Building reproducible analytical pipelines with R

Bruno Rodrigues

1/6/23

Table of contents

Preface

In the summer of 2022, a former colleague from my first job asked me if I wanted to help him teach a class at the University of Luxembourg. It was a class for the Master's of Data Science, and the class was supposed to be taught by non-academics like us. The idea was to teach the students some "real-world" skills from the industry. It was a 40 hours class, and naturally we split them equally between us; my colleague focused on time series statistics but I really didn't know what I should do. I knew I wanted to teach, I always liked teaching, but I am a public servant in the ministry of higher education and research in Luxembourg. I still code a lot, but I don't do exciting machine learning anymore, or advanced econometrics like my colleague. Before (re)joining the public service I was a senior data scientist and then manager in one of the big four accounting firms. Before that, and this is where my colleague and I met, I was a research assistant in the research department of the national statistical institute of statistics in Luxembourg, and my colleague is still an applied researcher there.

What could I teach these students? What "skills from the industry" could I possibly share with them? I am an expert in nothing in particular. Actually, I don't really know anything very deeply, but know at least a little about many different things. There are many self-help books out there that state that it's better to know a lot about only a few, maybe even only one, topic, than know a lot about many topics. I tend to disagree with this; at least in my experience, knowing enough about many different topics always allowed me to communicate effectively with many different people, from researchers focusing on very specific topics that needed my help to assist them in their research, to clients from a wide range of industries that were sharing their problems with me in my consulting years. If I needed to deepen my knowledge on a particular topic before I could intervene, I had the necessary theoretical background to grab a few books and learn the material. Also, I was never afraid of asking questions.

This is reflected in my blogging. As I'm writing these lines (beginning of 2023), I have been blogging for about ten years. Most of my blog posts are me trying to lay out a problem I had at work and how I solved it. Sometimes I do some things for pleasure or curiosity, like the two posts on the video game nethack, or the ones on 19th century newspapers where I learned a lot about NLP. Because I was lucky enough to work with different people from many backgrounds, I always had to solve a very wide range of problems.

But that still didn't really help me to find a topic to teach… but then it dawned on me. Even though in my career I had to help many different people with many different backgrounds and needs, there were two things that everyone always required: traceability and reliability.

Everyone wanted to know how I came to the conclusions that I came to, and most of them even wanted to be able to reproduce my steps as a form of double checking what I did (consultants are expensive, so you better make sure that they're worth their hourly rate!). When I was a manager, I applied the same logic to my teammates. I wanted to be able to understand what they were doing, or at least know that if I needed to review their work deeply, the possibility was there.

So what I had to teach these students of data science was some best practices in software engineering. Most people working with data don't get taught software engineering skills. Courses focus on probability theory, linear algebra, algorithms, and programming but not software engineering. That's because software engineering skills get taught to software engineers. But while statisticians, data scientists, (or whatever we get called these days), are not software engineers, they do write a lot of code. And code that is quite important at that. And yet, most of us do work like pigs (no disrespect to pigs).

For example, how much of the code you write that produces very sensitive and important results, be it in science or in industry, is thoroughly tested? How much of the code you use relies on a single person showing up for work and using some secret knowledge that is undocumented? What if that person ends up under a bus? How much code do you run that no one dares touch anymore because that one person from before did end up under a bus?

How many people do you have to ping when you need to get an update to a quarterly report? How many people do you have to ping to know how Table 3 from that report from 2020 that was quickly put together during the Covid-19 lockdowns was computed? Are all the people involved even working in your company still?

When collaborating with teammates to write a report or scientific paper, do you consider potential risks? (If you're wondering *What risks?* then you're definitely not considering them.)

Are you able to tell anyone, *exactly*, how that number that gets used by the CEO in that one report was made? What if there's an investigation, or some external audit? Would the auditors be able to run the code and understand what is going on with as little intervention as possible (ideally none) from you? *But I don't work in an industry that gets audited*, you may think. Well, maybe not, or maybe one day your work will get audited anyways. Maybe it'll get audited internally for whatever reason. Maybe there's a new law that went into force that requires your work, or parts of your work, to be easily traceable.

And if you're a scientist, your work does get audited, or at least it should be in theory. I don't know any scientist (and I know more scientists than the average person, thanks to my background and current job) that is against the idea of open science, open data, reproducibility, and so on. Not one. But in practice, how many papers are truly reproducible? How many scientific results are auditable and traceable?

Lack of traceability and reproducibility can sometimes lead to serious consequences. If you're in the social sciences, you likely know about the *Reinhart and Rogoff* paper. Reinhard and Rogoff

are two American economists that published a paper in 2010 that showed that when countries are too much in debt (over 60% of GDP according to the authors) then annual growth decreases by two percent. These papers provided an empirical justification for austerity measures in the aftermath of the 2009 European debt crisis. But there was a serious problem with the Reinhard and Rogoff paper. It's not that they somehow didn't use the *correct* theoretical framework or modelling procedure in their paper. It's not that their assumptions were disputable or too unrealistic. It's that they performed their calculations inside an Excel spreadsheet and did not, and this is not a joke, they did not select every country's real GDP growth to compute the average real GDP growth for high-debt countries:

	B	C	I	J	K	L	M
2			Real GDP growth				
3			Debt/GDP				
4	Country	Coverage	30 or less	30 to 60	60 to 90	90 or above	30 or less
26			3.7	3.0	3.5	1.7	5.5
27	Minimum		1.6	0.3	1.3	-1.8	0.8
28	Maximum		5.4	4.9	10.2	3.6	13.3
29							
30	US	1946-2009	n.a.	3.4	3.3	-2.0	n.a.
31	UK	1946-2009	n.a.	2.4	2.5	2.4	n.a.
32	Sweden	1946-2009	3.6	2.9	2.7	n.a.	6.3
33	Spain	1946-2009	1.5	3.4	4.2	n.a.	9.9
34	Portugal	1952-2009	4.8	2.5	0.3	n.a.	7.9
35	New Zealand	1948-2009	2.5	2.9	3.9	-7.9	2.6
36	Netherlands	1956-2009	4.1	2.7	1.1	n.a.	6.4
37	Norway	1947-2009	3.4	5.1	n.a.	n.a.	5.4
38	Japan	1946-2009	7.0	4.0	1.0	0.7	7.0
39	Italy	1951-2009	5.4	2.1	1.8	1.0	5.6
40	Ireland	1948-2009	4.4	4.5	4.0	2.4	2.9
41	Greece	1970-2009	4.0	0.3	2.7	2.9	13.3
42	Germany	1946-2009	3.9	0.9	n.a.	n.a.	3.2
43	France	1949-2009	4.9	2.7	3.0	n.a.	5.2
44	Finland	1946-2009	3.8	2.4	5.5	n.a.	7.0
45	Denmark	1950-2009	3.5	1.7	2.4	n.a.	5.6
46	Canada	1951-2009	1.9	3.6	4.1	n.a.	2.2
47	Belgium	1947-2009	n.a.	4.2	3.1	2.6	n.a.
48	Austria	1948-2009	5.2	3.3	-3.8	n.a.	5.7
49	Australia	1951-2009	3.2	4.9	4.0	n.a.	5.9
50							
51			4.1	2.8	2.8	=AVERAGE(L30:L44)	

Figure 1: You can see that not all countries are selected…

And this is not the only problem with this paper.

The problem is not that this mistake was made. Reinhard and Rogoff are only human and mistakes can happen. What's problematic is that this was picked up and corrected too late. In an ideal world, Reinhard and Rogoff would not have used tools that make mistakes like this almost impossible to find once they're made. Instead, they would have used tools that would have made such a thing not happen in the first place, or, as a second best, making it easier

and faster for someone else to find this mistake. And this is not something that is only useful in research, but also in any industry. Being able to trust results, tracing back calculations and auditing are not only concerns of researchers.

So this is what I decided to teach the students: how they could structure their projects in such a way that they could spot problems like that during development, but also make it easy to reproduce and retrace who did what and when. I wrote my course notes into a freely available bookdown that I used for teaching. When I started compiling my notes, I discovered the concept *Reproducible Analytical Pipelines* as developed by the Office for National Statistics. I found the name "Reproducible Analytical Pipeline" really perfect for what I was aiming at. The ONS team for evangelising RAPs also published a free ebook in 2019 already. Another big source of inspiration is Software Carpentry to which I was exposed during my PhD years, around 2014-ish if memory serves. While working on a project with some German colleagues from the University of Bonn, the PI made us work using these concepts to manage the project. I was really impressed by it, and these ideas and techniques stayed with me since then.

The bottom line is: the ideas I'm presenting here are nothing new. It's just that I took some time to compile them and make them accessible and interesting (at least I hope so) for users of the R programming language.

At least my students found the course interesting. But not just students. I tweeted about this course and shared the notes with a wider audience, and this is when I got very positive feedback from people that were not my students. People wanted to buy this as a book and go deeper into the topics laid out. This is when I realised that, as far as I know, there is not a practical book available discussing these topics. So I decided to write one, but I took my time getting started. What finally, really, got me working on it was when Dmytro Perepolkin reached out to me and suggested I contact several persons to get their inputs and ideas and get started. I followed his advice, and this led to very fruitful discussions with Sébastien Rochette, Miles McBain and Dmytro. Their ideas and inputs definitely improved the quality of this book, so many thanks to them.

This book is divided into two parts. The first part teaches you what I believe is essential knowledge you should possess in order to write truly reproducible pipelines. This essential knowledge is constituted of:

- Version control with Git and how to manage projects with Github;
- Functional programming;
- Literate programming.

The main idea from part 1 is "don't repeat yourself". Git and Github will help us avoid losing code, and losing track of who should do what in a project (even if you're working alone on a project, you will see that using Git and Github will save you many hours and headaches). Getting familiar with functional and literate programming should improve the quality of our code by avoiding two common sources of mistakes: computing results that rely on the state

of our program (and later, the state of the whole hardware we are using) and copy and paste mistakes.

The second part of the book will then build upon this knowledge to introduce several tools that will help us go beyond the benefits of version control and functional and literate programming:

- Dependency management with `{renv}`;
- Build automation with `{targets}`;
- Reproducible environments with Docker;
- Continuous integration and delivery.

While this is not a book for beginners (you really should be familiar with R before reading this), I will not assume that you have any knowledge of the tools presented in part 2. In fact, even if you're already familiar with Git, Github, functional programming and literate programming, I think that you will still learn something useful from reading part 1.

I hope that you will enjoy reading this book and applying the ideas in your day-to-day, ideas which hopefully should improve the reliability, traceability and reproducibility of your code. You can read this book for free on https://raps-with-r.dev/ and will also be able to buy a physical copy, soon.

If you have feedback, drop me an email at bruno [at] brodrigues [dot] co.

Enjoy!

1 Introduction

This book will not teach you about machine learning, statistics or visualisation. The goal is to teach you a set of tools, practices and project management techniques that should make your projects easier to reproduce, replicate and retrace. These tools and techniques can be used right from the start of your project at a minimal cost, such that once you're done with the analysis, you're also done with making the project reproducible. Your projects are going to be reproducible simply because they were engineered, from the start, to be reproducible.

1.1 Who is this book for?

This book is for anyone that uses raw data to build any type of output based on that raw data. This can be a simple quarterly report for example, in which the data is used for tables and graphs, or a scientific article for a peer reviewed journal or even an interactive web application. It doesn't matter, because the process is, at its core, always very similar:

- Get the data;
- Clean the data;
- Write code to analyse the data;
- Put the results into the final product.

This book will already assume some familiarity with programming, and in particular the R programming language. However, if you're comfortable with another programming language like Python, you could still learn a lot from reading this book. Some tools presented in this book are specific to R, but there will always be an alternative for the language you prefer using, meaning that you could apply the advice from this book to your needs and preferences.

1.2 What is the aim of this book?

The aim of this book is to make the process of analysing data as reliable, retraceable, and reproducible as possible, and do this by design. This means that once we're done with the analysis, we're done. We don't want to spend time, nor have the time, to rewrite or refactor an analysis and make it reproducible after the fact. We all know that this is not going to happen. Once an analysis is done, it's time to go to the next analysis. And if we need to rerun an older analysis (for example, because the data got updated), then we'll simply figure it out at

that point. Hopefully, we will remember every quirk of our code and know which script to run at which point in the process, which comments are outdated and can be safely ignored, what features of the data need to be checked (and when they need to be checked), and so on…

Going forward, we're going to refer to the process above as a "reproducible analytical pipeline", or RAP for short. There are only two ways to make such a RAP reproducible; either we are lucky enough to have someone on the team whose job is to do this, or we do it ourselves. And this second option is very likely the most common. The issue is, as stated above, that we simply don't do it. We are always in the rush to get to the results, and don't think about making the process reproducible. This is because we always think that making the process reproducible takes time and this time is better employed to perform the analysis itself. But this is a misconception, for two reasons.

The first reason is that employing the techniques that we are going to discuss in this book, won't actually take much time. As you will see, they're not really things that you "add on top of the analysis" that take time, but will be part of the analysis and project management themselves. And some of these techniques will even save you time (especially testing) and headaches.

The second reason is that an analysis is never, ever, a one-shot. Only the most simple things, like pulling out a number from some data base may be a one-shot. And even then, chances are that once you provide that number, you'll be asked to pull out a variation of that number (for example, by disaggregating by one or several variables). Or maybe you'll get asked for an update to that number in six months. So you will learn very quickly to keep that SQL query in a script somewhere to make sure that you provide a number that is consistent. But what about more complex analyses? Is keeping the script enough? Well that's already a good start. The problem is that very often, there is no script, or not a script for each step of the analysis.

I've seen this play out many times in many different organisations. It's that time of the year again, we have to write a report. 10 people are involved, and just gathering the data is already complicated. Some get their data from Word documents attached to emails, some from a website, some from a report from another department that is a PDF… I remember a story that a senior manager at my previous job used to tell us: once, a client put out a call for a project that involved helping them setting up a PDF scraper. They periodically needed data from another department that came in PDFs. The manager asked what was, at least from our perspective, an obvious question: why can't they send you the underlying data from that PDF in a machine readable format? They had never thought to ask. So my manager went to that department, and talked to the people putting that PDF together. Their answer? "Well, we could send them the data in any format they want, but they've asked us to send the tables in a PDF format".

So the first, and probably most important lesson here is: when starting to build a RAP, make sure that you talk with all the people involved.

1.3 Prerequisites

You should be comfortable with the R programming language. This book will assume that you have been using R for some projects already, and want to improve not only your knowledge of the language itself, but also how to successfully manage complex projects. Ideally, you should know about packages, how to install them, you should have written some functions already, know about loops and have some basic knowledge of data structures like lists. While this is not a book on visualisation, we will be making some graphs using the `{ggplot2}` package, so if you're familiar with that, that's good. If not, no worries, visualisation, data munging or data analysis is not the point of this book.

Ideally, you should also not be afraid of not using Graphical User Interfaces (GUI). While you can follow along using an IDE like RStudio, we will not be teaching any features from any program with a GUI. This is not to gatekeep, but because interacting graphically with a program is simply not reproducible. This is the second lesson of building RAPs: there should be no human intervention needed to get the outputs once the RAP is started. So our target is to write code that can be executed non-interactively by a machine. This is because one necessary condition for a workflow to be reproducible and get referred to as a RAP, is for the workflow to be able to be executed by a machine, automatically, without any human intervention. If this is the case, then your workflow is likely reproducible, or can at least be made reproducible much more easily than if it requires some special manipulation by a human somewhere in the loop.

1.4 What is reproducibility?

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

1.4.1 Using open-source tools to build a RAP is a hard requirement

Open source is a hard requirement for reproducibility.

No ifs nor buts. And I'm not only talking about the code you typed for your research paper/report/analysis. I'm talking about the whole ecosystem that you used to type your code and build the workflow.

Is your code open? That's good. Or is it at least available to other people from your organisation, in a way that they could re-execute it if needed? Good.

But is it code written in a proprietary program, like STATA, SAS or MATLAB? Then your project is not reproducible. It doesn't matter if this code is well documented and written and

available on a version control system (internally to your company or open to the public). This project is not reproducible. Why?

Because there is no way to re-execute your code with the exact same version of this proprietary program down the line. As I'm writing these lines, MATLAB, for example, is at version R2022b. And buying an older version is not guaranteed. I'm sure if you contact their sales department they might be able to sell you an older version. Maybe you can even simply redownload older versions that you've already bought. But maybe it's not that simple. Or maybe they won't offer this option anymore in the future, who knows? In any case, if you google "purchase old version of Matlab" you will see that many researchers and engineers have this need.

Figure 1.1: Wanting to run older versions of analytics software is a recurrent need.

And if you're running old code written for version, say, R2008a, there's no guarantee that it will produce the exact same results on version 2022b. And let's not even mention the toolboxes (if

you're not familiar with MATLAB's toolboxes, they're the equivalent of packages or libraries in other programming languages). These evolve as well, and there's no guarantee that you can purchase older versions of said toolboxes. And it's likely that newer versions of toolboxes cannot even run on older versions of Matlab.

And let me be clear, what I'm describing here with MATLAB could also be said for any other proprietary programs still commonly (unfortunately) used in research and in statistics (like STATA or SAS). And even if some, or even all, of the editors of these proprietary tools provide ways to buy and run older versions of their software, my point is that the fact that you have to rely on them for this is a barrier to reproducibility, and there is no guarantee they will provide the option to purchase older versions forever. Also, who guarantees that they will be around forever? Or, and that's more likely, that they will keep offering a program that you install on your machine instead of shifting to a subscription based model?

For just $199 a month, you can execute your SAS/STATA/MATLAB scripts on the cloud! Worry about data confidentiality? No worries, data gets encrypted and stored safely on our secure servers! Run your analysis from anywhere and don't worry about losing your work if your cat knocks over your coffee on your laptop! And if you purchase the pro licence, for an additional $100 a month, you can even execute your code in parallel!

Think this is science fiction? Google "SAS cloud" to see SAS's cloud based offering.

1.4.2 There are hidden dependencies that can hinder the reproducibility of a project

Then there's another problem: let's suppose you've written a nice, thoroughly tested and documented workflow, and made it available on Github (and let's even assume that the data is available for people to freely download, and that the paper is open access). Or, if you're working in the private sector, you did everything above as well, the only difference being that the workflow is only available to people inside the company instead of being available freely and publicly online.

Let's further assume that you've used R or Python, or any other open source programming language. Could this study/analysis be said to be reproducible? Well, if the analysis ran on a proprietary operating system, then the conclusion is: your project is not reproducible.

This is because the operating system the code runs on can also influence the reproducibility of the project. There are some particularities in operating systems that may make certain things work differently. Admittedly, this is in practice rarely a problem, but it does happen, especially if you're working with very high precision floating point arithmetic like you would do in the financial sector.

Thankfully, there is no need to change operating systems to deal with this issue, and we will learn how to use Docker to safeguard against this problem.

1.4.3 The requirements of a RAP

So where does that leave us? Basically, for something to be truly reproducible, it has to respect the following bullet points:

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

Figure 1.2: A real sentence from a real paper published in *THE LANCET Regional Health*. How about *make the data available and I won't scratch your car*, how's that for a reasonable request?

1.5 Are there different types of reproducibility?

Let's take one step back: we live in the real world, and in the real world, there are some constraints that are outside of our control. These constraints can make it impossible to build a true RAP, so sometimes we need to settle for something that might not be a true RAP, but a second or even third best thing.

In what follows, let's assume this: in the discussion below, code is tested and documented, so let's only discuss the code running the pipeline itself.

The *worst* reproducible pipeline would be something that works, but only on your machine. This can be simply due to the fact that you hardcoded paths that only exist on your laptop. Anyone wanting to rerun the pipeline would need to change the paths. This is something that

needs to be documented in a README which we assumed was the case, so there's that. But maybe this pipeline only runs on your laptop because the computational environment that you're using is hard to reproduce. Maybe you use software, even if it's open source software, that is not easy to install (anyone that tried to install R packages on Linux that depend on the `{rJava}` package know what I'm talking about).

So a least worse pipeline would be one that could be run more easily on any similar machine as yours. This could be achieved by not using hardcoded absolute paths, and by providing instructions to set up the environment. For example, in the case of R, this could be as simple as providing a script called something like `install_deps.R` that would be a call to `install.packages()`. It could look like this:

```
install.packages(c("package1",
                   "package2",
                   etc))
```

The issue here is that you need to make sure that the right versions of the packages get installed. If your script uses `{ggplot2}` version 2.2.1, then users should install this version as well, and by running the script above, the latest version of `{ggplot2}` (as of writing, version 3.4.0) will get installed. Maybe that's not a problem, but it can be if your script uses a function from version 2.2.1 that is not available anymore in the latest version (or maybe its name got changed, or maybe it was modified somehow and doesn't provide the exact same result). And the more packages the script uses (and the older it is), the higher the likelihood that some package version will not be compatible. There is also the issue of the R version itself. Generally speaking, recent versions of R seem to not be too bad when it comes to running older code written in R. I know this because in 2022 I've ran every example that comes bundled with R since version 0.6.0 on the then current version of R, version 4.2.2. Here is the result of this experiment:

This graph shows the following: for each version of R, starting with R version 0.6.0 (released in 1997), how well the examples that came with a standard installation of R run on the current version of R (version 4.2.2 as of writing). These are the examples from the default packages like `{base}`, `{stats}`, `{stats4}`, and so on. Turns out that more than 75% of the example code from version 0.6.0 still work on the current version of R. A small fraction output a message (which doesn't mean the code doesn't work), some 5% raise a warning, which again doesn't necessarily mean that the code doesn't work, and finally around 20% or so errors. As you can see, the closer we get to the current release, the less errors get raised (if you want to run the code for yourself, check out this Github repository).

(But something important should be noted: just because some old piece of code runs without error, doesn't mean that the result is exactly the same. There might be cases where the same function returns different results on different versions of R.)

But while this is evidence of R itself being quite stable through time, there are studies that

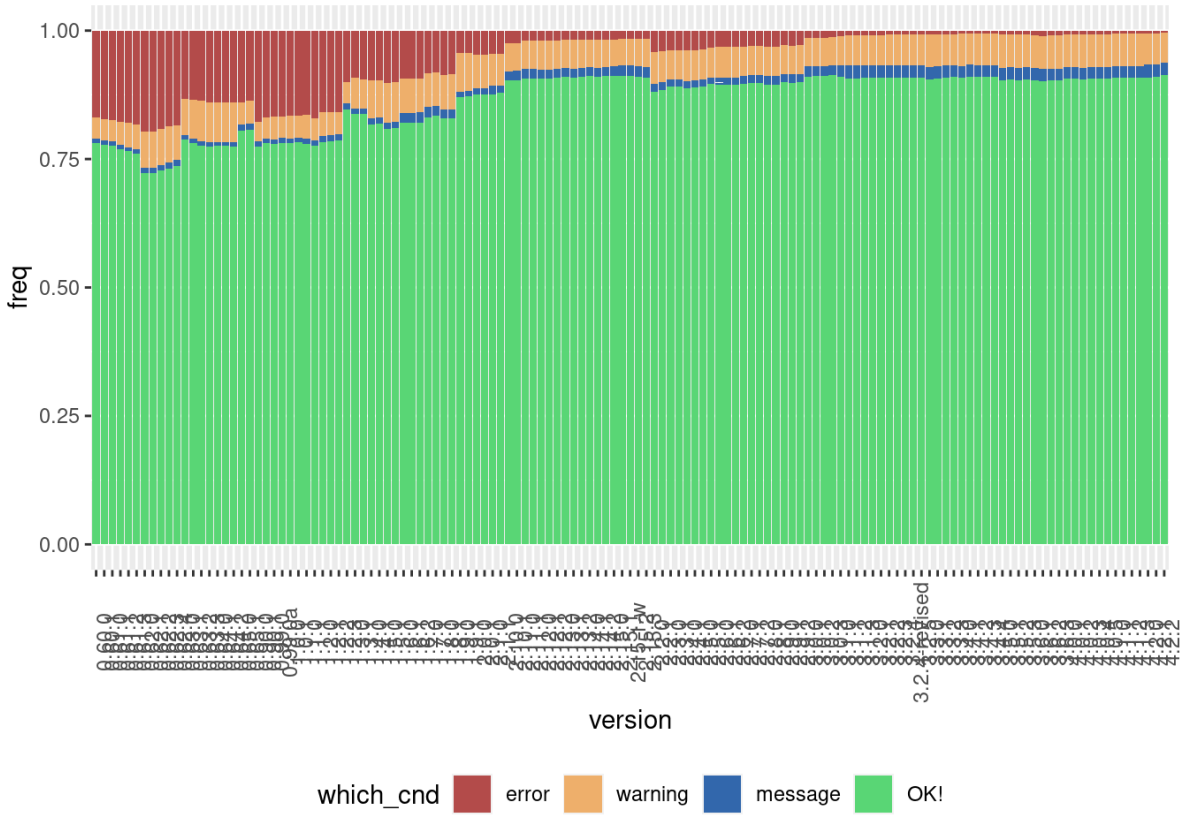


Figure 1.3: Examples from older versions of R run most of the time successfully on the current version of R

show a less rosy picture. In a recent study (Trisovic et al. (2022) [1]), some researchers tried to rerun up to 9000 R scripts downloaded from the Harvard Dataverse. There were several issues when trying to rerun the scripts, which lead to, and I quote the paper here, "[...] 74% of R files [failing] to complete without error in the initial execution, while 56% failed when code cleaning was applied, showing that many errors can be prevented with good coding practices".

The take-away message is that counting on the language itself being stable through time as a sufficient condition for reproducibility is not enough. We have to set up the code in a way that it actually is reproducible.

So what does this all mean? This means that reproducibility is on a continuum, and depending on the constraints you face your project can not very reproducible to totally reproducible. Let's consider the following list of anything that can influence how reproducible your project truly is:

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

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

This is not a novel, or new idea. Peng (2011) already discussed this concept but named in the reproducibility spectrum. In part 2 of this book, I will reintroduce the idea and call it the "reproducibility iceberg".

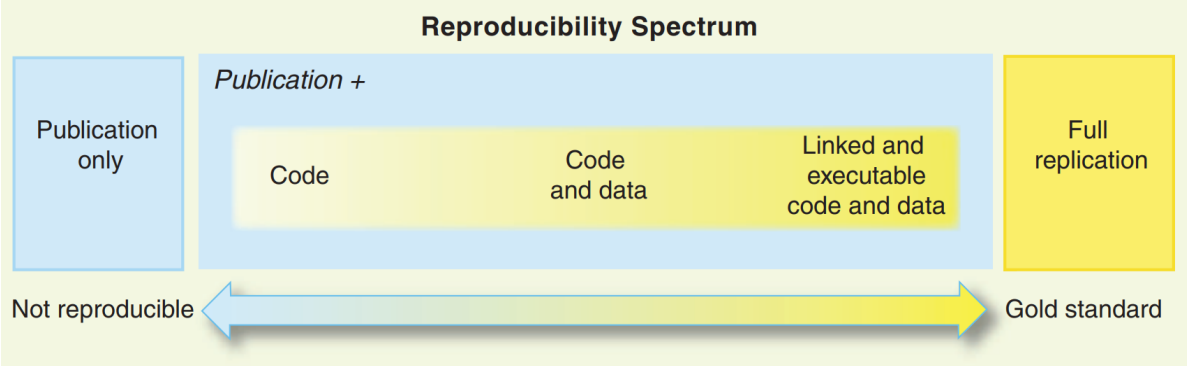


Figure 1.4: The reproducibility spectrum from Peng's 2011 paper.

[1] https://www.nature.com/articles/s41597-022-01143-6

Let me just finish this introduction by the last item on the previous list: hardware architecture by discussing a fairly recent event in computing. You see, Apple has changed hardware architecture recently, their new computers switched from Intel based hardware to their own proprietary architecture (Apple Silicon) based on the ARM specification. And what does that mean concretely? It means that all the binary packages that were built for Intel based Apple computers cannot work on their new computers. Which means that if you have a recent M1 Macbook and need to install old CRAN packages to rerun a project (and we will learn how to do this later in the book), these need to be compiled to work on M1. You cannot even install older versions of R, unless you also compile those from source! This is because these older versions of R and packages were compiled to run on the previous architecture that Apple used for their computers, and cannot be run on the current architecture. Now I have read about a compatibility layer called Rosetta which enables to run binaries compiled for the Intel architecture on the ARM architecture, and maybe this works well with R and CRAN binaries compiled for Intel architecture. Maybe, I don't know. But my point is that you never know what might come in the future, and thus needing to be able to compile from source is important, because compiling from source is what requires the least amount of dependencies that are outside of your control. Relying on binaries is not future-proof (and which is again, another reason why open-source tools are a hard requirement for reproducibility).

And for you Windows users, don't think that the preceding paragraph does not concern you. I think that it is very likely that Microsoft will push in the future for OEM manufacturers to develop more ARM based computers. There is already an ARM version of Windows after all, and it has been around for quite some time, and I think that Microsoft will not kill that version any time in the future. This is because ARM is much more energy efficient than other architectures, and any manufacturer can build its own ARM cpus by purchasing a license, which can be quite interesting. For example in the case of Apple Silicon cpus, Apple can now get exactly the cpus they want for their machines and make their software work seamlessly with it. I doubt that others will pass the chance to do the same.

Also, something else that might happen is that we might move towards more and more cloud based computing, but I think that this scenario is less likely than the one from before. But who knows. And in that case it is quite likely that the actual code will be running on Linux servers that will likely be ARM based because of energy costs. Here again, if you want to run your historical code, you'll have to compile old packages and R versions from source.

Ok, so this might seem all incredibly complicated. How on earth are we supposed to manage all these risks and balance the immediate need for results with the future need of rerunning an old project? And what if rerunning this old project is not even needed in the future?

This is where this book will help you. By employing the techniques discussed in this book, not only will it be very easy and quick to set up a project from the ground up that is truly reproducible, the very fact of building the project this way will also ensure that you avoid mistakes and producing results that are wrong. It will be easier and faster to iterate and improve your code, to collaborate, and ultimately to trust the results of your pipelines. So

even if no one will rerun that code ever again, you will still benefit from the best practices presented in this book. Let's dive in!

Part I

Part 1: Don't repeat yourself

Introduction

Part 1 will focus on teaching you the fundamental ingredients to reproducibility. By fundamental ingredients I mean those tools that you absolutely need to have in your toolbox before even attempting to make a project reproducible. These tools are so important, that a good chunk of this book is dedicated to them:

- Version control;
- Functional programming;
- Literate programming.

You might already be familiar with these topics, and maybe already use them in your day to day. If that's the case, you still might want to at least skim part 1 before tackling part 2 of the book, which will focus on another set of tools to actually build reproducible analytical pipelines (RAPs).

So this means that part 1 will not teach you how to build reproducible pipelines. But I cannot immediately start building reproducible analytical pipelines without first making sure that you understand the core concepts laid out above. To help us understand these concepts, we will start by analysing some data together. We ar going to download, clean and plot some data, and we will achieve this by writing two scripts. These scripts will be written in a very "typical non software engineery" way, as to mimic how analysts, data scientists or researchers without any formal training in computer science would perform such an analysis. This does not mean that the quality of the analysis will be low. But it means that, typically, these programmers have delievering results fast, and by any means necessary, as the top priority. Our goal with this book is to show you, and hopefully convince you, that by adopting certain simple ideas from software engineering it is possible to deliver just as fast as before, but in a more consistent and robust way.

Let's get started!

2 Project start

In this chapter, we are going to work together on a very simple project. This project will stay with us until the end of the book. For now, we are going to keep it simple; our goal here is to get an analysis done. We are going to download some data, and analyse it. After we're done with our analysis, we are going to keep it on the side for some time: we will then learn about tools and new programming paradigms and then rewrite our analysis at the start of the second part of this book using the techniques we've learned from part 1.

But for now, our main concern is to get our work done.

2.1 Housing in Luxembourg

We are going to download data about house prices in Luxembourg. Luxembourg is a little Western European country the author hails from that looks like a shoe and is about the size of .98 Rhode Islands. Did you know that Luxembourg is a constitutional monarchy, and not a kingdom like Belgium, but a Grand-Duchy, and actually the last Grand-Duchy in the World? Also, what you should know to understand what we will be doing is that the country of Luxembourg is divided into Cantons, and each Cantons into Communes. If Luxembourg was the USA, Cantons would be States and Communes would be Counties (or Parishes or Boroughs). What's confusing is that "Luxembourg" is also the name of a Canton, and of a Commune, which also has the status of a city and is the capital of the country. So Luxembourg the country, is divided into Cantons, one of which is called Luxembourg as well, cantons are divided into communes, and inside the canton of Luxembourg there's the commune of Luxembourg which is also the city of Luxembourg, sometimes called Luxembourg-City, which is the capital of the country.

What you should also know is that the population is about 645.000 as of writing (January 2023), half of which are foreigners. Around 400.000 persons work in Luxembourg, of which half do not live in Luxembourg; so every morning from Monday to Friday, 200.000 people enter the country to work, and leave in the evening to go back to either Belgium, France or Germany, the neighbouring countries. As you can imagine, this puts enormous pressure on the transportation system and on the roads, but also on the housing market; everyone wants to live in Luxembourg to avoid the horrible daily commute, and everyone wants to live either in the capital city, or in the second largest urban area in the south, in a city called Esch-sur-Alzette.

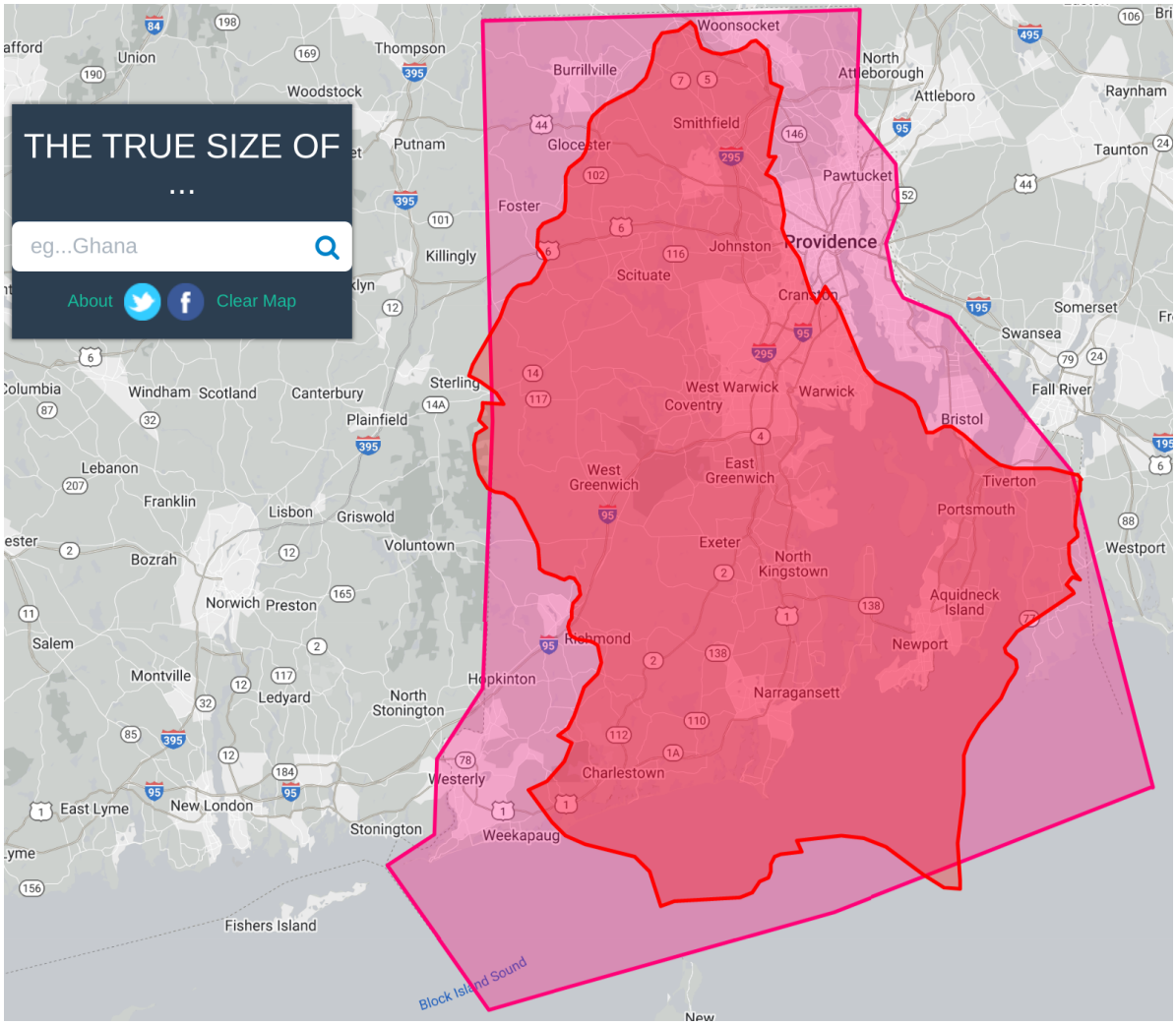
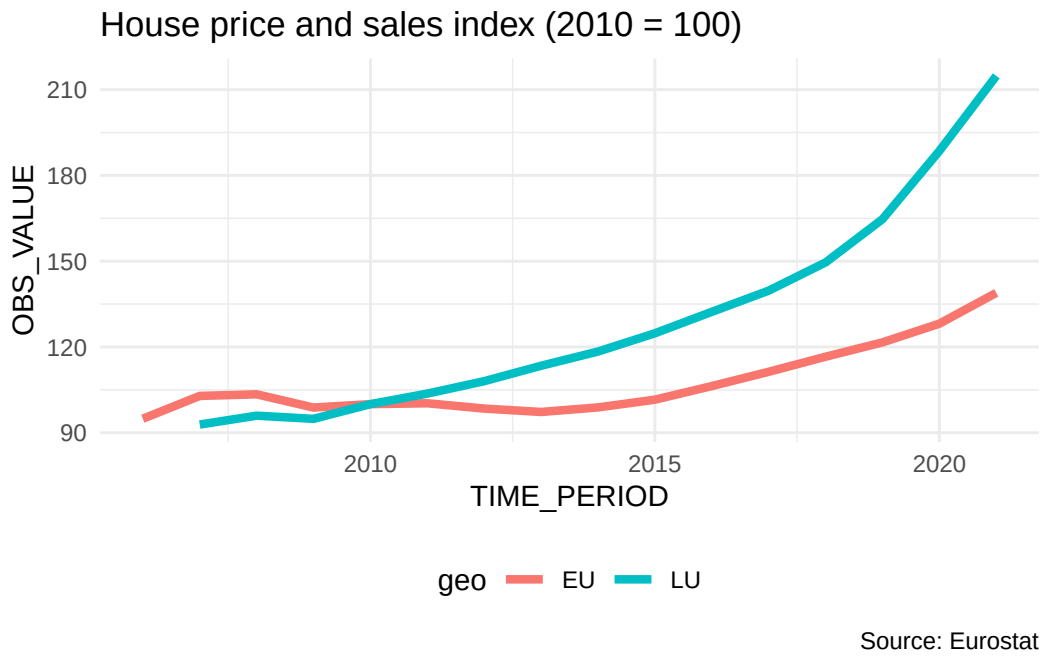


Figure 2.1: Luxembourg is about as big as the US State of Rhode Island

The plot below shows the value of the House Price Index through time for Luxembourg and the European Union:



If you want to download the data, click here.

Let us paste the definition of the HPI in here (taken from the HPI's metadata page):

The House Price Index (HPI) measures inflation in the residential property market. The HPI captures price changes of all types of dwellings purchased by households (flats, detached houses, terraced houses, etc.). Only transacted dwellings are considered, self-build dwellings are excluded. The land component of the dwelling is included.

So from the plot, we can see that the price of dwellings more than doubled between 2010 and 2021; the value of the index is 214.81 in 2021 for Luxembourg, and 138.92 for the European Union as a whole.

There is a lot of heterogeneity though; the capital and the communes immediately next to the capital are much more expensive than communes from the less densely populated north, for example. The south of the country is also more expensive than the north, but not as much as the capital and surrounding communes. Not only is price driven by demand, but also by scarcity; in 2021, .5% of residents owned 50% of the buildable land for housing purposes (Source: *Observatoire de l'Habitat, Note 29*, archived download link).

Our project will be quite simple; we are going to download some data, supplied as an Excel file, compiled by the Housing Observatory (*Observatoire de l'Habitat*, a service from the Ministry of Housing, which monitors the evolution of prices in the housing market, among other useful

services like the identification of vacant lots). The advantage of their data when compared to Eurostat's data is that the data is disaggregated by commune. The disadvantage is that they only supply nominal prices, and no index. Nominal prices are the prices that you read on price tags in shops. The problem with nominal prices is that it is difficult to compare them through time. Ask yourself the following question: would you prefer to have had 500€ (or USDs) in 2003 or in 2023? You probably would have preferred them in 2003, as you could purchase a lot more with $500 then than now. In fact, according to a random inflation calculator I googled, to match the purchasing power of $500 in 2003, you'd need to have $793 in 2023 (and I'd say that we find very similar values for €). But it doesn't really matter if that calculation is 100% correct: what matters is that the value of money changes, and comparisons through time are difficult, hence why an index is quite useful. So we are going to convert these nominal prices to real prices. Real prices take inflation into account and so allow us to compare prices through time. So we will need to also get some data to achieve this.

So to summarise; our goal is to:

- Get data trapped inside an Excel file into a neat data frame;
- Convert nominal to real prices using a simple method;
- Make some tables and plots and call it a day (for now).

We are going to start in the most basic way possible; we are simply going to write a script and deal with each step separately.

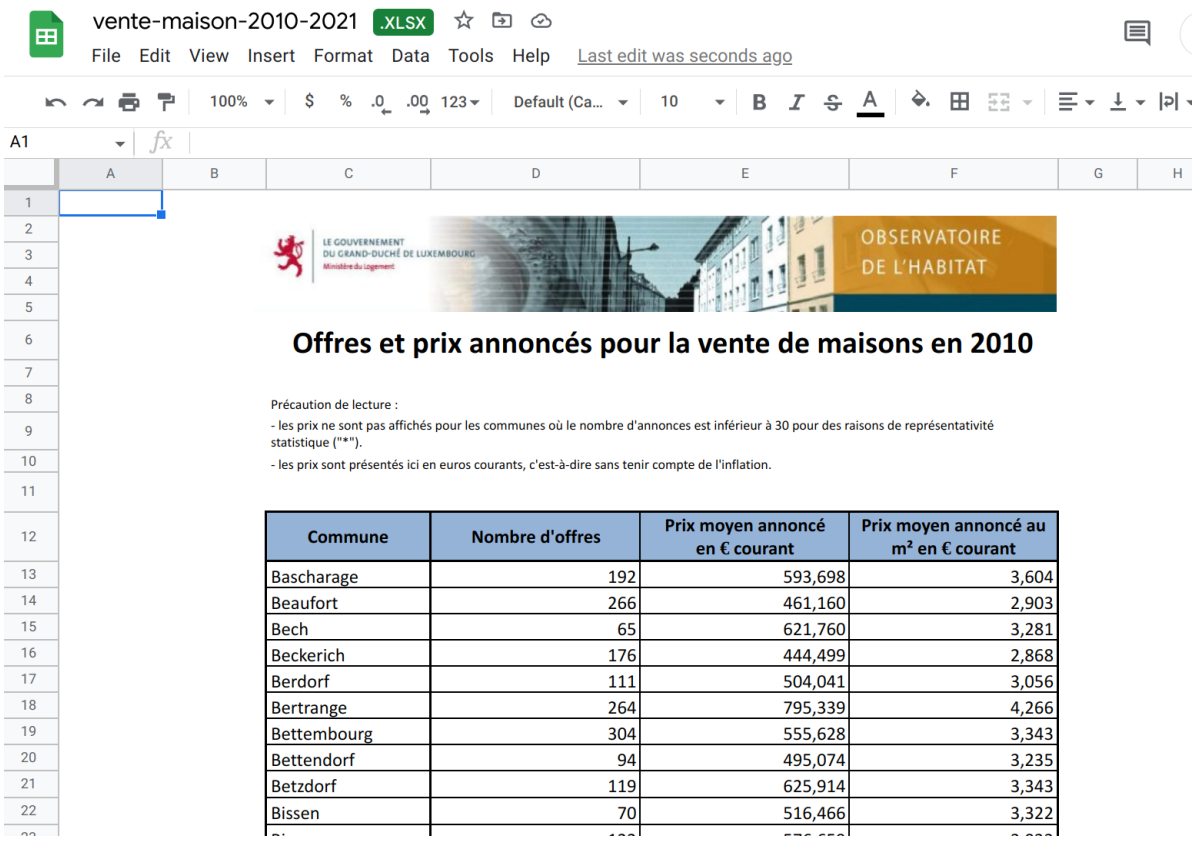
2.2 Saving trapped data from Excel

Getting data from Excel into a tidy data frame can be very tricky. This is because very often, Excel is used as some kind of dashboard, or presentation tool. So data is made human-readable, in contrast to machine readable. Let us quickly discuss this topic as it is essential to grasp the difference between the two (and in our experience, a lot of collective pain inflicted to statisticians and researchers could have been avoided if this concept was more well-known). The picture below shows an Excel made for human consumption:

So why is this file not machine-readable? Here are some issues:

- The table does not start in the top-left corner of the spreadsheet, which is where most importing tools expect it to be;
- The spreadsheet starts with a header that contains an image and some text;
- Numbers are text and use "," as the thousands separator;
- You don't see it in the screenshot, but each year is in a separate sheet.

