Building Reproducible Analytical Pipelines

Master of Data Science, University of Luxembourg - 2023

Bruno Rodrigues

20/11/2023

Table of contents

Introduction

This is the 2023 edition of the course. If you're looking for the 2022 edition, you can click here

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

- 2022/10/25, morning: Introduction and data exploration with R
- 2022/10/25, afternoon: Functional programming
- 2022/10/31, afternoon: Git (optional?)
- 2022/11/07, afternoon: Package development and unit testing
- 2022/11/08, morning: Build automation
- 2022/11/08, afternoon: Building Data products
- 2022/11/21, afternoon: Self-contained pipelines with Docker
- 2022/11/22, morning: CI/CD with Github Actions

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.

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 19'000 packages (as of October 2022) 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`…, 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).

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 I can pull it: 2 points;
- Data and functions to run pipeline are in a tested, documented package? 3 points;
- I don't need to do anything to load data: 5 points;
- I can download and install your pipeline's dependencies in one command line: 5 points;
- I can run your pipeline in one command line: 5 points
- Extra points: pipeline is dockerized and 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.

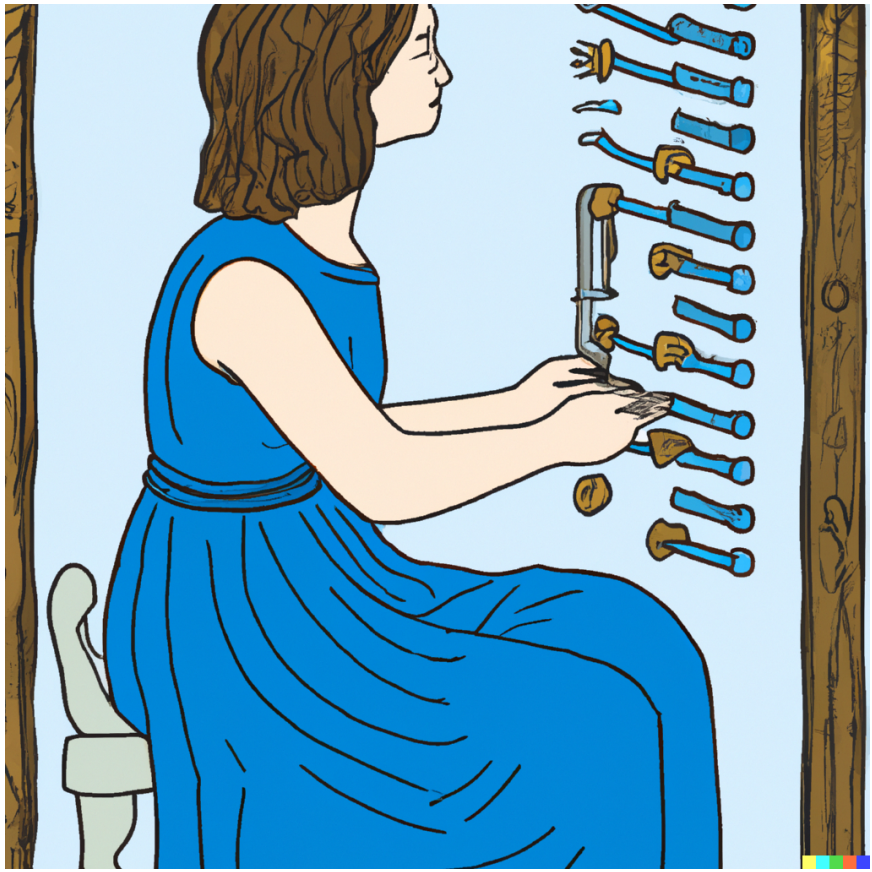
Further reading

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

License

This course is licensed under the WTFPL.

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()
```

```
[1] 2
```

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)
```

```
Attaching package: 'dplyr'

The following objects are masked from
'package:stats':

    filter, lag

The following objects are masked from
'package:base':

    intersect, setdiff, setequal, union
```

```
4 %>%
  sqrt()
```

```
[1] 2
```

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.

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)
```

```
 [1]  1.2529614 -1.1705061 -0.4026287 -1.0083698
-0.3530011  0.1846006
 [7] -1.2095581 -0.1363026  0.2885044 -1.0096258
```

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)
```

```
 [1] -1.2070657  0.2774292  1.0844412 -2.3456977
0.4291247  0.5060559
 [7] -0.5747400 -0.5466319 -0.5644520 -0.8900378
```

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)
```

```
[1] -1.2070657  0.2774292  1.0844412 -2.3456977
0.4291247  0.5060559
[7] -0.5747400 -0.5466319 -0.5644520 -0.8900378
```

```
rnorm(10)
```

```
[1] -0.47719270 -0.99838644 -0.77625389  0.06445882
0.95949406 -0.11028549
[7] -0.51100951 -0.91119542 -0.83717168  2.41583518
```

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

```
[1] -1.2070657  0.2774292  1.0844412 -2.3456977
0.4291247  0.5060559
[7] -0.5747400 -0.5466319 -0.5644520 -0.8900378
```

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)
})
```

```
 [1] -1.2070657  0.2774292  1.0844412 -2.3456977
0.4291247  0.5060559
 [7] -0.5747400 -0.5466319 -0.5644520 -0.8900378
```

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)
```

```
 [1] -1.2070657  0.2774292  1.0844412 -2.3456977
0.4291247  0.5060559
 [7] -0.5747400 -0.5466319 -0.5644520 -0.8900378
```

`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:

```
suppressPackageStartupMessages(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))
```

```
# A tibble: 3 x 2
  is_human mean_height
```

```
  <lgl>          <dbl>
1 FALSE           172.
2 TRUE            177.
3 NA              181.
```

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")
```

```
# A tibble: 10 x 3
   sex    statistic         value
   <chr>  <chr>             <dbl>
 1 female total_individuals   6
 2 female min_mass            45
 3 female mean_mass           56.3
 4 female sd_mass             16.3
 5 female max_mass            75
 6 male   total_individuals   5
 7 male   min_mass            79
 8 male   mean_mass           90.5
 9 male   sd_mass             19.8
10 male   max_mass           120
```

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

2.3 Functional programming

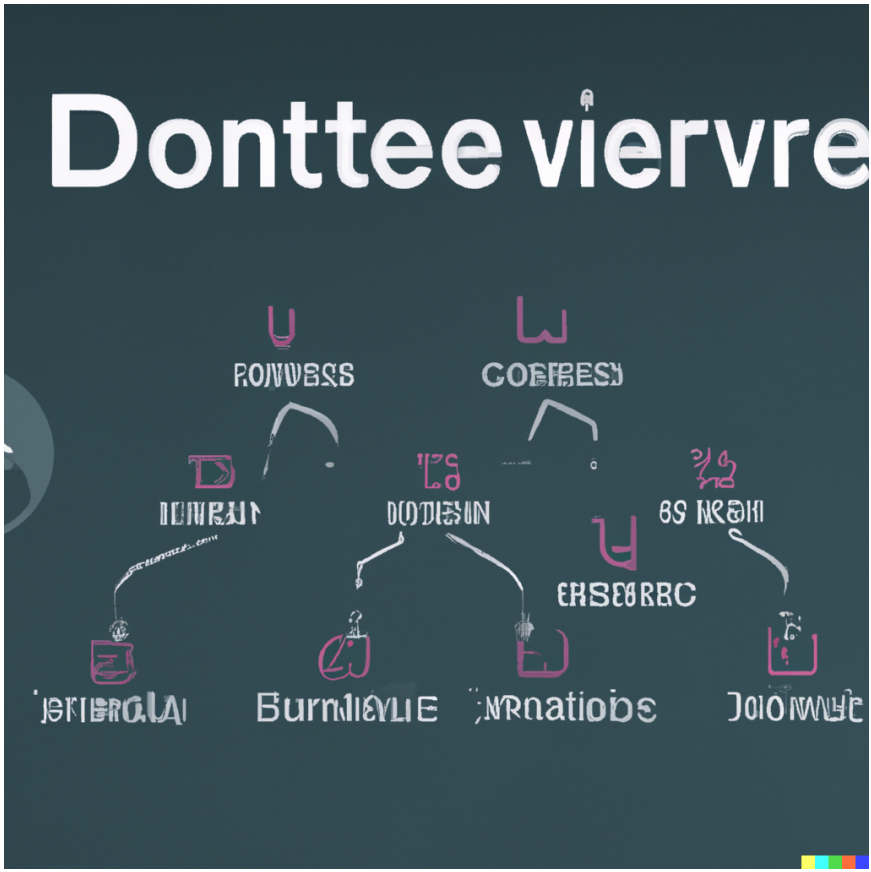
You should ideally work through the whole of chapter 7, and then tackle chapter 8. What's important there are:

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

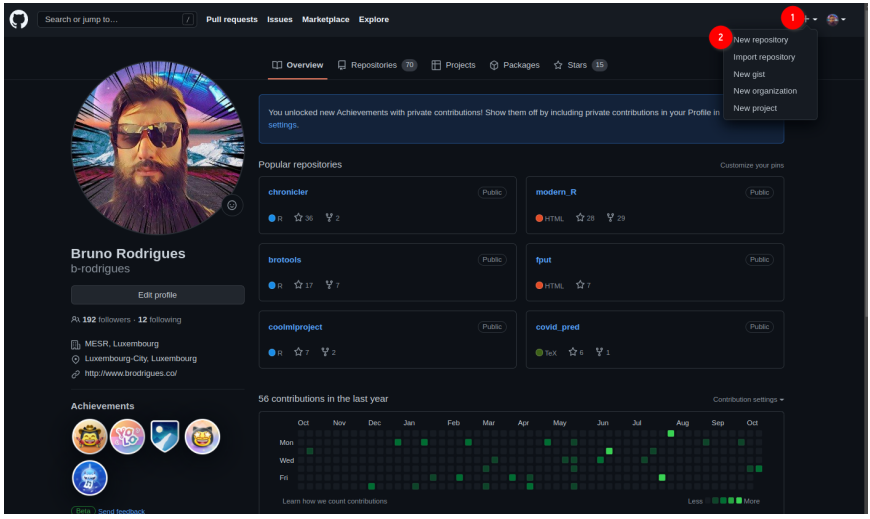
3.2 Installing Git

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

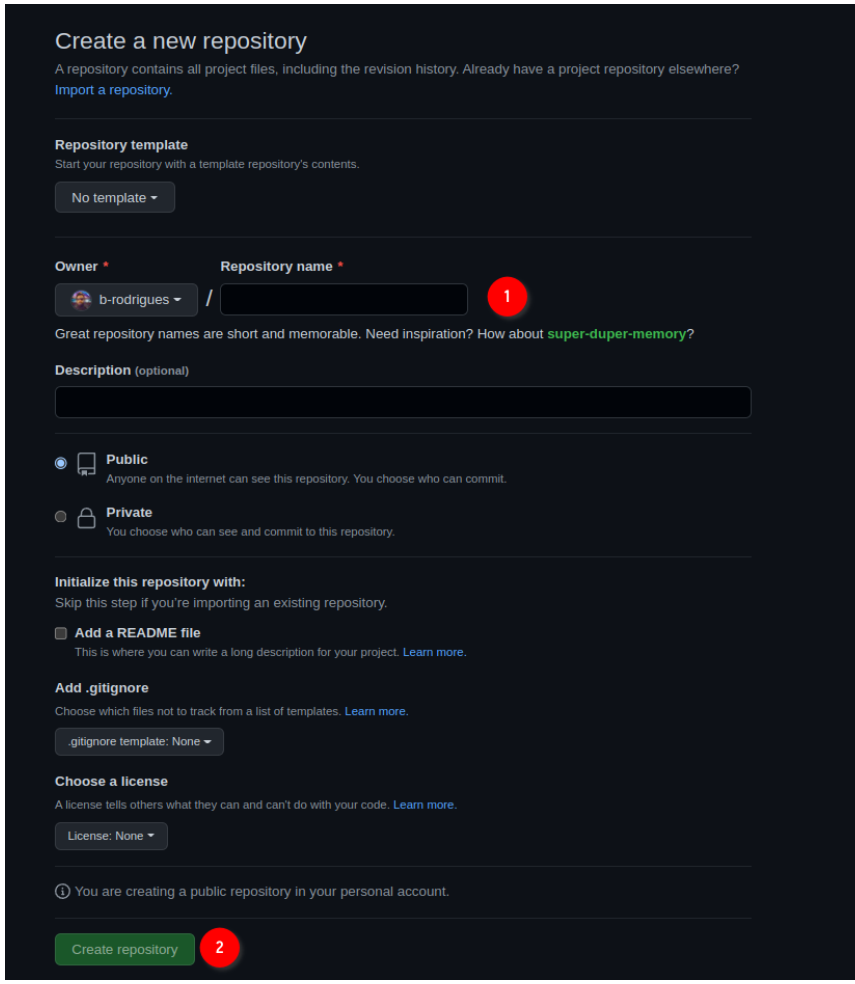
homebrew or macports then install the binary package from https://sourceforge.net/projects/git-osx-installer/.

3.3 Setting up a repo

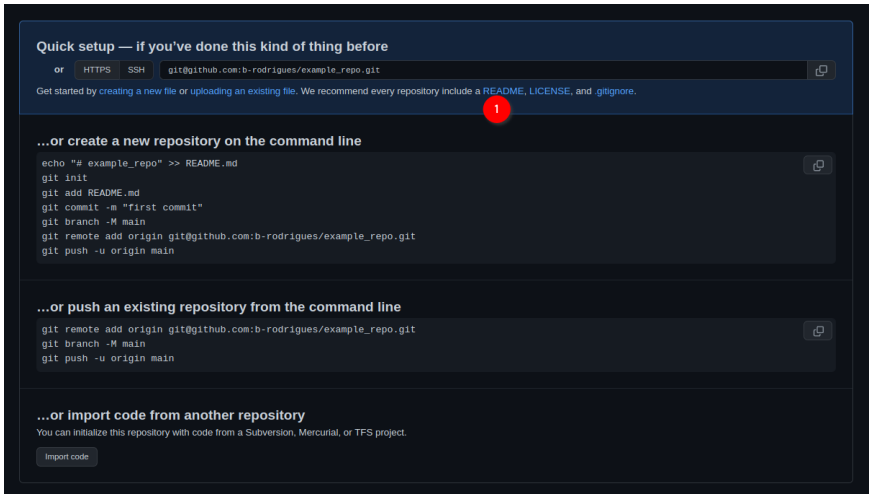
Ok so now that Git is installed, we can actually start using it. First, let's start by creating a new repository on github.com. As I've mentioned in the introductory paragraph, Git will allow you to interact with github.com, and you'll see in what ways soon enough. For now, login to your github.com account,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



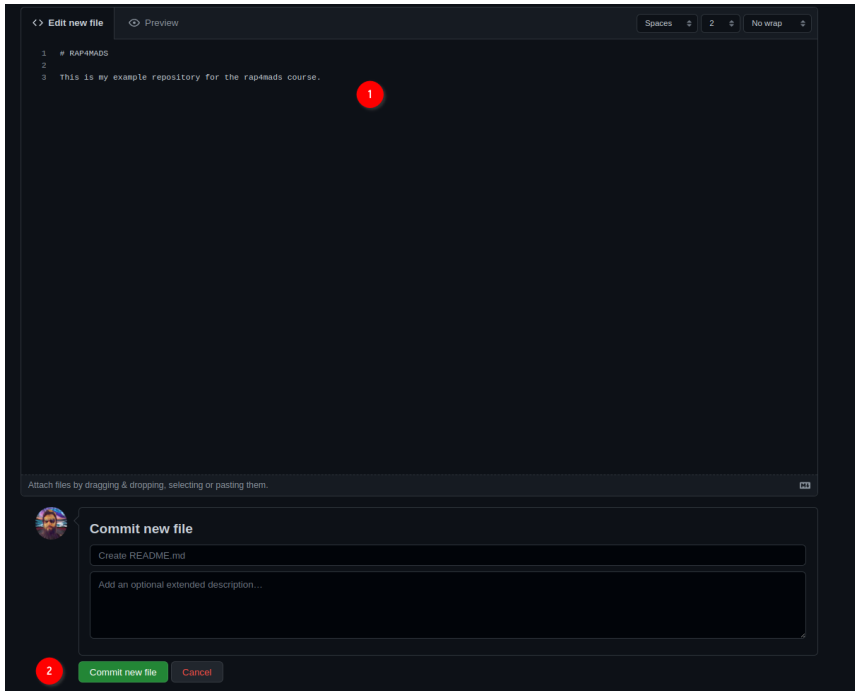
In the next screen, choose a nice name for your repository and ignore the other options, they're not important for now. Then click on 'Create repository':



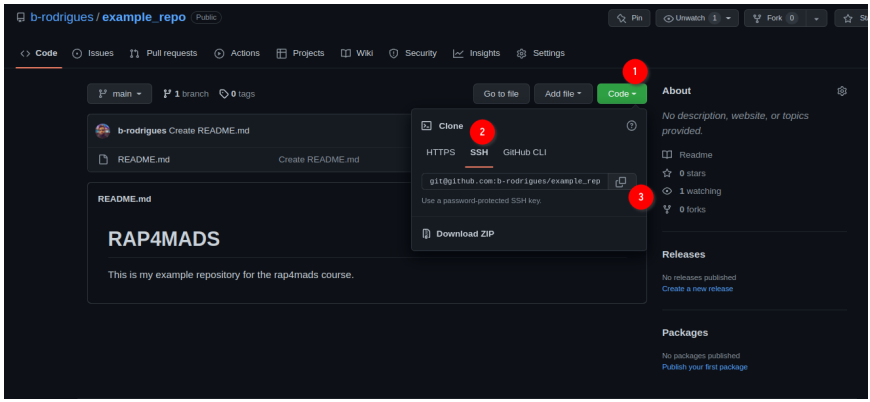
Ok, we're almost done with the easy part. The next screen tells us we can start interacting with the repository. For this, we're first going to click on 'README':



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



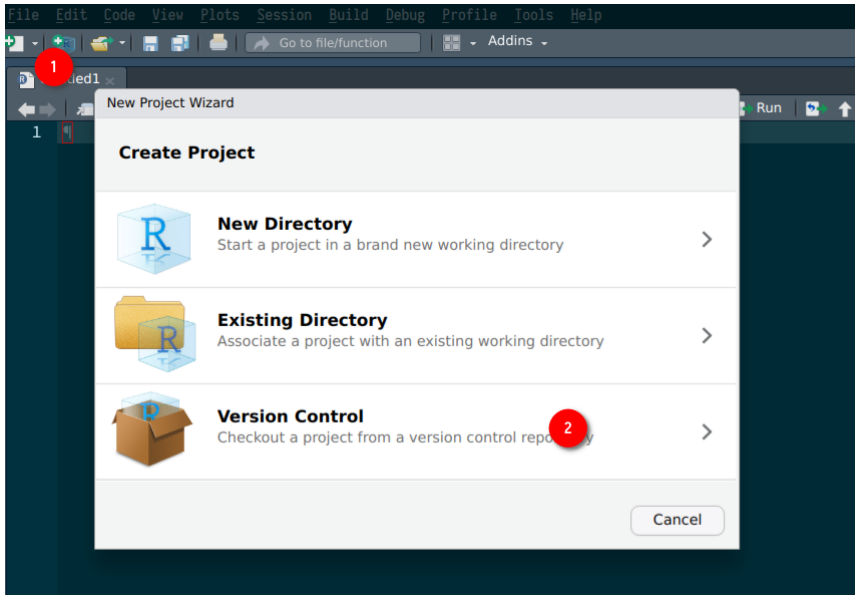
Add some lines to the file, and then click on ‘Commit new file’. You’ll end up on the main page of your freshly created repository. We are now done with setting up the repository on github.com. We can now *clone* the repository onto our machines. For this, click on ‘Code’, then ‘SSH’ and then on the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



