

Building Reproducible Analytical Pipelines

**Master of Data Science, University of
Luxembourg - 2025**

Bruno Rodrigues

2025-09-15

Table of contents

Introduction	1
Schedule	1
Reproducible analytical pipelines?	2
Data products?	2
Machine learning?	3
What actually is reproducibility?	4
The requirements of a RAP	4
Large Language Models	5
Why R? Why not [insert your favourite programming language]	6
Nix	8
Pre-requisites	10
Grading	10
Jargon	11
R	11
Python	12
Nix	13
Further reading	14
License	15
1 Reproducibility with Nix	17
1.1 Learning Outcomes	17
1.2 Why Reproducibility? Why Nix? (1h30)	18
1.2.1 Motivation: Reproducibility in Scientific and Data Workflows	18
1.2.2 Problems with Ad-Hoc Tools	18

Table of contents

1.2.3	Nix, a declarative package manager	19
1.2.4	The <code>rix</code> package	19
1.2.5	Installing Nix	20
1.2.6	Temporary shells	23
1.3	Session 1.2 – Dev Environments with Nix (<i>1h30</i>)	24
1.3.1	Some Nix concepts	24
1.3.2	Derivations	25
1.3.3	Using <code>{rix}</code> to generate development environments	27
1.3.4	Using <code>nix-shell</code> to Launch Environments	29
1.3.5	Pinning with <code>nixpkgs</code>	29
1.3.6	Installing Python packages not available via <code>nixpkgs</code> (impure)	30
1.4	Configuring your IDE	33
1.4.1	Pre-requisites	33
1.4.2	<code>direnv</code>	34
1.4.3	In summary and next steps	36
1.5	Starting a new project:	37
1.6	Hands-On Exercises	39
2	Version Control with Git	41
2.1	Introduction	42
2.2	Installing Git	43
2.3	Setting up a repo	44
2.4	Cloning the repository onto your computer	48
2.5	Setting up SSH authentication	49
2.6	Your first commit	51
2.7	Understanding Git workflow commands	53
2.8	Working with commit history	54
2.9	Collaborating and handling conflicts	55
2.9.1	Strategy 1: Merging (The Default)	56
2.9.2	Strategy 2: Rebasing (The Cleaner Way)	57
2.10	Working with branches	60
2.11	Advanced workflow with branches	61

Table of contents

2.12	Essential daily workflow	62
2.13	A Better Way to Collaborate: Trunk-Based Development	63
2.13.1	How to Work with Short-Lived Branches .	64
2.14	Contributing to someone else's repository	66
2.15	Working with LLMs and Git: Managing AI-Generated Changes	68
2.15.1	The LLM workflow with Git	68
2.15.2	Examining LLM changes	69
2.15.3	Interactive staging: Accepting changes chunk by chunk	69
2.15.4	Example: Reviewing LLM changes to an R script	70
2.15.5	Advanced chunk management	72
2.15.6	Creating meaningful commits after LLM review	72
2.15.7	Working with multiple files modified by LLM	73
2.15.8	Handling LLM-generated new files	74
2.15.9	Using Git to compare LLM suggestions .	74
2.15.10	Best practices for LLM + Git workflow .	75
2.15.11	Example complete workflow	76
2.16	Hands-on Exercises	77
2.16.1	Exercise 1: Setup and First Commit (skip if you followed along)	77
3	Functional Programming	85
3.1	Introduction: From Scripts to Functions	86
3.1.1	Why Does This Matter for Data Science?	89
3.2	Purity and Side Effects	90
3.2.1	Handling "Impure" Operations like Randomness	91
3.2.2	Can We Make This Truly Pure?	93
3.3	Writing Your Own Functions	97

Table of contents

3.4	The Functional Toolkit: Map, Filter, and Reduce	100
3.4.1	1. Mapping: Applying a Function to Each Element	100
3.4.2	2. Filtering: Keeping Elements That Match a Condition	102
3.4.3	3. Reducing: Combining All Elements into a Single Value	104
3.5	The Power of Composition	106
3.5.1	The Challenge of Composition in Python	107
3.6	Conclusion: Functions as the Bedrock of Reproducibility	110
3.7	Exercises	111
3.7.1	1. From Impure to Pure	112
3.7.2	2. Mapping	113
4	Unit Testing: The Safety Net for Your Code	117
4.1	Introduction: Proving Your Code Works	118
4.2	The Philosophy of a Good Unit Test	119
4.3	Unit Testing in Practice	119
4.3.1	Testing in R with <code>{testthat}</code>	120
4.3.2	Testing in Python with <code>pytest</code>	122
4.4	Testing as a Design Tool	125
4.5	The Modern Data Scientist's Role: Reviewer and AI Collaborator	127
4.5.1	Using LLMs to Write Tests	127
4.5.2	Testing and Code Review	128
4.5.3	A Note on Packaging and Project Structure	128
4.5.4	Hands-On Exercises	129
5	A Short Intro to Packaging Your Code in R and Python	137
5.1	Introduction: Why Bother Packaging?	138

5.2	Part 1: Creating an R Package with <code>{usethis}</code> and <code>{devtools}</code>	139
5.2.1	Step 1: Project Setup	140
5.2.2	Step 2: Write and Document a Function .	141
5.2.3	Step 3: Add Unit Tests	142
5.2.4	Step 4: Check and Install	143
5.2.5	Step 5: Install from GitHub	144
5.2.6	Step 5bis: Install it locally	145
5.3	Part 2: Creating a Minimal Python Package with <code>uv</code>	146
5.3.1	Step 1: Project Setup with <code>uv</code>	146
5.3.2	Step 2: Write a Function and Declare Dependencies	148
5.3.3	Step 3: Add Unit Tests	150
5.3.4	Step 4: Build and Install	150
5.3.5	Step 5: Install from GitHub	152
5.4	Conclusion: The Packaging Mindset in the Age of AI	153
5.5	Hands-on Exercises	156
5.5.1	Exercise 1: Build Your First Python Package with Nix	156
5.5.2	Exercise 2: Build and Package an R Tool with Nix	158
6	Building Reproducible Pipelines with Nix and <code>{trixpress}</code>	161
6.1	Introduction: From Scripts and Notebooks to Pipelines	162
6.2	Our First Polyglot Pipeline	165
6.2.1	Step 0: Use Git	166
6.2.2	Step 1: Defining the Environment	166
6.2.3	Step 2: Defining the Pipeline	167
6.2.4	Step 3: Building and Inspecting the Pipeline	172

