2022), which is 3.4.0.

We now have all the ingredients and basic knowledge to build a fully reproducible pipeline.

## 8.7 Building a truly reproducible pipeline

Ok so we are almost there; we now know how to run code in an environment that is completely stable, so our results are 100% reproducible. However, there are still some things that we can learn in order to make our pipeline even better. First of all, we can make it run faster by creating an image that has already all the packages that we need installed. This way, whenever we will need to build it, no packages will need to be installed. We will also put this image on Docker Hub, so in the future, people that want to run our pipeline can do so by pulling the pre-built image from Docker, instead of having to rebuild it using the `Dockerfile`. In order to get an image on Docker Hub, you first need to create an account there. Once logged in, you can click on `Create repository`:




You can then give a name to this repository. Let's now create an image that we will push. Let's restart from the `Dockerfile` that we used, and add a bunch of stuff:

```
FROM rocker/r-ver:4.2.1

RUN apt-get update && apt-get install -y \
    libglpk-dev \
    libxml2-dev \
    libcairo2-dev \
```

```
    libgit2-dev \
    default-libmysqlclient-dev \
    libpq-dev \
    libsasl2-dev \
    libsqlite3-dev \
    libssh2-1-dev \
    libxtst6 \
    libcurl4-openssl-dev \
    libharfbuzz-dev \
    libfribidi-dev \
    libfreetype6-dev \
    libpng-dev \
    libtiff5-dev \
    libjpeg-dev \
    unixodbc-dev \
    wget

RUN wget
https://github.com/quarto-dev/quarto-cli/releases/download/v1.2
-O /home/quarto.deb
RUN apt-get install --yes /home/quarto.deb
RUN rm /home/quarto.deb

RUN R -e "install.packages(c('devtools',
'tidyverse', 'janitor', \
   'shiny', 'targets', 'tarchetypes', \
   'quarto', 'shiny', 'testthat', \
   'usethis', 'rio'))"

RUN R -e
"devtools::install_github('b-rodrigues/myPackage',
ref = 'e9d9129de3047c1ecce26d09dff429ec078d4dae')"
```

```
CMD ["R"]
```

This `Dockerfile` starts off with `r-ver:4.2.1` and adds the dependencies that we will need for our pipelines. Then, I install development libraries, these are required to run the R packages (maybe not all of them though). I found the list here; this is a script that gets used by some of the `Dockerfile`s provided by the Rocker Project. I only copied the parts I needed. Then I download the Quarto installer for Ubuntu, and install it. Finally I install the packages for R, as well as the package we've developed together. This `Dockerfile` should not look too intimidating IF you're familiar with Ubuntu. If not… well this is why I said in the intro that familiarity with Ubuntu would be helpful. Now you probably see why Rocker is so useful; if you start from an `rstudio` image all of these dependencies come already installed. But because we're using an image made specifically for reproducibility, *only* the frozen repos were set up, which is why I had to add all of this manually. But no worries, you can now use this `Dockerfile` as a reference.

Anyways, we can now build this image using:

```
docker build -t r421_rap .
```

And now we need to wait for the process to be done. Once it's finished, we can run it using:

```
docker run --rm -ti --name r421_rap_container
r421_rap
```

(notice the `-ti` argument here; this is needed because we want to have an interactive session with R opened, if you omit this flag, R will get launched, but then immediately close). We can test it by loading some packages and see that everything is alright.

Let's now get this image on Dockerhub; this way, we can pull it instead of having to build it in the future. First logging to Docker Hub from the terminal:

```
docker login
```

You should then enter your username and password. We are now ready to push, so check the image id using `docker images`:

```
docker images
REPOSITORY          TAG       IMAGE ID       CREATED
SIZE
r421_rap            latest    864350bf1143   5
minutes ago    1.98GB
```

Tag the image, in this case the tag I've used is `version1`:

```
docker tag 864350bf1143
your_username_on_docker_hub/r421_rap:version1
```

And now I can push it, so that everyone can use it:

```
docker push brodriguesco/r421_rap:version1
```

We can now use this as a base for our pipelines! Let's now create a new `Dockerfile` that will use this image as a base and run the plot from before:

```
FROM brodriguesco/r421_rap:version1

RUN mkdir /home/graphs

COPY my_graph.R /home/graphs/my_graph.R

CMD R -e "source('/home/graphs/my_graph.R')"
```

save this `Dockerfile` in a new folder, and don't forget to add the `my_graph.R` script with it. You can now build the image using:

```
docker build -t my_pipeline .
```

You should see this:

```
Sending build context to Docker daemon  3.584kB
Step 1/4 : FROM brodriguesco/r421_rap:version1
version1: Pulling from brodriguesco/r421_rap
eaead16dc43b: Pull complete
```

As you can see, now Docker is pulling the image I've uploaded… and what's great is that this image already contains the correct versions of the required packages and R.