That being said, this Excel file is still very tame, and going from this Excel to a tidy data frame will not be too difficult. In fact, we suspect that whoever made this Excel file is well aware of the contradicting requirements of human and machine readable formatting of data,



Commune	Nombre d'offres	Prix moyen annoncé en € courant	Prix moyen annoncé au m² en € courant
Bascharage	192	593,698	3,604
Beaufort	266	461,160	2,903
Bech	65	621,760	3,281
Beckerich	176	444,499	2,868
Berdorf	111	504,041	3,056
Bertrange	264	795,339	4,266
Bettembourg	304	555,628	3,343
Bettendorf	94	495,074	3,235
Betzdorf	119	625,914	3,343
Bissen	70	516,466	3,322

Figure 2.2: An Excel file meant for human eyes

and strove to find a compromise. Because more often than not, getting human readable data into a machine readable formatting is a nightmare.

This is actually the file that we are going to use for our project, so if you want to follow along, you can download it here (downloaded on January 2023 from the luxembourguish open data portal). But you don't need to follow along with code, because I will link the scripts for you to download later.

Each sheet contains a dataset with the following columns:

- Commune: the commune
- Nombre d'offres: the total number of selling offers
- Prix moyen annoncé en Euros courants: Average selling price in nominal Euros
- Prix moyen annoncé au m2 en Euros courants: Average selling price in square meters in nominal Euros

For ease of presentation, we are going to show you each function here separately, but we'll be putting everything together in a single script once we're done explaining each step. So first, let's read in the data. The following lines do just that:

```
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
```

```
library(purrr)
library(readxl)
library(stringr)
library(janitor)
```

```
Attaching package: 'janitor'
```

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

    chisq.test, fisher.test
```

The code below downloads the data, and puts it in a data frame:

```
url <- "https://github.com/b-rodrigues/rap4all/raw/master/datasets/vente-maison-2010-2021.

raw_data <- tempfile(fileext = ".xlsx")

download.file(url, raw_data)

sheets <- excel_sheets(raw_data)

read_clean <- function(..., sheet){
  read_excel(..., sheet = sheet) |>
    mutate(year = sheet)
}

raw_data <- map(
  sheets,
  ~read_clean(raw_data,
              skip = 10,
              sheet = .)
                   ) |>
  bind_rows() |>
  clean_names()
```

```
New names:
* `*` -> `*...3`
* `*` -> `*...4`
```

```
raw_data <- raw_data |>
  rename(
    locality = commune,
    n_offers = nombre_doffres,
    average_price_nominal_euros = prix_moyen_annonce_en_courant,
    average_price_m2_nominal_euros = prix_moyen_annonce_au_m2_en_courant,
    average_price_m2_nominal_euros = prix_moyen_annonce_au_m2_en_courant
  ) |>
  mutate(locality = str_trim(locality)) |>
```

```
    select(year, locality, n_offers, starts_with("average"))
```

If you are familiar with the `{tidyverse}` the above code should be quite easy to follow. We start by downloading the raw Excel file and save the sheet names into a variable. We then use a function called `read_clean()`, which takes the path to the Excel file and the sheet names as an argument to read the required sheet into a data frame. We use `skip = 10` to skip the first 10 lines in each Excel sheet because the first 10 lines contain a header. The last thing this function does is add a new column called `year` which contains the year of the data. We're lucky, because the sheet names are the years: "2010", "2011" and so on. We then map this function to the list of sheet names, thus reading in all the data from all the sheets into one list of data frames. We then use `bind_rows()`, to bind each data frame into a single data frame, by row. Finally, we rename the columns (by translating their names from French to English) and only select the required columns. If you don't understand each step of what is going on, don't worry too much about it; this book is not about learning how to use R.

Running this code results in a neat data set:

```
str(raw_data)
```

```
tibble [1,343 x 5] (S3: tbl_df/tbl/data.frame)
 $ year                          : chr [1:1343] "2010" "2010" "2010" "2010" ...
 $ locality                      : chr [1:1343] "Bascharage" "Beaufort" "Bech" "Beckerich" .
 $ n_offers                      : num [1:1343] 192 266 65 176 111 264 304 94 119 70 ...
 $ average_price_nominal_euros   : chr [1:1343] "593698.31000000006" "461160.29" "621760.22"
 $ average_price_m2_nominal_euros: chr [1:1343] "3603.57" "2902.76" "3280.51" "2867.88" ...
```

But there's a problem: columns that should be of type numeric are of type character instead (`average_price_nominal_euros` and `average_price_m2_nominal_euros`). There's also another issue, which you would eventually catch as you would be exploring the data: naming of the communes is not consistent. Let's take a look:

```
raw_data |>
  filter(grepl("Luxembourg", locality)) |>
  count(locality)
```

```
# A tibble: 2 x 2
  locality             n
  <chr>            <int>
1 Luxembourg           9
2 Luxembourg-Ville     2
```

We can see that the city of Luxembourg is spelled in two different ways. It's the same with another commune, Pétange:

```
raw_data |>
  filter(grepl("P.tange", locality)) |>
  count(locality)
```

```
# A tibble: 2 x 2
  locality     n
  <chr>    <int>
1 Petange      9
2 Pétange      2
```

So sometimes it is spelled correctly, with an "é", sometimes not. Let's write some code to correct this:

```
raw_data <- raw_data |>
  mutate(locality = ifelse(grepl("Luxembourg-Ville", locality),
                           "Luxembourg",
                           locality),
         locality = ifelse(grepl("P.tange", locality),
                           "Pétange",
                           locality)
         ) |>
  mutate(across(starts_with("average"), as.numeric))
```

```
Warning in mask$eval_all_mutate(quo): NAs introduced by coercion

Warning in mask$eval_all_mutate(quo): NAs introduced by coercion
```

Now this is interesting – converting the `average` columns to numeric resulted in some `NA` values. Let's see what happened:

```
raw_data |>
  filter(is.na(average_price_nominal_euros))
```

```
# A tibble: 290 x 5
   year  locality                                          n_off~1 avera~2 avera~3
   <chr> <chr>                                               <dbl>   <dbl>   <dbl>
 1 2010  Consthum                                              29      NA      NA
```

```
 2 2010  Esch-sur-Sûre                                          7      NA      NA
 3 2010  Heiderscheid                                          29      NA      NA
 4 2010  Hoscheid                                              26      NA      NA
 5 2010  Saeul                                                 14      NA      NA
 6 2010  <NA>                                                  NA      NA      NA
 7 2010  <NA>                                                  NA      NA      NA
 8 2010  Total d'offres                                     19278      NA      NA
 9 2010  <NA>                                                  NA      NA      NA
10 2010  Source : Ministère du Logement - Observatoire ~       NA      NA      NA
# ... with 280 more rows, and abbreviated variable names 1: n_offers,
#   2: average_price_nominal_euros, 3: average_price_m2_nominal_euros
```

It turns out that there are no prices for certain communes, but that we also have some rows with garbage in there. Let's go back to the raw data to see what this is about:

Commune	Nombre d'offres	Prix moyen annoncé en € courant	Prix moyen annoncé au m² en € courant
Consthum	29	*	*
Esch-sur-Sûre	7	*	*
Heiderscheid	29	*	*
Hoscheid	26	*	*
Saeul	14	*	*

Moyenne nationale		569,216	3,251

Total d'offres	19,278

Source : Ministère du Logement - Observatoire de l'Habitat (base prix 2010).

Figure 2.3: Always look at your data

So it turns out that indeed, there are some rows that we need to remove. We can start by removing rows where `locality` is missing. Then we have a row where `locality` is equal to "Total d'offres". This is simply the total of every offer from every commune. We could keep that in a separate data frame, or even remove it. Finally there's a row, the last one, that states the source of the data, which we can remove.

In the screenshot above, we see another row that we don't see in our filtered data frame: one where `n_offers` is missing. This row gives the national average for columns `average_prince_nominal_euros` and `average_price_m2_nominal_euros`. What we are going to do is create two datasets: one with data on communes, and the other on national prices. Let's first remove the rows stating the sources:

```
raw_data <- raw_data |>
  filter(!grepl("Source", locality))
```

Let's now only keep the communes in our data:

```
commune_level_data <- raw_data |>
    filter(!grepl("nationale|offres", locality),
           !is.na(locality))
```

And let's create a dataset with the national data as well:

```
country_level <- raw_data |>
  filter(grepl("nationale", locality)) |>
  select(-n_offers)

offers_country <- raw_data |>
  filter(grepl("Total d.offres", locality)) |>
  select(year, n_offers)

country_level_data <- full_join(country_level, offers_country) |>
  select(year, locality, n_offers, everything()) |>
  mutate(locality = "Grand-Duchy of Luxembourg")
```

```
Joining, by = "year"
```

Now the data looks clean, and we can start the actual analysis… or can we? Before proceeding, it would be nice to make sure that we got every commune in there. For this, we need a list of communes from Luxembourg. Thankfully, Wikipedia has such a list.

Let's scrape and save this list:

```
current_communes <- "https://en.wikipedia.org/wiki/List_of_communes_of_Luxembourg" |>
  rvest::read_html() |>
  rvest::html_table() |>
  purrr::pluck(1) |>
  janitor::clean_names()
```

We scrape the table from the Wikipedia page using `{rvest}`. `rvest::html_table()` returns a list of tables from the Wikipedia table, and then we use `purrr::pluck()` to keep the first table from the website, which is what we need (I made the calls to the packages explicit, because you might not be familiar with these packages). `janitor::clean_names()` transforms column

names written for human eyes into machine friendly names (for example `Growth rate in %` would be transformed to `growth_rate_in_percent`).

Let's see if we have all the communes in our data:

```
setdiff(unique(commune_level_data$locality), current_communes$commune)
```

```
 [1] "Bascharage"          "Boevange-sur-Attert" "Burmerange"
 [4] "Clémency"            "Consthum"            "Ermsdorf"
 [7] "Erpeldange"          "Eschweiler"          "Heiderscheid"
[10] "Heinerscheid"        "Hobscheid"           "Hoscheid"
[13] "Hosingen"            "Luxembourg"          "Medernach"
[16] "Mompach"             "Munshausen"          "Neunhausen"
[19] "Redange-sur-Attert"  "Rosport"             "Septfontaines"
[22] "Tuntange"            "Wellenstein"         "Kaerjeng"
```

We see many communes that are in our `commune_level_data`, but not in `current_communes`. There's one obvious reason: differences in spelling, for example, "Kaerjeng" in our data, but "Käerjeng" in the table from Wikipedia. But there's also a less obvious reason; since 2010, several communes have merged into new ones. So there are communes that are in our data, say, in 2010 and 2011, but disappear from 2012 onwards. So we need to do several things: first, get a list of all existing communes from 2010 onwards, and then, harmonise spelling. Here again, we can use a list of Wikipedia:

```
former_communes <- "https://en.wikipedia.org/wiki/Communes_of_Luxembourg#Former_communes"
  rvest::read_html() |>
  rvest::html_table() |>
  purrr::pluck(3) |>
  janitor::clean_names() |>
  dplyr::filter(year_dissolved > 2009)

former_communes
```

```
# A tibble: 20 x 3
   name                year_dissolved reason
   <chr>                        <int> <chr>
 1 Bascharage                    2011 merged to form Käerjeng
 2 Boevange-sur-Attert           2018 merged to form Helperknapp
 3 Burmerange                    2011 merged into Schengen
 4 Clemency                      2011 merged to form Käerjeng
 5 Consthum                      2011 merged to form Parc Hosingen
```

```
 6 Ermsdorf                      2011 merged to form Vallée de l'Ernz
 7 Eschweiler                    2015 merged into Wiltz
 8 Heiderscheid                  2011 merged into Esch-sur-Sûre
 9 Heinerscheid                  2011 merged into Clervaux
10 Hobscheid                     2018 merged to form Habscht
11 Hoscheid                      2011 merged to form Parc Hosingen
12 Hosingen                      2011 merged to form Parc Hosingen
13 Mompach                       2018 merged to form Rosport-Mompach
14 Medernach                     2011 merged to form Vallée de l'Ernz
15 Munshausen                    2011 merged into Clervaux
16 Neunhausen                    2011 merged into Esch-sur-Sûre
17 Rosport                       2018 merged to form Rosport-Mompach
18 Septfontaines                 2018 merged to form Habscht
19 Tuntange                      2018 merged to form Helperknapp
20 Wellenstein                   2011 merged into Schengen
```

As you can see, since 2010 many communes have merged to form new ones. We can now combine the list of current and former communes, as well as harmonise their names:

```
communes <- unique(c(former_communes$name, current_communes$commune))
# we need to rename some communes

# Different spelling of these communes between wikipedia and the data

communes[which(communes == "Clemency")] <- "Clémency"
communes[which(communes == "Redange")] <- "Redange-sur-Attert"
communes[which(communes == "Erpeldange-sur-Sûre")] <- "Erpeldange"
communes[which(communes == "Luxembourg-City")] <- "Luxembourg"
communes[which(communes == "Käerjeng")] <- "Kaerjeng"
communes[which(communes == "Petange")] <- "Pétange"
```

Let's run our test again:

```
setdiff(unique(commune_level_data$locality), communes)
```

```
character(0)
```

Great! When we compare the communes that are in our data with every commune that has existed since 2010, we don't have any commune that is unaccounted for. So are we done with cleaning the data? Yes, we can now actually start with analysing the data. Take a look here[1]

[1] https://is.gd/7PhUjd

to see the finalised script. Also read some of the comments the we've added. This is a typical R script, and at first glance, one might wonder what is wrong with it. Actually, not much, but the problem if you leave this script as it is, is that it is very likely that we will have problems rerunning it in the future. As it turns out, this script is not reproducible. But we will discuss this in much more detail later on. For now, let's analyse our cleaned data.

2.3 Analysing the data

We are now going to analyse the data. The first thing we are going to do is compute a Laspeyeres price index. This price index allows us to make comparisons through time; for example, the index at year 2012 measures how much more expensive (or cheaper) housing became relative to the base year (2010). However, since we only have one good, this index becomes quite simple to compute: it is nothing but the prices at year t divided by the prices in 2010 (if we had a basket of goods, we would need to use the Laspeyeres index formula to compute the index at all periods).

For this section, we will perform a rather simple analysis. We will immediately show you the R script: take a look at it here[2]. For our analysis we selected 5 communes and plotted the evolution of prices compared to the national average.

This analysis might seem trivially simple, but it contains all the needed ingredients to illustrate everything else that we're going to teach you in this book.

Most analyses would stop here: after all, we have what we need; our goal was to get the plots for the 5 communes of Luxemourg, Esch-sur-Alzette, Mamer, Schengen (which gave its name to the Schengen Area) and Wincrange. However, let's ask ourselves the following important questions:

- How easy would it be for someone else to rerun the analysis?
- How easy would it be to update the analysis once new data gets published?
- How easy would it be to reuse this code for other projects?
- What guarantee do we have that if the scripts get run in 5 years, with the same input data, we get the same output?

Let's answer these questions one by one.

[2] https://is.gd/qCJEbi

2.4 Your project is not done

2.4.1 How easy would it be for someone else to rerun the analysis?

The analysis is composed of two R scripts, one to prepare the data, another to actually run the analysis proper. This might seem quite easy, because each script contains comments as to what is going on, and the code is not that complicated. However, we are missing any project-level documentation, that would provide clear instructions as to how to run it. This might seem simple for us who wrote these scripts, but we are familiar with R, and this is still fresh in our brains. Should someone less familiar with R have to run the script, there is no clue for them as to how they should do it. And of course, should the analysis be more complex (suppose it's composed of a dozens scripts), this gets even worse. It might not even be easy for you to remember how to run this in 5 months!

And what about the required dependencies? Many packages were used in the analysis. How should these get installed? Ideally, the same versions of the packages you used and the same version of R should get used by that person to rerun the analysis.

All of this still needs to get documented, but documenting packages and their versions takes quite some time. Thankfully, in part 2, we will learn about the `{renv}` package to deal with this in a couple lines of code.

2.4.2 How easy would it be to update the project?

If new data gets published, all the points discussed previously are still valid, plus you need to make sure that the updated data is still close enough to the previous data that it can pass through the data cleaning steps you wrote. You should also make sure that the update did not introduce a mistake in past data, or at least alert you if that is the case. Sometimes, when new years get added, data for previous years also get corrected, so it would be nice to make sure that you know this. Also, in the specific case of our data, communes might get fused into a new one, or maybe even divided into smaller communes (even though this is has not happened in a long time, it is not entirely out of the question).

In summary, what is missing from the current project are enough tests to make sure that an update to the data can happen smoothly.

2.4.3 How easy would it be to reuse this code for another project?

Said plainly, not very easy. With code in this state you have no choice but to copy and paste it into a new script and change it adequately. For re-usability, nothing beats structuring your code into functions and ideally you would even package them. We are going to learn just that in future chapters of this book.

But sometimes you might not be interested in reusing code for another project: however, even if that's the case, structuring your code into functions and packaging them makes it easy to reuse even inside the same project. Look at the last part of the `analysis.R` script: we copy and pasted the same code 5 times and only slightly changed it. We are going to learn how not to repeat ourselves by using functions and you will immediately see the benefits of writing functions, even when simply to reuse inside the same project.

2.4.4 What guarantee do we have that the output is stable through time?

Now this might seem weird: after all, if we start from the same dataset, does it matter *when* we run the scripts? We should be getting the same result if we build the project today, in 5 months or in 5 years. Well, not necessarily. While it is true that R is quite stable, this cannot necessarily be said of the packages that get used. There is no guarantee that the authors of the packages will not change the package's functions to work differently, or take arguments in a different order, or even that the packages will all be available at all in 5 years. And even if the packages are still available and function the same, bugs in the packages might get corrected that could alter the result. This might seem like a non-problem; after all, if bugs get corrected, shouldn't you be happy to update your results as well? But this depends on what it is we're talking about. Sometimes it is necessary to reproduce results exactly as they were, even if it they were wrong, for example in the context of an audit.

So we also need a way to somehow snapshot and freeze the computational environment that was used to create the project originally.

2.5 Conclusion

We now have a basic analysis that has all we need to get started. In the coming chapters, we are going to learn about topics that will make it easy to write code that is more robust, better documented and tested, and most importantly easy to rerun (and thus to reproduce the results). The first step will actually not involve having to start rewriting our scripts though; next we are going to learn about Git, a tool that will make our life easier by versioning our code.

3 Version control

Modern software development would be impossible without version control systems, and the same goes for building analytical pipelines that are reproducible and robust. It doesn't really matter what the output of the pipeline is: a simple graph, a report with a statistical analysis, a scientific publication, a trained machine learning model that you want to hook to an API… if the code to the project is not versioned, you incur major risks and the pipeline is not reproducible.

But what is version control anyway?

Version control tools make it easy to keep track of the changes that were made to text files (like R scripts). Any change made to any file of a project is cataloged, making it possible to trace back how the file changed, who made the changes, and when these changes were made. Using version control it is also quite easy to collaborate on a project by forcing team members to deal explicitly with the potential conflicts that might arise when the same file got changed by different people at the same time. Should your computer get lost, stolen, or explode, your projects are safely backed up on a server: this is because version control tools make use of a server which keeps track of all the changes (and in some cases, this *server* is actually your teammates' computers!)

Version control tools also make it easy to experiment with new ideas. You can start new *branches* which essentially make a copy of your current project. In this new branch, you can safely experiment with new features, and if the experiments are not conclusive, you can simply discard this branch: the *original* copy of your project will remain untouched. We will also use branches to implement features, fix bugs quickly, and manage the project in a paradigm called *trunk-based development*.

There are several version control tools out there, but Git is undoubtedly the most popular one. You might have heard of Github; this is a service that hosts repositories for your projects, and provides other project management tools such as an issue tracker, project wiki, feature requests… and also very importantly continuous integration. Don't worry if this all sounds very abstract: by the end of this, and the next, chapter you will have all the basic knowledge to use Git and Github.com for your projects.

Git is a tool that you must install on your computer to get started. Once Git is installed, you can immediately start using it; you don't need to open an account on Github (or a similar service), but it is recommended to make collaboration easier (it is possible to collaborate with

several people using Git without a service like Github, by setting up a bare repository on a server or on a network drive you control, but this is outside the scope of this book).

You should know that Github offers private repositories for free, so if you don't want your work to be accessible to the public, that is possible. Only people that you invite to your private repositories will be able to see the code and collaborate with you. It is also possible that your work place has set up a self-hosted Git platform, ask your IT department! Usually these self-hosted platforms are Gitea or Gitlab instances. Gitea, Gitlab, Bitbucket, Codeberg, these are all similar services to Github. All have their advantages and disadvantages.

The advantages of Github are twofold:

- It has a very large community of users;
- Its continuous integration service is incredibly useful, and free for up to 2000 minutes a month.

Disadvantages are:

- It has been bought by Microsoft in 2018;
- It is not possible to self-host an instance of Github (not for free at least).

The fact it is owned by Microsoft may not seem like an issue, but Microsoft's track record of previous acquisitions is not great (Nokia, Skype), and the recent discussions about using source code hosted on Github to train machine learning models (Copilot) can make one uneasy about relying too much on Github.

So while we are going to use Github to host our projects in the remainder of this book, almost everything you are going to learn will be easily transferable to another code hosting platform such as Gitlab or Bitbucket, should you want to switch (or if your workplace has a self-hosted instance from one of Github's competitors). Installing and configuring Git will be exactly the same regardless of the hosting service we use, and all the commands we will use to actually interact with our repositories will be the same as well. So why did we write *almost everything* is the same across any of the code hosting platforms? Well, the two advantages we cited above really give Github an edge; many developers, researchers and data scientists have a Github account already and so if one day you need to collaborate with people, chances are they have an account on Github and not on another code hosting platform.

But what really sets up Github.com apart is Github Actions, Github's continuous integration service. Github Actions is literally a computer in the cloud that you can use to run a set of actions each time you interact with the repository (or at defined moments as well). For example, it would be possible to run automated tests each time a collaborator uploads some changes to the project. This way, we can make sure that no change introduced a bug. Or take this book; each time I write and upload a new section or chapter to Github, the website hosting this ebook gets updated automatically and the PDF of the book gets updated as well, and everything happens automatically. Each Github account gets 2000 minutes a month of

free computing time a month, which is really a lot. In part 2, we will make use of Github Actions to automatically test our code.

By the way, if you're using a cloud service like Dropbox, Onedrive, and the like, DO NOT put projects tracked by Git in them! We really need to stress this: do not track projects with both something like Dropbox and Git. This is because Dropbox and similar services do not deal gracefully with conflicts: if two collaborators change the same file, Dropbox makes two copies of the files. One of the collaborators then has to manually deal with this. The issue is that inside a project that is being tracked by Git, there is a hidden folder with many files that get used for synching the project and making sure that everything runs smoothly. If you put a Git-enabled project inside a Dropbox folder, these files will get accessed simultaneously by different people, and Dropbox will start making copies of these because of conflicts. This really messes up the project and can lead to data loss. Let Git handle the tracking and the collaborating for you. It might seem more complex than a service like Dropbox, and it is, but it is immensely more powerful, and what steep learning curve it might have, it more than makes up for it with the many features it makes available at your fingertips. Unlike Dropbox (or similar services), Git deals with conflicts not on a per-file basis, but on a per line basis. So if two collaborators change the same file, but different lines of this same file, there will be no conflict.

Finally, before starting, there is something important that you need to understand, and people sometimes get confused by it: if a repository is public, this does not mean that anyone can make changes to the code. What this means is that anyone can fork the repository (essentially making a copy of the repository to their Github account) and then *suggest* some changes in a so-called pull request. The maintainer and owner of the original project can then accept these edits or not.

In the remainder of this chapter, you are going to learn how to set up Git on your machine, open a Github account and start using it right away. Then, we are going to show you several scenarios:

- How to collaborate, as a team, on a project;
- How to contribute to someone else's project.

3.1 Installing Git and opening a Github account

Git is a program that you install on your computer. If you're running a Linux distribution, chances are Git is already installed. Try to run the following command in a terminal to see if this is the case:

```
which git
```

If a path like `/usr/bin/git` gets shown, congratulations, you can skip the rest of this paragraph. If something like:

```
/usr/bin/which: no git in (/home/username/.local/bin:/home/username/bin:/usr/local/bin:/usr/l
```

gets shown instead, then this means that Git is not installed on your system. To install Git, use your distribution's package manager, as it is very likely that Git is packaged for your system. On Ubuntu, arguably the most popular Linux distribution, this means running:

```
sudo apt-get update
sudo apt-get install git
```

On macOS and Windows, follow the instructions from the Git Book. It should be as easy as running an installer.

Depending on your operating system, a graphical user interface might have been installed with Git, making it possible to interact with Git outside of the command line. It is also possible to use Git from within RStudio and many other editors have interfaces to Git as well.

We are not going to use any graphical user interface however. This is because there is no common, universal graphical user interface; they all work slightly differently. The only universal is the command line. Also, learning how to use Git via the command line will make it easier the day you will need to use it from a server, which will very likely happen. It also makes our job easier: it is simpler to tell you which commands to run and explain them to you than littering the book with dozens upon dozens of screenshots that might get outdated as soon as a new version of the interface gets released.

Don't worry, using the command line is not as hard as it sounds.

If you don't have already a Github account, now is the time to create one. Just go over to https://github.com/ and simply follow the instructions and select the free tier to open your account.

Now that we have an opened account, we can go to the folder that contains the two scripts we wrote at the start of the book.

3.2 Git superbasics

Open the folder that contains the two scripts in a file explorer. On most Linux desktop environments you should be able to right-click inside that folder anywhere and select an option titled something like "Open Terminal here". On Windows, do the same, but the option is titled "Open Git Bash here". On macOS, you need to first activate this option. Simply google for

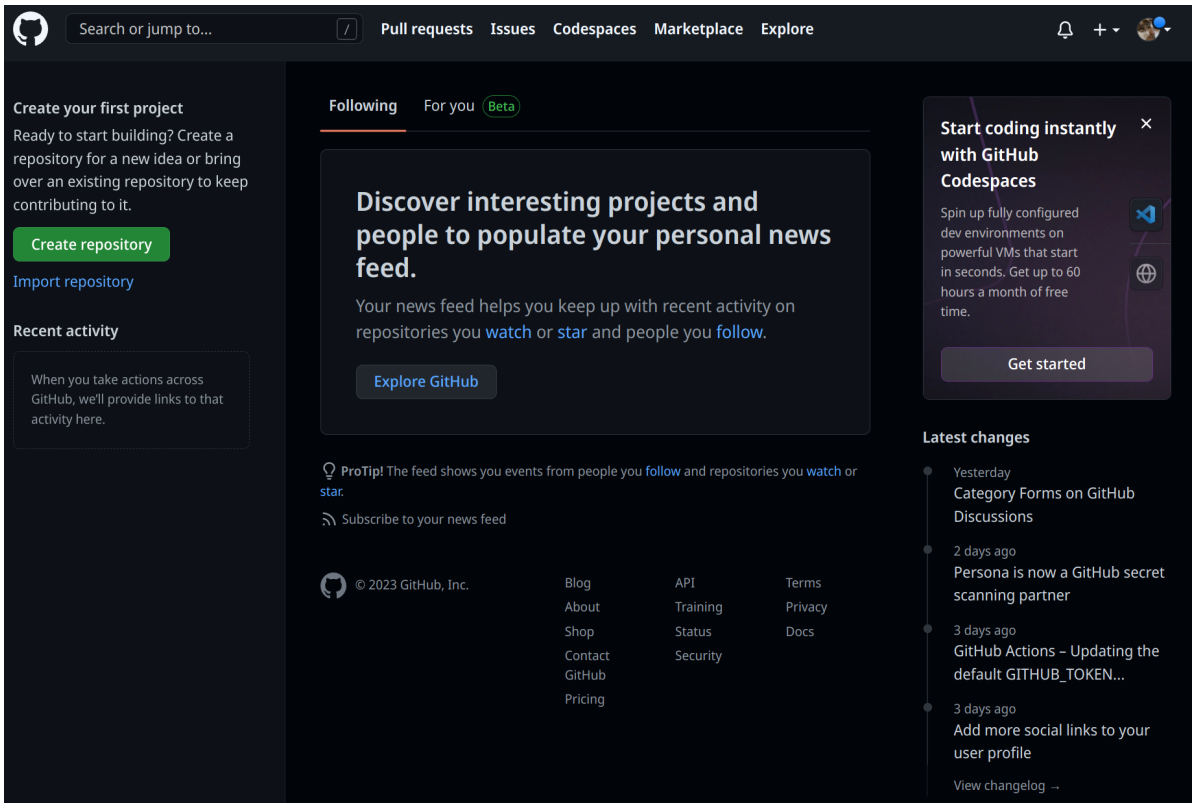


Figure 3.1: This is your Github dashboard

"open terminal at folder macOS" and follow the instructions. It is also possible to drag and drop a folder into a terminal which will then open the correct path in the terminal. Another option, of course, is to simply open a terminal and navigate to the correct folder using `cd` (**c**hange **d**irectory:

```
cd /home/user/housing/
```

(The above command assumes that our project is inside a folder called "housing"). Make sure that you are in the right folder by **l**i**s**ting the contents of the folder:

```
ls
```

One little thing: from now on, the prompt of a terminal (or Git bash terminal on Windows) will start with `owner@localhost`. This is the user called `owner` ("owner" simply because that will be the project manager in our examples from now on) and the computer `owner` uses is called `localhost`. So here is what happens when `owner` runs `ls` on the root directory of the project:

```
owner@localhost  ls
analysis.R save_data.R
```

(on Linux you could also try `ll` which is often available. It is an alias for `ls -l` which provides a more detailed view. There's also `ls -la` which also lists hidden files).

If you want to follow along, create a folder called `housing` and put the two scripts we developed before in there:

- save_data.R: https://is.gd/7PhUjd
- analysis.R: https://is.gd/qCJEbi

Open a terminal in that folder and run `ls` and make sure that you see the two files listed.

It's now time to start tracking these files using Git. In the same window in which we ran `ls`, run now the following `git` command:

```
git init
```

```
owner@localhost  git init
```

```
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/user/six_to/housing/.git/
```

Take some time to read the hints. Many `git` commands give you hints and it's always a good idea to read them. This hint here tells us that the default branch name is "master" and that this is subject to change. For example, if you create a repository on Github, they suggest "main" as the name for the default branch. You need to pay attention to this, because when we will start interacting with our Github repository, we need to make sure that we have the right branch name in mind. Also, note that because the "master" branch is the most important branch, it get sometimes referred to as the "trunk". Some teams that use trunk based development (which I will discuss in the next chapter) even name this branch "trunk". Let's now run this other `git` command:

```
owner@localhost  git status

On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        analysis.R
        save_data.R

nothing added to commit but untracked files present (use "git add" to track)
```

Git tells us quite clearly that it sees two files, but that they're currently not being tracked. So if we would modify them, Git would not keep track of the changes. So it's a good idea to just do what Git tells us to do, let's *add* them so that Git can track them:

```
owner@localhost  git add
```

```
Nothing specified, nothing added.
hint: Maybe you wanted to say 'git add .'?
hint: Turn this message off by running
hint: "git config advice.addEmptyPathspec false"
```

Shoot, simply running `git add` does not do us any good. We need to specify which files we want to add. We can name them one by one, for example `git add file1.R file2.txt` etc, but if we simply want to track all the files in the folder, we can simply use the `.` placeholder:

```
owner@localhost  git add .
```

No message this time… is that a good thing? Let's run `git status` and see what's going on:

```
owner@localhost  git status
```

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   analysis.R
        new file:   save_data.R
```

Nice! Our two files are being tracked now, we can commit the changes. *Committing* means that we are happy with our work, so we can snapshot it. These snapshots then get uploaded to Github by pushing them. This way, the changes will be available for our coworkers for them to pull. Don't worry if this is confusing, it won't be by the end of the chapter. So let's commit them, but I need to tell you something else first: each commit must have a commit message, and we can write this message as an option to the `git commit` command:

```
owner@localhost  git commit -am "Project start"
```

Apparently the `-am` option stands for *apply mailbox*, which I'm sure makes sense to some people, but I prefer to think of `-am` as standing for *a*dd *m*essage. All that remains is pushing this commit to Github. But let's run `git status` again:

```
owner@localhost  git status
```

```
On branch master
nothing to commit, working tree clean
```

This means that every change is accounted for in a commit. So if we were to push now, we could then set our computer on fire: every change would be safely backed up on Github.com.

Before pushing, let's see what happens if we change one file. Open "analysis.R" in any editor and simply change the start of the script by adding one line. So go from:

```
library(dplyr)
library(ggplot2)
library(purrr)
library(tidyr)
```

To:

```
# This script analyses housing data for Luxembourg

library(dplyr)
library(ggplot2)
library(purrr)
library(tidyr)
```

and now run `git status` again:

```
owner@localhost   git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   analysis.R

no changes added to commit (use "git add" and/or "git commit -a")
```

Because the file is being tracked, Git can now tell us that something changed and that we did not commit this change. So if our computer would self-combust, these changes would get lost forever. Better commit them and push them to Github.com as soon as possible!

So first, we need to add these changes to a commit using `git add .`:

```
owner@localhost  git add .
```

(You can run `git status` at this point to check if the file was correctly added to be committed.)

Then, we need to commit the changes and add a nice commit message:

```
owner@localhost  git commit -am "Added a comment to analysis.R"
```

Try to keep commit message as short and as explicit as possible. This is not always easy, but it really pays off to strive for short, clear messages. Also, ideally, you would want to keep commits as small as possible. For example, if you're adding and amending comments in scripts, once you're done with that make this a commit. Then, maybe clean up some code. That's another, separate commit. This makes rolling back changes or reviewing them much easier. This will be crucial later on when we will use trunk based development to collaborate with our teammates on a project. It is generally not a good idea to code all day and then only push one single big fat commit at the end of the day.

By the way, even if our changes are still not on Github.com, we can still now roll back to previous commits. For example, suppose that I delete the file accidentally by running `rm analysis.R`:

```
owner@localhost  rm analysis.R
```

Let's run `git status` and look for the changes (it's a line starting with the word `delete`):

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    analysis.R

no changes added to commit (use "git add" and/or "git commit -a")
```

Yep, `analysis.R` is gone. And deleting on the console usually means that the file is gone forever. Well technically no, there are still ways to recover deleted files, but sence we were using Git we can use it to recover the files! Because we did not commit the deletion of the file, we can simple tell Git to ignore our changes. A simple way to achieve this is to stash the changes, and then *drop* (or delete) the stash:

```
owner@localhost  git stash
```

```
Saved working directory and index state WIP on master: ab43b4b Added a comment to analysis.R
```

So the deletion was stashed away, (so in case we want it back we could get it back with `git stash pop`) and our project was rolled back to the previous commit. Simply take a look at the files:

```
owner@localhost  ls
```

```
analysis.R save_data.R
```

There it is! You can get rid of the stash with `git stash drop`. But what if we had deleted the file and committed the change? In this scenario we could not use `git stash`, but we would need to revert to a commit. Let's try, first let me remove the file:

```
owner@localhost  rm analysis.R
```

and check the status with `git status`:

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    analysis.R

no changes added to commit (use "git add" and/or "git commit -a")
```

Let's add these changes and commit them:

```
owner@localhost  git add .
owner@localhost  git commit -am "Removed analysis.R"
```

```
[master 8e51867] Removed analysis.R
 1 file changed, 131 deletions(-)
 delete mode 100644 analysis.R
```

What's the status now?

```
owner@localhost  git status
```

```
On branch master
nothing to commit, working tree clean
```

Now, we've done it! `git stash` won't be of any help now. So how to recover our file? For this, we need to know to which commit we want to roll back. Each commit not only has a message, but also an unique identifier that you can access with `git log`:

```
owner@localhost  git log

commit 8e51867dc5ae89e5f2ab2798be8920e703f73455 (HEAD -> master)
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:54:30 2023 +0100

    Removed analysis.R

commit ab43b4b1069cd987685253632827f19d7a402b27
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:41:52 2023 +0100

    Added a comment to analysis.R

commit df2beecba0101304f1b56e300a3cd713ce7366e5
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:32:26 2023 +0100

    Project start
```

The first one from the top is the last commit we've made. We would like to go back to the one with the message "Added a comment to analysis.R". See the very long string of characters after "commit"? That's the commit's unique identifier, called hash. You need to copy it (or only like the first 10 or so characters, that's enough as well). By the way, depending on your terminal and operating system, `git log` may open `less` to view the log. `less` is a program that makes it easy to view long documents. Quit it by simply pressing `q` on your keyboard. We are now ready to revert to the right commit with the following command:

```
owner@localhost  git revert ab43b4b1069cd98768..HEAD
```

and we're done! Check that all is right by running `ls` to see that the file magically returned, and `git log` to read the log of what happened:

```
owner@localhost  git log
```

```
commit b7f82ee119df52550e9ca1a8da2d81281e6aac58 (HEAD -> master)
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 18:03:37 2023 +0100

    Revert "Removed analysis.R"

    This reverts commit 8e51867dc5ae89e5f2ab2798be8920e703f73455.

commit 8e51867dc5ae89e5f2ab2798be8920e703f73455 (HEAD -> master)
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:54:30 2023 +0100

    Removed analysis.R

commit ab43b4b1069cd987685253632827f19d7a402b27
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:41:52 2023 +0100

    Added a comment to analysis.R

commit df2beecba0101304f1b56e300a3cd713ce7366e5
Author: User <owner@mailbox.com>
Date:   Sun Feb 5 17:32:26 2023 +0100

    Project start
```

This small example illustrates how useful Git is, even without using Github, and even if working alone on a project. At the very least it offers you a way to simply walk back changes and gives you a nice timeline of your project. Maybe this does not impress you much, because we live in a world where cloud services like Dropbox made things like this very accessible. But where Git (with the help of a service like Github) really shines is when collaboration is needed. Git and code housing services like Github make it possible to collaborate at very large scale: thousands of developers contribute to the Linux kernel, arguably the most successful open source project ever, powering most of today's smartphones, servers, supercomputers and embedded computer,[1] and you can use these tools to collaborate at a smaller scale very efficiently as well.

[1] https://www.zdnet.com/article/who-writes-linux-almost-10000-developers/

3.3 Git and Github

So we got some work done on our machine and made some commits. We are now ready to push these commits to Github. "Pushing" means essentially uploading these changes to Github. This makes them available to your coworkers if you're pushing to a private repository, or makes them available to the world if you're pushing to a public repository.

Before pushing anything to Github though, we need to create a new repository. This repository will contain the code for our project, as well as all the changes that Git has been tracking on our machine. So if, for example, a new team member joins, he or she will be able to clone the repository to his or her computer and every change, every commit message and every single bit of history of the project will be accessible. If it's a public repository, anyone will be able to clone the repository and contribute code to it. We are going to walk you true some examples of how to collaborate with Git using Github in the remained of this chapter.

So, let's first go back to https://github.com/ and create a new repository:

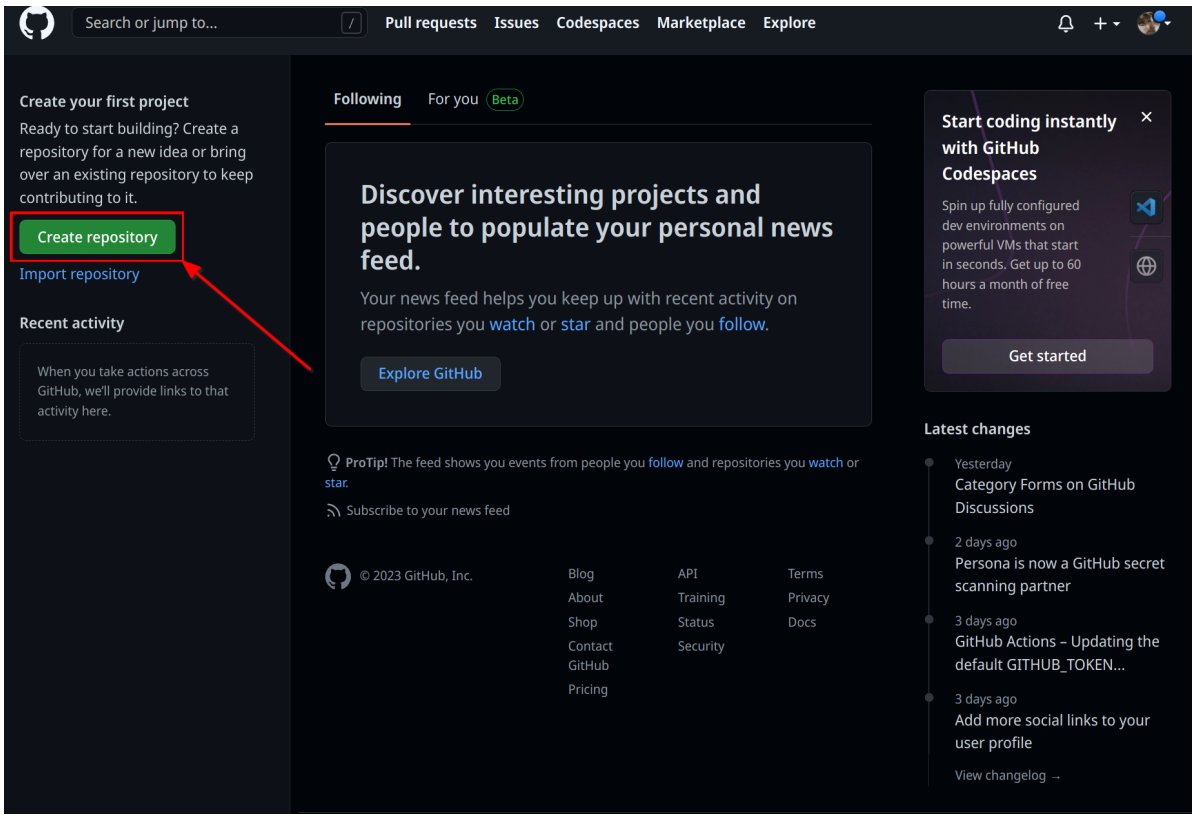


Figure 3.2: Creating a new repository from your dashboard

You will then land on this page:

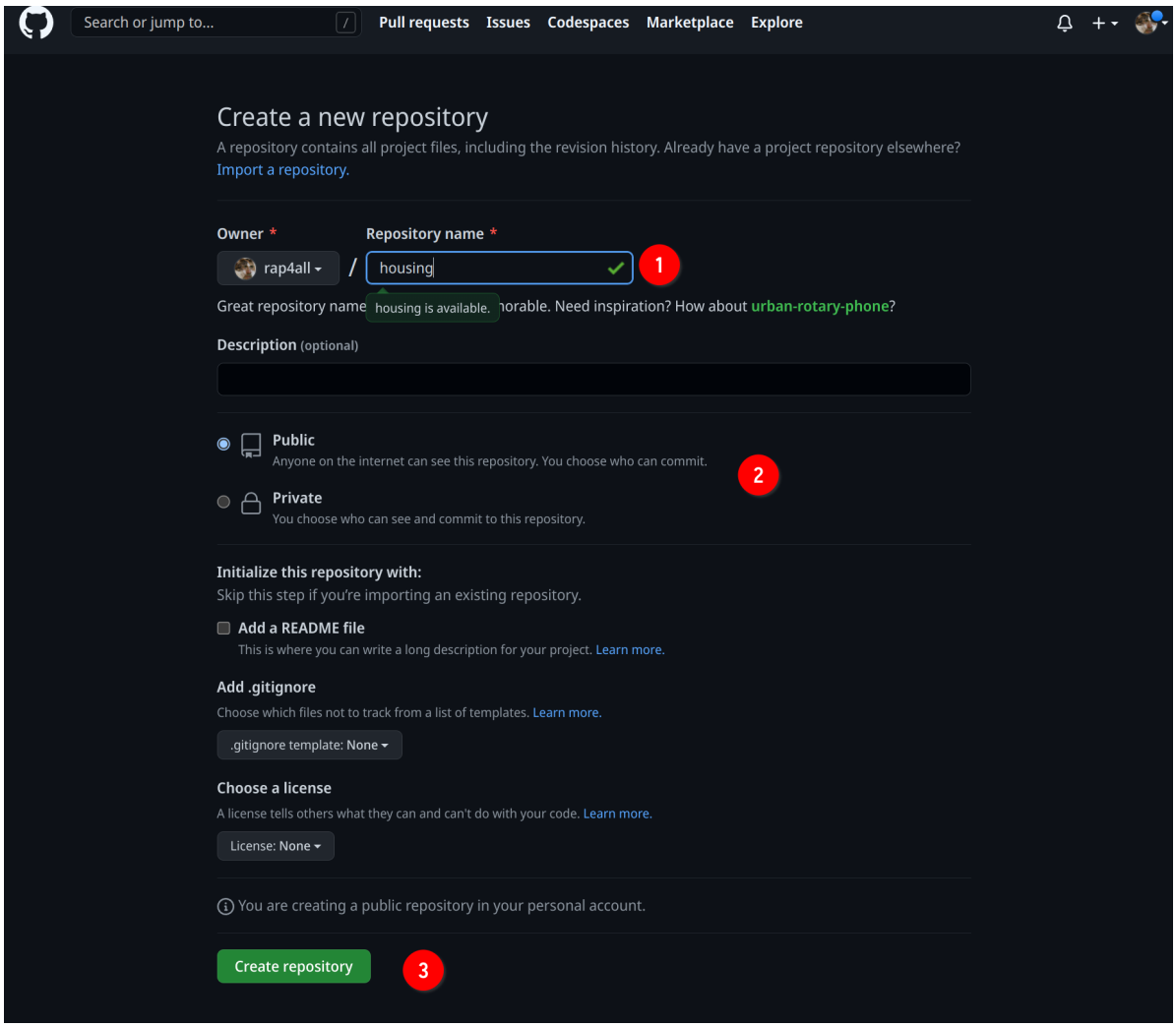


Figure 3.3: Name your repository and choose whether it's a public or private repository

Name your repository, and choose whether it should be open to the word or if it should be private and only accessible to your coworkers. We are going to make it a public repository, but you could make it private and follow along, this would change nothing to what we're going to learn.