Table of contents

6.3	Caching	173
6.4	Debugging and Working with Build Logs	174
6.5	Running Someone Else’s Pipeline: The Ultimate Test of Reproducibility	176
6.6	Hands-On Exercises	177
7	Containerization With Docker	179
7.1	Introduction	180
7.2	Docker essentials	182
7.2.1	Installing Docker	182
7.2.2	The Rocker Project and image registries .	183
7.2.3	Basic Docker workflow	186
7.2.4	Making our own images	188
7.2.5	Publishing images on Docker Hub	195
7.2.6	Sharing a compressed archive of your image	198
7.2.7	What if you don’t use Nix?	200
7.3	Dockerizing a <code>{rixpress}</code> Pipeline	202
7.3.1	Step 1: The Dockerfile	203
7.3.2	Step 2: The Export Script	205
7.3.3	Step 3: Build and Run	207
7.4	Further reading	208
7.5	Hands-on Exercises	208
7.5.1	Level 1: Foundational Skills	208
7.5.2	Level 2: Building and Distributing Images	210
7.5.3	Level 3: The Capstone Project	211
8	Intro to CI/CD with Github Actions	215
8.1	Introduction	216
8.2	Getting your repo ready for Github Actions . . .	217
8.3	Building a Docker image and pushing it to a registry	222
8.4	Running a pipeline straight from Github Actions	223
8.5	Running unit tests on Github Actions	224
8.6	Further reading	226

Table of contents

9	What else should you learn?	227
9.1	Touch typing	228
9.2	Vim	228
9.3	Statistical modeling	229
10	Conclusion	231
10.1	Why bother	231

Introduction

This is the 2025 edition of the course. If you're looking for the 2024 edition, you can [click here](#)

What's new:

- Focus on Nix as the canonical tool for reproducibility and build automation
- Integration of LLMs as an additional tool in the reproducers toolbox.

*This course is based on my book titled *Building Reproducible Analytical Pipelines with R*. This course focuses only on certain aspects that are discussed in greater detail in the book.*

Schedule

- 2025/09/15 - 4 hours, Intro (1 hour) and Nix (3 hours)
- 2025/09/22 - 4 hours, Git (4 hours)
- 2025/10/06 - 4 hours, Functional programming (1 hour), Unit testing (3 hours)
- 2025/10/13 - 4 hours, Packaging
- 2025/10/15 - 4 hours, Pipelines using Nix and rixpress
- 2025/10/20 - 4 hours, Docker
- 2025/10/21 - 4 hours, GitHub Actions

Reproducible analytical pipelines?

This course is my take on setting up code that results in some *data product*. This code has to be reproducible, documented and production ready. Not my original idea, but introduced by the UK's Analysis Function.

The basic idea of a reproducible analytical pipeline (RAP) is to have code that always produces the same result when run, whatever this result might be. This is obviously crucial in research and science, but this is also the case in businesses that deal with data science/data-driven decision making etc.

A well documented RAP avoids a lot of headache and is usually re-usable for other projects as well.

Data products?

In this course each of you will develop a *data product*. A data product is anything that requires data as an input. This can be a very simple report in PDF or Word format or a complex web app. This website is actually also a data product, which I made using the R programming language and Quarto. Dependencies are managed by the Nix package manager and the build runs on GitHub Actions, and the website you're seeing his hosted on GitHub Pages. By the end of the course, you'll have all the basic knowledge to achieve something similar.

The focus on the course will not be about the end product itself, which you will have to choose for your project, but instead we will focus on how to set up a pipeline that results in these data products in a reproducible way.

Machine learning?

No, being a master in machine learning is not enough to become a data scientist. Actually, the older I get, the more I think that machine learning is almost optional. What is not optional is knowing how:

- to write, test, and properly document code;
- to acquire (reading in data can be tricky!) and clean data;
- to work inside the Linux terminal/command line interface;
- to use Git, Docker for Dev(Git)Ops;
- the Internet works (what's a firewall? what's a reverse proxy? what's a domain name? etc, etc...);

But what about machine learning? Well, depending what you'll end up doing, you might indeed focus a lot on machine learning and/or statistical modeling. That being said, in practice, it is very often much more efficient to let some automl algorithm figure out the best hyperparameters of a XGBoost model and simply use that, at least as a starting point (but good luck improving upon automl...). What matters, is that the data you're feeding to your model is clean, that your analysis is sensible, and most importantly, that it could be understood by someone taking over (imagine you get sick) and rerun with minimal effort in the future. The model here should simply be a piece that could be replaced by another model without much impact. The model is rarely central... but of course there are exceptions to this, especially in research, but every other point I've made still stands. It's just that not only do you have to care about your model a lot, you also have to care about everything else.

So in this course we're going to learn a bit of all of this. We're going to learn how to write reusable code, learn some basics of the Linux command line, Nix, Git and Docker.

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.

The requirements of a RAP

For something to be truly reproducible, it has to respect the following bullet points:

- Source code must obviously be available and thoroughly tested and documented (which is why we will be using Git and GitHub);
- All the dependencies must be easy to find and install (we are going to deal with this using Nix);
- 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);
- 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.

Also, 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*.

Large Language Models

LLMs have rapidly become an essential powertool in the data scientist’s toolbox. But as with any powertool, beginners risk cutting their fingers if they’re not careful. So it is important to learn how to use them. This course will give you some pointers on how to integrate LLMs into your workflow. All of the exercises can and should be tackled using LLMs.

Why R? Why not [insert your favourite programming language]

R is a domain-specific language whose domain is statistics, data analysis/science and machine learning, and as such has many built-in facilities to make handling data very efficient.

If you learn R you have access to almost 25'000 packages (as of June 2025, including both CRAN and Bioconductor packages) to:

- clean data (see: `{dplyr}`, `{tidyverse}`, `{data.table}`...);
- work with medium and big data (see: `{arrow}`, `{sparklyr}`...);
- visualize data (see: `{ggplot2}`, `{plotly}`, `{echarts4r}`...);
- do literate programming (using Rmarkdown or Quarto, you can write books, documents even create a website);
- do functional programming (see: `{purrr}`...);
- call other languages from R (see: `{reticulate}` to call Python from R);
- do machine learning and AI (see: `{tidymodels}`, `{tensorflow}`, `{keras}`...)
- create webapps (see: `{shiny}`...)
- domain specific statistics/machine learning (see CRAN Task Views for an exhaustive list);
- and more

It's not just about what the packages provide: installing R and its packages and dependencies is rarely frustrating, which is not the case with Python (Python 2 vs Python 3, `pip` vs `conda`, `pyenv` vs `venv` vs `uv`, ..., dependency hell is a real place full of snakes)

Why R? Why not [insert your favourite programming language]



The reason this is the case is that anyone can push anything on to Pypi, and no package gets checked against its dependencies or reverse dependencies. That is not the case for R, where published packages need to declare their dependencies and can't break any of their reverse dependencies (when this happens, authors of reverse dependencies get two weeks to fix their packages or they get removed from CRAN).

