Building reproducible analytical pipelines with Python

Bruno Rodrigues

2024-01-13

Table of contents

Welcome!

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

This is the Python edition of the book "Building reproducible analytical pipelines", and it's a work-in-progress. If you're looking for the R version, visit this link.

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 and DevOps 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 team-mates 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://b-rodrigues.github.io/raps_with_py/ and you'll be able buy a DRM-free Epub or PDF on Leanpub[1] once there's more content.

This is the *Python edition* of my book titled Building reproducible analytical pipelines with R[2]. This means that a lot of text is copied over, but the all of the code and concepts are completely adapted to the Python programming language. This book is also shorter than the R version. Here's the topics that I will cover:

- Dependency management with `pipenv`;

[1] https://leanpub.com/
[2] https://raps-with-r.dev

- Some thoughts on functional programming with Python;
- Unit and assertive testing;
- Build automation with `ploomber`;
- Literate programming with Quarto;
- Reproducible environments with Docker;
- Continuous integration and delivery.

While this is not a book for beginners (you really should be familiar with Python before reading this), I will not assume that you have any knowledge of the tools discussed. 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 TO UPDATE

If you want to get to know me better, read my bio[3].

You can submit issues, PRs and ask questions on the book's Github repository[4].

[3]https://www.brodrigues.co/about/me/
[4]https://github.com/b-rodrigues/raps_with_py

Preface

I don't like Python. Or rather, don't like using it to analyze data. I believe that certain design choices were made that make Python a subpar language for analyzing data. That being said, Python is currently the most popular general purpose programming language, and also the most popular language to analyze data, especially in machine learning and AI. As such, it is a good idea to at least know your way around it even if, like me, you prefer using the R programming language.

I state that I don't like Python because I want to make something very clear, right from the start. I am not an expert in Python. I know enough to get things done, but it is not a language that I know well. I consider myself an expert in R, but definitely not in Python. So why write a book on Python in that case? Well, in truth, this is not a book about Python.

This book is about building reproducible pipelines. And in order to build reproducible pipelines, you need to apply certain general ideas. This book will discuss these ideas, and illustrate them using the Python programming language. I wrote such a book already: you can read it here[5]. The book was well received: lots of people contacted me to thank me for having written it, I was invited to give talks several times on the book already, give some workshops and some people are already asking me for a second edition. I thought it would be an interesting challenge to write

[5] www.raps-with-r.dev

a Python edition. It would be a good excuse to dive back into Python after having barely touched it since my Phd days, more than 10 years ago, where I used it alongside R for a project. It is also a good opportunity to see what is currently available in the Python programming language, and if it would be possible to reproduce the way I work with R.

There are currently some interesting packages available for Python that are quite close to some available for R. But R and Python have some very fundamental differences in their design that make using Python feel awkward if you're used to R, especially if take full advantage of R's functional programming features. But it's not just these design choices that differentiate both languages, there are also differences in culture, in how people use them. For example, notebooks are very popular in the Python world, but it's quite rare to see someone share R code through a notebook. In general, R users tend to write code as plain-text scripts or in Markdown using Rmarkdown or Quarto. Throughout this book, I will be writing Python with a heavy R accent, mainly for two reasons: first, because I'm fluent in R, but not in Python, as I've stated above. Second, because I also think that some of R's idioms actually make it more readable than equivalent Python code, and thus we can make Python more readable if we make it look more like R.

If the previous sentence didn't scare you into continuing and you are actually curious to see what I mean, then read on, you're in the perfect state of mind to approach this book!

Let me finish this section by talking a little bit about the history of these books (the original R edition and this one). It all started when a former colleague of mine contacted me in the summer of 2022 to ask me if I was interested in teaching a course for the Data Science Master degree at the University of Luxembourg. Long story short, I've decide to teach students how to

set up data science projects in a way that they're reproducible. 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 of *Reproducible Analytical Pipelines* as developed by the Office for National Statistics (henceforth ONS). I found the name "Reproducible Analytical Pipeline" (henceforth RAP) really perfect for what I was aiming at. The ONS team responsible for evangelising RAPs also published a free ebook[6] in 2019 already. Another big source of inspiration is Software Carpentry[7] 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 principal investigator 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. This was also the project where I've used Python.

