Building reproducible analytical pipelines with Python

Bruno Rodrigues

2024-08-12

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]https://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[6] 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[7] (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[8] in 2019 already. Another big source of inspiration is Software Carpentry[9] 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://rap4mads.eu/

[7]https://analysisfunction.civilservice.gov.uk/support/reproducible-analytical-pipelines/

[8]https://ukgovdatascience.github.io/rap_companion/

[9]https://software-carpentry.org/

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 Python programming language. As I've stated in the preface, I'm not a Python expert, but I'm comfortable enough with the language. In any case, this is not a book about programming itself, and bar a short discussion on the merits of the Polars package, I won't be teaching you programming.

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 you're done with the analysis, you're done. You don't want to spend time, which you often don't have anyways, to rewrite or refactor an analysis and make it reproducible after the fact. We both know that this is not going to happen. Once an analysis is done, it's time to go to the next analysis. And if you need to rerun an older analysis (for example, because the data got updated), then you'll simply figure it out at that point, right? That's a problem for future you, right? Hopefully, future you will remember every quirk of your 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… You better hope future you is a more diligent worker than you!

Going forward, I'm going to refer to a project that is reproducible as a "reproducible analytical pipeline", or RAP for short. There are only two ways to make such a RAP; either you are lucky enough to have someone on the team whose job is to turn your messy code into a RAP, or you do it yourself. And this second option is very likely the most common. The issue is, as stated above, that most of us 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 spent working on 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", but will be part of the analysis itself, and they will also help with managing the project. 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? Keeping the

script is already a good start of course. 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 Python programming language. This book will assume that you have been using Python 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 `plotnine` 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. Chapter 2, *Before we start* should help you gauge how easily you will be able to follow this book.

Ideally, you should also not be afraid of not using Graphical User Interfaces (GUIs). While you can follow along using an IDE like VS Code, I will not be teaching any features from any program with a GUI. This is not to make things harder than they should be (quite the contrary actually) but because interacting graphically with a program is simply not reproducible. So our aim 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. This is the second lesson of building RAPs: there should be no human intervention needed to get the outputs once the RAP is started. If you achieve this, 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 actually is reproducibility?

A reproducible project means that this project can be rerun by anyone at 0 (or very minimal) cost. But there are different levels of reproducibility, and I will discuss this in the next section.

Let's first discuss some requirements that a project must have to be considered a RAP.

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 just not reproducible. Why?

Because on a long enough time horizon, there is no way to re-execute your code with the exact same version of the proprietary programming language and on the exact same version of the operating system that was used at the time the project was developed. As I'm writing these lines, MATLAB, for example, is at version R2022b. And buying an older version may not be simple. I'm sure if you contact their sales department they might be able to sell you an older version. Maybe you can even simply re-download older versions that you've already bought from their website. But maybe it's not that simple. Or maybe

they won't offer this option anymore in the future, who knows? In any case, if you google "purchase old version of Matlab" you will see that many researchers and engineers have this need.

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

And if you're running old code written for version, say, R2008a, there's no guarantee that it will produce the exact same results on version 2022b. And let's not even mention the toolboxes (if you're not familiar with MATLAB's toolboxes, they're the equivalent of packages or libraries in other programming languages). These evolve as well, and there's no guarantee that you can purchase older versions of said toolboxes. And it's likely that

newer versions of toolboxes cannot even run on older versions of Matlab.

And let me be clear, what I'm describing here with MATLAB could also be said for any other proprietary programs still commonly (unfortunately) used in research and in statistics (like STATA, SAS or SPSS). 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 the editors of these tools 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 (or whatever) 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 outputs that your pipeline builds. 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[1], especially if you're working with very high precision floating point arithmetic like you would do in the financial sector for instance.

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);

---

[1]https://github.com/numpy/numpy/issues/9187

- 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 Python programming language);
- The project needs to be run on an open source operating system (thankfully, we can deal with this without having to install and learn to use a new operating system, thanks to Docker);
- Data and the paper/report need obviously to be accessible as well, if not publicly as is the case for research, then within your company. This means that the concept of "scripts and/or data available upon request" belongs in the trash.