We then land on this page:

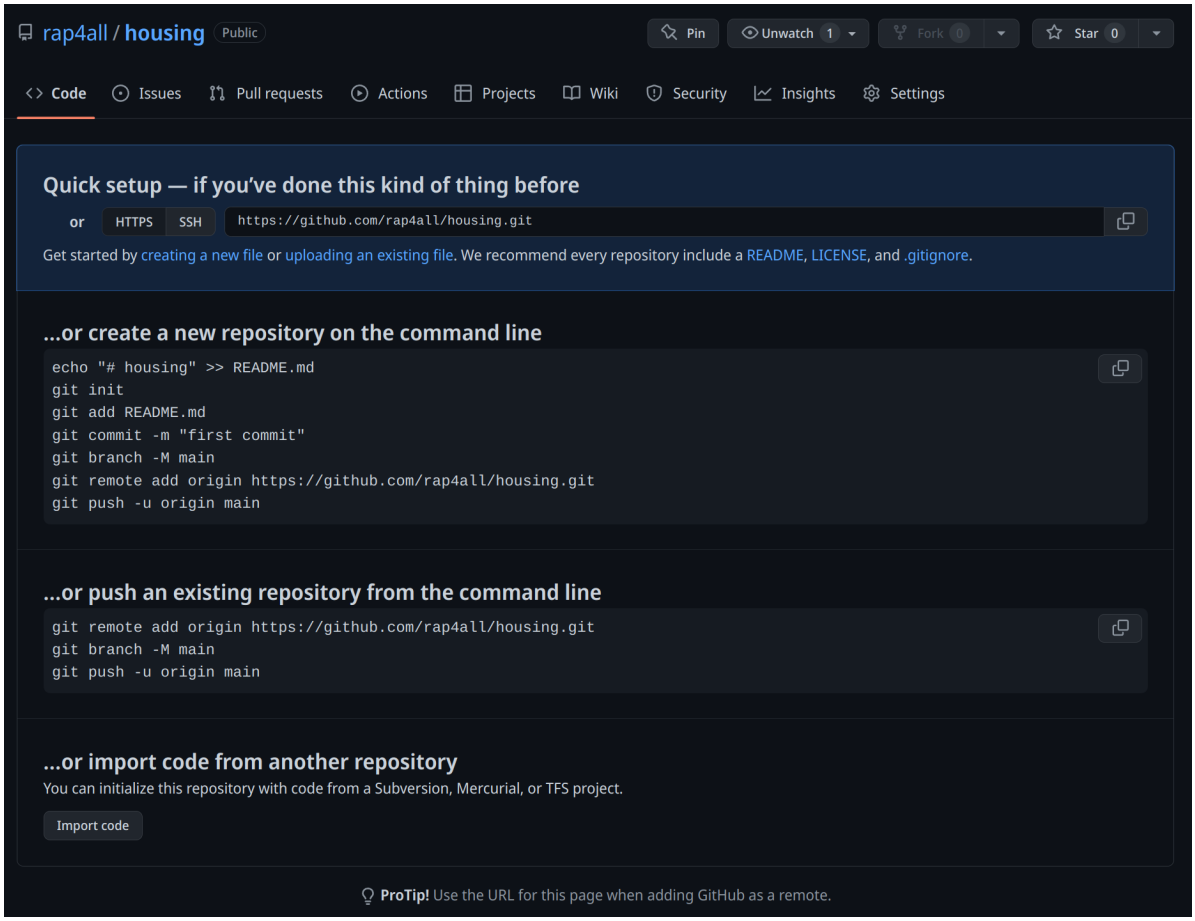


Figure 3.4: Some instructions to get you started

We get some instructions on how to actually get started with our project. The first thing you need to do though is to click on "SSH":

This will change the links in the instructions from `https` to `ssh`. We will explain why this is important in a couple of paragraphs. For now, let's read the instructions. Since we have already started working, we need to follow the instructions titled "...or push an existing repository from the command line". Let's review these commands. This is what Github suggests we run:

```
git remote add origin git@github.com:rap4all/housing.git
```

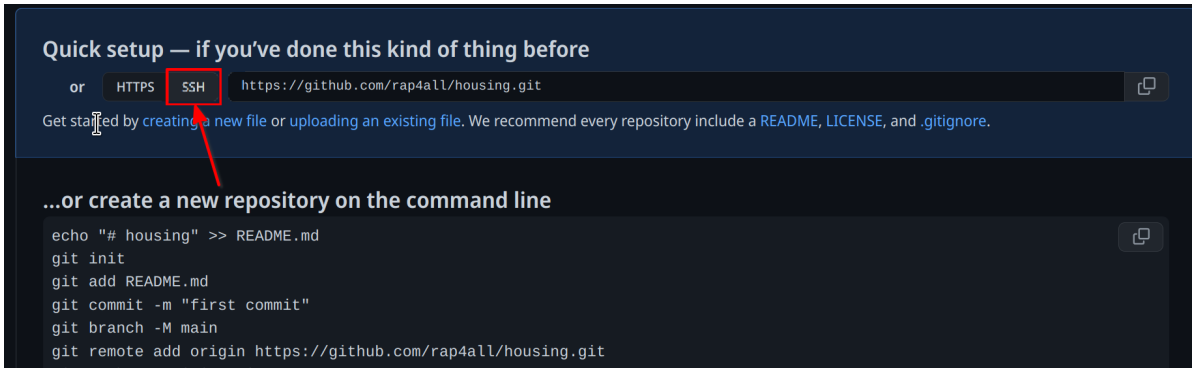


Figure 3.5: Some instructions to get you started

```
git branch -M main
git push -u origin main
```

What's really important is the first command and last command. The first command adds a remote that we name *origin*. The link you see is the link to our repository. If you're following along, you should copy the link from your repository here. It would look exactly the same, but the user name `rap4all` would be replaced by your Github username.

Adding the remote links to our folder in our machine to the Github repository online. So now, every time we push, our changes will get uploaded to Github. The second line renames the branch from "master" to "main". You are of course free to do so. I don't like changing the defaults from Git, so I will keep using the name "master". The last command pushes our changes to the "main" branch (but we need to change "main" to "master").

Let's do just that:

```
owner@localhost  git remote add origin git@github.com:rap4all/housing.git
```

This produces no output. We're now ready to push:

```
owner@localhost  git push -u origin master
```

and it fails:

```
ERROR: Permission to rap4all/housing.git denied to b-rodrigues.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

The reason is quite simple: Github has absolutely no idea who we are! Remember, if the repository is public, anyone can clone it. But that doesn't mean that anyone can simply push code to the repo! This means that we need a way to tell Github that we are the owner of the repository. For this, we need a way to log in securely, and we will do so using a public/private rsa key pair. The idea is quite simple; we are going to generate two files on our computer. These two files form a public/private key pair. We are going to upload the public key to Github; and every time we want to interact with Github, Github will check the public key to the private key that we keep on our machine (never, ever, send the private key to anyone). If they match, Github knows that we are who we claim to be and will let us push to the repository. This is why we switched from `https` to `ssh` before. `https` would allow us to log in by typing a password each time we push (but actually, not anymore, since password login was turned off some years ago). It is much easier to not have to log in manually and let our key pair do the job for us.

Let's generate a public/private rsa key pair. Open a terminal on Linux or macOS, or Git Bash on Windows and run the following command:

```
owner@localhost   ssh-keygen
```

The following lines will appear in your terminal:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
```

Simply leave this empty and press enter. This next message now appears:

```
Enter passphrase (empty for no passphrase):
```

Leave it empty as well. Entering a passphrase is not really needed, since the ssh key pair itself will deal with the login. In some situations, a passphrase might be useful if you're worried that someone might get physical access to your machine and push code by impersonating you. But if you work with such sensitive data and code that this is a real worry, maybe don't use Github?

So once you pressed enter, you get asked to confirm the passphrase:

```
Enter same passphrase again:
```

Here again, simply leave it empty and press enter on your keyboard. Once this is done, you should see this:

```
Your identification has been saved in /home/user/.ssh/id_rsa
Your public key has been saved in /home/user/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:tPZnR7qdN06mV53Mc36F3mASIyD55ktQJFBAVqJXNQw owner@localhost
The key's randomart image is:
+---[RSA 3072]----+
|   .*=E*=.       |
|   o o.oo.. .    |
|  . .  o.  o o   |
|   .  ..o.  . o  |
|       +S    o.+.|
|       .o.   o.o*|
|       . o. + +=*|
|        .  o ++*=|
|            ..=oo|
+----[SHA256]-----+
```

If now you go to the specified path on the first line (so in our case `/home/user/.ssh/` you should see two files, `id_rsa` and `id_rsa.pub`, the private and public keys respectively. We're almost done: what you need to do now is copy the contents of the `id_rsa.pub` file to Github. Go to your profile settings:

And then click on "SSH and GPG keys":

and then click on "New SSH key". Name this key (it's a good idea to write something that makes recognizing the machine the key was generated easily) and paste the contents of `id_rsa.pub` in the text box and click on "add SSH key":

We can now go back to our terminal and try to push again:

```
owner@localhost  git push -u origin master
```

The following message gets printed:

```
The authenticity of host 'github.com (140.82.121.3)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Type `yes` and then you should see the following:

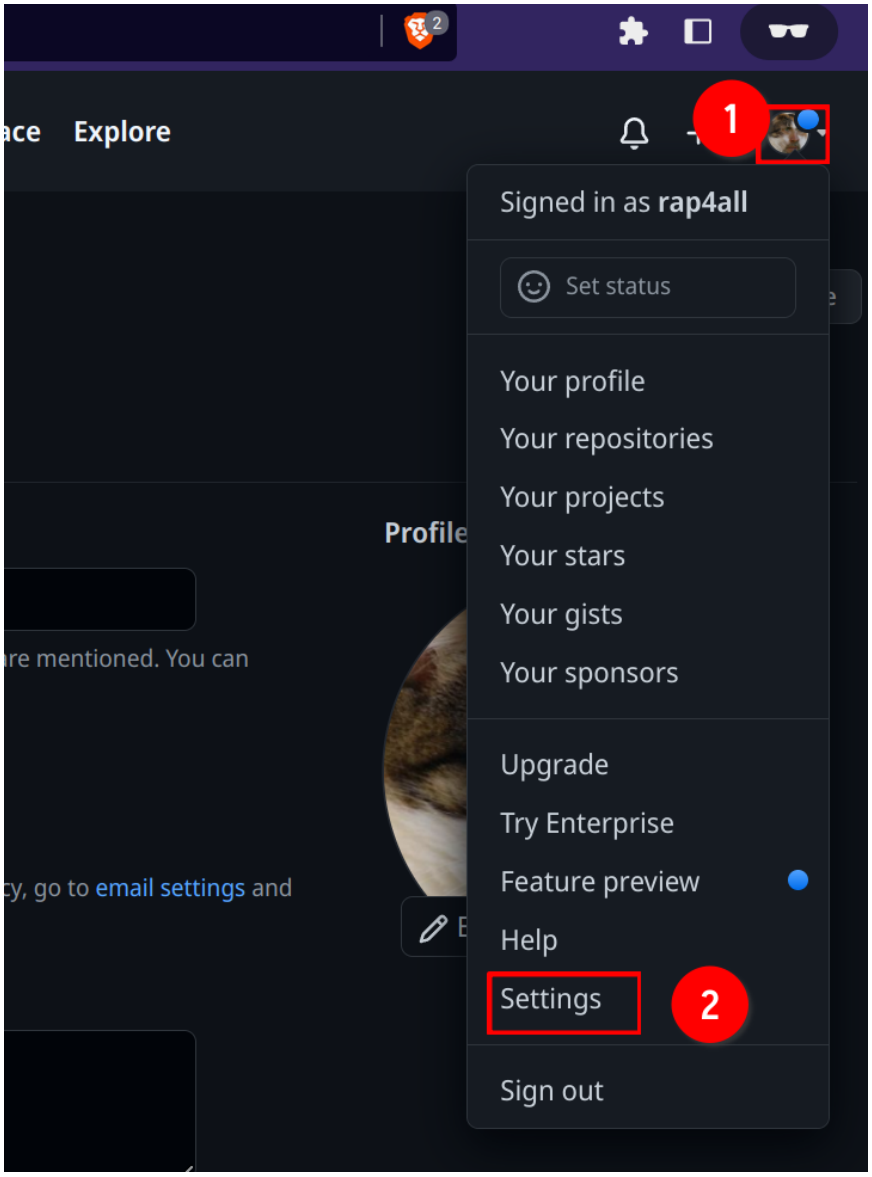


Figure 3.6: Click on your user profile's image in the top-right corner

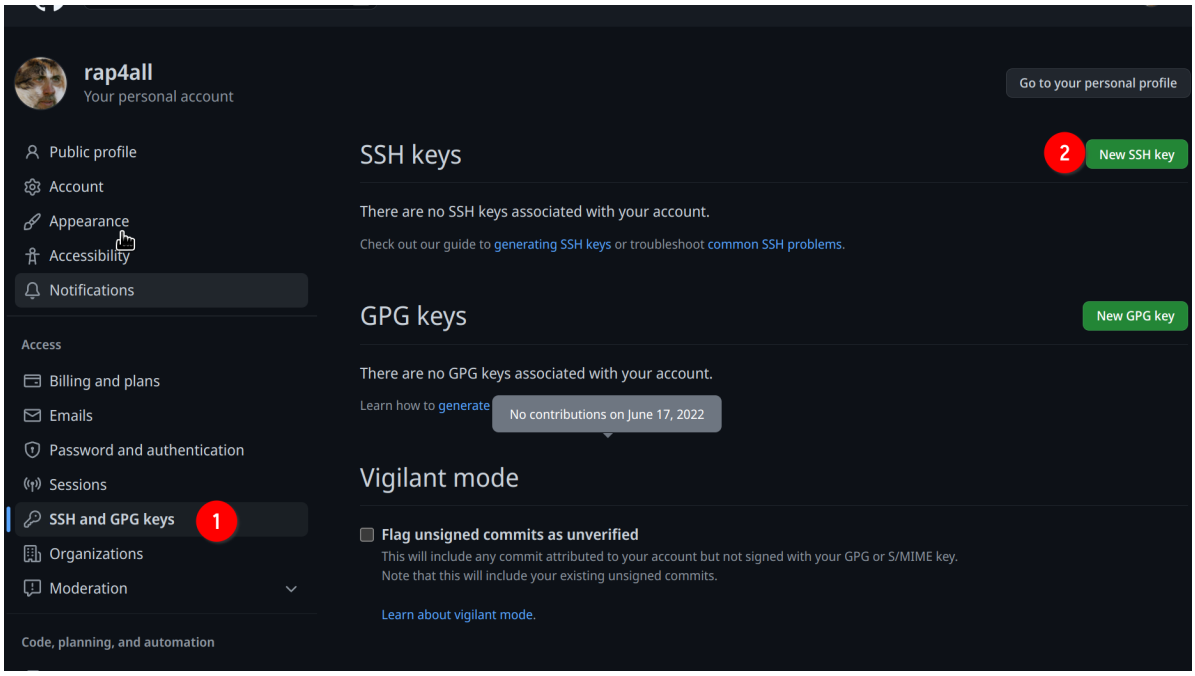


Figure 3.7: Go to your user settings and choose “SSH and GPG keys”

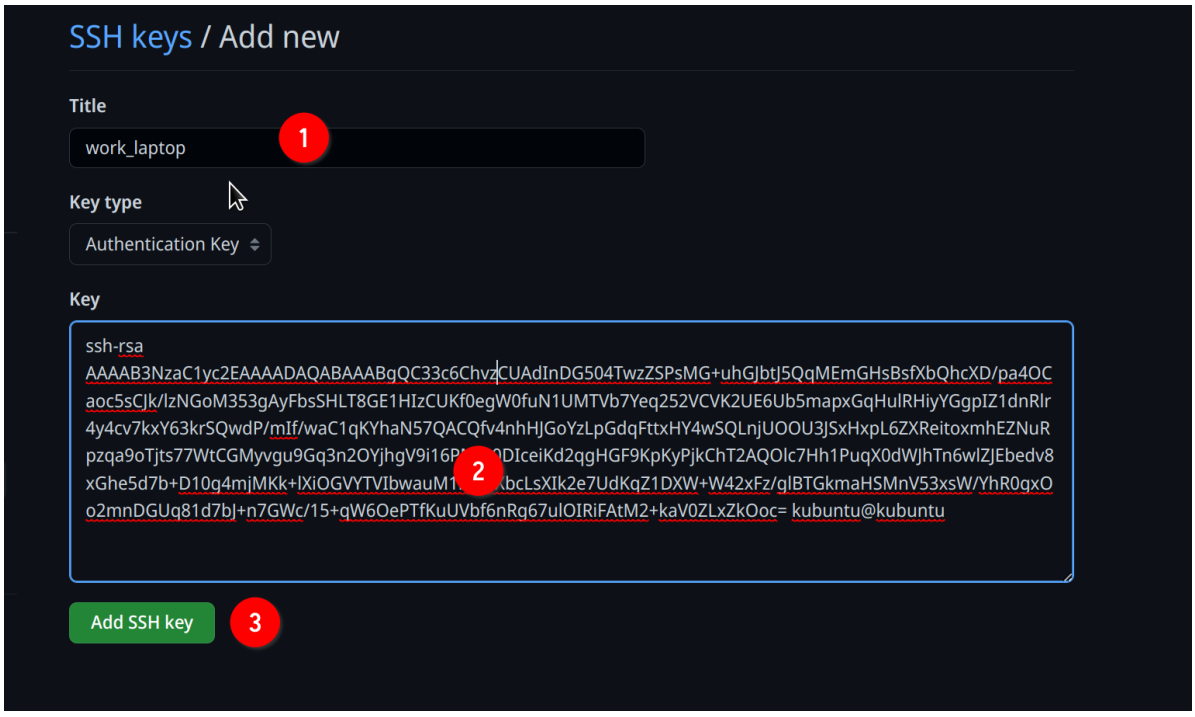


Figure 3.8: Copy the contents of the public key here

```
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (10/10), 2.77 KiB | 2.77 MiB/s, done.
Total 10 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To github.com:rap4all/housing.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

And we're done! Our commits are now safely backed up on Github. If we go to our repository's main page, we should see the following:

3.4 Getting to know Github

We have succeeded in installing Git and making it work with our Github account. If you use another machine for development, you will need to generate another rsa key pair on that machine and add the public key to Github. If you use another code hosting platform, you can use the same rsa key pair, but will need to add the public key to this other code hosting platform. You can even use the same key pair as a passwordless authentication method for ssh (for example to log into a server, but this is outside the scope of the present book). Before continuing we are going to take a little tour of Github.

Once you're on your repository's landing page you see the same files and folders as in the root directory of the project on your computer. In our case here, we see our two files. Github suggests that we add a README file; we are going to ignore this for now. Take a closer look at the menu at the top, below your repository's name:

Most important for our needs is the "Issues", "Pull requests", "Actions" and "Settings" tab.

In the next chapter we are going to learn about pull requests which are essential for collaborating using Git and Github.com. We will learn about the "Actions" tab in the second part of the book.

So let's start with "Settings".

There are many options that you can choose from, but what's important for our purposes is the "Collaborators" option. This is where you can invite people to contribute to the repository. People that are invited in this way can directly push to the repository. Let's invite the author of this book:

Start by typing the person's Github username. You can also invite collaborators by providing their email address.

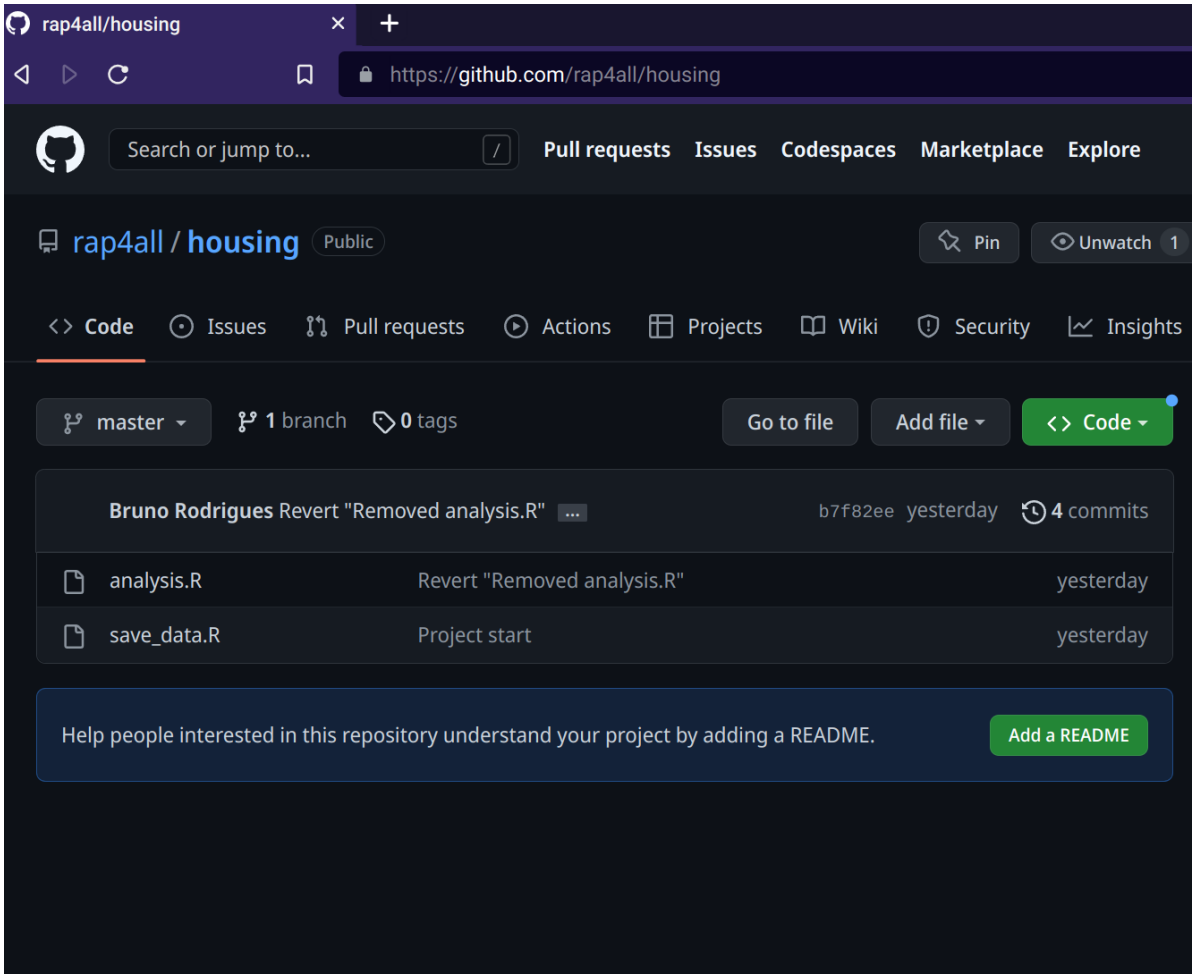


Figure 3.9: Finally!

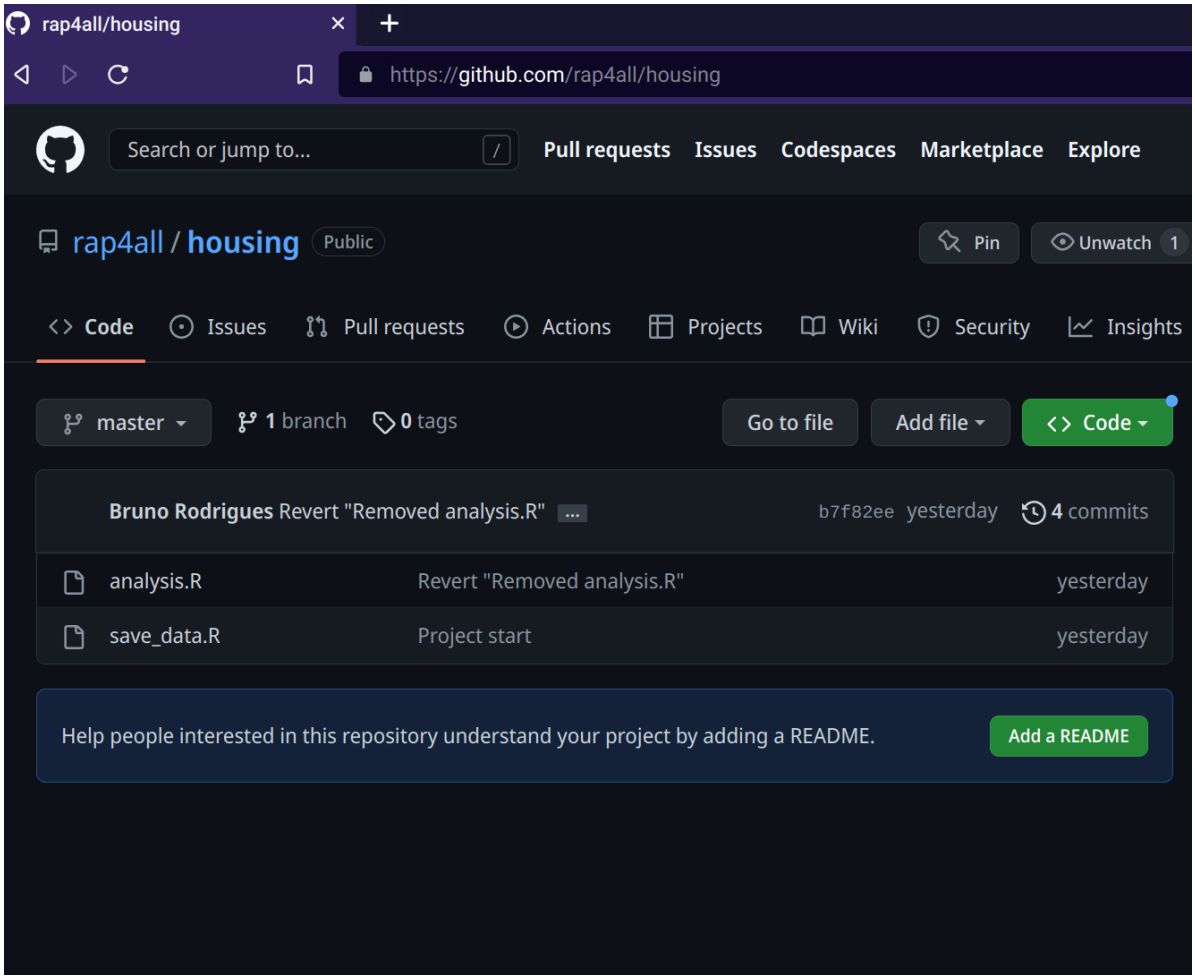


Figure 3.10: You repository's landing page

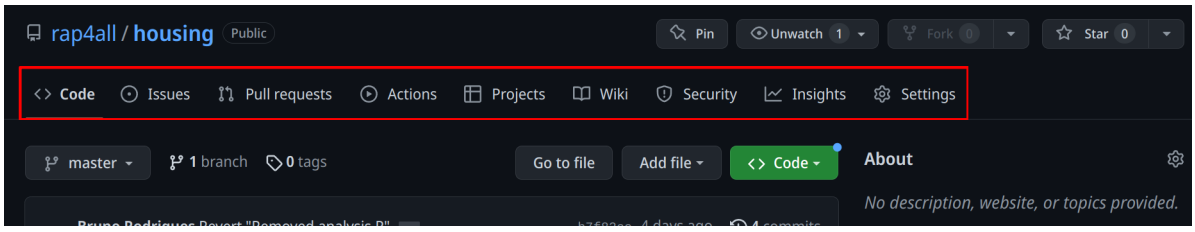


Figure 3.11: Several options to choose from

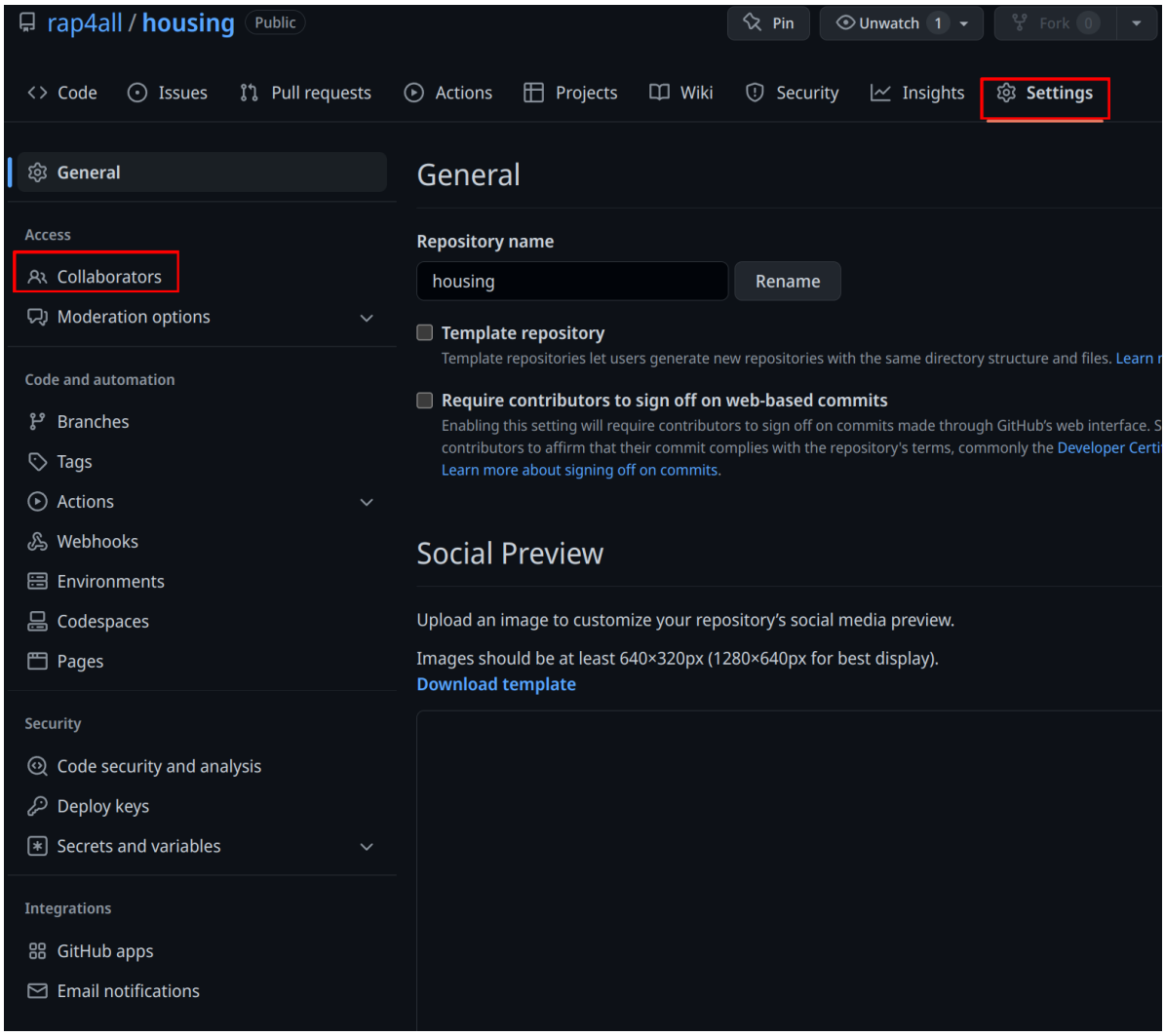


Figure 3.12: Choose the "Settings" tab

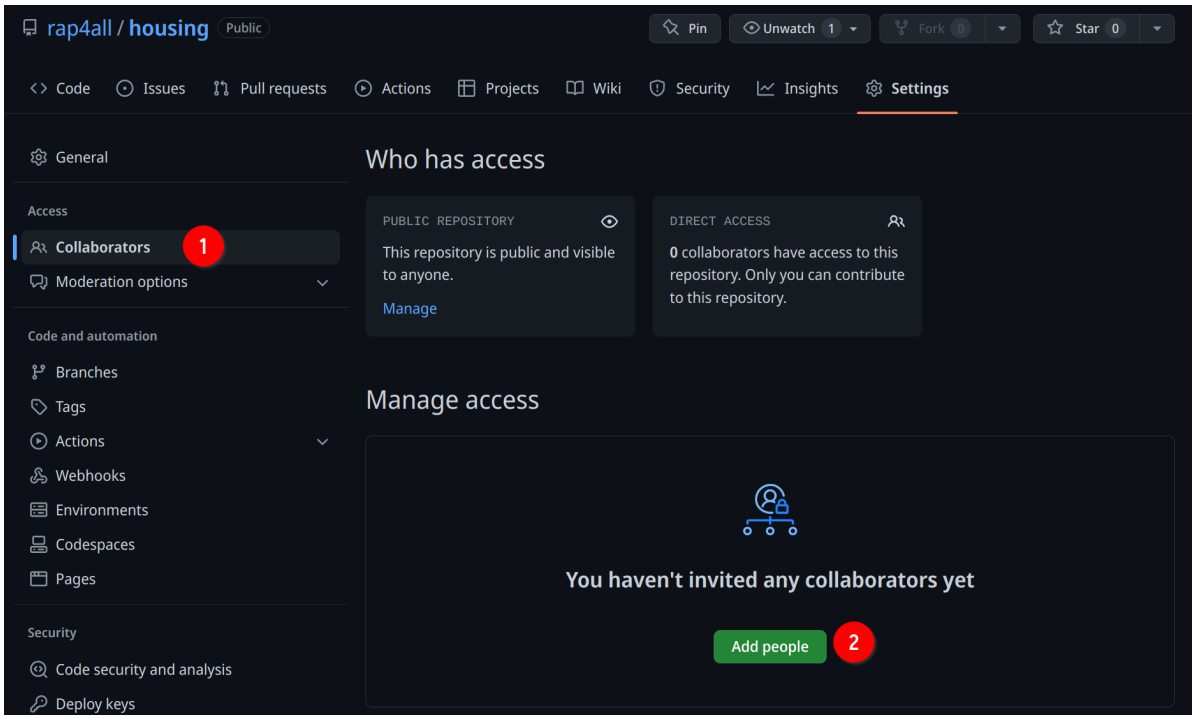


Figure 3.13: Choose the “Settings” tab

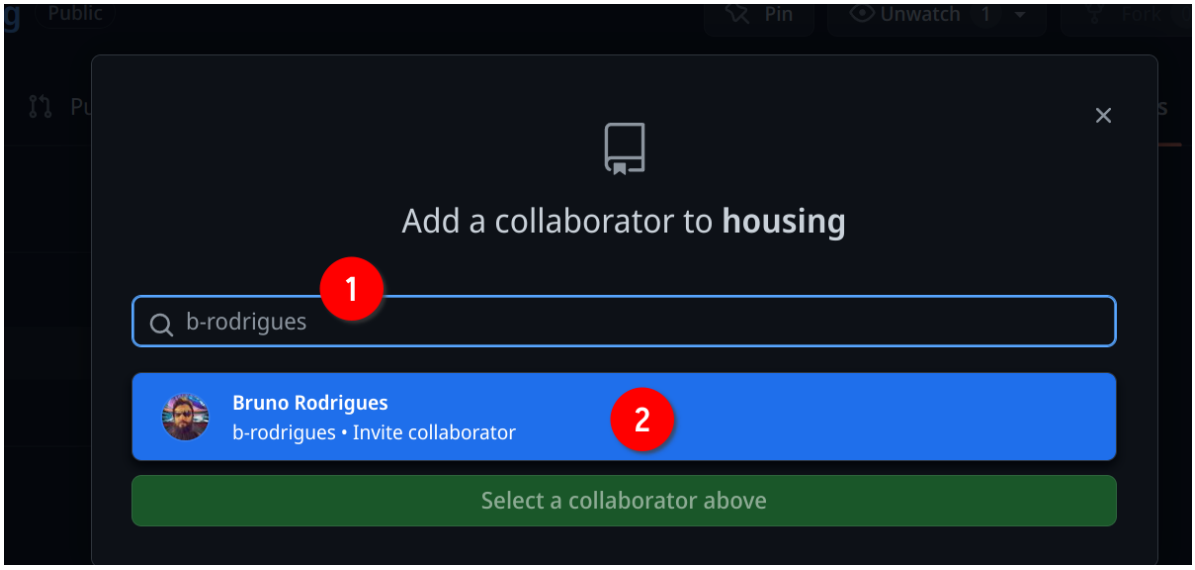


Figure 3.14: Choose the “Settings” tab

Click then on the user's profile and he or she should get an invitation per email.

This is what it looks like from the perspective of Bruno's account now:

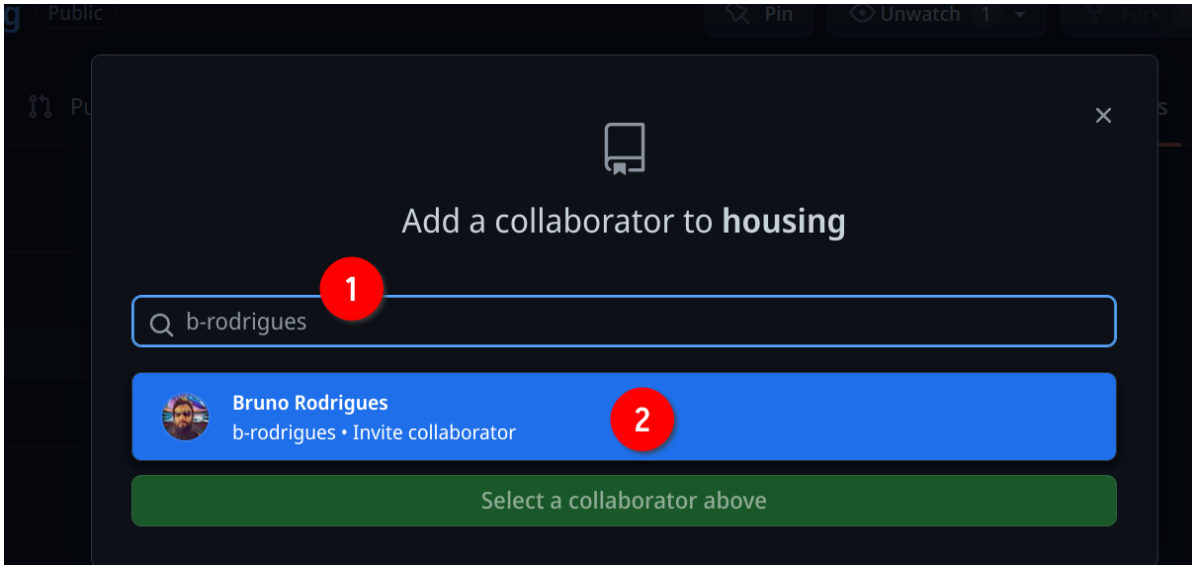


Figure 3.15: Bruno can now push as if he owned the repository

It's important to understand the distinction between inviting someone to contribute to the repository and have someone from outside the project contribute. We are going to explore these two scenarios in the next section, but before that, let's see what the "Issues" tab is about.

If the repository is public, anyone can open an issue to either submit a bug, or suggest some ideas, and if the repository is private, only invited collaborators can do this.

Let's open an issue to illustrate how this works:

Give a nice title to the issue (1), add a thorough description (2), (optionally) assign it to someone (3) and (optionally) add a label to it (4), finally click on "Submit new issue" (5) to submit the issue:

Sometimes issues don't need to be very long, and act more as reminders than anything else. For example here, the owner of the repository didn't have the time to add a Readme, but didn't want to forget to add one later on. The author assigned the issue to Bruno: so it'll be Bruno's job to add the Readme. Issue-driven project management is a very valid strategy when working asynchronously and in a decentralized fashion.