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) to users of the Python programming language, even if I'm not one.

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

Enjoy!

[6]https://ukgovdatascience.github.io/rap_companion/
[7]https://software-carpentry.org/

1 Setting up a development environment

I have to start with on of the hardest chapters of the book, how to set up a development environment for Python.

If you are already using Python, you already have an environment set up and might be familiar with `pyenv` or other similar tools. I would still suggest you read this chapter and see if you agree with how I approach this issue. You may want to adapt your current workflow to it, or keep on doing what you've been doing up until now, it is up to. If you're completely new to Python, then you definitely need to read this chapter, but also, I need to remind you that this is not a book about Python per se. So I won't be teaching you any Python (I wouldn't really be competent to do so either) and you might want to complement reading this book with another that focuses on actually teaching Python. Remember, this book is about building reproducible analytical pipelines!

1.1 Why is installing Python such a hard problem?

If you google "how to install Python" you will find a surprising amount of articles explaining how to do it. I say "surprising

amount" because one might expect to install Python like any other piece of software. If you're already familiar with R, you could think that installing Python would be done the same way: download the installer for your operating system, and then install it. And, actually, you can do just that for Python as well. So why are there 100s of articles online explaining how to install Python, and why aren't all of these articles simply telling you to download the installer to install Python? Why did I write this chapter on installing Python?

Well, there are several thing that we need to deal with if we want to install and use Python the "right way". First of all, Python is pre-installed on Linux distributions and older versions of macOS. So if you're using one of these operating systems, you could use the built-in Python interpreter, but this is not recommended. The reason being that these bundled versions are generally older, and that you don't control their upgrade process, as these get updated alongside the operating system. On Windows and newer versions of macOS, Python is, as far as I know, never bundled, so you'd need to install it anyways.

Another thing that you need to consider is that newer Python versions can introduce breaking changes, making code written for an earlier version of Python not run on a newer version of Python. This is not a Python-specific issue: it happens with any programming language. So this means that ideally you would want to bundle a Python version with your project's code.

The same holds true for individual packages: newer versions of packages might not even work with older releases of Python, so to avoid any issues, an analysis would get bundled with a Python release and Python packages. This bundle is what I call a development environment, and in order to build such development environment, specific tools have to be used. And there's a lot of these tools in the Python ecosystem… so much so that when

you're first starting, you might get lost. So here is he two tools that I use for this, and that I think work quite well: `pyenv` and `pipenv`.

1.2 First step: installing Python

In this section, I will assume that you have no version of Python already installed on your computer. If you do have Python, you may want to skip this section: but know that if you're not using `pyenv` to download and install Python versions, you will need to keep installing your environments manually. `pipenv` however install environments automatically if `pyenv` is available, so you might want to switch over to `pyenv`.

So first of all, let's install `pyenv`. `pyenv` is a tool that allows you to install as many versions of Python that you need, and that does not depend on Python itself, so there's no bootstrap problem. But it is only available for Linux and macOS: Windows users should install `pyenv-win`: the following subsection deals with installing `pyenv` on Linux and macOS, the next one with installing `pyenv-win` for Windows.

1.2.1 `pyenv` for Linux or macOS

I recommend you read and follow the official instructions[1], but I provide here the main steps. In case something doesn't work as expected, refer to the official documentation linked previously. First, install the build dependencies by following the instructions here[2]. On a Linux distribution like Ubuntu, this means

[1] https://github.com/pyenv/pyenv?tab=readme-ov-file#installation
[2] https://github.com/pyenv/pyenv/wiki#suggested-build-environment

installing a bunch of development libraries from your usual package manager. On macOS, this means installing Xcode and also a bunch of packages with Homebrew. Also, make sure you have installed Git as well. We will be using Git later in the book too, but it is required to install `pyenv`.

