

Building reproducible analytical pipelines with R

Bruno Rodrigues

06/01/2023

Table of contents

Welcome!	6
How using a few ideas from software engineering can help data scientists, analysts and researchers write reliable code	6
Preface	8
1 Introduction	13
1.1 Who is this book for?	13
1.2 What is the aim of this book?	14
1.3 Prerequisites	15
1.4 What is reproducibility?	15
1.4.1 Using open-source tools to build a RAP is a hard requirement	16
1.4.2 There are hidden dependencies that can hinder the reproducibility of a project	17
1.4.3 The requirements of a RAP	17
1.5 Are there different types of reproducibility?	18
 I Part 1: Don't Repeat Yourself	 22
Introduction	23
 2 Before we start	 24
2.1 R's anatomy	24
 3 Project start	 25
3.1 Housing in Luxembourg	25
3.2 Saving trapped data from Excel	27
3.3 Analysing the data	35
3.4 Your project is not done	36
3.4.1 How easy would it be for someone else to rerun the analysis?	36
3.4.2 How easy would it be to update the project?	37
3.4.3 How easy would it be to reuse this code for another project?	37
3.4.4 What guarantee do we have that the output is stable through time?	37
3.5 Conclusion	38

4	Version control	39
4.1	Installing Git and opening a Github account	41
4.2	Git superbasics	43
4.3	Git and Github	51
4.4	Getting to know Github	55
4.5	Conclusion	57
5	Collaborating with Github	58
5.1	Collaborating as a team using <i>trunk-based development</i>	58
5.1.1	TBD basics	58
5.1.2	Handling conflicts	62
5.1.3	Make sure you blame the right person	68
5.1.4	Simplified trunk-based development	68
5.1.5	Conclusion	69
5.2	Contributing to public repositories	69
5.3	Further reading	70
6	Functional programming	71
6.1	Introduction	71
6.1.1	The state of your program	72
6.1.2	Predictable functions	73
6.1.3	Referentially transparent and pure functions	77
6.2	Writing good functions	78
6.2.1	Functions are first-class objects	78
6.2.2	Optional arguments	82
6.2.3	Safe functions	83
6.2.4	Recursive functions	84
6.2.5	Anonymous functions	86
6.2.6	The Unix philosophy applied to R	86
6.3	Lists: a powerful data-structure	87
6.3.1	Lists all the way down	87
6.3.2	Lists can hold many things	88
6.3.3	Lists as the cure to loops	91
6.3.4	Data frames	95
6.4	Functional programming in R	107
6.4.1	Base capabilities	107
6.4.2	purrr	111
6.4.3	withr	113
6.5	Conclusion	115
7	Literate programming	116
7.1	A quick history of literate programming	117

7.2	<code>{knitr}</code> basics	122
7.2.1	Set up	123
7.2.2	Markdown ultrabasics	124
7.3	Keeping it DRY	127
7.3.1	Generating R Markdown code from code	128
7.3.2	Tables in R Markdown documents	134
7.3.3	Parametrized reports	135
7.4	Conclusion	138
8	Conclusion of part 1	140
II	Part 2: Write IT down	141
	The reproducibility iceberg	142
9	Rewriting our project	145
9.1	An Rmd for cleaning the data	146
9.2	An Rmd for analysing the data	154
9.3	Conclusion	158
10	Introduction to reproducibility	159
10.1	Recording packages' version with <code>{renv}</code>	160
10.1.1	Daily <code>{renv}</code> usage	166
10.1.2	Collaborating with <code>{renv}</code>	167
10.1.3	<code>{renv}</code> 's shortcomings	168
10.2	Becoming an R-cheologist	169
10.3	Conclusion	171
11	Packaging your code	172
11.1	Benefits of packages	172
11.2	<code>{fusen}</code> quickstart	173
11.3	Turning our Rmds into a package	178
11.4	Including datasets	185
11.5	Installing and sharing the package	186
11.5.1	Code is hosted	187
11.5.2	Code cannot be hosted	187
11.5.3	Marketing your work	188
11.6	Conclusion	189
12	Testing your code	190
12.1	Unit testing	190
12.2	Assertive programming	198
12.3	Test-driven development	203
12.4	Code coverage	203

12.5 Conclusion	204
13 Build automation	206
13.1 Introduction	206
13.2 {targets} quick-start	207
13.2.1 __targets.R's anatomy	208
13.3 A pipeline is a composition of pure functions	209
13.4 Handling files	214
13.5 The dependency graph	216
13.6 Running the pipeline in parallel	218
13.7 {targets} and RMarkdown (or Quarto)	222
13.8 Rewriting our project as a pipeline and {renv} redux	229
13.9 Some little tips before concluding	235
13.9.1 Load every target at once	235
13.9.2 Get metadata information on your pipeline	236
13.9.3 Make a target (or the whole pipeline) outdated	237
13.9.4 Customize the network's visualisation	237
13.9.5 Use targets from one pipeline in another project	237
13.9.6 Understanding this cryptic error message	238
13.10 Conclusion	238
14 Advanced topics in reproducibility	239
14.1 First steps with Docker	240
14.2 A primer on the Linux command line	240
14.3 Dockerizing your project	240
15 Continuous integration and continuous deployment/delivery	241
16 Conclusion of part 2	242
References	243

Welcome!

How using a few ideas from software engineering can help data scientists, analysts and researchers write reliable code

Data scientists, statisticians, analysts, researchers, and many other professionals write *a lot of code*.

Not only do they write a lot of code, but they must also read and review a lot of code as well. They either work in teams and need to review each other's code, or need to be able to reproduce results from past projects, be it for peer review or auditing purposes. And yet, they never, or very rarely, get taught the tools and techniques that would make the process of writing, collaborating, reviewing and reproducing projects possible.

Which is truly unfortunate because software engineers face the same challenges and solved them decades ago.

The aim of this book is to teach you how to use some of the best practices from software engineering to make your projects robust, reliable and reproducible. It doesn't matter if you work alone, in a small or in a big team. It doesn't matter if your work gets (peer-)reviewed or audited: the techniques presented in this book will make your projects more reliable and save you a lot of frustration!

As someone whose primary job is analysing data, you might think that you are not a developer. It seems as if developers are these genius types that write extremely high-quality code and create these super useful packages. The truth is that you are a developer as well. It's just that your focus is on writing code for your purposes to get your analyses going instead of writing code for others. Or at least, that's what you think. Because in *others*, your teammates are included. Reviewers and auditors are included. Any people that will read your code are included, and there **will** be people that will read your code. At the very least future you will read your code. By learning how to set up projects and write code in a way that future you will understand and not want to murder you, you will actually work towards improving the quality of your work, naturally.

The book can be read for free on <https://raps-with-r.dev> and you can buy a DRM-free Epub or PDF on [Leanpub](#)¹.

¹<https://leanpub.com/raps-with-r/>

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 *Reinhardt and Rogoff* paper. Reinhardt 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 Reinhart 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:

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. Reinhart 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, Reinhart 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. Also thanks to [David Solito](#) for proofreading certain parts of this book.

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}`;
- Package development with `{fusen}`;
- Unit and assertive testing;
- 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. But be warned, this book will require you to take the time to read it, and then type on your computer. Type *a lot*.

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/>, or if you want you can buy a DRM-free PDF or Epub over at <https://leanpub.com/raps-with-r> and will also be able to buy a physical copy, soon.

If you want to get to know me better, read my [bio](#)².

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

Enjoy!

²<https://www.brodrigues.co/about/me/>

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.

There are two main ideas in this book that you need to keep in mind at all times:

- DRY: Don't Repeat Yourself;
- WIT: Write IT down.

DRY WIT is not only the best type of humour, it is also the best way to write reproducible analytical pipelines.

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, which we often don't have anyways, 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.

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.

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:

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

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 fewer 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 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).

¹<https://www.nature.com/articles/s41597-022-01143-6>

- 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”.

The reproducibility spectrum from Peng’s 2011 paper.

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

The first idea we are going to focus on is Don't Repeat Yourself. Simply by avoiding having to repeat oneself, we will naturally implement best practices to make our pipelines reproducible.

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 are 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 engineering” 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 delivering 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 Before we start

2.1 R's anatomy

- read eval print loop
- console, editor
- packages, library
- CRAN
- .Rprofile
- workspace (don't save it)

Packages needed to follow along, look at `deps.R`

Get comfortable with paths

- absolute paths (no no)
- relative paths (yes)

`{here}` can help, but I won't discuss it.

What is a text editor, and what is text? Word is not a text editor, and work files are not text. Important to know the difference.

3 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.

3.1 Housing in Luxembourg

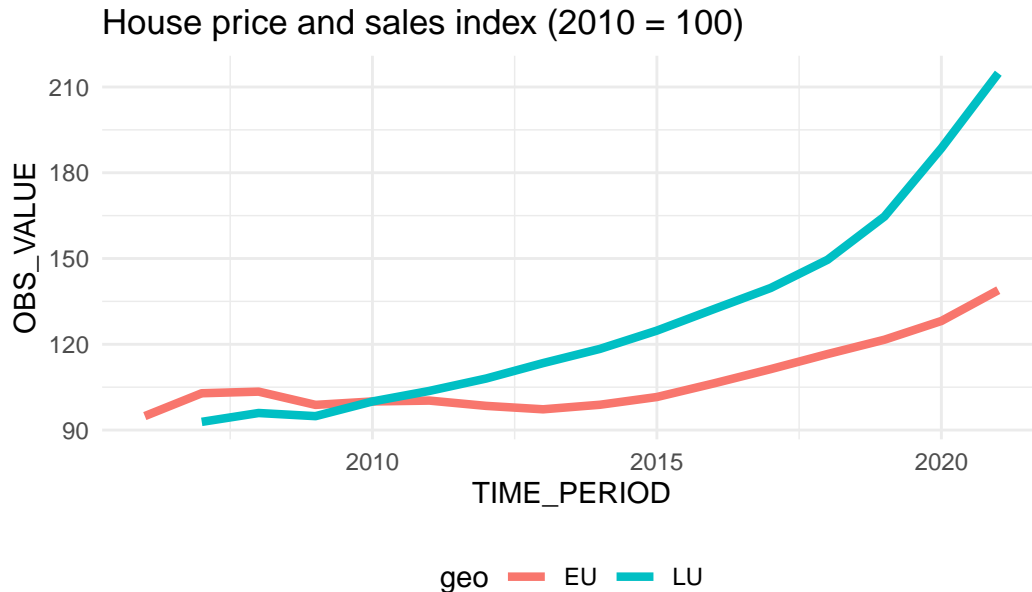
We are going to download data about house prices in Luxembourg. Luxembourg is a little Western European country the author hails from that looks like a shoe and is about the size of .98 Rhode Islands. Did you know that Luxembourg is a constitutional monarchy, and not a kingdom like Belgium, but a Grand-Duchy, 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.

Luxembourg is about as big as the US State of Rhode Island.

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.

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



Source: Eurostat

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.

3.2 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:

An Excel file meant for human eyes.

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, 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 [luxembourgish 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] "593698.310000000006" "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:

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_price_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 <chr>	year_dissolved <int>	reason <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](https://is.gd/7PhUjd)¹ 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.

3.3 Analysing the data

We are now going to analyse the data. The first thing we are going to do is compute a Laspeyres price index. This price index allows us to make comparisons through time; for

¹<https://is.gd/7PhUjd>

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 Laspeyres 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 Luxembourg, 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.

3.4 Your project is not done

3.4.1 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!

²<https://is.gd/qCJEbi>

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.

3.4.2 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 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.

3.4.3 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.

3.4.4 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.

3.5 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.

4 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.

4.1 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/
```

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.

This is your Github dashboard.

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.

4.2 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 “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` (change directory):

```
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 listing 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.addEmptyPathsSpec 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. Also, you should know that there is a special file, called `.gitignore`, that allows you to list files or folders that you want Git to ignore. This can be useful in cases where you are working with sensitive data and don't want it to be uploaded to Github. We will not use the `.gitignore` file just yet, but will do so in part two of the book. So for now, just remember that this is an option.

So let's commit our files, 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 *add message*. 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 since we were using Git we can use it to recover the files! Because we did not commit the deletion of the file, we can simply 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.

4.3 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 through some examples of how to collaborate with Git using Github in the remainder of this chapter.

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

Creating a new repository from your dashboard.

You will then land on this page:

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 world 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:

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”:

Make sure to select ‘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:

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

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

Click on your user profile's image in the top-right corner.

And then click on "SSH and GPG keys":

Go to your user settings and choose '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":

Copy the contents of the public key here.

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:

```
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:

Finally!

4.4 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.

Your repository's landing page.

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:

Several options to choose from.

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".

Choose the 'Settings' tab.

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:

Follow along to add a collaborator.

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

Look for your collaborators.

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:

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:

Click on ‘New issue’ in the ‘Issues’ tab of your project.

You will land on this interface:

Write what the issue’s about here.

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:

Try to provide as many details as possible.

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 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](#)).

4.5 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.

5 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.

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

5.1.1 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:

Bruno sees that the 'add_readme' branch has been recently updated.

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:

This screen makes it easy to see what changed.

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:

Let boss decide if this is good enough.

This is what Bruno sees now:

Github tells us that this branch can safely be merged.

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:

The owner was notified to review the pull request.

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

Time to review the pull request.

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).

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:

Choose how to protect the master branch.

By clicking the first option, more options appear:

Reviews are now required.

By choosing these options, the owner can basically enforces 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:

Time to review.

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

Check the code, and add comments if needed.

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

It's possible to add comments to lines.

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:

Good job!

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:

Nothing to complain about.

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

We're done, we can merge the pull request.

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

Let's get rid of this branch.

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

5.1.2 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.

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:

Bruno opens a pull request after finishing his changes.

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

No conflicts, for now...

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:

Let’s get rid of this branch.

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:

Conflict solving with trunk-based development.

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

First, let’s approve the changes.

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
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](https://is.gd/kTWtjr)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
```

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.

¹<https://is.gd/kTWtjr>

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.

The conflict has been gracefully solved.

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.

5.1.3 Make sure you blame the right person

If many people contribute to a single project, it might sometimes be difficult to know who changed what and when exactly. This is where the `git blame` command is useful. If you want to know who changed the file called `analysis.R` for example, simply run:

```
owner@localhost  git blame analysis.R
```

and you will see a detailed history, line by line, with the user name of the contributors and a date stamp:

```
b7f82ee1 (Bruno    2023-02-05 18:03:37 +0100 24) #Let's also compute it for the whole country
b7f82ee1 (Bruno    2023-02-05 18:03:37 +0100 25)
55804ccb (Owner    2023-02-11 22:33:20 +0000 26) country_level_data <- country_level_data |>
55804ccb (Owner    2023-02-11 22:33:20 +0000 27)   mutate(p0 = ifelse(year == "2010", average,
```

We can see that Bruno edited lines 24 and 25 on the 5th of February as part of the commit with the hash `b7f82ee1`, while the owner of the repository changed lines 26 and 27 on the 11th of February as part of the commit with the hash `55804ccb`.

Take advantage of `git blame` to have a clear overview of each file's changes.

5.1.4 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 merge his or her feature branch into 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.

5.1.5 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;
- 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.

5.2 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:

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:

Bruno needs to create a fork of the repository.

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

Bruno goes ahead with forking.

Bruno now sees the fork on his account as well:

Bruno’s fork.

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-rodriques/my_cool_project.git
```

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

Bruno fixed the typo in 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:

Bruno opens a pull request to contribute his fix upstream.

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).

The owner of the original repository can now accept Bruno’s fix.

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

The beauty of open source.

5.3 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/>.

6 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.

6.1 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 functions 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 paradigms 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*.

6.1.1 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.

6.1.2 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 feijoadada"
[[1]]
[1] "feijoadada"
```

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

```
[1] "Bruno likes feijoadada"
[[1]]
[1] "feijoadada"
```

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

```
[1] "Bruno likes feijoadada"
[[1]]
[1] "feijoadada"
```

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.

6.1.3 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.

6.2 Writing good functions

6.2.1 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 `sqrt()`:

```
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 2023-03-25 12
```

```
[3] "Total running time: 0.00107717514038086 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 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 Unix¹

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.

6.3 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.

6.3.1 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.

6.3.2 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 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.

6.3.3 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.

6.3.4 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 administrative 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 nrow
  <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
```

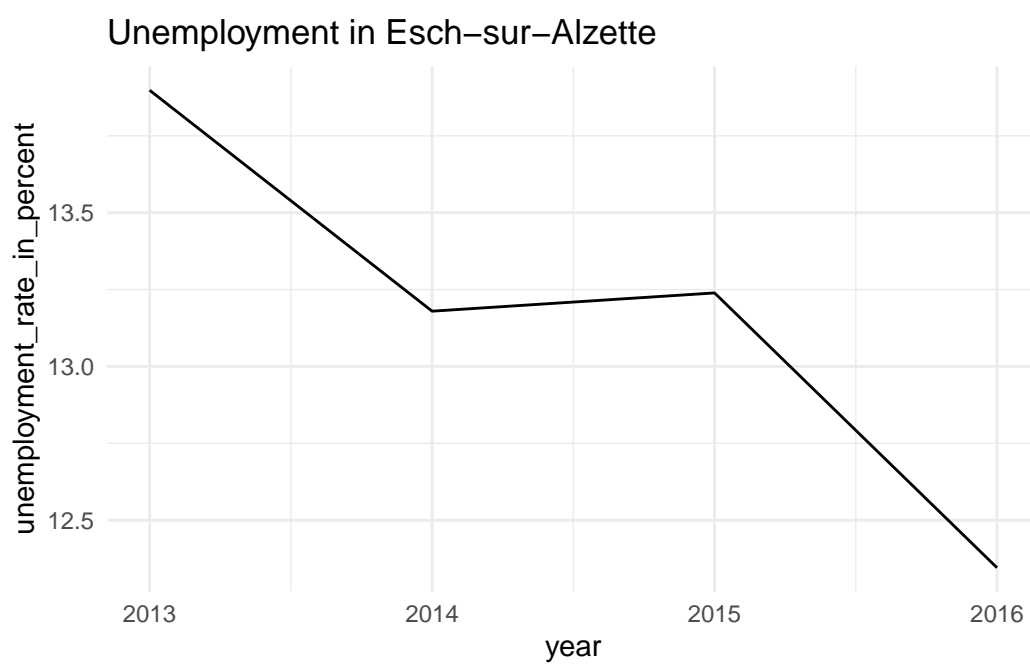
```
# A tibble: 3 x 3
  place_name      data plots
  <chr>          <list<tibble[,8]>> <list>
```

```
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
```

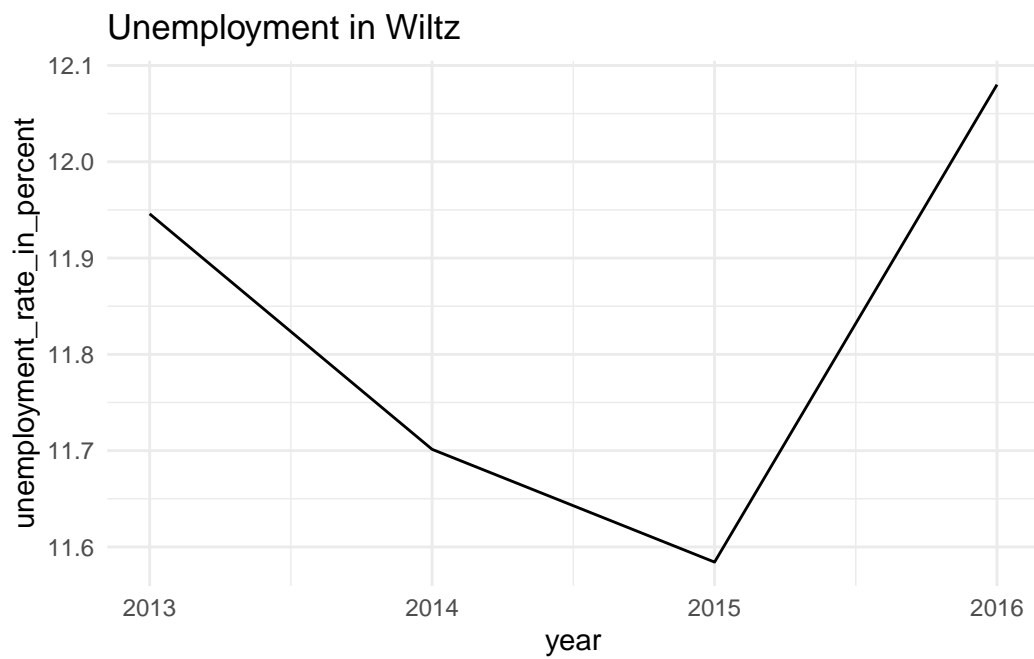
```
[[1]]
```



```
[[2]]
```



[[3]]



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]]



[[3]]



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 than 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.