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

1.5 Are there different types of reproducibility?

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

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

The *worst* reproducible pipeline would be something that works, but only on your machine. This can be simply due to the fact that you hardcoded paths that only exist on your laptop. Anyone wanting to rerun the pipeline would need to change the paths. This is something that needs to be documented in a README which we assumed was the case, so there's that. But maybe this pipeline only runs on your laptop because the computational environment that you're using is hard to reproduce. Maybe you use software, even if it's open source software, that is not easy to install (anyone that tried to install R packages on Linux that depend on the `{rJava}` package know what I'm talking about).

So a least worse pipeline would be one that could be run more easily on any similar machine to 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 Python, this could be as simple as providing a `requirements.txt` file that lists the dependencies of the project, and which could be easily install using `pip`:

```
pip install -r requirements.txt
```

Doing this ensures that others, or future you, will be able to install the required packages to reproduce a study. However, this is not enough, and I will be talking about `pipenv` to do this kind of thing instead of `pip`. In the next chapter I'll explain why.

You should also ensure that people run the same analysis on the same version of Python that was used to program it. Just installing the right packages is not enough. The same code can produce different results on different versions of Python, or not even work at all. If you've been using Python for some time, you certainly remember the switch from Python 2 to Python 3… and who knows, the switch to Python 4 might be just as painful!

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 and explicitly deal with versions of the language itself.

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

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

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

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

2 of this book, I will reintroduce the idea and call it the "reproducibility iceberg".

Reproducibility Spectrum

Publication +

| Publication only | Code | Code and data | Linked and executable code and data | Full replicati |

Not reproducible ⟵⟶ Gold stand
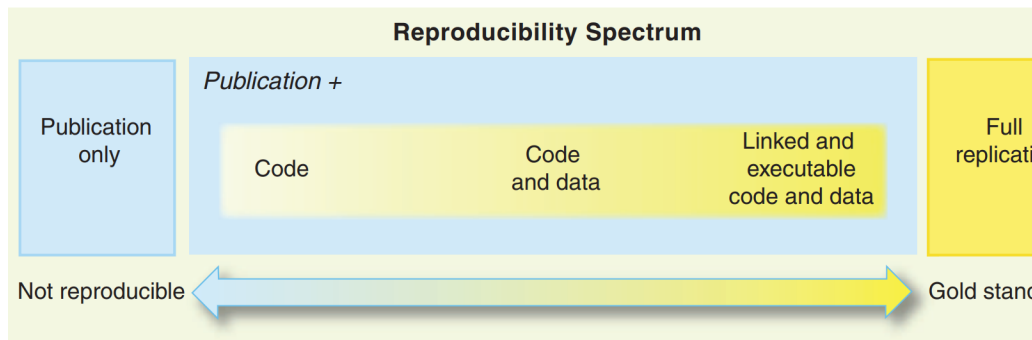
Figure 1.3: The reproducibility spectrum from Peng's 2011 paper.

Let me just finish this introduction by discussing the last item on the previous list: hardware architecture. You see, Apple has changed the hardware architecture of their computers recently. Their new computers don't use Intel based hardware anymore, but instead Apple's 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 run on their new computers (at least not without a compatibility layer). Which means that if you have a recent Apple Silicon Macbook and need to install old 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 Apple Silicon first. 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 older Python and package 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 build 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 from a business perspective. 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 (also, further locking in their users to their hardware). 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 and licensing 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!

2 Setting up a development environment

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

If you are already using Python, you are likely already familiar with setting up per-project development environments using tools like `pyenv`. 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 you Python. Remember, this book is about building reproducible analytical pipelines!

2.1 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 reason why you should install a Python version and manage it yourself, 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 environments, 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 are the two tools that I use for this, and that I think work quite well together: `micromamba` and `pipenv`.

The workflow is as follows:

- Install `micromamba`, a lightweight package manager.
- Using `micromamba`, create an environment that contains a Python interpreter and the `pipenv` package.
- Using this environment, install the required packages using `pipenv`.
- `pipenv` will automatically generate two very useful files, `Pipfile` and `Pipfile.lock`.

In the following sections I detail this process.

2.2 Creating a project-specific development environment

What we want is to have project-specific development environments that should include a specific Python version, specific versions of Python packages and all the code to actually run the project. If you do this, you have already achieved a great deal to make your analysis reproducible. Some would even argue that it is enough!