Then, on macOS, you can use Hombrew to install `pyenv`:

```
brew update
brew install pyenv
```

For Linux distributions, run the automatic installer by opening a terminal and executing this line here:

```
curl https://pyenv.run | bash
```

Now comes the more complex part of the installation process. You need to edit some files for your terminal, and these files are different depending on which shell you use. For most Linux distributions, if not all, the default terminal will use bash, and on macOS the default shell is zsh. The detailled instructions, which I recommend you read, are here[3], but here are my recommendations: if you're using a Linux distribution, you likely are using bash, to find out, run `echo $0` in a terminal. If your terminal is using bash, "bash" will get printed, if not, "zsh" will get printed instead. If your terminal using bash, run the following in a terminal:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'command -v pyenv >/dev/null || export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
```

[3] https://github.com/pyenv/pyenv?tab=readme-ov-file#set-up-your-shell-environment-for-pyenv

and then, you need to check whether you have a `.profile`, `.bash_profile` or a `.bash_login` file in your home directory. These are files used at startup by bash to set some variables and options. They're similar to the `.bashrc` file, but the `.bashrc` is executed when starting a terminal when you're already logged into the system, while `.bash_profile`, `.bash_login` and `.profile` get executed when logging in through ssh into the system. If you have a `.profile` file in your HOME, this should return something:

```
ls .profile
```

if nothing gets returned, you might have instead a `.bash_profile` or `bash_login` file instead, try with:

```
ls .bash_profile
```

or

```
ls .bash_login
```

If you have more than one, only add the lines to the `.bash_profile`:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >>
~/.bash_profile
echo 'command -v pyenv >/dev/null || export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bash_profile
echo 'eval "$(pyenv init -)"' >> ~/.bash_profile
```

but if you have only have a `.bash_login` file, run this:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >>
~/.bash_login
echo 'command -v pyenv >/dev/null || export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bash_login
echo 'eval "$(pyenv init -)"' >> ~/.bash_login
```

or if you only have a `.profile` file, or none of these three files at all, run these lines:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >>
~/.profile
echo 'command -v pyenv >/dev/null || export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.profile
echo 'eval "$(pyenv init -)"' >> ~/.profile
```

If you're running zsh (which would be the case on macOS, or if you changed form bash to zsh on Linux), then you need to add these lines to the `.zshrc` file:

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
echo '[[ -d $PYENV_ROOT/bin ]] && export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc
echo 'eval "$(pyenv init -)"' >> ~/.zshrc
```

and also to `.zprofile` or `.zlogin`, depending on which you have (if you have none, put the lines into `.zprofile`):

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >>
~/.zprofile
echo '[[ -d $PYENV_ROOT/bin ]] && export
PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zprofile
echo 'eval "$(pyenv init -)"' >> ~/.zprofile
```

If you're using another shell, such as fish, you can checkout the instructions on `pyenv`'s GitHub repository, or simply add these lines in the equivalent startup files.

Finally, also add this line to your `.bashrc` or `.zshrc`:

```
eval "$(pyenv virtualenv-init -)"
```

to load `pyenv-virtualenv` automatically (as stated by the installer).

1.2.2 `pyenv-win` for Windows

If you're using Windows, you should install `pyenv-win` instead, which you can find here[4]. Installation on Windows is similar to how you install `pyenv` on Linux, if you choose the PowerShell method. It's simply running this line (see here[5]) in Powershell:

```
Invoke-WebRequest -UseBasicParsing -Uri
"https://raw.githubusercontent.com/pyenv-win/pyenv-win/master
-OutFile "./install-pyenv-win.ps1";
&"./install-pyenv-win.ps1"
```

Open a new PowerShell prompt and run `pyenv`. If you see a "command not found" error, you need to manually add `pyenv` to the PATH. On Windows, this is done graphically. Click on the start menu, and look for "Edit environment variables for your account".

[4] https://github.com/pyenv-win/pyenv-win
[5] https://github.com/pyenv-win/pyenv-win/blob/master/docs/installation.md#powe

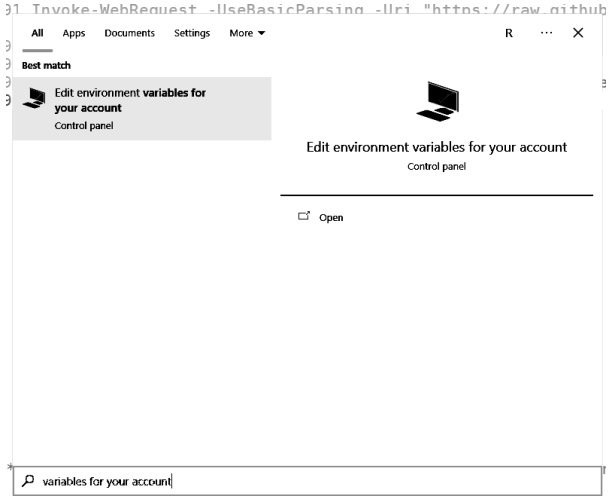


Figure 1.1: Look for 'Edit environment variables for your account'

Then, click on the "New…" button, and the `PYENV`, `PYENV_HOME` and `PYENV_ROOT`. Make sure to adjus the paths to the ones on your machine.