6.4 Functional programming in R

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

6.4.1 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.

Sometimes you may need to run code with side-effects, but want to avoid any interaction between these side-effects and the global environment. For example, you might want to run some code that create a plot and saves it to disk, or code that creates some data and write them to disk. `local()` can be used for this. `local()` runs code in a temporary environment that gets discarded at the end:

```
local({  
  a <- 2  
})
```

Variable `a` was created inside this local environment. Checking if it exists now yields `FALSE`:

```
exists("a")
```

```
[1] FALSE
```

We will be using this technique later in the book to keep our scripts pure.

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: 0x55efcf13ec90>
<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: 0x55efd0434f80>
<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).

6.4.2 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.

6.4.3 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", "feijoadada"), 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", "feijoadada"), 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 feijoadada"
```

```
[[1]]
```

```
[1] "feijoadada"
```

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 feijoadada"
```

```
[[1]]
```

```
[1] "feijoadada"
```

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](#).

6.5 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.

7 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!

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:

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.

7.1 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:

More than 20 years later, the output is still the same.

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, 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:

It's very close to the Sweave output.

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 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:

One format to rule them all.

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:

```{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.

## 7.2 `{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.

### 7.2.1 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 your 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:

It's very close to the Sweave output.

Congratulations, you just *knitted* your first Rmd document!

## 7.2.2 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 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.<sup>1</sup>

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):

---

<sup>1</sup><https://stackoverflow.com/a/68690065/1298051>

```

title: "Document title"
output: html_document
date: "2023-01-28"

```

```
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 `nrow(iris)` rows.

The last sentence from this example has an inline code chunk. This is 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 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^{\left[-0.74\text{bigl}\left\{\frac{\omega U}{\omega 19.5}\right\}g\right]\text{bigr}\omega} \\
&= \frac{\alpha g^2}{\omega^5} \exp\text{Bigl}\left[-0.74\text{Bigl}\left\{\frac{\omega U}{\omega 19.5}\right\}g\right]\text{Bigr}\omega \\
\end{align*}
```

The LaTeX code above results in this equation:

A rendered LaTeX equation.

7.3 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?*.

7.3.1 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_expand()`, 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 NULLs will also get printed (NULLs 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)).

7.3.2 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()
```

The output of the code above.

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!

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)
```

The output of the code above.

Here again, we won't get 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.

And one last thing: if you're a researcher, take a look at the `{rticles}` package, which provides Rmd templates to write articles for many scientific journals.

7.3.3 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:

```
---
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.

## 7.4 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 may 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!

## 8 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: Write IT down**

In this part of the book, we are now going to focus on the second main idea of this book: Write IT down. We now need to acknowledge that our brains are fallible (and ageing) and thus we need to write down many safeguards to ensure that our analyses can be of high quality.

We cannot leave code quality, documentation and finally its reproducibility be to chance. We need to write down everything we need to ensure the long-term reproducibility of our pipelines.

## The reproducibility iceberg

I think it is time to reflect on why we bothered with the first part of the book at all, because for now, we didn't really learn anything about reproducibility. 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*:

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 spreadsheets 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). Furthermore, 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 with 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). We implemented DRY ideas to the fullest to ensure that our code was of high quality.

At this depth, we are at a pivotal moment: in many cases, analysts may want to stop here because there is no more time or budget left. After all, the results were obtained and shared with higher-ups. It can be difficult, in some contexts, to justify spending more time to go deeper and write tests, documentation and otherwise ensure total reproducibility. So at this stage, we will see what we can do that is *very cheap* (in both time and effort) to ensure the minimal amount of reproducibility, which is recording packages versions. Recording packages means that the exact same versions of the packages that were used originally will get used regardless of *when* in the future we rerun the analysis.

But if budget and time allows we can still go deeper, and definitely should. We also want to make running the script as easy as possible, and ideally, *as non-interactively as possible*. Any human interaction with the analysis is a source of errors, so that's why we also need to thoroughly and systematically test our code. These tests also need to run non-interactively. Also, using the tools described in part two of this book, we can actually set up the project, right from the very beginning, in a way that it will be reproducible quite naturally. By using the right tools and setting things up right, we don't really need to invest more time to make things reproducible. The project will simply be reproducible because it was engineered, that way. And I insist, at practically no additional cost!

The other problem with only recording packages's version is that in practice, it is very often not enough. This is because installing older versions of packages can be a challenge. This can be the case for two reasons:

- 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 versions 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 use `{testthat}` for unit testing, and base R functions for assertive programming. At this stage, our code should be well-documented, easy to share, and thoroughly 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.



## 9 Rewriting our project

In this chapter, we will use what we've learned until now to rewrite our project. As a reminder, here is the [save\\_data.R](https://is.gd/LyvwHb)<sup>1</sup> and the [analysis.R](https://is.gd/X7XXJg)<sup>2</sup> files from our project. The `analysis.R` file already includes one change: the one from the chapter on collaborating with Github, where Bruno wrote a function to make the plots for each commune.

If you skipped part one of the book, or for any other reason do not have a Github repository with these two files yet, then now is the time to do so. Create a repository and name it *housing\_lux* or anything you'd like, and put these two files there. I will assume that you have these files safely versioned, and will not be telling you systematically when to commit and push. Simply do so as often as you'd like! You should have a repository with a *master* or *main* branch containing these two scripts. On your computer, calling `git status` in Git Bash (on Windows) or in a terminal (for Linux and macOS) should result in this:

```
owner@localhost git status
```

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

If that's the case, congrats, we can start working. Start by creating a new branch, and call it `rmd` for example:

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

```
Switched to a new branch 'rmd'
```

We will now be working on this branch. Simply work as usual, but when pushing, make sure to push to the `rmd` branch:

```
owner@localhost git add .
owner@localhost git commit -am "some changes"
owner@localhost git push origin rmd
```

---

<sup>1</sup><https://is.gd/LyvwHb>

<sup>2</sup><https://is.gd/X7XXJg>

This will push whatever change to the `rmc` branch. By doing this, you keep the original `.R` scripts in the main branch, and they will end up with the `.Rmd` files in the `rmc` branch.

Before moving forward now is the right moment to actually discuss why you would want to convert the script into an `Rmd`. There are several reasons. First, as argued in the chapter on literate programming, a document that mixes prose and code is easier to read and share than a script. Next, since this `Rmd` file can get knitted into any type of document (PDF, Word, etc...), it also makes it easier to arrive at what interests us, the output. A script is simply a means, it's not an end. The end is a document, (in most cases) so might as well use literate programming to avoid the cursed loop mentioned in the chapter on literate programming, of changing the script, editing the document, going back to the script, etc.

But there is yet another benefit; even if the `Rmd` file is not supposed to get shared with anyone else, we will, later on, use it as our starting point for the *Rmd first* method of package development as promoted by Sébastien Rochette, the author of `{fusen}`. This means that from a single `Rmd` file, we can not only have a document that is easier to read, easier to compile into our final output and, if needed, converted quite easily into a package. If you're still not convinced, just push on, and all the pieces of the puzzle will elegantly fit in a couple chapters.

## 9.1 An `Rmd` for cleaning the data

So, let's start with the `save_data.R` script. Since we are going to use functional programming and literate programming, we are going to start from an empty `.Rmd` file. So open an empty `.Rmd` file and start with the following lines:

```

title: "Nominal house prices data in Luxembourg - Data cleaning"
author: "Put your name in here"
date: "`r Sys.Date()`"

```{r, warning=FALSE, message=FALSE}
library(dplyr)
library(ggplot2)
library(janitor)
library(purrr)
library(readxl)
library(rvest)
library(stringr)
```
```

## `## Downloading the data`

So we start by writing a header to define the title of the document, the name of the author and the current date (yes, we can use inline R code to always get current date), or you can hardcode the date as a string if you prefer.

We then load packages in a chunk with options `warning=FALSE` and `message=FALSE` which will avoid showing packages' startup messages in the knitted document.

Then we start with a new section called `## Downloading the data`. We then add a paragraph explaining from where and how we are going to download the data:

This data is downloaded from the luxembourgish [Open Data Portal](<https://data.public.lu/fr/datasets/prix-annonces-des-logements-par-commune/>) (the data set called \*Série rétrospective des prix annoncés des maisons par commune, de 2010 à 2021\*), and the original data is from the "Observatoire de l'habitat". This data contains prices for houses sold since 2010 for each luxembourgish commune.

The function below uses the permanent URL from the Open Data Portal to access the data, but I have also rehosted the data, and use my link to download the data (for archival purposes):

This is much more detailed than using comments. Then comes a function to download and get the data. This function simply wraps the lines from our original script that did the downloading and the cleaning. As a reminder, here are the lines from the original script:

```
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()

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"))

```

and here is the same code, but as a function:

```

```{r, eval = F}
get_raw_data <- function(url = "https://data.public.lu/fr/datasets/r/14b0156e-ff87-4a36-a8

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

  download.file(url,
                raw_data,
                mode = "wb") # for compatibility with Windows

  sheets <- excel_sheets(raw_data)

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

  raw_data <- map_dfr(sheets,
                     ~read_clean(raw_data,
                                  skip = 10,
                                  sheet = .)) %>%

```

```

    clean_names()

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"))

}
...

```

As you see, it's almost exactly the same code. So why use a function? Our function has the advantage that it uses the url of the data as an argument. Which means that we can use it on other datasets (let's remember that we are here focusing on prices of houses, but there's another dataset of prices of apartments) or the same, but updated dataset (let's also remember that this is a dataset that gets updated yearly). We can now more easily re-use this function later on (especially once we've turned this Rmd into a package in the next chapter). You can decide to show the source code of the function or hide it with the chunk option `include=FALSE` or `echo=FALSE` (the difference between `include` and `echo` is that `include` hides both the source code chunk and the output of that chunk). The next part of the Rmd file is simply using the function we just wrote:

```

```{r}
raw_data <- get_raw_data(url = "https://github.com/b-rodrigues/rap4all/raw/master/datasets")
```

```

We can now continue by explaining what's wrong with the data and what cleaning steps need to be taken:

We need clean the data: "Luxembourg" is "Luxembourg-ville" in 2010 and 2011, then "Luxembourg". "Pétange" is also spelled non-consistently, and we also need to convert columns to right type. We also directly remove rows where the locality contains information on the "Source":

```

```{r}
clean_raw_data <- function(raw_data){
 raw_data %>%
 mutate(locality = ifelse(grepl("Luxembourg-Ville", locality),

```

```

 "Luxembourg",
 locality),
 locality = ifelse(grepl("P.tange", locality),
 "Pétange",
 locality)
) %>%
 filter(!grepl("Source", locality)) %>%
 mutate(across(starts_with("average"), as.numeric))
}
...

```{r}
flat_data <- clean_raw_data(raw_data)
...

```

The chunk above explains what we're doing and why we're doing it, and so we write a function (based on what we already wrote). Here again, the advantage of having this as a function will make it easier to run on updated data.

We now continue with establishing a list of communes:

We now need to make sure that we got all the communes/localities in there. There were mergers in 2011, 2015 and 2018. So we need to account for these localities.

We're now scraping data from wikipedia of former Luxembourgish communes:

```

```{r}
get_former_communes <- function(url = "https://en.wikipedia.org/wiki/Communes_of_Luxembourg",
 min_year = 2009,
 table_position = 3){
 read_html(url) %>%
 html_table() %>%
 pluck(table_position) %>%
 clean_names() %>%
 filter(year_dissolved > min_year)
}
...

```{r}
former_communes <- get_former_communes()
...

```

We can scrape current communes:

```
```{r}
get_current_communes <- function(url = "https://en.wikipedia.org/wiki/List_of_communes_of_
 table_position = 1){

 read_html(url) %>%
 html_table() %>%
 pluck(table_position) %>%
 clean_names()
}

```

```{r}
current_communes <- get_current_communes()
```
```

This is quite a long chunk, but there is nothing new in here, so I won't explain it line by line. What's important is to notice that the code doing the actual work is all being wrapped inside functions. I reiterate: this will make reusing, testing and documenting much easier later on. Using the object `former_communes` and `current_communes` we can now build the complete list:

Let's now create a list of all communes:

```
```{r}
get_test_communes <- function(former_communes, current_communes){

 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"

 communes
}
```

```

}

...

```{r}
former_communes <- get_former_communes()
current_communes <- get_current_communes()

communes <- get_test_communes(former_communes, current_communes)
...

```

Once again, we write a function for this. We need to merge these two lists, and need to make sure that the spelling of the communes' names is unified between this list and between the communes' names in the data.

We now run the actual test:

Let's test to see if all the communes from our dataset are represented.

```

```{r}
setdiff(flat_data$locality, communes)
...

```

If the above code doesn't show any communes, then this means that we are accounting for every commune.

This test is quite simple, and we will see how to create something a bit more robust and useful later on.

Now, let's extract the national average from the data and create a separate dataset with the national level data:

Let's keep the national average in another dataset:

```

```{r}
make_country_level_data <- function(flat_data){
  country_level <- flat_data %>%
    filter(grepl("nationale", locality)) %>%
    select(-n_offers)

  offers_country <- flat_data %>%
    filter(grepl("Total d.offres", locality)) %>%

```



```

      select(year, n_offers)

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

}

...

```{r}
country_level_data <- make_country_level_data(flat_data)
...

```

and finally, let's do the same but for the commune level data:

```

We can finish cleaning the commune data:

```{r}
make_commune_level_data <- function(flat_data){
  flat_data %>%
    filter(!grepl("nationale|offres", locality),
           !is.na(locality))
}

...

```{r}
commune_level_data <- make_commune_level_data(flat_data)
...

```

We can finish with a chunk to save the data to disk:

```

We now save the dataset in a folder for further analysis (keep chunk option to
`eval = F` to avoid running it when knitting):

```{r, eval = F}
write.csv(commune_level_data,
          "datasets/house_prices_commune_level_data.csv",
          row.names = FALSE)
write.csv(country_level_data,
          "datasets/house_prices_country_level_data.csv",

```

```
...      row.names = FALSE)
```

This last chunk is something I like to add to my Rmd files, but instead of showing it in the final document but not evaluating its contents using the chunk option `eval = F` you could hide it completely as well, so it doesn't appear in the compiled document. The first time you compile this document, you could change the option to `eval = T`, so that the data gets written to disk, and then change it to `eval = F` to avoid overwriting the data on subsequent knittings. This is up to you, and also who the audience of the knitted output is (do they want to see this chunk at all?).

Ok, and that's it. You can take a look at the finalised file [here](#)³. You can now remove the `save_data.R` script, as you have successfully ported the code over to a Rmd.

If you have not done it yet, you can commit these changes and push.

Let's now do the same thing for the analysis script.

9.2 An Rmd for analysing the data

We will follow the same steps as before to convert the analysis script into an analysis markdown. Instead of showing the whole file here, I will instead show you two important points.

The first point is removing redundancy. In the original script, we had the following lines:

```
#Let's compute the Laspeyeres index for each commune:

commune_level_data <- commune_level_data %>%
  group_by(locality) %>%
  mutate(p0 = ifelse(year == "2010", average_price_nominal_euros, NA)) %>%
  fill(p0, .direction = "down") %>%
  mutate(p0_m2 = ifelse(year == "2010", average_price_m2_nominal_euros, NA)) %>%
  fill(p0_m2, .direction = "down") %>%
  ungroup() %>%
  mutate(pl = average_price_nominal_euros/p0*100,
         pl_m2 = average_price_m2_nominal_euros/p0_m2*100)

#Let's also compute it for the whole country:

country_level_data <- country_level_data %>%
```

³<https://is.gd/eBbcsR>

```

mutate(p0 = ifelse(year == "2010", average_price_nominal_euros, NA)) %>%
fill(p0, .direction = "down") %>%
mutate(p0_m2 = ifelse(year == "2010", average_price_m2_nominal_euros, NA)) %>%
fill(p0_m2, .direction = "down") %>%
mutate(pl = average_price_nominal_euros/p0*100,
       pl_m2 = average_price_m2_nominal_euros/p0_m2*100)

```

As you can see, this is almost exactly twice the same code. The only difference is that we need to group by commune when computing the Laspeyeres index for the communes (remember, this index will make it easy to make comparisons). Instead of repeating 99% of the lines, we can create a function that will group the data if the data is the commune level data, and not group the data if it's the national data. Here is this function:

```

get_laspeyeres <- function(dataset, start_year = "2010"){

  which_dataset <- deparse(substitute(dataset))

  group_var <- if(grepl("commune", which_dataset)){
    quo(locality)
  } else {
    NULL
  }

  dataset %>%
    group_by(!group_var) %>%
    mutate(p0 = ifelse(year == start_year,
                      average_price_nominal_euros,
                      NA)) %>%
    fill(p0, .direction = "down") %>%
    mutate(p0_m2 = ifelse(year == start_year,
                          average_price_m2_nominal_euros,
                          NA)) %>%
    fill(p0_m2, .direction = "down") %>%
    ungroup() %>%
    mutate(pl = average_price_nominal_euros/p0*100,
           pl_m2 = average_price_m2_nominal_euros/p0_m2*100)

}

```

So, the first step is naming the function. We'll call it `get_laspeyeres()`, and it'll be a function of two arguments. The first is the data (commune or national level data) and the second is the starting date of the data. This second argument is has a default value of "2010". This is the year the data starts, and thus the year the Laspeyeres index will have a value of 100.

The following lines are probably the most complicated:

```
which_dataset <- deparse(substitute(dataset))

group_var <- if(grepl("commune", which_dataset)){
  quo(locality)
} else {
  NULL
}
```

The first line replaces the variable `dataset` by its bound value (that's what `substitute()` does) for example, `commune_level_data`, and then converts this variable name into a string (using `deparse()`). So when the user provides `commune_level_data`, `which_dataset` will be defined as equal to `"commune_level_data"`. We then use this string to detect whether the data needs to be grouped or not. So if we detect the word "commune" in the `which_dataset` variable, we set the grouping variable to `locality`, if not to `NULL`. But you might have a question: why is `locality` given as an input to `quo()`, and what is `quo()`.

A simple explanation: `locality` is a variable in the `commune_level_dataset`. If we don't *quote* it using `quo()`, our function will look for a variable called `locality` in body of the function, but since there is no variable defined that is called `locality` in there, the function will look for this variable in the global environment. But this is not a variable defined in the global environment, it is a column in our dataset. So we need a way to tell this to the column: *don't worry about evaluating this just yet, I'll tell you when it's time*.

So by using `quo()`, we can delay evaluation. So how can we tell the function that it's time to evaluate `locality`? This is where we need `!!`. If you take a look at the line where we group the data in the function:

```
group_by(!!group_var)
```

So if we are calling the function on `commune_level_dataset`, then `group_var` is equal to `locality`, if not it's `NULL`. `!!group_var` means that now it's time to evaluate `group_var` (or rather, `locality`). Because `!!group_var` gets replaced by `quo(locality)`, and because `group_by()` is a `{dplyr}` function that knows how to deal with quoted variables, `locality` gets looked up among the columns of the data frame. If it's `NULL` nothing happens, so the data doesn't get grouped.

This is a big topic unto itself, so if you want to know more you can start by reading the famous `{dplyr}` vignette called *Programming with dplyr* [here](https://dplyr.tidyverse.org/articles/programming.html)⁴ In case you use `{dplyr}` a lot, I recommend you do because mastering *tidy evaluation* (the name of this framework) is key to become comfortable with programming using `{dplyr}` (and other *tidyverse* packages). You can also read the chapter I wrote on this in my other [free ebook](#).

⁴<https://dplyr.tidyverse.org/articles/programming.html>

The next lines of the script that we need to port over to the Rmd are quite standard, we write code to create some plots (which were already refactored into a function in the chapter on collaborating on Github). But remember, we want to have an Rmd file that can be compiled into a document that can be read by humans. This means that to make the document clear, I suggest that we create one subsection by commune that we plot. Thankfully, we have learned all about child documents in the literate programming chapter, and this is what we will be using to avoid having to repeat ourselves. The first part is simply the function that we've already wrote:

```
```{r}
make_plot <- function(commune){

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

 data_to_plot <- bind_rows(
 country_level_data,
 commune_data
)

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

```
```

Now comes the interesting part:

```
```{r, results = "asis"}
res <- lapply(communes, function(x){

 knitr::knit_child(text = c(

 '\n',
 '## Plot for commune: `r x`',
 '\n',
 '```{r, echo = F}',
 'print(make_plot(x))',
 '```',
```

```

),
 envir = environment(),
 quiet = TRUE)

 })

 cat(unlist(res), sep = "\n")

 ...