Furthermore, and this is surprising to many people, R offers a much better package developing experience than Python.

That doesn't mean that R does not have any issues. Quite the contrary, R sometimes behaves in seemingly truly bizarre ways (as an example, try running `nchar("1000000000")` and then `nchar(1000000000)` and try to make sense of it). To know more about such bizarre behaviour, I recommend you read *The R Inferno* (linked at the end of this chapter). So, yes, R is far

Introduction

from perfect, but it sucks less than the alternatives (again, in my absolutely objective opinion).

```
nchar("1000000000")
```

That being said, Python remains extremely popular, and it is likely that you will continue writing Python. In my opinion, the future of data science is going to be more and more polyglot. Data products are evermore complex, and require being built using many languages; so ideally we would like to find a way to use whatever tool is best fit for the job at hand. Sometimes it can be R, sometimes Python, sometimes shell scripts, or any other language. This is where Nix will help us.

Nix

Nix is a package manager for Linux distributions, macOS and it even works on Windows if you enable WSL2. What's a package manager? If you're not a Linux user, you may not be aware. Let me explain it this way: in R, if you want to install a package to provide some functionality not included with a vanilla installation of R, you'd run this:

```
install.packages("dplyr")
```

It turns out that Linux distributions, like Ubuntu for example, work in a similar way, but for software that you'd usually install using an installer (at least on Windows). For example you could install Firefox on Ubuntu using:

```
sudo apt-get install firefox
```

(there's also graphical interfaces that make this process "more user-friendly"). In Linux jargon, `packages` are simply what we call software (or I guess it's all "apps" these days). These packages get downloaded from so-called repositories (think of CRAN, the repository of R packages, or Pypi, in the case of Python) but for any type of software that you might need to make your computer work: web browsers, office suites, multimedia software and so on.

So Nix is just another package manager that you can use to install software.

But what interests us is not using Nix to install Firefox, but instead to install R, Python and the R and Python packages that we require for our analysis. But why use Nix instead of the usual ways to install software on our operating systems?

The first thing that you should know is that Nix's repository, `nixpkgs`, is huge. Humongously huge. As I'm writing these lines, there's more than 120'000 pieces of software available, and the *entirety of CRAN and Bioconductor* is also available through `nixpkgs`. So instead of installing R as you usually do and then use `install.packages()` to install packages, you could use Nix to handle everything. But still, why use Nix at all?

Nix has an interesting feature: using Nix, it is possible to install software in (relatively) isolated environments. So using Nix, you can install as many versions of R and R packages that you need. Suppose that you start working on a new project. As you start the project, with Nix, you would install a project-specific version of R and R packages that you would only use for that particular project. If you switch projects, you'd switch versions of R and R packages.

Pre-requisites

I will assume basic programming knowledge, and not much more. Ideally you'll be following this course from a Linux machine, but if you're macOS, that's fine as well. On Windows, you will have to set up WSL2 to follow along.

Grading

The way grading works in this course is as follows: during lecture hours you will follow along. At home, you'll be working on setting up your own pipeline. For this, choose a dataset that ideally would need some cleaning and/or tweaking to be usable. If time allows, I'll leave some time during lecture hours for you to work on it and ask me and your colleagues for help. At the end of the semester, I will need to download your code and get it running. The less effort this takes me, the better your score. Here is a tentative breakdown:

- Code is on github.com and the repository is documented with a `Readme.md` file: 5 points;
- Data and functions to run pipeline are documented and tested: 5 points;
- Every software dependency is easily installed: 5 points;
- Pipeline can be executed in one command: 5 points.

The way to fail this class is to write an undocumented script that only runs on your machine and expect me to debug it to get it to run, there is no excuse for this, especially in the age of LLMs.

At the end of each chapter, there are exercises that should help you solidify your knowledge. These are not graded, and there purely for you to learn more.

Jargon

There's some jargon that is helpful to know when working with R, Python, or Nix.

Here's a non-exhaustive list to get you started:

R

- **CRAN**: the Comprehensive R Archive Network. This is a curated online repository of packages and R installers. When you type `install.packages("package_name")` in an R console, the package gets downloaded from there;
- **Library**: the collection of R packages installed on your machine;
- **R console**: the program where the R interpreter runs;
- **Posit/RStudio**: Posit (named RStudio in the past) are the makers of the RStudio IDE and of the *tidyverse* collection of packages;
- **tidyverse**: a collection of packages created by Posit that offer a common language and syntax to perform any task required for data science — from reading in data, to cleaning data, up to machine learning and visualisation;

Introduction

- **base R:** refers to a vanilla installation (and vanilla capabilities) of R. Often used to contrast a *tidyverse* specific approach to a problem (for example, using base R's `lapply()` in contrast to the *tidyverse* `purrr::map()`);
- **package::function():** functions can be accessed in several ways in R, either by loading an entire package at the start of a script with `library(dplyr)` or by using `dplyr::select()`;
- **Function factory (sometimes adverb):** a function that returns a function;
- **Variable:** the variable of a function (as in `x` in `f(x)`) or the variable from statistical modeling (synonym of feature);
- **<- vs =:** in practice, you can use `<-` and `=` interchangeably. I prefer `<-`, but feel free to use `=` if you wish.

Python

- **PyPI:** the Python Package Index. Similar to CRAN for R, it is the default online repository of Python packages, used when you run `pip install package_name`;
- **Virtual environment (venv):** an isolated environment containing its own Python interpreter and packages, useful for keeping dependencies separate between projects;
- **Conda:** an alternative package and environment manager to `pip/venv`, popular in data science;

- **Jupyter notebook:** an interactive environment for running Python code in cells, mixing code, plots, and prose;
- **PEP:** Python Enhancement Proposal. Design documents that describe new features or conventions (for example, PEP 8 is the style guide for Python code);
- **Decorator:** a function that takes another function and extends or modifies its behavior (for example, `@staticmethod`);
- **List comprehension:** a compact syntax for creating lists, such as `[x * x for x in range(10)]`;
- **`__init__.py`:** a special file that marks a directory as a Python package;
- **`__main__`:** the entry point of a Python program (`if __name__ == "__main__":`).