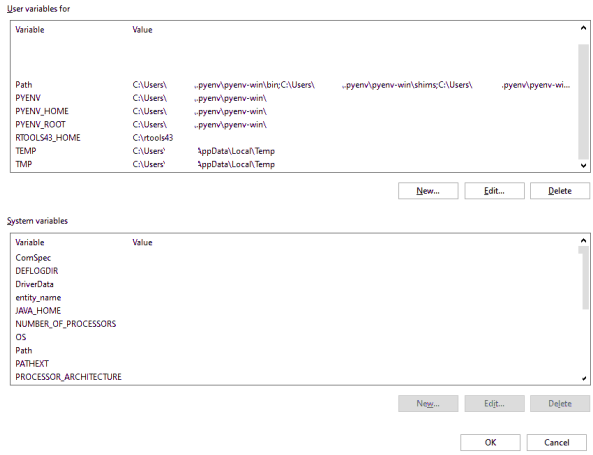


Figure 1.2: Make sure your environment variables are correctly set

Also check the FAQ[6] if you need further details.

1.2.3 Using pyenv

Now that you have `pyenv` installed, we can use it to install as many different Python versions that we need, typically one per project. First, I'll install a version of Python that I want to use. Let's got with the latest available. Let's see what is available:

```
pyenv install -l
```

As of writing, the latest version that is not in beta is `3.10.5`, so let's install this one (if you're reading this from the future and a new version of Python is available, install `3.10.5` nonetheless):

[6]https://github.com/pyenv-win/pyenv-win/wiki

```
pyenv install 3.10.5
```

this will install Python `3.10.5`. But how do we use it now? Let's suppose that we want to use it for a project. Let's create a folder that will contain the different scripts that we are going to write in the next chapter and called it `housing`.

Open a terminal inside that folder, or open a terminal and `cd` into that folder:

```
cd ~/home/Documents/housing
```

(depending on your operating system, you could first open the folder, then right click somewhere empty and then click on *Open in terminal*, which would open a terminal in that folder). We would like to use Python `3.10.5` for this project, and we would like to do so each time we interact with the files in that folder. We need to set version `3.10.5` as the *local* Python version, meaning, the Python version that will be used for this project only:

```
pyenv local 3.10.5
```

This will generate a `.python-version` with the following line in it:

```
3.10.5
```

This file will only be used by `pyenv`, not `pipenv`, but it's nice to keep it as a reminder of which Python version you actually need for this project. `pipenv` specifies the Python version in its own file, which I will discuss in the next section.

1.3 Second step: pipenv

We can now use our local Python interpreter to install `pipenv`:

```
pip install --user pipenv
```

We can now use `pipenv` to install the packages for our project. But why don't we just keep using `pip` instead of `pipenv`? Why introduce yet another tool? In my opinion, `pipenv` has one absolutely crucial feature for reproducibility: `pipenv` enables deterministic builds, which means that when using `pipenv`, we will *always* get exactly the same packages installed.

"But isn't that exctaly what using `requirements.txt` file does?" you wonder. You are not entirely wrong. After all, if you make the effort to specify the packages you need, and their versions, wouldn't running `pip install -r requirements.txt` also install exactly the same packages?

Well, not quite. Imagine for example that you need a package called `hello` and you put it into your `requirements.txt` like so:

```
hello==1.0.1
```

Suppose that `hello` depends on another package called `ciao`. If you run `pip install -r requirements.txt` today, you'll get `hello` at version `1.0.1` and `ciao`, say, at version `0.3.2`. But if you run `pip install -r requirements.txt` in 6 months, you would still get `hello` at version `1.0.1` but you might get a newer version of `ciao`. This is because `ciao` itself is not specified in the `requirements.txt`, unless you made sure to add it (and then also add its dependencies, and their dependencies…). `pipenv` takes care of this for you, and adds further security checks by comparing sha256 hashes from the lock file to the ones from

the downloaded packages, making sure that you are actually installing what you believe you are.