I'm going to assume that you don't have any Python version available on your computer and need to get one. Let's suppose that Python version 3.12.1 is the latest released version, and let's also suppose that you would like to use that version to start working on a project. To first get the right Python interpreter ready, you should install `micromamba`. Please refer to

the `micromamba` documentation here[1] but on Linux, macOS and Git Bash on Windows (there's also instructions for Powershell, if you don't have Git Bash installed on Windows), it should be as easy as running this in your terminal:

```
"${SHELL}" <(curl -L micro.mamba.pm/install.sh)
```

for Poweshell use instead `Invoke-Expression ((Invoke-WebRequest -Uri https://micro.mamba.pm/install.ps1).Content)` instead.

We can now use `micromamba` to create an environment that will contain a Python interpreter for our project and `pipenv`. Type the following command to create this environment:

```
micromamba create -n housing python=3.12.1 pipenv
```

This will create an environment named "pipenv_env" that includes `pipenv` and the version 3.12.1 of Python. If you're just starting a project, you can safely choose the very latest released version (check the releases pages on Python's website).

We can now use `pipenv` to install the packages for our project. But why don't we just use the more common `pip` instead of `pipenv`, or even `micromamba` which can also install any other Python package that we require for our projects? 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.

---

[1]https://mamba.readthedocs.io/en/latest/installation/micromamba-installation.html#automatic-install

"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? (If you don't know what a `requirements.txt` file is, you can think of it as a simple text file that lists the required packages and their versions for an analysis).

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…). This mismatch in the versions of `ciao` can cause issues. `pipenv` takes care of this for you by generating a so-called *lock* file automatically, 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.

What about using `micromamba`? `micromamba` could indeed be used to install the project's dependencies, but would require another tool called `conda-lock` to generate lock files, and in my

experience, using `conda-lock` doesn't always work. I have had 0 issues with `pipenv` on the other hand.

In any case, the point is that you should use a tool that specifies dependencies very strictly and precisely. Use whatever you're comfortable with if you already are familiar with one such tool. If not, and you want to follow along, use `pipenv` but take some time to check out other options. I personally use Nix, which is not specific to Python, but I decided not to discuss Nix in this book, because to properly discuss it, it would require a book on its own.

Now that `pipenv` is installed, let's start using it to install the packages we need for our project. Because the Python interpreter was installed using `micromamba`, we either need to activate the environment to get access to it, or we should use `micromamba run` to run the Python intpreter from this environment. First, create a folder called `housing`, which will contain our analysis scripts. Then, from that folder, run the following command:

```
micromamba run -n housing pipenv install
↪  polars==1.1.0 plotnine beautifulsoup4 pandas
↪  plotnine lxml pyarrow requests xlsx2csv
```

As you can see, I chose to install specific versions of `polars`. This is because I want you to follow along with the same versions as in the book. You could remove the `==x.y.z` string from the command above to install the latest versions of `polars` 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. You could also specify versions for the other packages if you wish. A little sidnote: some of these packages we are not really going to be using, but they're needed either as dependencies for

`polars` or because we need one single function from them. The
packages in question are `pandas`, `lxml` and `pyarrow`.

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

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

[packages]
polars = "==1.1.0"
plotnine = "*"
beautifulsoup4 = "*"
pandas = "*"
skimpy = "*"

[dev-packages]

[requires]
python_version = "3.12"
```

I think that this file is pretty self-evident: the packages being
used for this project are listed alongside their versions. The
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 one you want for your project. In this case,
I highly recommend you change the version to the right one.
Also, you'll notice that here the Python version is "3.12", but
we specified version "3.12.1" with `micromamba` when we created
the environment. I would recommend that you add the missing
".1" for maximum reproducibility. If you edited the `Pipfile` then

you need to run `pipenv lock` to regenerate the `Pipfile.lock` file as well:

```
micromamba run -n housing 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 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. These two files are very important, because they will make sure that it will be possible to regenerate the same environment on another machine.

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

```
micromamba run -r housing pipenv shell
```

and check that the right version of Python is being used:

```
python --version
```

This should print `Python 3.12.1` in the terminal. Start the Python interpreter and let's check `polars`'s version:

```python
python
```

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

```python
import polars as pl
pl.__version__
```

```
'1.12.0'
```

You should see `1.1.0` as the listed version. Quit the shell, and then quit the environment with `exit`.

2.3 One last thing

Before continuing, it would be nice if we would automatically drop into the environment each time we are in the right folder; so for example, each time we navigate to the `housing` folder for our project on housing, the `housing` environment starts. We can achieve this by using a tool called `direnv`.

I won't go into details to install `direnv`, simply consult the documentation.