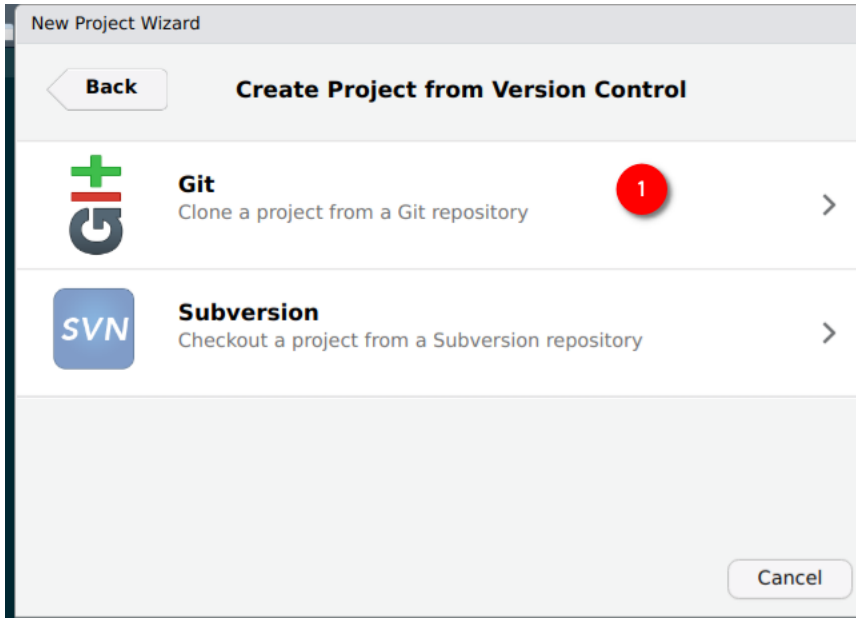
Now, to make things easier on you, we're going to use Rstudio as an interface for Git. But you should know that Git can be used independently from a terminal application on Linux or macOS, or from Git Bash on Windows, and you should definitely get familiar with the Linux/macOS command line at some point 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

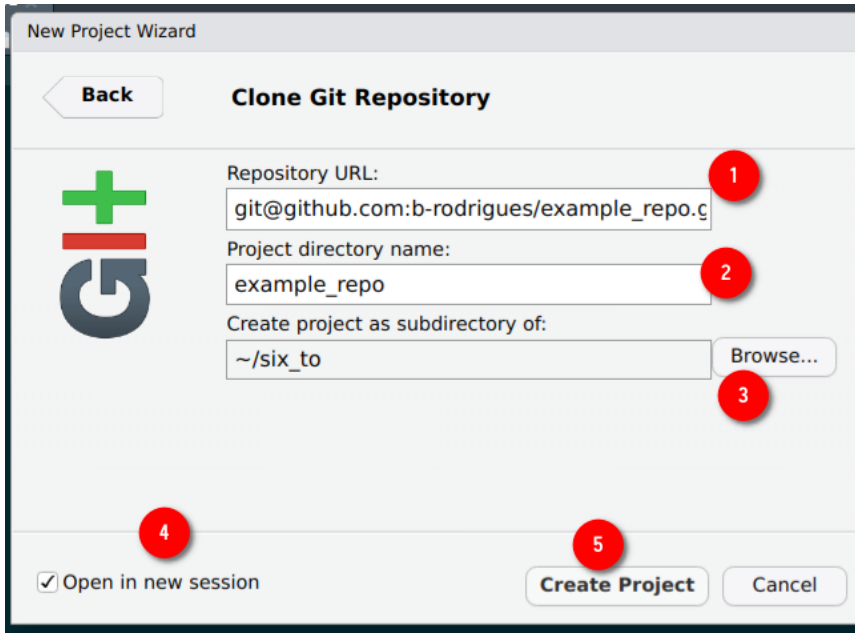
Start Rstudio and click on 'new project' and then 'Version Control':



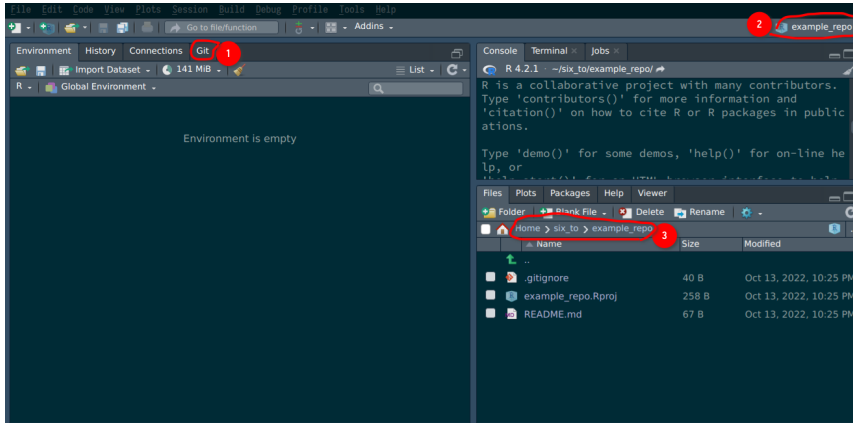
Then choose ‘Git’:



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:

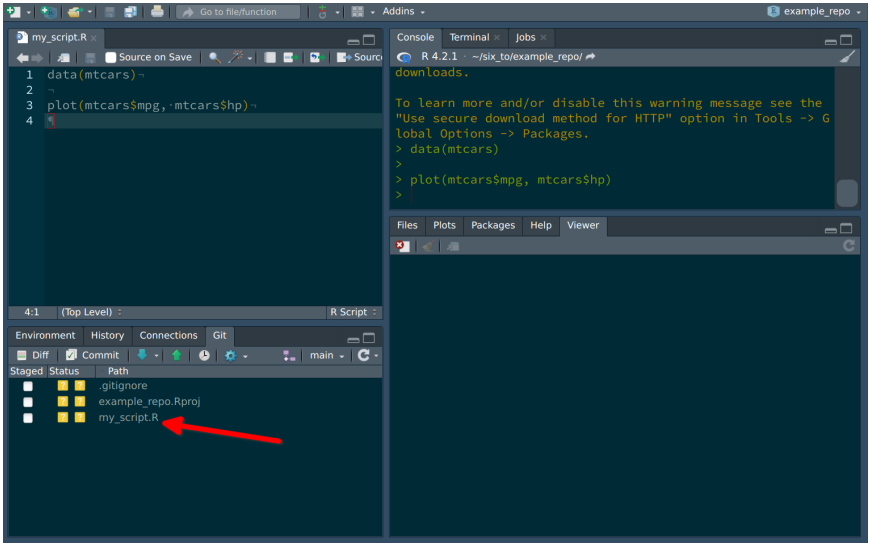


Icon (1) indicates that this project is *git-enabled* so to speak.

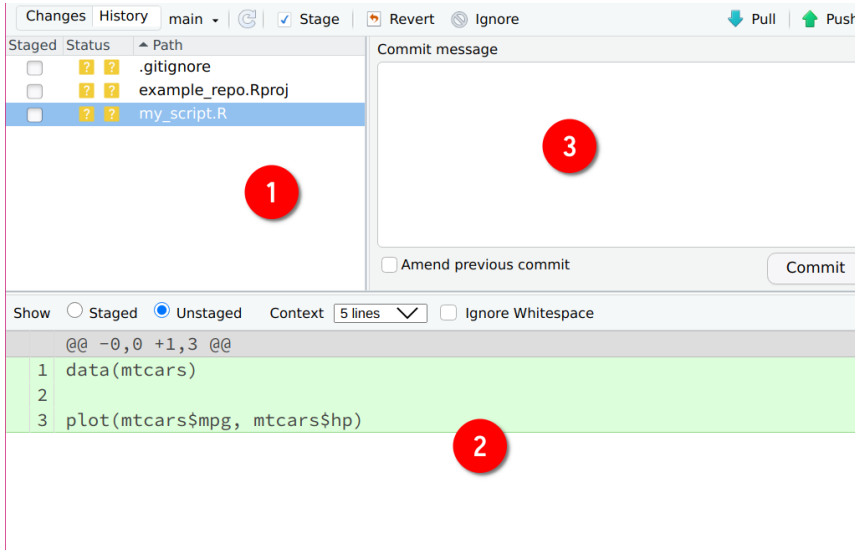
(2) shows you that Rstudio is open inside the `example_repo` (or whatever you named your repo to) project, and (3) shows you the actual repository that was downloaded from github.com at the path you chose before. You will also see the `README` file that we created before.

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:

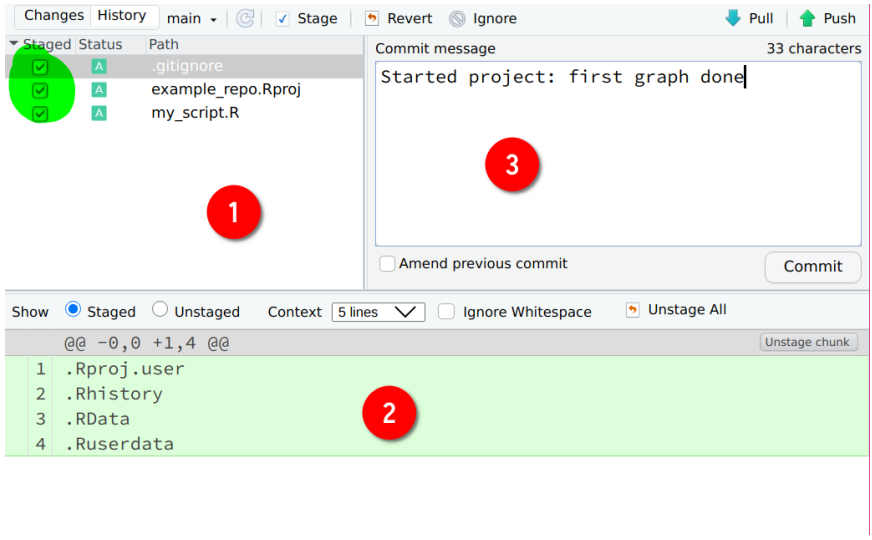


We are now ready to commit the file, but first let's check out what actually changed. If you click on `Diff`, a new window will open with the different files that changed since last time:



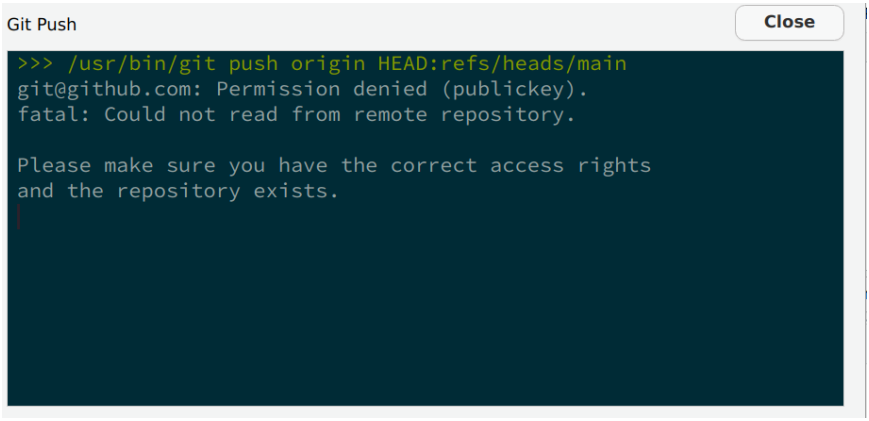
Icon (1) shows you the list of files that changed. We only created the file called `my_script.R`, but two other files are listed as well. These files are automatically generated when starting a new project. `.gitignore` lists files and folders that Git should not track, meaning, any change that will 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 re-

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

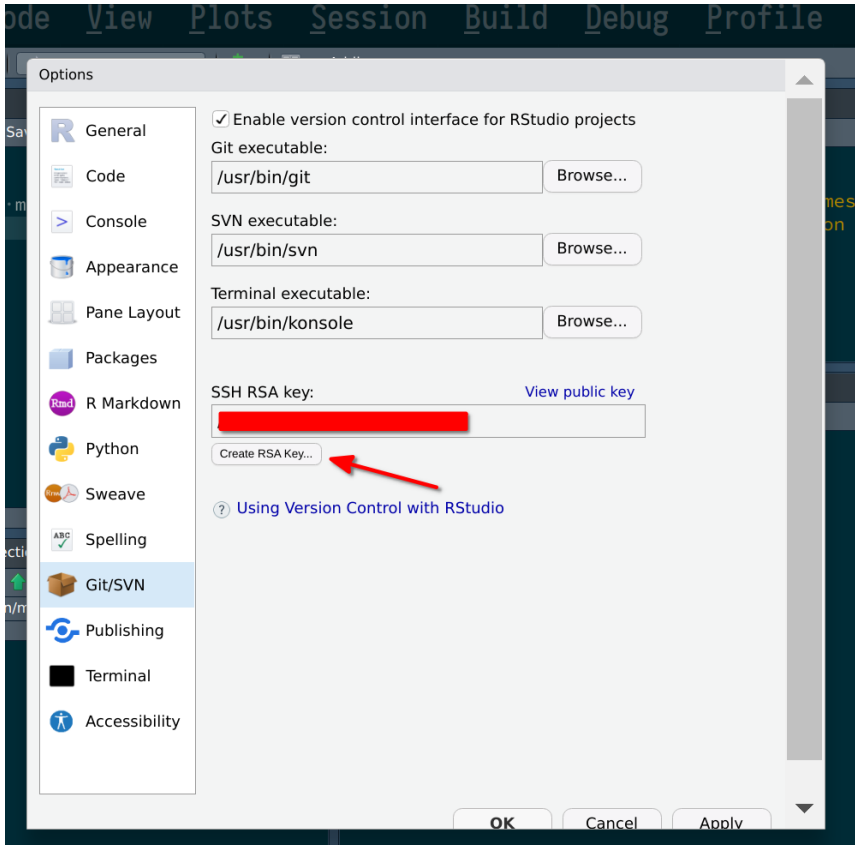


Ooooooops! Something's wrong! Apparently, we do not have access rights to the repo? This can sound weird, because after all, we created the repo with our account and then cloned it. So what's going on? Well, remember that anyone can clone a public repository, but only authorized people can push changes 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.

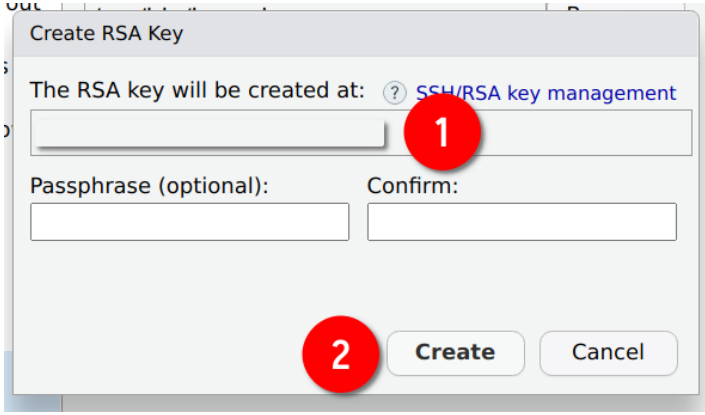
Now if you thought that Git was confusing, I'm sorry to say that what's coming confuses students in general even more. Ok so what's a ssh key, and why does Git need it? *An* ssh key is actually a misnomer, because we should really be talking about

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.

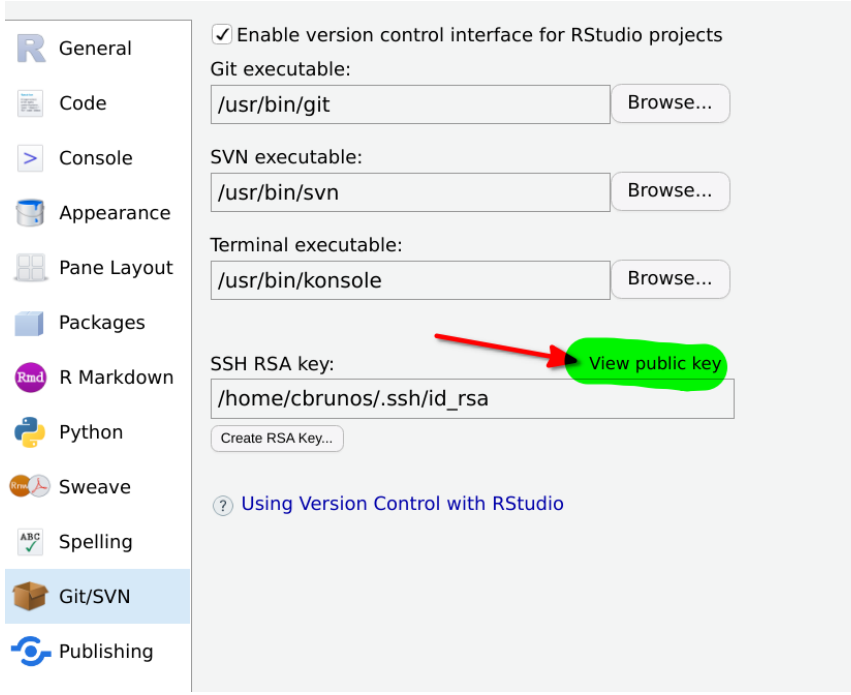
So let's now generate an ssh key pair. For this, go to `Tools > Global Options > Git/Svn`, and then click on the `Create RSA Key...`



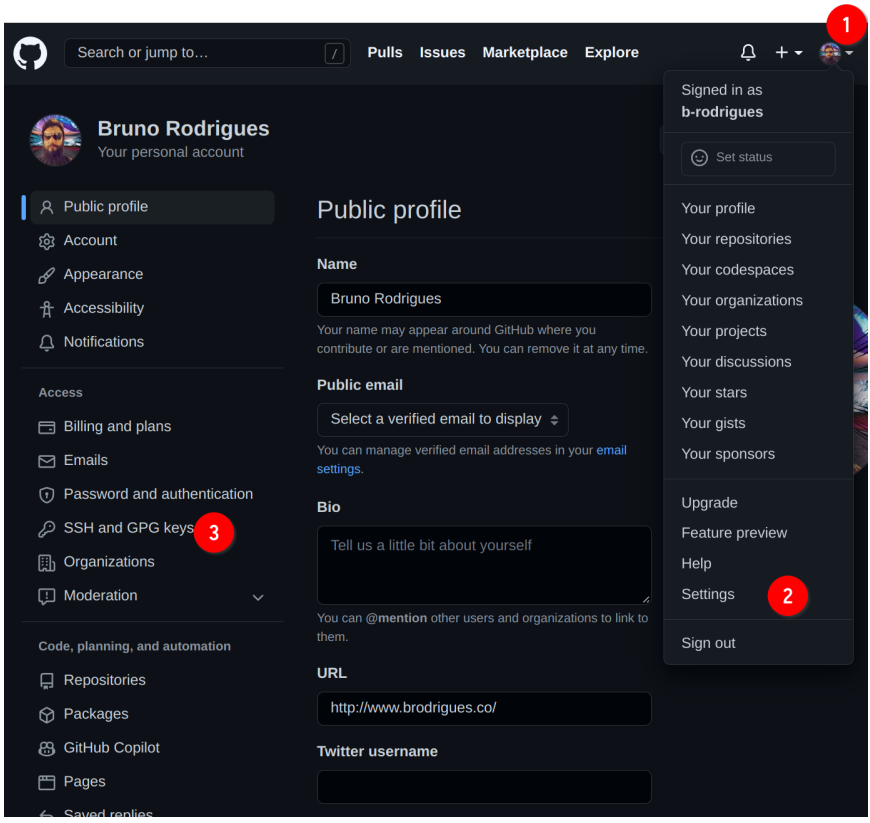
Icon (1) shows you the path where the keys will be saved. This is only useful if you have reasons to worry that your private key might be compromised, but without physical access to your machine, an attacker would have a lot of trouble retrieving it (if you keep your OS updated…). Finally click on `Create`:



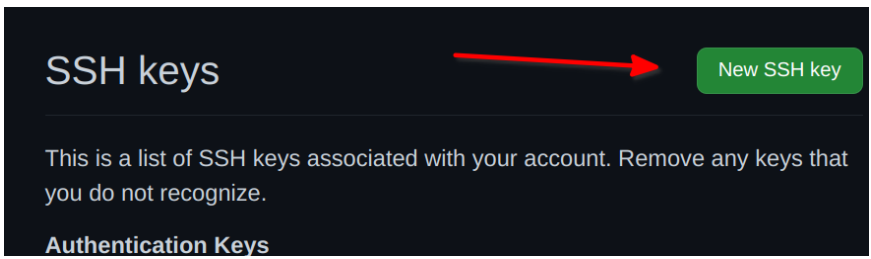
Ok so now that you have generated these keys, let's copy the public key in our clipboard (because we need to paste the key into github.com). You should be able to find this key from RStudio. Go back to `Tools > Global Options > Git/Svn`, and then click on `View public key`:



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`:

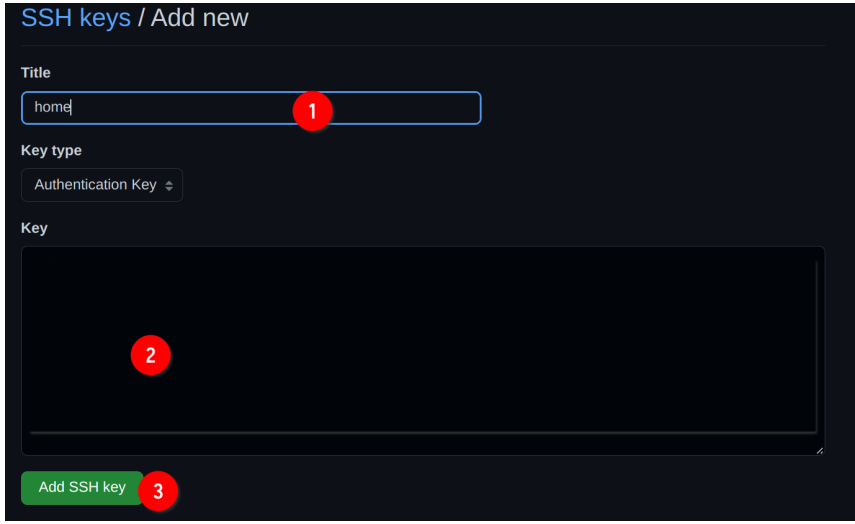


Then, on the new screen click on `New SSH key`:

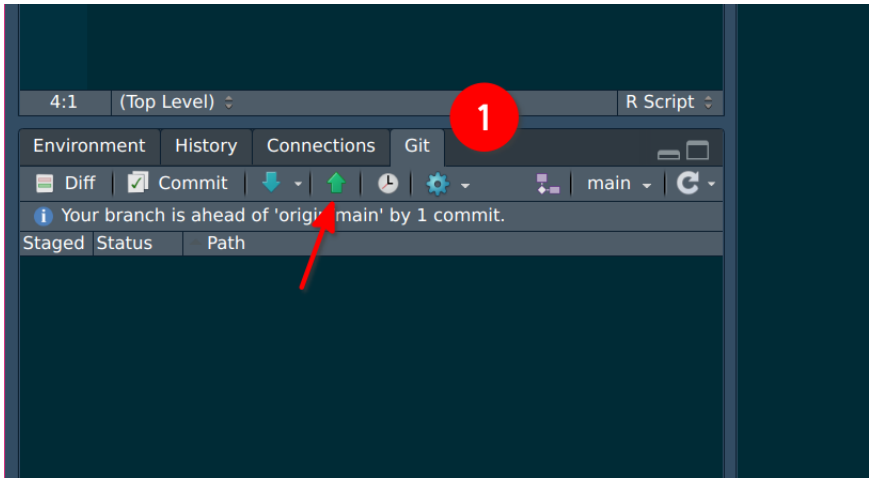


You can now add your key. Add a title, for example `home` for

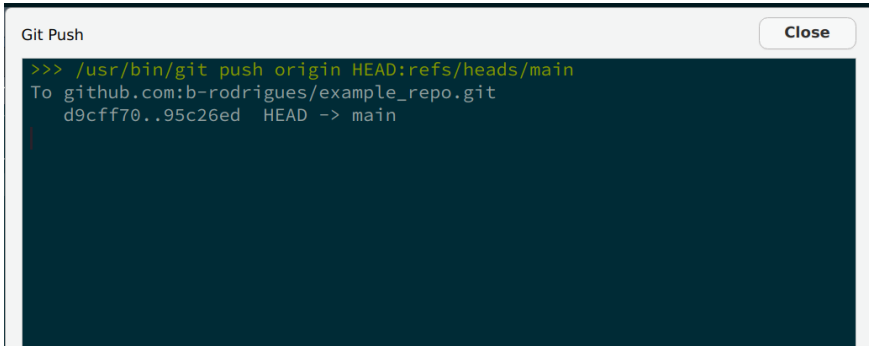
your home computer, or `work` for your work laptop. Paste the key from RStudio into the field (2), and then click on `Add SSH key`:



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`:



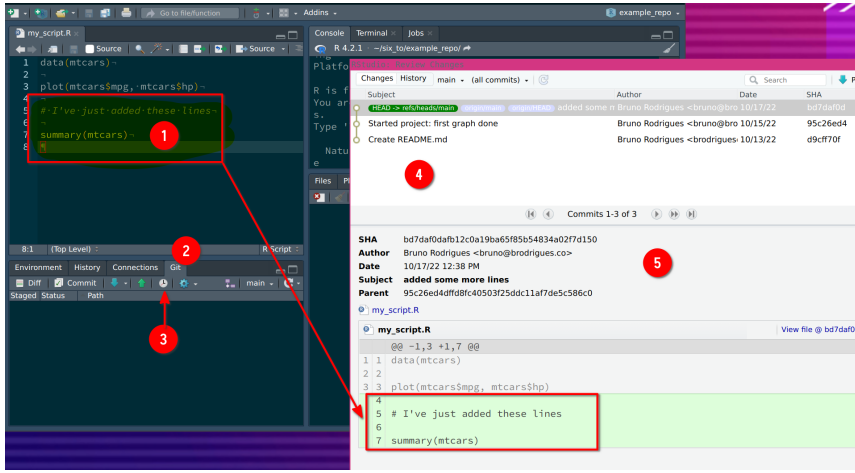
A new window will open, this time showing you that the upload went through:



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

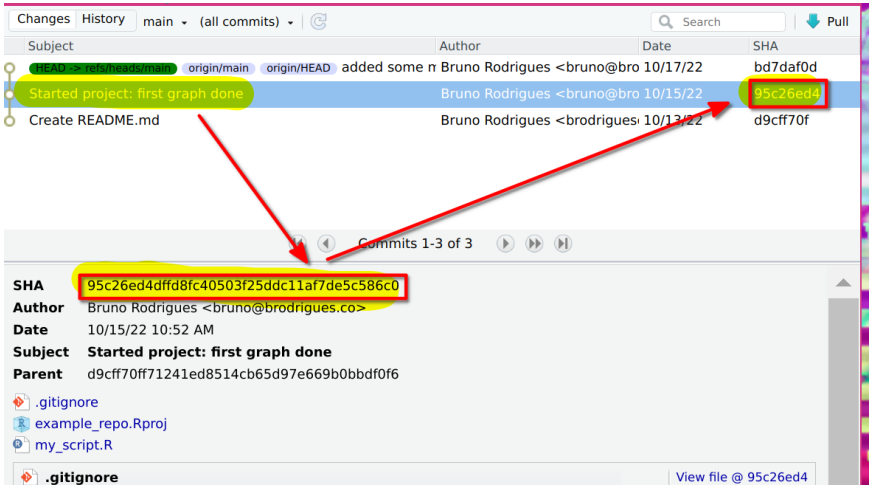
will need to provide the public key to the central server. All of this ssh key pair business IS NOT specific to github.com, so make sure that you understand this well, because sooner rather later, you will need to provide another public key, either because you work from several computers or because the 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):

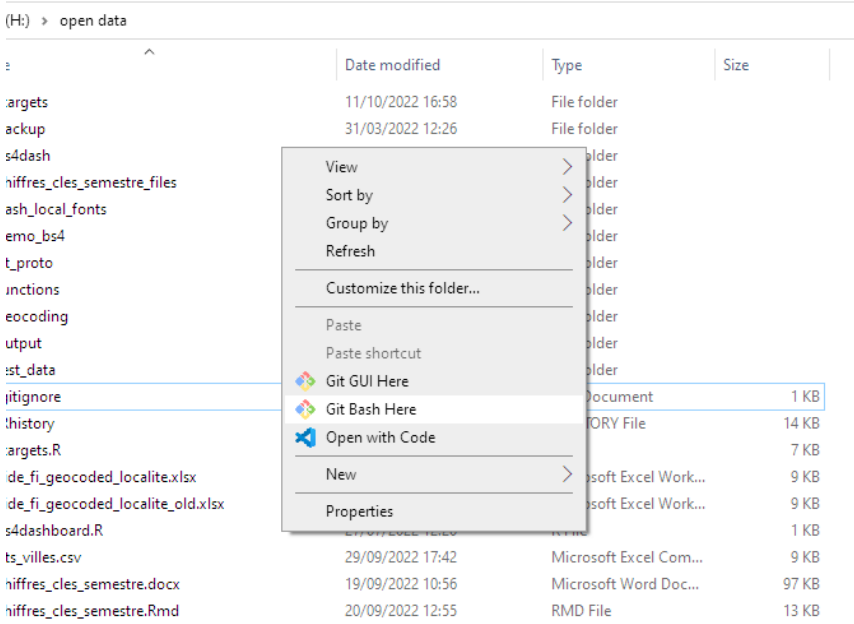


As you can see from the picture above, clicking on `History` shows every commit since the beginning of the repo. It also

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]

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
```

[1] Remember the introduction to this book, where I discussed everything else that you should know…

```
# 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
##
## 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.

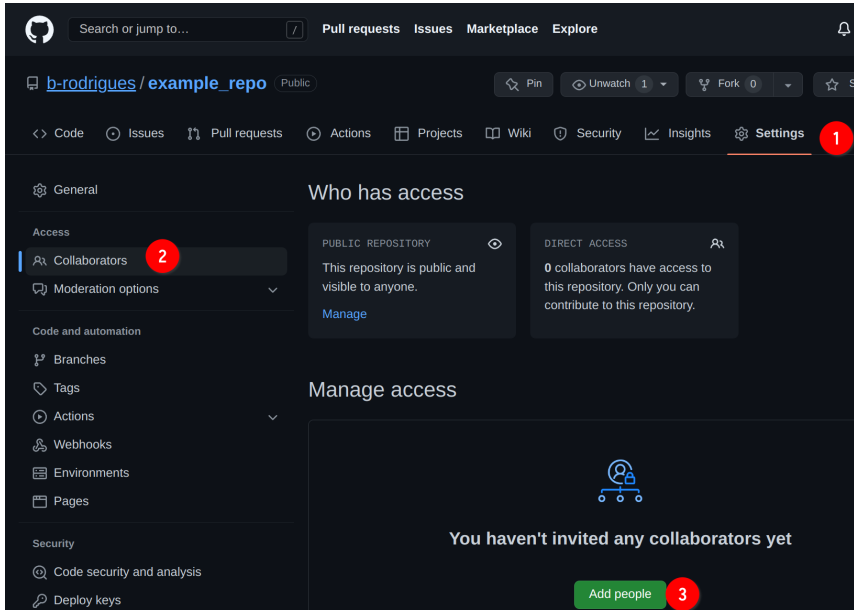
Ok, we're doing with the basics. Let's now see how we can contribute to some repository.