Before continuing now, let's make something very clear: the image that I made available on Docker Hub is prebuilt, which means that anyone building a project on top of it, will not have to rebuild it. This means also, that in theory, there would be no need to create an image built on top of an image like `rocker/r-ver:4.2.1` with frozen repositories. Because most users of the `brodriguesco/r421_rap` image would have no need to rebuild it. However, in cases where users would need to rebuild it, it is best practice to use such a stable image as `rocker/r-ver:4.2.1`. This makes sure that if the image gets rebuilt in the future, then it still pulls the exact same R and packages versions as today.

Ok, so now to run the pipeline this line will do the job:

```
docker run --rm --name my_pipeline_container -v
/home/cbrunos/docker_folder:/home/scripts:rw
my_pipeline
```

So basically, all you need for your project to be reproducible is a Github repo, where you make the `Dockerfile` available, as well as the required scripts, and give some basic instructions in a `Readme`.

To conclude this section, take a look at this repository. This repository defines in three files a pipeline that uses Docker for reproducibility:

- A `Dockerfile`;
- `_targets.R` defining a `{targets}` pipeline;
- `functions.R` which includes needed functions for the pipeline.

Try to run the pipeline, and study the different files. You should recognize the commands used in the `Dockerfile`.

Now it's your turn to build reproducible pipelines!

## 8.8 One last thing

It should be noted that you can also use `{renv}` in combination with Docker. What you could do is copy an `{renv}` lockfile into Docker, and restore the packages with `{renv}`. You could then push this image, which would contain every package, to Docker Hub, and then provide this image to your future users instead. This way, you wouldn't need to use a base image with frozen CRAN repos as we did. That's up to you.

If you want an example of this, look here.

## 8.9 Further reading

- https://www.statworx.com/content-hub/blog/wie-du-ein-r-skript-in-docker-ausfuehrst/ (in German, English translation: https://www.r-bloggers.com/2019/02/running-your-r-script-in-docker/)
- https://colinfay.me/docker-r-reproducibility/
- https://jsta.github.io/r-docker-tutorial/
- http://haines-lab.com/post/2022-01-23-automating-computational-reproducibility-with-r-using-renv-docker-and-github-actions/

# 9 Intro to CI/CD with Github Actions

What you'll have learned by the end of the chapter: very basic knowledge of Github Actions, but enough to run your RAP in the cloud.

## 9.1 Introduction

We are almost at the end; actually, we could have stopped at the end of the previous chapter. We have reached our goal; we are able to run pipeline in a 100% reproducible way. However, this requires some manual steps. And maybe that's not a problem; if your image is done, and users only need to pull it and run the container, that's not really a big problem. But you should keep in mind that manual steps don't scale. Let's imagine another context; let's suppose that you are part of a company and that you are part of a team that needs to quickly ship products to clients. Maybe several people contribute to the product using an internal version control solution (like a Gitlab instance that is deployed on the premises of your company). Maybe you even need to work on several products in the same day; you (and your teammates) should only be focusing writing code (and `Dockerfiles`)… your time and resources cannot get clogged by building images (which depending on what you're working on, can take quite some time). So ideally, we would want to automate this step. That is what we are going to learn in this chapter.

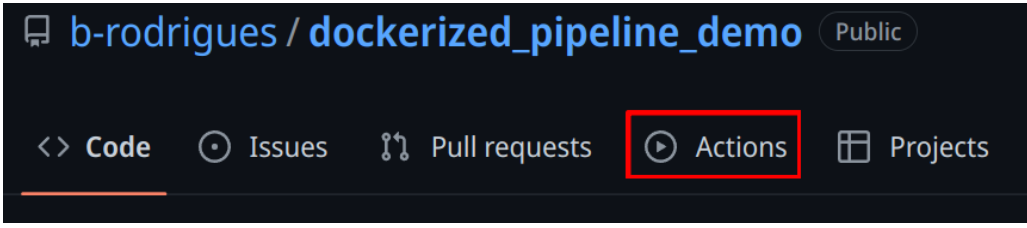
This chapter will introduce you to the basic ideas of CI/CD (Continuous Integration and Continuous Deployment/Delivery) and DevOps with Github Actions. Because we're using Git to trigger all the events and automate the whole pipeline, this can also be referred to as GitOps. What's Dev(Git)Ops? I think

that the Atlassian page on DevOps makes a good job of explaining it. The bottom line is that DevOps makes it easy for developers to focus on coding, and makes it easy for them to ship data products. The core IT team provides the required infrastructure and tools to make this possible. GitOps is a variant of DevOps where the definition of the infrastructure is versioned, and can be changed by editing simple text files. Through events, such as pushing to the repository, new images can be built, or containers executed. Data products can then also be redeployed automatically. All the steps we've been doing manually, with one simple push! It's also possible, in the context of package development, to execute unit tests when code gets pushed to repo, or get documentation and vignettes compiled. This also means that you could be developing on a very thin client with only a text editor and git installed. Pushing to Github would then execute everything needed to have a package ready for sharing.