The next step is to start a shell in your environment by running `micromamba run -n housing pipenv shell`. This will start a shell inside the `housing` environment by executing `pipenv shell`. Take note of the command that appears, in my case it's this: `. /home/b-rodrigues/.local/share/virtualenvs/py_housing-` (pay attention to the `.` character at the very beginning of this command!).

In the root folder of the `housing` project, create an empty text file and name it `.envrc`. Inside of that file, paste the line from before into the empty `.envrc` file. Finally, run `direnv allow` in a terminal in that folder, and each time you will navigate to this folder using a terminal, that development environment will be used. Many development interfaces can work together with `direnv`, refer to their documentation to learn how to configure your IDE to make use of `direnv`.

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

Ok, so to summarise, we installed `micromamba` which will make it easy to install any version of Python that we require, and we also installed `pipenv`, which we use to install packages. The advantage of using `pipenv` is that we get deterministic builds, and `pipenv` works well with `micromamba` 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 and then install packages by specyifing their versions, like so:

```
micromamba create -n project_name python=X.YY.Z
 ↪ pipenv
```

then, use this environment in a fresh folder to install the packages you need, for instance:

```
micromamba run -n housing pipenv install
beautifulsoup4==4.12.2 polars==1.1.0
plotnine==0.12.4
```

(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`. This is also the command that we will use later, in the chapter on CI/CD.

If you need to add packages to an environment, run: `micromamba run -n housing pipenv install great_tables` (or simply if `pipenv install great_tables` if you're already in the activated `housing` environment).

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. 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.

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.
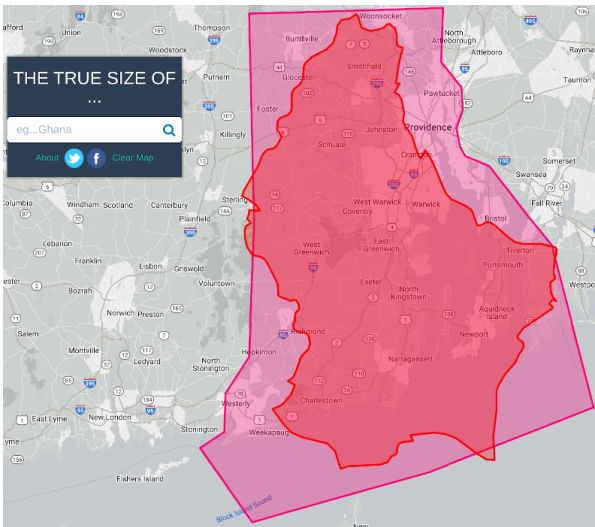

Figure 3.1: Luxembourg is about as big as the US State of Rhode Island.

What you should also know is that the population is about 672'050 people as of writing (July 2024), 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 (HPI) over time for Luxembourg and the European Union:
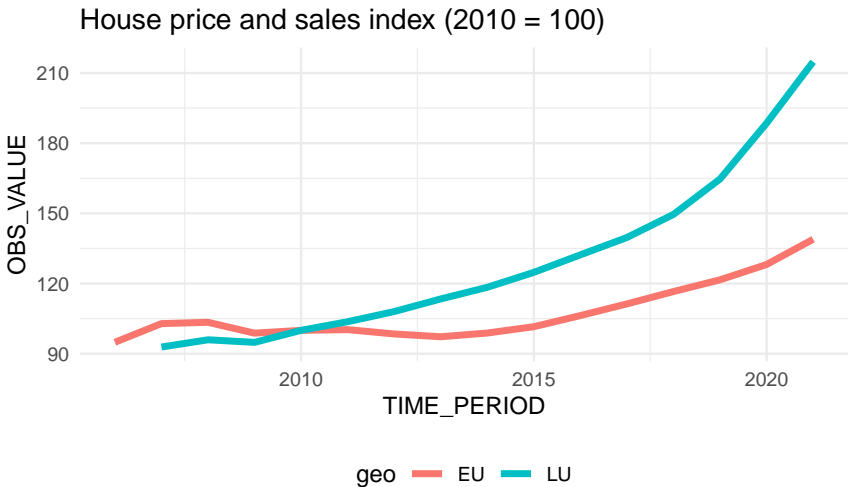
Source: Eurostat

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

Let us take a look at the definition of the HPI (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,

---

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

[2]https://archive.is/OrQwA, archived link for posterity.

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

---

[3]https://archive.org/download/note-29/note-29.pdf

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.