```

I won't explain this now in great detail, since that was already done in the chapter on literate programming. Before continuing, really make sure that you understand what is going on here. Take a look at the finalised file [here](#)<sup>5</sup>.

## 9.3 Conclusion

This chapter was short, but quite dense, especially when we converted the analysis script to an Rmd, because we've had to use two advanced concepts, *tidy evaluation* and Rmarkdown child documents. *Tidy* evaluation is not a topic that I wanted to discuss in this book, because it doesn't have anything to do with the main topic at hand. However, part of building a robust, reproducible pipeline is to avoid repetition. In this sense, programming with `{dplyr}` and tidy evaluation are quite important. As suggested before, take a look at the linked vignette above, and then the chapter from my other (free) ebook. This should help get you started.

The end of this chapter marks an important step: many analyses stop here, and this can be due to a variety of reasons. Maybe there's no time left to go further, and, after all, we got the results we wanted. Maybe this analysis is useful, but we don't necessarily need it to be reproducible in 5, 10 years, so all we want is to make sure that we can at least rerun it in some months or a couple of years (but be careful with this assessment, sometimes an analysis that wasn't supposed to be reproducible turns out it needs to be reproducible for way longer than expected...)

Because I want this book to be a pragmatic guide, we will now talk about inputting the least amount of effort to make your current analysis reproducible, and this is by freezing package versions, which we will do in the next chapter.

---

<sup>5</sup><https://is.gd/L2GICG>

## 10 Introduction to reproducibility

We are at a stage where the analysis is done. Converting our scripts into Rmds was quite easy to justify because writing the Rmds is also writing the report that we need to send to our boss (or our research paper, etc). But it might be harder to justify writing further documentation or package the functions we've had to write for reuse later and otherwise ensure that the analysis is and stays reproducible. So we are going to start with the least good solution to the reproducibility issue, which is recording the versions of the packages that were used. This is quite easy and quick to do and provides at least some hope that the analysis will stay reproducible. But this will not do anything to make the analysis more easily re-usable, will not improve the documentation, nor ensure that what we wrote is indeed correct. For this, we would need to write tests, which are missing from our current analysis. We only wrote one test, making sure that all the communes were accounted for. This is why going with a package (and I need to stress here that the goal of making this package is not to have it on CRAN) is so useful: packages offer us a great framework for documenting, testing and sharing our code (even if only sharing internally in your company/team, or even just future you). Also, at this stage, your analysis is much closer to being a package than you think. As you shall see in the next chapter, going from our Rmds to a fully functioning package is much easier than you expect, even if you've never written a package in your life.

So I hope that I made my point clear: it is not recommended to stop at this stage, but I also recognize that we live in the real world with real physical constraints. So because we live in this imperfect world, sometimes we need to deliver imperfect work. So let's see what we can do that is very cheap in terms of effort and time, but still allow us to have some hope of having our analysis reproducible using the `{renv}` package. It is quite easy to get started with `{renv}`: simply install it, and get a record of the used packages and their versions with one single command. This record gets saved inside a file that can then be used to restore this project's library in the future, and without interfering with the other packages that you already have installed on your machine. You see, `{renv}` creates a per-project library (a library is the set of packages installed on your machine) which means that you can have as many versions of, say, `{dplyr}` as needed (one per project). The right version of `{dplyr}` will be installed and used for the right project only, and without interfering with other installed versions.

Let's see how this works by creating such a project-wide library for our little project.

## 10.1 Recording packages' version with {renv}

So now that you've used functional and literate programming, we need to start thinking about the infrastructure surrounding our code. By infrastructure I mean:

- the R version;
- the packages used for the analysis;
- and otherwise the whole computational environment, even the computer hardware itself.

{renv} is a package that takes care of point number 2: it allows you to easily record the packages that were used for a specific project. This record is a file called `renv.lock` which will appear at the root of your project once you've set up {renv} and executed it. You can use {renv} once you're done with an analysis like in our case, or better yet, immediately at the start, as soon as you start writing `library(somePackage)`. You can keep updating the `renv.lock` file as you add or remove packages from your analysis. The `renv.lock` file can then be used to restore the exact same package library that was used for your analysis on another computer, or on the same computer but in the future.

This works because {renv} does more than simply create a list of the used packages and recording their versions inside the `renv.lock` file: it actually creates a per-project library (remember, the *library* is the set of packages installed on your computer) that is completely isolated for the main, default, R library on your machine, but also from the other {renv} libraries that you might have set up for your other projects. {renv} enables you to create **R**eproducible **E**nvironments. To save time when setting up an {renv} library, packages simply get copied over from your main library instead of being re-downloaded and re-installed (if the required packages are already installed in your default library).

To get started, install the {renv} package (make sure to start a fresh R session):

```
install.packages("renv")
```

and then go to the folder containing the Rmds we wrote together in the previous chapter. Make sure that you have the two following files in that folder:

- `save_data.Rmd`, the script that downloads and prepares the data;
- `analyse_data.Rmd`, the script that analyses the data.

Also, make sure that the changes are correctly backed up on Github.com, so if you haven't already, commit and push any change to the `rmd` branch. Because we will be experimenting with a new feature, create a new branch called `renv`. You should know the drill by now, but if not simply follow along:

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



Switched to a new branch 'renv'

We will now be working on this branch. Simply work as usual, but when pushing, make sure to push to the `renv` branch:

```
owner@localhost git add .
owner@localhost git commit -am "some changes"
owner@localhost git push origin renv
```

Once this is done, start an R session, and simply type the following in a console:

```
renv::init()
```

You should see the following:

```
* Initializing project ...
* Discovering package dependencies ... Done!
* Copying packages into the cache ... [76/76] Done!
The following package(s) will be updated in the lockfile:

CRAN =====
and then a long list of packages

The version of R recorded in the lockfile will be updated:
- R [*] -> [4.2.2]

* Lockfile written to 'path/to/housing/renv.lock'.
* Project 'path/to/housing' loaded. [renv 0.16.0]
* renv activated -- please restart the R session.
```

Let's take a look at the files that were created (if you prefer using your file browser, feel free to do so, but I prefer the command line):

```
owner@localhost ls -la
```

```
total 1070
drwxr-xr-x 1 LLP685 Domain Users 0 Feb 27 12:44 .
drwxr-xr-x 1 LLP685 Domain Users 0 Feb 27 12:35 ..
-rw-r--r-- 1 LLP685 Domain Users 27 Feb 27 12:44 .Rprofile
drwxr-xr-x 1 LLP685 Domain Users 0 Feb 27 12:40 .git
-rw-r--r-- 1 LLP685 Domain Users 306 Feb 27 12:35 README.md
```

```
-rw-r--r-- 1 LLP685 Domain Users 2398 Feb 27 12:38 analyse_data.Rmd
drwxr-xr-x 1 LLP685 Domain Users 0 Feb 27 12:44 renv
-rw-r--r-- 1 LLP685 Domain Users 20502 Feb 27 12:44 renv.lock
-rw-r--r-- 1 LLP685 Domain Users 6378 Feb 27 12:38 save_data.Rmd
```

As you can see, there are two new files and one folder. The files are the `renv.lock` file that I mentioned before and a file called `.Rprofile`. The folder is simply called `renv`. The `renv.lock` is the file that lists all the packages used for the analysis. `.Rprofile` files are files that get read by R automatically at startup (as discussed at the very beginning of part one of this book). You should have a system-wide one that gets read on startups of R, but if R discovers an `.Rprofile` file in the directory it starts on, then that file gets read instead. Let's see the contents of this file (you can open this file in any text editor, like Notepad on Windows, but then again I prefer the command line):

```
cat .Rprofile
```

```
source("renv/activate.R")
```

This file runs a script on startup called `activate.R`, which you can find in the `renv` folder. Let's take a look at the contents of this folder:

```
ls renv
```

```
activate.R library settings.dcf
```

So inside the `renv` folder, there is another folder called `library`: this is the folder that contains our isolated library for just this project. That's something that we would not want to back up on Github as it grows quite large. So here is the right moment to introduce the `.gitignore` file (notice the `.` at the start of the name). This is a file that sits in the root of the directory, and simply lists files and folders that you want Git to ignore. Any file listed there will not get pushed to Github. Create this file using your favourite text editor and simply add the following:

```
renv/
```

This will prevent the `renv/` folder from being tracked and backed up. If you are working with sensitive, or large, data, you can also add the folder where this data is to the `.gitignore` file as well.

`activate.R` is the script that tells R to use the packages from the `library` folder at startup. The other file, `settings.dcf` contains general settings for `{renv}` which you can safely ignore.

Let's start a fresh R session in our project's directory; you should see the following startup message:

```
* Project 'path/to/housing' loaded. [renv 0.16.0]
```

This means that this R session will use the packages installed in the isolated library we've just created. Let's now take a look at the `renv.lock` file:

```
cat renv.lock
```

```
{
 "R": {
 "Version": "4.2.2",
 "Repositories": [
 {
 "Name": "CRAN",
 "URL": "https://packagemanager.rstudio.com/all/latest"
 }
]
 },
 "Packages": {
 "MASS": {
 "Package": "MASS",
 "Version": "7.3-58.1",
 "Source": "Repository",
 "Repository": "CRAN",
 "Hash": "762e1804143a332333c054759f89a706",
 "Requirements": []
 },
 "Matrix": {
 "Package": "Matrix",
 "Version": "1.5-1",
 "Source": "Repository",
 "Repository": "CRAN",
 "Hash": "539dc0c0c05636812f1080f473d2c177",
 "Requirements": [
 "lattice"
]
 }
 }
}
```

\*\*\*and many more packages\*\*\*

The `renv.lock` file is a json file listing all the packages, as well as their dependencies that are used for the project, but it started by stating the R version that was used when it was generated. It is important to remember that when you'll use `{renv}` to restore a project's library on a new machine, the R version will not be restored: so you will be running these old packages on a newer version of R, which may sometimes be a problem (but we're going to discuss this later).

So... that's it. You've generated the `renv.lock` file, which means that future you, or someone else can restore the library that you used to write this analysis. All that's required is for that person (or future you) to install `{renv}` and then use the `renv.lock` file that you generated to restore the library. Let's see how this works by cloning the following Github repository on this [link](#)<sup>1</sup> (forked from this one [here](#)<sup>2</sup>):

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

You should see a `targets-minimal` folder on your computer now. Start an R session in that folder and type the following command:

```
renv::restore()
```

You should be prompted to activate the project before restoring:

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

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

Type Y and you should see a list of packages that need to be installed. You'll get asked once more if you want to proceed, type y and watch as the packages get installed. If you pay attention to the links, you should see that many of them get pulled from the CRAN archive, for example:

```
Retrieving 'https://cloud.r-project.org/src/contrib/Archive/vroom/vroom_1.5.5.tar.gz' ...
```

Notice the word “Archive” in the url? That's because this project uses `{vroom}` 1.5.5, but as of writing (early 2023), `{vroom}` is at version 1.6.1.

---

<sup>1</sup><https://is.gd/jMVfCu>

<sup>2</sup><https://is.gd/AAnByB>

Now, maybe you've run `renv::restore()`, but the installation of the packages failed. If that's the case, let me explain what likely happened.

I tried restoring the project's library on two different machines: a Windows laptop and a Linux workstation. `renv::restore()` failed on the Windows laptop, but succeeded on the Linux workstation.

Why does that happen? Well in the case of the Windows laptop, compilation of the `{dplyr}` package failed. This is likely because my Windows laptop does not have the right version of Rtools installed. If you look inside the `renv.lock` file that came with the `targets-minimal` project, you should notice that the recorded R version is 4.1.0, but I'm running R 4.2.2 on my laptop. So libraries get compiled using Rtools 4.2 and not Rtools 4.0 (which includes the libraries for R 4.1 as well).

So in order to run this project successfully, I should install the right version of R and Rtools, and this is usually not so difficult, especially on Windows. But that might be a problem on other operating systems. Does that mean that `{renv}` is useless? No, not at all.

At a minimum, `{renv}` ensures that a project's library doesn't interfere with another project's library. This is especially useful if you're working on a project for some time (say, several months at least) and want to make sure that you can keep working on other projects in parallel. That's because what often happens is that you update your packages to use that sweet new feature from some package but when you go back to your long-term project and try to run, it, lo and behold it doesn't work anymore. This is because another function coming from some other package that also got updated and that you use in your long-term project got removed, or renamed, or simply works differently now. In this scenario, you wouldn't be troubled by trying to restore the project, since you're simply using `{renv}` to isolate the project's library (but even if you had to restore the library, that would work since you're using the same R version).

But also, apart from that already quite useful feature, `renv.lock` files provide a very useful blueprint for Docker, which we are going to explore in a future chapter. Only to give you a little taste of what's coming: since the `renv.lock` file lists the R version that was used to record the packages, we can start from a Docker image that contains the right version of R. From there, restoring the project using `renv::restore()` should succeed without issues. If you have no idea what this all means, do not worry, you will know by the end of the book, so hang in there.

So should you use `{renv}`? I see two scenarios where it makes sense:

- You're done with the project and simply want to keep a record of the packages used. Simply call `renv::init()` at the end of the project and commit and push the `renv.lock` file on Github.
- You want to use `{renv}` from the start to isolate the project's library from your whole R installation's library to avoid any interference (I would advise you to do it like this).

In the next section, we'll quickly review how to use `{renv}` on a “daily basis”.

### 10.1.1 Daily `{renv}` usage

So let's say that you start a new project and want to use `{renv}` right from the start. You start with an empty directory, and add a template `.Rmd` file, and let's say it looks like this:

```

title: "My new project"
output: html_document

```{r setup, include=FALSE}
library(dplyr)
```

Overview

Analysis
```

Before continuing, make sure that it correctly compiles into a HTML file by running `rmarkdown::render("test.Rmd")` in the correct directory.

In the `setup` chunk you load the packages that you need. Now, save this file, and start a fresh session in the same directory and run `renv::init()`. You should see the familiar prompts described above, as well as the `renv.lock` file (which will only contain `{dplyr}` and its dependencies).

Now, after the `library(dplyr)` line, add the following `library(ggplot2)` (or any other package that you use on a daily basis). Make sure to save the `.Rmd` file and try to render it again by using `rmarkdown::render("test.Rmd")` (or if you're using RStudio, by clicking the right button), but, spoiler alert, it won't work. Instead you should see this:

```
Quitting from lines 7-9 (my_new_project.Rmd)
Error in library(ggplot2) : there is no package called 'ggplot2'
```

Don't be confused: remember that `{renv}` is now activated, and that each project where `{renv}` is enabled has its own project-wide library. You may have `{ggplot2}` installed on your system-wide library, but this project does not have it yet. This means that you need to install `{ggplot2}` for your project. To do so, simply start an R session within your project

and run `install.packages("ggplot2")`. If the version installed on your system-wide library is the latest version available on CRAN, the package will simply be copied over, if not, the latest version will be installed on your project's library. You can now update the `renv.lock` file. This is done using `renv::snapshot()`; this will show you a list of new packages to record inside the `renv.lock` file and ask you to continue:

```
list of many packages over here

Do you want to proceed? [y/N]:
* Lockfile written to 'path/to/my_new_project/renv.lock'.
```

If you now open the `renv.lock` file, and look for the string `"ggplot2"` you should see it listed there alongside its dependencies. Let me reiterate: this version of `{ggplot2}` is now unique to this project. You can work on other projects with other versions of `{ggplot2}` without interfering with this one. You can even install arbitrary versions of packages using `renv::install()`. For example, to install an older version of `{data.table}`:

```
renv::install("AER@1.0-0") # this is a version from August 2008
```

But just like in the previous section, where we wanted to restore an old project that used `{renv}`, installation of older packages may fail. If you need to use old packages, there are approaches that work better, which we are also going to explore in this chapter.

Back to daily usage of `{renv}`: keep installing the required packages for your project and calling `renv::snapshot()` to keep a record of the library for reproducibility purposes. Once you're done with your project, you have two possibilities:

- You can `renv::snapshot()` one last time to make sure that every dependency is correctly accounted for;
- You update every package in the library and in the lockfile and make sure your project runs with the latest versions of every package. You then provide this updated `renv.lock` file for future use.

The second option can be interesting if your project took some time to be developed, and you want to deliver something that depends on current packages. However, only do so if you have written enough tests to detect if a package update could break your project, or else you run the risk of providing a lock file that will install packages with which your project can't actually run! If you want to play it safe, simply go for the first option.

### 10.1.2 Collaborating with `{renv}`

`{renv}` is also quite useful when collaborating. You can start the project and generate the lock file, and when your team-mates clone the repository from Github, they can get the exact

same package versions as you. You all only need to make sure that everyone is running the same R version to avoid any issues.

There is a vignette on just this that I invite you to read for more details, see [here](#)<sup>3</sup>.

### 10.1.3 {renv}'s shortcomings

In the next section, I'm going to go over two packages that make it easy to install old packages, which can be useful for reproducibility as well. But before that, let's discuss {renv}'s shortcomings (which we already alluded to before). It is quite important to understand what {renv} does and what it doesn't do, and why {renv} alone is not enough.

The first problem, and I'm repeating myself here, is that {renv} only records the R version used for the project, but does not restore it when calling `renv::restore()`. You need to install the right R version yourself. On Windows this should be fairly easy to do, but you then need to make sure that you're running the right version of R with the right scripts, which can get confusing.

There is the {rig} package that makes it easy to install and switch between R versions that you could check [out](#)<sup>4</sup> if you're interested. However, I don't think that {rig} should be used for our purposes. I believe that it is safer to use Docker instead, and we shall see how to do so in the coming chapters.

The other issue of using {renv} is that future you, or your team-mates or people that want to reproduce your results need to install packages that may be quite difficult to install, either because they're very old by now, or because their dependencies are difficult to satisfy. Have you ever tried to install a package that depended on {rJava}? Or the {rgdal} package? Installing these packages can be quite challenging, because they need specific system requirements that may be impossible for you to install (either because you don't have admin rights on your workstation, or because the required version of these system dependencies is not available anymore). Having to install these packages (and potentially quite old versions at that) can really hinder the reproducibility of your project. Here again, Docker provides a solution. Future you, your team-mates or other people simply need to be able to run a Docker container, which is a much lower bar than installing these old libraries.

I want to stress that this does not mean that {renv} is useless: we will keep using it, but together with Docker to ensure the reproducibility of our project. As I've written above already, *at a minimum {renv} ensures that a project's library doesn't interfere with another project's library* and this is in itself already quite useful.

Let's now quickly discuss two other packages before finishing this chapter, which provide an answer to the question: *how to rerun an old analysis if no renv.lock file is available?*

---

<sup>3</sup><https://is.gd/sXpWVp>

<sup>4</sup><https://is.gd/dvH2Sj>



## 10.2 Becoming an R-cheologist

So let's say that you need to run an old script, and there's no `renv.lock` file around for you to restore the packages as they were. There might still be a solution (apart from running the script on the current version on R and packages, and hope that everything goes well), but for this you need to at least know roughly *when* that script was written. Let's say that you know that this script was written back in 2017, somewhere around October. If you know that, you can use the `{rang}` and `{groundhog}` packages to download the packages as of October 2018 in a separate library and then run your script.

`{rang}` is fairly recent as of writing (February 2023) so I won't go into too much detail now, as it is likely that the package will keep evolving rapidly in the coming weeks. So if you want to use it already and follow its development, take a look at its Github repository [here](#)<sup>5</sup> and read the [preprint](#) (Chan and Schoch (2023)).

`{groundhog}` is another option that has been around for more time and is fairly easy to use. Suppose that you have a script from October 2018 that looks like this:

```
library(purrr)
library(ggplot2)

data(mtcars)

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

ggsave("/home/project/output/myplot.pdf", myplot)
```

If you want to run this script with the versions that were current in October 2017 for the `{purrr}` and `{ggplot2}` packages, you can achieve this by simply changing the `library()` calls:

```
groundhog::groundhog.library("
 library(purrr)
 library(ggplot2)",
 "2017-10-04"
)

data(mtcars)

myplot <- ggplot(mtcars) +
```

---

<sup>5</sup><https://is.gd/sQu7NV>

```
geom_line(aes(y = hp, x = mpg))

ggsave("/home/project/output/myplot.pdf", myplot)
```

but you will get the following message:

```

| IMPORTANT.
| Groundhog says: you are using R-4.2.2, but the version of R current
| for the entered date, '2017-10-04', is R-3.4.x. It is recommended
| that you either keep this date and switch to that version of R, or
| you keep the version of R you are using but switch the date to
| between '2022-04-22' and '2023-01-08'.
|
| You may bypass this R-version check by adding:
| `tolerate.R.version='4.2.2'` as an option in your groundhog.library()
| call. Please type 'OK' to confirm you have read this message.
| >ok
```

So here again, we are advised to switch to the version of R that was current at that time. If we follow the message's advice, and add `tolerate.R.version = '4.2.2'`, we may get the script to run:

```
groundhog.library("
 library(purrr)
 library(ggplot2)",
 "2017-10-04",
 tolerate.R.version = "4.2.2")

data(mtcars)

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

ggsave("/home/project/output/myplot.pdf", myplot)
```

But just like for `{renv}` (or `{rang}`), installation of the packages can fail, and for the same reasons (unmet system requirements most of the time).