So our goal here is, in short, to do exactly the same as what we have been doing on our computer (so build an image, run a container, and get back 3 plots), but on Github.

## 9.2 Getting your repo ready for Github Actions

You should see an "Actions" tab on top of any Github repo:



This will open a new view where you can select a lot of available, ready to use actions. Shop around for a bit, and choose the right one (depending on what you want to do). You should know that there is a very nice repository with many actions for R. Once you're done choosing an action, a new view in which you can edit a file will open. This file will have the name of the chosen action, and have the `.yml` extension. This file will be automatically added to your repository, in the following path: `.github/workflows`.

Let's take a look at such a workflow file:

```
name: Hello world
on: [push]
jobs:
  say-hello:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello from Github Actions!"
      - run: echo "This command is running from an
      Ubuntu VM each time you push."
```

Let's study this workflow definition line by line:

```
name: Hello world
```

Simply gives a name to the workflow.

```
on: [push]
```

When should this workflow be triggered? Here, whenever something gets pushed.

```
jobs:
```

What is the actual things that should happen? This defines a list of actions.

```
  say-hello:
```

This defines the `say-hello` job.

```
    runs-on: ubuntu-latest
```

This job should run on an Ubuntu VM. You can also run jobs on Windows or macOS VMs, but this uses more compute minutes than a Linux VM (you have 2000 compute minutes for free per month).

```
    steps:
```

What are the different steps of the job?

```
      - run: echo "Hello from Github Actions!"
```

First, run the command `echo "Hello from Github Actions!"`. This commands runs inside the VM. Then, run this next command:

```
      - run: echo "This command is running from an
      Ubuntu VM each time you push."
```

Let's push, and see what happens on github.com:

If we take a look at the commit we just pushed, we see this yellow dot next to the commit name. This means that an action is running. We can then take a look at the output of the job, and see that our commands, defined with the `run` statements in the workflow file, succeeded and echoed what we asked them.

So, the next step is running our Docker image and getting back our plots. This next example can be found in this repository.

This is what our workflow file looks like:

```
name: Reproducible pipeline

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Build the Docker image
      run: docker build -t my-image-name .
    - name: Docker Run Action
      run: docker run --rm --name
      my_pipeline_container -v
      /github/workspace/fig/:/home/graphs/:rw
      my-image-name
    - uses: actions/upload-artifact@v3
      with:
        name: my-figures
        path: /github/workspace/fig/
```

For now, let's focus on the `run` statements, because these should be familiar:

```
run: docker build -t my-image-name .
```

and:

```
run: docker run --rm --name my_pipeline_container -v
/github/workspace/fig/:/home/graphs/:rw
my-image-name
```

The only new thing here, is that the path has been changed to `/github/workspace/`. This is the home directory of your repository, so to speak. Now there's the `uses` keyword that's new:

```
uses: actions/checkout@v3
```

This action checkouts your repository inside the VM, so the files in the repo are available inside the VM. Then, there's this action here:

```
- uses: actions/upload-artifact@v3
  with:
    name: my-figures
    path: /github/workspace/fig/
```

This action takes what's inside `/github/workspace/fig/` (which will be the output of our pipeline) and makes the contents available as so-called "artifacts". Artifacts are the outputs of your workflow. In our case, as stated, the output of the pipeline. So let's run this by pushing a change, and let's take a look at these artifacts!

As you can see from the video above, a zip file is now available and can be downloaded. This zip contains our plots! It is thus possible to rerun our workflow in the cloud. This has the advantage that we can now focus on simply changing the code, and not have to bother with boring manual steps. For example, let's change this target in the `_targets.R` file:

```
tar_target(
    commune_data,
    clean_unemp(unemp_data,
                place_name_of_interest =
                c("Luxembourg", "Dippach",
                                        "Wiltz",
                                        "Esch/Alzette",
                                        "Mersch",
                                        "Dudelange"),
                col_of_interest = active_population)
)
```

I've added "Dudelange" to the list of communes to plot. Let me push this change to the repo now, and let's take a look at the artifacts. The video below summarises the process:

As you can see in the video, the `_targets.R` script was changed, and the changes pushed to Github. This triggered the action we've defined before. The plots (artifacts) get refreshed, and we can download them. We see then that Dudelange was added in the `communes.png` plot!

It is also possible to "deploy" the plots directly to another branch, and do much, much more. I just wanted to give you a little taste of Github Actions (and more generally GitOps). The possibilities are virtually limitless, and I still can't get over the fact that Github Actions is free (well, up to 2000 compute minutes and 500MB storage per month).