3.6 Collaborating

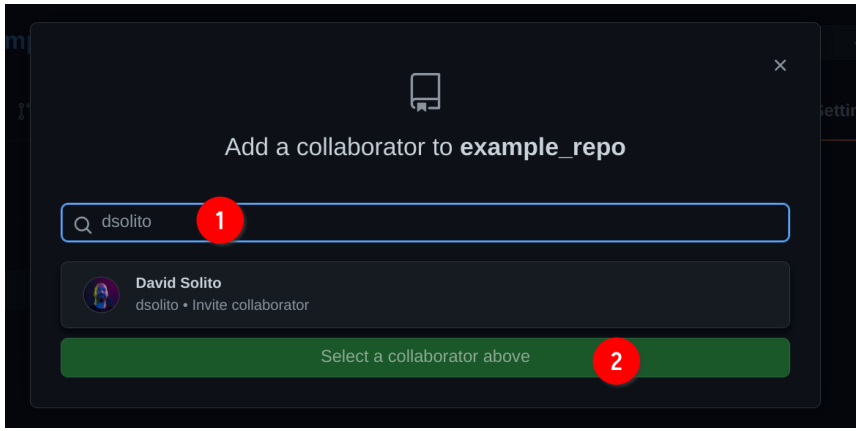
Github (and similar services) allow you to collaborate with people. There are two ways of achieving this. You can invite people to work with you on the same project, by giving them writing rights to the repository. This is what we are going to cover in this section. The other way to collaborate is to 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:



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.

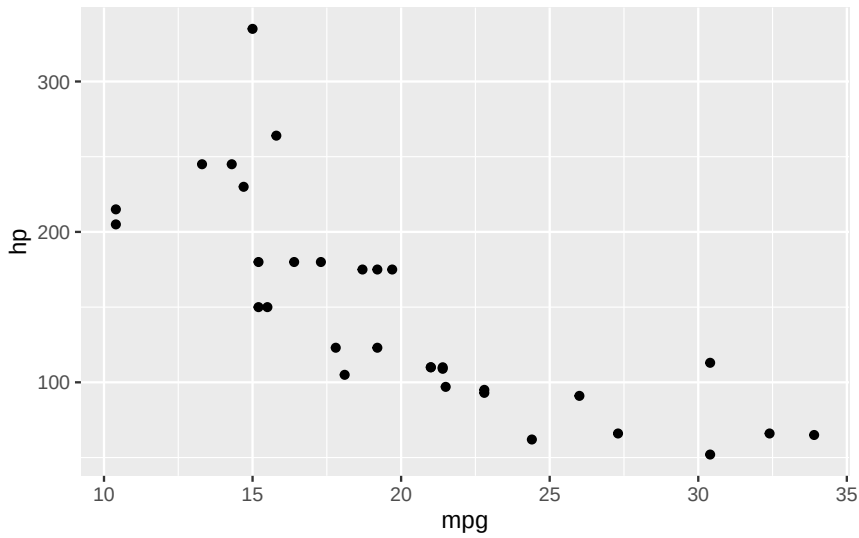
The file should look like this (yours might be different, it doesn't matter):

```
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
```

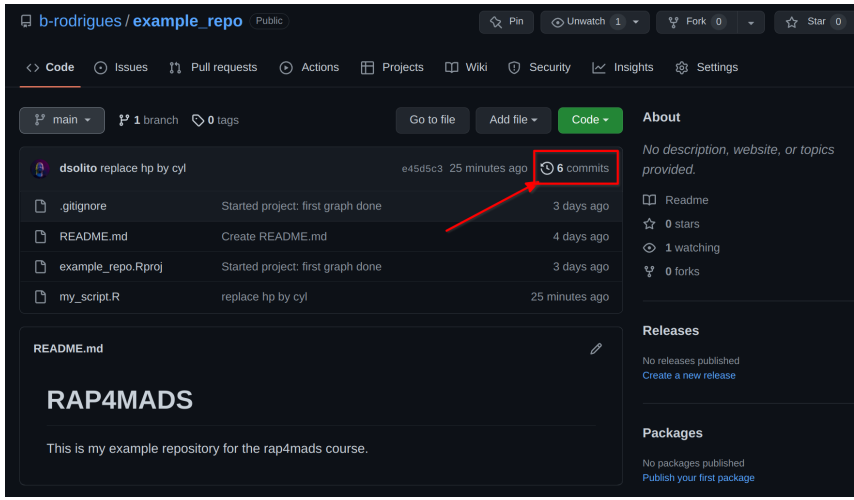
And this is what happens:

```
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.
```

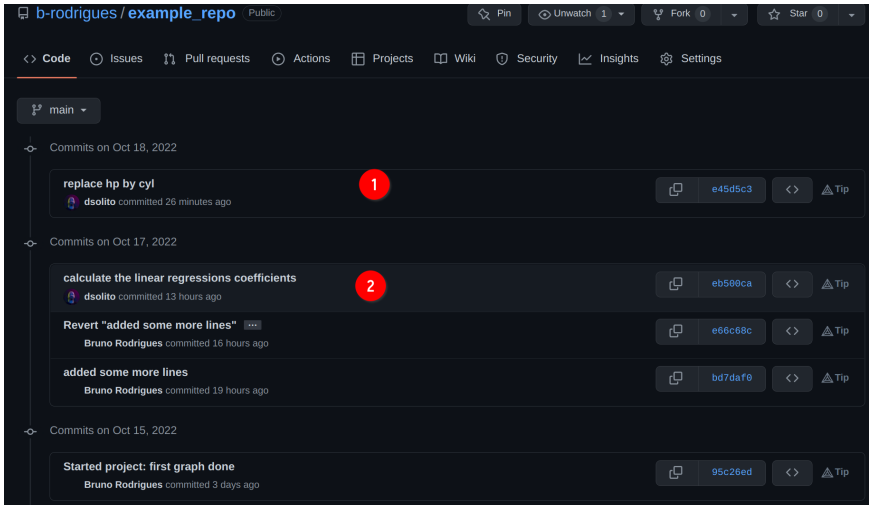
What this all means is that David already pushed some changes while I was working on the project as well. It says so very cleary *Updates were rejected because the remote contains work*

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:



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
```

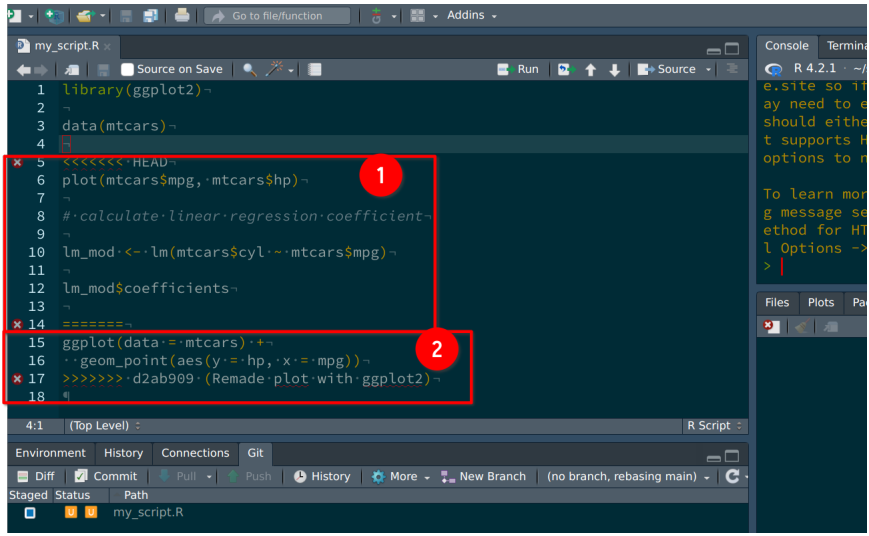
`--rebase` is a flag that keeps the commit history linear. There

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.

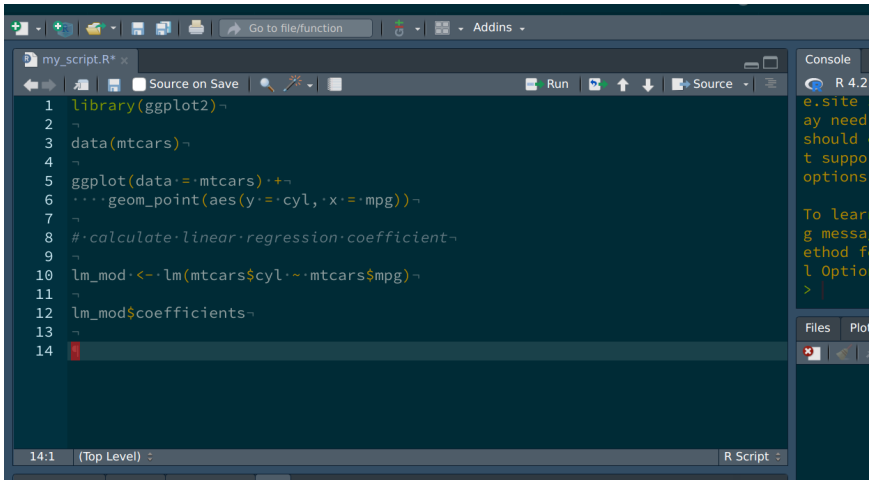
Once `git pull --rebase` is done, we get the following message:

```
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:



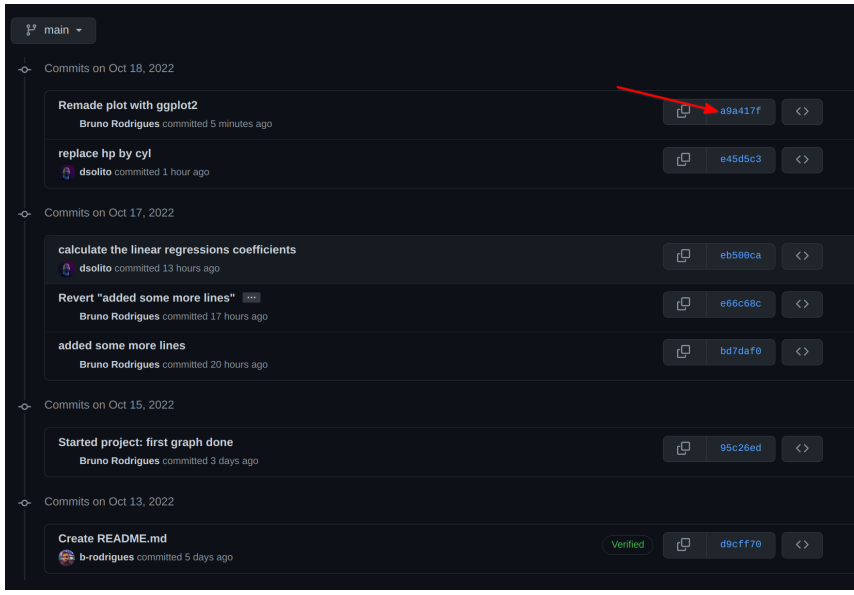
We can now save, and continue following the hints from Git, namely, adding the changed file to the next commit and then use `git rebase --continue`:

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

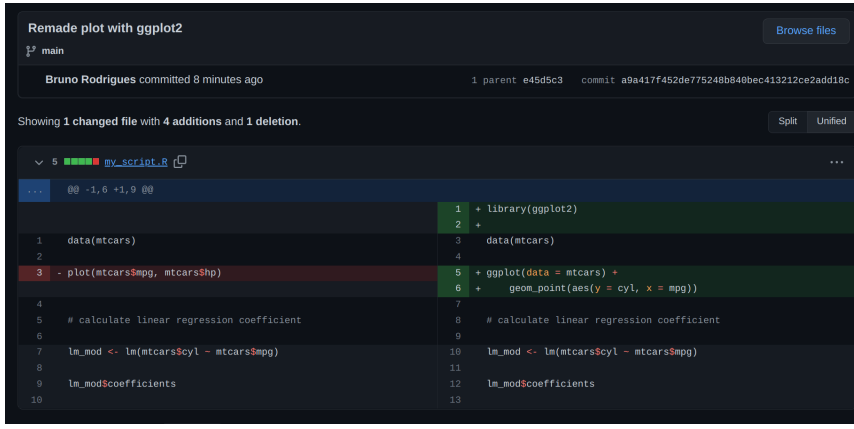
This will once again open the editor in your terminal. Simply close it with `:q`. Let's now push:

```
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):



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.

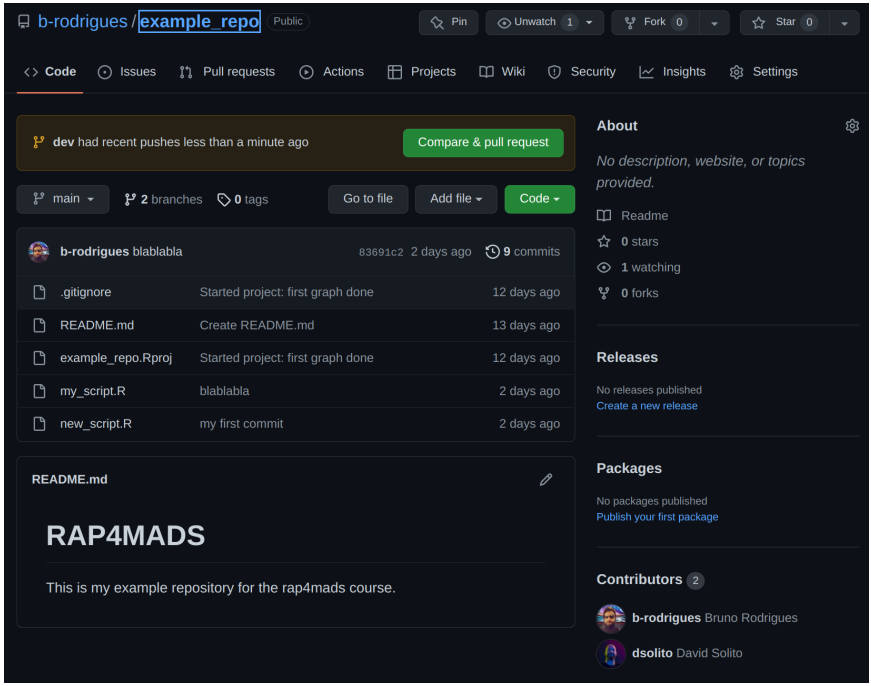
If you want to keep it simple, following this tutorial should be enough. If you want something more fancy, use this other tutorial. I have not followed either, so I don't know if they work, but by the looks of it they should, and it should work on both Linux and macOS I believe. If these don't work,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



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:

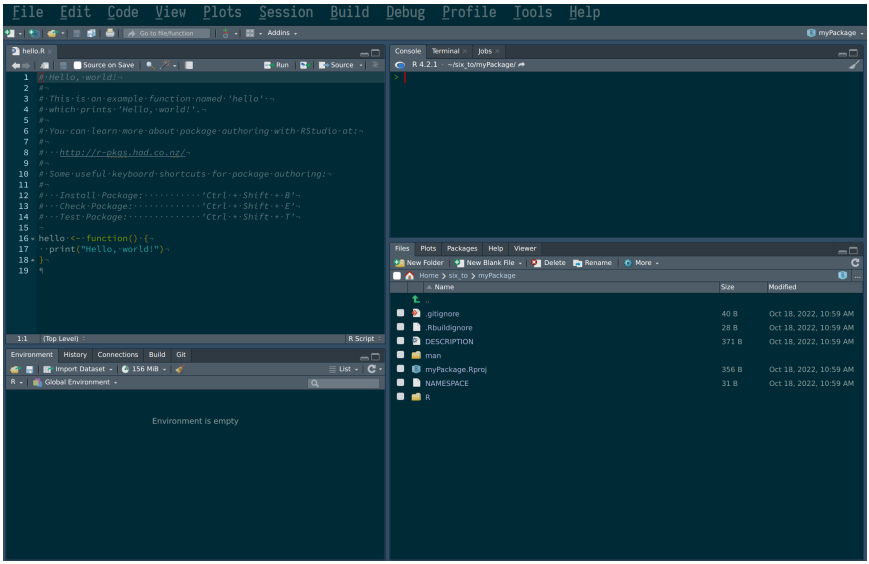
```
remotes::install_github("github_username/some_package")
```

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:

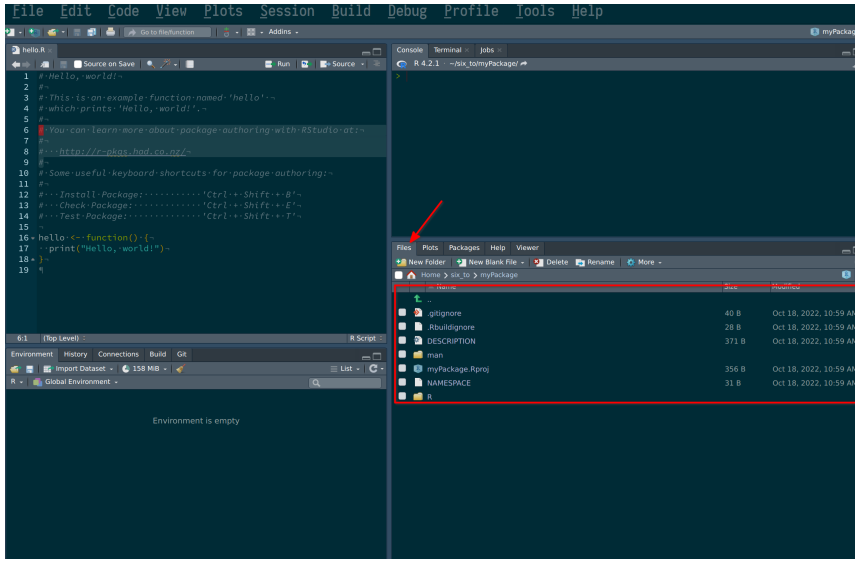


We can remove this script, but do take note of the following sentence:

```
# 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:

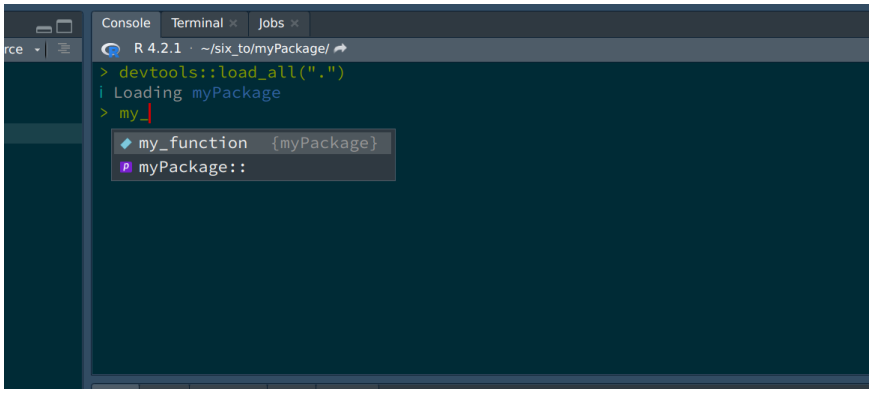


But you can also navigate to the folder from inside a file explorer. The folder that will matter to us the most for now is the `R` folder. This folder will contain your scripts, which will contain your package's functions. Let's start by adding a new script.