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 my 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:
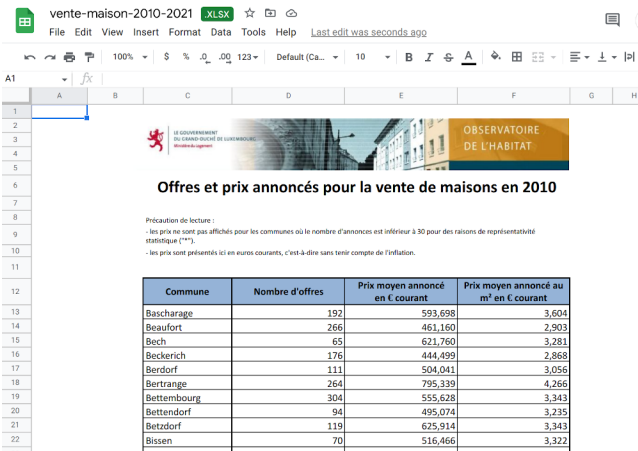
Figure 3.2: An Excel file meant for human eyes.

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 actually 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.

For ease of presentation, I'm going to show you each step of the analysis here separately, but I'll be putting everything together

---

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

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

in a single script once I'm done explaining each step. So first, let's import the main packages I'll be using for the analysis (I'll need to import some others as I go):

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

I will be using the **polars** package to manipulate data and **polars.selectors** contains handy functions to refer to columns while **re** is for regular expressions which I will need below.

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"

# Let's first download the file into a temporary
 ↪  file and then
# return its path
from tempfile import NamedTemporaryFile
from requests import get

response = get(url)

with NamedTemporaryFile(delete = False, suffix =
 ↪  '.xlsx') as temp_file:
    temp_file.write(response.content)
    temp_file_path = temp_file.name
```

220016

The code above downloads the Excel file, saves it in a tempo-
rary folder and returns the path to the file. We can now use
`temp_file_path` to read the data using `pl.read_excel()`.

Next, I will write a function to read all the sheets of the Excel
workbook. Ideally, this wouldn't be needed, because `polars`
can read all the sheets of an Excel workbook in one go and
return a list of sheets, but I want to add a column with the
year. So I write this function that reads one sheet and adds the
year column, and then I map this function over a list of sheet
names:

```python
def read_excel(excel_file, sheet):
    out = pl.read_excel(
            engine = 'xlsx2csv',
            source = excel_file,
            sheet_name = sheet,
            schema_overrides = {"Nombre d'offres":
  ↪  pl.String},
            read_options = {
            "skip_rows": 6,
            "has_header": True
            }

  ↪  ).with_columns(pl.lit(sheet).alias("year"))
    return out
```

I set some options in the `pl.read_excel()` function to correctly
read the file. I'll let you play around with these options to see
what happens if you don't set them.

I now need a little helper function that I will use to map over
the sheet names. This function sets the `excel_file` argument
of the previously defined `read_excel()` function we wrote to

`temp_file_path` which points to the Excel file. I will then map over its `sheet` argument.

```python
def wrap_read_excel(sheet):
    out = read_excel(excel_file = temp_file_path,
                     sheet = sheet)
    return out
```

Let's now create the list of sheet names:

```python
sheets = list(map(str, range(2010, 2022)))
```

I can now map the function over the list of sheets and concatenate them into a single polars data frame using `pl.concat()`:

```python
raw_data = pl.concat(list(map(wrap_read_excel,
 ↪   sheets)))
```

In the preface, I mentioned that the Python code in this book would have a very distinct R-like accent. Mapping over a list instead of writing a for-loop is an example of this. Feel free to use a for-loop if you're more comfortable with them!

The next function will be used below to clean the column names. For example, a column called something like `"House prices in €"` would get changed to `"house_prices_in_"`, removing the `"€"` sign, replacing spaces with `"_"` setting the string to lower case. If I was using `pandas`, I could have used `clean_columns()` from the `skimpy` package, but unfortunately this function doesn't work with `polars` data frames. So I wrote this little function to clean the column names instead, partly inspired by the `clean_names()` function from the `{janitor}` R package.

```python
def clean_names(string):
    # inspired by
    ↪ https://nadeauinnovations.com/post/2020/11/python-tr:
    clean_string = [s for s in string if s.isalnum()
↪ or s.isspace()]
    out = "".join(clean_string).lower()
    out = re.sub(r"\s+", "_", out)
    out = out.encode("ascii",
↪ "ignore").decode("utf-8")
    return out
```

We need to map `clean_names()` to each column. We can select all the columns using `pl.all()` and map the function to the column's `name` attribute.