## 9.3 Building a Docker image and pushing it to a registry

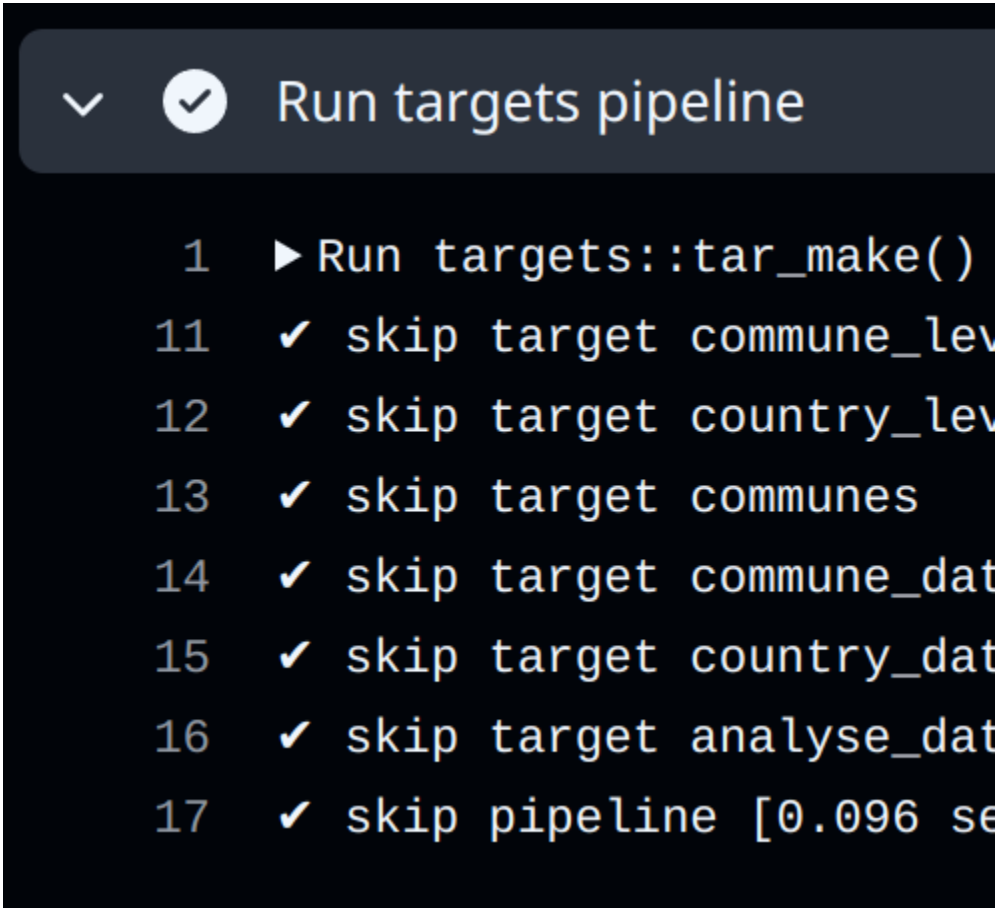
It is also possible to build a Docker image and have it made available on an image registry. You can see how this works on this repository. This images can then be used as a base for other RAPs, as in this repository. Why do this? Well because of "separation of concerns". You could have a repository which builds in image containing your development environment: this could be an image with a specific version of R and R packages. And then have as many repositories as projects that run RAPs using that development environment image as a basis. Simply add the project-specific packages that you need for each project.

## 9.4 Running a pipeline straight from Github Actions

Using Docker on Github Actions has the advantage that you can use the same image to develop locally on your computer, and then also on CI. However, you could also run the pipeline straight from a Github Actions runner, but it'll take some effort to set up the environment on CI. Take a look at the example from this repository.