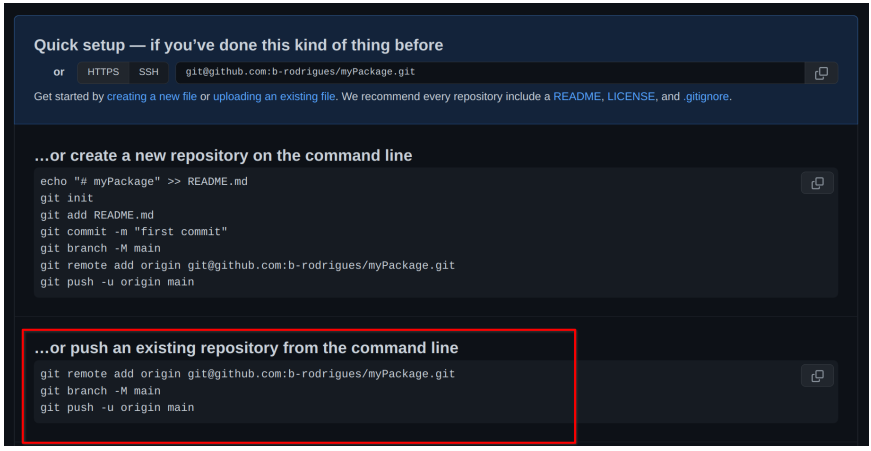
4.3 Adding functions

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.

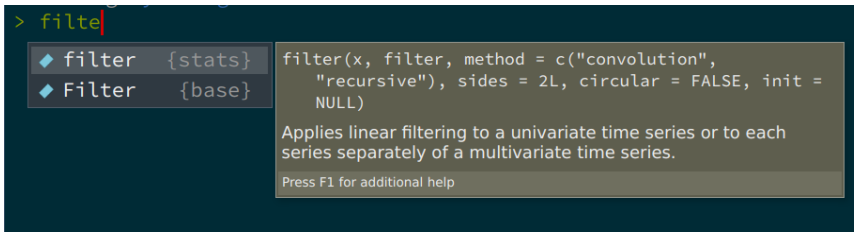
4.3.1 Functions dependencies

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 situtation 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-r
```

```
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:

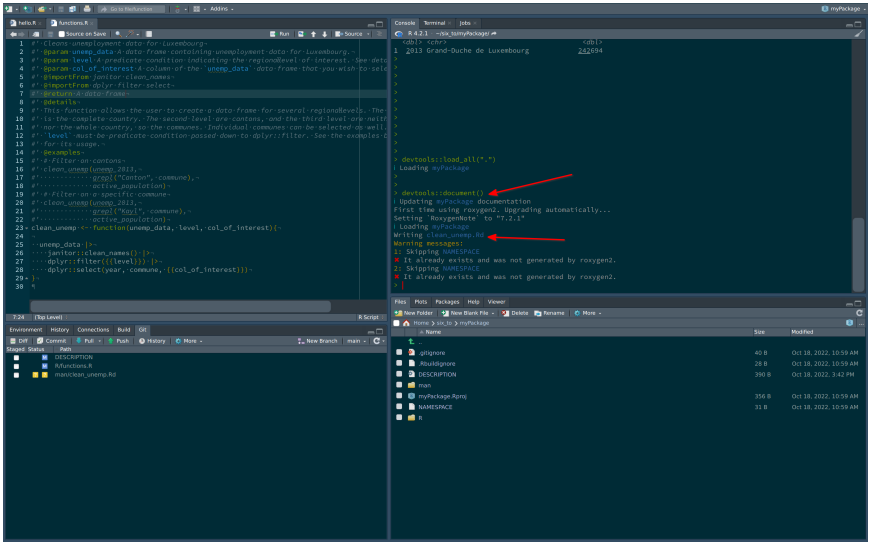
```
#' Easily filter unemployment data for
 ↪  Luxembourg
#' @param unemp_data A data frame containing
 ↪  unemployment data for Luxembourg.
#' @param level A predicate condition indicating
 ↪  the regional level of interest. See details
 ↪  for more.
#' @param col_of_interest A column of the
 ↪  `unemp_data` 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 im-

portant 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:



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:

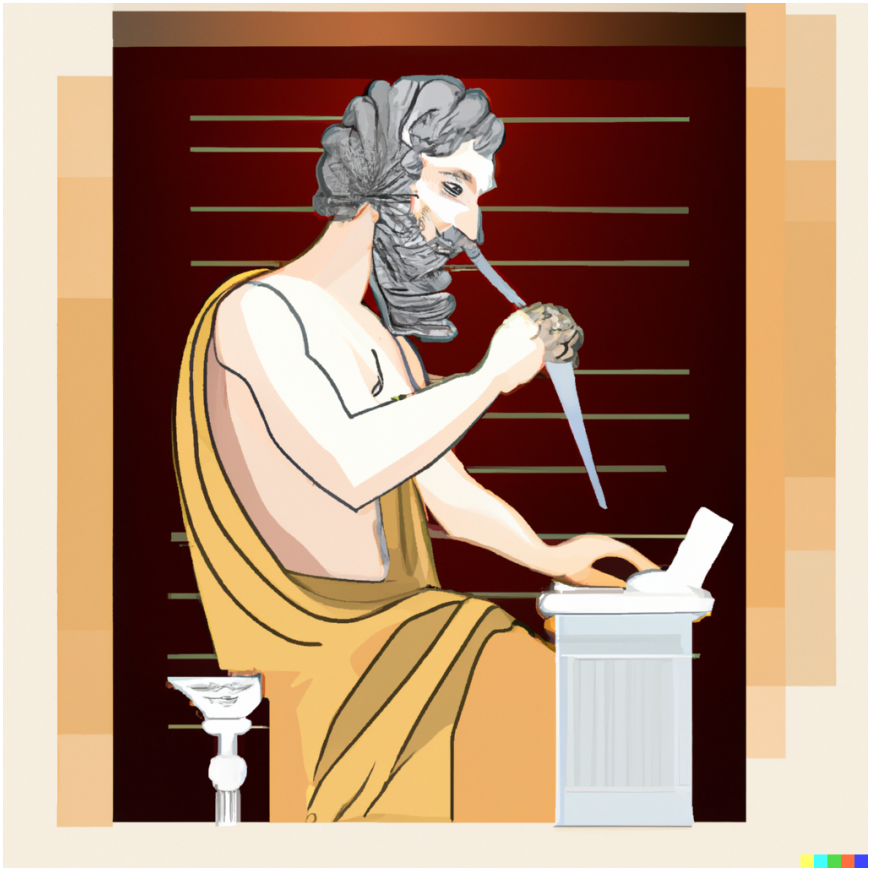
```
To install the package, please use the following
line

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.

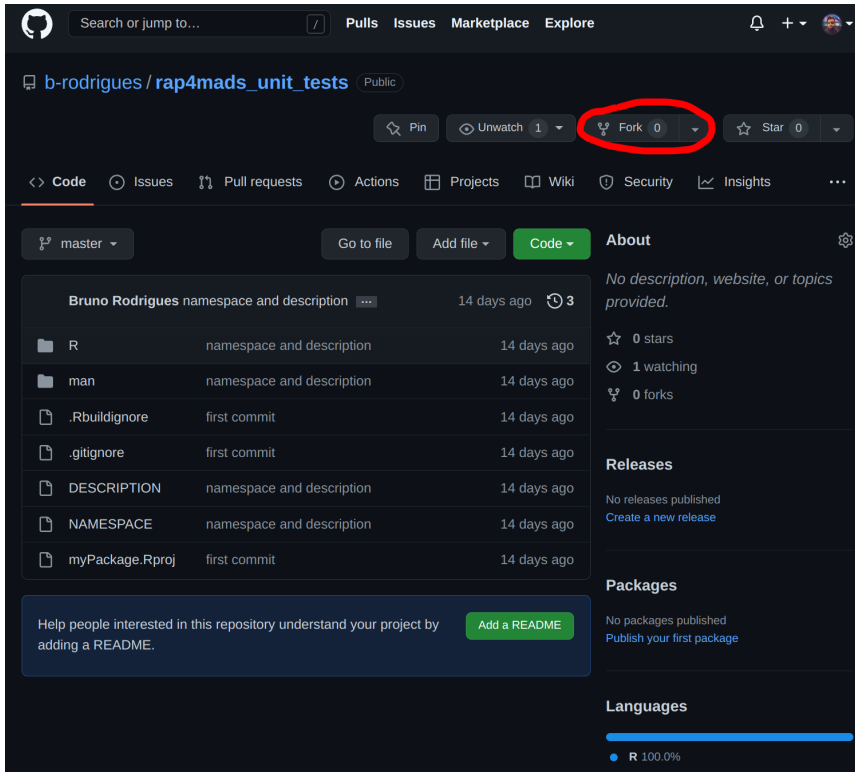
5.1 Introduction

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).

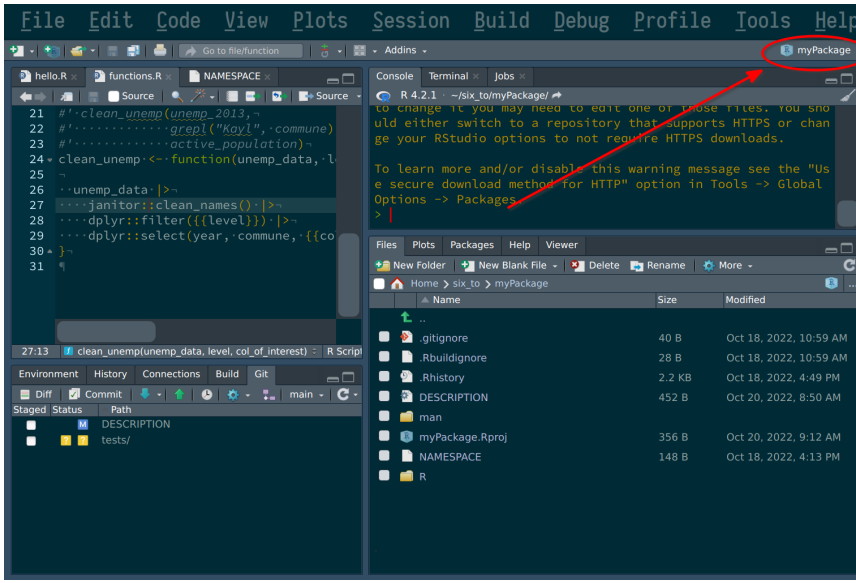
5.2 Testing your package

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.



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-r

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

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.