So here again, the solution is to take care of the missing piece of the reproducibility puzzle, which is the whole computational environment itself.

## 10.3 Conclusion

In this chapter you had a first (maybe a bit sour taste) of reproducibility. This is because while the tools presented here are very useful, they will not be sufficient if we want our project to be truly reproducible. There are too many things that can go wrong when re-installing old package versions, so we must instead provide a way for users to not have to do it at all. This is where Docker is going to be helpful. But before that, we need to hit the development bench again. We are actually not quite done with our project; before going to full reproducibility, we should turn our analysis into a package. And so you will see, this is going to be much, much, easier than you might expect. You already did 95% of the job! There are many advantages to turning our analysis into a package, and not only from a reproducibility perspective.

# 11 Packaging your code

In this chapter, you're going to learn how to create your own package. And let me be clear right from the start: the goal here is not to convert your analysis as a package to then get it published on CRAN. No, that's not it. The goal is to convert your analysis into a package because when your analysis goes into *package development mode*, you can, as the developer, leverage many tools that will help you improve the quality of your analysis. These tools will make it easier for you to:

- document the functions you had to write for your analysis;
- test these functions;
- properly define dependencies;
- use all the code you wrote into a true reproducible pipeline.

Turning the analysis into a package will also make the separation between the software development work you had to write for your analysis (writing functions to clean data for instance) from the analysis itself much clearer. The package itself can be published on Github (if there's nothing particularly sensitive about it) and can also be very easily installed from R itself from Github, or you can store it inside your organisation and then simply install it locally.

By turning your analysis into a package you will essentially end up with two *things*:

- a well-documented, and tested package;
- an analysis that uses this package like any other package.

Making this separation will then make it easier to record dependencies of your analysis using `{renv}`, as your package will be a package like any other that needs to be recorded. And what's more, we can start with the `.Rmd` files that we have already written! The `{fusen}` package will bridge the gap between the `.Rmd` files and the package: as Sébastien Rochette, the author of `{fusen}`, says:

If you have written an `Rmd` file, you have (almost) already written a package.

## 11.1 Benefits of packages

Let's first go over the benefits of turning your analysis into a package once again, as this is crucial.

The main point is not to turn the analysis into a package to publish on CRAN (but you can if you want to). The point is that when you analyse data, more often than not you have to write a lot of custom code, and very often, you don't expect to write that much custom code. Let's think about our little project: all we wanted was to create some plots from Luxembourgish houses' price data. And yet, we had to scrape Wikipedia on two occasions, clean an Excel file, and write a test... the project was quite modest, and yet, the amount of code (and thus opportunities to make mistakes) is quite large. But, that's not something that we could have anticipated, hence why we never really start by writing a package, but a script (or rather, an `.Rmd`) instead. But then as this script grows larger and larger, we realise that we might need something else than a simple `.Rmd` file.

The other benefit of turning all this code into a package is that we get a clear separation between the code that we wrote purely to get our analysis going (what I called the *software development part* before) from the analysis itself (which would then typically consist in computing descriptive statistics, running regression or machine learning models, and visualisation). This then in turn means that we can more easily maintain and update each part separately. So the pure software development part goes into the package, which then gives us the possibility to use many great tools to ensure that our code is properly documented and tested, and then the analysis can go inside a purely reproducible pipeline.

## 11.2 {fusen} quickstart

If you haven't already, install the `{fusen}` package:

```
install.packages("fusen")
```

`{fusen}` makes the *documentation first* method proposed by Sébastien Rochette, `{fusen}`'s author, reality. The idea is to start from documentation in the form of an `.Rmd` file and go from there to a package. Let's dive right into it by starting from a template included in the `{fusen}` package. Start an R session from your home (or Documents) directory and run the following:

```
fusen::create_fusen(path = "fusen.quickstart", template = "minimal")
```

This will create a directory called `fusen.quickstart` inside your home (or Documents) directory. Inside that folder, you will find another folder called `dev/`. Let's see what's inside (I use the command line to list the files, but you're free to use your file explorer program):

```
owner@localhost ls dev/
```

```
0-dev_history.Rmd flat_minimal.Rmd
```

dev/ contains two .Rmd files, 0-dev\_history.Rmd and flat\_minimal.Rmd. They're both important, so let me explain what they do:

- flat\_minimal.Rmd is only an example, a stand-in for our own .Rmd files. When doing actual work, we will be using the Rmd file(s) that we have written before (analyse\_data.Rmd and save\_data.Rmd) instead.
- 0-dev\_history.Rmd contains lines of code that you typically run when you're developing a package. For example, a line to initialise Git for the project, a line to add some dependencies, etc, etc. The idea is to **write down everything** that you type in the console in this file. This leaves a trace of what you have been doing, and also acts as a checklist so that you can make sure that you didn't forget anything.

Before describing these files in detail, I want to show you this image taken from {fusen}'s [website](#)<sup>1</sup>:

fusen takes care of the boring stuff for you!

On the left-hand side of the image, we see the two template .Rmd files from {fusen}. 0-dev\_history.Rmd contains a chunk called `description`. This is the main chunk in that file that we need to execute to get started with {fusen}. Running this chunk will create the package's DESCRIPTION file (don't worry if you don't know about this file yet, I will explain). Then, the second file flat\_minimal.Rmd (or our very own .Rmd files) contain functions, tests, examples, and everything we need for our analysis. When we *inflate* the Rmd file, {fusen} places every piece from this .Rmd file in the right place: the functions get copied into the package's R/ folder, tests go into the tests/ folder, and so on. {fusen} simply takes care of everything for us!

But, for {fusen} to be able to work its magic, we do need to prepare our .Rmd file a bit. But don't worry, it is mostly simply giving adequate names to our code chunks. Let's take a look at the flat\_minimal.Rmd file that was just generated. If you open it in a text editor, you should see that it is a fairly normal .Rmd file. There is a comment telling you to first run the `description` chunk in the 0-dev\_history.Rmd file before changing this one. But let's keep reading flat\_minimal.Rmd. What's important comes next:

```
my_fun

```{r function-my_fun}
#' my_fun Title
#'
#' @return 1
#' @export
#'
#' @examples
```

¹<https://is.gd/5pJi2h>

```

my_fun <- function() {
  1
}
...

```{r examples-my_fun}
my_fun()
...

```{r tests-my_fun}
test_that("my_fun works", {

})
...

```

This is a section titled `my_fun`. Then comes the definition of `my_fun()`, inside a chunk titled `function-my_fun`, then comes an example, inside a chunk titled `examples-my_fun` and finally a test in a chunk titled `tests-my_fun`.

This is essentially how we need to rewrite our own `.Rmd` files to be able to use `{fusen}`. What's really nice, is that most of it is actually done already. Using `{fusen}` just *forces* us to clean up our code and define examples and tests (if we want them) more cleanly and explicitly. Also, you might have noticed that in the chunk with the function definition there are a bunch of comments that start with `#'`. These are `{roxygen2}` type comments. As the package's documentation gets built, these comments get automatically turned into the documentation you see when you type `help(my_fun)` in an R console.

So, basically, a `{fusen}`-ready `.Rmd` file is nothing more than an `.Rmd` file with some structure imposed on it. Instead of documenting your functions as simple comments, document them using `{roxygen2}` comments, which then get turned into the package's documentation automatically. Instead of writing ad-hoc tests, or worse, instead of testing your functions on your console only, and manually one by one (and we've all done this), write down the test inside the `.Rmd` file itself, right next to the function you're testing. Instead of trying your function out on some mock data in your console, write down that example inside the `.Rmd` file itself.

Write it down, write it down, write it down... you're already documenting and testing things, so why not just write it down once and for all, so you don't have to rely on your ageing, mushy brain so much? Don't make yourself remember things, just write them down! `{fusen}` gives you a perfect framework to do this. The added benefit is that it will improve your package's quality through the tests and examples that are not directly part of the analysis itself but are still required to make sure that the analysis is of high quality. So if you start messing with your functions, you have the tests right there to tell you if you introduced breaking changes.

Let's go back to the template and inflate it into a package. Open 0-dev_history.Rmd and take a look at the `description` code chunk:

```
```{r description, eval=FALSE}
Describe your package
fusen::fill_description(
 pkg = here::here(),
 fields = list(
 Title = "Build A Package From Rmarkdown File",
 Description = "Use Rmarkdown First method to build your package.
 Start your package with documentation.
 Everything can be set from a Rmarkdown file in your project.",
 `Authors@R` = c(
 person("Sebastien", "Rochette", email = "sebastien@thinkr.fr",
 role = c("aut", "cre"), comment = c(ORCID = "0000-0002-1565-9313")),
 person(given = "ThinkR", role = "cph")
)
)
)
Define License with use_*_license()
usethis::use_mit_license("Sébastien Rochette")
```
```

Running this chunk will create the package's `DESCRIPTION` file. [Here²](https://is.gd/PfvkSZ) is an example of such a file. This file provides some information on who wrote the package, the purpose of the package, as well as some metadata such as the package's version. What's also quite important are the packages listed below the keywords `Depends:`, `Imports:` and `Suggests:`. `Depends:` is where you list packages (or R versions) that must be installed for your package to work (if they're not installed, they will be installed alongside your package). This is the same with `Imports:`, and the difference with `Depends:` is most of the time irrelevant: packages listed under `Depends` will not only be loaded when you load your package, but also attached. This means that the functions from these packages will also be available to the end-user when loading your package. Packages listed under `Imports:` will only be loaded, meaning that their functions will only be available to your packages' functions, not the end-users themselves. If that's confusing, don't worry too much about it. Finally, `Suggests:` are dependencies that are not critical, usually these are only necessary if you want to run the code from the packages vignettes or examples. As you can imagine, listing the right packages under the right category can be a daunting task. But don't worry, `{fusen}` takes care of this automatically for us! Simply focus on writing your `.Rmd` files.

The last line of this chunk runs `usethis::use_mit_license()`. `{usethis}` is a package that contains many helper functions to help you develop packages. You can choose among many

²<https://is.gd/PfvkSZ>

licenses, but this is only relevant if you want to publish your package on CRAN, and you don't need to think too much about it at the start, since you can always change the license later. And if you don't want to publish your package anywhere (nor CRAN, nor Github) and keep it completely internal to your organisation, you don't need to think about it at all. My very personal take on licenses is that you should use copyleft licenses as much as possible (so licenses like the GPL) which ensure that if someone takes your code and changes it, their changes also have to be republished to the public under the GPL – but only if they wish to publish their changes. They could always keep their modifications totally private, which means that companies can, and do, use GPL'ed code in their internal products.

It's when that product gets released to the public, that the source code must be released as well. This ensures that open code stays open.

However, licenses like the MIT allow private companies to take open source and freely available code and incorporate it inside their own proprietary tools, without having to give back their modifications to the community. Some people argue that this is the *true* free license, because anyone is then also free to use any code and they also have the liberty of not having to give anything back to the community. I think that this is a very idiotic argument, and when proponents of permissive licenses like the MIT (or BSD) get their code taken and not even thanked for it (as per the license, which doesn't even force anyone to cite the software), and their software gets used for nefarious purposes, [the levels of cope are through the roof](https://is.gd/PS45xu)³ (archived link for posterity). Anyway, I got side-tracked here, let's go back to our package.

Run the code inside the description chunk in an R console (don't change anything for now, and make sure that the R session was started on the root of the project, so in the `fusen.quickstart/` folder), and see the `DESCRIPTION` file appear magically in the root of the folder (as well as the `LICENSE` file, containing the license).

For now, we can ignore the rest of the `0-dev_history.Rmd` file: actually, everything that follows is totally optional, but still useful. If you look at them, you see that lines that follow simply help you remember to do useful things, like initialising Git, creating a `Readme` file, add some usual dependencies, as so on. But let's ignore this for now, and go to the `flat_minimal.Rmd` file.

Got at the end of the file, and take a look at the chunk titled `development-inflate`. This is the chunk that will convert the `.Rmd` file into a fully functioning package. This process is called *inflating* the `.Rmd` file (because a *fusen* is a type of origami figure that you fold in a certain way, which can then get *literally* inflated into a box). Run the code in that chunk, and see your analysis become a package automagically.

If you look now at the `projects'` folder, you will see several new sub-folder:

- `R/`: the folder that contains the functions;
- `man/`: contains the functions' documentation;

³<https://is.gd/PS45xu>

- `tests/`: contains the tests;
- `vignettes/`: contains the vignettes.

Every function defined in the `flat_minimal.Rmd` file is now inside the `R/` folder; all the documentation written as `{roxygen2}` comments is now neatly inside `man/`, the tests are in `tests/`, and `flat_minimal.Rmd` has been converted to an actual vignette (without all the `development` chunks). This is now a package that can be installed immediately using `devtools::install()`, or that can be shared on Github and installed from there. Right now, without doing anything else. You can even generate a website for your package: got back to the `0-dev_history.Rmd` and check the last code chunk, under the title *Share the package*. Start a new, fresh session at the root of your project and run the two following lines from that last chunk:

```
# set and try pkgdown documentation website
usethis::use_pkgdown()
pkgdown::build_site()
```

This will build a website for your package and open your web browser and show you how it looks like. The files for this website are in the newly created `docs/` folder in the root of your package folder. This can then be hosted, for free, with Github so people can explore the package's functions and documentation without having to install the package!

11.3 Turning our Rmds into a package

Ok, so I hope to have convinced you that `{fusen}` is definitely something that you should add to your toolbox. Let's now turn our analysis into a package, but before diving right into it, let's think about it for a moment.

We have two `.Rmd` files, one for getting and cleaning the data, which we called `save_data.Rmd` and another for analysing this data, called `analyse_data.Rmd`.

In both `.Rmd` files, we defined a bunch of functions, but most of the functions were defined in the `save_data.Rmd` script. In fact, in the `analyse_data.Rmd` file we defined only two functions, `get_laspeyeres()`, the function to get the Laspeyeres price index, and `make_plot()`, the function to create the plots for our analysis.

We are faced with the following choice here:

- make both these `.Rmd` files *fusen-ready*, and inflate them both. This would put the functions from both `save_data.Rmd` and `analyse_data.Rmd` into the inflated package `R/` folder;
- put all the functions into `save_data.Rmd` and only inflate that file. The other, `analyse_data.Rmd` can then be used exclusively for the analysis *stricto sensu*.

This is really up to you, there is no right or wrong answer. You could even go for another option if you wanted. It all depends on how much time you want to invest into this. If you want to get done quickly, the first option, where you simply inflate both files is the fastest. If you have more time, the last option, where you neatly split everything might be better. I propose that we go for the second option. This way, we only have to inflate one file, and in our case here, it won't take much time anyways. It's literally only moving two code chunks from `analyse_data.Rmd` to `save_data.Rmd`. So before continuing, let's go back to our repository and switch back to the `rmd` branch that contains the `.Rmd` files (let's ignore freezing packages with `{renv}` and thus the `renv` branch for now):

```
owner@localhost  git checkout rmd
```

From, there, let's create a new branch called `fusen`:

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

```
Switched to a new branch 'fusen'
```

We will now be working on this branch. Simply work as usual, but when pushing, make sure to push to the `fusen` branch:

```
owner@localhost  git add .
owner@localhost  git commit -am "some changes"
owner@localhost  git push origin fusen
```

By now, that repository should have four branches:

- *master*, or *main* with the simple `.R` scripts;
- *rmd*, with the `.Rmd` files
- *renv*, containing the `.Rmd` files as well, and the `renv.lock` file
- *fusen*, the branch we will be using now.

If you've skipped the first part of the book, or didn't diligently create the branches and push, you can fork this [repository](https://is.gd/jGZrMF)⁴ and then clone it to start from a sane base. Switch to the `rmd` branch, and create a branch called `fusen`.

First order of business: create a `{fusen}` template in the folder. Start a fresh R session inside the `housing/` folder, and run the following:

⁴<https://is.gd/jGZrMF>

```
fusen::create_fusen(path = ".", template = "minimal", overwrite = TRUE)
```

Because we already have a folder for our project, called `housing/` we use the `"."` which essentially means “right here”. We need the `overwrite = TRUE` option because the folder exists already. Running the above command will add the `dev/` folder. Move `save_data.Rmd` inside; remember, we only want to inflate that one: `analyse_data.Rmd` will be a simple `.Rmd` that will use our package to load the needed functions and data.

Next step, move the functions `get_laspeyeres()` and `make_plot()` from `analyse_data.Rmd` to `save_data.Rmd`. Simple cut and paste these functions from one `.Rmd` to the other. Make sure `save_data.Rmd` looks something like [this](#)⁵, take a look at the end of the script to find the functions we’ve moved over. The `analyse_data.Rmd` script is exactly the same, minus the functions that we’ve just moved over.

Ok, so now, we need to make `save_data.Rmd` ready to be inflated. Take inspiration from the `flat_minimal.Rmd` that `fusen::create_fusen()` put in the `dev/` folder. This is what the end-result should [look like](#) (no worries, I’m going to explain how I got there).

Let’s start by the first function, `get_raw_data()`. If you compare the [before](#)⁶ and [after](#), the differences are that we have named the chunk `function-get_raw_data` and added documentation in the form of `{roxygen2}` comments. Naming the chunks is essential: this is how `{fusen}` knows that this chunk contains a function that should go into the `R/` folder. `{roxygen2}` are strictly speaking not required, but it is highly advised that you add them: this way, your function will get documented and users (including future you) will be able to read the documentation by typing `help(get_raw_data)`. Another difference is that I have made all the functions referentially transparent. Take a closer look at `make_plot()` in the before and after `.Rmd`’s. You will see that I’ve added two arguments to `make_plot()`, `country_level_data` and `commune_level_data`. This is really important, so don’t forget to do it!

Remember when I mentioned that the good thing about turning our analysis into a package is that it gives us a framework to develop high quality code by using nice development tools? `{roxygen2}` type comments for documentation is the first such tool in this list. When you’re writing your functions, I’m pretty sure that you add comments to it most of the time. You explain what the inputs are, what the outputs are going to be, and also how to use with a little example. Using `{fusen}` (and `{roxygen2}`), you simply continue doing the same, but with some added structure. This added structure is not costly to impose on yourself, and comes with many added benefits (in this case, free documentation!). I’m repeating myself but I really want to drive this point home: the goal is not to have to *add* code on top of what you already did. The point is to do what you always do, but within a framework.

Let’s now look at the functions’ `{roxygen2}`-type comments. The first line:

⁵<https://is.gd/n3m6In>

⁶<https://is.gd/n3m6In>

```
#' get_raw_data Gets raw nominal house price data from LU Open Data Portal
```

will create the title of the function's help page. Then come the `@param` lines (in this case we only have one):

```
#' @param url Optional: Persistent url to the data
```

This lists the parameters of the function. Here you can explain exactly what the inputs should be. Suppose that the function has several parameters and that you forget to document one. If that happens, when you will inflate the file, you will get a warning in the console:

```
inflate warings and errors: Undocumented arguments in documentation object 'get_raw_data'
  'url'
```

Then come the `@importFrom` statements. This is where you list dependencies:

```
#' @importFrom readxl excel_sheets read_excel
#' @importFrom utils download.file
#' @importFrom dplyr mutate rename select
#' @importFrom stringr str_trim
#' @importFrom janitor clean_names
#' @importFrom purrr map_dfr
```

This is important, because the statements will write the dependencies into the package's `NAMESPACE` file. This file is important, because any function defined there will be available to your package's functions when you load the package. So if your function use `dplyr::mutate()` for example, your package needs to know where to look for `mutate()`. This is where the `NAMESPACE` file comes into play. Take the opportunity to list the dependencies of your function to review them: maybe your using a package for a single dependency that you could easily remove. For example, I'm using `stringr::str_trim()` to remove whitespace around characters. But I could be using the base R function `trimws()` instead, which would remove this dependency. I'm going to keep it here, because I'm lazy though. It might seem like extra work to add these statements. But you have to see it this way: you are writing the functions here, once, that need to be available to your functions for them to work. The alternative is to have to write:

```
library("readxl")
library("utils")
library("dplyr")
library("stringr")
```

```
library("janitor")
library("purrr")
```

on top of each script that uses your functions. This gets old pretty fast.

You will also notice the following `importFrom` statement:

```
#' @importFrom utils download.file
```

`download.file()` is included in the `{utils}` package, itself included with a base installation of R. So you don't really need to specify it; but when inflating the file, you get the following message:

```
Consider adding
  importFrom("utils", "download.file")
to your NAMESPACE file.
```

hence why I've added it, to silence this message. Again, not mandatory, but why not do it?

Now comes the `@return` keyword: this simply tells your users what the function returns. If the function doesn't return anything, because it only has a side effect (for example, writing something to disk, or printing something on screen), then you could return `NULL`.