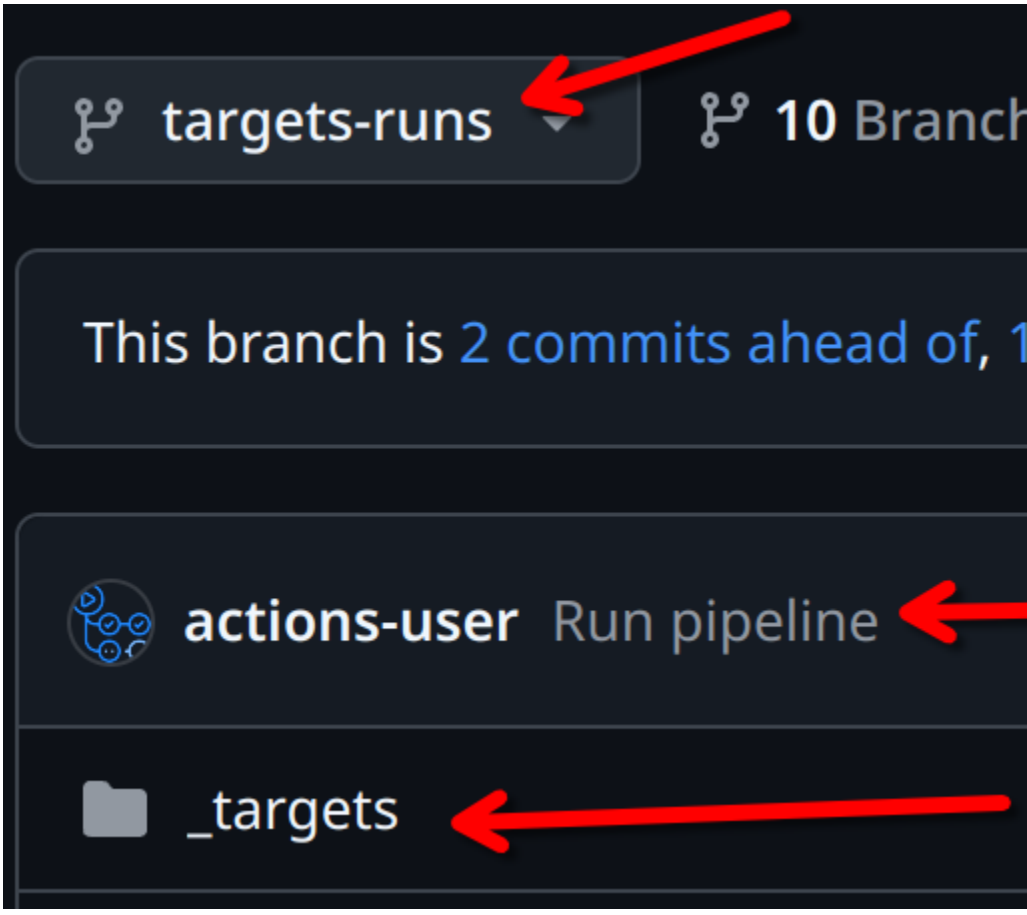
The yaml file used in this action (which you can find here) was generated by running `targets::tar_github_actions()` and was then modified further, mostly to add the required development libraries to compile the needed R packages (under the `Install Linux system dependencies` step).

This action takes advantage of the included Github Actions cache to backup the targets from the pipeline, so they can also get skipped with subsequent runs:




This can also be achieved with Docker by mounting volumes, but requires more manual setup.

Another difference with Docker is that the outputs are not saved as an artifact, but instead get pushed to the `targets-runs` branch:




## 9.5 Running unit tests on Github Actions

Setting up your project as a package (or at least, the parts of your project that can be reused for others) as a package also has the advantage that it becomes very easy to run unit tests on CI. See for example the `{myPackage}` package that we developed

together, in particular this file. This action runs on each push and pull request on Windows, Ubuntu and macOS:

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  rcmdcheck:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest,
        macos-latest]
```

Several steps are executed, all using pre-defined actions from the `r-lib` project:

```
    steps:
    - uses: actions/checkout@v4
    - uses: r-lib/actions/setup-r@v2
    - uses: r-lib/actions/setup-r-dependencies@v2
      with:
        extra-packages: any::rcmdcheck
        needs: check
    - uses: r-lib/actions/check-r-package@v2
```

An action such as `r-lib/actions/setup-r@v2` will install R on any of the supported operating systems without requiring any configuration from you. If you didn't use such an action, you would need to define three separate actions: one that would be executed on Windows, on Ubuntu and on macOS.

Each of these operating-specific actions would install R in their operating-specific way.

Check out the workflow results to see how the package could be improved here.

## 9.6 Further reading

- http://haines-lab.com/post/2022-01-23-automating-computational-reproducibility-with-r-using-renv-docker-and-github-actions/
- https://orchid00.github.io/actions_sandbox/
- https://www.petefreitag.com/item/903.cfm
- https://dev.to/mihinduranasinghe/using-docker-containers-in-jobs-github-actions-3eof

# 10 Reproducibility with Nix

## 10.1 The Nix package manager

Nix is a package manager that can be used to build completely reproducible development environments. These environments can be used for interactive data analysis or running pipelines in a CI/CD environment.

If you're familiar with the Ubuntu Linux distribution, you likely have used `apt-get` to install software. On macOS, you may have used `homebrew` for similar purposes. Nix functions in a similar way, but has many advantages over classic package managers. The main advantage of Nix, at least for our purposes, is that its repository of software is huge. As of writing, it contains more than 80.000 packages, and the entirety of CRAN and Bioconductor is available through Nix's repositories. This means that using Nix, it is possible to install not only R, but also all the packages required for your project. The obvious question is why use Nix instead of simply installing R and R packages as usual. The answer is that Nix makes sure to install every dependency of any package, up to required system libraries. For example, the `{xlsx}` package requires the Java programming language to be installed on your computer to successfully install. This can be difficult to achieve, and `{xlsx}` bullied many R developers throughout the years (especially those using a Linux distribution, `sudo R CMD javareconf` still plagues my nightmares). But with Nix, it suffices to declare that we want the `{xlsx}` package for our project, and Nix figures out automatically that Java is required and installs and configures it. It all just happens without any required intervention from the user. The second advantage of Nix is that it is possible to pin a certain revision of the Nix packages' repository (called nixpkgs) for our project. Pinning a revision ensures that every package that Nix installs will always be at exactly the same versions, regardless of when in the future the packages get installed.

With Nix, it is essentially possible to replace {renv} and Docker combined, or if you're using mainly Python, you can replace `conda` or `requirements.txt` files. If you need other tools or languages like Python or Julia, this can also be done easily. Nix is available for Linux, macOS and Windows (via WSL2). Important remark: since using Nix on Windows must go through WSL, when we refer to "Linux" in the context of Nix, this includes Windows by default as well. It is also possible to build multi-language environments, containing R and Python, a LaTeX distribution and packages and so on.

## 10.2 The Nix programming language

Nix is not just useful because it is possible to install many packages and even install older packages, but also because it comes with a complete functional programming language. This programming language is used to write *expressions*, and these expressions in turn are used to build software. Essentially, when you install a package using Nix, an expression gets downloaded from the Nix package repository (more on that in the next section), and it gets evaluated by the Nix package manager. This expression contains a so-called *derivation*. A derivation defines a build: some inputs, some commands, and then an output. Most of the time, a derivation downloads source code, builds the software from the source and then outputs a compiled binary. Derivations are extremely flexible, and you could write a derivation to build a complete environment and then build a complete reproducible pipeline. The output could be any of the discussed data products.

Learning the Nix programming language is a good idea if you want to contribute to the Nix package repository, but you might

not have to learn it in-depth if you simply wish to use it to build reproducible environments, as we will learn now. If you wish to learn about the programming language, I highly recommend a tour of Nix[1].

## 10.3 The Nix package repository

So, there's the Nix package manager, the Nix programming language and the Nix package repository (henceforth nixpkgs). To look for packages click here[2]. The source code of all the packages (so the whole set of Nix expressions) can be found on this Github repository[3]. For example, here[4] is the Nix expression that contains the derivation to build `quarto`. As you can see, the derivation uses the the pre-built Quarto binaries instead of building it from source. Adding packages to nixpkgs (or updating them) can be done by opening pull requests. For example, here[5] is a pull request to make Quarto available to all platforms (before this PR Quarto was only available for Linux). PRs get reviewed and approuved by maintainers that also have the right to merge the PR into master. Once merged, the new or updated package is available for download. Because nixpkgs is a "just" Github repository, it is possible to use a specific commit hash to install the packages as they were at a specific point in time. For example, if you use this commit, 7c9cc5a6e, you'll get the packages as of the 19th of October 2023, but if you used this one instead: 976fa3369, you'll get packages from the 19th of August

[1] https://nixcloud.io/tour/?id=introduction/nix
[2] https://search.nixos.org/packages
[3] https://github.com/NixOS/nixpkgs
[4] https://github.com/NixOS/nixpkgs/blob/master/pkgs/development/libraries/quar
[5] https://github.com/NixOS/nixpkgs/pull/259443

2023. Using specific hashes is called "pinning" and you can read more about it here. We will make extensive use of pinning.

## 10.4 The NixOS operating system, Docker and Github Actions

NixOS is a Linux distribution that uses the Nix package manager as its package manager. I won't go into detail here, but you should know it exists. What's perhaps more interesting for our purposes is to use Nix within Docker. Because Nix can be installed as any other tool, you could very well build a Docker image that starts by installing Nix, and then uses Nix to install, in a reproducible manner, all the tools you need for your project.

There are also a series of Github Actions that you can use to install Nix on runners and build development environments. We will also look that.

## 10.5 A first Nix expression

The following expression is the one that defines the development environment to build this book:

```
let
 pkgs = import (fetchTarball
 "https://github.com/NixOS/nixpkgs/archive/976fa3369d722e76f37c
 {};
 rpkgs = builtins.attrValues {
```