Nix

- **Nixpkgs:** the main collection of Nix packages. Most software you install via Nix comes from here (CRAN and Bioconductor are mirrored, but for Python, only individual packages get packaged for Nix);
- **Derivation:** the low-level build instruction in Nix. Every package in Nix is ultimately a derivation;
- **Store:** the `/nix/store/` directory, where all Nix-built packages and their dependencies live, each in its own

Introduction

hash-prefixed path;

- **Expression:** Nix code describing how to build something (usually written in `.nix` files);
- **Flake:** a standardized way to package and distribute Nix code with reproducible inputs and outputs. We will not study flakes in this course, as they are still marked as *experimental*;
- **Shell:** a reproducible development environment created by Nix;
- **GC (garbage collection):** Nix can remove unused packages from the store;
- **Overlay:** a mechanism to extend or override packages in Nixpkgs. This out of the scope of this course;
- **Hydra:** the continuous integration service often used with Nix to build packages at scale;
- **Pinning:** fixing your Nixpkgs (or other inputs) to a specific commit to ensure reproducibility.

Further reading

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

- How Nix Works

License

This course is licensed under the WTFPL.

1 Reproducibility with Nix



1.1 Learning Outcomes

By the end of this chapter, you will:

- Understand the need for environment reproducibility in modern workflows
- Use `{frix}` to generate `default.nix` files

1 Reproducibility with Nix

- Build cross-language environments for data work or software development

1.2 Why Reproducibility? Why Nix? (1h30)

1.2.1 Motivation: Reproducibility in Scientific and Data Workflows

To ensure that a project is reproducible you need to deal with at least four things:

- Make sure that the required/correct version of R (or any other language) is installed;
- Make sure that the required versions of packages are installed;
- Make sure that system dependencies are installed (for example, you'd need a working Java installation to install the `{rJava}` R package on Linux);
- Make sure that you can install all of this for the hardware you have on hand.

But in practice, one or most of these bullet points are missing from projects. The goal of this course is to learn how to fulfill all the requirements to build reproducible projects.

1.2.2 Problems with Ad-Hoc Tools

Tools like Python's `venv` or R's `renv` only deal with some pieces of the reproducibility puzzle. Often, they assume an underlying OS, do not capture system-level dependencies (like `libxml2`,

1.2 Why Reproducibility? Why Nix? (1h30)

`pandoc`, or `curl`), and require users to “rebuild” their environments from partial metadata. Docker helps but introduces overhead, security challenges, and complexity, and just adding it to your project doesn’t make it reproducible if you don’t explicitly take some precautionary steps.

Traditional approaches fail to capture the entire dependency graph of a project in a deterministic way. This leads to “it works on my machine” syndromes, onboarding delays, and subtle bugs.

1.2.3 Nix, a declarative package manager

Nix is a tool for reproducible builds and development environments, often introduced as a package manager. It captures complete dependency trees, from your programming language interpreter to every system-level library you rely on. With Nix, environments are not recreated from documentation, but rebuilt precisely from code.

Nix can be installed on Linux distributions, macOS and it even works on Windows if you enable WSL2. In this course, we will use Nix mostly as a package manager (but towards also as a build automation tool).

However Nix has quite a steep learning curve, so this is why for the purposes of this course we are going to use an R package called `{rix}` to set up reproducible environments.

1.2.4 The `rix` package

`{rix}` is an R package (I’m the author) and its goal is to make writing Nix expressions easy. With `{rix}` you can declare the

1 Reproducibility with Nix

environment you need using the provided `rix()` function, which is the package’s main function. Calling it generates a file called `default.nix` which is then used by the Nix package manager to build that environment. Ideally, you would set up such an environment for each of your projects. You can then use this environment to either work interactively, or run R or Python scripts. It is possible to have as many environments as projects, and software that is common to environments will simply be re-used and not get re-installed to save space. Environments are isolated from each other, but can still interact with your system’s files, unlike with Docker where a volume must be mounted. While this is useful, it can sometimes lead to issues. For example, if you already have R installed, and a user library of R packages, more caution is required to properly use environments managed by Nix.

You don’t need to have R installed or be an R user to use `{rix}`. If you have Nix installed on your system, it is possible to “drop” into a temporary environment with R and `{rix}` available and generate the required Nix expression from there.

But first, let’s install Nix and try to use temporary shells.

1.2.5 Installing Nix

1.2.5.1 For Windows users only: some prerequisites

If you are on Windows, you need the Windows Subsystem for Linux 2 (WSL2) to run Nix. If you are on a recent version of Windows 10 or 11, you can simply run this as an administrator in PowerShell:

1.2 Why Reproducibility? Why Nix? (1h30)

```
wsl --install
```

You can find further installation notes at this official MS documentation.

I recommend to activate `systemd` in Ubuntu WSL2, mainly because this supports other users than `root` running Nix. To set this up, please do as outlined this official Ubuntu blog entry:

```
# in WSL2 Ubuntu shell  
  
sudo -i  
nano /etc/wsl.conf
```

This will open the `/etc/wsl.conf` in a nano, a command line text editor. Add the following line:

```
[boot]  
systemd=true
```

Save the file with CTRL-O and then quit nano with CTRL-X. Then, type the following line in powershell:

```
wsl --shutdown
```

and then relaunch WSL (Ubuntu) from the start menu. For those of you running Windows, we will be working exclusively from WSL2 now. If that is not an option, then I highly recommend you set up a virtual machine with Ubuntu using VirtualBox for example, or dual-boot Ubuntu.

1 Reproducibility with Nix

Installing (and uninstalling) Nix is quite simple, thanks to the installer from Determinate Systems, a company that provides services and tools built on Nix, and works the same way on Linux (native or WSL2) and macOS.

1.2.5.2 Actually installing Nix

Do not use your operating system's package manager to install Nix. Instead, simply open a terminal and run the following line (on Windows, run this inside WSL):

```
curl --proto '=https' --tlsv1.2 -sSf \
-L https://install.determinate.systems/nix | \
sh -s -- install
```

Then, install the `cachix` client and configure the `rstats-on-nix` cache: this will install binary versions of many R packages which will speed up the building process of environments:

```
nix-env -iA cachix -f
↪ https://cachix.org/api/v1/install
```

then use the cache:

```
cachix use rstats-on-nix
```

You only need to do this once per machine you want to use `{rix}` on. Many thanks to Cachix for sponsoring the `rstats-on-nix` cache!