```
#' @return A data frame
```

Last but not least, the `@export` keyword:

```
#' @export
```

This makes the function available to users that load the package using `library(housing)`. If you don't add this keyword, the function will be only available to the other functions of the package. Another way to see this: functions decorated with the `@export` keyword are public, functions without it are private. But private functions don't really exist in R. You can always access a "private" function by using `:::` (three times the `:`), as in `package:::private_function()`.

The other functions are documented in the same manner, so I won't comment them here. Something else you might have noticed: I replaced every `%>%` by the base pipe `|>`. You don't have to do it, but the advantage of using the base pipe is that it removes the dependency on the `{magrittr}` package, needed for `%>%`. If you want to use `%>%`, you can keep it, but then should run the line:

```
usethis::use_pipe()
```

in the `0-dev_history.Rmd` file, which will take care of adding this dependency correctly for you (by editing the `NAMESPACE` file).

Next comes the test we wrote. As a reminder, here is how it looked like in our original `.Rmd` file:

```
Let's test to see if all the communes from our dataset are represented.
```

```
```{r}
setdiff(flat_data$locality, communes)
```
```

The objects `communes` and `flat_data` have to obviously exist for this test to pass. This was a very simple test that must be monitored interactively. If commune names are returned here, then this means that there are communes left that we need to include in our data. But remember: we are aiming at building a RAP, and don't want to have to look at it as is running to see if everything is alright. What we need is a test that returns an error if it fails. So for this we use the `{testthat}` package, and write a so-called *unit test*. We're going to deep-dive into unit testing (and assertive testing) in the next chapter, so for now, let me simply comment the test:

```
```{r tests-clean_flat_data}
We now need to check if we have them all in the data.
The test needs to be self-contained, hence
why we need to redefine the required variables:

former_communes <- get_former_communes()

current_communes <- get_current_communes()

communes <- get_test_communes(
 former_communes,
 current_communes
)

raw_data <- get_raw_data(url =
 "https://github.com/b-rodrigues/rap4all/raw/master/datasets/vente-maison-2010-2021.xlsx"
)

flat_data <- clean_raw_data(raw_data)
```

```
testthat::expect_true(
 all(communes %in% unique(flat_data$locality))
)

```

The first thing that you need to know is that tests need to be self-contained. This is why we define `former_communes` and `current_communes` again. The reason is that `{fusen}` will take this whole chunk and save it inside a script in the package's `tests/` folder. When executed, the test will run in a fresh session where the `communes` object is not defined. So that's why you need to redefine every variable the test needs to run. For the test itself, we use `testthat::expect_true()`. This function expects a piece of code that should evaluate to `TRUE`: if not, we get an error, and the whole pipeline stops here, forcing us to see what's going on. This is exactly what we want: when our code fails, it needs to fail as early and as spectacularly as possible. If you rely of future you to have to manually check console output or logs and look for errors, you deserve everything that's going to happen to you.

Under the section titled “Functions used for analysis”, I copy and pasted the functions from the `analyse_data.Rmd` and documented them as well. What's new is that I've added examples:

```
```{r examples-get_laspeyeres, eval = FALSE}
#' \dontrun{
#' commune_level_data_laspeyeres <- get_laspeyeres(commune_level_data)
#' }
---
```

But I don't want these examples to run, simply to appear in the documentation. This is because, just like for tests, examples have to be self-contained. So for this example to run successfully, I would need to redefine `commune_level_data` from scratch. I don't want to do this now, so hence why I wrapped the example around `\dontrun` and used roxygen-style comments with `#'`. I did the same with the function to plot the data.

We're almost done; take a look again at the template `flat_minimal.Rmd`. I advised you to take inspiration from it to get `save_data.Rmd` `fusen`-ready. At the end of that file, we can see this chunk:

```
```{r development-inflate, eval=FALSE}
Run but keep eval=FALSE to avoid infinite loop
Execute in the console directly
fusen::inflate(flat_file = "dev/flat_minimal.Rmd",
 vignette_name = "Minimal")

```

This chunk contains the code that we need to run, manually, to inflate the package. However,



I've removed it, and the reason is that I prefer to have it inside the `0-dev_history.Rmd` file. I think that it makes more sense to have it there. Take a look at my `0-dev_history.Rmd` [here](#)<sup>7</sup>. By reading that file, you see all the different developer actions that were taken. Your teammates, or future you could read this, and immediately understand what happened, and what was done. Under the section title “Inflate save\_data.Rmd”, you see that the chunk to inflate the `.Rmd` file and generate the package is there. I can run this chunk from `0-dev_history.Rmd` and have my package successfully generated. Something important to notice as well: my fusen-ready `.Rmd` file is simply called `save_data.Rmd`, while the generated, inflated file, that will be part of the package under the `vignettes/` folder is called `dev-save_data.Rmd`.

I suggest that you stop here, and really try to get this working as well. You can start by simply cloning your fork of this [repository](#)<sup>8</sup> I linked in the beginning of this chapter, and follow along. After inflating, take a look at the vignette generated from the inflated `dev-save_data.Rmd`, which you can find under the `vignettes/` folder. One thing you need to understand, is that the `save_data.Rmd` file that you inflate, under `dev/`, is a working file for *developers*. The generated vignette on the other hand, can be read by stakeholders other than developers as well. In my case, I've added the prefix `dev-` because this vignette deals with preparing data for including in the package, and there is not much point for a stakeholder other than a developer to read this vignette. You will notice that the generated vignette does not contain the function chunks. This is normal, because after inflating the `.Rmd` file, the functions get saved under the `R/` folder. Really take some time to understand this. Because what follows will assume that you have groked `{fusen}`.

## 11.4 Including datasets

Another difference between our initial `.Rmd` and the fusen-ready `.Rmd`, is that the fusen-ready `save_data.Rmd` file does not save the datasets as `.csv` files anymore. This is because it is much better to include them directly in the package, and make them available to users like so:

```
data("commune_level_data")
```

For this however, we need the package to already be built; only once the package exists can we include data sets. This is why we need to inflate `save_data.Rmd` first. So, how do we include data sets in a package? If you are developing packages in the usual manner (meaning, without `{fusen}`) then you have to do the following steps:

- write a script that generates the data set (and save this script inside the `data-raw/` folder for future reference)
- save the datasets inside the `data/` folder.

---

<sup>7</sup><https://is.gd/JsJJVN>

<sup>8</sup><https://is.gd/jGZrMF>

But we are using `{fusen}`, so instead, we can use the documentation first approach! And actually, the first step is done already: we have our vignette `save_data.Rmd`! Let's not forget that the whole point of `save_data.Rmd` file was, initially, to build these datasets and save them. So why not simply re-use this vignette? If you take a look at the inflated `dev-save_data.Rmd`, you will see that everything is right there! We have used it in the past to build our datasets, so of course everything is in there. So remember, we don't want to have to repeat ourselves. The vignette is right there with the code we need, so we are going to use it.

If you look at `0-dev_history.Rmd`, everything is explained under the header "Including datasets". The idea is to run the code inside the vignette, which creates our datasets, and then save these datasets in the right place using `usethis::use_data()`, mimicking the steps above. I wrapped all the code around the `local()` function to run all these steps inside a temporary, local environment. This way, any variable that gets made by knitting the vignette gets discarded once we're done.

Finally, we need to document the datasets. For this, we use another `.Rmd` file that we inflate as well. You can find it under `dev/data_doc.Rmd`, or by clicking [here](#)<sup>9</sup>. Datasets get defined inside chunks, just like functions, using `{roxygen2}`-type comments.

This basically covers what you need to know to package code. Of course, there are many other topics that we could discuss, but for our purposes, this is enough. We now know how to take advantage of the tools that make package development easy, and have diverted them for our use. If you want to develop a proper package and push it to CRAN, then I highly recommend you read the [second edition of \*R packages\*](#)<sup>10</sup> by Wickham and Bryan (2023). This book goes into all the nitty gritty details of full package development. But let me be clear: this does not mean that you cannot develop a full, CRAN-ready, package using `{fusen}`. You absolutely can! It's just that this is outside the scope of the present book.

## 11.5 Installing and sharing the package

To install the package on the same machine that you developed it, you can simply run the line `remotes::install_local()` on line 46 of the `0-dev_history.Rmd` file (ideally in a fresh R session). But how can you share it with colleagues or future you?

Now that the package is ready, you need to be able to share it. This really depends on whether you can publish the code on Github or not, or whether your company/institution has a self-hosted version control system. In this section, we're going to discover the following two scenarios: the package is hosted on Github (or in a private self-hosted version control system), or the package cannot be hosted for whatever reason.

---

<sup>9</sup><https://is.gd/wjkNAO>

<sup>10</sup><https://r-pkgs.org/>

### 11.5.1 Code is hosted

So if the code is hosted on Github (or on a self-hosted, private, version control system), users of the package can install it directly from Github. This can be done using the `{remotes}` package, like this:

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

It is also possible to install the package from a specific branch:

```
remotes::install_github("github_username/repository_name@repo_name")
```

it is even possible to install the package exactly how it was at a specific commit:

```
remotes::install_github("github_username/repository_name@repo_name",
 ref = "commit_hash")
```

For example, if you want to install the package we have developed together from my Github account, you could run the following (the commit hash is actually wrong so you don't install this one by mistake):

```
remotes::install_github("rap4all/housing@fusen",
 ref = "ae42601")
```

So the package in the `fusen` branch and at commit “ae42601” gets installed. Keep in mind that you can specify the commit hash to install the exact version you need, because this is going to do wonders for reproducibility.

### 11.5.2 Code cannot be hosted

If the code cannot be hosted, then you have to share it *manually*. That's less than ideal, but sometimes there simply is no alternative. In that case, you need to prepare a compressed archive that you can share. This is easily done using `devtools::build()`. Start a new session in the root directory of your package, and run `devtools::build()`. This will create a `.tar.gz` file that you can send to your team-mates, or archive for future you. Ideally, before creating this file, you should go to `0-dev_history.Rmd` and update the version number in the `fusen::fill_description()` function, like so:

```
fusen::fill_description(
 pkg = here::here(),
 fields = list(
 version = "0.0.0.9000"
)
)
```

```

Title = "Housing Data For Luxembourg",
Version = "0.1", # notice that I've added a version number here
Description = "This package contains functions to get,
 clean and analyse housing price data for Luxembourg.",
`Authors@R` = c(
 person("Bruno", "Rodrigues", email = "bruno@brodrigues.co",
 role = c("aut", "cre"),
 comment = c(ORCID = "0000-0002-3211-3689"))
)
, overwrite = TRUE) # you need to add overwrite = TRUE to overwrite the file

```

You have to be very disciplined here, because you have to make sure that you keep updating this and documenting which version of the package should get used for which project. Also, make sure that you can store generated `.tar.gz` alongside the project and that you provide clear installation instructions. To install a package from a `.tar.gz` file, open a new R session and run the following:

```
remotes::install_local("path/to/package/housing_0.0.0.9000.tar.gz")
```

### 11.5.3 Marketing your work

Once your package is done, whether it is destined for CRAN or not, whether it can only be shared within your organisation or not, it is important to market it and make it discoverable. This is where building a website for the package is important, and thankfully, it but takes two lines of code to build a fully functioning site. In the introduction we built the website for the template included with `{fusen}`, let's now build a website for our housing package.

This website can then be hosted online if you wish, or it can be shared internally to your organisation, offline, as a means of providing documentation.

Take a look at the very last section of the `0-dev_history.Rmd` file, titled “Share the package”. If you execute the lines in that chunk (ideally from a fresh R session), a website will be built automatically. This website will first be in the `docs/` folder: open the `index.html` file using a web-browser and you can start navigating the documentation!

If your package is on Github, you can also host the website for free on Github pages. For this, first make sure that the `.gitignore` file in the root of your package does not contain the `docs/` folder. If yes remove it. Then, commit and push. This will upload the `docs/` package on Github. Then, go to the repository's settings, and “Pages” and then choose the branch that contains the `docs/` folder:

Choose these options to host your package's website for free!

As an example, you can visit the website of the package we've built together [here](https://rap4all.github.io/housing/)<sup>11</sup>.

The package's `README` will be shown, if available, on the starting page of the website. So if you want to add a `README` to your package, go to the `0-dev_history.Rmd` file and execute the line `usethis::use_readme_rmd()`, which adds a template `README` file in the root of your package. Regardless of whether you want to build a website, adding a `README` to it is always a good idea! You could explain what the main features of the package are, and how to install it, especially if you want your users or future you to install the package at a certain commit, it is quite useful to write it down clearly in the instructions. Something like:

```
To install this package, run the following lines of code:
```

```
...

remotes::install_github("rap4all/housing@fusen",
 ref = "ae42601")
...
```

## 11.6 Conclusion

Turning our analysis into a package is useful, because we can divert a lot of tools that are originally intended for package development towards improving our analysis. We can now more easily document the code, define its dependencies, and also share it with teammates, our future selves or the world. What's more, we clearly separate two tasks from each other: the pure software engineering part, which consisted in building the package, from the pure data analysis part, which will eventually become our pipeline.

There is one chapter left before we actually build a full-fledged pipeline. In the next chapter, we will learn how to use unit and assertive testing to further improve the code of our package, which will thus also improve the quality of our analysis.

---

<sup>11</sup><https://rap4all.github.io/housing/>

## 12 Testing your code

Testing code is crucial, and we all do it in some form or another. The problem is that it is not something that we do consistently. Usually code gets tested in the beginning of a project, but then, as we start focusing on the analysis more and more and need to respect deadlines, testing gets forgotten.

In this chapter, you are going to learn how to make testing your code consistent and, very importantly, fully automatic. Just like in the previous chapter, the key is to *write everything down*. Don't just do a little test in the console to see if the function you've just written works as expected. Write it down! And don't rely on future you to run tests, because future you is just as unreliable as you are. Tests need to be run each time *any* of the code from a project gets changed. This might seem overkill (why test a function that you didn't even touch for weeks?), but because there are dependencies between your functions, a change in one function can affect another. Especially if the output of function A is the input of function B, and now, because of your changes, the output of function A changed in a way that it breaks function B, or also modifies its output in an unexpected way.

There are several types of tests that we can use:

- unit testing: these are written while developing, and executed while developing;
- assertive testing: these are executed at runtime. These make sure, for example, that the inputs a function receives are sane.

Let's start with unit testing.

### 12.1 Unit testing

Unit testing is the testing of units. What's a unit? Functions are units! We actually already encountered one unit test before, in the `save_data.Rmd` script:

```
```{r tests-clean_flat_data}  
# We now need to check if we have them all in the data.  
# The test needs to be self-contained, hence  
# why we need to redefine the required variables:
```

```

former_communes <- get_former_communes()

current_communes <- get_current_communes()

communes <- get_test_communes(
  former_communes,
  current_communes
)

raw_data <- get_raw_data(
  url = "https://github.com/b-rodrigues/rap4all/raw/master/datasets/vente-maison-2010-2021"
)

flat_data <- clean_raw_data(raw_data)

testthat::expect_true(
  all(communes %in% unique(flat_data$locality))
)
...

```

When using `{fusen}`, a unit test should be a self-contained chunk that can be executed completely independently. This is why in this chunk we re-created the different variables that we needed, `communes` and `flat_data`. If you were developing the package without `{fusen}`, you would do the same, so don't think that this is somehow a limitation of `{fusen}`.

The test above ensures that we find all the former and current communes of Luxembourg in our dataset. Let me explain again why we want to write such a test down in a script and not simply try it out in our console.

For this test to pass, a lot of moving pieces have to fall together. If anything changes, be it because you changed something in either `get_raw_data()` or `clean_raw_data()` or because something changed with the Wikipedia tables you scraped, this test will not pass. And you should be made aware of failures as soon as possible! Also, this test ensures that when the data gets updated, you are certain that if you use the code in `save_data.Rmd` you will get a new dataset that is likely correct, even if new communes merge. And mergers will happen around 2024 by the way, the communes of Groussbous and Wal will merge, and the communes of Bous and Waldbredimus as well. So you need to make sure that when this happens, your code knows how to handle this, or at least gives out an error as early as possible.

Ideally, we would test every function that we wrote, but sometimes that's not really possible, either due to lack of time, or because the function is quite trivial, so maybe no test is warranted. But be careful what you consider trivial though, I have personally been bitten in the past by trivially simple functions! For example, a function like this one:

```

```{r function-make_commune_level_data}
#' make_commune_level_data Makes the final data at commune level
#'
#' @param flat_data Flat data df as returned by clean_flat_data()
#' @importFrom dplyr filter
#' @return A data frame
#' @export
make_commune_level_data <- function(flat_data){
 flat_data |>
 filter(!grepl("nationale|offres", locality),
 !is.na(locality))
}

```

```

might not need to be unit-tested. An assertion, which we will learn about in the next section, is likely better suited. However, as functions become more complex, unit tests are highly recommended. This is because it can become very difficult to make sure that changing some part of the function somewhere does not affect some other part. This is where writing several unit tests can be useful. As long as all unit tests keep succeeding (or passing) you are somewhat sure that what you're doing is not breaking stuff. And unit tests are especially useful when collaborating using trunk-based development! As the project leader, you could for example refuse to merge changes that break unit tests (the first project I worked on that used unit tests was organized like this, so we all better made sure that we wrote sensible code).

Before continuing, let's rewrite the test we have already. While it is fully working, I didn't really write it in the canonical form. Inside `dev/save_data.Rmd`, change the code of the test to the following:

```

```{r tests-clean_flat_data}
We now need to check if we have them all in the data.
The test needs to be self-contained, hence
why we need to redefine the required variables:

former_communes <- get_former_communes()

current_communes <- get_current_communes()

communes <- get_test_communes(
 former_communes,
 current_communes
)

```



```

raw_data <- get_raw_data(
 url = "https://github.com/b-rodrigues/rap4all/raw/master/datasets/vente-maison-2010-2021"
)

flat_data <- clean_raw_data(raw_data)

test_that("Check if all communes are accounted for", {

 expect_true(
 all(communes %in% unique(flat_data$locality))
)

})
```

```

The only difference is that instead of calling `testthat::expect_true()` directly, we have wrapped this call inside `testthat::test_that()`. This way, we can add a description to the test. This is useful if the test fails.

Save `dev/save_data.Rmd` and go back to `0-dev_history.Rmd` to inflate `save_data.Rmd` again. Everything should work without problems.

If the test fails, you get an informative message. To illustrate, I've added a typo in the test and inflated `save_data.Rmd`. Because tests always run when files get inflated, this test failed and here is what happened:

```

Failed tests
Error ('test-get_raw_data.R:18'): Check if all communes are accounted for
Error in `communs %in% unique(flat_data$locality)`: object 'communs' not found
Backtrace:
  1. testthat::expect_true(all(communs %in% unique(flat_data$locality))) at test-get_raw_data.R:18:18
  2. testthat::quasi_label(enquo(object), label, arg = "object")
  3. rlang::eval_bare(expr, quo_get_env(quo))
  4. communs %in% unique(flat_data$locality)

[ FAIL 1 | WARN 2 | SKIP 0 | PASS 0 ]
Error: Test failures
Execution halted

```

The file `test-get_raw_data.R` contains our test, generated by inflating `save_data.Rmd`. You can find it under the `tests/testthat/` folder of your inflated package. You can also see

the description that we've added, which helps us find the test that failed. In cases like this, you should go back to the function that makes the test fail and correct it, until the test passes. You should also make sure that everything is alright with the test itself. If there really is a typo in the test, you should of course correct the test (in `dev/save_data.Rmd`, not in `tests/testthat/`)!

Now, let's add a unit test to another function, `get_laspeyeres()`. This function seems to me like a good candidate for testing, as it is not that trivial.

Let's try with something simple. `get_laspeyeres()` expects either `commune_level_data` or `country_level_data`. What happens if we provide another dataset? Very likely an error. So let's test for this. Go back to the `save_data.Rmd` file and add the following, under the function definition of `get_laspeyeres()`:

```
```{r tests-get_laspeyeres}
test_that("Wrong data", {

 expect_error(
 get_laspeyeres(mtcars)
)

})
```
```

Since we expect an error, we used `expect_error()`, which succeeds if the code fails! If you're confused, no worries, we've all been there. But let's think about it: what would you want to happen if you provided a wrong data set? Surely, you'd like for the function to scream an error at you, and not somehow do something and return *something*. So testing that functions fail when they should is actually quite important as well. Let's add another, similar, test:

```
```{r tests-get_laspeyeres}

test_that("Wrong data", {

 expect_error(
 get_laspeyeres(mtcars)
)

})

test_that("Empty data", {

 expect_error(
```

```

 get_laspeyeres(subset(mtcars, am == 2))
)

})

...