```
  inherit (pkgs.rPackages) quarto Ecdat devtools
  janitor plm pwt9 rio targets tarchetypes testthat
  tidyverse usethis formatR;
};
  tex = (pkgs.texlive.combine {
  inherit (pkgs.texlive) scheme-small amsmath framed
  fvextra environ fontawesome5 orcidlink pdfcol
  tcolorbox tikzfill;
});
 system_packages = builtins.attrValues {
  inherit (pkgs) R glibcLocalesUtf8 quarto;
};
  in
  pkgs.mkShell {
    LOCALE_ARCHIVE = if pkgs.system ==
    "x86_64-linux" then
    "${pkgs.glibcLocalesUtf8}/lib/locale/locale-archive"
    else "";
    LANG = "en_US.UTF-8";
    LC_ALL = "en_US.UTF-8";
    LC_TIME = "en_US.UTF-8";
    LC_MONETARY = "en_US.UTF-8";
    LC_PAPER = "en_US.UTF-8";
    LC_MEASUREMENT = "en_US.UTF-8";

    buildInputs = [  rpkgs tex system_packages  ];

  }
```

The first line imports a specfic hash of nixpkgs (pinning):

```
pkgs = import (fetchTarball
"https://github.com/NixOS/nixpkgs/archive/976fa3369d722e76f37
{};
```

Then, I define the set of R packages that we require:

```
rpkgs = builtins.attrValues {
  inherit (pkgs.rPackages) quarto Ecdat devtools
  janitor plm pwt9 rio targets tarchetypes testthat
  tidyverse usethis formatR;
};
```

I then do something similar for LaTeX packages:

```
tex = (pkgs.texlive.combine {
  inherit (pkgs.texlive) scheme-small amsmath framed
  fvextra environ fontawesome5 orcidlink pdfcol
  tcolorbox tikzfill;
});
```

Finally, I define the set of "system" packages, so the R language itself, and Quarto (and glibcLocalesUtf8 to set the locale variables to utf-8):

```
system_packages = builtins.attrValues {
  inherit (pkgs) R glibcLocalesUtf8 quarto;
};
```

Finally, all these definitions are used to define a *shell*:

```
in
pkgs.mkShell {
  LOCALE_ARCHIVE = if pkgs.system == "x86_64-linux"
  then
  "${pkgs.glibcLocalesUtf8}/lib/locale/locale-archive"
  else "";
  LANG = "en_US.UTF-8";
  LC_ALL = "en_US.UTF-8";
  LC_TIME = "en_US.UTF-8";
  LC_MONETARY = "en_US.UTF-8";
```

```
  LC_PAPER = "en_US.UTF-8";
  LC_MEASUREMENT = "en_US.UTF-8";

  buildInputs = [rpkgs tex system_packages];
}
```

In this block, a Nix shell environment is defined using pkgs.mkShell. The LOCALE_ARCHIVE variable is conditionally set based on the system architecture. Several environment variables (LANG, LC_ALL, LC_TIME, LC_MONETARY, LC_PAPER, and LC_MEASUREMENT) are set to "en_US.UTF-8". The buildInputs attribute specifies the list of inputs needed for this shell environment, which includes the three sets defined above: R packages (rpkgs), TeX packages (tex), and system packages (system_packages).

This Nix expression defines a development environment with specific R and TeX packages, system packages, and locale settings. When this expression is evaluated using Nix, it will generate a shell environment that includes all the specified dependencies, allowing you to work with R and TeX in a controlled and reproducible environment.

This environment can be built using the `nix-build` command, and users can then *drop* into that shell using `nix-shell`.

Writing these Nix expressions is not easy, and there is a lot of boilerplate code. To simplify the process of writing these expressions, a package I wrote, called `{rix}`, can help you.

## 10.6 The {rix} package

{rix} is an R package that provides functions to help you write Nix expressions: these expressions can then be used by the Nix

package manager to build completely reproducible development environments. These environments can be used for interactive data analysis or running pipelines in a CI/CD environment. Environments built with Nix contain R and all the required packages that you need for your project: there are currently more than 80.000 pieces of software available through the Nix package manager, including the entirety of CRAN and Bioconductor packages. The Nix package manager is extremely powerful: not only it handles all the dependencies of any package extremely well, it is also possible with it to reproduce environments containing old releases of software. It is thus possible to build environments that contain R version 4.0.0 (for example) to run an old project originally developed on that version of R.

First, you need to install the Nix package manager on your system. For this, we are going to use the installer from Determinate Systems. Simply run the following command in a terminal:

```
curl --proto '=https' --tlsv1.2 -sSf -L
https://install.determinate.systems/nix | sh -s --
install
```

If you wish to uninstall Nix, run the same command. Then, if you already have R installed on your system, you can install the `{rix}` package using:

```
install.packages("rix")
```

From there, you can start a new R session and try out `{rix}` like so:

```
rix(r_ver = "latest",
    r_pkgs = c("dplyr", "ggplot2"),
    system_pkgs = NULL,
```