If you get this warning when trying to install software with Nix:

1.2 Why Reproducibility? Why Nix? (1h30)

```
warning: ignoring the client-specified setting  
'trusted-public-keys', because it is a restricted  
setting and you are not a trusted user  
warning: ignoring untrusted substituter  
'https://rstats-on-nix.cachix.org', you are not a  
trusted user.
```

Run `man nix.conf` for more information on the
'substituters' configuration option.

```
warning: ignoring the client-specified setting  
'trusted-public-keys', because it is a restricted  
setting and you are not a trusted user
```

Then this means that configuration was not successful. You need
to add your user to /etc/nix/nix.custom.conf:

```
sudo vim /etc/nix/nix.custom.conf
```

then simply add this line in the file:

```
trusted-users = root YOURUSERNAME
```

where YOURUSERNAME is your current login user name.

1.2.6 Temporary shells

You now have Nix installed; before continuing, let's see if
everything works (close all your terminals and reopen them) by
dropping into a temporary shell with a tool you likely have not
installed on your machine.

Open a terminal and run:

```
which sl
```

1 Reproducibility with Nix

you will likely see something like this:

```
which: no sl in ....
```

now run this:

```
nix-shell -p sl
```

and then again:

```
which sl
```

this time you should see something like:

```
/nix/store/cndqpx74312xkrrgp842ifinkd4cg89g-sl-5.05/bin/sl
```

This is the path to the `sl` binary installed through Nix. The path starts with `/nix/store`: the *Nix store* is where all the software installed through Nix is stored. Now type `sl` and see what happens!

You can find the list of available packages here.

1.3 Session 1.2 – Dev Environments with Nix (*1h30*)

1.3.1 Some Nix concepts

While temporary shells are useful for quick testing, this is not how Nix is typically used in practice. Nix is a declarative pack-

age manager: users specify what they want to build, and Nix takes care of the rest.

To do so, users write files called `default.nix` that contain the a so-called Nix expression. This expression will contain the definition of a (or several) *derivations*.

In Nix terminology, a derivation is *a specification for running an executable on precisely defined input files to repeatably produce output files at uniquely determined file system paths.* (source)

In simpler terms, a derivation is a recipe with precisely defined inputs, steps, and a fixed output. This means that given identical inputs and build steps, the exact same output will always be produced. To achieve this level of reproducibility, several important measures must be taken:

- All inputs to a derivation must be explicitly declared.
- Inputs include not just data files, but also software dependencies, configuration flags, and environment variables, essentially anything necessary for the build process.
- The build process takes place in a *hermetic* sandbox to ensure the exact same output is always produced.

The next sections of this document explain these three points in more detail.

1.3.2 Derivations

Here is an example of a *simple* Nix expression:

```
let
  pkgs = import (fetchTarball
    "https://github.com/rstats-on-nix/nixpkgs/archive/2025-01-01.tar.gz");
in
```

1 Reproducibility with Nix

```
in

pkgs.stdenv.mkDerivation {
  name = "filtered_mtcars";
  buildInputs = [ pkgs.gawk ];
  dontUnpack = true;
  src = ./mtcars.csv;
  installPhase = ''
    mkdir -p $out
    awk -F',' 'NR==1 || $9=="1" { print }' $src >
    $out/filtered.csv
  '';
}
```

I won't go into details here, but what's important is that this code uses `awk`, a common Unix data processing tool, to filter the `mtcars.csv` file to keep only rows where the 9th column (the `am` column) equals 1. As you can see, a significant amount of boilerplate code is required to perform this simple operation. However, this approach is completely reproducible: the dependencies are declared and pinned to a specific dated branch of our `rstats-on-nix/nixpkgs` fork (more on this later), and the only thing that could make this pipeline fail (though it's a bit of a stretch to call this a *pipeline*) is if the `mtcars.csv` file is not provided to it. This expression can be *instantiated* into a derivation, and the derivation is then built into the actual output that interests us, namely the filtered `mtcars` data.

The derivation above uses the `Nix` builtin function `mkDerivation`: as its name implies, this function *makes a derivation*. But there is also `mkShell`, which is the function that builds a shell instead.

Nix expressions that built a shell is the kind of expressions {rix} generates for you.

1.3.3 Using {rix} to generate development environments

If you have successfully installed Nix, but don't have yet R installed on your system, you could install R as you would usually do on your operating system, and then install the {rix} package, and from there, generate project-specific expressions and build them. But you could also install R using Nix. Actually, I would even recommend you uninstall R and delete all your packages from your computer and only manager R environments using Nix.

Running the following line in a terminal will drop you in an interactive R session that you can use to start generating expressions:

```
nix-shell -p R rPackages.rix
```

This will drop you in a temporary shell with R and {rix} available. Navigate to an empty directory to help a project, call it **rix-session-1**:

```
mkdir rix-session-1
```

and start R and load {rix}:

```
R
```

1 Reproducibility with Nix

```
library(rix)
```

you can now generate an expression by running the following code:

```
rix(  
  date = "2025-08-04",  
  r_pkgs = c("dplyr", "ggplot2"), # add  
  ↵ languageserver if you plan to use VS Code  
  py_conf = list(  
    py_version = "3.13",  
    py_pkgs = c("polars", "great-tables")  
,  
  ide = "none",  
  project_path = ".",  
  overwrite = TRUE  
)
```

This will write a file called `default.nix` in your project's directory. This `default.nix` contains a Nix expression which will build a shell that comes with R, `{dplyr}` and `{ggplot2}` as they were on the the 4th of August 2025 on CRAN. This will also add Python 3.13 and the `polars` and `great-tables` Python packages as they were at the time in `nixpkgs` (more on this later). Finally, the `ide` argument is set to `"none"`, because we don't want to use Nix to manage an IDE. However, you should know that this is possible and useful in some cases. See this vignette for learning how to setup your IDE with Nix if you wish to do so. At the end of this chapter we will learn how to set up Positron.

1.3.4 Using nix-shell to Launch Environments

Once your file is in place, simply run:

```
nix-shell
```

This gives you an isolated shell session with all declared packages available. You can test code, explore APIs, or install further tools within this session.

To remove the packages that were installed, call `nix-store --gc`. This will call the garbage collector. If you want to avoid that an environment gets garbage-collected, use `nix-build` instead of `nix-shell`. This will create a symlink called `result` in your project's root directory and `nix-store --gc` won't garbage-collect this environment until you manually remove `result`.