If you encountered a bug and want to open an issue, it is very important that you provide a minimal, reproducible example (MRE). MREs are snippets of code that can be run very easily by someone other than yourself and which produce the bug reliably. Interestingly, if you

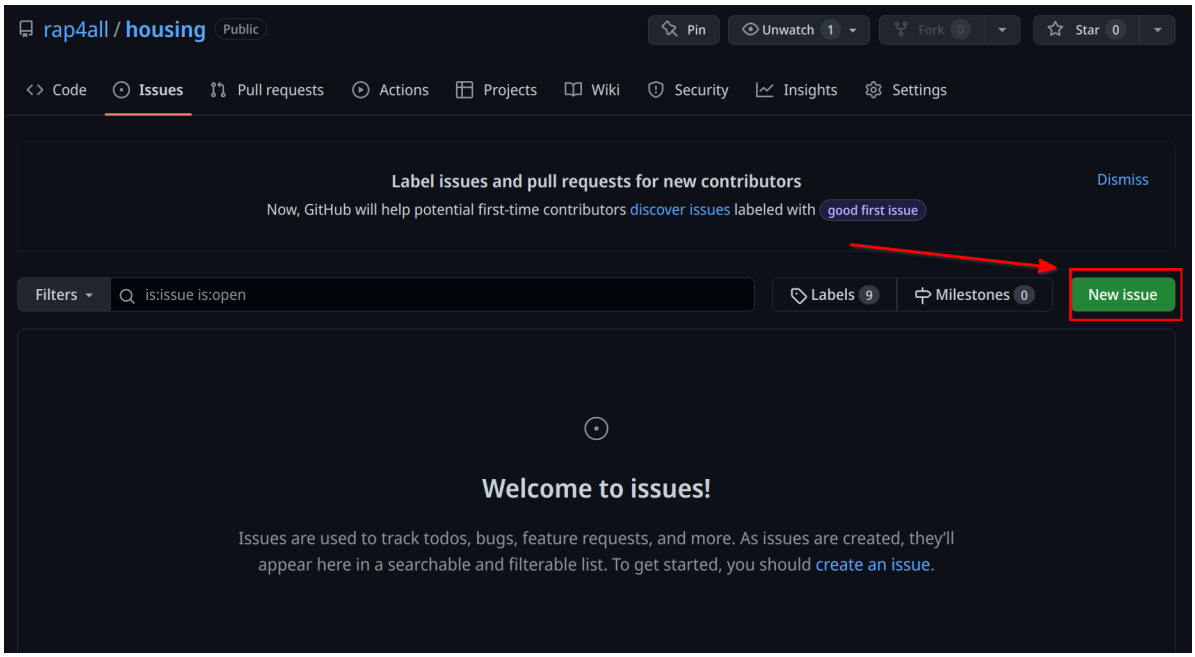


Figure 3.16: Anyone can open an issue in a public repository

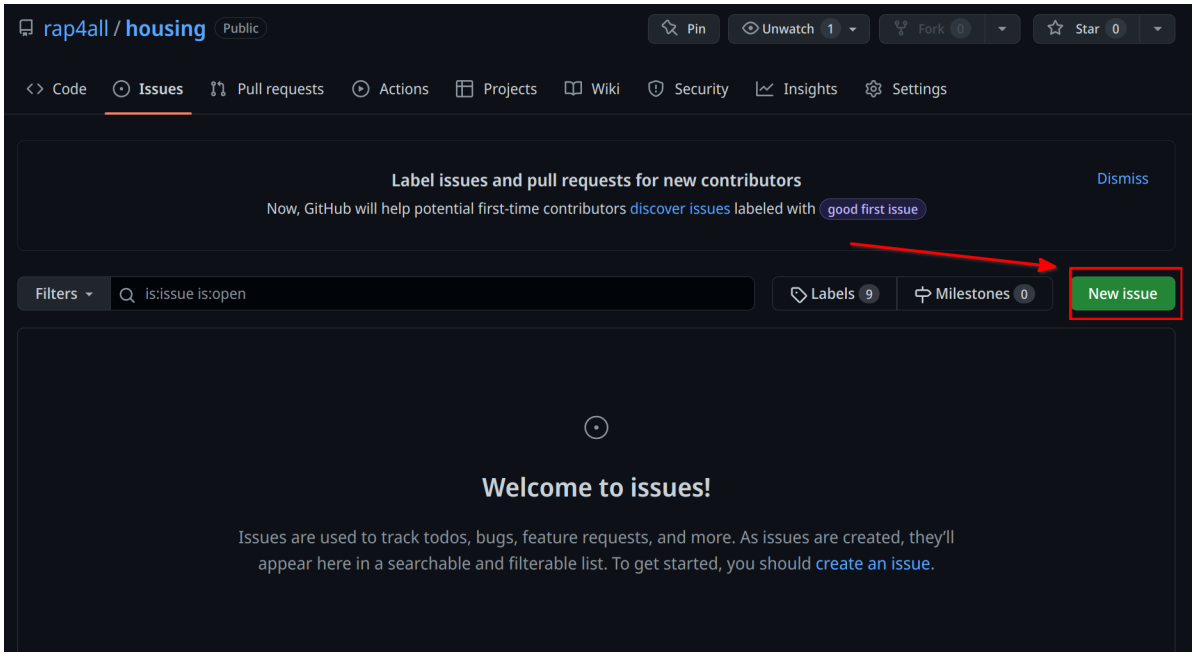


Figure 3.17: Write what the issue's about here

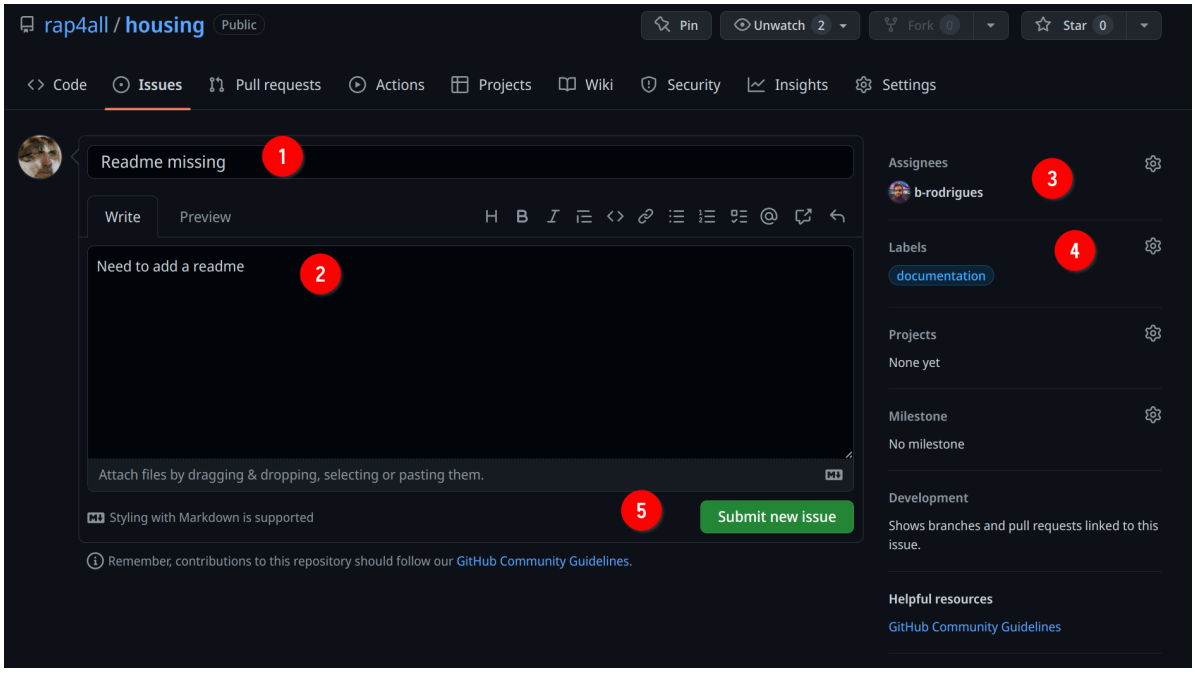


Figure 3.18: Try to provide as many details as possible

understand what makes an MRE minimal and reproducible, you understand what will make our pipelines reproducible as well. So what's important for an MRE?

First, the code needs to be self-contained. For example, if some data is required you need to provide the data. If the data is sensitive, you need to think about the bug in greater detail: is the bug due to the structure of the data, or does the bug manifest itself on any kind of data? If that's the case, use some of the built-in datasets to R (`iris`, `mtcars`, etc) for your MRE.

Does your MRE require extra packages to run? Then make this as clear as possible, and not only provide the package names, but also their versions (it is a good idea to copy and paste the output of `sessionInfo()` at the end of the issue.

Finally, does your example depend on some object defined in the global state? If yes, you also need to provide the code to create this object.

The bar you need to set for an MRE is as follows: bar needed package dependencies that may need to be installed beforehand, people that try to help you should be able to run your script by simply copy-and-pasting it into an R console. Any other manipulation that you require from them is unacceptable: remember that in open source development, developers very often work during their free time, and don't owe you tech support! And even if they did, it is always a good idea to make it as easy as possible for them to help you, because it simply increases the likelihood that they will actually help.

Also, writing an MRE can usually make you actually debug the code yourself. Just like in rubber duck debugging, the fact of simply trying to explain the problem can lead to finding what's wrong. But by writing an MRE, you're also reducing the problem into its most basic parts, and removing everything unnecessary. By doing so, you might realize that what you thought was a bug of the library was maybe rather a problem between the keyboard and the chair.

So don't underestimate the usefulness of creating high-quality MREs for your issues! One package that can assist you with this is {reprex} (read about it here).

3.5 Conclusion

You should now have your first repository and know the very basics of using Git and Github.com. If you did not understand everything, take some time to rerun the commands from above. Maybe add some more files to your repo, remove them, try to revert certain commits, etc. Create a new repo and try to push some files or scripts to it. Really take the time to understand what is going on and how to use these tools, because they are essential for reproducibility.

4 Collaborating with Github

As already mentioned several times, there are two ways of collaborating with Git (and Github): either as a team, or either as an external dev (external, as in, not part of the development team) who wishes to provide some code to a project (this only works for repositories that are public).

We are going to learn about these two ways of collaborating. Let's first focus on collaboration within a team.

4.1 Collaborating as a team using *trunk-based development*

4.1.1 TBD basics

Remember the issue we opened and assigned to Bruno? Bruno will now solve this issue by adding a Readme. This will be also the opportunity to introduce trunk-based development. The idea of trunk-based development is simple; team members should work on separate branches to add features or fix bugs, and then merge their branch to the "trunk" (in our case the *master* branch) to add their changes back to the main code-base. And this process should happen quickly, ideally every day, or as soon as some code is ready. This way, if conflicts arise, they can be dealt with quickly. This also makes code review much easier, because the reviewer only needs to review little bits of code at a time. The alternative would be for each team member to work on his or her own branch for days or even weeks. Once the branches get merged into the trunk, reviewing all the changes and solving the conflicts that will arise would be very painful. To avoid this, it is best to merge every day or each time a piece of code is added, and, **very importantly**, this code does not break the whole project (we will be using unit tests for this later).

So in summary: to avoid a lot of pain later by merging branches that moved away too much from the trunk, we will create branches, add our code, and merge them to the trunk as soon as possible. *As soon as possible* can mean several things, but usually this means as soon as the feature was added, bug was fixed, or as soon as we added some code that does not break the whole project, even if the feature we wanted to add is not done yet. The philosophy is that if merging fails, it should fail as early as possible. Early failures are easy to deal with.

So, back to our issue. First, Bruno needs to clone the repository:

```
git clone git@github.com:rap4all/housing.git
```

Because Bruno was added as a collaborator, Bruno can work on the repository just like the author.

Bruno will now create a new branch by using the `git checkout` command with the `-b` flag:

```
bruno@computer  git checkout -b "add_readme"
```

The project automatically switches to the new branch:

```
Switched to a new branch 'add_readme'
```

We can also run `git status` to double-check:

```
bruno@computer  git status
```

```
On branch add_readme
nothing to commit, working tree clean
```

Let's add a file called `README.md` and add the following to it:

```
# Housing data for Luxembourg

These scripts for the R programming language download nominal housing prices
from the *Observatoire de l'Habitat* and tidy them up into a flat data frame.

- save_data.R: downloads, cleans, and creates data frames from the data
- analysis.R: creates plots of the data
```

Let's save this and run `git status` to see what happened:

```
bruno@computer  git status
```

Git tells us that the `README.md` file is not being tracked:

```
On branch add_readme
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

So next we are going to track it and push the changes. Also, we are going to use a neat trick when pushing: we are going to use the commit message to state the issue was fixed, which will automatically close the issue for us:

```
bruno@computer  git add .
bruno@computer  git commit -am "fixed #1"
```

`#1` refers to the number of the issue. Because it's the first issue, it can simply be referred to as `#1`. Bruno will now push:

```
bruno@computer  git push origin add_readme
```

As you can see from the command above, Bruno pushes to "add_readme", not "master". If he tried to push to "master" a message saying that "master" is up-to-date would get printed. Let's see the output of pushing to "add_readme":

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 501 bytes | 501.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'add_readme' on GitHub by visiting:
remote:      https://github.com/rap4all/housing/pull/new/add_readme
remote:
To github.com:rap4all/housing.git
 * [new branch]      add_readme -> add_readme
```

Git tells us that we now need to create a pull request. What is that? Well, if we want to merge our brunch back to the trunk, we need to do so by using a pull request. Let's see what Bruno sees on Github:

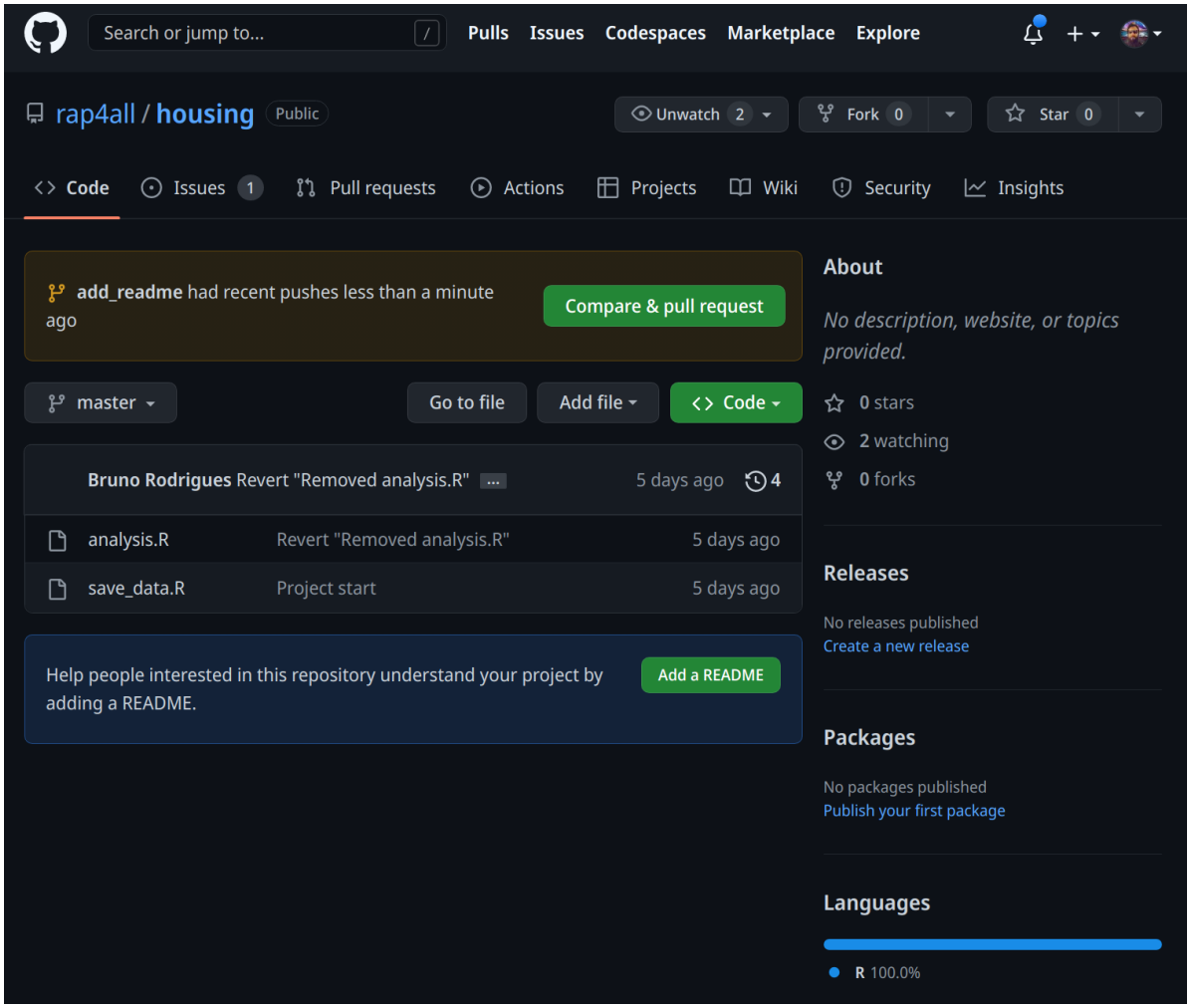


Figure 4.1: Bruno sees that the "add_readme" branch has been recently updated

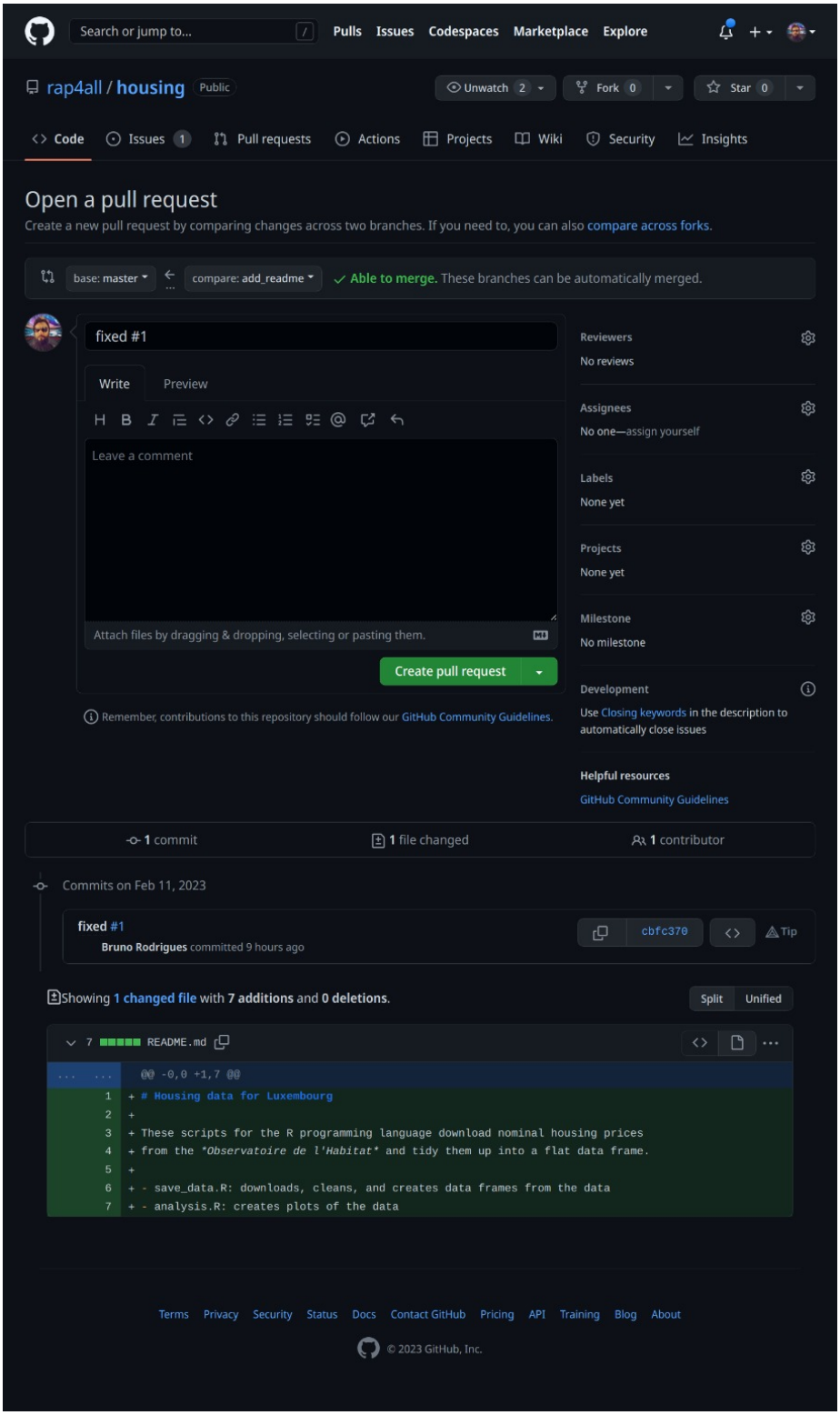


Figure 4.2: This screen makes it easy to see what changed

Bruno can now decide to continue working on this branch, or, since the purpose of this branch was only to add the readme file, decide instead to do a pull request. By clicking on the "Compare & pull request" button Bruno now sees this:

Bruno can leave a comment, and see what changed (in this case, a single file was added) and most importantly, add a reviewer if needed:

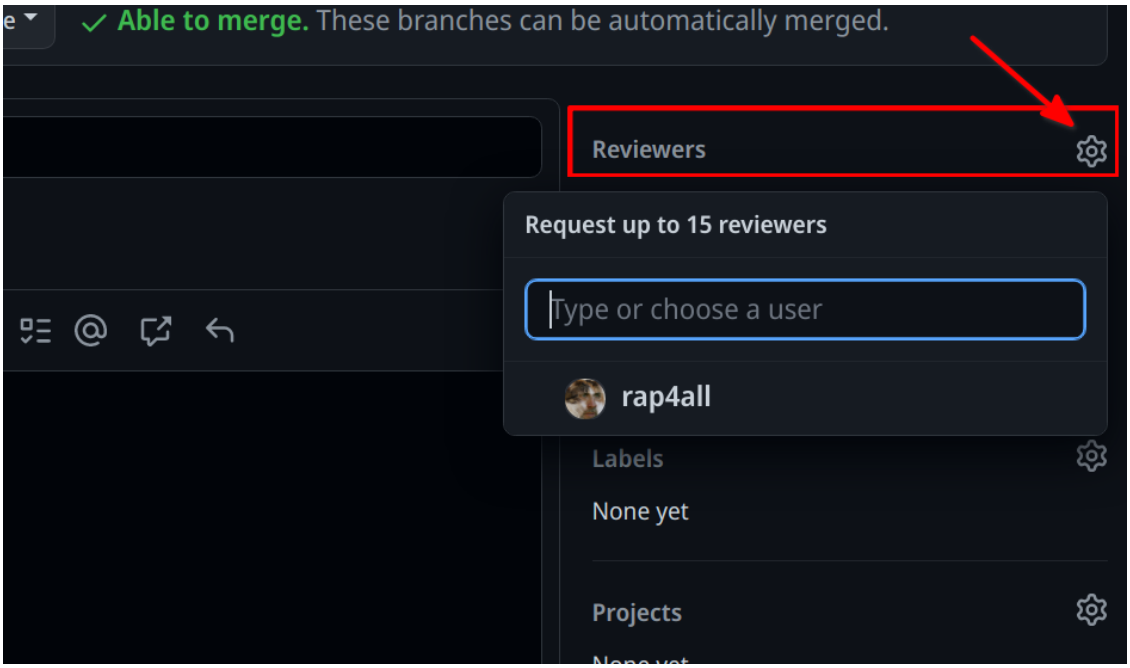


Figure 4.3: Let boss decide if this is good enough

This is what Bruno sees now:

Bruno requested the review, but Github tells us that the branch can safely be merged. This is because we added a file and did not touch anything else, and also because the owner of the repository was asleep while Bruno was working, so there was no opportunity for conflicts to arise.

Let's see what the owner now sees. The owner should have gotten a notification to review the pull request:

By clicking on the notification, the owner gets taken to this view:

Here, the reviewer can check the commit, the files that were changed, and see if there are any conflicts between this code and the code base on the master (or trunk) branch. Github also tells us two interesting things: the owner can add a rule that states that any pull request must be approved, and also that continuous integration has not been set up (we are going to see what this means in the second part of this book).

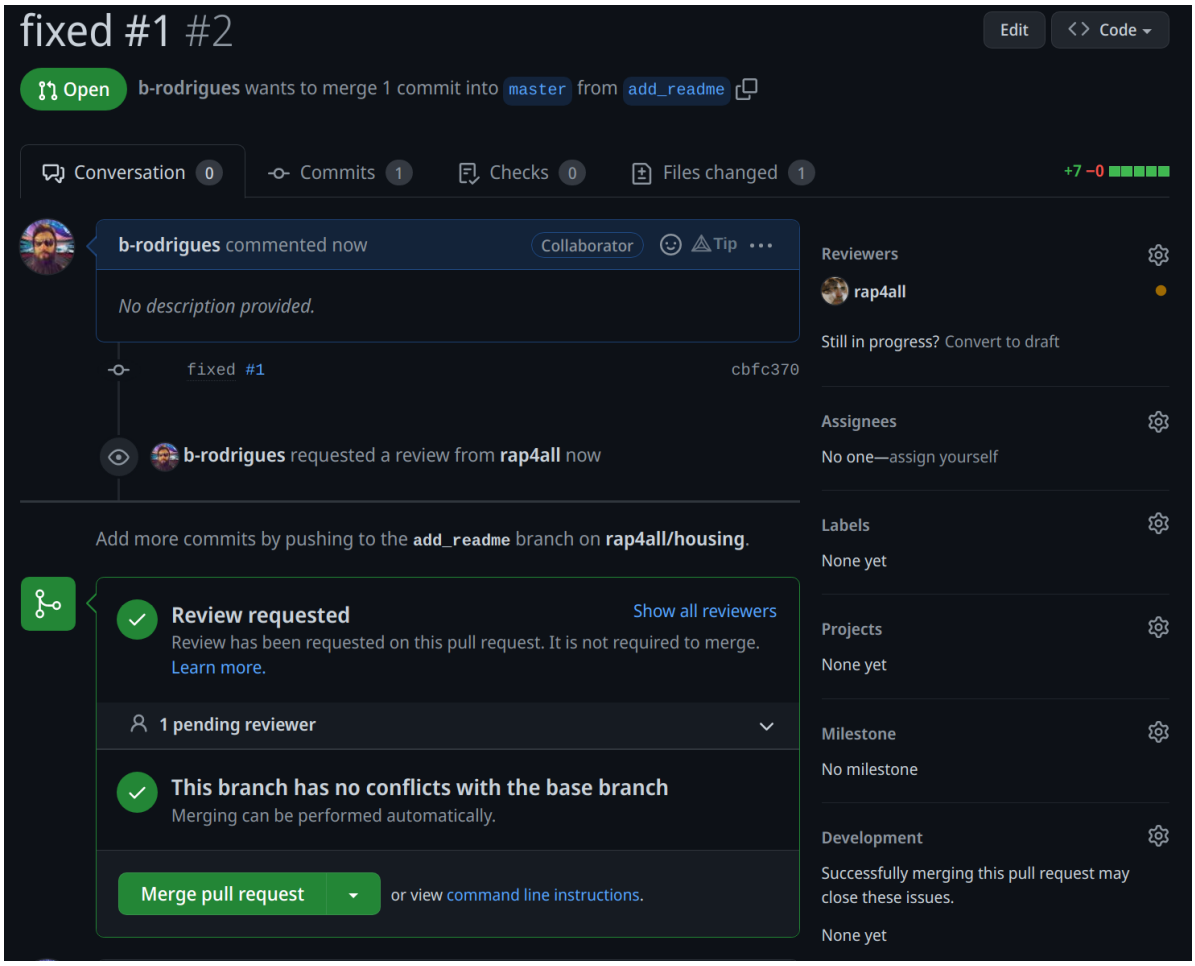


Figure 4.4: Github tells us that this branch can safely be merged

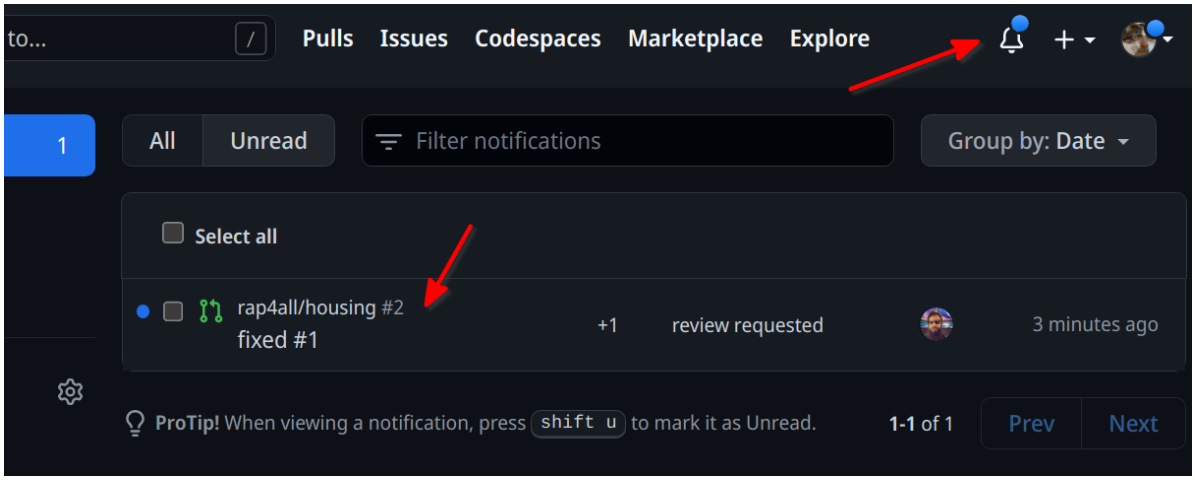


Figure 4.5: The owner was notified to review the pull request

Let's go ahead and add a rule that each pull request has to be approved. By clicking on "Add rule", the following screen appears:

By clicking the first option, more options appear:

By choosing these options, the owner can basically enforce trunk-based development (well, collaborators still have to submit pull requests frequently enough though, because if they don't, we can be in a situation where merging can be very difficult).

Let's choose one last option: by scrolling down, it's possible to select the option "Do not allow bypassing the above settings". This makes sure that even administrations (the owners of the project) must abide by the same rules.

Let's go back to the pull request. We can see now that the review is required:

So now the owner actually has to go and see the files that were changed:

It's possible to add comments to single lines if needed:

By clicking on the plus sign, a box appears and it's possible to leave a comment. In this case, everything is fine, so the owner is going to click on the "Viewed" button:

Then, by clicking on "Review changes", it's possible to either add a general comment, approve the pull request, or request changes that must be addressed before merging. Let's go ahead and approve:

By submitting the review, the reviewer is taken back to the issue:

The reviewer can now merge the pull request by clicking on the "Merge pull request" button. Github even suggests we deleted the branch, which served its purpose:

Let's delete it (it's always possible to restore it).

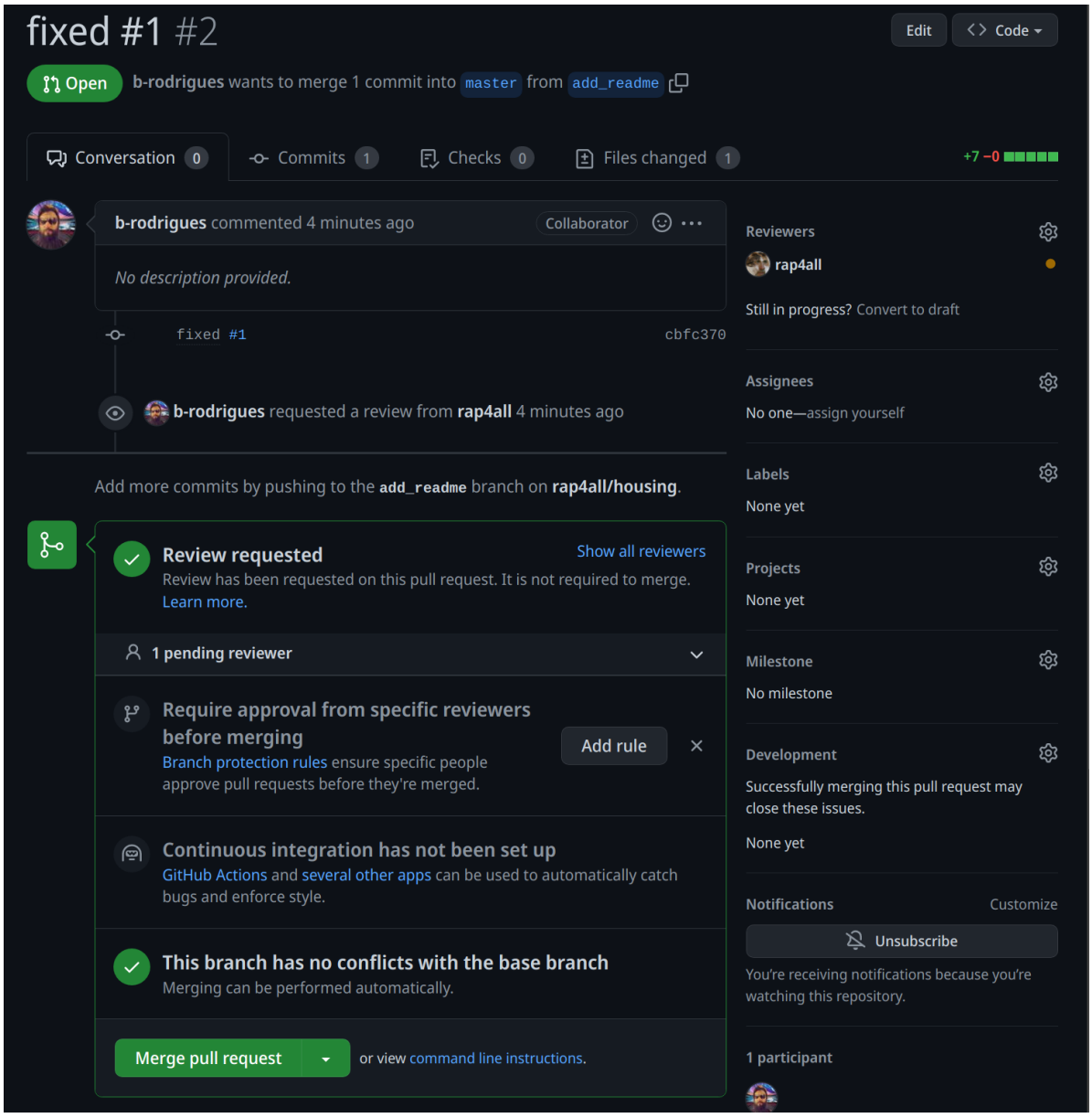


Figure 4.6: Time to review the pull request

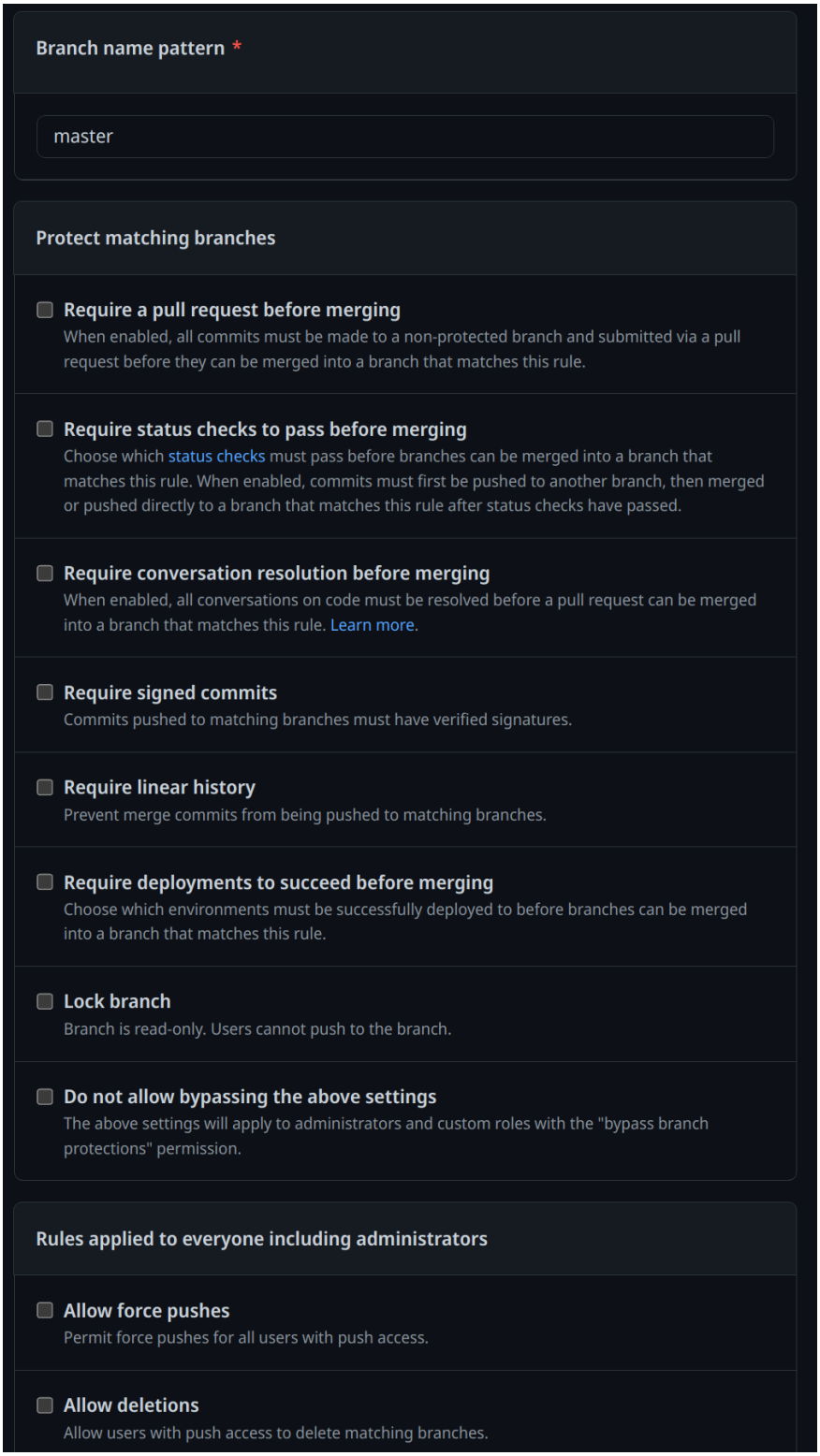


Figure 4.7: Choose how to protect the master branch

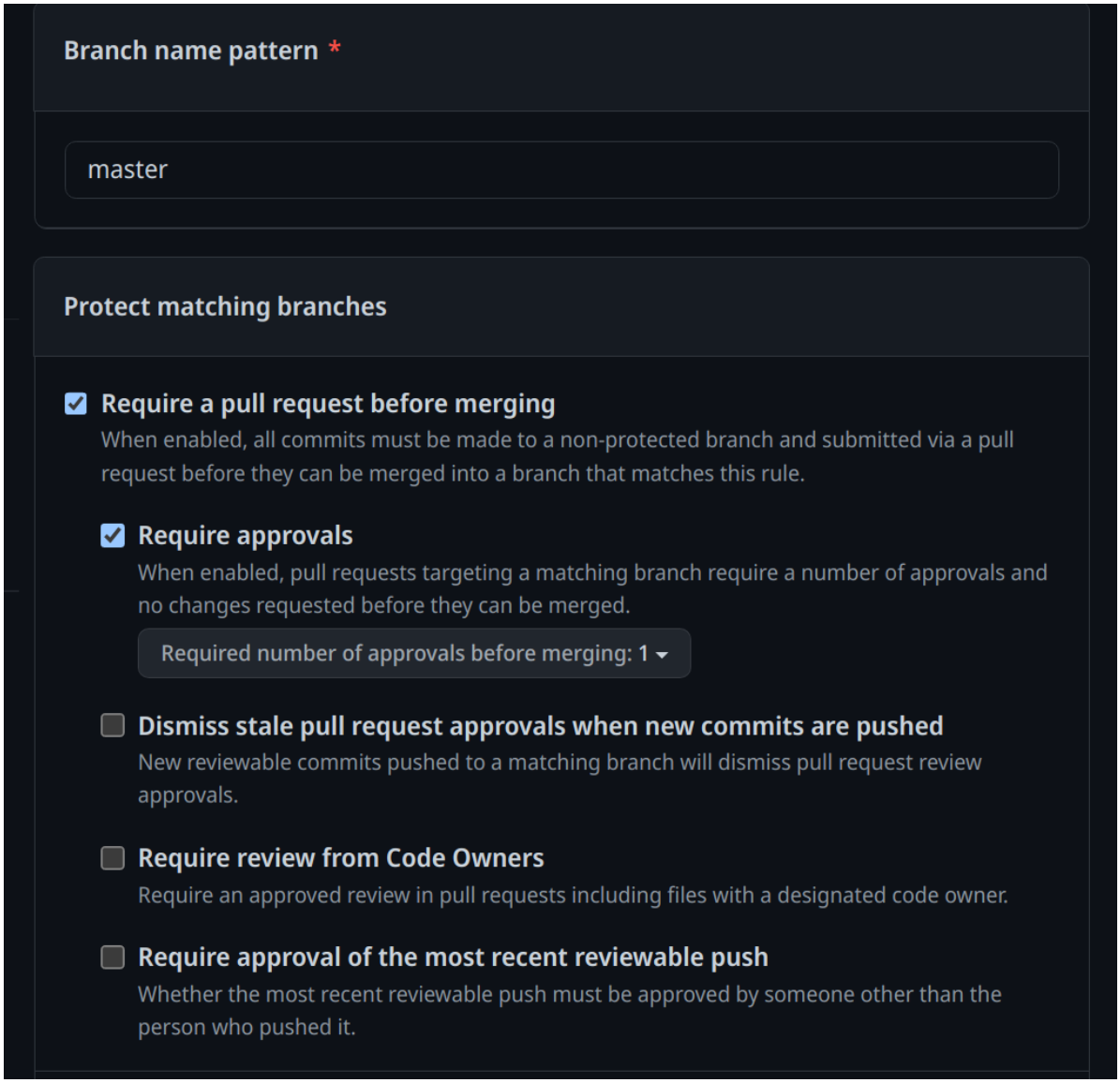


Figure 4.8: Reviews are now required

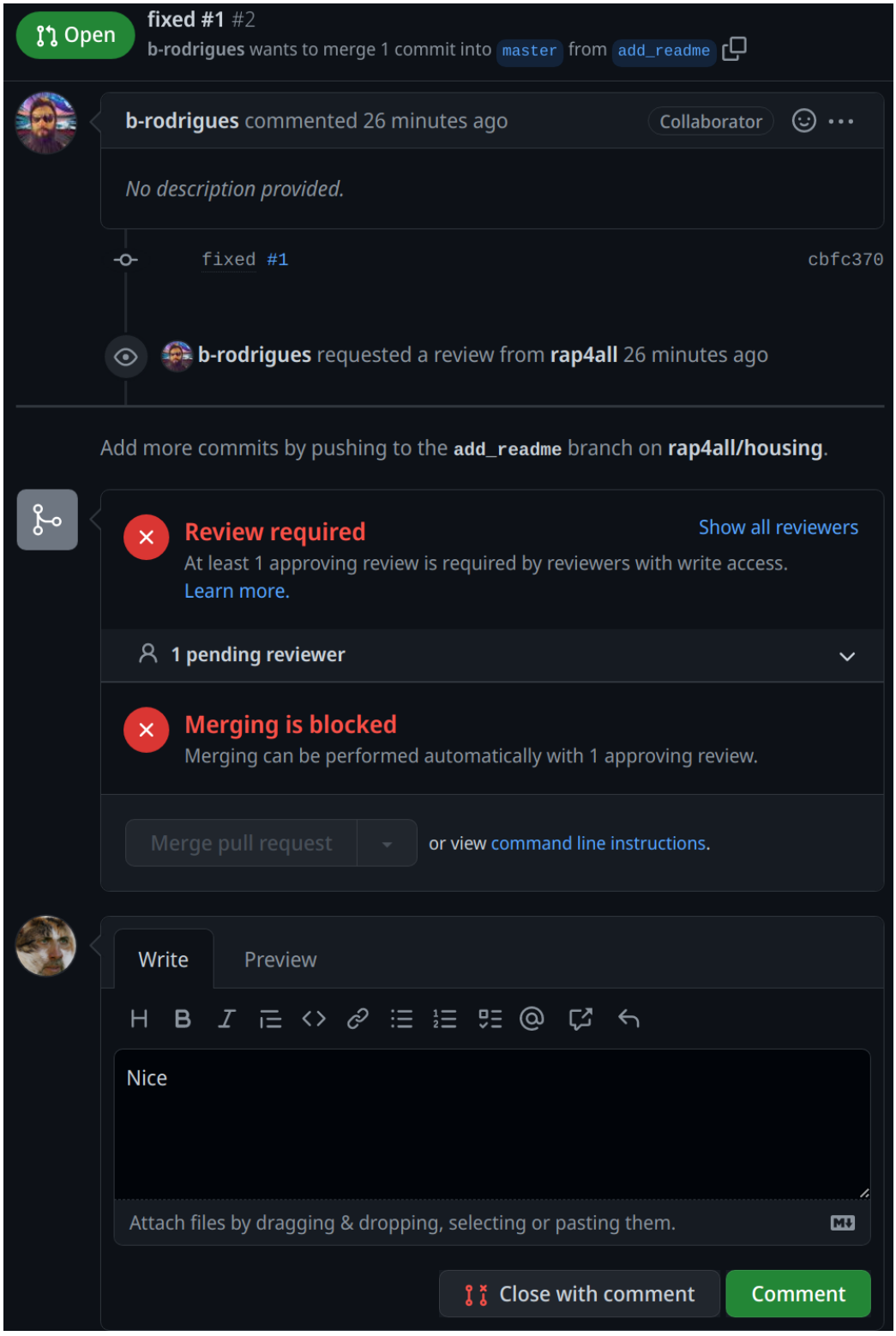


Figure 4.9: Time to review

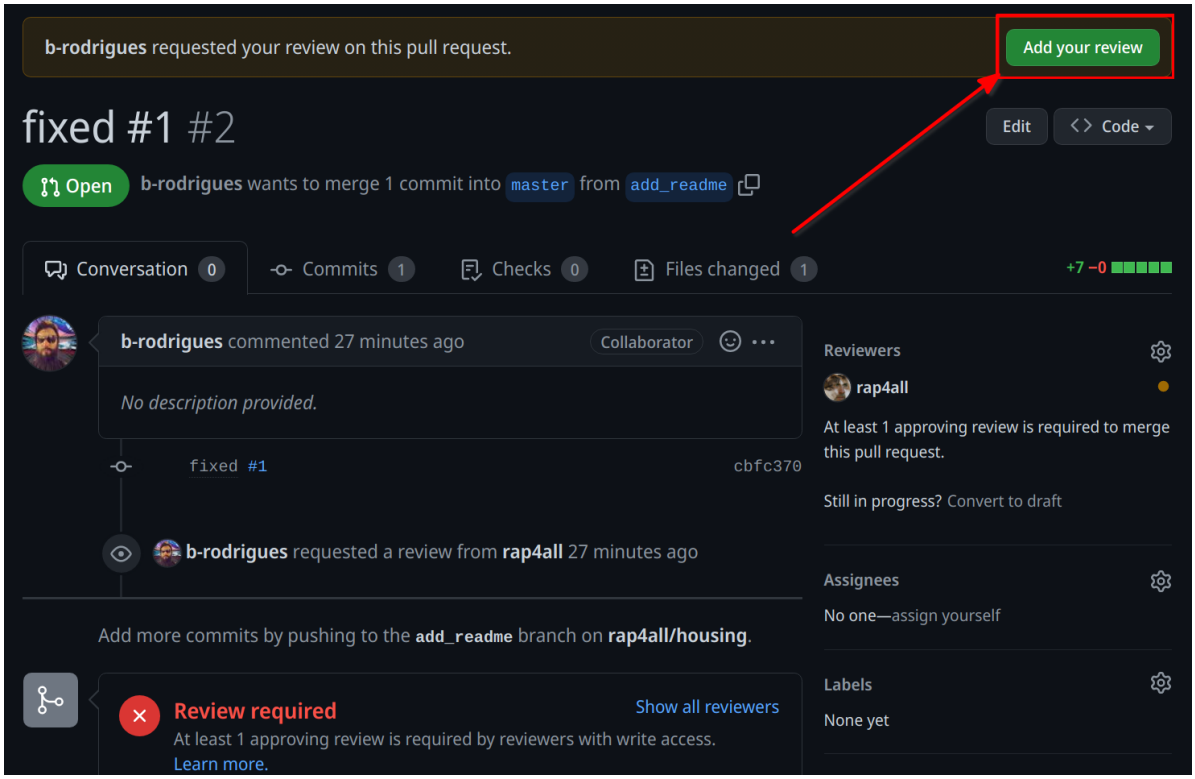


Figure 4.10: Check the code, and add comments if needed

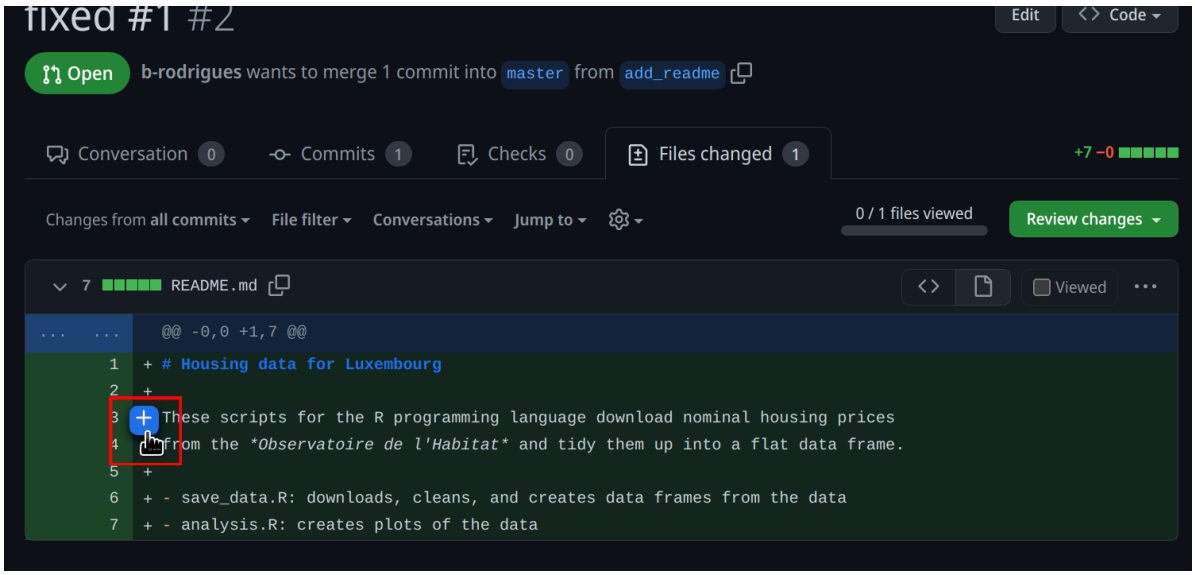


Figure 4.11: It's possible to add comments to lines

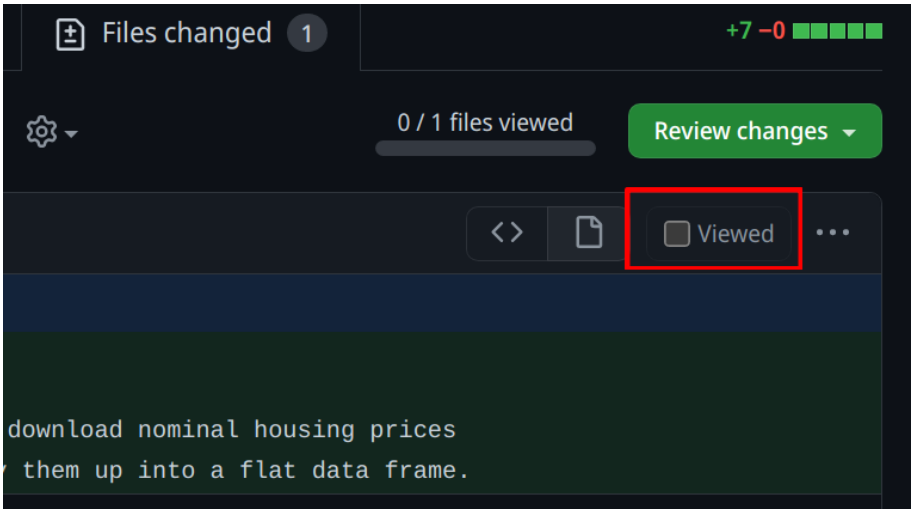


Figure 4.12: Good job!