```python
raw_data =
↪ raw_data.select(pl.all().name.map(clean_names))
```

Finally, let's rename some columns and do some more cleaning: - converting column types - ensuring consistent names for the communes (for example, sometimes the commune of Luxembourg is spelled "Luxembourg", sometimes "Luxembourg-Ville" - converting columns to their right types (but not the `n_offers` column yet…)

This is all done in the snippet below:

```python
raw_data = (
    raw_data
    .rename(
      {
        "commune": "locality",
```

```
        "nombre_doffres": "n_offers",
        "prix_moyen_annonc_en_courant":
          ↪  "average_price_nominal_euros",
        "prix_moyen_annonc_au_m_en_courant":
          ↪  "average_price_m2_nominal_euros"
    }
  )
  .with_columns(
    cs.all().str.strip_chars()
  )
  .with_columns(
    cs.contains("average").cast(pl.Float64, strict
  ↪  = False)
  )
  .with_columns(
    # In some sheets it's "Luxembourg", in others
      ↪  it's "Luxembourg-Ville"

  ↪  pl.col("locality").str.replace_all("Luxembourg.*",
  ↪  "Luxembourg")
  )
  .with_columns(
    # In some sheets it's "Pétange", in others
      ↪  it's "Petange"
    pl.col("locality").str.replace_all("P.*tange",
  ↪  "Pétange")
  )
)
```

In practice, it is unlikely that you would have written the above
code in one go. Instead you would have checked the data, done
something on it, then checked it again, etc. But as I mentioned

this is not a book about learning how to clean data, so let's just take this code as-is.

If you are familiar with the `{tidyverse}` (Wickham et al. 2019) family of packages from the R programming language, the above code should be quite relatively easy to follow. If you are more familiar with `pandas`, I believe that reading the code above should still be easy as well. What might be more difficult if you were brought up on Python exclusively is mapping functions to elements of lists instead of using for-loops. Don't worry about it too much, and use for-loops if you wish. Just remember that this book is not about Python per se, but about building reproducible pipelines. The focus will be on other things.

Running this code results in a tidy data set:

```
raw_data
```

```
shape: (1_313, 5)

  locality                n_offers
average_price_nominal_euro
average_price_m2_nominal_   year
  ---                     ---        s
  euros                     ---
  str                     str      ---
  ---                       str
                                    f64
  f64

  Bascharage              192        593698.31
  3603.57                   2010
  Beaufort                266        461160.29
  2902.76                   2010
```

Bech	65	621760.22	
3280.51			2010
Beckerich	176	444498.68	
2867.88			2010
Berdorf	111	504040.85	
3055.99			2010
...	...	...	
...	...		
Winseler	41	979696.17	
3921.98			2021
Wormeldange	50	1052340.8	
6391.339			2021
Moyenne nationale	null	1317473.9	
6743.722			2021
Total d'offres	11925	null	
null			2021
Source : Ministère du	null	null	
null			2021
Logement...			

But as you can see at the bottom, we still have some stuff that doesn't belong in the `locality` column. Let's also check for missing values in the `"average_price_nominal_euros"` column:

```
(
raw_data

↪   .filter(pl.col("average_price_nominal_euros").is_null())
)
```

shape: (186, 5)

```
 locality               n_offers
average_price_nominal_euro
average_price_m2_nominal_   year
 ---                    ---        s
 euros                     ---
 str                    str        ---
 ---                       str
                                   f64
 f64

 Consthum               29        null
 null                      2010
 Esch-sur-Sûre          7         null
 null                      2010
 Heiderscheid           29        null
 null                      2010
 Hoscheid               26        null
 null                      2010
 Saeul                  14        null
 null                      2010
 …                      …         …
 …                      …
 Waldbredimus           25        null
 null                      2021
 Weiler-la-Tour         28        null
 null                      2021
 Weiswampach            28        null
 null                      2021
 Total d'offres         11925     null
 null                      2021
 Source : Ministère du  null      null
 null                      2021
```

```
Logement…
```

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 3.3: 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` would be 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(~pl.col("locality").str.contains("Source"))
)
```