1.3.5 Pinning with nixpkgs

To ensure long-term reproducibility, a pinned the version of Nix-pkgs is used:

```
let
  pkgs = import (fetchTarball
    { url = "https://github.com/rstats-on-nix/nixpkgs/archive/2025-";
      sha256 = "0000000000000000000000000000000000000000000000000000000000000000";
    });
in
...
```

This is done automatically by `{rix}`. You could change the date manually if you prefer, but I would recommend to always regenerate the `default.nix` using `{rix}`.

1.3.6 Installing Python packages not available via nixpkgs (impure)

Not all Python packages can be installed through Nix; unlike CRAN, Pypi doesn't get automatically mirrored and individual packages fixed by volunteers. Instead, specific Python packages get packaged individually for Nix. Thus, it could very well be the case that a specific Python package (or version of a Python package) that you need for a project is not available via nixpkgs.

In this case, it is still possible to use Python-specific package managers, like `uv`, to install packages. This is also useful if you work on a project with colleagues that use `uv` and that don't want (yet) to use Nix. `uv`, is 10-100x faster than `pip` and also generates a lock file for improved reproducibility.

The idea is to install `uv` in your shell (but not any Python nor Python packages):

```
rix(  
  ...  
  system_pkgs = c("uv"),  
  ...  
)
```

And then use `uv` from your shell as you would usually. We recommend specifying Python packages in a `requirements.txt` file, and specifying explicit versions (e.g., `scipy==1.11.4`). Finally, we also recommend setting a shell hook to set up the virtual environment and install the packages from the `requirements.txt` when entering the shell (mind the quotes):

```
rix(
    ...
    system_pkgs = c("uv"),
    shell_hook =
        if [ ! -f pyproject.toml ]; then
            uv init --python 3.13.5 # or whichever
        ↵ Python version you need
        fi
        uv add --requirements requirements.txt
        # Create alias so python uses uvs environment
        alias python='uv run python'
    ",
)
)
```

After running `nix-shell`, `uv` should initialize a Python project with the specified Python version and install the packages listed in `requirements.txt` within the nix environment. This will take place each time `nix-shell` is called, however this will be cached and not installed each time.

To make sure everything works fine, you could simply start a Python interpreter and try to load `numpy`. This should work fine, but if it doesn't, the following error could be raised:

```
ImportError:

IMPORTANT: PLEASE READ THIS FOR ADVICE ON HOW TO
        ↵ SOLVE THIS ISSUE!

Importing the numpy C-extensions failed. This error
        ↵ can happen for
many reasons, often due to issues with your setup or
        ↵ how NumPy was
```

1 Reproducibility with Nix

installed.

We have compiled some common reasons and

↳ troubleshooting tips at:

↳ <https://numpy.org/devdocs/user/troubleshooting-importer>

Please note and check the following:

- * The Python version is: Python3.13 from
 - ↳ "/home/user/projects/rix_uv/.venv/bin/python3"
- * The NumPy version is: "2.2.6"

and make sure that they are the versions you expect.

↳ Please carefully study the documentation linked above for further help.

Original error was: libstdc++.so.6: cannot open

↳ shared object file: No such file or directory

That is an issue when using *wheels* (wheels are binaries of Python packages that get installed by default using uv). These wheels expect certain libraries to be in certain places. One way to solve this is to add the following to your shell hook:

```
shellHook = ''  
# Export LD_LIBRARY is required for python  
# packages that dynamically load libraries,  
# such as numpy  
export  
↳ LD_LIBRARY_PATH="${pkgs.lib.makeLibraryPath  
↳ (with pkgs; [ zlib gcc.cc glibc stdenv.cc.cc  
↳ ])}":LD_LIBRARY_PATH;
```

```
...  
';
```

Your environment should now work.

If this seems complicated: yes, and that is actually exactly the type of problems that Nix aims to solve. However, there are just too many Python packages to automate their inclusion into nixpkgs like how it's done for R. If you can, prefer using the Python packages included in nixpkgs.

1.4 Configuring your IDE

1.4.1 Pre-requisites

We now need to configure an IDE to use both our Nix shells as development environments, and GitHub Copilot. You are free to use whatever IDE you want but the instructions below are going to focus on Positron, which is a fork of VS Code geared towards data science. It works well with both Python and R and makes it quite easy to choose the right R or Python interpreter (which you'll have to do to make sure you're using the one provided by Nix, see [here](#)).

If you want to use VS Code proper, you can follow all the instructions here, but you need to install the REditorSupport and the Python extension. You will also need to add the `{languageserver}` R package to the shell:

1 Reproducibility with Nix

```
rix(  
  date = "2025-08-04",  
  r_pkgs = c("dplyr", "ggplot2", "languageserver"),  
  py_conf = list(  
    py_version = "3.13",  
    py_pkgs = c("polars", "great-tables")  
)  
  ),  
  ide = "none",  
  project_path = ".",  
  overwrite = TRUE  
)
```

and re-create the `default.nix` file.

You'll need to carefully select the Python interpereter provided by Nix, and not the other environments that you might already have on your system. Also, we recommend you uninstall R if it's installed system-wide and also remove your local library of packages and instead only use dedicated Nix shells to manage your projects. While we made our possible for Nix shells to not interfere with a system-installed R, we recommend users go into the habit of taking some minutes at the start of a project to properly set up their development environment.

On Windows, you need to install Positron on Windows, not inside WSL. Then, install the Open Remote extension right from Positron itself.

1.4.2 direnv

Once Positron is installed, you need to install a piece of software called `direnv`: `direnv` will automatically load Nix shells when you open a project that contains a `default.nix` file in an editor.

It works on any operating system and many editors support it, including Positron. If you’re using Windows, install `direnv` in WSL (even though you’ve just installed Positron for Windows). To install `direnv` run this command:

```
nix-env -f '<nixpkgs>' -iA direnv
```

This will install `direnv` and make it available even outside of Nix shells!

Then, we highly recommend to install the `nix-direnv` extension:

```
nix-env -f '<nixpkgs>' -iA nix-direnv
```

It is not mandatory to use `nix-direnv` if you already have `direnv`, but it’ll make loading environments much faster and seamless.

Finally, if you haven’t used `direnv` before, don’t forget this last step to make your terminal detected and load `direnv` automatically.