4.1.2 Handling conflicts

As mentioned in the previous chapter, Git makes it easy to handle conflicts. Well, let's be clear; it can be very tricky sometimes to resolve conflicts. But you should know that when solving a conflict with Git is difficult, this usually means that it would be impossible to do any other way, and would inevitably result in someone having to reconcile the files by hand. What makes handling conflicts easier with Git though, is that Git is able to tell you where you can find clashes on a per-line basis. So for instance, if you change the ten first lines of a script, and I change the ten next lines, there would be no conflict, and Git will automatically merge both our contributions into a single file. Other tools, like Dropbox, would fail in a situation like this, because these tools can only handle conflicts on a per file basis. The same file was changed by two different persons? Regardless of where these changes happened, you now have a conflict to deal with on your hand… and worse, you don't even know where! You will need to scan the two resulting copies of the file by hand. Git, in the case where the same lines were changed, highlights them very clearly so that you can quickly find them and deal with the problems.

We will see all of this in the coming sections.

So how do conflicts happen in practice? Let's imagine the following scenario. Both Bruno and the project owner will create a branch, and edit the same file. Perhaps they talked over the phone and decided to add a feature or correct a bug. Perhaps they decided that it wasn't worth it to open an issue on Github and assign someone to do it. After all, they discussed this on the phone and decided that Bruno should do it. Or was it the owner who needed to solve the issue? No one remembers now. Either way, they both did, and changed the same file, so a conflict will ensue.

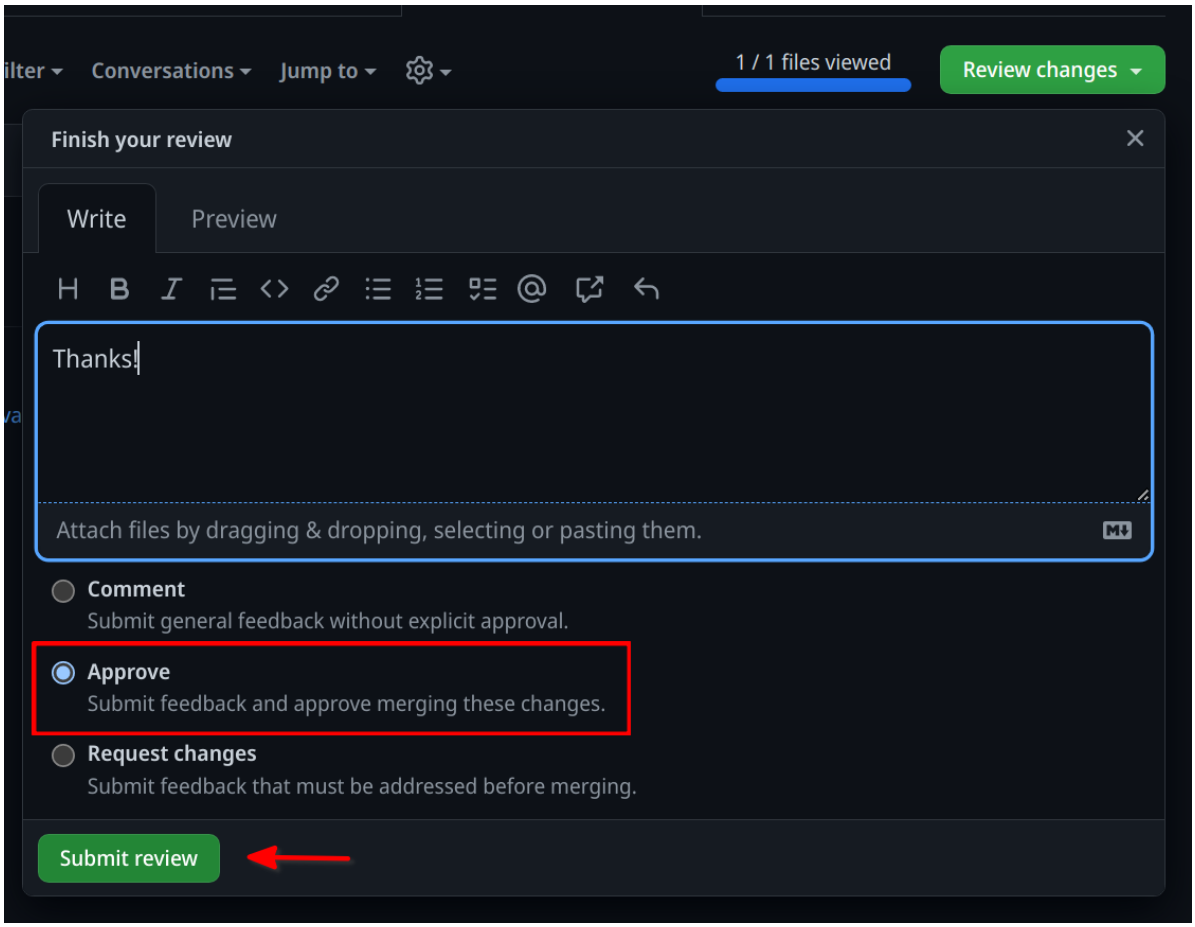


Figure 4.13: Nothing to complain about

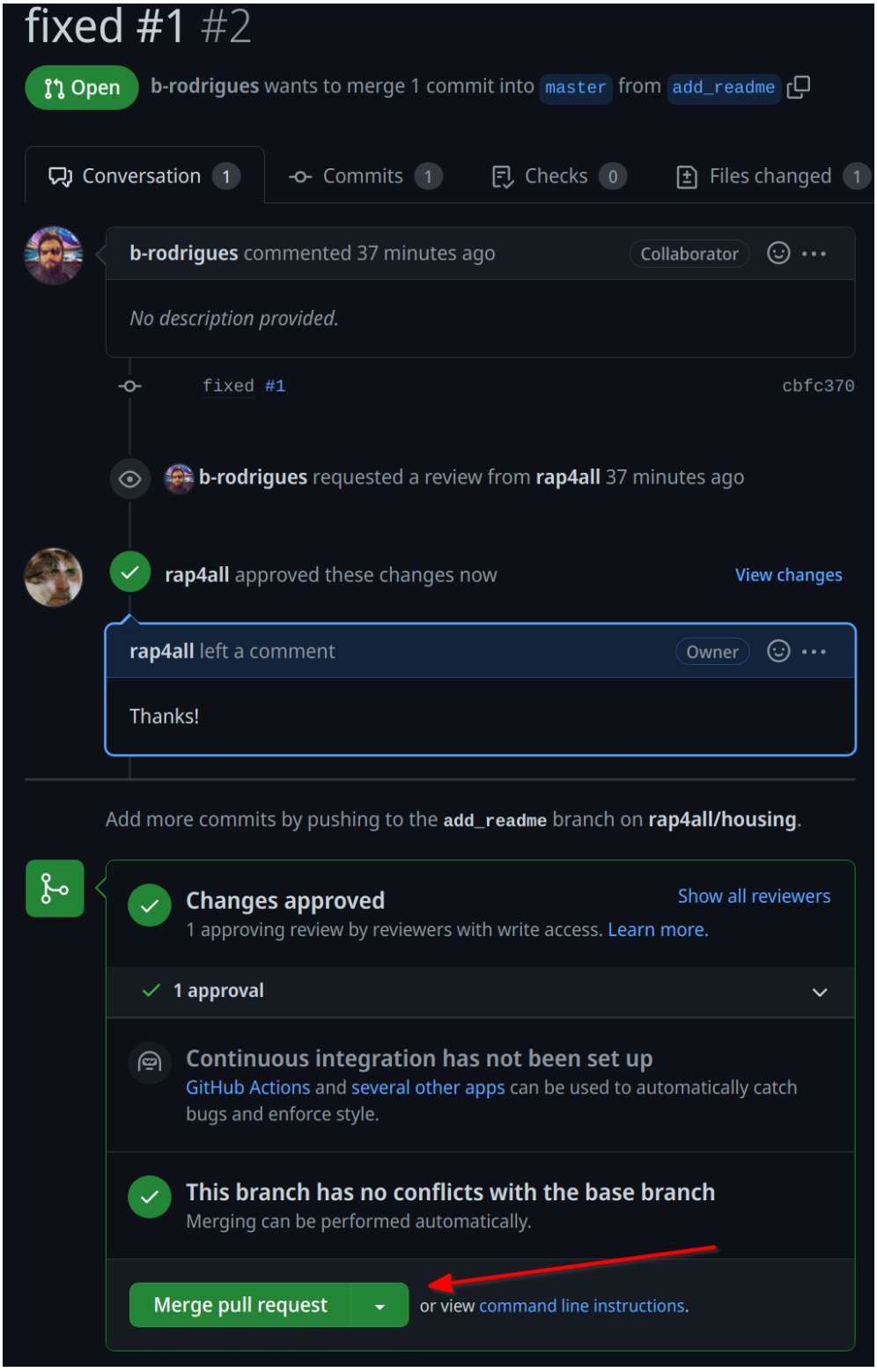


Figure 4.14: We're done, we can merge the pull request

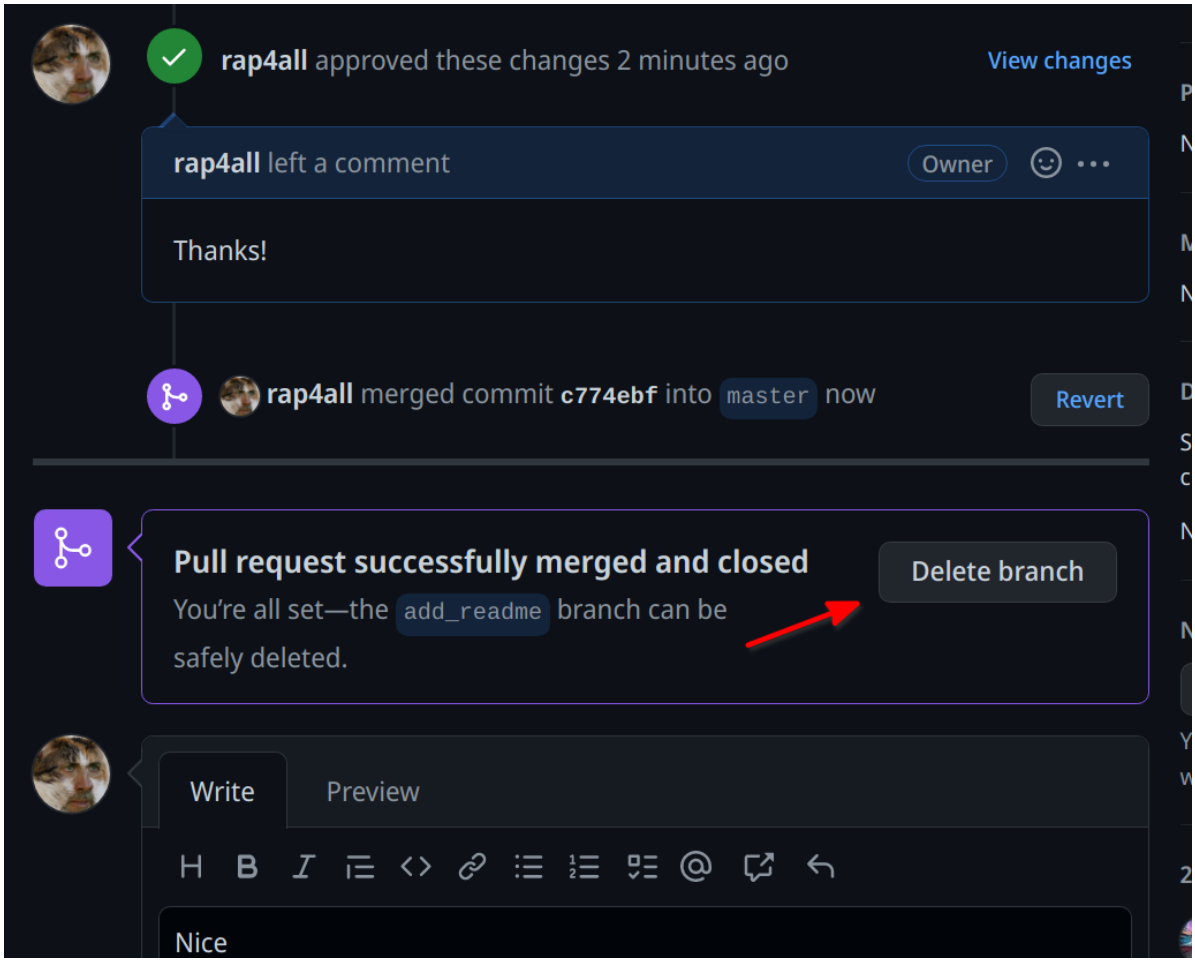


Figure 4.15: Let's get rid of this branch

First, Bruno needs to switch back to the master branch on his computer:

```
bruno@computer  git checkout master
```

```
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

Git tells us to update the code on our computer by running `git pull`. We use `git push` to upload code to Github, and use `git pull` to download code from Github. Let's run it and see what happens:

```
bruno@computer  git pull
```

```
Updating b7f82ee..c774ebf
Fast-forward
 README.md | 7 +++++++
 1 file changed, 7 insertions(+)
 create mode 100644 README.md
```

The owner of the project (called `owner`, remember?) can do the same and will see the same. Now, Bruno creates a new branch to work on the new feature:

```
bruno@computer  git checkout -b add_cool_feature
```

And the project owner also create a new branch:

```
owner@localhost  git checkout -b add_nice_feature
```

They now edit the same file, `analysis.R`. Bruno added this function:

```
make_plot <- function(country_level_data,
                      commune_level_data,
                      commune){

  filtered_data <- commune_level_data %>%
    filter(locality == commune)

  data_to_plot <- bind_rows(
```

```
    country_level_data,
    filtered_data
  )

  ggplot(data_to_plot) +
    geom_line(aes(y = pl_m2,
                  x = year,
                  group = locality,
                  colour = locality))
}
```

This way, Bruno could delete the repeating code and create plots like this:

```
lux_plot <- make_plot(country_level_data,
                      commune_level_data,
                      communes[1])


# Esch sur Alzette

esch_plot <- make_plot(country_level_data,
                       commune_level_data,
                       communes[2])

# and so on...
```

The end effect is the same, but by using this function, the code is now shorter, and clearer. Also, if someone wants to change, say, the theme of the plot, now this only needs to be changed in one place and not for each commune. Now, what did the owner change? The owner started by removing the line that loaded the `{purrr}` package, as no function from the package was used in the script, and then also changed every `%>%` to `|>`. It seems that much more than just who would make the changes got lost in translation… Anyways, both now push their changes to their respective branches. This is Bruno:

```
bruno@computer  git add .
bruno@computer  git commit -am "make_plot() for plotting"
bruno@computer  git push origin add_cool_feature


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), 647 bytes | 647.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'add_cool_feature' on GitHub by visiting:
remote:      https://github.com/rap4all/housing/pull/new/add_cool_feature
remote:
To github.com:rap4all/housing.git
 * [new branch]      add_cool_feature -> add_cool_feature
```

and this is the owner:

```
owner@localhost  git add .
owner@localhost  git commit -am "cleanup"
owner@localhost  git push origin add_sweet_feature
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 449 bytes | 449.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'add_sweet_feature' on GitHub by visiting:
remote:      https://github.com/rap4all/housing/pull/new/add_sweet_feature
remote:
To github.com:rap4all/housing.git
 * [new branch]      add_sweet_feature -> add_sweet_feature
```

So, let's think about what just happened: two developers changed the same file, `analysis.R` in two separate branches. They did what they had to do, and now these two branches need to be merged back to the trunk. So Bruno does a pull request:

First, Bruno selects the feature branch (1), then clicks on "Contribute" (2) and then "Open pull request" (3). Bruno gets taken to this screen:

Now Bruno can click on "Create pull request", but remember, because reviews are required, automatic merging is disabled.

If now we go see what happens from the project owner's side of things, first of all, there's now a notification for a pending review:

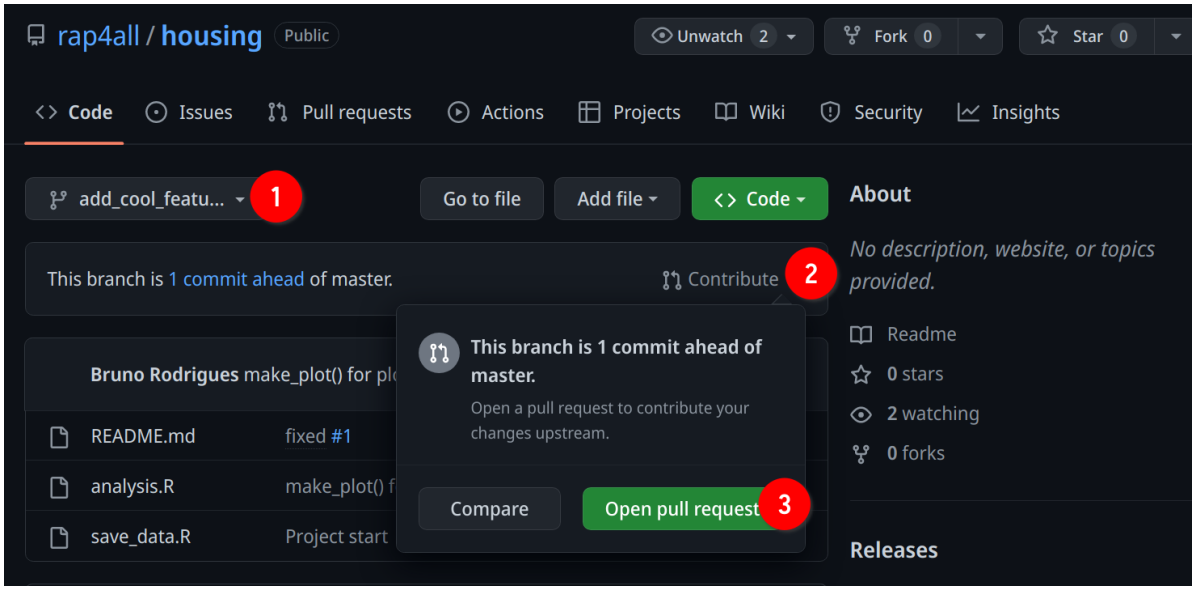


Figure 4.16: Bruno opens a pull request after finishing his changes

By clicking on it, the project owner can review the pull request and decide what to do with it. So at this point, the owner did not open a pull request for the feature he or she worked on yet. And maybe that's a good thing, because now the project owner can see that the changes that Bruno made on the file will conflict with the project owner's changes.

So how to move forward? Simple: the project owner can decide to approve the pull request, which will merge Bruno's changes into the master branch (or the trunk). Then, instead of opening a pull request for merging his or her changes into trunk, which will cause a conflict, the project owner can instead merge the changes from the trunk into his or her feature branch. This will also create a conflict, but now the project owner can easily deal with it on his or her machine, and then push a new commit with both changes integrated gracefully. The image below illustrates this workflow:

First step, the owner reviews and approves Bruno's pull request:

The pull request can get merged and Bruno's feature branch deleted. Now, it wouldn't make sense for the project owner to create a pull request to merge his or her changes. They would conflict with what Bruno did. So the project owner goes back to the computer and essentially updates the code in his or her feature branch by merging master into it.

So, the project owner checks that he or she is working on the feature branch:

```
owner@localhost  git status

On branch add_sweet_feature
```

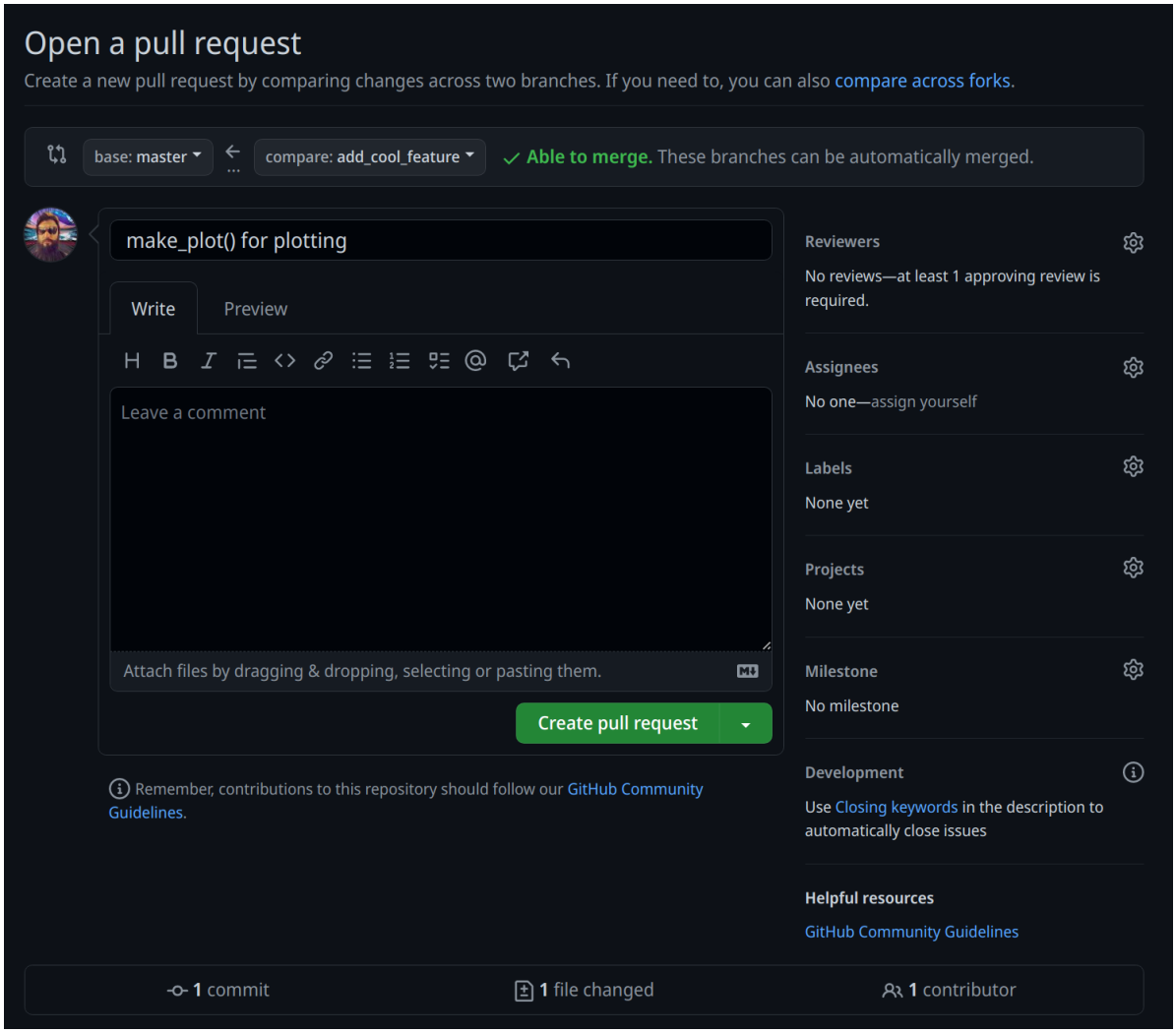


Figure 4.17: No conflicts, for now...

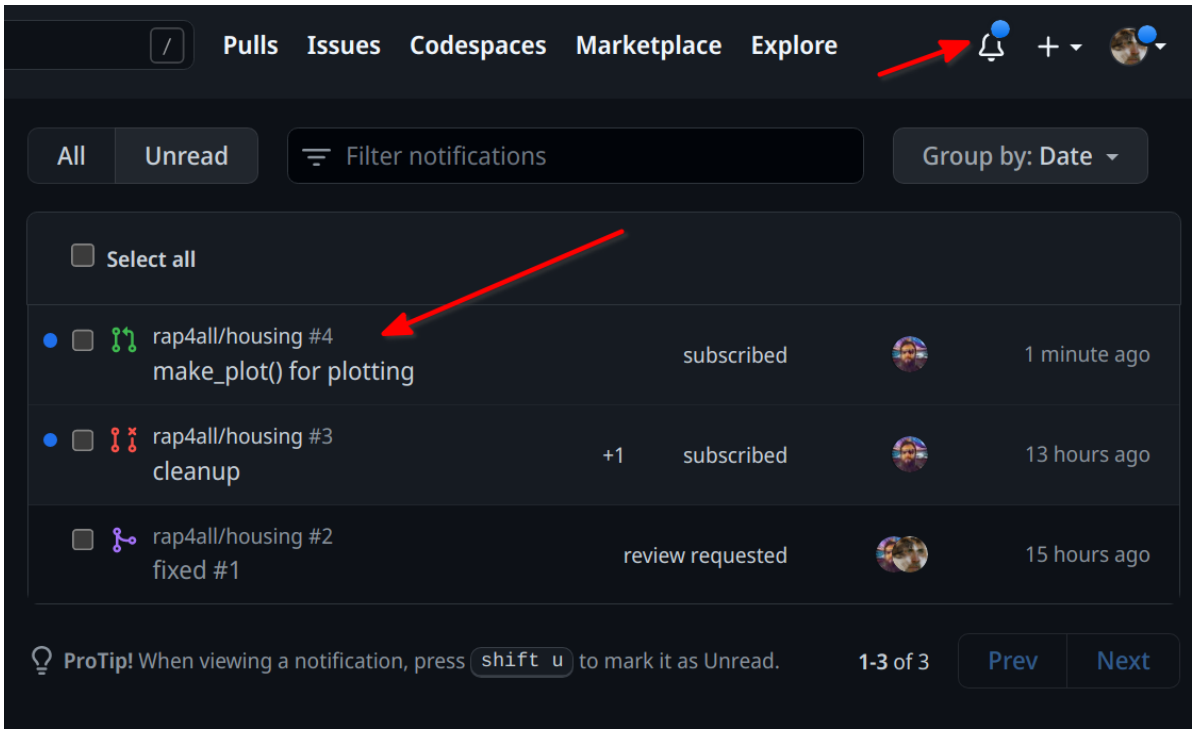


Figure 4.18: New review pending

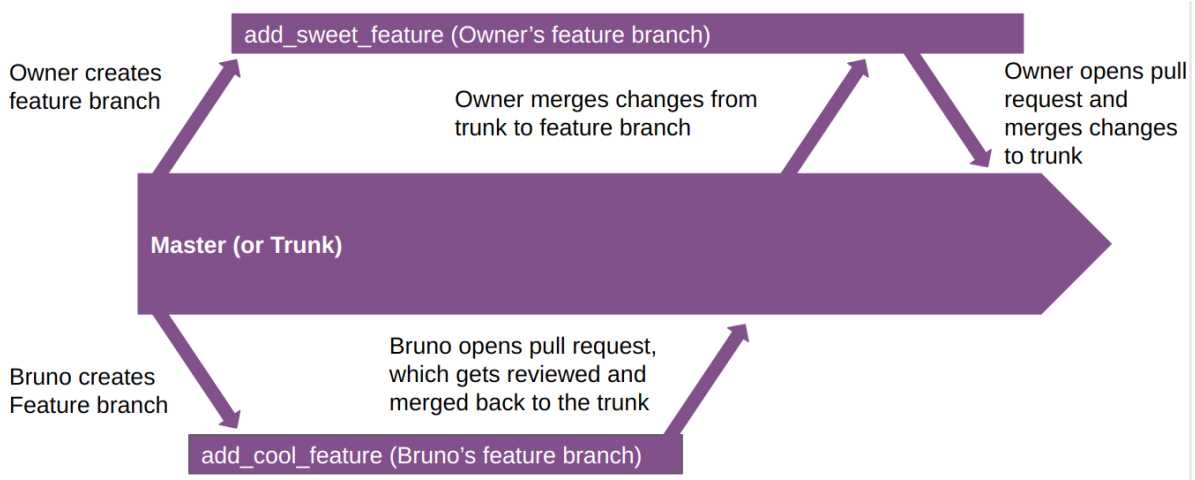


Figure 4.19: Conflict solving with trunk-based development

Figure 4.20: First, let's approve the changes

```
nothing to commit, working tree clean
```

Ok, so now let's get the updated code from master, by pulling from master:

```
owner@localhost  git pull origin master
```

The owner now sees this:

```
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (4/4), 1.23 KiB | 418.00 KiB/s, done.
From github.com:rap4all/housing
 * branch            master     -> FETCH_HEAD
   c774ebf..a43c68f  master     -> origin/master
Auto-merging analysis.R
CONFLICT (content): Merge conflict in analysis.R
Automatic merge failed; fix conflicts and then commit the result.
```

Git detect that there's some conflicts and tells the owner to fix them, and then commit the results. So let's open `analysis.R` and see how it looks like (you can view the file online on this link[1]. First of all, you will see Git deals with conflicts on a per line basis. So each line that the owner changed that does not conflict with Bruno's change gets immediately updated to reflect the owner's changes. For example, remember that the owner removed the line that loaded the `{purrr}` package? This line was also removed by pulling the changes from master into the feature branch. Also, you should notice that every `%>%` was changed into `|>` as well.

Then, you should understand what happens when a conflict gets detected on some lines. For example, this is the first conflict you should see:

```
<<<<<<< HEAD
filtered_data <- commune_level_data |>
  filter(locality == communes[1])
=======
  filtered_data <- commune_level_data %>%
    filter(locality == commune)
>>>>>>> a43c68f5596563ffca33b3729451bffc762782c3
```

[1] https://is.gd/ktWtjr

We both see how the lines look on the owner's computer and how they look in the master branch (or trunk). These are the lines between `<<<<<<< HEAD` and `=======`. The lines between `=======` and `>>>>>>> a43c68f5596563ffca33b3729451bffc762782c3` are how they look in the master branch (or trunk). This very long chain of characters that starts with `a43c68f` is the hash of the commit from which these lines come from.

So this makes things quite easy; one simply needs to remove the outdated code, and then commit and push the fixed file! The project owner only needs to remove `<<<<<<< HEAD` and `=======` and what's between these lines, as well as the lines that show the hash commit. The project owner can now commit and push the changes, open a pull request, ask Bruno to review the changes one last time and merge everything back to master.

In (1) we see the commit that deals with the conflict, in (2) the owner asks Bruno for a review and then in (3) we see that Bruno reviewed and approved. Finally, the pull request can be merged (4) and the feature branch deleted.

4.1.3 Simplified trunk-based development

The workflow that we showed here may seem a bit too rigid for smaller teams (below 4 or 5 contributors). It is possible to adopt a simplified version of trunk-based development, where contributors don't have to open pull requests to merge their feature branches into the trunk, and no reviewer is needed. In cases like this, Git forces you to pull changes if someone already merged his or her feature branch into the trunk before you could. This way, when pulling, conflicts (if any) arise at that point. It is then your responsibility to solve the conflicts (and this works just like in the previous section) and then commit and push the commits with the conflicts resolved. Another contributor who then wishes to merged his or fear feature branch into to the trunk will have to pull again, ensuring that conflicts get resolved before he or she can merge. If no conflicts arise (for example, you both worked on different files, or on different lines of the same files), then no resolution is needed and the feature branch can be merged into master.

4.1.4 Conclusion

The main ideas of trunk-based development are:

- Each contributor opens a new branch to develop a feature or fix a bug, and works alone on his or her own little branch;
- At the end of the day at the latest (or a previously agreed upon duration), branches need all to get merged;
- Conflicts need to be taken care of at that point;

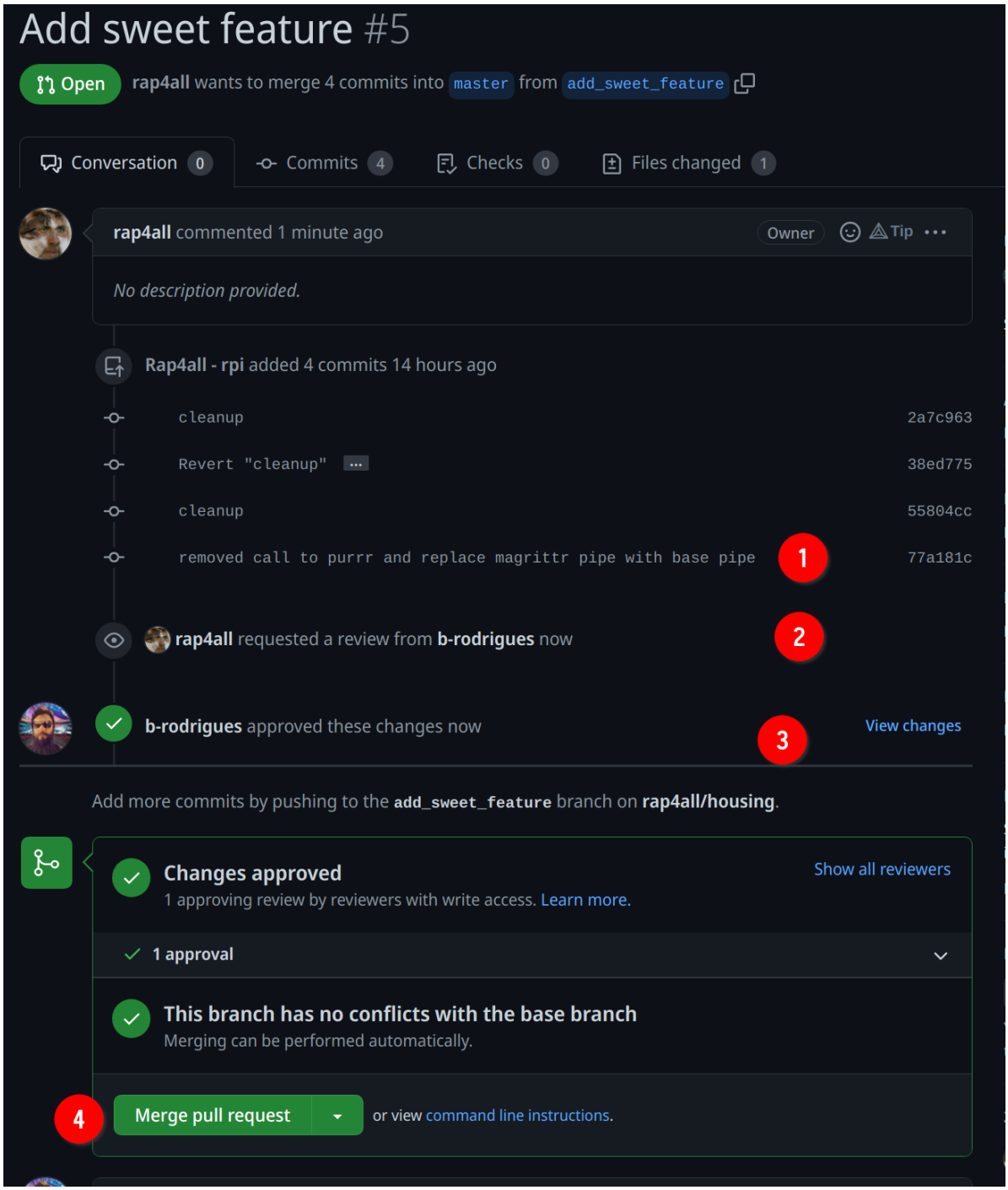


Figure 4.21: The conflict has been gracefully solved

- If adding a feature would take more time than just one day, then the task needs to be split in a manner that small contributions can be merged daily. In the beginning, these contributions can be simple placeholders that will be gradually enriched with functioning code until the feature is successfully implemented. This strategy is called branching by abstraction;
- The master branch (or trunk) contains always working, production-grade code;
- To enforce discipline, it might be worth it to make opening pull requests mandatory for merging back to the trunk, and require a review.

4.2 Contributing to public repositories

In this last section, we are going to briefly discuss how to contribute to a project when we are not a team member of that project. For example, maybe we use an R package and notice a bug, and want to propose a fix. Or maybe we simply spotted a typo in the README of said package, and want to propose a correction. Whatever it may be, if the repository is public, anyone can propose a fix. For example, consider this repository:

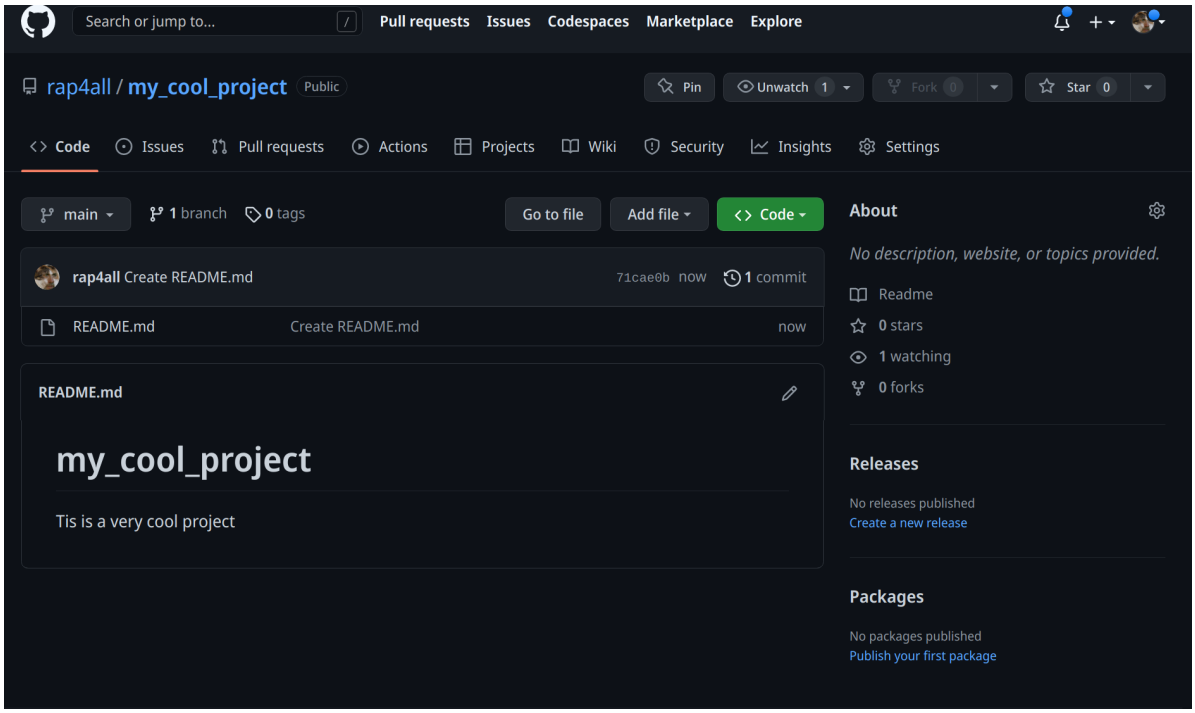


Figure 4.22: A public repository

This repository contains code written by a fellow called “rap4all”, and Bruno uses this code daily. However, Bruno notices a typo in the readme, and wants to propose a fix.

First, Bruno visits the repository on Github (since it's a public repository, anyone can view it online) and creates a fork:

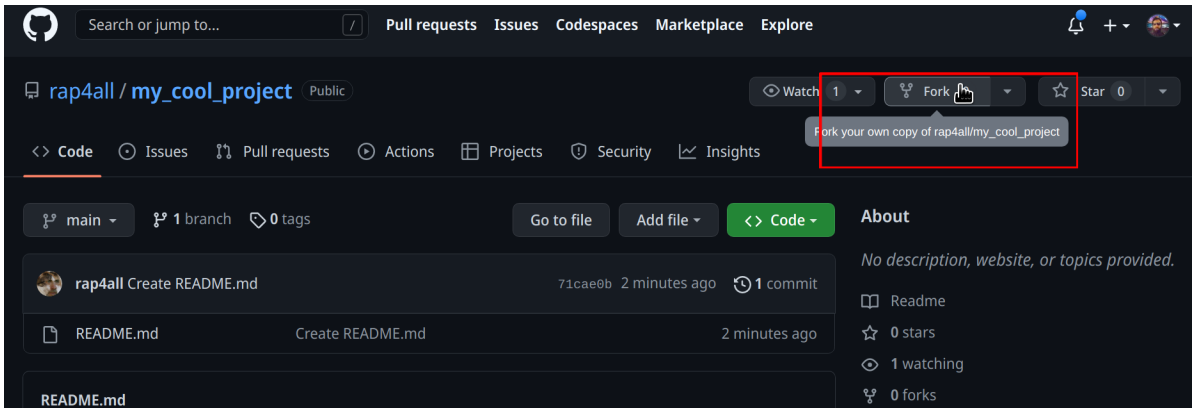


Figure 4.23: Bruno needs to create a fork of the repository

Forking creates a copy of the repository to Bruno's account:

Bruno now sees the fork on his account as well:

So now, Bruno can clone this repository and work on it, because he is working on a copy of the repository that he owns. Anything Bruno does on this copy will not affect the original repository.