```

This second test checks what happens if we provide an empty dataset. This should not happen, but hey, it's always a good idea to see what could happen. Here we also expect an error, so we use `expect_error()` as well. Inflating `save_data.Rmd` runs the tests again, all of them successfully.

Now, I know what you're thinking. Probably something along the lines of *"Bruno, you told me that making my projects reproducible and reliable and robust would not take much more time than what I was already doing before. This certainly doesn't feel like it!"* to wish I answer that your feelings on the issue are wrong. It may not feel like it, but doing this does two things:

- It ultimately saves you time. You typed the test once, and can now rerun it automatically every time you inflate the `.Rmd` files. You don't need to remember to test the code, and don't need to remember how to test the code.
- This saves you a lot of headaches. You don't have to live in fear that you might forget to test the code, or forget how to test the code. You wrote the tests down, and now you're free to concentrate on adding features or using the existing code knowing that you can trust its outputs.

Trust the process.

Let's go back to the two tests from before: `get_laspeyeres()` fails, as expected, when we provide a random dataset to it. But it would be interesting to know why it fails. Simply run `get_laspeyeres(mtcars)` in the console. This is what we get back:

```

Error in `mutate()` :
! Problem while computing `p0 = ifelse(year == "2010",
 average_price_nominal_euros, NA)`.
Caused by error in `ifelse()` :
! object 'year' not found
Run `rlang::last_error()` to see where the error occurred.

```

So the functions fails but for the wrong reason. It fails because the column `year` cannot be found in the data. But what if there was a column `year`? The code would continue, but then likely fail for something else. It would be much safer to make it fail as soon as it detects that the provided data sets are not one of `commune_level_data` and `country_level_data`. But for this, we need assertive programming, which we will discuss in the next section. Remember, unit testing tests should run during development time, and assertive testing is for run-time.

In the next section we will be changing the function to fail when the right datasets are not provided, but our unit test will not need to change; the function still fails, but this time it'll be for the right reasons.

This is another advantage of writing unit tests: it forces you to think about what you're doing. It very often improves your code quite a lot, and not just from a pure algorithmic perspective, but also from a user experience perspective. Writing these tests made us think about the failure of our function when we provide a random dataset, and made us realise that it would be much better for users if the returned error message is something such as "Wrong dataset, please provide either `commune_level_data` or `country_level_data`".

Let's continue with testing `get_laspeyeres()`. It would be nice to see if the function actually does what it's supposed to do correctly. For this, we need to start from an input, and then create the expected output. It doesn't matter how you create this output, what matters is that you make absolutely sure that it is correct, and then, never touch it ever again. Let's call this output the "truth". Then, you provide `get_laspeyeres()` with this input and save the output that `get_laspeyeres()` generates. You then compare the "truth" to this output. If everything matches, congratulations, your function produces the right output.

So let's start. Remember that unit tests should be self-contained, so I'm going to create the input dataset and the expected data set (what I called the "truth") in the test itself. This is the code I'm going to use to create the mock, input dataset:

```
input_df <- expand.grid(
 list("year" = c(2010, 2011),
 "locality" = c("Bascharage", "Luxembourg"))
)

input_df$n_offers <- c(123, 101, 1230, 1010)
input_df$average_price_nominal_euros <- c(234, 345, 560, 670)
input_df$average_price_m2_nominal_euros <- c(23, 34, 56, 67)
```

This creates a data frame with two years, two communes and some mock prices. Now, I need to create the output. I start from the input, and add the columns that `get_laspeyeres()` computes "by hand". Remember, you need to make sure that these results are correct!

```
expected_df <- input_df

p0 should be always equal to the value in the first year
expected_df$p0 <- c(234, 234, 560, 560)
expected_df$p0_m2 <- c(23, 23, 56, 56)

p1 should be equal to the price divided by p0
```

```

expected_df$pl <- expected_df$average_price_nominal_euros/expected_df$p0 * 100
expected_df$pl_m2 <- expected_df$average_price_m2_nominal_euros/expected_df$p0_m2 * 100

```

If you look at each line, you see that this is basically what `get_laspeyeres()` does. We can inspect the results, maybe even calculate each cell using a pocket calculator. It doesn't matter, what's important is that the `expected_df` is correct and saved. This is what the full test looks like:

```

```{r, eval = F}
test_that("get_laspeyeres() produces correct results", {

  input_df <- expand.grid(
    list("year" = c(2010, 2011),
        "locality" = c("Bascharage", "Luxembourg"))
  )

  input_df$n_offers <- c(123, 101, 1230, 1010)
  input_df$average_price_nominal_euros <- c(234, 345, 560, 670)
  input_df$average_price_m2_nominal_euros <- c(23, 34, 56, 67)

  expected_df <- input_df

  # p0 should be always equal to the value in the first year
  expected_df$p0 <- c(234, 234, 560, 560)
  expected_df$p0_m2 <- c(23, 23, 56, 56)

  # pl should be equal to the price divided by p0
  expected_df$pl <- expected_df$average_price_nominal_euros/expected_df$p0 * 100
  expected_df$pl_m2 <- expected_df$average_price_m2_nominal_euros/expected_df$p0_m2 * 100

  expect_equivalent(
    expected_df, get_laspeyeres(input_df)
  )

})
```

```

Notice that I've used `expect_equivalent()` and not `expect_equal()`. This is because `expected_df` is of class `data.frame`, while `get_laspeyeres()` outputs a tibble. So if you use `expect_equal()` the test would not pass. Sometimes, this level of strictness is required, but not always, as is the case here.

Once again, inflate `save_data.Rmd`. This will run the tests, and if everything went well,

you should end up, again, with a functioning package. I highly advise that you consult `{testthat}`'s documentation to learn about all the other functions that you can use for writing unit tests.

Let's now go to assertive programming.

## 12.2 Assertive programming

Remember in chapter 6, where I discussed safe functions? As a refresher, here's the `nchar()` function, providing a correct output when the input is a character:

```
nchar("100000000")
```

```
[1] 9
```

and here is `nchar()` providing a *surprising* result when the input is a number:

```
nchar(100000000)
```

```
[1] 5
```

This is because 100000000 gets converted to `1e+08` and then this gets converted into the string `"1e+08"` which is 5 characters long. So in that section, I suggested to define your own `nchar2()` that makes sure that the provided input is a character:

```
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 now returns an error if the input is a number, instead of doing all these silent conversions. The technique we have used here is what we call assertive programming. `stop()` and `stopifnot()` are functions included with R that can be used for assertive programming. Here is an example using `stopifnot()`:

```
nchar3 <- function(x, result = 0){

 stopifnot("Input x must be a character" =
 isTRUE(is.character(x)))

 if(x == ""){
 result
 } else {
 result <- result + 1
 split_x <- strsplit(x, split = "")[[1]]
 nchar3(paste0(split_x[-1],
 collapse = ""), result)
 }
}
```

If we go back to `get_laspeyeres()`, we should be using assertive programming to make sure that the provided datasets are one of `commune_level_data` and `country_level_data`. This is how we could rewrite the function:

```
get_laspeyeres <- function(dataset){

 which_dataset <- deparse(substitute(dataset))

 stopifnot("dataset must be one of `commune_level_data` or `country_level_data`" =
 (which_dataset %in% c("commune_level_data", "country_level_data")))

 group_var <- if(grepl("commune", which_dataset)){
 quo(locality)
 } else {
 NULL
 }

 dataset |>
 group_by(!!group_var) |>
 mutate(p0 = ifelse(year == "2010", average_price_nominal_euros, NA)) |>
 fill(p0, .direction = "down") |>
 mutate(p0_m2 = ifelse(year == "2010", average_price_m2_nominal_euros, NA)) |>
 fill(p0_m2, .direction = "down") |>
```

```

 ungroup() |>
 mutate(pl = average_price_nominal_euros/p0*100,
 pl_m2 = average_price_m2_nominal_euros/p0_m2*100)
 }

```

We can now also edit the unit test from before, the one where we provide the wrong data. This unit test would fail, as expected, but for the wrong reason. We now want to make sure that it fails for the right reason, so for this we change the unit tests like this:

```

test_that("Wrong data", {

 expect_error(
 get_laspeyeres(mtcars),
 regexp = "dataset must be one of"
)

})

```

I use the `regexp` argument of `expect_error` to enter a regular expression that matches the error message. So the string “dataset must be one of” will match the message returned by the error, and if they match (remember, the provided string is a regular expression), then I know I get the *correct* error. Here is what happens if I use the wrong message as the `regex` argument:

```

Failed tests
Failure ('test-get_laspeyeres.R:6'): Wrong data
`get_laspeyeres(mtcars)` threw an error with unexpected message.
Expected match: "message is wrong"
Actual message: "dataset must be one of `commune_level_data` or `country_level_data`"

```

So now, not only does our function fail for the right reasons, our test is able to tell us that as well!

Before inflating to run these tests, you should also change the test titled “get\_laspeyeres() provides correct answers”. This is because the name of the input dataset used for the test is `input_df`. So if you leave it like this, the assertion that we’ve included in the function will make this test fail. So change this test by simply saving `input_df` `commune_level_data`:

```

commune_level_data <- input_df

expect_equivalent(

```



```
 expected_df, get_laspeyeres(commune_level_data)
)
```

if you forget to do this, don't worry, the unit test would fail to remind you!

Go back to `0-dev_history.Rmd` and inflate the file again to update it. The unit test we wrote before should keep passing, but now, it passes for the right reason: we provided a dataset that is neither `commune_level_data` nor `country_level_data` and not because there is no column `year` in the dataset!

You can also make sure that the provided input is of the right class:

```
any_function <- function(dataset){

 stopifnot("`dataset` must be a data frame" =
 inherits(dataset, "data.frame"))

 print("No problem")
}
```

This will succeed:

```
any_function(mtcars)
```

```
[1] "No problem"
```

But this will fail:

```
any_function("this is not a data frame")
```

```
Error in any_function("this is not a data frame") :
 `dataset` must be a data frame
```

`inherits()` checks if an option inherits from a certain class. So for example, a `tibble` or a `data.table` that are classes that are defined by inheriting attributes from the `data.frame` class, will also successfully pass the test above. You can be as strict as you need: for example, do you need any type of number? You could do the following:

```
inherits(2, "numeric")
```

```
[1] TRUE
```

But do you actually need integers, and want to force this? Then you could be stricter in your assertion:

```
inherits(2, "integer")
```

```
[1] FALSE
```

If you want the above to evaluate to TRUE, an integer must be provided:

```
inherits(2L, "integer")
```

```
[1] TRUE
```

Do you want, for some reason, that your functions only accept `tibbles` and not `data.frames`? Be as strict as you need:

```
inherits(tibble::as_tibble(mtcars), "tbl_df")
```

```
[1] TRUE
```

```
inherits(mtcars, "tbl_df")
```

```
[1] FALSE
```

You could also use more complex assertions. For example, suppose that you need to clean data using many functions, with several filters. Something could go wrong in any of these functions for a variety of reasons. So each of these functions could test if all the individuals are still in the data, and that you didn't remove any of them by mistake. A test like:

```
summary_stats <- function(dataframe, var){
 stopifnot("Some individuals are missing!" =
 all(unique(dataframe[[var]]) %in% c(0,1)))

 # and then some computations here
}
```

Now, when running `summary_stats(mtcars, "am")`, if somehow the level “1” or “0” is missing from `mtcars`, the function would throw an error.

There are several packages for assertive programming that you might want to check out:

- `{assertthat}`
- `{chk}`
- `{checkmate}`

I won't discuss any of them; what's important is for you to know that assertive programming is something that is useful, and that you should add to your toolbox.

## 12.3 Test-driven development

Test-driven development, or TDD, is the programming paradigm in which instead of writing a function and then several tests to ensure that it's working as expected, you start with several tests. Of course, since there is no function to test, these tests will all fail at first. But the goal is to then write a function such that the tests pass.

TDD is interesting in at least two scenarios:

- You want to write a function, but don't know exactly where to start. Maybe it's a very complex function. So writing tests can help you think about it, and already fix certain properties that this function should have.
- You use the tests as a way to write requirements for a codebase. This can be useful when working in a team, and you don't want to "waste" time writing requirements, so instead you already write the tests. Careful though, because a "smart" programmer could write code that passes the tests but doesn't actually do anything otherwise useful.

I tend to use TDD when I need to write a function but don't quite know where to start. I start by writing the most basic tests and make them ever more complicated. At some point, I start having an idea for the function's implementation and have a go at it. Sometimes this makes me even realise that I was testing for something irrelevant!

Some programmers only do TDD; so they start by writing many, many tests, and then only start writing their functions. Personally, I think that this is also not ideal, because you could waste a lot of time writing meaningless tests.

## 12.4 Code coverage

It is useful to have an idea of which functions are tested and which are not, but also *how much* of a function is being tested. For example, suppose that you have an `if...else...` clause somewhere in a function. Did you write a test for each of the outcomes of this clause? Maybe you only wrote a test when this clause evaluates to `TRUE`, but forgot to write a test for the case it is `FALSE`.

The packages `{covr}` allows you to track the test coverage of your package. Install `{covr}` and run `report()` in the console to get the results:

```
covr::report()
```

This should open a tab in your web browser with some statistics. You can click on the individual scripts to see the source code of your functions: each line that is highlighted in green represents a line that is being test, and lines in red are lines that are not being tested:

The output of `report()` inside a web browser.

You could strive to get 100% coverage by painting all the lines green (by writing unit tests that test these lines). But in practice, it is not always so easy to get 100% coverage, so don't fret if don't achieve perfection.

If you're working on a server (and thus do not have access to a graphical user interface) you can instead opt for the `covr::package_coverage()` function which provides you with the following results (inside your console):

```
housing Coverage: 73.33%
R/get_laspeyeres.R: 57.14%
R/get_raw_data.R: 80.65%
```

The percentage represents the share of lines of code that are tested by our unit tests. We see that the share of lines being tested in `get_laspeyeres().R` is 57%: this is because the script `get_laspeyeres()` contains two functions, `get_laspeyeres()` and `make_plot()`. We do not test `make_plot()` at all, hence why the percentage is so low. We could move `make_plot()` to another script by simply putting the function under a level two header in the original `.Rmd` file and then inflating again. But in any case, this would not improve the overall coverage of the package; we would ideally need to write a test for `make_plot()`. This is left as an exercise to the reader.

## 12.5 Conclusion

Testing is crucial and useful. Not just because it gives you peace of mind but also because writing tests forces you to think about your code, by putting yourself in the shoes of your users (which includes future you as well). In most cases, it is even something that you've been doing but perhaps not as systematically as you should.

There really is no other way to say this: you need to consider writing tests as an integral part of the project, and need to take the required time it takes to write them into account when planning projects. But keep in mind that writing them makes you gain a lot of time in the long run, so actually, you might even be faster by writing tests! Tests also allow you to

immediately see where something went wrong, when something goes wrong. So tests save you time here as well. Without tests, when something goes wrong, you have a hard time finding where the bug comes from, and end up wasting precious time. And worse, something things go wrong and break, but silently. You still get an output that may look ok at first glance, and only realise something is wrong way too late. Testing helps avoiding such situations.

So remember: it might *feel* like packaging your code and writing tests for it takes time, but:

- you're actually already doing it, albeit casually;
- it saves you time in the long run.

The tools I've showed you in this chapter and in the last are probably the fastest, easiest way to go from your analysis to a documented and tested package in a matter of hours. The benefits these provide however are measured in days of work.

# 13 Build automation

We are finally ready to actually build a pipeline. For this, we are going to be using a package called `{targets}` which is a so-called build automation tool.

If you go back to the reproducibility iceberg, you will see that we are quite low now.

Without a build automation tool, a pipeline is nothing but a series of scripts that get called one after the other, or perhaps the pipeline is only one very long script that does the required operations successfully.

There are several problems with this approach, so let's see how build automation can help us.

## 13.1 Introduction

Script-based workflows are problematic for several reasons. The first is that scripts can, and will, be executed out of order. You can mitigate the problems this can make by using pure functions, but you still need to make sure to not run the scripts out of order. But what does that actually mean? Well, suppose that you changed a function, and only want to re-execute the parts of the pipeline that are impacted by that change. But this supposes that you can know, in your head, which part of the script was impacted and which was not. And this can be quite difficult to figure out, especially when the pipeline is huge. So you will run certain parts of the script, and not others, in the hope that you don't need to re-run everything.

Another issue, but this one is perhaps more subjective, is that pipelines written as scripts are usually quite difficult to read and understand. To mitigate this, what you'd typically do is write a lot of comments. But here again you face the problem of needing to maintain these comments, and once the comments and the code are out of synch... the problems start (or rather, they continue).

Running the different parts of the pipeline in parallel is also very complicated if your pipeline is defined as script. You would need to break the script into independent parts (and make really sure that these parts are independent) and execute them in parallel, perhaps using a separate R session for each script. The good news is that if you followed the advice from this book you have been using functional programming and so your pipeline is a series of function calls, which is "easy" to break up and run in parallel.

But by now you should know that software engineers also faced similar problems when they needed to build their software, and you should also suspect that they likely came up with something to alleviate these issues. Enter build automation tools.

When using a build automation tool, what you end up doing is writing down a recipe, that will not be very different than a standard script in your favourite programming language, that defines how the source code should be “cooked” into the software (or in our case, a report, a cleaned dataset or any data product).

The build automation tool can then figure out the following things:

- any change in any of the code. Only the outputs that are affected by the changes you did will be re-computed (and their dependencies as well);
- any change in any of the tracked files. For example, if a file gets updated daily, you can track this file and the build automation tool will only execute the parts of the pipeline affected by this update;
- which parts of the pipeline can safely run in parallel (with the option to thus run the pipeline on multiple CPU cores).

Just like many of the other tools that we have encountered in this book, what build automation tools do is allow you to not have to rely on your brain. You *write down* the recipe once, and then you can focus again on just the code of your actual project. You shouldn’t have to think about the pipeline itself, nor think about how to best run it. Let your computer figure that out for you, it’s much better at such tasks than you.

## 13.2 {targets} quick-start

First thing’s first: to know everything about the {targets} package, you should read the excellent [{targets} manual](#)<sup>1</sup>. Everything’s in there. So what I’m going to do is really just give you a very quick intro to what I think are really the main points you should know about to get started.

Let’s start by a “hello-world” type pipeline. Create a new folder called something like `targets_intro/`, and start a fresh R session in it. For now, let’s ignore {renv}. We will see how {renv} works together with {targets} to provide a reproducible pipeline later. In that fresh session inside the `targets_intro/` run the following line:

```
targets::tar_script()
```

this will create a template `_targets.R` file in that directory. This is the file in which we will define our pipeline. Open in it in your favourite editor. A ‘`_targets.R`’ pipeline is roughly divided into three parts:

---

<sup>1</sup><https://is.gd/VS6vSs>

- first is where packages are loaded and helper functions defined;
- second is where pipeline-specific options are defined;
- third is the pipeline itself, defined as a series of *targets*.

Let's go through all these parts one by one.

### 13.2.1 `_targets.R`'s anatomy

The first part of the pipeline is where packages get loaded as well as helper functions. In the template, the very first line is a `library(targets)` call followed by a function definition. There are two important things here that you need to understand.

If your pipeline needs, say, the `{dplyr}` package to run, you could write `library(dplyr)` right after the `library(targets)` call. However, it is best to actually do as in the template, and load the packages using `tar_option_set(packages = "dplyr")`. This is because if you execute the pipeline in a cluster, you need to make sure that all the packages are available to all the workers. If you load the packages at the top of the `_targets.R` script, the packages will be available for the session that called `library(...)`, but not any worker sessions spawned for parallel execution.

So, the idea is that at the very top of your script, you only load the `{targets}` library and other packages that are required for running the pipeline itself (as we shall see in coming sections). But packages that are required by functions that are running inside the pipeline should ideally be loaded as in the template. Another way of saying this: at the top of the script, think “pipeline infrastructure” packages (`{targets}` and some others), but inside `tar_option_set()` think “packages for functions that run inside the pipeline” packages.

Part two is where you set some global options for the pipeline. As discussed previously, this is where you should load packages that are required by the functions that are called inside the pipeline. I won't list all the options here, because I would simply be repeating what's in the [documentation](https://is.gd/lm4QoO)<sup>2</sup>. This second part is also where you can define some functions that you might need for running the pipeline. For example, you might need to define a function to load and clean some data: this is where you would do so. We have developed a package, so we might not need this. But sometimes your analysis doesn't require you to write any custom functions, or maybe just a few, and perhaps you don't see the benefit of building a package just for one or two functions. So instead, you have two other options: you either define them directly inside the `_targets.R` script, like in the template, or you create a `functions/` folder next to the `_targets.R` script, and put your functions there. It's up to you, but I prefer this second option.

Finally, comes the pipeline itself. Let's take a closer look at it:

---

<sup>2</sup><https://is.gd/lm4QoO>