Then, in Positron, install the `direnv` extension. Finally, add a file called `.envrc` and simply write the following two lines in it (this `.envrc` file should be in the same folder as your project’s `default.nix`):

```
use nix  
mkdir $TMP
```

in it. On Windows, *remotely connect to WSL* first, but on other operating systems, simply open the project’s folder using `File`

1 Reproducibility with Nix

> Open Folder... and you will see a pop-up stating `direnv: /PATH/TO/PROJECT/.envrc is blocked` and a button to allow it. Click Allow and then open an R script. You might get another pop-up asking you to restart the extension, so click Restart. Be aware that at this point, `direnv` will run `nix-shell` and so will start building the environment. If that particular environment hasn't been built and cached yet, it might take some time before Code will be able to interact with it. You might get yet another popup, this time from the R Code extension complaining that R can't be found. In this case, simply restart Code and open the project folder again: now it should work every time.

Sometimes, depending on platform and installed packages, Positron might not be able to correctly load a Nix-managed Python interpreter. This happens when it tries to load its own version of `ipykernal`. To make sure this doesn't cause issues, add these lines to the shell's definition in `default.nix`:

```
shellHook = ''  
  # Export LD_LIBRARY is required for python  
  #   packages that dynamically load libraries,  
  #   such as numpy  
  export  
  #   LD_LIBRARY_PATH="${pkgs.lib.makeLibraryPath  
  #   (with pkgs; [ zlib gcc.cc glibc stdenv.cc.cc  
  #   ])}"  
  LD_LIBRARY_PATH;  
  ...  
'';
```

1.4.3 In summary and next steps

For a new project, simply repeat this process:

- Generate the project’s `default.nix` file;
- Build it using `nix-build`;
- Create an `.envrc` and write the two lines from above in it;
- Open the project’s folder in Code and click allow when prompted;
- Restart the extension and Code if necessary.

Another option is to create the `.envrc` file and write `use nix` in it, then open a terminal, navigate to the project’s folder, and run `direnv allow`. Doing this before opening Positron should not prompt you anymore.

If you’re on Windows, using Positron like this is particularly interesting, because it allows you to install Positron on Windows as usual, and then you can configure it to interact with a Nix shell, even if it’s running from WSL. This is a very seamless experience.

Now configure Positron to use GitHub Copilot: [click here](#).

You can enable LLM tab-completion if you want. I’m not a fan of this, I prefer simply using the interface and using LLMs as an advanced rubber duck: <https://github.com/copilot/>.

1.5 Starting a new project:

To start a new project, create a new directory, and `cd` into it. Then, in that folder, create a new script called `gen-env.R` and add the following:

1 Reproducibility with Nix

```
library(rix)

rix(
  date = "2025-09-22",
  r_pkgs = c(
    "languageserver", # if you want to use VS Code
    "tidyverse" # or whatever packages
  ),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE,
  print = TRUE
)
```

Then call:

```
nix-shell -I
↳ nixpkgs=https://github.com/rstats-on-nix/nixpkgs/archive/re
↳ -p R rPackages.rix
```

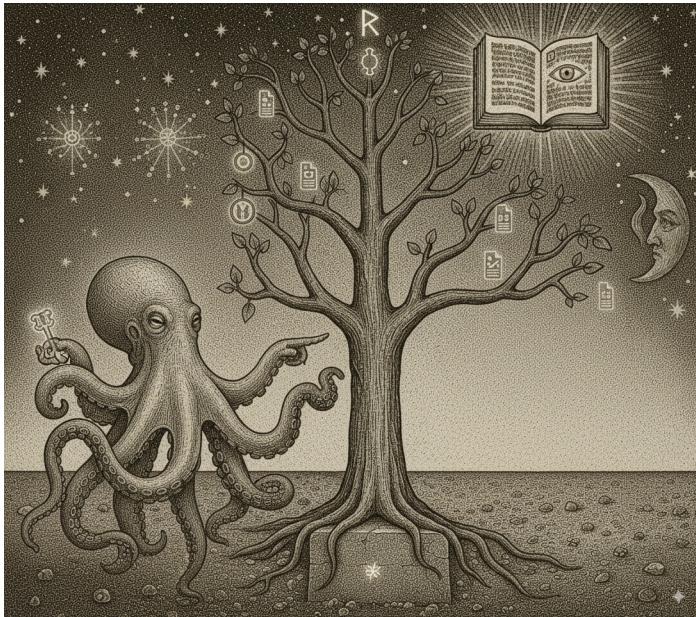
providing a url to my fork of `nixpkgs` makes sure that you get the latest `rix` version (change the date in the url to a more recent one if needed). Start R by typing `R` in the shell, and then `source("gen-env.R")`. That will create a `default.nix` which you can then drop into using `nix-shell`. Then create an `.envrc` and add the lines from before, and you're done!

Note: if you want to use `uv` instead of Nix to manage the Python packages, follow the instructions above.

1.6 Hands-On Exercises

1. Start a temporary shell with R and {rix} again using `nix-shell -p R rPackages.rix`. Start an R session (by typing R) and then load the {rix} package (using `library(rix)`). Run the `available_dates()` function: using the latest available date, generate a new `default.nix`.
2. Inside of an activated shell, type `which R` and `echo $PATH`. Explore what is being added to your environment. What is the significance of paths like `/nix/store/...`?
3. Break it on purpose: generate a new environment with a wrong R package name, for example `dplyrnaught`. Try to build the environment. What happens?
4. Go to <https://search.nixos.org/packages> and look for packages that you usually use for your projects to see if they are available.

2 Version Control with Git



What you'll learn by the end of this chapter:

- How to manage your own data science projects using Git's core command-line tools.
- How to collaborate effectively with a team using professional workflows like Pull Requests and Trunk-Based Development.
- How to safely review, manage, and integrate code generated by AI assistants like GitHub Copilot.

2.1 Introduction

Git is a software for version control. Version control is absolutely essential in software engineering, or when setting up a RAP. If you don't install a version control system such as Git, don't even start trying to set up a RAP. But what does a version control system like Git actually do? The basic workflow of Git is as follows: you start by setting up a repository for a project. On your computer, this is nothing more than a folder with your scripts in it. However, if you're using Git to keep track of what's inside that folder, there will be a hidden `.git` folder with a bunch of files in it. You can forget about that folder, this is for Git's own internal needs. What matters, is that when you make changes to your files, you can first *commit* these changes, and then push them back to a repository. Collaborators can copy this repository and synchronize their files saved on their computers with your changes. Your collaborators can then also work on the files, then commit and push the changes to the repository as well.

You can then pull back these changes onto your computer, add more code, commit, push, etc... Git makes it easy to collaborate on projects either with other people, or with future you. It is possible to roll back to previous versions of your code base, you can create new branches of your project to test new features (without affecting the main branch of your code), collaborators can submit patches that you can review and merge, and and and...