```
bruno@computer  git clone git@github.com:b-rodrigues/my_cool_project.git
```

Bruno now fixes the typo in the `README.md` file, commits and pushes to his fork:

As you can see, Bruno's fork is now ahead of the original repo by one commit. By clicking on "Contribute", Bruno can open a pull request to propose his fix to the original repository. This pull request will be opened over at the original repository:

What does the owner of the original repository, "rap4all" see? The pull request Bruno opened is now in the original repository's "Pull request" menu, and the owner can check what the contribution is, if it breaks code or not, etc. This is essentially the same workflow as the one presented before in trunk-based development with pull requests and reviews before merging (minus the forking of the repository).

By merging the fix, the owner can now benefit from a grammatically correct Readme file as well:

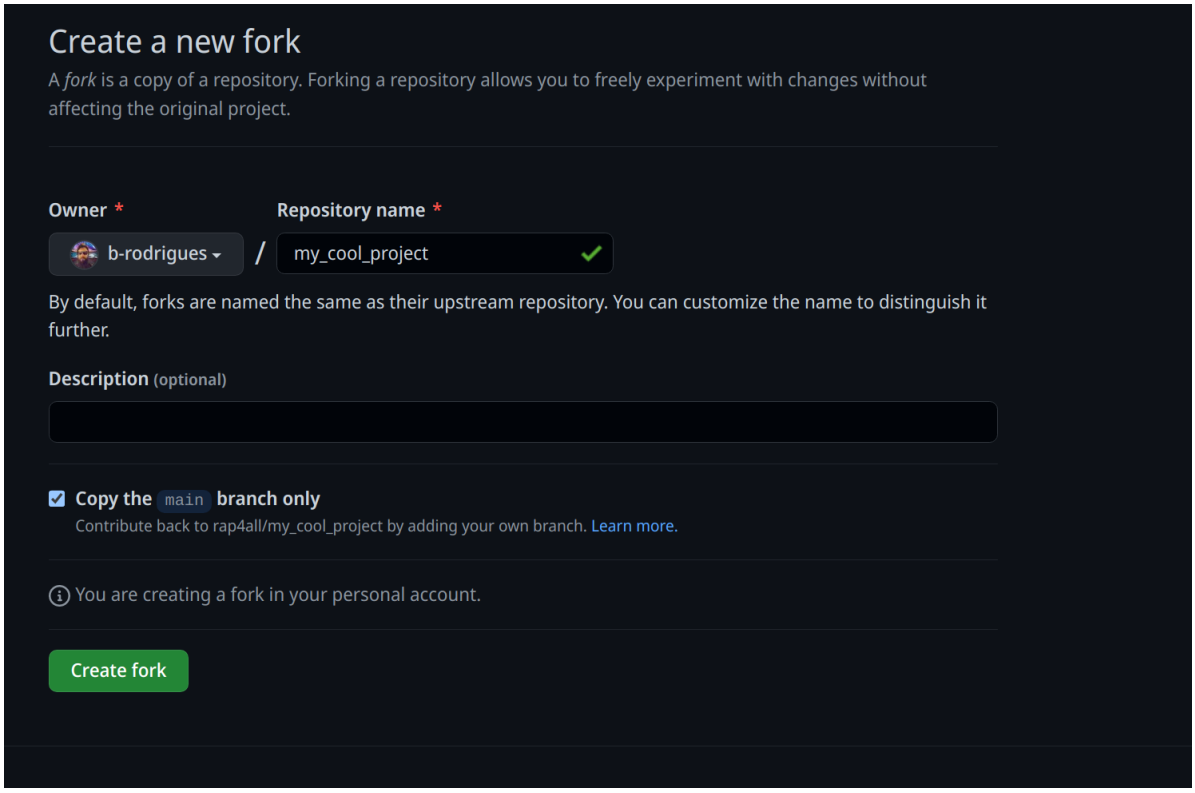


Figure 4.24: Bruno goes ahead with forking

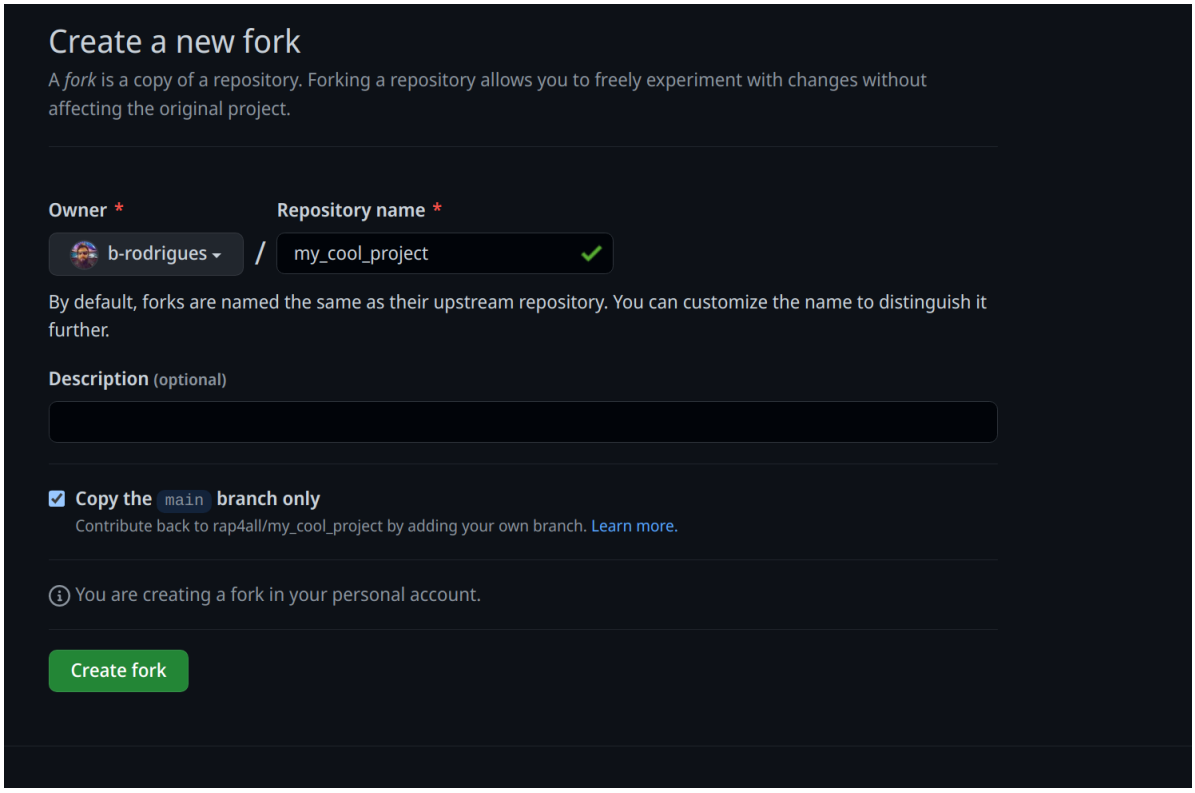


Figure 4.25: Bruno's fork

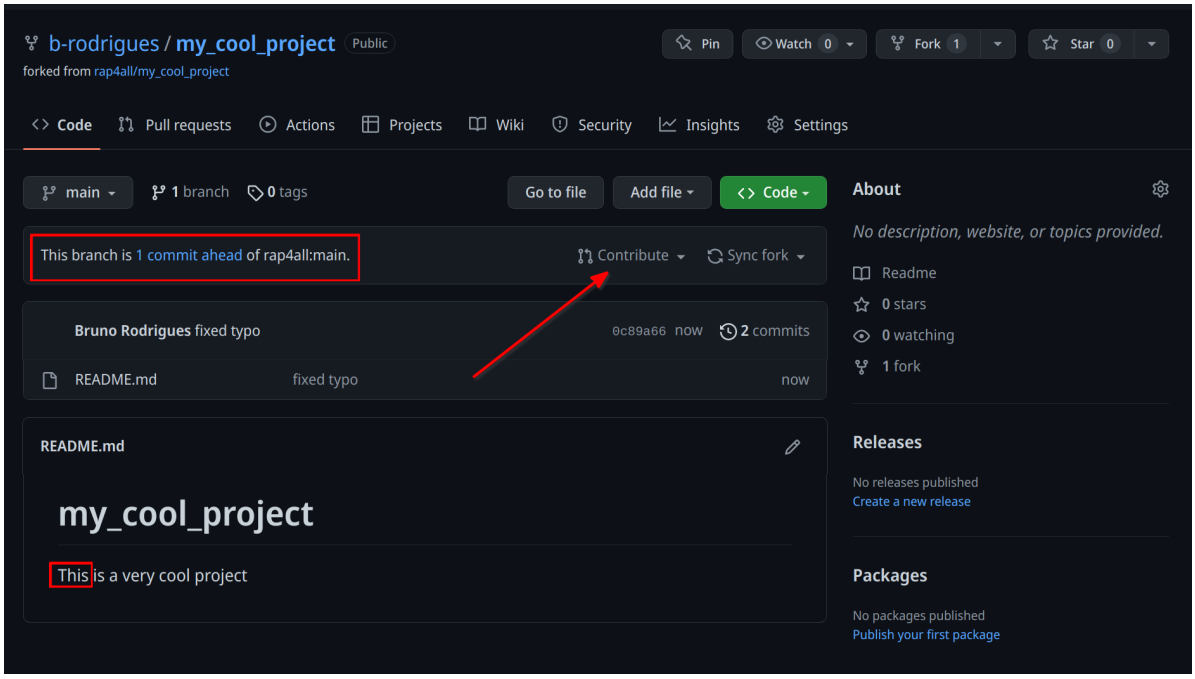


Figure 4.26: Bruno fixed the typo in his fork

4.3 Further reading

To know everything about trunk-based development, check out Hammant (2020). A free, online, version of the book is available at https://trunkbaseddevelopment.com/.

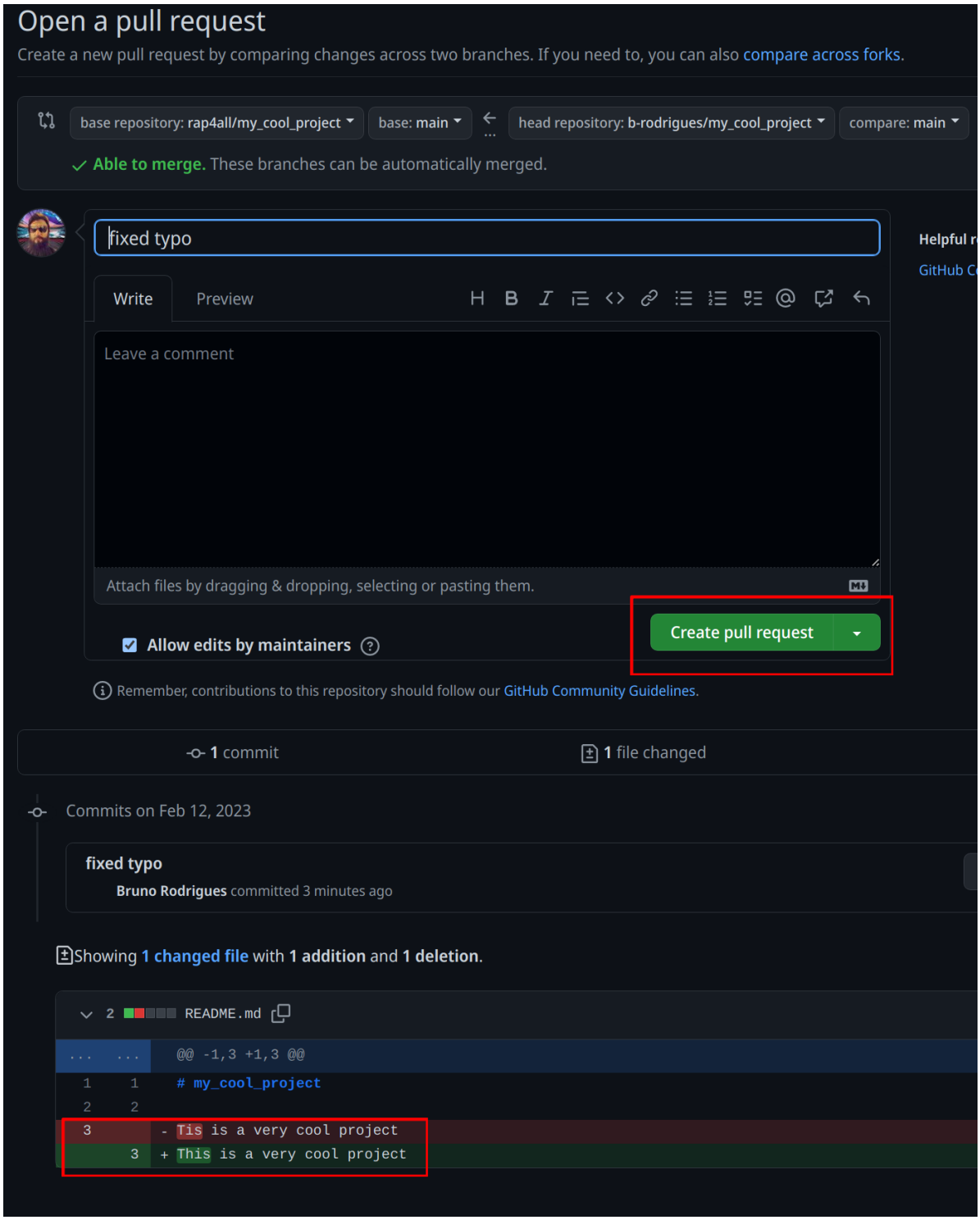


Figure 4.27: Bruno opens a pull request to contribute his fix upstream

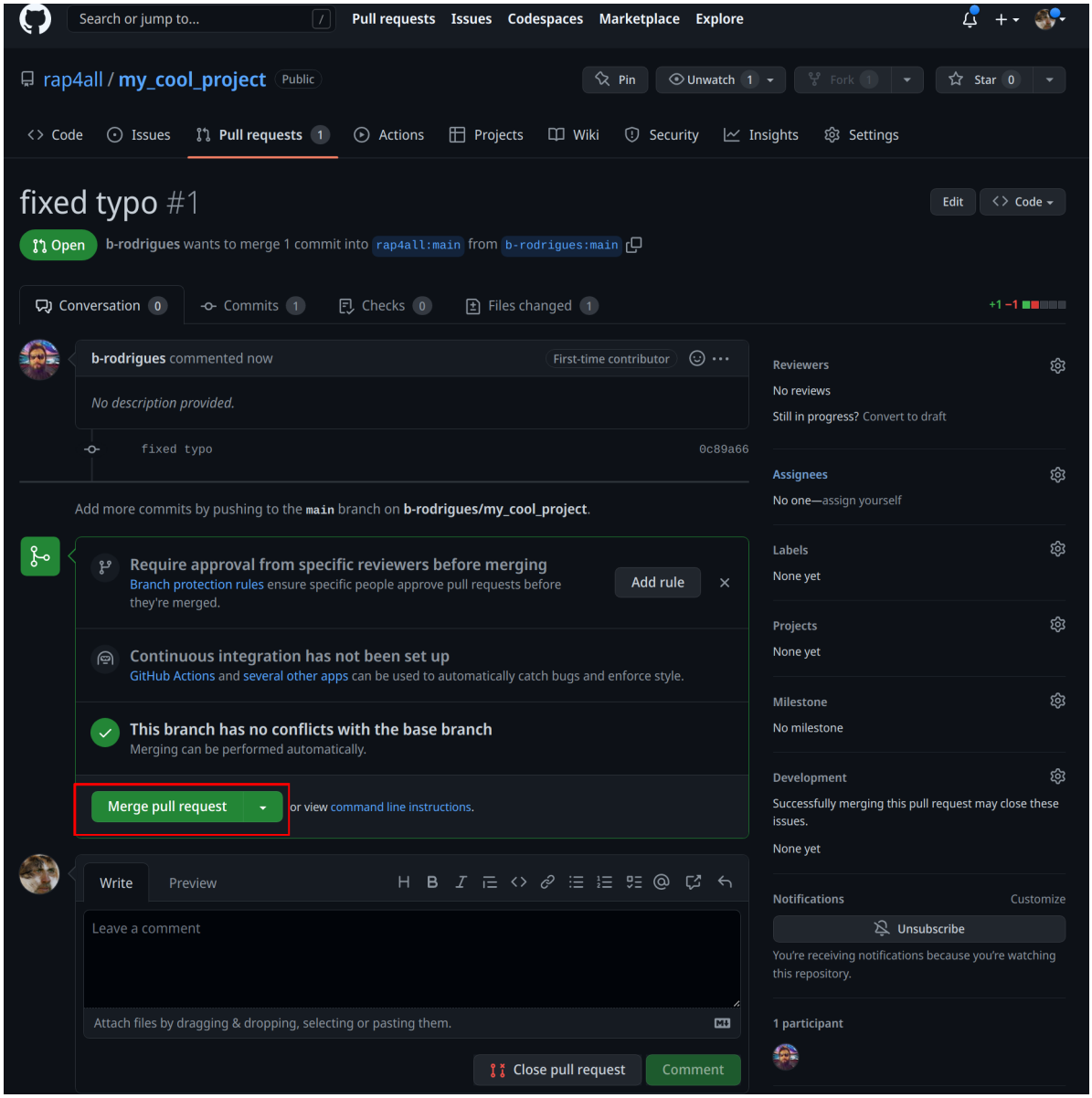


Figure 4.28: The owner of the original repository can now accept Bruno's fix

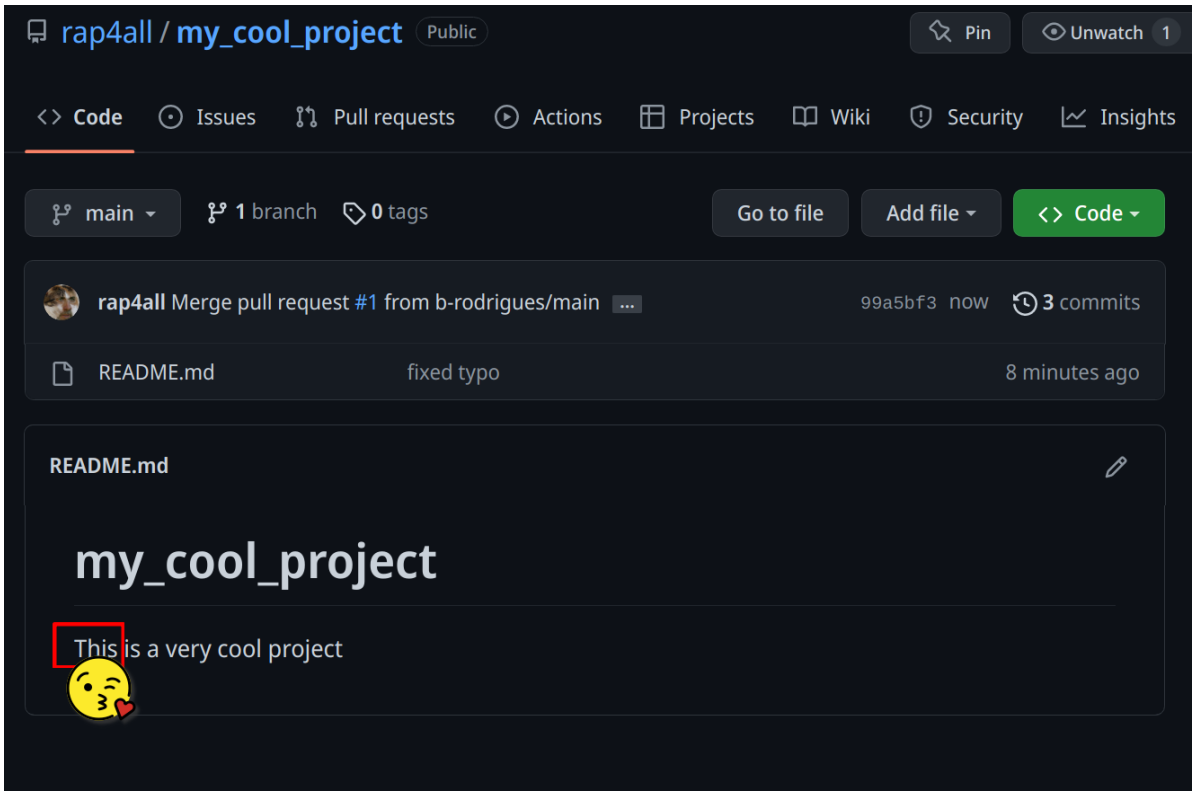


Figure 4.29: The beauty of open source

5 Functional programming

Now that we are familiar with Git and Github, we can start with writing code. We will learn how to write code using the functional programming paradigm.

This chapter will teach you the fundamentals of functional programming. *Functional programming* might sound scary, but we will focus on only a handful of concepts that are quite accessible while providing many benefits. Using these functional programming concepts will make your code more reliable, easier to test, document, share, and ultimately rerun.

5.1 Introduction

Remember that the philosophy of part one of this book is "don't repeat yourself". In this chapter we will see how we can reduce the amount of code as much as possible. In the previous chapter we've seen how Bruno was able to get rid of many lines of code (that were all the same) by writing a single function:

```
make_plot <- function(country_level_data,
                      commune_level_data,
                      commune){

  filtered_data <- commune_level_data %>%
    filter(locality == commune)

  data_to_plot <- bind_rows(
    country_level_data,
    filtered_data
  )

  ggplot(data_to_plot) +
    geom_line(aes(y = pl_m2,
                  x = year,
                  group = locality,
                  colour = locality))
}
```

Now we are going to go one step further and not only learn how to write good functions, but also how we can push the concept of "not repeating oneself" to the extreme by using higher-order fuctions and function factories.

You are very likely already familiar with at least two elements of functional programming: functions and lists. But functional programming is a complete programming paradigm, so using functional programming is more than simply using functions and lists (which you can use with other programming paradigms as well). Programming paradagims are ways to structure programs (or scripts).

Functional programming is a paradigm that relies exclusively on the evaluation of functions to achieve the desired result. If you have already written your own functions in the past, what follows will not be very new. But in order to write a good functional program, the functions that you write and evaluate have to have certain properties. Before discussing these properties, let's start with *state*.

5.1.1 The state of your program

Let's suppose that you start a fresh R session, and immediately run this line:

```
ls()
```

If you did not modify any of R's configuration files that get automatically loaded on startup, you should see the following:

```
character(0)
```

Let's suppose that now you load some data:

```
data(mtcars)
```

and define a variable `a`:

```
a <- 1
```

Running `ls()` now shows the following:

```
[1] "a"      "mtcars"
```

You have just altered the state of your program. You can think of the *state* as a box that holds everything that gets defined by the user and is accessible at any time. Let's now define a simple function that prints a sentence:

```
f <- function(name){
  print(paste0(name, " likes lasagna"))
}

f("Bruno")
```

and here's the output:

```
[1] "Bruno likes lasagna"
```

Let's run `ls()` again:

```
[1] "a"      "f"      "mtcars"
```

Function `f()` is now listed there as well. This function has two nice properties:

- For a given input, it always returns exactly the same output. So `f("Bruno")` will always return "Bruno likes lasagna".
- When running this function, the state of the program does not get altered in any way.

5.1.2 Predictable functions

Let's now define another function called `g()`, which does not have the same properties as `f()`. First, let's define a function which does not always return the same output given a particular input:

```
g <- function(name){
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)
  print(paste0(name, " likes ", food))
}
```

For the same input, "Bruno", this function now produces (potentially) a different output:

```
g("Bruno")
[1] "Bruno likes lasagna"
```

```
g("Bruno")
[1] "Bruno likes feijoada"
```

And now let's consider function `h()` that modifies the state of the program:

```
h <- function(name){
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

  if(exists("food_list")){
    food_list <<- append(food_list, food)
  } else {
    food_list <<- append(list(), food)
  }

  print(paste0(name, " likes ", food))
}
```

This function uses the `<<-` operator. This operator saves definitions that are made inside the body of functions[^The body of a function are all the instructions that go between the curly braces.] in the global environment. Before calling this function, run `ls()` again. You should see the same objects as before, plus the new functions we've defined:

```
[1] "a"         "f"          "g"         "h"         "mtcars"
```

Let's now run `h()` once:

```
h("Bruno")
[1] "Bruno likes feijoada"
```

And now `ls()` again:

```
[1] "a"         "f"         "food_list" "g"         "h"         "mtcars"
```

Running `h()` did two things: it printed the message, but also created a variable called "food_list" in the global environment with the following contents:

```
food_list
```

```
[[1]]
[1] "feijoada"
```

Let's run `h()` again:

```
h("Bruno")
[1] "Bruno likes cassoulet"
```

and let's check the contents of "food_list":

```
food_list
```

```
[[1]]
[1] "feijoada"

[[2]]
[1] "cassoulet"
```

If you keep running `h()`, this list will continue growing. Let me just say that I hesitated to show you this; this is because if you didn't know `<<-`, you might find the example above useful. But while useful, it is quite dangerous as well. Generally, we want to avoid using functions that change the state as much as possible because these functions are unpredictable, especially if randomness is involved. It is much safer to define `h()` like this instead:

```
h <- function(name, food_list = list()){

  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

  food_list <- append(food_list, food)

  print(paste0(name, " likes ", food))

  food_list
}
```

The difference now is that we made `food_list` the second argument of the function. Also, we defined it as being optional by writing:

```
food_list = list()
```

This means that if we omit this argument, the empty list will get used by default. This avoids the users from having to manually specify it.

We can call it like this:

```
food_list <- h("Bruno", food_list) # since food_list is already defined,
                                   # we don't need to start with an empty list
```

```
[1] "Bruno likes feijoada"
```

We save the output back to `food_list`. Let's now check its contents:

```
food_list
```

```
[[1]]
[1] "feijoada"

[[2]]
[1] "cassoulet"

[[3]]
[1] "feijoada"
```

The only thing that we need now to deal with is the fact that the food item gets chosen randomly. I'm going to show you the simple way of dealing with this, but later in this chapter we are going to use the `{withr}` package for situations like this. Let's redefine `h()` one last time:

```
h <- function(name, food_list = list(), seed = 123){

  # We set the seed, making sure that we get the same
  # selection of food for a given seed

  set.seed(seed)
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

  # We now need to unset the seed, because if we don't,
  # guess what, the seed will stay set for the whole session!

  set.seed(NULL)

  food_list <- append(food_list, food)

  print(paste0(name, " likes ", food))

  food_list
}
```

Let's now call `h()` several times with its default arguments:

```
h("Bruno")
```

```
[1] "Bruno likes feijoada"
[[1]]
[1] "feijoada"
```

```
h("Bruno")
```

```
[1] "Bruno likes feijoada"
[[1]]
[1] "feijoada"
```

```
h("Bruno")
```

```
[1] "Bruno likes feijoada"
[[1]]
[1] "feijoada"
```

As you can see, every time this function runs, it now outputs the same result. Users can change the seed to have this function output, consistently, another result.

5.1.3 Referentially transparent and pure functions

A referentially transparent function is a function that does not use any variable that is not also one of its inputs. For example, the following function:

```
bad <- function(x){
  x + y
}
```

is not referentially transparent, because `y` is not one of the function's inputs. What happens if you run `bad()` is that `bad()` needs to look for `y`. Because `y` is not one of its inputs, `bad()` then looks for it in the global environment. If `y` is defined there, it then gets used. Defining and using such functions must be avoided at all costs because these functions are unpredictable. For example:

```
y <- 10

bad <- function(x){
  x + y
}
```

```
bad(5)
```

This will return `15`. But if `y <- 45` then `bad(5)` would this time around return `50`. It is much safer, and clearer to make `y` an explicit input of the function instead of having to keep track of `y`'s value (and it's so easy to do, why just not do it):

```
good <- function(x, y){
  x + y
}
```

`good()` is a referentially transparent function; it is much safer than `bad()`. `good()` is also a pure function, because it's a function that does not interact in any way with the global environment. It does not write anything to the global environment, nor requires anything from the global environment. Function `h()` from the previous section was not pure, because it created an object and wrote it to the global environment (the `food_list` object). Turns out that pure functions are thus necessarily referentially transparent.

So the first lesson in your functional programming journey that you have to remember is to only use pure functions.

5.2 Writing good functions

5.2.1 Functions are first-class objects

In a functional programming language, functions are first-class objects. Contrary to what the name implies, this means that functions, especially the ones you define yourself, are nothing special. A function is an object like any other, and can thus be manipulated as such. Think of anything that you can do with any object in R, and you can do the same thing with a function. For example, let's consider the `+()` function. It takes two numeric objects and returns their sum:

```
1 + 5.3
```

```
[1] 6.3
```

```
# or alternatively: `+`(1, 5.3)
```

You can replace the numbers with functions that return numbers:

```
sqrt(1) + log(5.3)
```

```
[1] 2.667707
```

It's also possible to define a function that explicitly takes another function as an input:

```
h <- function(number, f){
  f(number)
}
```

You can call then use `h()` as a wrapper for `f()`:

```
h(4, sqrt)
```

```
[1] 2
```

```
h(10, log10)
```

```
[1] 1
```

Because `h()` takes another function as an argument, `h()` is called a higher-order function.

If you don't know how many arguments `f()`, the function you're wrapping, has, you can use the `...`:

```
h <- function(number, f, ...){
  f(number, ...)
}
```

`...` are simply a placeholder for any potential additional argument that `f()` might have:

```
h(c(1, 2, NA, 3), mean, na.rm = TRUE)
```

```
[1] 2
```

```
h(c(1, 2, NA, 3), mean, na.rm = FALSE)
```

```
[1] NA
```

`na.rm` is an argument of `mean()`. As the developer of `h()`, I don't necessarily know what `f()` might be, or maybe I know `f()` and know all its arguments, but don't want to have to rewrite them all to make them arguments of `h()`, so I can use `...` instead. The following is also possible:

```
w <- function(...){
paste0("First argument: ", ..1, ", second argument: ", ..2, ", last argument: ", ..3)
}

w(1, 2, 3)
```

```
[1] "First argument: 1, second argument: 2, last argument: 3"
```

If you want to learn more about `...`, type `?dots` in an R console.

Because functions are nothing special, you can also write functions that return functions. As an illustration, we'll be writing a function that converts warnings to errors. This can be quite useful if you want your functions to fail early, which often makes debugging easier. For example, try running this:

```
sqrt(-5)
```

```
Warning in sqrt(-5): NaNs produced
```

```
[1] NaN
```

This only raises a warning and returns `NaN` (Not a Number). This can be quite dangerous, especially when working non-interactively, which is what we will be doing a lot later on. It is much better if a pipeline fails early due to an error, than dragging a `NaN` value. This also happens with `log()`:

```
sqrt(-10)
```

```
Warning in sqrt(-10): NaNs produced
```

```
[1] NaN
```

So it could be useful to redefine these functions to raise an error instead, for example like this:

```
strict_sqrt <- function(x){

  if(x <= 0) stop("x is negative")
```

```
  sqrt(x)

}
```

This function now throws an error for negative `x`:

```
strict_sqrt(-10)
```

```
Error in strict_sqrt(-10) : x is negative
```

However, it can be quite tedious to redefine every function that we need in our pipeline, and remember, we don't want to repeat ourselves. The other thing you need to remember is that functions are nothing special. Which means that we can define a function that takes a function as an argument, converts any warning thrown by that function into an error, and returns a new function. For example:

```
strictly <- function(f){
  function(...){
    tryCatch({
      f(...)
    },
    warning = function(warning)stop("Can't do that chief"))
  }
}
```

This function makes use of `tryCatch()` which catches warnings raised by an expression (in this example the expression is `f(...)`) and then raises an error instead with the `stop()` function. It is now possible to define new functions like this:

```
s_sqrt <- strictly(sqrt)
```

```
s_sqrt(-4)
```

```
Error in value[[3L]](cond) : Can't do that chief
```

```
s_log <- strictly(log)
```

```
s_log(-4)
```

```
Error in value[[3L]](cond) : Can't do that chief
```

Functions that return functions are called *function factories* and they're incredibly useful. I use this so much that I've written a package, available on CRAN, called `{chronicler}`, that does this:

```
s_sqrt <- chronicler::record(sqrt)
```

```
result <- s_sqrt(-4)

result
```

```
NOK! Value computed unsuccessfully:
---------------
Nothing

---------------
This is an object of type `chronicle`.
Retrieve the value of this object with pick(.c, "value").
To read the log of this object, call read_log(.c).
```

Because the expression above resulted in an error, `Nothing` is returned. `Nothing` is a special value defined in the `{maybe}` package (check it out, very interesting package!). We can then even read the log to see what went wrong:

```
chronicler::read_log(result)
```

```
[1] "Complete log:"
[2] "NOK! sqrt() ran unsuccessfully with following exception: NaNs produced at 2023-02-21 17
[3] "Total running time: 0.00133323669433594 secs"
```

The `{purrr}` package also comes with function factories that you might find useful (`{possibly}`, `{safely}` and `{quietly}`).

In part 2 we will also learn about assertive programming, another way of making our functions safer, as an alternative to using function factories.

5.2.2 Optional arguments

It is possible to make functions' arguments optional, by using `NULL`. For example:

```
g <- function(x, y = NULL){
  if(is.null(y)){
    print("optional argument y is NULL")
    x
  } else {
    if(y == 5) print("y is present"); x+y
  }
}
```

Calling `g(10)` prints the message "Optional argument y is NULL", and returns 10. Calling `g(10, 5)` however, prints "y is present" and returns 15. It is also possible to use `missing()`:

```
g <- function(x, y){
  if(missing(y)){
    print("optional argument y is missing")
    x
  } else {
    if(y == 5) print("y is present"); x+y
  }
}
```

I however prefer the first approach, because it is clearer which arguments are optional, which is not the case with the second approach, where you need to read the body of the function.

5.2.3 Safe functions

It is important that your functions are safe and predictable. You should avoid writing functions that behave like `nchar()`, a base R function. Let's see why this function is not safe:

```
nchar("10000000")
```

```
[1] 8
```

It returns the expected result of 8. But what if I remove the quotes?

```
nchar(10000000)
```

```
[1] 5
```

What is going on here? I'll give you a hint: simply type `10000000` in the console:

```
10000000
```

```
[1] 1e+07
```

`10000000` gets represented as `1e+07` by R. This number in scientific notation gets then converted into the character "1e+07" by `nchar()`, and this conversion happens silently. `nchar()` then counts the number of characters, and *correctly* returns 5. The problem is that it doesn't make sense to provide a number to a function that expects a character. This function should have returned an error message, or at the very least raised a warning that the number got converted into a character. Here is how you could rewrite `nchar()` to make it safer:

```
nchar2 <- function(x, result = 0){

  if(!isTRUE(is.character(x))){
    stop(paste0("x should be of type 'character', but is of type '",
                typeof(x), "' instead."))
  } else if(x == ""){
    result
  } else {
    result <- result + 1
    split_x <- strsplit(x, split = "")[[1]]
    nchar2(paste0(split_x[-1],
                  collapse = ""), result)
  }
}
```

This function now returns an error message if the input is not a character:

```
nchar2(10000000)
```

```
Error in nchar2(10000000) : x should be of type 'character', but is of type 'integer' instead
```

This section is in a sense an introduction to assertive programming. As mentioned in the section on function factories, we will be learning about assertive programming in greater detail in part 2 of the book.

5.2.4 Recursive functions

You may have noticed in the last lines of `nchar2()` defined above, that `nchar2()` calls itself. A function that calls itself in its own body is called a recursive function. It is sometimes

easier to define a function in its recursive form than in an iterative form. The most common example is the factorial function. However, there is an issue with recursive functions (in the R programming language, other programming languages may not have the same problem, like Haskell): while it is sometimes easier to write a function using a recursive algorithm than an iterative algorithm, like for the factorial function, recursive functions in R are quite slow. Let's take a look at two definitions of the factorial function, one recursive, the other iterative:

```
fact_iter <- function(n){
  result = 1
  for(i in 1:n){
    result = result * i
    i = i + 1
  }
  result
}

fact_recur <- function(n){
  if(n == 0 || n == 1){
  result = 1
  } else {
    n * fact_recur(n-1)
  }
}
```

Using the `{microbenchmark}` package we can benchmark the code:

```
microbenchmark::microbenchmark(
  fact_recur(50),
  fact_iter(50)
)
```

```
Unit: microseconds
           expr    min     lq     mean median      uq    max neval
 fact_recur(50) 21.501 21.701 23.82701 21.901 22.0515 68.902   100
  fact_iter(50)  2.000  2.101  2.74599  2.201  2.3510 21.000   100
```

We see that the recursive factorial function is 10 times slower than the iterative version. In this particular example it doesn't make much of a difference, because the functions only take microseconds to run. But if you're working with more complex functions, this is a problem. If you want to keep using the recursive function and not switch to an iterative algorithm, there are ways to make them faster. The first is called *trampolining*. I won't go into details, but

if you're interested, there is an R package that allows you to use trampolining with R, aptly called {trampoline}. Another solution is using the {memoise} package.

5.2.5 Anonymous functions

It is possible to define a function and not give it a name. For example:

```
function(x)(x+1)(10)
```

Since R version 4.1, there is even a shorthand notation for anonymous functions:

```
(\(x)(x+1))(10)
```

Because we don't name them, we cannot reuse them. So why is this useful? Anonymous functions are useful when you need to apply a function somewhere inside a pipe once, and don't want to define a function just for this. This will become clearer once we learn about lists, but before that, let's philosophize a bit.

5.2.6 The Unix philosophy applied to R

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Doug McIlroy, in A Quarter Century of Unix[1]

We can take inspiration from the Unix philosophy and rewrite it like this for our purposes:

Write functions that do one thing and do it well. Write functions that work together. Write functions that handle lists, because that is a universal interface.

Strive for writing simple functions that only perform one task. Don't hesitate to split a big function into smaller ones. Small functions that only perform one task are easier to maintain, test, document and debug. These smaller functions can then be chained using the `|>` operator. In other words, it is preferable to have something like:

```
a |> f() |> g() |> h()
```

where `a` is for example a path to a data set, and where `f()`, `g()` and `h()` successively read, clean, and plot the data, than having something like:

[1] https://stackoverflow.com/a/68690065/1298051

```
big_function(a)
```

that does all the steps above in one go.

This idea of splitting the problem into smaller chunks, each chunk in turn split into even smaller units that can be handled by functions and then the results of these function combined into a final output is called composition.

The advantage of splitting `big_function()` into `f()`, `g()` and `h()` is that you can *eat the elephant one bite at a time*, and also reuse these smaller functions in other projects more easily. So what's important is that you can make small functions work together by sharing a common interface. The list is usually a good candidate for this.

5.3 Lists: a powerful data-structure

Lists are the second important ingredient of functional programming. In the R philosophy inspired by the UNIX philosophy, I stated that *lists are a universal interface* in R, so our functions should handle lists. This of course depends on what it is you're doing. If you need functions to handle numbers, then there's little value in placing these numbers inside lists. But in practice, you will very likely manipulate objects that are more complex than numbers, and this is where lists come into play.

5.3.1 Lists all the way down

Lists are extremely flexible, and most of the very complex objects classes that you manipulate are actually lists, but just fancier. For example, a data frame is a list:

```
data(mtcars)

typeof(mtcars)
```

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

A fitted model is a list:

```
my_model <- lm(hp ~ mpg, data = mtcars)

typeof(my_model)
```

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

A ggplot is a list:

```
library(ggplot2)

my_plot <- ggplot(data = mtcars) +
  geom_line(aes(y = hp, x = mpg))

typeof(my_plot)
```

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

It's lists all the way down, and it's not a coincidence. It's because, as stated, lists are very powerful. So it's important to know what you can do with lists.

5.3.2 Lists can hold many things

If you write a function that needs to return many objects, the only solution is to place them inside a list. For example, consider this function:

```
sqrt_newton <- function(a, init = 1, eps = 0.01, steps = 1){
    stopifnot(a >= 0)
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        steps <- steps + 1
    }
    list(
      "result" = init,
      "steps" = steps
    )
}
```

This function returns the square root of a number using Newton's algorithm, as well as the number of steps, or iterations, it took to reach the solution:

```
result_list <- sqrt_newton(1600)

result_list
```

```
$result
[1] 40
```

```
$steps
[1] 10
```

It is quite common to instead print the number of steps to the console instead of returning them. But the issue with a function that prints something to the console instead of returning it, is that such a function is not pure, as it changes something outside of its scope. And if you need the information that got printed (for example, if you want to count all the steps it took to run the script from start to finish), it is lost. It gets printed, and that's it. It is preferable to instead make the function pure by returning everything inside a neat list. It is then possible to separately save these objects if needed:

```
result <- result_list$result

result_steps <- result_list$steps
```

Or you could define functions that know how to deal with the list:

```
f <- function(result_list){
  list(
    "result" = result_list$result * 10,
    "steps" = result_list$steps + 1
    )
}

f(result_list)
```

```
$result
[1] 400

$steps
[1] 11
```

It all depends on what you want to do. But it is usually better to keep everything neatly inside a list.

Lists can also hold objects of different types:

```
list(
  "a" = head(mtcars),
  "b" = ~lm(y ~ x)
  )
```

```
$a
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

$b
~lm(y ~ x)
```

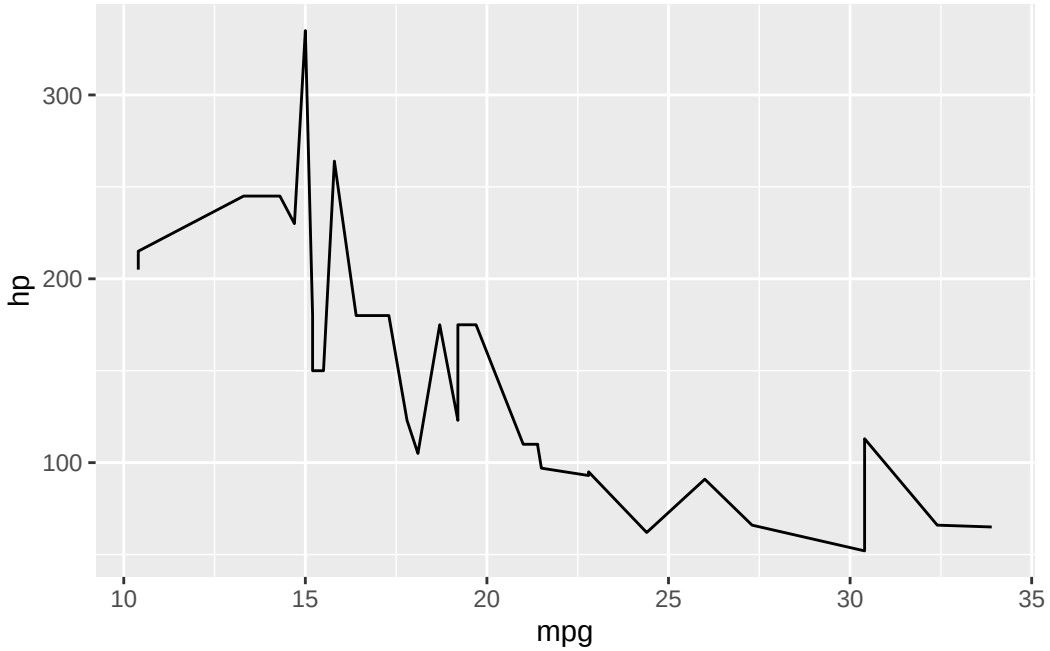
The list above has two elements, the first is the head of the `mtcars` data frame, the second is a formula object. Lists can even hold other lists:

```
list(
  "a" = head(mtcars),
  "b" = list(
    "c" = sqrt,
    "d" = my_plot # Remember this ggplot object from before?
    )
  )
```

```
$a
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

$b
$b$c
function (x)  .Primitive("sqrt")

$b$d
```



Use this to your advantage.

5.3.3 Lists as the cure to loops

Loops are incredibly useful, and you are likely familiar with them. The problem with loops is that they are a concept from iterative programming, not functional programming, and this is a problem because loops rely on changing the state of your program to run. For example, let's suppose that you wish to use a for-loop to compute the sum of the first 100 integers:

```
result <- 0
for (i in 1:100){
  result <- result + i
}

print(result)
```

```
[1] 5050
```

If you run `ls()` now, you should see that there's a variable `i` in your global environment. This could cause issues further down in your pipeline if you need to re-use `i`. Also, writing loops is, in my opinion, quite error prone. But how can we avoid using loops? For looping in a

functional programming language, we need to use higher-order functions and lists. A reminder: a higher-order function is a function that takes another function as an argument. Looping is a task like any other, so we can write a function that does the looping for us. We will write a function, and call it `looping()`, which will take a function as an argument, as well as a list. The list will serve as the container to hold our numbers:

```
looping <- function(a_list, a_func, init = NULL, ...){

  # If the user does not provide an `init` value,
  # set the head of the list as the initial value
  if(is.null(init)){
    init <- a_list[[1]]
    a_list <- tail(a_list, -1)
  }

  # Separate the head from the tail of the list
  # and apply the function to the initial value and the head of the list
  head_list = a_list[[1]]
  tail_list = tail(a_list, -1)
  init = a_func(init, head_list, ...)