```
list(
 tar_target(
 data,
 data.frame(x = sample.int(100),
 y = sample.int(100))
),
 tar_target(
 summary,
 summ(data)) # Call your custom functions as needed.
)
```

The pipeline is nothing but a list (told you lists were a very important object) of *targets*. A target is defined using the `targets::tar_target()` function and has at least two inputs: the first is the name of the target (without quotes) and the second is the function that generates the target. So a target defined as `tar_target(y, f(x))` can be understood as `y <- f(x)`. The next target can use the output of the previous target as an input, so you could have something like `tar_target(z, f(y))` (just like in the template).

### 13.3 A pipeline is a composition of pure functions

This pipeline can immediately be run using the `targets::tar_make()` command in a console. Doing so shows you the following output:

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

- start target data
- built target data [0.82 seconds]
- start target summary
- built target summary [0.02 seconds]
- end pipeline [1.71 seconds]

The pipeline is done running! So, now what? This pipeline simply built some summary statistics, but where are they? Typing `summary` in the console to try to inspect this output results in the following:

```
summary
```

```
function (object, ...)
```

```
UseMethod("summary")
<bytecode: 0x000001f1a5436d78>
<environment: namespace:base>
```

What is going on?

First, you need to remember our chapter on functional programming. We want our pipeline to be a sequence of pure functions. This means that our pipeline running successfully should not depend on anything in the global environment (apart from loading the packages in the first part of the script, and the options set with `tar_option_set()` for the others) and it should not change anything outside of its scope. This means that the pipeline should not change anything in the global environment either. This is exactly how a `{targets}` pipeline operates. A pipeline defined using `{targets}` will be pure and so the output of the pipeline will not be saved in the global environment. Now, strictly speaking, the pipeline is not exactly pure. Check the folder that contains the `_targets.R` script. There should now be a `_targets/` folder in there as well. If you go inside that folder, and then open the `objects/` folder, you should see two objects, `data` and `summary`. These are the outputs of our pipeline.

So each target that is defined inside the pipeline gets saved there in the `.rds` format. This is an R-specific format that can be used to save *any* type of object. It doesn't matter what it is: a simple data frame, a fitted model, a ggplot, whatever, any object can be saved into this format using the `saveRDS()` function, and can be read back into another R session using `readRDS()`. `{targets}` makes use of these two functions to save every target. Keep this in mind if you use Git to version the code of your pipeline (which you are doing of course), and add the `targets/` folder to the `.gitignore` (unless you really want to also version it, but it shouldn't be necessary).

Ok, so back to `summary`. What happened before when we typed `summary` in the console? Well, now you know why the result didn't appear: this is because the target called `summary` gets saved in that folder, but not in the global environment. So why did we see something anyways, which was not looking like a data summary at all?

This is because R has a `summary()` function. So when you write the function's name in the console without `()` you see the function's source code. Let's try again:

```
summary

function (object, ...)
 UseMethod("summary")
<bytecode: 0x000001f1a5436d78>
<environment: namespace:base>
```

Ok, so here, strictly speaking, you don't see the source code from `summary()`. This is because `summary()` is a generic function, that simply picks the right `summary()` depending on what

type of object you pass to it. Try with `summary.data.frame()` and you should see the source code. So when you type `summary` after running the pipeline, because the pipeline is pure and nothing gets saved to the global environment, you see `summary`'s source code and not the target that got computed by the pipeline.

So that was a long explanation, but I think that it was worth the effort. It is quite important to not forget that a targets pipeline is the composition of many pure functions, but since we need to be able to access the outputs produced by the pipeline, they also get saved inside a folder. To retrieve the outputs you can use `targets::tar_read()` or `targets::tar_load()`. The difference is that `tar_read()` simply reads the output and shows it in the console (just like when you type `mtcars` in a console for example) but `tar_load()` reads and saves the object into the global environment (just like doing something like `x <- mtcars` in an R console). So to retrieve our `summary` object let's use `tar_load(summary)`:

```
tar_load(summary)
```

Now, typing `summary` shows the computed output (now the function `base::summary()` is masked):

```
summary

 mean_x
1 500000.5
```

It is possible to load all the outputs using `targets::tar_load_everything()` so that you don't need to load each output one by one.

Before continuing with more `{targets}` features, I want to really stress the fact that the pipeline is the composition of pure functions. So functions that only have a side-effect will be difficult to handle. For example, plotting in base R consists in a series of calls to functions with side-effects. If you open an R console and type `plot(mtcars)`, you will see a plot. But the function `plot()` does not create any output. It just prints a picture on your screen, which is a side-effect. To convince yourself that `plot()` does not create any output and only has a side-effect, try to save the output of `plot()` in a variable:

```
a <- plot(mtcars)
```

doing this will show the plot, but then if you call `a`, the plot will not appear, and instead you will get `NULL`:

```
a

NULL
```

This is also why saving plots in R is awkward, it's because there's no object to actually save!

So because `plot()` is not a pure function, if you try to use it in a `{targets}` pipeline, you will get `NULL` as well when loading the target. To see this, change the list of targets like this:

```
list(
 tar_target(
 data,
 data.frame(x = sample.int(100),
 y = sample.int(100))
),

 tar_target(
 summary,
 summ(data)
),

 tar_target(
 data_plot,
 plot(data)
)
)
```

I've simply added a new target using `tar_target()`. Run the pipeline again using `tar_make()` and then type `tar_load(data_plot)` to load the `data_plot` target. But typing `data_plot` only shows `NULL` and not the plot!

There are several workarounds for this. The first is to use `ggplot()` instead. This is because the output of `ggplot()` is an object of type `ggplot`. You can do something like `a <- ggplot()` and then type `a` to see an empty canvas. Doing `str(a)` also shows an output (the structure of the object `a` of class `ggplot`).

The second workaround is to save the plot to disk. For this, you need to write a new function, for example:

```
save_plot <- function(filename, ...){

 png(filename = filename)
 plot(...)
 dev.off()

}
```

If you put this in the `_targets.R` script, before defining the list of `tar_target` objects, you could use this instead of `plot()` in the last target:

```
summ <- function(dataset) {
 summarize(dataset, mean_x = mean(x))
}

save_plot <- function(filename, ...){
 png(filename = filename)
 plot(...)
 dev.off()

 filename
}

Set target-specific options such as packages.
tar_option_set(packages = "dplyr")

End this file with a list of target objects.
list(
 tar_target(
 data,
 data.frame(x = sample.int(100),
 y = sample.int(100))
),

 tar_target(
 summary,
 summ(data)
),

 tar_target(
 data_plot,
 save_plot(filename = "my_plot.png",
 data),
 format = "file")
)
```

After running this pipeline you should see a file called `my_plot.png` in the folder of your pipeline. If you type `tar_load(data_plot)`, and then `data_plot` you will see that this target returns the `filename` argument of `save_plot()`. This is because a target needs to return something, and in the case of functions that save a file to disk returning the path is recommended. This is because if I then need to use this file in another target, I could do

`tar_target(x, f(data_plot))`. Because the `data_plot` target returns a path, I can write `f()` in such a way that it knows how to handle this path. If instead I write `tar_target(x, f("path/to/my_plot.png"))`, then `{targets}` would have no way of knowing that the target `x` depends on the target `data_plot`. The dependency between these two targets would break. Hence why the first option is preferable.

Finally, you will have noticed that the last target also has the option `format = "file"`. This will be topic of the next section.

## 13.4 Handling files

In this section, we will learn how `{targets}` handles files. First, run the following lines in the folder that contains the `_targets.R` script that we've been using up until now:

```
data(mtcars)

write.csv(mtcars,
 "mtcars.csv",
 row.names = F)
```

This will create the file `"mtcars.csv"` in that folder. We are going to use this in our pipeline.

Write the pipeline like this:

```
list(
 tar_target(
 data_mtcars,
 read.csv("mtcars.csv")
),

 tar_target(
 summary_mtcars,
 summary(data_mtcars)
),

 tar_target(plot_mtcars,
 save_plot(
 filename = "mtcars_plot.png",
 data_mtcars),
 format = "file")
)
```

You can now run the pipeline and will get a plot at the end. The problem however, is that the input file "mtcars.csv" is not being tracked for changes. Try to change the file, for example by running this line in the console:

```
write.csv(head(mtcars), "mtcars.csv", row.names = F)
```

If you try to run the pipeline again, our changes to the data are ignored:

```
skip target data_mtcars
skip target plot_mtcars
skip target summary_mtcars
skip pipeline [0.1 seconds]
```

As you can see, because {targets} is not tracking the changes in the mtcars.csv file, from its point of view nothing changed. And thus the pipeline gets skipped because according to {targets}, it is up-to-date.

Let's change the csv back:

```
write.csv(mtcars, "mtcars.csv", row.names = F)
```

and change the first target such that the file gets tracked. Remember that targets need to be pure functions and return something. So we are going to change the first target to simply return the path to the file, and use the format = "file" option in tar\_target():

```
path_data <- function(path){
 path
}

list(
 tar_target(
 path_data_mtcars,
 path_data("mtcars.csv"),
 format = "file"
),
 tar_target(data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 summary(data_mtcars)
),
)
```

```

tar_target(plot_mtcars,
 save_plot(filename = "mtcars_plot.png",
 data_mtcars),
 format = "file")
)

```

To drive the point home, I use a function called `path_data()` which takes a path as an input and simply returns it. This is totally superfluous, and you could define the target like this instead:

```

tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
)

```

This would have exactly the same effect as using the `path_data()` function.

So now we got a target called `path_data_mtcars` that returns nothing but the path to the data. But because we've used the `format = "file"` option, `{targets}` now knows that this is a file that must be tracked. So any change on this file will be correctly recognized and any target that depends on this input file will be marked as being out-of-date. The other targets are exactly the same.

Run the pipeline now using `targets::tar_make()`. Now, change the input file again:

```

write.csv(head(mtcars),
 "mtcars.csv",
 row.names = F)

```

Now, run the pipeline again using `targets::tar_make()`: this time you should see that `{targets}` correctly identified the change and runs the pipeline again accordingly!

## 13.5 The dependency graph

As you've seen in the previous section (and as I told you in the introduction) `{targets}` keeps track of changes in files, but also in the functions that you use. Any change in any of these files will result in `{targets}` identifying which targets are now out-of-date and which should be re-computed (alongside any other target that depends on them). It is possible to visualise this using `targets::tar_visnetwork()`. This opens an interactive network graph in your web browser that looks like this:

This image opens in your web-browser.



In the image above, each target has been computed, so they are all up-to-date. If we now change the input data as before, here is what you will see instead:

Because the input data got changed, we need to run the pipeline again.

Because all the targets depend on the input data, we need to re-run everything. Let's run the pipeline again to update all the targets using `tar_make()`.

Now let's add another target to our pipeline, one that does not depend on the input data. Then, we will modify the input data again, and call `tar_visnetwork()` as well. Change the pipeline like so:

```
list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
),
 tar_target(
 data_iris,
 data("iris")
),
 tar_target(
 summary_iris,
 summary(data_iris)
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 summary(data_mtcars)
),
 tar_target(
 plot_mtcars,
 save_plot(filename = "mtcars_plot.png",
 data_mtcars),
 format = "file")
)
```

Before running the pipeline, we can call `targets::tar_visnetwork()` again to see the entire workflow:

We clearly see that the pipeline has two completely independent parts.

We can see that there are now two independent parts, as well as two unused functions, `path_data()` and `summ()` which we could remove.

Running the pipeline using `targets::tar_make()` builds everything successfully. Let's add the following target, just before the very last one:

```
tar_target(
 list_summaries,
 list(
 "summary_iris" = summary_iris,
 "summary_mtcars" = summary_mtcars
)
)
```

This target creates a list with the two summaries that we compute. Call `targets::tar_visnetwork()` again:

The two separate workflows end up in one output.

Finally, run the pipeline one last time to compute the final output.

## 13.6 Running the pipeline in parallel

`{targets}` make it easy to run independent parts of our pipeline in parallel. In the example from before, it was quite obvious to know which parts were independent, but when the pipeline grows in complexity, it can be very difficult to see which parts are independent.

Let's now run the example from before in parallel. But first, we need to create a function that takes some time to run. `summary()` is so quick that running both of its calls in parallel is not worth it (and would actually even run slower, I'll explain why at the end). Let's define a new function called `slow_summary()`:

```
slow_summary <- function(...){
 Sys.sleep(30)
 summary(...)
}
```

and replace every call to `summary()` with `slow_summary()` in the pipeline:

```
list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
```

```

 format = "file"
),
 tar_target(
 data_iris,
 data("iris")
),
 tar_target(
 summary_iris,
 slow_summary(data_iris)
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 slow_summary(data_mtcars)
),
 tar_target(
 list_summaries,
 list(
 "summary_iris" = summary_iris,
 "summary_mtcars" = summary_mtcars
)
),
 tar_target(
 plot_mtcars,
 save_plot(filename = "mtcars_plot.png",
 data_mtcars),
 format = "file")
)

```

here's what the pipeline looks like before running:

`slow_summary()` is used instead of `summary()`.

(You will also notice that I've removed the unneeded functions, `path_data()` and `summ()`).

Running this pipeline sequentially will take about a minute. To re-run the pipeline completely from scratch, call `targets::tar_destroy()`. This will make all the targets outdated. Then, run the pipeline from scratch with `targets::tar_make()`:

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

- start target path\_data\_mtcars
- built target path\_data\_mtcars [0.18 seconds]
- start target data\_iris
- built target data\_iris [0 seconds]
- start target data\_mtcars
- built target data\_mtcars [0 seconds]
- start target summary\_iris
- built target summary\_iris [30.26 seconds]
- start target plot\_mtcars
- built target plot\_mtcars [0.16 seconds]
- start target summary\_mtcars
- built target summary\_mtcars [30.29 seconds]
- start target list\_summaries
- built target list\_summaries [0 seconds]
- end pipeline [1.019 minutes]

But because computing `summary_iris` is completely independent of `summary_mtcars`, these two computations could be running at the same time on two separate CPU cores. To do this, we need to first load two additional packages, `{future}` and `{future.callr}` at the top of the script. Then, we also need to call `plan(callr)` before defining our pipeline. Here is what the complete `_targets.R` looks like:

```
library(targets)
library(future)
library(future.callr)
plan(callr)

Sometimes you gotta take your time
slow_summary <- function(...) {
 Sys.sleep(30)
 summary(...)
}

Save plot to disk
save_plot <- function(filename, ...){
 png(filename = filename)
 plot(...)
 dev.off()

 filename
}
```

```

Set target-specific options such as packages.
tar_option_set(packages = "dplyr")

list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
),
 tar_target(
 data_iris,
 data("iris")
),
 tar_target(
 summary_iris,
 slow_summary(data_iris)
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 slow_summary(data_mtcars)
),
 tar_target(
 list_summaries,
 list(
 "summary_iris" = summary_iris,
 "summary_mtcars" = summary_mtcars
)
),
 tar_target(
 plot_mtcars,
 save_plot(
 filename = "mtcars_plot.png",
 data_mtcars,
 format = "file"
)
)
)

```

You can now run this pipeline in parallel using `targets::tar_make_future()` (and sequentially as well, just as usual with `targets::tar_make()`). Don't forget to run

`targets::tar_destroy()` to run your pipeline from 0 again:

```
Set workers = 2 to use 2 cpu cores
targets::tar_make_future(workers = 2)

• start target path_data_mtcars
• start target data_iris
• built target path_data_mtcars [0.2 seconds]
• start target data_mtcars
• built target data_iris [0.22 seconds]
• start target summary_iris
• built target data_mtcars [0.2 seconds]
• start target plot_mtcars
• built target plot_mtcars [0.35 seconds]
• start target summary_mtcars
• built target summary_iris [30.5 seconds]
• built target summary_mtcars [30.52 seconds]
• start target list_summaries
• built target list_summaries [0.21 seconds]
• end pipeline [38.72 seconds]
```

As you can see, this was faster but not quite twice as fast, but faster nonetheless. The reason this isn't exactly twice as fast is because there is some overhead to run code in parallel. New R session have to be spawned by `{targets}` and data needs to be transferred, and packages loaded, in these new sessions. This is why it's only worth parallelizing code that takes some time to run. If you decrease the number of sleep seconds in `slow_summary(...)` (for example to 10), running the code in parallel might even be slower, because of that overhead. But if you have several long-running computations, it's really worth the very small price that you pay for the initial setup.

## 13.7 `{targets}` and RMarkdown (or Quarto)

It is also possible to compile documents using RMarkdown (or Quarto) and `{targets}`. The way this works is by setting up a pipeline that produce the outputs you need in the document. For example, if you're showing a table in the document, create a target in the pipeline that builds the underlying data. Do the same for a plot, or a statistical model that you run on some data. Then, in the `.Rmd` (or `.Qmd`) source file, use `targets::tar_load()` or `targets::tar_read()` to load the different objects you need.

Consider the following `_targets.R` file:

```

library(targets)

tar_option_set(packages = c("dplyr", "ggplot2"))

list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 summary(data_mtcars)
),
 tar_target(
 clean_mtcars,
 mutate(data_mtcars,
 am = as.character(am))
),
 tar_target(
 plot_mtcars,
 {ggplot(clean_mtcars) +
 geom_point(aes(y = mpg,
 x = hp,
 shape = am)))}
)
)

```

This pipeline loads the `.csv` file from before and creates a summary of the data as well as plot. But we don't simply want these objects to be saved as `.rds` files by the pipeline, we want to be able to use them to write a document (either in the `.Rmd` or `.Qmd` format). For this, we need another package, called `{tarchetypes}`. This package comes many functions that allow you to define new targets. The new target factory (or function) that we need is `tarchetypes::tar_render()`. As you can probably guess from the name, this function renders an `.Rmd` file. Write the following lines in an `.Rmd` file and save it next to the pipeline:

```

title: "mtcars is the best data set"
author: "mtcars enjoyer"
date: today

Load the summary

```{r, eval = FALSE}
tar_read(summary_mtcars)
```

```

Here is the `_targets.R` file (I loaded `{tarchetypes}` at the top and added a new target at the bottom):

```

library(targets)
library(tarchetypes)

tar_option_set(packages = c("dplyr", "ggplot2"))

list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 summary(data_mtcars)
),
 tar_target(
 clean_mtcars,
 mutate(data_mtcars,
 am = as.character(am))
),
 tar_target(
 plot_mtcars,
 {ggplot(clean_mtcars) +

```



```

 geom_point(aes(y = mpg,
 x = hp,
 shape = am)))}
),
 tar_render(
 my_doc,
 "my_document.Rmd"
)
)
)

```

Running this pipeline with `targets::tar_make()` will now create an `.html` file that you can open in your web-browser. Even if you want to build another format, I advise you to develop using the `.html` format. This is because you can open the `.html**` file in the web-browser, and keep working on the source. Each time you run the pipeline after you made some changes to the file, you simply need to refresh the web-browser to see your changes. If instead you compile a Word document, you will need to always close the file, and then re-open it to see your changes, which is quite annoying. So if you open the output file, you should be seeing something quite plain looking:

The output of our pipeline. Our first data product!

Don't worry, we will make it look nice, but right at the end. Don't waste time making things look good too early on. Ideally, try to get the pipeline to run on a simple example, and then keep adding features. Also, try to get *as much feedback* as possible on the *content* as *early* as possible. Let's now get the `ggplot` of the data to show in the document as well.

For this, simply add:

```
tar_read(plot_mtcars)
```

at the bottom of the `.Rmd` file. Running the pipeline again will now add the plot to the document. Before continuing, let me just remind you, again, of the usefulness of `{targets}` by changing the underlying data. Run the following:

```

write.csv(head(mtcars),
 "mtcars.csv",
 row.names = F)

```

and run the pipeline again. Because the data changed and every target depends on the data, the document gets entirely rebuilt. I hope that you see why this is really great: in case you need to be build weekly, daily, heck, even hourly reports, by using `{targets}` the updated report can now get built automatically. Restore the data by running:

```
write.csv(mtcars,
 "mtcars.csv",
 row.names = F)
```

and rebuilding the document by running the pipeline. We can work a little on the document itself, by transforming the output into nice looking table using `{flextable}`. But there is an issue however: the output of `summary()` on a `data.frame` object is not a `data.frame`, but a `table`. But `flextable()` expects a `data.frame`. So if you call `flextable()` on the output of `summary()`, you'll get an error message. Instead, we need a replacement for `summary()` that outputs a `data.frame`. This replacement is `skimr::skim()`; let's go back to our pipeline and change the call to `summary()` to `skim()` (after adding the `{skimr}` to the list of loaded packages, as well as `{flextable}`):

```
library(targets)
library(tarchetypes)

tar_option_set(packages = c(
 "dplyr",
 "flextable",
 "ggplot2",
 "skimr"
))

list(
 tar_target(
 path_data_mtcars,
 "mtcars.csv",
 format = "file"
),
 tar_target(
 data_mtcars,
 read.csv(path_data_mtcars)
),
 tar_target(
 summary_mtcars,
 skim(data_mtcars)
),
 tar_target(
 clean_mtcars,
 mutate(data_mtcars,
 am = as.character(am))
)
)
```

```

),
 tar_target(
 plot_mtcars,
 {ggplot(clean_mtcars) +
 geom_point(aes(y = mpg,
 x = hp,
 shape = am))}
),
 tar_render(
 my_doc,
 "my_document.Rmd"
)
)
)

```

In the .Rmd file, we can now pass the output of `tar_read(summary_mtcars)` to `flextable()`:

```

Load the summary

```{r}
tar_read(summary_mtcars) %>%
  flextable()
```

```

If you look at the output now, you'll see a nice table with a lot of summary statistics. Since the output of `skim()` is a `data.frame`, you can only keep the stats you want by `select()`ing the columns you need:

```

Load the summary

```{r}
tar_read(summary_mtcars) %>%
  select(Variable = skim_variable,
         Mean = numeric.mean,
         SD = numeric.sd,
         Histogram = numeric.hist) %>%
  flextable()
```

```

If you want to hide the R code in the output document, simply use `knitr::opts_chunk$set(echo = F)` in a chunk with the `include = FALSE` option. Here is what the final source code of the .Rmd could look like:

```

title: "mtcars is the best data set"
author: "mtcars enjoyer"
date: today

```{r, include = FALSE}
knitr::opts_chunk$set(echo = F)
```

Load the summary statistics

I really like to see the distribution of the variables as a cell of a table:

```{r}
tar_read(summary_mtcars) %>%
  select(Variable = skim_variable,
         Mean = numeric.mean,
         SD = numeric.sd,
         Histogram = numeric.hist) %>%
  flextable() %>%
  set_caption("Summary statistics for mtcars")
```

Graphics

The plot below is really nice, just look at it:

```{r, fig.cap = "Scatterplot of `mpg` and `hp` by type of transmission"}
tar_read(plot_mtcars) +
  theme_minimal() +
  theme(legend.position = "bottom")
```

```

As you can see, once I've used `targets::tar_read()`, I get back the object just as if I had generated it from within the `.Rmd` source file, and can simply add stuff to it (like changing the theme of the `ggplot`). Once you're happy with the contents, you can add `output:`

`word_document` to the header of the document (just below `date: today` for example) to generate a Word document.

## 13.8 Rewriting our project as a pipeline and `{renv}` redux

It is now time to turn our little project into a full-fledged pipeline. For this, we're going back to our project's folder and specifically to the `fusen` branch. This is the branch where we used `{fusen}` to turn our `.Rmd` into a package. This package contains the functions that we need to update the data. But remember, we wrote the analysis in another `.Rmd` file that we did not inflate, `analyse_data.Rmd`. We are now going to write a `{targets}` pipeline that will make use of the inflated package and compute all the targets required for the analysis. The first step is to create a new branch, but you could also create an entirely new repository if you want. It's up to you. If you create a new branch, start from the `rmd` branch, since this will provide a nice starting point.

```
owner@localhost git checkout rmd #switch to the rmd branch
owner@localhost git checkout -b pipeline #create and switch to the new branch called pipeline
```

If you start with a fresh repository, you can grab the `analyse_data.Rmd` from [here](#)<sup>3</sup>.

First order of business, let's delete `save_data.Rmd` (unless you started with an empty repo). We don't need that file anymore, since everything is now available in the package we developed:

```
owner@localhost rm save_data.Rmd
```

Let's now start an R session in that folder and install our `{housing}` package. Whether you followed along and developed the package, or skipped the previous parts and didn't develop the package by following along, install it from my Github repository. This ensures that you have exactly the same version as me. Run the following lines (but make sure that you started a fresh R session in the right folder):

```
remotes::install_github("rap4all/housing@fusen",
 ref = "1c86095")
```

This will install the package from my Github repository, and very specifically the version from the `fusen` branch at commit `387a52b` (you may need to install the `{remotes}` package first). Now that the package is installed, we can start building the pipeline. In the same R session, call `targets::tar_script()` which will give us a nice template `_targets.R` file:

---

<sup>3</sup><https://is.gd/L2GICG>

```
targets::tar_script()
```

You should at most have three files: `README.md`, `_targets.R` and `analyse_data.Rmd` (unless you started with an empty repo, in which case you don't have the `README.md` file). The `_targets.R` script will contain the required code to perform the computations that we need. Inside `analyse_data.Rmd`, we will simply load the required, pre-computed targets, and show them in the document.

First, we need to load the data. The two datasets we use are now part of the package, so we can simply load them using `data(commune_level_data)` and `data(country_level_data)`. But remember, `{targets}` only loves pure functions, and `data()` is not pure! Let's see what happens when you call `data(mtcars)`. If you're using RStudio, this is really visible: in a fresh session, calling `data(mtcars)` shows the following in the `Environment` pane:

What's described as 'mtcars' is not a 'data.frame', yet.

At this stage, `mtcars` is only a **promise**. It's only if you need to interact with `mtcars` that the **promise** becomes a `data.frame`. So `data()` returns a promise, does this mean that we can save that into a variable? If you try the following:

```
x <- data(mtcars)
```

And check out `x`, you will see that `x` contains the string `"mtcars"` and is of class `character`! So `data()` returns a promise by saving it directly to the global environment (this is a side-effect) but it returns a string. Because `{targets}` needs pure functions, if we write:

```
tar_target(
 target_mtcars,
 data(mtcars)
)
```

the `target_mtcars` target will be equal to the `"mtcars"` character string. This might be confusing, but just remember: a target must return something, and functions with side-effects don't always return something, or the thing we want. Also remember the example on plotting with `plot()`, which does not return an object. It's actually the same issue here.

So to solve this, we need a pure function that returns the `data.frame`. This means that it first needs to load the data, which results in a promise (which gets loaded into the environment directly), and then evaluate that promise. The function to achieve this is as follows:

```
read_data <- function(data_name, package_name){

 temp <- new.env(parent = emptyenv())
```

```

 data(list = data_name,
 package = package_name,
 envir = temp)

 get(data_name, envir = temp)
}

```

This function takes `data_name` and `package_name` as arguments, both strings.

Then, I used `data()` but with two arguments: `list =` and `package =`. `list =` is needed because we pass `data_name` as a string. If we did something like `data(data_name)` instead, hoping that `data_name` would get replaced by its bound value (`"commune_level_data"`) it would result in an error. This is because `data()` would be looking for a data set literally called `data_name` instead of replacing `data_name` by its bound value. The second argument, `package =` simply states that we're looking for that dataset in the `{housing}` package and uses the value of `package_name`. Now comes the `envir =` argument. This argument tells `data()` where to load the data set. By default, `data()` loads the data in the global environment. But remember, we want our function to be pure, meaning, it should only return the data object and not load anything into the global environment! So that's where the temporary environment created in the first line of the body of the function comes into play. What happens is that the functions loads the data object into this temporary environment, which is different from the global environment. Once we're done, we can simply discard this environment, and so our global environment stays clean.

The final step is using `get()`. Remember that once the line `data(list = data_name...)` has run, all we have is a promise. So if we stop there, the target would simply hold the character `"commune_level_data"`. In order to turn that promise into the data frame, we use `get()`. We've already encountered this function in Chapter 7. `get()` searches an object by name, and returns it. So in the line `get(data_name)`, `data_name` gets first replaced by its bound value, so `get("commune_level_data")` and hence we get the dataset back. Also, `get()` looks for that name in the temporary environment that was set up. This way, there is literally no interaction with the global environment, so that function is pure: it always returns the same output for the same input, and does not pollute, in any way, the global environment. After that function is done running, the temporary environment is discarded.

This seems overly complicated, but it's all a consequence of `{targets}` needing pure functions that actually return something to work well. Unfortunately some of R's default functions are not pure, so we need this kind of workaround. However, all of this work is not in vain! By forcing us to use pure functions, `{targets}` contributes to the general quality and safety of our pipeline. Once our pipeline is done running, nothing the global environment should have stayed untouched. This is because having stuff pollute the global environment can cause interactions with subsequent runs of the same (or another) pipeline.

Let's continue the pipeline: here is what it will ultimately look like:

```

library(targets)
library(tarchetypes)

tar_option_set(packages = "housing")

source("functions/read_data.R")

list(
 tar_target(
 commune_level_data,
 read_data("commune_level_data",
 "housing")
),

 tar_target(
 country_level_data,
 read_data("country_level_data",
 "housing")
),

 tar_target(
 commune_data,
 get_laspeyeres(commune_level_data)
),

 tar_target(
 country_data,
 get_laspeyeres(country_level_data)
),

 tar_target(
 communes,
 c("Luxembourg",
 "Esch-sur-Alzette",
 "Mamer",
 "Schengen",
 "Wincrange")
),

 tar_render(
 analyse_data,

```



```

 "analyse_data.Rmd"
)
)

```

Remember, the idea is that all the computations are taken out of the `.Rmd` file and put into a `{targets}` pipeline. This way, if certain targets need to get updated, we don't need to recompute everything when knitting the `.Rmd` document. Only the outdated targets get recomputed, and because the document simply retrieves the required targets from the cache, it gets knitted very quickly. Here is what `analyse_data.Rmd` now looks like:

```

title: "Nominal house prices data in Luxembourg"
author: "Bruno Rodrigues"
date: "`r Sys.Date()`"

Let's load the datasets (the Laspeyeres price index is already computed):

```{r}
tar_load(commune_data)
tar_load(country_data)
```

We are going to create a plot for 5 communes and compare the price evolution in
the communes to the national price evolution. Let's first load the communes:

```{r}
tar_load(communes)
```

```{r, results = "asis"}
res <- lapply(communes, function(x){

  knitr::knit_child(text = c(

    '\n',
    '## Plot for commune: `r x`',
    '\n',
    '```{r, echo = F}',
    'print(make_plot(country_data, commune_data, x))',
    '```'
  ))
})

```

```

    ),
    envir = environment(),
    quiet = TRUE)

})

cat(unlist(res), sep = "\n")

...

```

As you can see, the data gets loaded by using `targets::tar_load()` which loads the two pre-computed data sets. The end portion of the document looks fairly similar to what we had before turning our analysis into a package and then a pipeline. We use a child document to generate as many sections as required automatically (remember, Don't Repeat Yourself!). Try to change something in the pipeline, for example remove some communes from the `communes` object, and rerun the whole pipeline using `tar_make()`.

We are now done with this introduction to `{targets}`: we have turned our analysis into a pipeline, and now we need to ensure that the outputs it produces are reproducible. So the first step is to use `{renv}`; but as already discussed, this will not be enough, but it is essential that you do it! So let's initialize `{renv}`:

```
renv::init()
```

This will create an `renv.lock` file with all the dependencies of the pipeline listed. Very importantly, our Github package also gets listed:

```

"housing": {
  "Package": "housing",
  "Version": "0.1",
  "Source": "GitHub",
  "RemoteType": "github",
  "RemoteHost": "api.github.com",
  "RemoteRepo": "housing",
  "RemoteUsername": "rap4all",
  "RemoteRef": "fusen",
  "RemoteSha": "1c860959310b80e67c41f7bbdc3e84cef00df18e",
  "Hash": "859672476501daeea9b719b9218165f1",
  "Requirements": [
    "dplyr",
    "ggplot2",
    "janitor",

```

```

    "purrr",
    "readxl",
    "rlang",
    "rvest",
    "stringr",
    "tidyr"
  ]
},

```

If you look at the fields titled `RemoteSha` and `RemoteRef` you should recognize the commit hash and repository that were used to install the package:

```

"RemoteRef": "fusen",
"RemoteSha": "1c860959310b80e67c41f7bbdc3e84cef00df18e",

```

This means that if someone wants to re-run our project, by running `renv::restore()` the correct version of the package will get installed! To finish things, we should edit the `Readme.md` file and add instructions on how to re-run the project. This is what the `Readme.md` file could look like:

```

# How to run

- Clone the repository: `git clone git@github.com:rap4all/housing.git`
- Switch to the `pipeline` branch: `git checkout pipeline`
- Start an R session in the folder and run `renv::restore()` to install the project's dependencies
- Run the pipeline with `targets::tar_make()`.
- Checkout `analyse_data.html` for the output.

```

If you followed along, don't forget to change the `url` of the repository to your own.

13.9 Some little tips before concluding

In this section I'll be showing you some other useful functions included in the `{targets}` package that I think you should know!

13.9.1 Load every target at once

It is possible to load every target included in the cache at once using `tar_load_everything()`. But be careful, if your pipeline builds many targets, this can take some time!

13.9.2 Get metadata information on your pipeline

`tar_meta()` will return a data frame with some information about the pipeline. I find it quite useful after running the pipeline, and it turns out that some warnings or errors were raised. This is how this data frame looks like:

```
targets::tar_meta()
```

(I have removed the last columns, or it would fit)

```
# A tibble: 11 × 18
  name      type data  command depend [...]
<chr>    <chr> <chr> <chr>    <chr>  [...]
1 analyse_d... stem  c251... 995812... 3233e... [...]
2 commune_d... stem  024d... 85c2ab... ec7f2... [...]
3 commune_l... stem  fb07... f48470... ce0d8... [...]
4 commune_l... stem  fb07... 2549df... 15e48... [...]
5 communes    stem  b097... be3c56... a3dad... [...]
6 country_d... stem  ae21... 9dc7a6... 1d321... [...]
```

There are more columns than those I'm showing, of great interest are the columns **warnings** and **error**. In the example below, I have changed the code to `read_data()`, and now it raises a warning:

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

```
• start target commune_level_data
• built target commune_level_data [0.61 seconds]
• start target country_level_data
• built target country_level_data [0.02 seconds]
  skip target communes
  skip target commune_data
  skip target country_data
  skip target analyse_data
• end pipeline [0.75 seconds]
```

Warning messages:

1: this is a warning

2: this is a warning

3: 2 targets produced warnings. Run `tar_meta(fields = warnings, complete_only = TRUE)` for the

>

Because warnings were raised, the pipeline raises another warning telling you to run `tar_meta(fields = warnings, complete_only = TRUE)` so let's do it:

```
tar_meta(fields = warnings, complete_only = TRUE)
```

This code returns a data frame with the name of the targets that produced the warning, alongside the warning.

```
# A tibble: 2 × 2
  name                warnings
  <chr>              <chr>
1 commune_level_data this is a warning
2 country_level_data this is a warning
```

So now you can better see what is going on.

13.9.3 Make a target (or the whole pipeline) outdated

With `tar_invalidate()` you can make a target outdated, so that when you rerun the pipeline, it gets re-computed (alongside every target that depends on it). This can sometimes be useful to make sure that everything is running correctly. Try running `tar_invalidate("communes")` and see what happens. It is also possible to completely nuke the whole pipeline and rerun it from scratch using `tar_destroy()`. You'll get asked to confirm if that's what you want to do, and if yes, rerunning the pipeline after that will start from scratch.

13.9.4 Customize the network's visualisation

By calling `visNetwork::visNetworkEditor(tar_visnetwork())`, a Shiny app gets started that lets you customize the look of your pipeline's network. You can play around with the options and see the effect they have on the look of the network. It is also possible to generate R code that you can then paste into a script to consistently generate the same look.

13.9.5 Use targets from one pipeline in another project

If you need to load some targets inside another project (for example, because you need to reference an older study), you can do so easily with `{withr}`:

```
withr::with_dir(
  "path/to/root/of/project",
  targets::tar_load(target_name))
```

13.9.6 Understanding this cryptic error message

Sometimes, when you try to run the pipeline, you get the following error message:

Error:

```
! Error running targets::tar_make()
  Target errors: targets::tar_meta(fields = error, complete_only = TRUE)
  Tips: https://books.ropensci.org/targets/debugging.html
  Last error: argument 9 is empty
```

The last line is what interests us here: “Last error: argument 9 is empty”. It is not clear which target is raising this error: that’s because it’s not a target that is raising this error, but the pipeline itself! Remember that the pipeline is nothing but a list of targets. If your last target ends with a `,` character, `list()` is expecting another element. But because there’s none, this error gets raised. Try the following `list(1,2,)` and you will get the same error message. Simply check your last target, there is a `,` in there that you should remove!

13.10 Conclusion

I hope to have convinced you that you need to add build automation tools to your toolbox. `{targets}` is a fantastic package, because it takes care of something incredibly tedious for you. By using `{targets}` you don’t have to remember which objects you need to recompute when you need to change code. You don’t need to rewrite your code to make it run in parallel. And by using `{renv}`, other users can run your pipeline in a couple of lines and reproduce the results.

In the next chapter, we will be going deeper in the iceberg of reproducibility still.

14 Advanced topics in reproducibility

If the book ended at the end of the previous chapter, it would have been titled “Building analytical pipelines with R”, because we have not ensured that the pipeline we built is reproducible. We did our best though:

- we used functional and literate programming;
- we documented, tested and versioned the code;
- we used `{renv}` to record the dependencies of the project;
- the project is now a pipeline and re-running it is as easy as it can possibly get.

But there are still many variables that we need to consider. If we go back to the reproducibility iceberg, you will notice that we can still go deeper. As the code stands now, we did our best using programming paradigms and libraries, but now we need to consider another tool, called Docker.

As already mentioned in the introduction and in Chapter 10, `{renv}` *only* restores package versions. The R version used for the analysis only gets recorded. So to make sure that the pipeline reproduces the same results, you’d need to install the same R version that was used to build the pipeline originally.

Next comes the operating system on which the pipeline was developed. In practice, this rarely matters, but there have been cases where different code produces different results on different operating systems.

And finally, I believe that we are in a transition period when it comes to hardware architecture. Apple will very likely completely switch over to an ARM architecture with their Apple silicon CPUs (as of writing, the Mac Pro is the only computer manufactured by Apple that doesn’t use an Apple silicon CPU and only because it was released in 2019) and it wouldn’t surprise me if other manufacturers follow suit and develop their own ARM cpus.

So, as I explained in the previous chapter, we want our pipeline to be the composition of pure functions. Nothing in the global environment (apart from `{target}`-specific options) should influence the runs of the pipeline. But, what about the environment R is running in? The R engine is itself running in some kind of environment. This is what I’ve explained above: operating system (and all the math libraries these ship that R relies on to run code) and hardware.

Think about it this way: when you running a pure function `f()` of one argument you think you do this:

`f(1)`

but actually what you're doing is:

```
f(1, "windows 10 - build 22H2 - patch 10.0.19045.2075", "intel x86_64 cpu i9-13900F", "R ver
```

and so on. `f()` is only pure as far as the R version currently running `f()` is concerned. But everything else should also be taken into account! Remember, in technical terms this means that our function is not referentially transparent.

And this is what Docker allows us to do: turn our the pipeline referentially transparent, by fixing R's and the operating system's versions (and the CPU architecture us well).

14.1 First steps with Docker

To write your own Dockerfile, you need some familiarity with the Linux cli, so here's...

14.2 A primer on the Linux command line

14.3 Dockerizing your project

15 Continuous integration and continuous deployment/delivery

16 Conclusion of part 2

References

- Arel-Bundock, Vincent. 2022. “modelssummary: 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.
- Chan, Chung-hong, and David Schoch. 2023. “RANG: Reconstructing Reproducible r Computational Environments.” arXiv. <https://doi.org/10.48550/ARXIV.2303.04758>.
- 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.
- Wickham, Hadley, and Jenny Bryan. 2023. *R Packages (2e)*. <https://r-pkgs.org/>.
- 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.