```
    git_pkgs = NULL,
    ide = "other",
    project_path = ".",
    overwrite = TRUE)
```

this will create a `default.nix` file in the project root. Open a terminal where `default.nix` is, and run `nix-build`. This will create a file called `result` in the same folder. You can now *drop* into a shell with the specified packages using `nix-shell`.

## 10.7 Running a pipeline with Nix

Once you've built and environment, and "dropped" into it, it's possible to run R by simply typing `R` in the console. If instead you've installed an IDE, you can start is as well by typing the IDE name's. You can then work interactively with your data. But it is also possible to run a command from that environment. For instance, if you have a `{targets}` pipeline that you wish to run in an environment built with Nix, you could run the following command (inside the folder containing the `default.nix` file):

```
nix-shell default.nix --run "Rscript -e
'targets::tar_make()'"
```

This will run the pipeline and build the output. If the output is a rendered Quarto document for instance, you will then see the document appear in the specified output folder.

## 10.8 CI/CD with Nix

It is also possible to run a `{targets}` pipeline on Github Actions quite easily. Run `rix::tar_nix_ga()` to add the file `.github/workflows/run-pipeline.yaml` to your project. Now, each time you push changes to your Github repository, the pipeline will be executed. Don't forget to give read and write rights to the Github Actions bot. You will find the outputs of the pipeline in the `targets-run` branch of your repository. See this repository for an example.

## 10.9 A polyglot environment

A polyglot environment is an environment that supplies several languages, for data science that would typically be Python and R. See this example, which outputs a shell with both R and Python packages. It is also possible to orchestrate how the languages "talk" to each other using Nix itself. For example suppose that you want to generate a Quarto document that needs both R and Python. You could of course use dedicated chunks within the Quarto document for this. But another way is instead to use Nix as a build automation tool: this means that with only Nix, you can replace a tool such as `{targets}` and another tool for dependency management. This repository contains such an example.

## 10.10 Conclusion

Nix is a very powerful tool, with a steep learning curve. Check out these resources to learn more:

- Official Nix docs: https://nix.dev/
- Nix pills: https://nixos.org/guides/nix-pills/00-preface
- INRIA Nix tutorial: https://nix-tutorial.gitlabpages.inria.fr/nix-tutorial/
- rix's website (check out the Articles section): https://docs.ropensci.org/rix/
- rix's Github: https://github.com/ropensci/rix
- My blog posts: https://www.brodrigues.co/tags/nix/

# 11 What else should you learn?

Here's a list of things I think would be nice for you to invest some time in, in no particular order.

## 11.1 Touch typing

One of the things I NEVER see discussed when talking "upskilling" is improving your typing speed. According to a survey (which I'm sure is not statistically, nor scientifically sound, but still…) by onlinetyping.org (which you can find here, most back office workers (who spend all day typing) have a typing speed of 20 to 30 wpm (words per minute). According to this article by the Atlantic people write about 41638 words in email per year. You as programmers (yes, even if you're focused on data, you're a programmer) very surely type twice or thrice this amount of words per year. But let's stay with 41638 words per year. That would translate to almost 28 days of non stop typing at a typing speed of 25 words per minute. Doubling to 50 wpm is actually quite easy, and reaching 70 is really doable. This could improve productivity, or better yet, make you go home earlier instead of working until 19h00 every day because you type like a snail.

You need to learn touch typing, meaning, typing without looking at your keyboard.

## 11.2 Vim

Yes, I think you should learn vim, or at the very least, your text editor of choice, by heart. You should know every keyboard shortcut and every possibility that your text editor offers. You should never touch the mouse when writing text. This is not just because of productivity, but also for your health. Grabbing the mouse to click one or twice, and then go back to typing, then go back to moving the mouse, etc, will destroy your shoulder. By keeping your hands on the keyboard at all times and minimizing mouse usage, you may be able to grow old healthy. Vim helps with that because it is a modal text editor (and most editors actually ship a Vim-mode). Watch this video to get a quick introduction on Vim, and how to enable Vim mode in Vscode.

## 11.3 Statistical modeling

Statistical modeling is crucial, and if you didn't major in stats, you very likely lack this knowledge. Here's a reading (and watching) list:

- Regression and other stories (has a free PDF)
- Statistical Rethinking 2022 (on youtube)
- Mostly harmless econometrics

# 12 Conclusion

## 12.1 Why bother

We're at the end of this course, which I hope you enjoyed. There is now yet another question we need to ask ourselves: is this worth it? Why should we bother with making our pipelines reproducible? I believe that there are two, fundamental, essential reasons.

The first one, is that time is finite, and working manually does not scale. Reproducible pipelines do take time to set up, but they allow us to win this time back once we start re-running them. Wasting time and resources running things manually (with the potential for introducing errors) is simply not acceptable. This freed up time can then be used to provide further value to your employer, yourself, and ideally your community as well.

The second reason, is that setting up RAPs is in itself an enjoyable activity, which requires the full depth and breadth of your skills. If you're working in science, there is the added benefit that by setting up a RAP you're doing actual science: providing a reproducible analysis where an hypothesis gets tested (an not writing papers).

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27.