Now that `pipenv` is installed, let's start using it to install the packages we need for our project:

```
pipenv --python 3.10.5 install
beautifulsoup4==4.12.2 pandas==2.1.4
plotnine==0.12.4 skimpy==0.0.11
```

(Little sidenote: in Chapter 2, we will be working together on a project using real data with `polars` instead of `pandas`. But for now, let's get `pandas` installed: I'll do a short comparison of the two, and then we'll keep on using `polars` for the remainder of the book.)

As you can see, I chose to install specific versions of packages. This is because I want you to follow along with the same versions as in the book. You could remove the `==x.y.z` strings from the command above to install the latest versions of the packages available if you prefer, but then there would be no guarantee that you would find the same results as I do in the remainder of the book. Also, if you don't specify versions, the `Pipfile` file won't specify them either, only the `Pipfile.lock` will, which in certain cases could lead to undesired behaviour. I won't go into specifics, because I highly recommend you always take the time to specify package versions when installing. Check package versions on Pypi. For example, here[7] is the page to check out versions of the pandas package.

You should see now two new files in the `housing` folder, `Pipfile` and `Pipfile.lock`. Start by opening `Pipfile`, it should look like this:

[7] https://pypi.org/project/pandas/

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
beautifulsoup4 = "==4.12.2"
pandas = "==2.1.4"
plotnine = "==0.12.4"
skimpy = "==0.0.11"

[dev-packages]

[requires]
python_version = "3.10"
python_full_version = "3.10.5"
```

I think that this file is pretty self-evident: the packages being used for this project are listed alongside their versions. The exact Python version is also listed. Sometimes, depending on how you set up the project, it could happen that the Python version listed is not the same, or that only the field `python_verison=` is listed. In this case, I highly recommend you change it to match the same version as in the `.python-version` file that was generated by `pyenv`, and add the `python_full_version=` field if necessary, in order to avoid any possible differences in how the code runs. If you edited the `Pipfile` then you need to run `pipenv lock` to regenerate the `Pipfile.lock` file as well:

```
pipenv lock
```

this will make sure to also set the required/correct Python version in there.

If you open the `Pipfile.lock` in a text editor, you will see that it is a json file and that is also lists the dependecies of your project, but also the dependencies' dependencies. You will also notice several fields called `hashes`. These are there for security reasons: whenever someone, (or you in the future) will regenerate this environment, the packages will get downloaded and their hashes will get compared to the ones listed in the `Pipfile.lock`. If they don't match, then something very wrong is happening and packages won't get installed.

To check whether everything installed correctly, drop into the development shell using:

```
pipenv shell
```

and start the Python interpreter:

```
python
```

Then check that the correct versions of the packages were installed:

```
import pandas as pd
pd.__version__
```

You should see `2.1.4` as the listed version.

1.4 Debugging

It can sometimes happen that `pipenv` does not use the right version of Python. Running `pipenv shell` and then starting the Python interpreter will start a different version of Python than the one you expect/need.

In that case, it might be best to simply start over. Start by completely removing the virtual environment and packages by running:

```
pipenv --rm
```

Make sure that the correct version of Python (or rather, the version you need) is specified in the `Pipfile`. For example, in my case:

```
[requires]
python_full_version = "3.10.5"
```

(you might instead see `python_version = "3.10"`, or any other version, in that case I recommend you specify `python_full_version` as I did above, check the `.python-version` file that was generated by `pyenv` if you don't remember the right version).

Now, restore the environment and packages by running:

```
pipenv install
```

`pipenv install` will regenerate the `Pipfile.lock` file to list the required Python version, and then install the packages. It essentially runs `pipenv lock` to generate the lock file, and then `pipenv sync` to install the packages from the lock file.

Check if `pipenv` now uses the correct Python version.