  # Check if we're done: if there is still some tail,
  # rerun the whole thing until there's no tail left
  if(length(tail_list) != 0){
    looping(tail_list, a_func, init, ...)
  }
  else {
    init
  }
}
```

Now, this might seem much more complicated than a for loop. However, now that we have abstracted the loop away inside a function, we can keep reusing this function:

```
looping(as.list(seq(1:100)), `+`)
```

```
[1] 5050
```

Of course, because this is so useful, `looping()` actually ships with R, and is called `Reduce()`:

```
Reduce(`+`, seq(1:100)) # the order of the arguments is `function` then `list` for `Reduce
```

```
[1] 5050
```

But this is not the only way that we can loop. We can also write a loop that applies a function to each element of a list, instead of operating on the whole list:

```
result <- as.list(seq(1:5))
for (i in seq_along(result)){
  result[[i]] <- sqrt(result[[i]])
}

print(result)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068
```

Here again, we have to pollute the global environment by first creating a vessel for our results, and then apply the function at each index. We can abstract this process away in a function:

```
applying <- function(a_list, a_func, ...){

  head_list = a_list[[1]]
  tail_list = tail(a_list, -1)
  result = a_func(head_list, ...)

  # Check if we're done: if there is still some tail, rerun the whole thing until there's
  if(length(tail_list) != 0){
    append(result, applying(tail_list, a_func, ...))
  }
  else {
```

```
    result
  }
}
```

Once again this might seem complicated, and I would agree. Abstraction is complex. But once we have it, we can focus on the task at hand, instead of having to always tell the computer what we want:

```
applying(as.list(seq(1:5)), sqrt)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Of course, R ships with its own, much more efficient, implementation of this function:

```
lapply(list(seq(1:5)), sqrt)
```

```
[[1]]
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

In other programming languages, `lapply()` is often called `map()`. The `{purrr}` package ships with other such useful higher-order functions that abstract loops away. For example, there's the function called `map2()`, that maps a function of two arguments to each element of two atomic vectors or lists, two at a time:

```
library(purrr)

map2(
  .x = seq(1:5),
  .y = seq(1:5),
  .f = `+`
  )
```

```
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
```

```
[1] 6

[[4]]
[1] 8

[[5]]
[1] 10
```

If you have more than two lists, you can use `pmap()` instead.

5.3.4 Data frames

As mentioned in the introduction of this section, data frames are a special type of list of atomic vectors. This means that just as I can use `lapply()` to compute the square root of the elements of an atomic vector, as in the previous example, I can also operate on all the columns of a data frame. For example, it is possible to determine the class of every column of a data frame like this:

```
lapply(iris, class)
```

```
$Sepal.Length
[1] "numeric"

$Sepal.Width
[1] "numeric"

$Petal.Length
[1] "numeric"

$Petal.Width
[1] "numeric"

$Species
[1] "factor"
```

Unlike a list however, the elements of a data frame must be of the same length. Data frames remain very flexible though, and using what we have learned until now it is possible to use the data frame as a structure for all our computations. For example, suppose that we have a data frame that contains data on unemployment for the different subnational divisions of the Grand-Duchy of Luxembourg, the country the author of this book hails from. Let's suppose that I want to generate several plots, per subnational division and per year. Typically, we

would use a loop for this, but we can use what we've learned here, as well as some functions from the `{dplyr}`, `{purrr}`, `{ggplot2}` and `{tidyr}` packages. I will be downloading data that I made available inside a package, but instead of installing the package, we will download the `.rda` file directly (which is the file format of packaged data) and then load that data into our R session:

```
# Create a temporary file
unemp_path <- tempfile(fileext = ".rda")

# Download the data and save it to the path of the temporary file
download.file("https://github.com/b-rodrigues/myPackage/raw/main/data/unemp.rda",
              destfile = unemp_path)

# Load the data. The data is now available as 'unemp'
load(unemp_path)
```

Let's load the required packages and take a look at the data:

```
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
```

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

glimpse(unemp)
```

```
Rows: 472
Columns: 9
$ year                          <dbl> 2013, 2013, 2013, 2013, 2013, 2013, 2013,~
```

```
$ place_name                   <chr> "Luxembourg", "Capellen", "Dippach", "Gar~
$ level                        <chr> "Country", "Canton", "Commune", "Commune"~
$ total_employed_population    <dbl> 223407, 17802, 1703, 844, 1431, 4094, 214~
$ of_which_wage_earners        <dbl> 203535, 15993, 1535, 750, 1315, 3800, 187~
$ of_which_non_wage_earners    <dbl> 19872, 1809, 168, 94, 116, 294, 272, 113,~
$ unemployed                   <dbl> 19287, 1071, 114, 25, 74, 261, 98, 45, 66~
$ active_population            <dbl> 242694, 18873, 1817, 869, 1505, 4355, 224~
$ unemployment_rate_in_percent <dbl> 7.947044, 5.674773, 6.274078, 2.876870, 4~
```

Column names are self-descriptive, but the `level` column needs some explanations. `level` contains the adiministrative divisions of the country, so the country of Luxembourg, then the Cantons and then the Communes.

Remember that Luxembourg can refer to the country, the canton or the commune of Luxembourg. Now let's suppose that I want a separate plot for the three communes of Luxembourg, Esch-sur-Alzette and Wiltz. Instead of creating three separate data frames and feeding them to the same ggplot code, I can instead take advantage of the fact that data frames are lists, and are thus quite flexible. Let's start with filtering:

```
filtered_unemp <- unemp %>%
  filter(
    level == "Commune",
    place_name %in% c("Luxembourg", "Esch-sur-Alzette", "Wiltz")
   )

glimpse(filtered_unemp)
```

```
Rows: 12
Columns: 9
$ year                         <dbl> 2013, 2013, 2013, 2014, 2014, 2014, 2015,~
$ place_name                   <chr> "Esch-sur-Alzette", "Luxembourg", "Wiltz"~
$ level                        <chr> "Commune", "Commune", "Commune", "Commune~
$ total_employed_population    <dbl> 12725, 39513, 2344, 13155, 40768, 2377, 1~
$ of_which_wage_earners        <dbl> 12031, 35531, 2149, 12452, 36661, 2192, 1~
$ of_which_non_wage_earners    <dbl> 694, 3982, 195, 703, 4107, 185, 710, 4140~
$ unemployed                   <dbl> 2054, 3855, 318, 1997, 3836, 315, 2031, 3~
$ active_population            <dbl> 14779, 43368, 2662, 15152, 44604, 2692, 1~
$ unemployment_rate_in_percent <dbl> 13.898099, 8.889043, 11.945905, 13.179778~
```

We are now going to use the fact that data frames are lists, and that lists can hold any type of object. For example, remember this list from before where one of the elements is a data frame, and the second one a formula:

```
  list(
    "a" = head(mtcars),
    "b" = ~lm(y ~ x)
    )
```

```
$a
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

$b
~lm(y ~ x)
```

`{dplyr}` comes with a function called `group_nest()` which groups the data frame by a variable (such that the next computations will be performed group-wise) and then nests the other columns into a smaller data frame. Let's try it and see what happens:

```
  nested_unemp <- filtered_unemp %>%
    group_nest(place_name)
```

Let's see what this looks like:

```
  nested_unemp
```

```
# A tibble: 3 x 2
  place_name                    data
  <chr>            <list<tibble[,8]>>
1 Esch-sur-Alzette            [4 x 8]
2 Luxembourg                  [4 x 8]
3 Wiltz                       [4 x 8]
```

`nested_unemp` is a new data frame of 3 rows, one per commune ("Esch-sur-Alzette", "Luxembourg", "Wiltz"), and of two columns, one for the names of the communes, and the other contains every other variable inside a smaller data frame. So this is a data frame that has one column where each element of that column is itself a data frame. Such a column is called a list-column. This is essentially a list of lists.

Let's now think about this for a moment. If the column titled `data` is a list of data frames, it should be possible to use a function like `map()` or `lapply()` to apply a function on each of these data frames. Remember that `map()` or `lapply()` require a list of elements of whatever type and a function that accepts objects of this type as input. So this means that we could apply a function that plots the data to each element of the column titled `data`. Since each element of this column is a data frame, this functions needs a data frame as an input. As a first, simple, example to illustrate this, let's suppose that we want to determine the number of rows of each data frame. This is how we would do it:

```
nested_unemp %>%
  mutate(nrows = map(data, nrow))
```

```
# A tibble: 3 x 3
  place_name                    data nrows
  <chr>            <list<tibble[,8]>> <list>
1 Esch-sur-Alzette           [4 x 8] <int [1]>
2 Luxembourg                 [4 x 8] <int [1]>
3 Wiltz                      [4 x 8] <int [1]>
```

The new column, titled `nrows` is a list of integers. We can simplify it by converting it directly to an atomic vector of integers by using `map_int()` instead of `map()`:

```
nested_unemp %>%
  mutate(nrows = map_int(data, nrow))
```

```
# A tibble: 3 x 3
  place_name                    data nrows
  <chr>            <list<tibble[,8]>> <int>
1 Esch-sur-Alzette           [4 x 8]     4
2 Luxembourg                 [4 x 8]     4
3 Wiltz                      [4 x 8]     4
```

Let's try for a more complex example now. What if we want to filter rows? (The simplest way would of course to filter the rows we need before nesting the data frame). We need to apply the function `filter()` where its first argument is a data frame and the second argument is a predicate:

```
nested_unemp %>%
  mutate(nrows = map(data, \(x)filter(x, year == 2015)))
```

```
# A tibble: 3 x 3
  place_name                    data nrows
  <chr>            <list<tibble[,8]>> <list>
1 Esch-sur-Alzette           [4 x 8] <tibble [1 x 8]>
2 Luxembourg                 [4 x 8] <tibble [1 x 8]>
3 Wiltz                      [4 x 8] <tibble [1 x 8]>
```

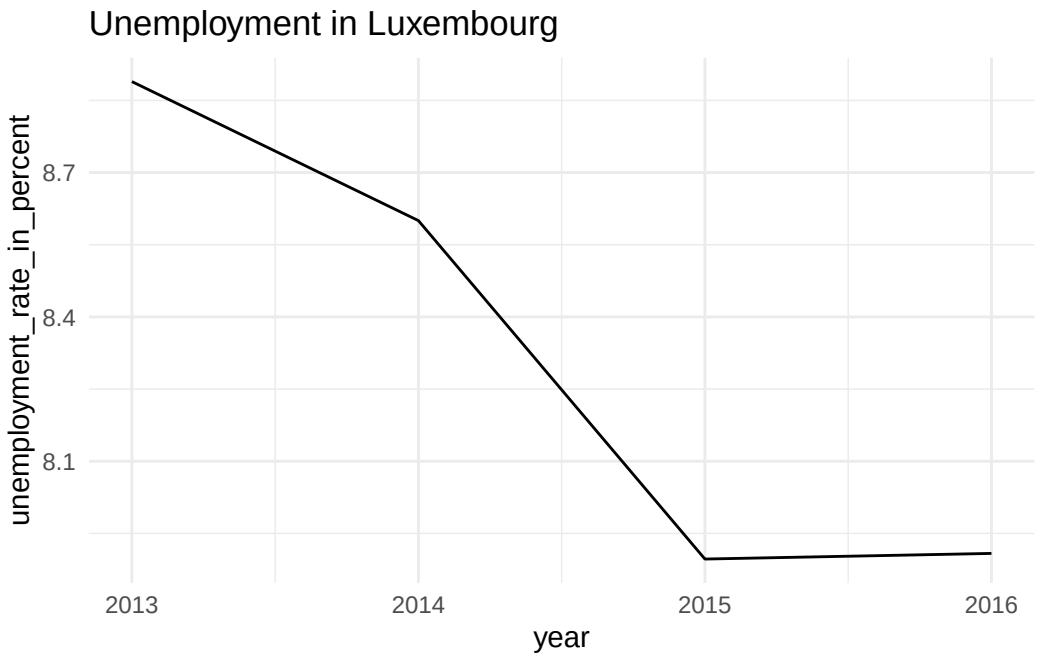
In this case, we need to use an anonymous function. This is because `filter()` has two arguments and we need to make clear what it is we are mapping over and what argument stays fixed; we are mapping over, or iterating if you will, data frames, but the predicate `year == 2015` stays fixed.

We are now ready to plot our data. The best way to continue is to first get the function right by creating one plot for one single commune. Let's select the dataset for the commune of `Luxembourg`:

```
lux_data <- nested_unemp %>%
  filter(place_name == "Luxembourg") %>%
  unnest(data)
```

To plot this data, we can now write the required `ggplot2()` code:

```
ggplot(data = lux_data) +
  theme_minimal() +
  geom_line(
    aes(year, unemployment_rate_in_percent, group = 1)
   ) +
  labs(title = "Unemployment in Luxembourg")
```



To turn the lines of code above into a function, you need to think about how many arguments that function would have. There is an obvious one, the data itself (in the snippet above, the data is the `lux_data` object). Another one that is less obvious is in the title:

```
    labs(title = "Unemployment in Luxembourg")
```

```
$title
[1] "Unemployment in Luxembourg"

attr(,"class")
[1] "labels"
```

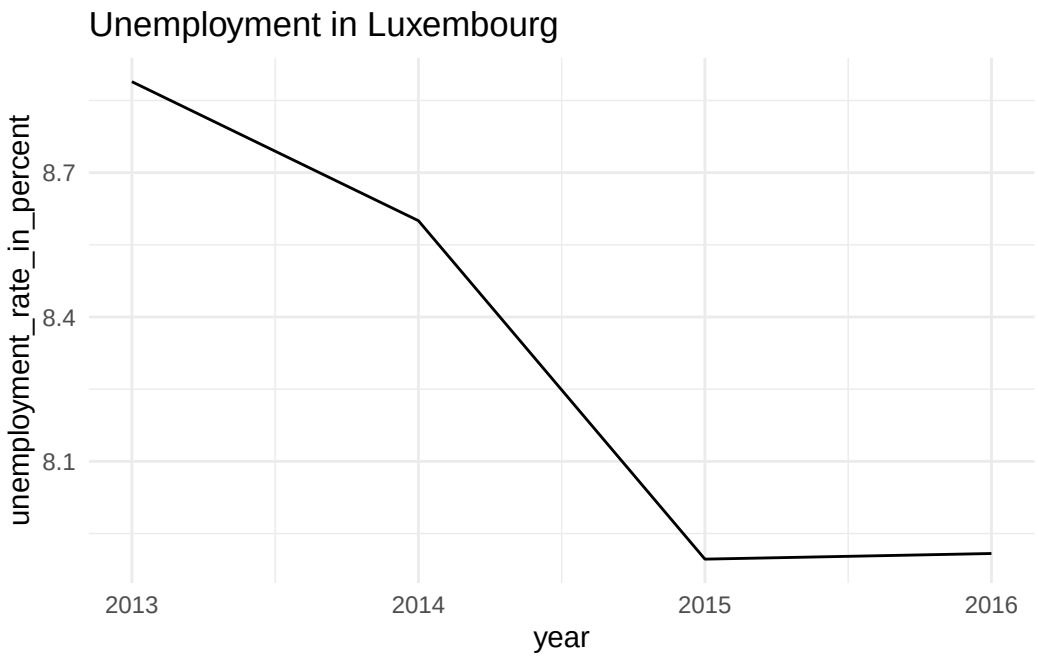
Ideally, we would want that title to change depending on the data set. So we could write the function like so:

```
make_plot <- function(x, y){
  ggplot(data = x) +
    theme_minimal() +
    geom_line(
      aes(year, unemployment_rate_in_percent, group = 1)
      ) +
```

```
    labs(title = paste("Unemployment in", y))
}
```

Let's try it on our data:

```
make_plot(lux_data, "Luxembourg")
```



Ok, so now, we simply need to apply this function to our nested data frame:

```
nested_unemp <- nested_unemp %>%
  mutate(plots = map2(
    .x = data,
    .y = place_name,
    .f = make_plot
  ))

nested_unemp
```

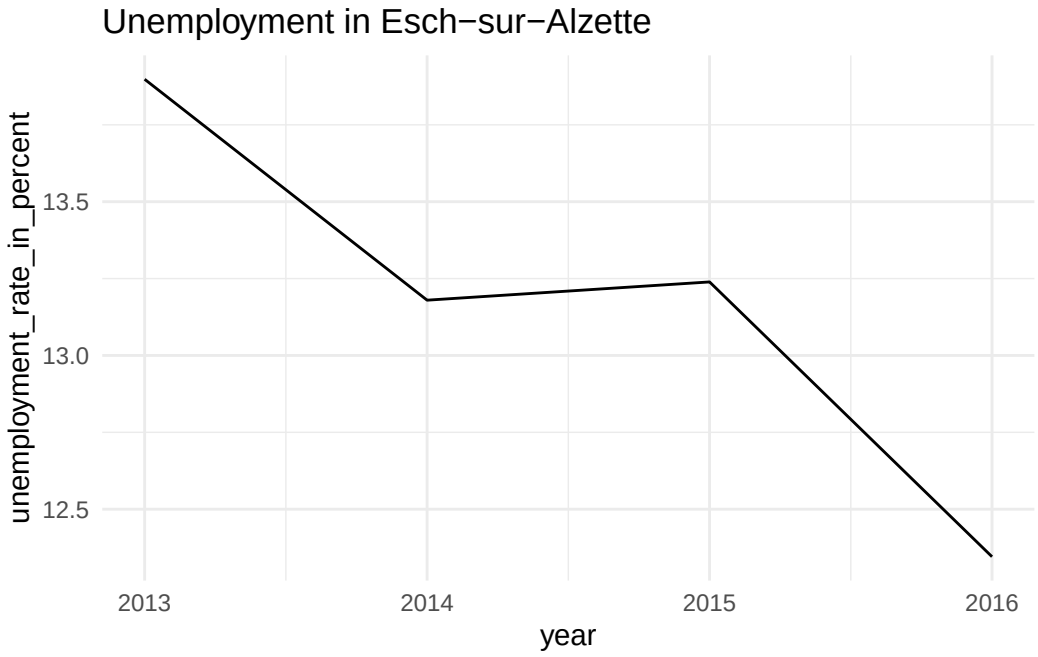
```
# A tibble: 3 x 3
  place_name                    data plots
  <chr>            <list<tibble[,8]>> <list>
```

```
1 Esch-sur-Alzette            [4 x 8] <gg>
2 Luxembourg                  [4 x 8] <gg>
3 Wiltz                       [4 x 8] <gg>
```

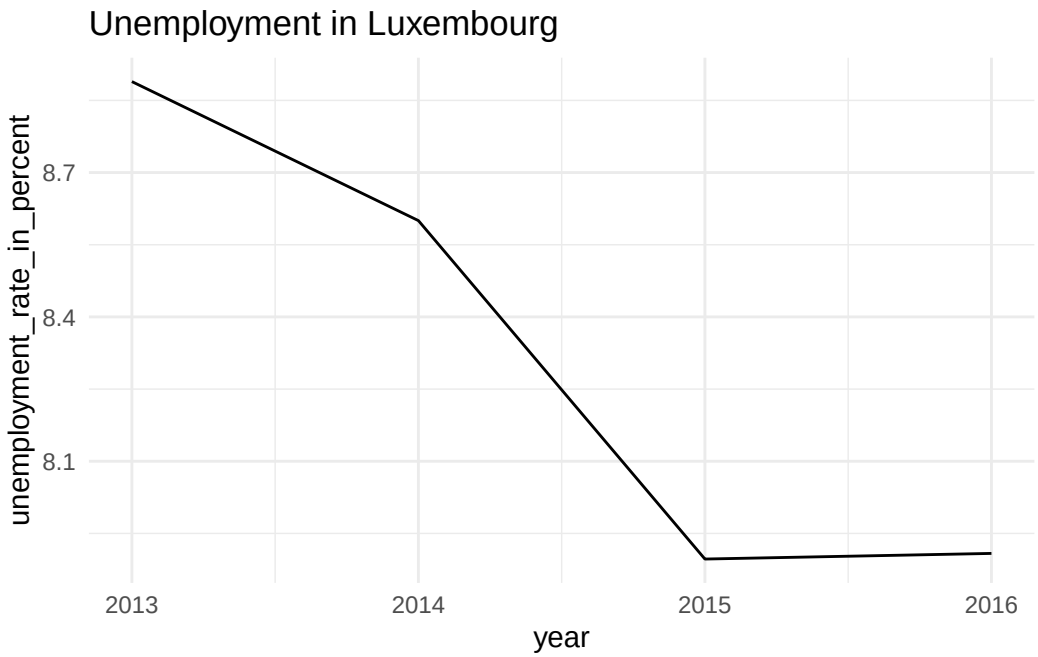
If you look at the `plots` column, you see that it is a list of `gg` objects: these are our plots. Let's take a look at them:

```
nested_unemp$plots
```

```
[[1]]
```




```
[[2]]
```


Unemployment in Luxembourg
unemployment_rate_in_percent
8.7
8.4
8.1
2013
2014
2015
2016
year

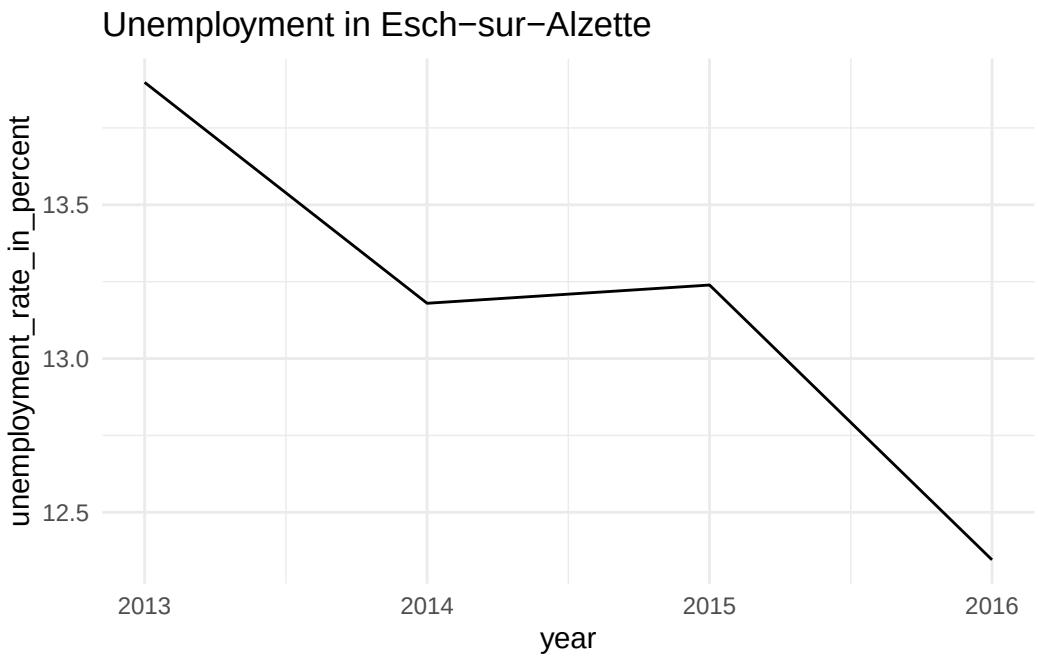
```
[[3]]
```


Unemployment in Wiltz
unemployment_rate_in_percent
12.1
12.0
11.9
11.8
11.7
11.6
2013
2014
2015
2016
year

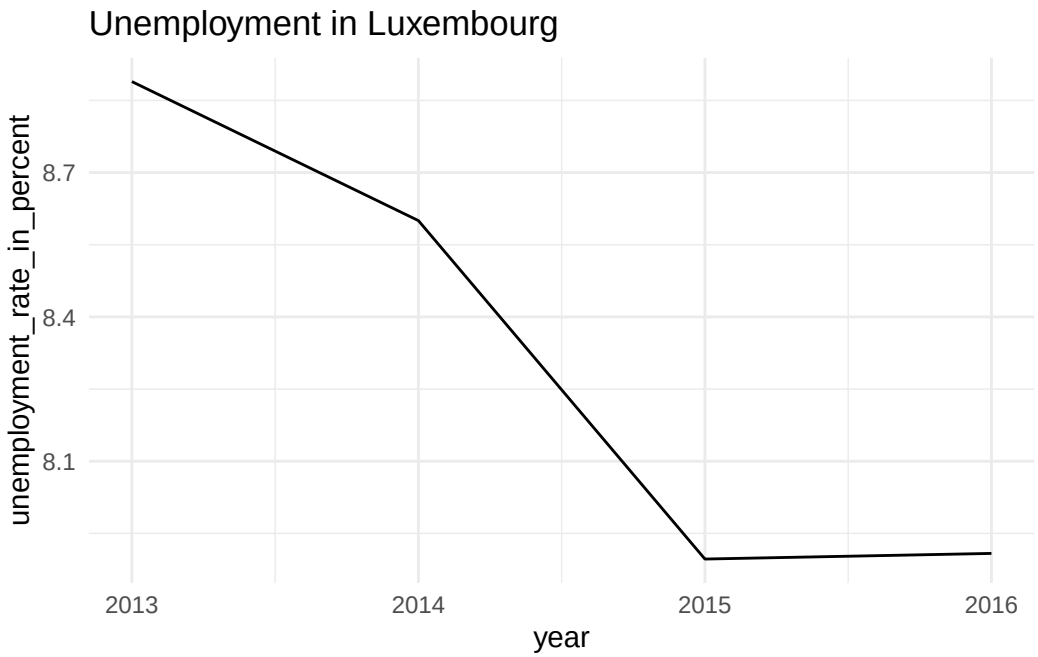
We could also have used an anonymous function:

```
nested_unemp %>%
  mutate(plots2 = map2(
    .x = data,
    .y = place_name,
    .f = \(.x,.y)(
                ggplot(data = .x) +
                  theme_minimal() +
                  geom_line(
                    aes(year, unemployment_rate_in_percent, group = 1)
                   ) +
                  labs(title = paste("Unemployment in", .y))
                  )
           )
         ) %>%
  pull(plots2)
```

```
[[1]]
```




```
[[2]]
```


Unemployment in Luxembourg
unemployment_rate_in_percent
8.7
8.4
8.1
2013
2014
2015
2016
year

```
[[3]]
```


Unemployment in Wiltz
unemployment_rate_in_percent
12.1
12.0
11.9
11.8
11.7
11.6
2013
2014
2015
2016
year

This list-column based workflow is extremely powerful and I highly advise you to take the required time to master it. Remember, we never want to have to repeat ourselves. This might seem more complicated that repeating yourself, but imagine that you need to do this for various countries, various variables, etc… What are you going to do, copy and paste code everywhere? This gets very tedious and more importantly, very error-prone, because you will forget to update the code in some places. You could of course use a loop instead of this list-column based workflow. But as mentioned, the issue with loops is that you have to interact with the global environment, which can lead to other issues. But whatever you end up using, you need to avoid copy and pasting at all costs.

5.4 Functional programming in R

Up until now I focused on general concepts than on specifics of the R programming language when it comes to functional programming. In the section, we will be focusing entirely on R-specific capabilities and packages for functional programming.

5.4.1 Base capabilities

R is a functional programming language, (but not only), and as such it comes with many functions out of the box to write functional code. We have already discussed `lapply()` and `Reduce()`. You should know that depending on what you want to achieve, there are other functions that are similar to `lapply()`: `apply()`, `sapply()`, `vapply()`, `mapply()` and `tapply()`. There's also `Map()` which is a wrapper around `mapply()`. Each function performs the same basic task of applying a function over all the elements of a list or list-like structure, but it can be hard to keep them apart and when you should use one over another. This is why `{purrr}`, which we will discuss in the next section, is quite an interesting alternative to base R's offering.

Another one of the quintessential functional programming functions (alongside `Reduce()` and `Map()`) that ships with R is `Filter()`. If you know `dplyr::filter()` you should be familiar with the concept of filtering rows of a data frame where the elements of one particular column satisfy a predicate. `Filter()` works the same way, but focusing on lists instead of data frame:

```
Filter(is.character,
       list(
         seq(1:5),
         "Hey")
       )
```

```
[[1]]
[1] "Hey"
```

The call above only returns the elements where `is.character()` evaluates to `TRUE`.

Another useful function is `Negate()` which is a function factory that takes a boolean function as an input and returns the opposite boolean function. As an illustration, suppose that in the example above we wanted to get everything *but* the character:

```
Filter(Negate(is.character),
       list(
         seq(1:5),
         "Hey")
       )
```

```
[[1]]
[1] 1 2 3 4 5
```

There are some other functions like this that you might want to check out: type `?Negate` in console to read more about them.

Before continuing with R packages that extend R's functional programming capabilities it's also important to stress that just as R is a functional programming language, it is also an object oriented language. In fact, R is what John Chambers called a *functional OOP* language (Chambers (2014)). We won't delve too much into what this means (read Wickham (2019) for this), but as a short discussion, consider the `print()` function. Depending on what type of object the user gives it, it seems as if somehow `print()` knows what to do with it:

```
print(5)
```

```
[1] 5
```

```
print(head(mtcars))
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
  print(str(mtcars))
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
NULL
```

This works by essentially mixing both functional and object-oriented programming, hence functional OOP. Let's take a closer look at the source code of `print()` by simply typing `print` without brackets, into a console:

```
  print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x5613f0205958>
<environment: namespace:base>
```

Quite unexpectedly, the source code of `print()` is one line long and is just `UseMethod("print")`. So all `print()` does is use a generic method called "print". If your text editor has autocompletion enabled, you might see that there are actually quite a lot of `print()` functions. For example, type `print.data.frame` into a console:

```
  print.data.frame
```

```
function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
    row.names = TRUE, max = NULL)
{
    n <- length(row.names(x))
    if (length(x) == 0L) {
```

```
        cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
            "data frame with 0 columns and %d rows"), n), "\n",
            sep = "")
    }
    else if (n == 0L) {
        print.default(names(x), quote = FALSE)
        cat(gettext("<0 rows> (or 0-length row.names)\n"))
    }
    else {
        if (is.null(max))
            max <- getOption("max.print", 99999L)
        if (!is.finite(max))
            stop("invalid 'max' / getOption(\"max.print\"): ",
                max)
        omit <- (n0 <- max%/%length(x)) < n
        m <- as.matrix(format.data.frame(if (omit)
            x[seq_len(n0), , drop = FALSE]
        else x, digits = digits, na.encode = FALSE))
        if (!isTRUE(row.names))
            dimnames(m)[[1L]] <- if (isFALSE(row.names))
                rep.int("", if (omit)
                  n0
                else n)
            else row.names
        print(m, ..., quote = quote, right = right, max = max)
        if (omit)
            cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
                n - n0, "rows ]\n")
    }
    invisible(x)
}
<bytecode: 0x5613f131f810>
<environment: namespace:base>
```

This is the `print` function for `data.frame` objects. So what `print()` does is look at the class of its argument `x`, and then look for the right `print` function. In more traditional OOP languages, users would type something like:

```
mtcars.print()
```

In these languages, objects encapsulate methods (the equivalent of our functions), so if `mtcars` is a data frame, it encapsulates a `print()` method that then does the printing. R is different,

because classes and methods are kept separate. If a package developer creates a new object class, then the developer also must implement the required methods. For example in the `{chronicler}` package, the `chronicler` class is defined alongside the `print.chronicler()` function to print these objects.

All of this to say that if you want to extend R by writing packages, learning some OOP essentials is also important. But for data analysis, functional programming does the job perfectly. To learn more about R's different OOP systems (yes, R can do OOP in different ways and the one I sketched here is the simplest, but probably the most used as well), take a look at Wickham (2019).

5.4.2 purrr

The `{purrr}` package, developed by Posit (formerly RStudio), contains many functions to make functional programming with R go more smoothly. In the previous section, we discussed the `apply()` family of function; they all do a very similar thing, which is looping over a list and applying a function to the elements of the list, but it is not quite easy to remember which one does what. Also, for some of these functions like `apply()`, the list comes first, and then the function, but in the case of `mapply()`, the function comes first. This type of inconsistencies can be frustrating. Another issue with these functions is that it is not always easy to know what type the output is going to be. List? Atomic vector? Something else?

`{purrr}` solves this issue by offering the `map()` family of functions, which behave in a very consistent way. The basic function is called `map()` and we've already used it:

```
map(seq(1:5), sqrt)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068
```

But there are many interesting variants:

```
map_dbl(seq(1:5), sqrt)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

`map_dbl()` coerces the output to an atomic vector of doubles instead of a list of doubles. Then there's:

```
map_chr(letters, toupper)
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

for when the output needs to be an atomic vector of characters.

There are many others, so take a look at the document with `?map`. There's also `walk()` which is used if you're only interested in the side-effect of the function (for example if the function takes paths as input and saves something to disk).

`{purrr}` also has functions to replace `Reduce()`, simply called `reduce()` and `accumulate()`, and there are many, many other useful functions. Read through the documentation of the package and take the time to learn about all it has to offer.

5.4.3 withr

`{withr}` is a powerful package that makes it easy to "purify" functions that behave in a way that can cause problems. Remember the function from the introduction that randomly gave out a recipe Bruno liked? Here it is again:

```
h <- function(name, food_list = list()){

  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

  food_list <- append(food_list, food)

  print(paste0(name, " likes ", food))

  food_list
}
```

Because this function returns results that are not consistent for a fixed input, this function is not referentially transparent. So we improved the function by adding calls to `set.seed()` like this:

```
h2 <- function(name, food_list = list(), seed = 123){

  # We set the seed, making sure that we get the same selection of food for a given seed
  set.seed(seed)
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

  # We now need to unset the seed, because if we don't, guess what, the seed will stay set
  set.seed(NULL)

  food_list <- append(food_list, food)

  print(paste0(name, " likes ", food))

  food_list
}
```

The problem with this approach is that we need to modify our function. We can instead use `withr::with_seed()` to achieve the same effect:

```
withr::with_seed(seed = 123,
                 h("Bruno"))
```

```
[1] "Bruno likes feijoada"

[[1]]
[1] "feijoada"
```

It is also easier to create a wrapper if needed:

```
h3 <- function(..., seed){
  withr::with_seed(seed = seed,
                   h(...))
}
```

```
h3("Bruno", seed = 123)
```

```
[1] "Bruno likes feijoada"
```

```
[[1]]
[1] "feijoada"
```

In a previous example we downloaded a dataset and loaded it into memory; we did so by first created a temporary file, then downloading it and then loading it. Suppose that instead of loading this data into our session, we simply wanted to test whether the link was still working. We wouldn't want to keep the loaded data in our session, so to avoid having to delete it again manually, we could use `with_tempfile()`:

```
withr::with_tempfile("unemp", {
  download.file("https://github.com/b-rodrigues/myPackage/raw/main/data/unemp.rda",
                destfile = unemp)
  load(unemp)
  nrow(unemp)
  }
)
```

```
[1] 472
```

The data got downloaded, and then loaded, and then we computed the number of rows of the data, without touching the global environment, or state, of our current session.

Just like for `{purrr}`, `{withr}` has many useful functions which I encourage you to familiarize yourself with.

5.5 Conclusion

If there is only one thing that you should remember from this chapter, it would be pure functions. Writing pure functions is in my opinion not very difficult to do and comes with many benefits. But, avoiding loops and replacing them with higher-order functions (`lapply()`, `Reduce()`, `purrr::map()` – and its variants –) also pays off. While this chapter stresses the advantages of functional programming, you should not forget that R is not a pure, and solely, functional programming language and that other paradigms, like object-oriented programming, are also available to you. So if your goal is to master the language (instead of "just" using it to solve data analysis problems), then you also need to know about R's OOP capabilities.

6 Literate programming

We now know about version control, how to collaborate using Github.com and functional programming. By only learning about this, we have actually already made some massive steps towards making our projects reproducible. Especially by using Git and Github. Even if you're using private repos and work in the private sector, by using version control, you ensure that reusing this code for future projects is much easier. Auditing is greatly simplified as well.

But this book is still far from over. Let's think about our project up until now. We have downloaded some data, and wrote code to analyse it. Fair enough. But usually, we don't really stop there. We now need to write a report, or maybe a Powerpoint presentation. If you're a researcher, you still need to write a paper, just getting the results is not enough!

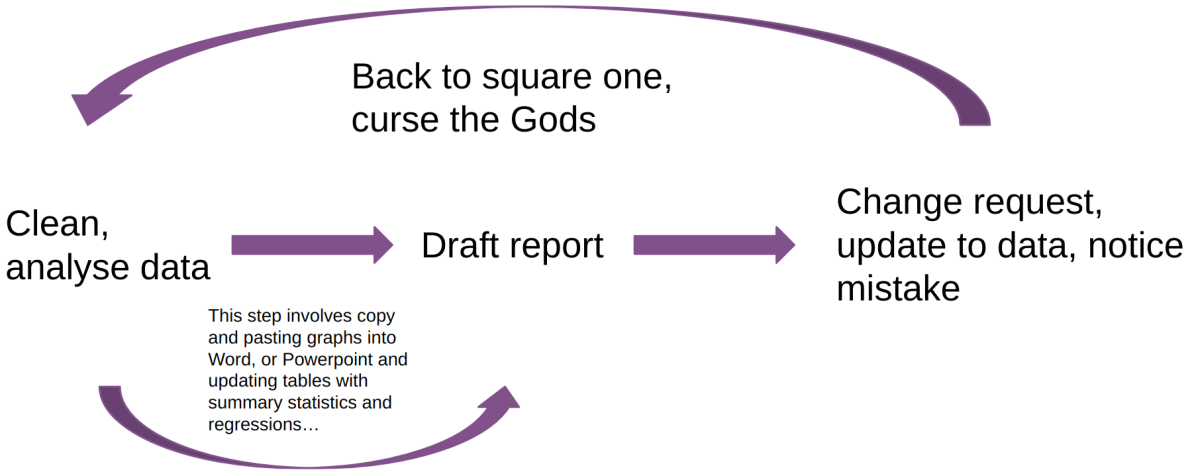


Figure 6.1: The cursed report drafting loop

The problem is that this workflow is very often quite horrible. The picture above illustrates this cursed report drafting loop. Get some results, copy and paste images into Word or Powerpoint, get a change request, or notice a mistake, and start from scratch again. If you're using LaTeX it'll be easier for pictures, but you'll still need to update tables by hand each time you need to touch your analysis code.

Worse, what if you start with a Word or LaTeX document, but then get asked to make a Powerpoint presentation as well? Then you need to copy and paste everything into Powerpoint

as well... and if you get a change request after you're done and need to start over, you might seriously consider raising goats instead of dealing with this again.

But if we can make the loop look like this instead:

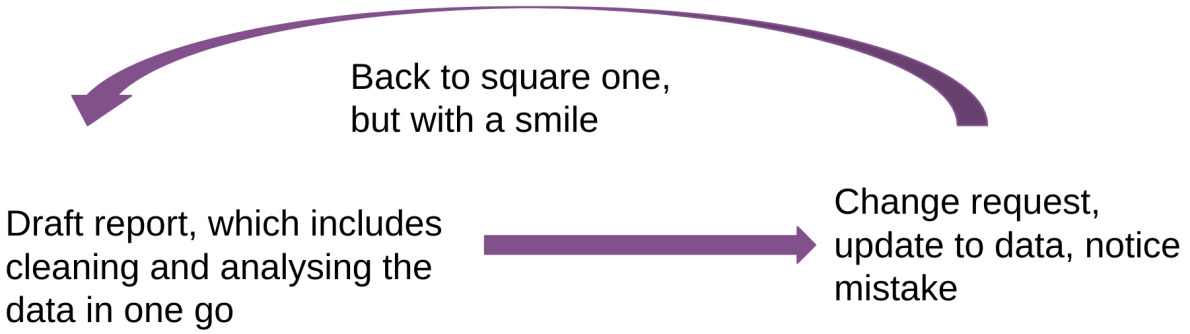


Figure 6.2: The holy report drafting loop

Basically, everything from cleaning, analysing and drafting is done in one single step? Well, this is what literate programming enables you to do. And even if you get asked to make a Powerpoint presentation, you can start from the same source code as the original report, and remove everything that you don't need and compile to a Powerpoint (or Beamer) presentation.

6.1 A quick history of literate programming

In literate programming, we mix code and prose together, which makes the output of our programs not just a series of tables, or graphs or predictions, but a complete report that contains the results of our analysis directly. Scripts written using literate programming are also very easy to compile, or render, into a variety of document formats like `html`, `docx`, `pdf` or even `pptx`. R supports several frameworks for literate programming: Sweave, knitr and Quarto.

Sweave was the first tool available to R (and S) users, and allowed the mixing of R and LaTeX code to create a document. Friedrich Leisch developed Sweave in 2002 and described it in his 2002 paper (Leisch (2002)). As Leisch argues, *the traditional way of writing a report as part of a statistical data analysis project uses two separate steps*: running the analysis using some software, and then copy and pasting the results into a word processing tool (as illustrated above). To really drive that point home: the problem with this approach is that much time is wasted copy and pasting things, so experimenting with different layouts or data analysis techniques is very time consuming. Copy and paste mistakes will also happen (it's not a question of if, but when) and updating reports (for example, when new data comes in) means that someone will have, again, to copy and paste the updated results into a new document.

Sweave provided (and still provides, as it is still well functioning!) a way to embed the analysis in the document itself, in this case a LaTeX source file, and R code was executed whenever the document was compiled. This gave researchers considerable time savings when it was time to update a report or drafting a research paper.

The snippet below shows the example from Leisch's paper:

```
\documentclass[a4paper]{article}

\begin{document}

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

<<>>=
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
@

which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:

\begin{center}
<<fig=TRUE,echo=FALSE>>=
boxplot(Ozone ~ Month, data = airquality)
@
\end{center}

\end{document}
```

Even if you've never seen a LaTeX source file, you should be able to figure out what's going on. The first line states what type of document we're writing. Then comes `\begin{document}` which tells the compiler where the document starts. Then comes the content. You can see that it's a mixture of plain English with R code defined inside chunks starting with `<<>>=` and ending with `@`. Finally, the documents ends with `\end{document}`. Getting a human readable PDF from this source is a two-step process: first this source gets converted into a `.tex` file and then this `.tex` file into a PDF. Sweave is included with every R installation since version 1.5.0, and still works to this day. For example, we can test that our Sweave installation works just fine by compiling the example above. This is what the final output looks like:

Let us just state that the fact that it is still possible to compile this example more than 20 years later is an incredible testament to how mature and stable this software is (both R, Sweave,

In this example we embed parts of the examples from the `kruskal.test` help page into a LaTeX document:

```
> data (airquality)
> kruskal.test(Ozone ~ Month, data = airquality)

        Kruskal-Wallis rank sum test

data:  Ozone by Month
Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06
```

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

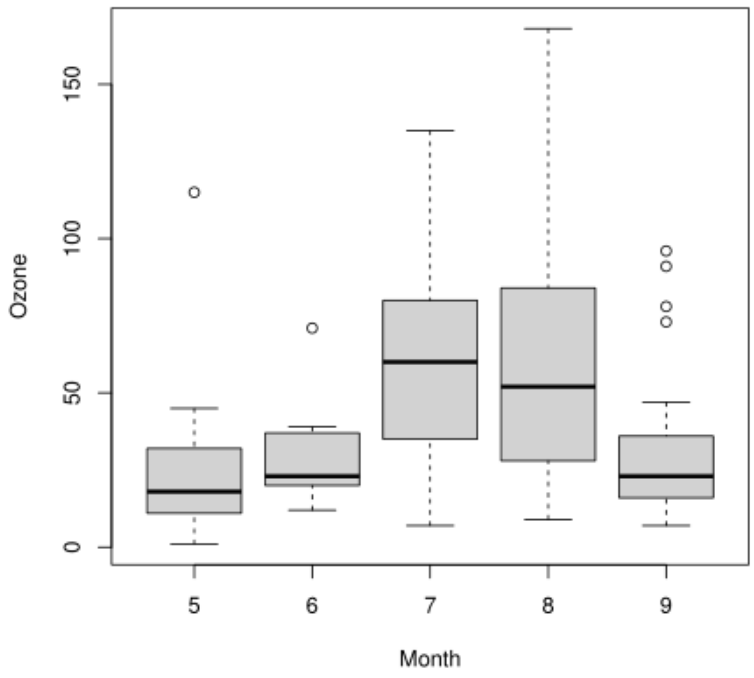


Figure 6.3: More than 20 years later, the output is still the same

and LaTeX). But as impressive as this is, LaTeX has a steep learning curve, and Leisch even advocated the use of the Emacs text editor to edit Sweave files, which also has a very steep learning curve (but this is entirely optional; for example we edited and compiled the example on the Rstudio IDE).