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

```
commune_level_data = (
    raw_data

↪    .filter(~pl.col("locality").str.contains("nationale|offre
    .filter(pl.col("locality").is_not_null())
    # This is needed on Windows...
    .with_columns(
        pl.col("locality").str.replace_all("\351",
↪   "é")
    )
    .with_columns(
        pl.col("locality").str.replace_all("\373",
↪   "û")
    )
    .with_columns(
        pl.col("locality").str.replace_all("\344",
↪   "ä")
    )
)
```

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

```
country_level = (
    raw_data

↪   .filter(pl.col("locality").str.contains("nationale"))
    .select(cs.exclude("n_offers"))
)

offers_country = (
    raw_data
    .filter(pl.col("locality").str.contains("Total
↪   d.offres"))
    .select(["year", "n_offers"])
)

country_level_data = (
    country_level.join(offers_country, on = "year")
    .with_columns(pl.lit("Grand-Duchy of
↪   Luxembourg").alias("locality"))
)
```

Let's take a look at it:

```
country_level_data
```

shape: (12, 5)

locality	average_price_nominal_euros	average_price_m2_nominal_euro	year	n_offers
---	---			s
---	---			
str	f64			
---			str	str

```
f64
```

```
  Grand-Duchy of    569216.0
3251.0                                 2010    19278
  Luxembourg
```

```
  Grand-Duchy of    597784.711711
3375.088978                            2011    21253
  Luxembourg
```

```
  Grand-Duchy of    596347.972382
3408.711732                            2012    14773
  Luxembourg
```

```
  Grand-Duchy of    644884.979694
3589.790108                            2013    13298
  Luxembourg
```

```
  Grand-Duchy of    663639.733049
3647.123608                            2014    9852
  Luxembourg
```

```
  …                 …                                           …
  …         …
  Grand-Duchy of    812919.0678
4274.039466                            2017    16055
  Luxembourg
```

```
  Grand-Duchy of    874685.96
4562.288                               2018    13534
  Luxembourg
```

```
  Grand-Duchy of   970589.13
5038.614                        2019    12171
  Luxembourg


  Grand-Duchy of   1180466.6
6364.361                        2020    11757
  Luxembourg


  Grand-Duchy of   1317473.9
6743.722                        2021    11925
  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 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. Let's first load the required packages:

---

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

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
from pandas import read_html
from io import StringIO
```

and let's get the raw data:

```
url = 'https://b-rodrigues.github.io/list_communes/'

html = urlopen(url)

tables = (
    BeautifulSoup(html, 'html.parser')
    .find_all("table")
)

current_communes_raw =
↪   read_html(StringIO(str(tables[1])))[0]
```

I won't go into much details, but the using `Beautifoulsoup()` it is possible to parse the `html` from the web page and get the tables out using the `.find_all()` method. The first table from that list is the one we're interested in, and using the `read_html()` function from the `pandas` package we can get that table into a data frame (turns out we can't completely avoid using `pandas`!).

We can now use `polars` to clean the table:

```
# current_communes has a MultiIndex, so drop it
current_communes_raw.columns =
↪   current_communes_raw.columns.droplevel()
```

```
current_communes_pl = (
    pl.DataFrame(current_communes_raw)
    .select(pl.col("Name.1").alias("commune"))
    .with_columns(
        pl.col("commune").str.replace_all("\351",
↪   "é")
    )
    .with_columns(
        pl.col("commune").str.replace_all("\373",
↪   "û")
    )
    .with_columns(
        pl.col("commune").str.replace_all("\344",
↪   "ä")
    )
    .with_columns(
    # This removes the dagger symbol next to certain
    ↪   communes names
    # in other words it turns "Commune †" into
    ↪   "Commune".
        pl.col("commune").str.replace_all(" .$", "")
    )
)
```

Finally, we can save the communes into a simple list:

```
current_communes =
↪   list(current_communes_pl["commune"])
```

Let's see if we have all the communes in our data, if the code below results in an empty list, then we're good:

```
(
commune_level_data

 ↪  .filter(~pl.col("locality").is_in(current_communes))
 .get_column("locality")
 .unique()
 .sort()
 .to_list()
)
```

```
['Bascharage', 'Boevange-sur-Attert', 'Burmerange',
'Clémency', 'Commune', 'Consthum', 'Ermsdorf',
'Erpeldange', 'Eschweiler', 'Heiderscheid',
'Heinerscheid', 'Hobscheid', 'Hoscheid', 'Hosingen',
'Kaerjeng', 'Luxembourg', 'Medernach', 'Mompach',
'Munshausen', 'Neunhausen', 'Rosport',
'Septfontaines', 'Tuntange', 'Wellenstein']
```