```
pipenv run python --version
```

1.5 A high-level description of how to set up a project

Ok, so to summarise, we install `pyenv` which will make it easy to install any version of Python that we require, and then we installed `pipenv`, which we use to install packages. The advantage of using `pipenv` is that we get deterministic builds, and `pipenv` works well with `pyenv` to build project-specific environments.

The way I would suggest you use these tools now, is that for each project, you install the latest available version of Python, define it as the local version using `pyenv local`, and then install packages by specyifing their versions, like so:

```
pipenv --python 3.10.5 install
beautifulsoup4==4.12.2 pandas==2.1.4
plotnine==0.12.4 skimpy==0.0.11
```

(Replace the versions of Python and packages by the latest, or those you need.)

This will ensure that your project uses the correct software stack, and that collaborators or future you will be able to regenerate this environment by calling `pipenv sync`. `pipenv sync` differs from `pipenv install` because it will not touch the `Pipfile.lock` file, and will just download the packages as listed in the lock file. This is also the command that we will use later, in the chapter on CI/CD.

2 Project start

In this chapter, we are going to work together on a very simple project. This project will stay with us until the end of the book. As we will go deeper into the book together, you will rewrite that project by implementing the techniques I will teach you. By the end of the book you will have built a reproducible analytical pipeline. To get things going, we are going to keep it simple; our goal here is to get an analysis done, that's it. We won't focus on reproducibility (well, not beyond what was done in the previous chapter to set up our development environment). We are going to download some data, and analyse it, that's it. But before all of that, I will present the Polars package. In the preface, I said that this wasn't supposed to be a book about Python, so why am I talking about a specific package? It's because I think that Polars, unlike Pandas, has some features and design choices that actually improve reproducibility.

2.1 The Polars package and why you should ditch Pandas in its favour

2.2 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.

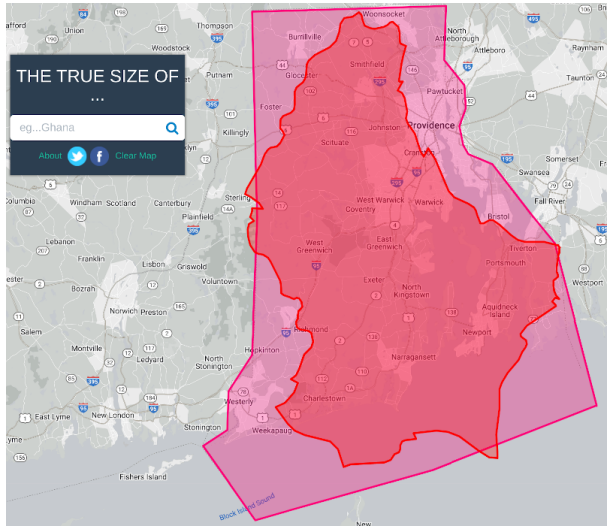
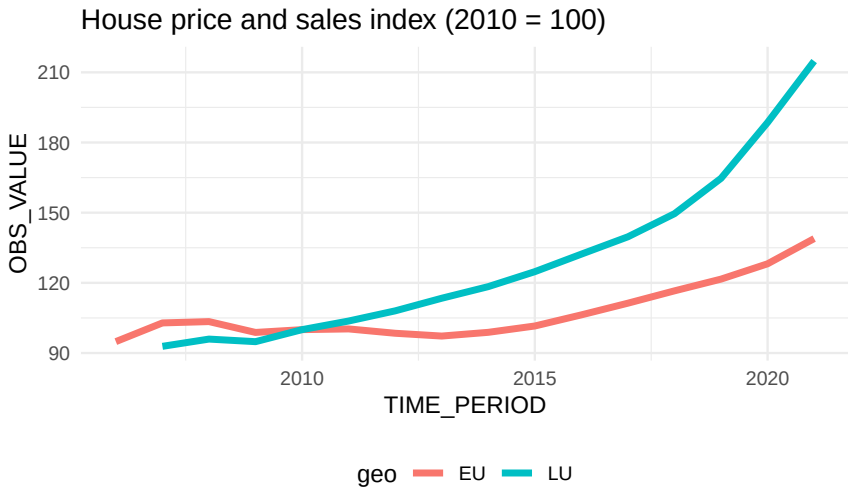


Figure 2.1: Luxembourg is about as big as the US State of Rhode Island.

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 then 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 over time for Luxembourg and the European Union:



Source: Eurostat

If you want to download the data, click here[1].

[1] https://is.gd/AET0ir

Let us paste the definition of the HPI in here (taken from the HPI's metadata[2] 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 right 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, 0.5% of residents owned 50% of the buildable land for housing purposes (Source: *Observatoire de l'Habitat, Note 29*, archived download link[3]).

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

[2]https://archive.is/OrQwA, archived link for posterity.
[3]https://archive.org/download/note-29/note-29.pdf

no index (and the data is trapped inside Excel and not ready for analysis with R). 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.

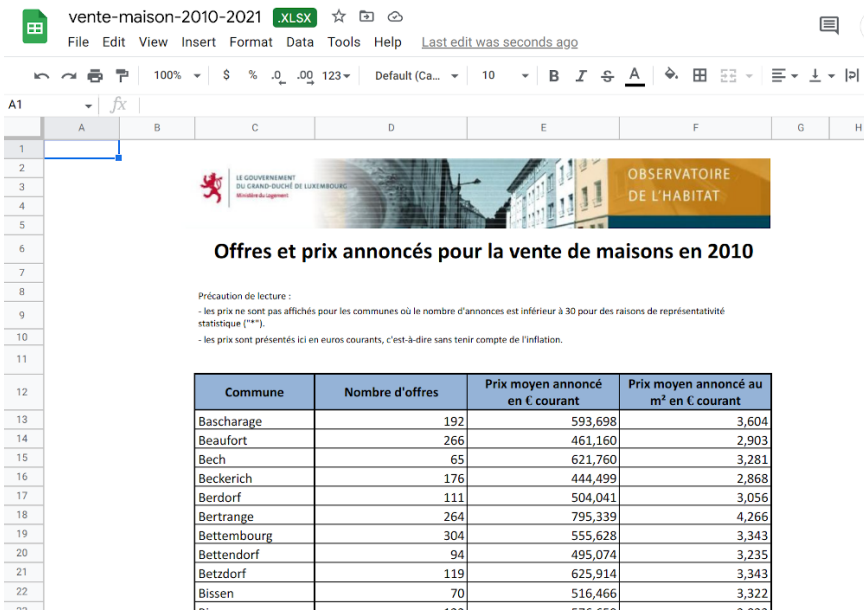
So to summarise; our goal is to:

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

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

2.3 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 file made for human consumption:


vente-maison-2010-2021 .XLSX

File Edit View Insert Format Data Tools Help Last edit was seconds ago

LE GOUVERNEMENT DU GRAND-DUCHÉ DE LUXEMBOURG

OBSERVATOIRE DE L'HABITAT

Offres et prix annoncés pour la vente de maisons en 2010

Précaution de lecture :

- les prix ne sont pas affichés pour les communes où le nombre d'annonces est inférieur à 30 pour des raisons de représentativité statistique (**).
- les prix sont présentés ici en euros courants, c'est-à-dire sans tenir compte de l'inflation.

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

Figure 2.2: An Excel file meant for human eyes.

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 format is a nightmare. We could call data like this *machine-friendly* data.

If you want to follow along, you can download the Excel file here[4] (downloaded on January 2023 from the luxembourguish open data portal[5]). But you don't need to follow along with code, because I will link the completed scripts for you to download later.

Each sheet contains a dataset with the following columns:

- *Commune*: the commune (the smallest administrative division of territory);
- *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.

[4] https://is.gd/1vvBAc

[5] https://data.public.lu/en/datasets/prix-annonces-des-logements-par-commune/

For ease of presentation, I'm going to show you each step of the analysis here separately, but I'll be putting everything together in a single script once I'm done explaining each step. So first, let's load some packages:

```
import polars as pl
import polars.selectors as cs
import re
```

I will be using the `polars` package to manipulate data.

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

```
# The url below points to an Excel file
# hosted on the book's github repository
url <- "https://is.gd/1vvBAc"

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

download.file(url, raw_data,
              method = "auto",
              mode = "wb")

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"))
```

If you are familiar with the `{tidyverse}` (Wickham et al. 2019) the above code should be quite easy to follow. We start by downloading the raw Excel file and saving 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:

```
raw_data
```

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'll explore the data: the naming of the communes is not consistent. Let's take a look:

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

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)
```

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

```
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))
```

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))
```

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

Commune	Nombre d'offres	Prix moyen annoncé en € courant	Prix moyen annoncé au m² en € courant
Consthum	29	*	*
Esch-sur-Sûre	7	*	*
Heiderscheid	29	*	*
Hoscheid	26	*	*
Saeul	14	*	*
Moyenne nationale		569,216	3,251
Total d'offres	19,278		

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

Figure 2.3: Always look at your data.

So it turns out that 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. The very last row states the source of the data and we can also remove it. Finally, in the screenshot above, we see another row that we don't see in our filtered data frame: one where `n_offers` is missing. This row gives the national average for columns `average_prince_nominal_euros` and `average_price_m2_nominal_euros`. What we are going to do is create two datasets: one with data on communes, and the other on national prices. Let's first remove the rows stating the sources:

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

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

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

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

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

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

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

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[6].

An issue with scraping tables off the web is that they might change in the future. It is therefore a good idea to save the page by right clicking on it and then selecting save as, and then re-hosting it. I use Github pages to re-host the Wikipedia page above here[7]. I now have full control of this page, and won't

[6] https://w.wiki/6nPu
[7] https://is.gd/lux_communes

get any bad surprises if someone decides to eventually update it. Instead of re-hosting it, you could simply save it as any other file of your project.

So let's scrape and save this list:

```
current_communes <- "https://is.gd/lux_communes"
 ↪  |>
  rvest::read_html() |>
  rvest::html_table() |>
  purrr::pluck(2) |>
  janitor::clean_names() |>
  dplyr::filter(name_2 != "Name") |>
  dplyr::rename(commune = name_2) |>
  dplyr::mutate(commune =
 ↪  stringr::str_remove(commune, " .$"))
```

We scrape the table from the re-hosted 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 second 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`) and then I use the `{dplyr}` package for some further cleaning and renaming; the very last step removes a dagger symbol next to certain communes names, in other words it turns "Commune †" into "Commune".

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

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

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 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 from Wikipedia, and here again, I decide to re-host it on Github pages to avoid problems in the future:

```
former_communes <-
 ↪  "https://is.gd/lux_former_communes" |>
  rvest::read_html() |>
  rvest::html_table() |>
  purrr::pluck(3) |>
  janitor::clean_names() |>
  dplyr::filter(year_dissolved > 2009)

former_communes
```

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)
```

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 start with analysing the data. Take a look here[8] to see the finalised script. Also read some of the comments that I'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

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

it in the future. As it turns out, this script is not reproducible. But we will discuss this in much more detail later on. For now, let's analyse our cleaned data.

2.4 Analysing the data

We are now going to analyse the data. The first thing we are going to do is compute a Laspeyeres price index. This price index allows us to make comparisons through time; for example, the index at year 2012 measures how much more expensive (or cheaper) housing became relative to the base year (2010). However, since we only have one 'good' (housing), this index becomes quite simple to compute: it is nothing but the prices at year t divided by the prices in 2010 (if we had a basket of goods, we would need to use the Laspeyeres index formula to compute the index at all periods).

For this section, I will perform a rather simple analysis. I will immediately show you the R script: take a look at it here[9]. For the analysis I 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 I'm 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[10]) and Wincrange. However, let's ask ourselves the following important questions:

[9] https://is.gd/qCJEbi
[10] https://en.wikipedia.org/wiki/Schengen_Area

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

Let's answer these questions one by one.

2.5 Your project is not done

2.5.1 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, and another to actually run the analysis proper. Performing the analysis 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 the analysis. 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 dozens of scripts), this gets even worse. It might not even be easy for you to remember how to run this in 5 months!

And what about the required dependencies? Many packages were used in the analysis. How should these get installed? Ide-

ally, 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 be documented, but listing the packages that were used for an analysis and their versions takes quite some time. Thankfully, in part 2, we will learn about the `{renv}` package to deal with this in a couple lines of code.

2.5.2 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 such 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.

2.5.3 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 code even inside the same project. Look at the last part of the `analysis.R` script: we copied 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 reusing them inside the same project.

2.5.4 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 we use. 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 which 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 they were wrong, for example in the context of an audit.

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

2.6 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.

References

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686.