The next generation of literate programming tools was provided by a package called `{knitr}` in 2012. From the perspective of the user, the biggest change from Sweave is that `{knitr}` is able to use many different formats as source files. The one that became very likely the most widely used format is a flavour of the Markdown markup language, R Markdown (Rmd). But this is not the only difference with Sweave:`{knitr}` can also run code chunks for other languages, such as Python, Perl, Awk, Haskell, bash and more (Xie (2014)). Since version 1.18, `{knitr}` uses the `{reticulate}` package to provide a Python engine for the Rmd format. To illustrate the Rmd format, let's rewrite the example from Leisch's Sweave paper into it:

````
---
output: pdf_document
---

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```




which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:

```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```
````

This is what the output looks like:

Just like in a Sweave document, an Rmd source file also has a header in which authors can define a number of options. Here we only specified that we wanted a pdf document as an output file. We then copy and pasted the contents from the Sweave source, but changed the chunk delimiters from `<<>>=` and `@` to ```` ```{r} ```` to start an R chunk and ```` ``` ```` to end it. Remember; we

In this example we embed parts of the examples from the `kruskal.test` help page into a LaTeX document:

```
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  Ozone by Month
## Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06
```

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

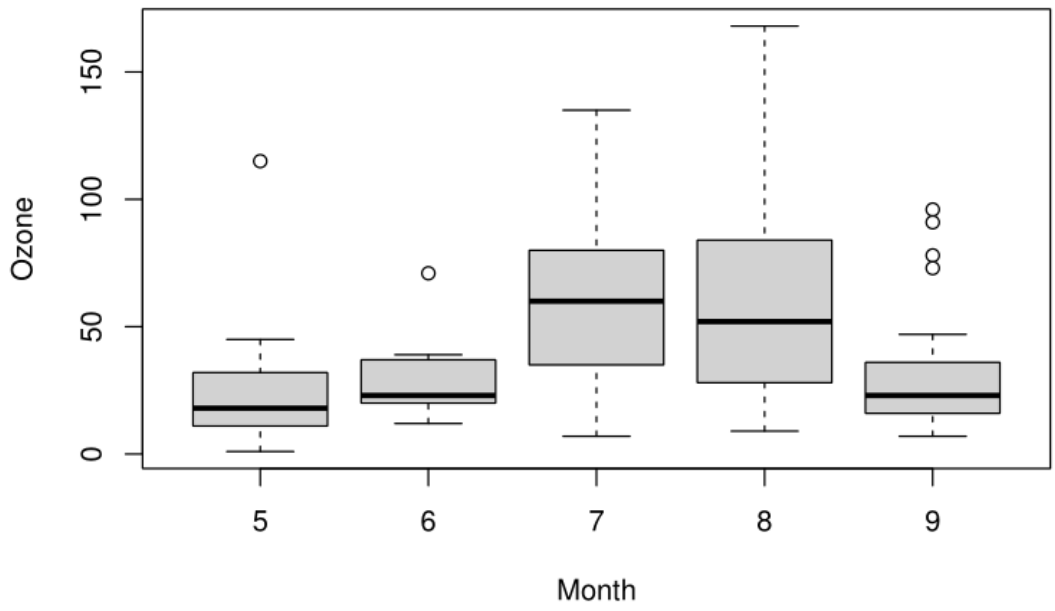


Figure 6.4: It's very close to the Sweave output

need to specify the engine in the chunk because `{knitr}` supports many engines. For example, it is possible to run a bash command by adding this chunk to the source:

````
---
output: pdf_document
---

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:


```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```

```{bash}
pwd
```
````

(bash's `pwd` command shows the current working directory). You may have noticed that we also keep two LaTeX commands in the source Rmd, `\texttt{}` and `LaTeX`. This is because Rmd files get first converted into `LaTeX` files and then into a PDF. If you're using RStudio, this document can be compiled by clicking a button or using a keyboard shortcut, but you can also use the `rmarkdown::render()` function. This function does two things transparently: it first converts the Rmd file into a source LaTeX file, and then converts it into a PDF. It is of course possible to convert the document to a Word document as well, but in this case, LaTeX commands will be ignored. Html is another widely used output format.

If you're a researcher and prefer working with LaTeX directly instead of having to switch to Markdown, you can either use Sweave, or use `{knitr}` but instead of writing your documents using the R Markdown format, you can use the `Rnw` format which is basically the same as Sweave, but uses `{knitr}` for compilation. Take a look at this example from the `{knitr}` github repository for example.

You should know that `{knitr}` makes it possible to author many, many different types of documents. It is possible to write books, blogs, package documentation (and even entire packages, as we shall see later in this book), Powerpoint slides… It is extremely powerful because we can use the same general R Markdown knowledge to build many different outputs:

Finally, the latest in literate programming for R is a new tool developed by Posit, called Quarto. If you're an R user and already know `{knitr}` and the Rmd format, you should be able to immediately use Quarto. So what's the difference? In practice and for R users not much but there are some things that Quarto is able to do out of the box for which you'd need extensions with `{knitr}`. Quarto has some nice defaults; in fact this book is written in Quarto's Markdown flavour and compiled with Quarto instead of `{knitr}` because the default Quarto output looks nicer than the default `{knitr}` output. However, there may even be things that Quarto can't do at all (at least for now) when compared to `{knitr}`. So why bother switching? Well, Quarto provides sane defaults and some nice features out of the box, and the cost of switching from the Rmd format to Quarto's Qmd format is basically 0. Also, and this is probably the biggest reason to use Quarto, Quarto is not tied to R. Quarto is actually a standalone tool that needs to be installed alongside your R installation, and works completely independently. In fact, you can use Quarto without having R installed at all, as Quarto, just like `{knitr}` supports many engines. This means that if you're primarily using Python, you can author documents with Quarto. Quarto also supports the Julia programming language and Observable JS, making it possible to include interactive visualisations into an Html document. Let's take a look at how the example from Leisch's paper looks as a Qmd file:

````
---
output: pdf
---

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```




which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:
````

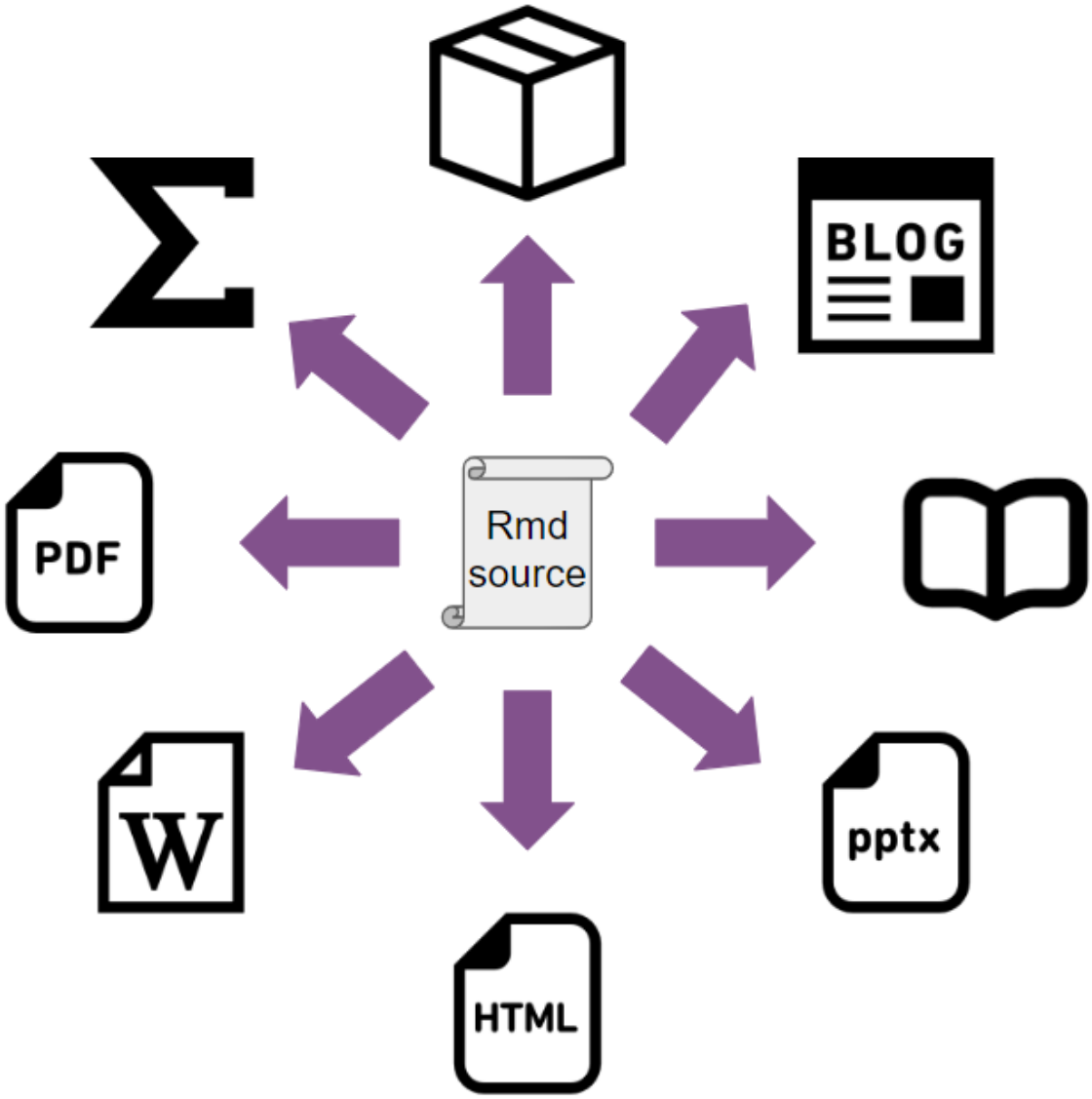


Figure 6.5: One format to rule them all

````
```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```
````

(I've omitted the bash chunk from before, not because Quarto does not support it, but to keep close to the original example from the paper.)

As you can see, it's exactly the same as the Rmd file from before. The only difference is in the header. In the Rmd file we specified the output format as:

```
---
output: pdf_document
---
```

whereas in the Qmd file we changed it to:

```
---
output: pdf
---
```

While Quarto is the latest option in literate programming, it is quite recent, and as such, I feel it might be better to stick with `{knitr}` and the Rmd format for now, so that's what we're going to use going forward. Also, the `{knitr}` and the Rmd format are here to stay, so there's little risk in keeping using it, and anyways, as already stated, if switching to Quarto becomes a necessity, the cost of switch would be very, very low. In what follows, I won't be focused on anything really `{knitr}` or Rmd specific, so should you want to use Quarto instead, you should be able to follow along without any problems at all, since the Rmd and Qmd formats have so much overlap.

In the next two sections, we will discuss how to set up and use `{knitr}` as well as give you a quick overview of the R Markdown syntax. However, we will very quickly focus on the templating capabilities of `{knitr}`: expanding text, using child documents, and parameterised

reports. These are advanced topics and not easy to tackle if you're not comfortable with R already. Just as functions and higher-order functions like `lapply()` avoid having to repeat yourself, so does templating, but for literate programming. The goal is to write functions that return literal R Markdown code, so that you can loop over these functions to build entire sections of your documents. However, the learning curve for these features is quite steep, but by now, you should have noticed that this book expects a lot from you. Keep going, and you shall be handsomely rewarded.

6.2 {knitr} basics

This section will be a very small intro to {knitr}. We are going to teach you just enough to get started, and we are going to focus on the Rmd format. There are many resources out there that you can use if you want to dig deeper, for instance the R Markdown website from Posit, or the R Markdown: The Definitive Guide and R Markdown Cookbook eBooks. We will also not assume that you are using the RStudio IDE and give you instead the lower level commands to render documents. If you use RStudio and want to know how to use it effectively to author Rmd documents, you should take a look at Quick Tour page. In fact, this section will basically focus on the same topics, but without focusing on how to use RStudio.

6.2.1 Set up

The first step is to install the {knitr} and the {rmarkdown} packages. That's easy, just type:

```
install.packages("rmarkdown")
```

in an R console. Since {knitr} is required to install {rmarkdown}, it gets installed automatically. If you want to compile PDF documents, you should also have a working LaTeX distribution. You can skip this next part if you're only interested in generating PDF and Word files. For what follows, we will only be rendering Html documents, so no need to install LaTeX (by the way, you don't even need a working Word installation to compile documents to the `docx` format). However, if you already have a working LaTeX installation, you shouldn't have to do anything else to generate PDF documents. If you don't have a working LaTeX distribution, then Yihui Xie, the creator of {knitr} created an R package called {tinytex} that makes it extremely easy to install a LaTeX distribution. In fact, this is the way I recommend installing LaTeX even if you're not an R user (it is possible to use the tinytex distribution without R; it's just that the {tinytex} R package provides many functions that makes installing and maintaining it very easy). Simply run these commands in an R console to get started:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

and that's it! If you need to install specific LaTeX packages, then refer to the **Maintenance** section on tinytex's website. For example, to compile the example from Leisch's article on Sweave discussed previously, we had to install the `grfext` LaTeX package (as explained by the error output in the console when we tried compiling). So, we simply needed to run the following command to get it:

```
tlmgr_install("grfext")
```

After you've installed `{knitr}`, `{rmarkdown}` and, optionally, `{tinytex}`, simply try to compile the following document:

```
---
output: html_document
---

# Document title

## Section title

### Subsection title

This is **bold** text. This is *text in italics*.

My favourite programming language for statistics is ~~SAS~~ R.
```

save this document into a file called `rmd_intro.rmd` using you're favourite text editor. Then render it into an Html file by running the following command in the R console:

```
rmarkdown::render("path/to/rmd_test.rmd")
```

This should create a file called `rmd_test.html`; open it with your web browser and you should see the following:

Congratulations, you just *knitted* your first Rmd document!

6.2.2 Markdown ultrabasics

R Markdown is a flavour of Markdown, which means that you should know some Markdown to really take full advantage of R Markdown. The example document from before should have

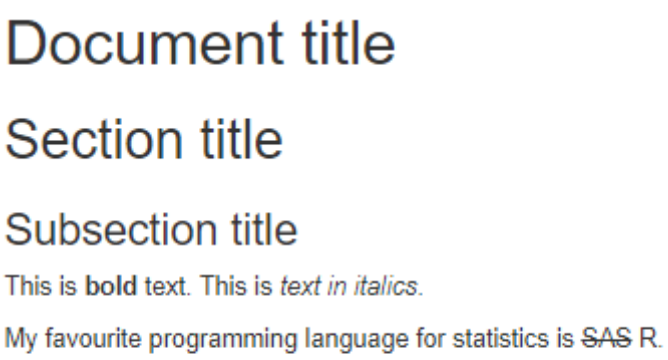


Figure 6.6: It's very close to the Sweave output

already shown you some basics: titles, sections and subsections all start with a `#` and the depth level is determined by the number of `#`s. For bold text, simply put the words in between `**` and for italics use only one `*`. If you want ***bold and italics***, use `***`. The original designer of Markdown did not think that underlining text was important, so there is no *easy* way of doing it unfortunately. For this, you need to use a somewhat hidden feature; without going into too much technical details, the program that converts Rmd files to the final output format is called Pandoc, and it's possible to use some of Pandoc's features to format text. For example, for underlining:

```
[This is some underlined text in a R Markdown document]{.underline}
```

This will underline the text between square brackets.[1]

The next step is actually to mix code and prose. As you've seen from Leisch's canonical example, this is quite easily achieved by using R code chunks. The R Markdown example below shows various code chunks alongside some options. For example, a code chunk that uses the `echo = FALSE` option will not appear (but the output of the computation will):

```
---
title: "Document title"
output: html_document
date: "2023-01-28"
---
```

[1] https://stackoverflow.com/a/68690065/1298051

````
# R code chunks

This below is an R code chunk:

```{r}
data(mtcars)

plot(mtcars)
```



The code chunk above will appear in the final output. The code chunk below will be hidden:

```{r, echo = FALSE}
data(iris)

plot(iris)
```



This next code chunk will not be evaluated:

```{r, eval = FALSE}
data(titanic)

str(titanic)
```



The last one runs, but code and output from the code is not shown in the final
document. This is useful for loading libraries and hiding startup messages:

```{r, include = FALSE}
library(dplyr)
```
````

If you use RStudio and create a new R Markdown file from the menu, a new R Markdown file is generated for you to fill out. The first R chunk is this one:

````
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
````

This is an R chunk named `setup` and with the option `include = FALSE`. Naming chunks is optional, but we are going to make use of this later on. What this chunk runs is one line of code that defines a global option to show all chunks by default (which is the default behaviour). You can change `TRUE` to `FALSE` if you want to hide every code chunk instead (if you're using Quarto, global options are set differently).

Something else you might have noticed in the previous example, is that we've added some more content in the header:

```
---
title: "Document title"
output: html_document
date: "2023-01-28"
---
```

There are several other options available that you can define in the header. Later on, when we'll be building our project together, we will provide some more options (like having a table of contents).

The finish this part on code chunks, you should know about inline code chunks. Take a look at the following example:

````
---
title: "Document title"
output: html_document
date: "2023-01-28"
---

# R code chunks

```{r, echo = FALSE}
data(iris)
```
````

```
The iris dataset has `r nrow(iris)` rows.
```

The last sentence from this example has an inline code chunk. This quite useful, as it allows to parameterise sentences and paragraphs, and thus avoids needing to copy and paste (and we will go quite far into how to avoid copy and pasting, thanks to more advanced features we will shortly discuss).

To finish this crash course, you should know that to use footnotes you need to write the following:

```
This sentence has a footnote.[^1]

[^1]: This is the footnote.
```

and that you can write LaTeX formulas as well. For example, add the following into the the example from before and render either a PDF or a html document (don't put the LaTeX formula below inside a chunk, simply paste it as if it were normal text. This doesn't work for Word output because Word does not support LaTeX equations):

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

Th LaTeX code above results in this equation:

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

Figure 6.7: A rendered LaTeX equation

6.3 Keeping it DRY

Remember; we never, ever, want to have to repeat ourselves. Copy and pasting is forbidden. Striving for this 0 copy and pasting will make our code much more robust and likely to be correct.

We started by using functions, as discussed in the previous chapter, but we can much farther than that. For example, suppose that we need to write a document that has the following structure:

- A title
- A section
- A table inside this section
- Another section
- Another table inside this section
- Yet another section
- Yet another table inside this section

Is there a way to automate the creation of such a document by taking advantage of the repeating structure? Of course there is. The question is not, *is it possible to do X?*, but *how to do X?*.

6.3.1 Generating R Markdown code from code

The example below is a fully working minimal example of this. Copy it inside a document titled something like `rmd_templating.Rmd` and render it. You will see that the output contains more sections than defined in the source. This is because we use templating at the end. Take some time to read the document, as the text inside explains what is going on:

````
---
title: "Templating"
output: html_document
date: "2023-01-27"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
````

````
## A function that creates tables

```{r}
create_table <- function(dataset, var){
  table(dataset[var]) |>
    knitr::kable()
}
```



The function above uses the `table()` function to create frequency tables,
and then this gets passed to the `knitr::kable()` function that produces a
good looking table for our rendered document:

```{r}
create_table(mtcars, "am")
```



Let' suppose that we want to generate a document that would look like this:

- first a section title, with the name of the variable of interest
- then the table

So it would look like this:

## Frequency table for variable: "am"

```{r}
create_table(mtcars, "am")
```



We don't want to create these sections for every variable by hand.
````

````
Instead, we can define a function that returns the R markdown code required
to create this. This is this function:

```{r}
return_section <- function(dataset, var){
  a <- knitr::knit_expand(text = c("## Frequency table for variable: {{variable}}",
                                   create_table(dataset, var)),
                          variable = var)
  cat(a, sep = "\n")
}
```




This new function, `return_section()` uses `knitr::knit_expand()` to generate R
Markdown code. Words between `{{}}` get replaced by the provided `var` argument
to the function. So when we call `return_section("am")`, `{{variable}}` is
replaced by `"am"`. `"am"` then gets passed down to `create_table()` and the
frequency table gets generated. We can now generate all the section by simply
applying our function to a list of column names:

```{r, results = "asis"}
invisible(lapply(colnames(mtcars), return_section, dataset = mtcars))
```
````

The last function, named `return_section()` uses `knitr::knit_expdand()`, which is the function that does the heavy lifting. This function returns literal R Markdown code. It returns `## Frequency table for variable: {{variable}}` which creates a level 2 section title with the text *Frequency table for variable: xxx* where the `xxx` will get replaced by the variable passed to `return_section()`. So calling `return_section(mtcars, "am")` will print the following in your console:

```
## Frequency table for variable: am
|am | Freq|
|:--|----:|
|0  |   19|
|1  |   13|
```

We now simply need to find a clever way to apply this function to each variable in the `mtcars` dataset. For this, we are going to use `lapply()` which implements a for loop (you could use `purrr::map()` just as well for this):

```
invisible(lapply(colnames(mtcars),
                 return_section,
                 dataset = mtcars))
```

This will create, for each variable in `mtcars`, the same R Markdown code as above. Notice that the R Markdown chunk where the call to `lapply()` is has the option `results = "asis"`. This is because the function returns literal Markdown code, and we don't want the parser to have to parse it again. We tell the parser "don't worry about this bit of code, it's already good". As you see, the call to `lapply()` is wrapped inside `invisible()`. This is because `return_section()` does not return anything, it just prints something to the console. No object is returned. `return_section()` is a function with only a side-effect: it changes something outside its scope. So if you don't wrap the call to `lapply()` inside `invisible()`, then a bunch of `NULL`s will also get printed (`NULL`s get returned by functions that don't return anything). To avoid this, use `invisible()` (and use `purrr::walk()` rather than `purrr::map()` if you want to use tidyverse packages and functions).

Click here to see the output.

This is not an easy topic, so take the time to play around with the example above. Try to print another table, try to generate more complex Markdown code, remove the call to `invisible()` and knit the document and see what happens with the output, replace the call to `lapply()` with `purrr::walk()` or `purrr::map()`. Really take the time to understand what is going on.

While extremely powerful, this approach using `knitr::knit_expand()` only works if your template only contains text. If you need to print something more complicated in the document, you need to use child documents instead. For example, suppose that instead of a table we wanted to show a plot made using `{ggplot2}`. This would not work, because a ggplot object is not made of text, but is a list with many elements. The `print()` method for ggplot objects then does some magic and prints a plot. But if you want to show plots using `knitr::knit_expand()`, then the contents of the list will be shown, not the plot itself. This is where child documents come in. Child documents are exactly what you think they are: they're smaller documents that get knitted and then embedded into the parent document. You can define anything within these child documents, and as such you can even use them to print more complex objects, like a ggplot object. Let's go back to the example from before and make use of a child document (for ease of presentation, we will not use a separate Rmd file, but will inline the child document into the main document). Read the Rmd example below carefully, as all the steps are explained:

```
---
title: "Templating with child documents"
output: html_document
date: "2023-01-27"
---
```

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(ggplot2)
```

A function that creates ggplots

```{r}
create_plot <- function(dataset, aesthetic){

  ggplot(dataset) +
    geom_point(aesthetic)

}
```

The function above takes a dataset and an aesthetic made using `ggplot2::aes()` to create a plot:

```{r}
create_plot(mtcars, aes(y = mpg, x = hp))
```

Let's suppose that we want to generate a document that would look like this:

- first a section title, with the dataset used;
- then a plot

So it would look like this:

Dataset used: "mtcars"

```{r}
create_plot(mtcars, aes(y = mpg, x = hp))
```

We don't want to create these sections for every aesthetic by hand.

Instead, we can make use of a child document that gets knitted separately
and then embedded in the parent document. The chunk below makes use of this trick:

```
```{r, results = "asis"}

x <- list(aes(y = mpg, x = hp),
          aes(y = mpg, x = hp, size = am))

res <- lapply(x,
              function(dataset, x){

  knitr::knit_child(text = c(

    '\n',
    '## Dataset used: `r deparse(substitute(dataset))`',
    '\n',
    '```{r, echo = F}',
    'print(create_plot(dataset, x))',
    '```'

     ),
     envir = environment(),
     quiet = TRUE)

}, dataset = mtcars)


cat(unlist(res), sep = "\n")
```
```

The child document is the `text` argument to the `knit_child()` function. `text` is literal R Markdown code: we define a level 2 header, and then an R chunk. This child document gets knitted, so we need to specify the environment in which it should get knitted. This means that the child document will get knitted in the same environment as the parent document (our current global environment). This way, every package that gets loaded and every function or variable that got defined in the parent document will also be available to the child document.

To get the dataset name as a string, we use the `deparse(substitute(dataset))` trick; this substitutes "dataset" by its bound value, so `mtcars`. But `mtcars` is an expression and we don't want it to get evaluated, or the contents of the entire dataset would be used in the title of the section. So we use `deparse()` which turns unevaluated expressions into strings.

```
We then use `lapply()` to loop over two aesthetics with an anonymous function
that encapsulates the child document. So we get two child documents that get
knitted, one per aesthetic. This gets saved into variable `res`. This is thus a
list of knitted Markdown.

Finally, we need unlist `res` to actually merge the Markdown code from the child
documents into the parent document.
```

Click here to take a look at the output.

Here again, take some time to play with the above example. Change the child document, try to print other types of output, really take your time to understand this. To know more about child documents, take a look at this section of the R Markdown Cookbook (Xie, Dervieux, and Riederer (2020)).

6.3.2 Tables in R Markdown documents

Getting tables right in Rmd documents is not always an easy task. There are several packages specifically made just for this task.

In this short section, we want to point you towards two packages that check the following boxes:

- Work the same way regardless of output format we want to knit our document into:
- Work for any type of table: summary tables, regression tables, two-way tables, etc.

Let's start with the simplest type of table, which would be showing the head of a dataset for example. `{knitr}` comes with the `kable()` function, but this function generates a very plain looking output. For something publication-worthy, we recommend the `{flextable}` package, developed by Gohel and Skintzos (2023):

```
library(flextable)

my_table <- head(mtcars)

flextable(my_table) |>
  set_caption(caption = "Head of the mtcars dataset") |>
  theme_booktabs()
```

We won't go into much detail on how `{flextable}` works, but it is very powerful, and the fact that it works for PDF, Html, Word and Powerpoint outputs is really a massive plus. If you want to learn more about `{flextable}`, there's a whole, free, ebook on it. `{flextable}` can create very complicated tables, so really take the time to dig in!

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Figure 6.8: The output of the code above

The next package is `{modelsummary}`, by Arel-Bundock (2022), and this one focuses on regression and summary tables. It is extremely powerful as well, and just like `{flextable}`, works for any type of output. It is very simple to get started:

```
library(modelsummary)

model_1 <- lm(mpg ~ hp + am, data = mtcars)
model_2 <- lm(mpg ~ hp, data = mtcars)

models <- list("Model 1" = model_1,
               "Model 2" = model_2)

modelsummary(models)
```

Here again, we won't got into much detail, but recommend instead that you read the package's website which has very detailed documentation.

These packages can help you keeping it DRY, so take some time to learn them.

6.3.3 Parametrized reports

Templating and child documents are very powerful, but sometimes you don't want to have one section dedicated to each unit of analysis within the same report, but rather, you want a complete separate report by unit of analysis. This is also possible thanks to parameterised reports.

Let's modify the example from before, which consisted in creating one section per column of the `mtcars` dataset and a frequency table, and make it now one separate report for each column. The R Markdown file will look like this:

	Model 1	Model 2
(Intercept)	26.585	30.099
	(1.425)	(1.634)
hp	−0.059	−0.068
	(0.008)	(0.010)
am	5.277	
	(1.080)	
Num.Obs.	32	32
R2	0.782	0.602
R2 Adj.	0.767	0.589
AIC	164.0	181.2
BIC	169.9	185.6
Log.Lik.	−78.003	−87.619
RMSE	2.77	3.74

Figure 6.9: The output of the code above

````
---
title: "Report for column `r params$var` of dataset `r params$dataset`"
output: html_document
date: "2023-01-27"
params:
  dataset: mtcars
  var: "am"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```




## Frequency table for `r params$var`

```{r, echo = F}
create_table <- function(dataset, var){

  dataset <- get(dataset)

  table(dataset[var]) |>
    knitr::kable()
}
```




The table below is for variable `r params$var` of dataset `r params$dataset`.

```{r}
create_table(params$dataset, params$var)
```

```{r, eval = FALSE, echo = FALSE}
# Run these lines to compile the document
# Set eval and echo to FALSE, so that this does not appear
# in the output, and does not get evaluated when knitting
rmarkdown::render(
````
````

```
              input = "param_report_example.Rmd",
              params = list(
                dataset = "mtcars",
                var = "cyl"
              )
            )

```

Save the code above into an Rmd file titled something like `param_report_example.Rmd` (preferably inside its own folder). At the end of the document, we wrote the lines to render this document inside a chunk that does not get shown to the reader, nor gets evaluated:

```
```{r, eval = F, echo = FALSE}
rmarkdown::render(
              input = "param_report_example.Rmd",
              params = list(
                dataset = "mtcars",
                var = "cyl"
              )
            )
```
```

You need to run these lines yourself to knit the document.

This will pass the list `params` with elements "mtcars" and "cyl" down to the report. Every `params$dataset` and `params$var` in the report gets replaced by "mtcars" and "cyl" respectively. Also, notice that in the header of the document, we defined default values for the params. Something else you need to be aware of, is that the function `create_table()` inside the report is slightly different than before. It now starts with the following line:

```
dataset <- get(dataset)
```

Let's break this down. `params$dataset` contains the string "mtcars". I made the decision to pass the dataset as a string, so that I could use it in the title of the document. But then, inside the `create_table()` function, I have the following code:

```
dataset[var]
```

`dataset` can't be a string here, but needs to be a variable name, so `mtcars` and not "mtcars". This means that I need to *convert* that string into a name. `get()` searches an object by name, and then makes it possible to save it to a new variable called `dataset`. The rest of the function is then the same as before. This little difficulty can be avoided by hard-coding the dataset

inside the R Markdown file, or by passing the dataset as the `params$dataset` and not the string, in the render function. However, if you pass down the name of the dataset as a variable instead of the dataset name as a string, then you need to covert it to a string if you want to use it in the text (so `mtcars` to "mtcars", using `deparse(substitute(dataset))` as in child documents example).

If you instead want to create one report per variable, you could compile all the documents at once with:

````
```{r, eval = F, echo = F}
columns <- colnames(mtcars)

lapply(columns,
  (\(x)rmarkdown::render(
                    input = "param_report_example.Rmd",
                    output_file = paste0("param_report_example_", x, ".html"),
                    params = list(
                      dataset = "mtcars",
                      var = x
                    )
                  )
  )
)
```
````

By now, this should not intimidate you anymore; we use `lapply()` to loop over a list of column names (that we get using `colnames()`). Because we don't want to overwrite the report we need to change the name of the output file. We do so by using `paste0()` which creates a new string that contains the variable name, so each report gets its own name. `x` inside the `paste0()` function is each element, one after the other, of the `columns` variable we defined first. Think of it as the `i` in a for loop. We then must also pass this to the `params` list, hence the `var = x`. The complete call to `rmarkdown::render()` is wrapped inside an anonymous function, because we need to use the argument `x` (which is each column defined in the columns list) in different places.

6.4 Conclusion

Before continuing, I highly recommend that you try running this yourself, and also that you try to build your own little parameterised reports. Maybe start by replacing "mtcars" by "iris" in the code to compile the reports and see what happens, and then when you're comfortable with parameterised reports, try templating inside a parameterised report!

It is important to not fall to the temptation of copy and pasting sections of your report, or parts of your script, instead of using these more advanced features provided by the language. It is tempting, especially under time pressure, to just copy and paste bits of code and get things done instead of writing what seems to be unnecessary code to finally achieve the same thing. The problem however, is that in practice copy and pasting code to simply get things done will come bite you sooner rather than later. Especially when you're still in the exploration/drafting phase of the project. It make take more time to set up, but once you're done, it is much easier to experiment with different parameters, test the code or even re-use the code for other projects. Not only that, but forcing you to actually think about how to set up your code in a way that avoids repeating yourself also helps with truly understanding the problem at hand. What part of the problem is constant and does not change? What does change? How often, and why? Can you also fix these parts or not? What if instead of five sections that I need to copy and paste, I had 50 sections? How could I scale that up?

Asking yourself these questions, and solving them, will ultimately make you better programmer.

Remember: don't repeat yourself!

7 Conclusion of part 1

We're at the end of part 1, and I need to congratulate you for making it this far. If you took the time to digest what we've learned up until now, you should be ready for what's coming, which should be a bit easier, at least some of the parts.

But before continuing, let's quickly summarise what we've learned so far.

We started our journey with two scripts that download and analyse data about housing in Luxembourg. We then learned about tools and programming paradigms that we will now use in part 2 to make our scripts more robust:

- Version control;
- Functional programming;
- Literate programming.

In some ways, you might think that the we've made our life unnecessarily complicated for very little gain. For example, functional programming seems to be only about putting restrictions on how you code. Same with using trunk-based development; why make it so restrictive?

What you need to understand is that these restrictions actually play a role. They force us to work in a much more structured way, which then ensures that our projects will be well-managed and ultimately reproducible. So while these techniques come with a cost, the benefits are far greater.

We will start part 2 by rewriting our scripts using what we've learned, and then, we will think about approaching the core problem differently, and structuring our project not as a series of scripts (or R Markdown files in the case of literate programming) but instead as a pipeline. Because until now, there's no pipeline still.

We will also learn about tools that capture the computational environment that was used to set up this pipeline and how to use them effectively to make sure that our project is reproducible.

Part II

Part 2: Reproducibility

The reproducibility iceberg

We are done with the first part of the book, and I think it is time to reflect on why we bothered with it at all. Why not just go straight to the reproducibility part?

Remember the introduction, where I talked about the reproducibility continuum or spectrum? It is now time to discuss this in greater detail. I propose a new analogy, *the reproducibility iceberg*:

Why an iceberg? Because the parts of the iceberg that you see, those that are obvious, are like running your analyses in a click-based environment like Excel. This is what's obvious, what's easy. No special knowledge or even training is required. All that's required is time, so people using these tools are not efficient and thus compensate by working insane hours (*I can't go home and enjoy time with my family I have to stay at the office and update the spreadsheeeeeeeeet* **clicks furiously**).

Let's go one level deeper: let's write a script. This is where we started. Our script was not too bad, it did the job. Unlike a click-based workflow, we could at least re-read it, someone else could read it, and it would be possible to run in the future but likely with some effort unless we're lucky. By that I mean that for such a script to run successfully in the future, that script cannot rely on packages that got updated in such a way that the script cannot run anymore (for example, if functions get renamed, or if their arguments get renamed). Also, if that script relies on a data source, the original authors also have to make sure that the same data source stays available. Another issue is collaborating when writing this script. Without any version control tools nor code hosting platform, collaborating on this script can very quickly turn into a nightmare.

This is where Git and Github came into play, one more level deeper. The advantage now is that collaboration was streamlined. The commit history is available to all the teammates and it is possible to revert changes, experiment new features using branches and overall manage the project. In this layer we also employ new programming paradigms to make the code of the project less verbose, using functional programming, with the added benefits of making it easier to test, document and share (which we will discuss to its fullest in this part of the book). Using literate programming, it is also much easier to go to our final output (which is usually a report).

But we can still go deeper: we need to find a way to freeze the packages that we used and make it that our project keeps using the same version of the packages regardless of *when* we run the script. We also want to make running the script as easy as possible, and ideally, *as non-interactively as possible**. Any human interaction with the pipeline is a source of errors, so that's why we also need to thouroughly and systematically test our code. These tests also need to run non-interactively.

Freezing the packages' versions is not enough though. As we shall see in part 2 of the book, installing older package versions can be a challenge. This can be the case for two reasons:

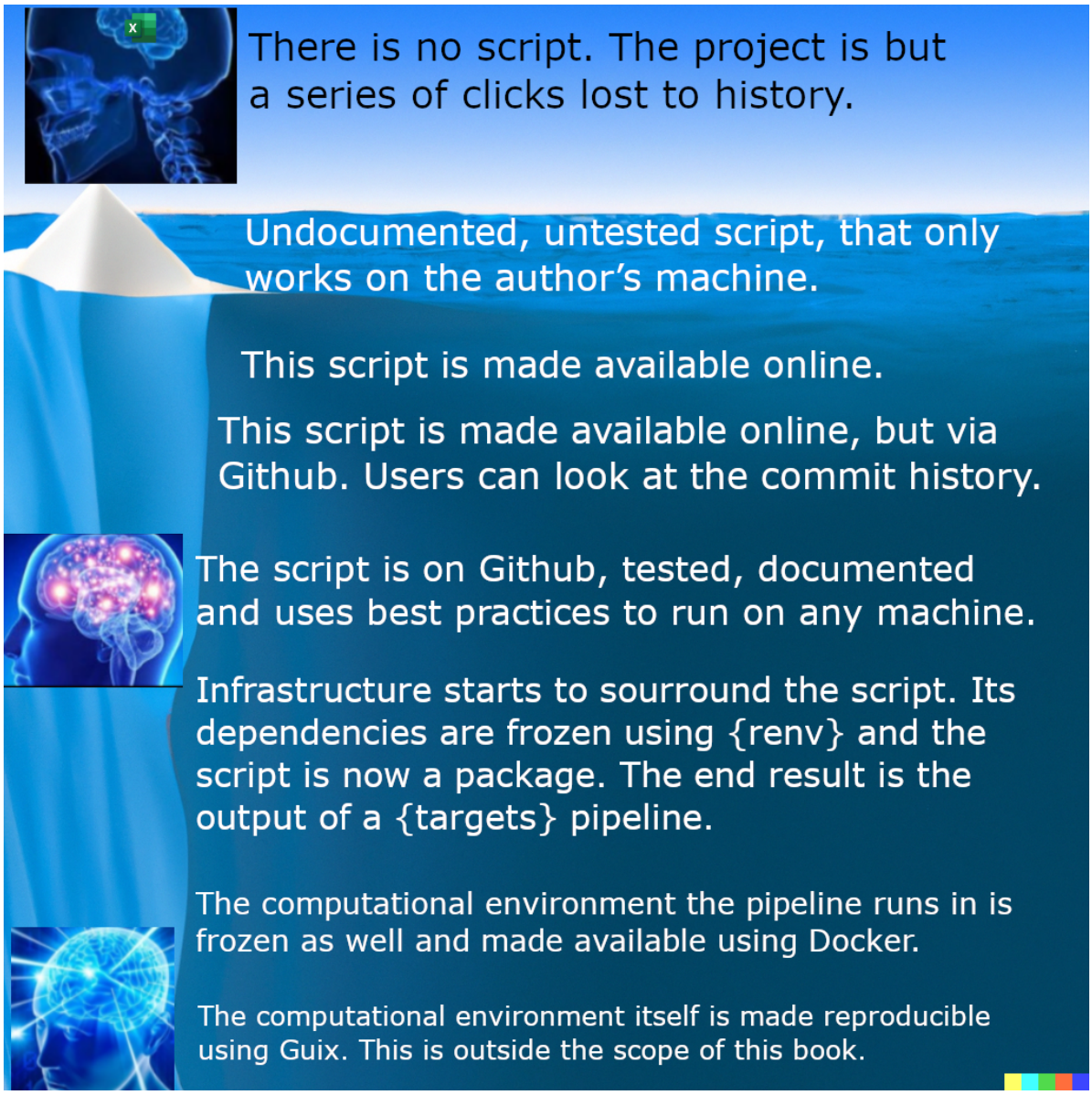


Figure 7.1: The reproducibility iceberg

- These older packages need also an older version of R, and installing old versions of R can be tricky, depending on your operating system;
- These older packages might need to get compiled and thus depend themselves on older version of development libraries needed for compilation.

So to solve this issue, we will also need a way to freeze the computational environment itself, and this is where we will use Docker.

Finally, and this is the last level of the iceberg, and not part of this book, is the need to make the building of the computational environment reproducible as well. *Guix* is the tool that enables one to do just that. However, this is a very deep topic unto itself, and there are workarounds to achieve this using Docker, so that's why we will not show be discussing *Guix*.

We will travel down the iceberg in the coming chapters. First, we will use what we've learned up until now to rewrite our project using functional and literate programming. Our project will not be two scripts anymore, but two Rmd files that we can knit and that we can then read and also send to non-technical stakeholders.

Then, we are going to turn these two Rmds files into a package. This will be done by using Sébastien Rochette's package {fusen}. `{fusen}` makes it very easy to go from our Rmd files to a package, by using what Sébastien named the *Rmarkdown first* method. If at this stage it's not clear why you would want to turn your analysis into a package, don't worry, it'll be once we're done with this chapter.

Once we have a package, we can then leverage `{testthat}` and `{assertthat}`, which are packages for unit and assertive testing respectively. At this stage, our code should be well documented, easy to share, and thouroughly tested.

Once this is achieved, we can build a true pipeline using `{targets}`, an incredibly useful library for build automation.

Once we reached this stage, this is when we can finally start introducing reproducibility concretely. The reason it will take so long to actually make our pipeline reproducible is that we need solid foundations. There is no point in making a shaky analysis reproducible.

8 Rewriting our project

In this chapter, we will use what we've learned until now to rewrite our project.

9 Packaging your code

In this chapter you're going to learn how to create your own package.

We should make clear that this does not mean publishing the package on CRAN.

9.1 Benefits of packages

9.2 Intro to packge dev

This is where fusen comes into play I guess; so we start from the Qmd file that was written before, containing the functions an the analysis, and see how we can now create a package from it, and use that file as a vignette? Copying here what Sébastien said on the matter

9.3 Document your package (?)

I guess fusen makes this process easy and leverages roxygen?

9.4 Managing package dependencies (?)

Discuss NAMESPACE and DESCRIPTION and all that. I think it's important to also discuss here how to define dependencies from remotes, not just CRAN.

9.5 Unit testing

This is where I think we should discuss unit testing

9.6 pkgdown

10 Testing your code

10.1 Assertive programming

The analysis is still in Quarto, so how could the readers of this book test their code? Copying here what Miles wrote on the subject:

'Assertive programming' is a topic that might be missing from the book. I think of it as a kind of dual of unit testing. Unit testing is for more generally applicable packaged code. But when you have functions in your analysis pipeline that operate on a very specific kind of input data, unit testing becomes kind of nonsensical because you're left to dream up endless variations of your input dataset that may never occur. It's a bit easier to flip the effort to validating the assumptions you have about your input and output data, which you can do in the pipeline functions themselves rather than separate unit testing ones. This is nice because it ensures the validation is performed in the pipeline run, and so is backed by the same reproducibility guarantees.

I think at the end of the chapter we should hint at unit testing, but leave it as a subsection of the next chapter that deals with packaging code.

https://www.brodrigues.co/blog/2022-05-26-safer_programs/

11 Build automation

Why build automation: removes cognitive load, is a form of documentation in and of itself, as Miles said

It is possible to communicate a great deal of domain knowledge in code, such that it is illuminating beyond the mere mechanical number crunching. To do this well the author needs to make use of certain styles and structures that produce code that has layers of domain specific abstraction a reader can traverse up and down as they build their understanding of the project. Functional programming style, coupled with a dependency graph as per {targets} are useful tools in this regard.

12 Introduction to reproducibility

Since we said in the intro to the book that reproducibility is on a continuum, I think that this chapter should focus on the bare minimum, which would culminate with renv

Then at the end, explain why renv is not enough (does nothing for R itself, nor the environment the code is running on)

13 Advanced topics in reproducibility

Now that the readers are familiar with renv, but also its shortcomings, we can go a step further and introduce Docker. I think some primer on the Linux command line could be included here

13.1 First steps with Docker

To write your own Dockerfile, you need some familiarity with the Linux cli, so here's...

13.2 A primer on the Linux command line

13.3 Dockrizing your project

14 Continuous integration and continuous deployment/delivery

15 Conclusion of part 2

References

Arel-Bundock, Vincent. 2022. "modelsummary: Data and Model Summaries in R." *Journal of Statistical Software* 103 (1): 1–23.

Chambers, John M. 2014. "Object-Oriented Programming, Functional Programming and R." *Statistical Science* 29 (2): 167–80.

Gohel, David, and Panagiotis Skintzos. 2023. *Flextable: Functions for Tabular Reporting*.

Hammant, Paul. 2020. *Trunk-Based Development and Branch by Abstraction*. Leanpub.

Leisch, Friedrich. 2002. "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis." In *Compstat*, edited by Wolfgang Härdle and Bernd Rönz, 575–80. Physica-Verlag HD.

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27.

Trisovic, Ana, Matthew K Lau, Thomas Pasquier, and Mercè Crosas. 2022. "A Large-Scale Study on Research Code Quality and Execution." *Scientific Data* 9 (1): 60.

Wickham, Hadley. 2019. *Advanced r*. CRC press.

Xie, Yihui. 2014. "Knitr: A Comprehensive Tool for Reproducible Research in R." In *Implementing Reproducible Computational Research*, edited by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. Chapman; Hall/CRC.

Xie, Yihui, Christophe Dervieux, and Emily Riederer. 2020. *R Markdown Cookbook*. Chapman; Hall/CRC.