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:

```
# Need to also check former communes
url =
 ↪  'https://b-rodrigues.github.io/former_communes/#Former_co
```

```
html = urlopen(url)

tables = (
    BeautifulSoup(html, 'html.parser')
    .find_all("table")
)

# The third table (...hence the '2' in tables[2]...)
 ↪  is the one we need
former_communes_raw =
 ↪  read_html(StringIO(str(tables[2])))[0]

former_communes_pl = (
    pl.DataFrame(former_communes_raw)
    .with_columns(
        pl.col("Name").str.replace_all("\351", "é")
    )
    .with_columns(
        pl.col("Name").str.replace_all("\373", "û")
    )
    .with_columns(
        pl.col("Name").str.replace_all("\344", "ä")
    )
    .select(pl.col("Name").alias("commune"))
)

former_communes_pl
```

```
shape: (40, 1)

  commune
  ---
```

```
str

Arsdorf
Asselborn
Bascharage
Bastendorf
Bigonville
…
Rosport
Septfontaines
Tuntange
Wellenstein
Wilwerwiltz
```

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:

```
# Combine former and current communes

communes = (
    pl.concat([former_communes_pl,
 ↪  current_communes_pl])
    .get_column("commune")
    .unique()
    .sort()
    .to_list()
)


(
commune_level_data
    .filter(~pl.col("locality").is_in(communes))
```

```
    .get_column("locality")
    .unique()
    .sort()
    .to_list()
)
```

```
['Clémency', 'Commune', 'Erpeldange', 'Kaerjeng',
'Luxembourg']
```

And now we can harmonize the spelling:

```
# There's certain communes with different spelling
  ↪  between
# wikipedia and our data, so let's correct the
  ↪  spelling
# on the wikipedia ones
# ['Clémency', 'Erpeldange', 'Kaerjeng',
  ↪  'Luxembourg', 'Pétange']

communes_clean = (
    pl.concat([former_communes_pl,
  ↪  current_communes_pl])
    .with_columns(

  ↪  pl.when(pl.col("commune").str.contains("Cl.mency"))
          .then(pl.lit("Clémency"))

  ↪  .otherwise(pl.col("commune")).alias("commune")
    )
    .with_columns(

  ↪  pl.when(pl.col("commune").str.contains("Erpeldange"))
          .then(pl.lit("Erpeldange"))
```

```
↪    .otherwise(pl.col("commune")).alias("commune")
     )
     .with_columns(

↪    pl.when(pl.col("commune").str.contains("City"))
          .then(pl.lit("Luxembourg"))

↪    .otherwise(pl.col("commune")).alias("commune")
     )
     .with_columns(

↪    pl.when(pl.col("commune").str.contains("K.*jeng"))
          .then(pl.lit("Kaerjeng"))

↪    .otherwise(pl.col("commune")).alias("commune")
     )
     .with_columns(

↪    pl.when(pl.col("commune").str.contains("P.*tange"))
          .then(pl.lit("Pétange"))

↪    .otherwise(pl.col("commune")).alias("commune")
     )
     .get_column("commune")
     .unique()
     .sort()
     .to_list()
)
```

Let's run our test again:

```
(
commune_level_data

  ↳   .filter(~pl.col("locality").is_in(communes_clean))
      .get_column("locality")
      .unique()
      .sort()
      .to_list()
)
```

```
['Commune']
```

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 Python 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 Laspeyeres price index. This price

---

[8]https://is.gd/bGvNKG

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 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:

- 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.

---

[9]https://is.gd/7MqLjX
[10]https://en.wikipedia.org/wiki/Schengen_Area

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 Python 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 Python, and this is still fresh in our brains. Should someone less familiar with Python 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? Ideally, the same versions of the packages you used and the same version of Python should get used by that person to rerun the analysis. Thankfully, you can share the `Pipfile` and the `Pipfile.lock` files from the previous chapters to make it easier for them, so by setting up our development environment, we also made it easier for future you or replicators to install the required dependencies!

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 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.

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 code even inside the same project. Look at the last part of

the `analysis.py` 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.

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. Programming languages such as Python evolve quickly, and packages even more so. 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.

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.

References

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27.

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.