In my experience, learning Git is one of the most difficult things there is for students. And this is because Git solves a complex problem, and there is no easy way to solve a complex problem. But I would however say that Git is not unnescessarily complex, and in any case it's absolutely essential in our line of work. It

is simply not possible to not know at least some basics of Git. And this is what we're going to do, learn the basics, it'll keep us plenty busy already.

But for now, let's pause for a brief moment and watch this video that explains in 2 minutes the general idea of Git.

Let's get started.

You might have heard of github.com: this is a website that allows programmers to set up repositories on which they can host their code. The way to interact with github.com is via Git; but there are many other website like github.com, such as gitlab.com and bitbucket.com.

For this course, you should create an account on github.com. This should be easy enough. Then you should install Git on your computer.

Another advantage of using GitHub is that, as students, you will have access to Copilot for free. We will be using Copilot as our LLM for pair programming throughout the rest of this course. Get GitHub education here.

2.2 Installing Git

Installing Git is not hard; it installs like any piece of software on your computer. If you're running a Linux distribution, chances are you already have Git installed. To check if it's already installed on a Linux system, open a terminal and type `which git`. If a path gets returned, like `usr/bin/git`, congratulations, it's installed, if the command returns nothing you'll have to install it. On Ubuntu, type `sudo apt-get install git` and just wait a bit. If you're using macOS or Windows, you will need to install

2 Version Control with Git

it manually. For Windows, download the installer from here, and for macOS from here; you'll see that there are several ways of installing it on macOS, if you've never heard of homebrew or macports then install the binary package from here.

It would also be possible to install it with Nix, but because Git is also useful outside of development shells, it is better to have it installed at the level of your operating system.

Next, configure git:

```
git config --global user.name "Your Name"  
git config --global user.email  
  ↵ "your.email@example.com"
```

2.3 Setting up a repo

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

2.3 Setting up a repo

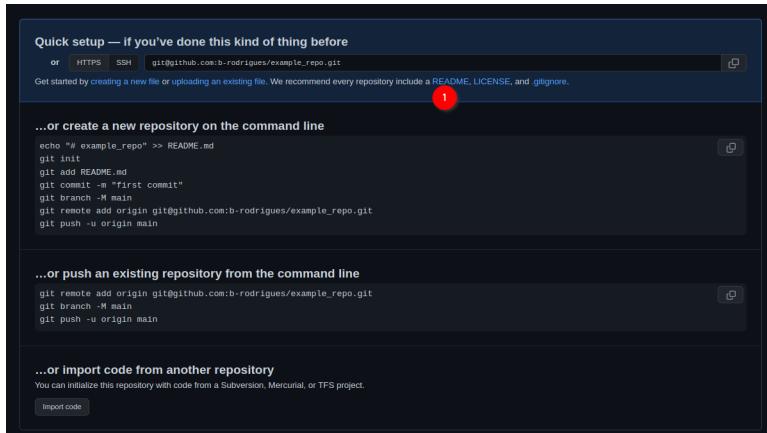
The screenshot shows a GitHub profile for a user named Bruno Rodrigues (b-rodrigues). The profile picture is a circular image of a man with a beard and sunglasses. Below the profile picture, the user's name 'Bruno Rodrigues' and handle 'b-rodrigues' are displayed, along with a link to their profile ('Edit profile'). The user has 192 followers and 12 following. Their location is listed as MESR, Luxembourg, Luxembourg-City, Luxembourg, with a link to their website (<http://www.brodrigues.co/>). A section titled 'Achievements' shows several small icons representing completed challenges. To the right, there is a grid of popular repositories: 'chronicler' (Public), 'modern_R' (Public), 'brotools' (Public), 'fput' (Public), 'coolmp3project' (Public), and 'covid_pred' (Public). Below these, a chart titled '56 contributions in the last year' shows a heatmap of activity across months and days. A tooltip for the heatmap indicates 'Learn how we count contributions'. On the top right, a red circle with the number '2' is overlaid on the 'New repository' button in the top right corner of the header.

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

The screenshot shows the 'Create a new repository' form. At the top, it says 'Create a new repository' and provides a note: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' Below this, there is a 'Repository template' section with a note: 'Start your repository with a template repository's contents.' A 'No template' button is shown. The main form fields are 'Owner' (set to 'b-rodrigues') and 'Repository name' (containing '/'). A red circle with the number '1' is overlaid on the 'Repository name' field. Below these, a note says 'Great repository names are short and memorable. Need inspiration? How about [super-duper-memory?](#)'. There is also a 'Description (optional)' field with a placeholder text area. Under 'Initialize this repository with:', there is a note: 'Skip this step if you're importing an existing repository.' and a 'Add a README file' checkbox, which is checked. A note below it says 'This is where you can write a long description for your project. [Learn more](#)'. There is also an 'Add .gitignore' checkbox and a 'Choose a license' section with a note: 'A license tells others what they can and can't do with your code. [Learn more](#)'. A note at the bottom says 'You are creating a public repository in your personal account.' A green 'Create repository' button is at the bottom, with a red circle with the number '2' overlaid on it.

2 Version Control with Git

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



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

2.3 Setting up a repo



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



2 Version Control with Git

Now we're going to work exclusively from the command line. While graphical interfaces for Git exist, learning the command line is essential because:

1. Most servers run Linux and only provide command line access
2. The command line gives you access to all Git features
3. Understanding the command line makes you more versatile as a developer
4. Many advanced Git operations can only be done from the command line

2.4 Cloning the repository onto your computer

Open your terminal (Linux/macOS) or WSL2 if on Windows. First, let's navigate to where we want to store our repository. For example, let's create a directory for our projects:

```
mkdir ~/Documents/projects  
cd ~/Documents/projects
```

Now let's clone the repository. Use the SSH URL you copied from GitHub:

```
git clone  
→ git@github.com:yourusername/your-repo-name.git
```

Replace `yourusername` and `your-repo-name` with your actual GitHub username and repository name.

After cloning, navigate into the repository:

```
cd your-repo-name  
ls -la
```

You should see the files from your repository, including the README file you created, plus a hidden `.git` directory that contains Git's internal files.

2.5 Setting up SSH authentication

Before we can push code from our computer to GitHub, we need a way to prove that we are who we say we are. While you can use a username and password (HTTPS), a more secure and professional method is to use SSH (Secure Shell) keys.

Think of it this way:

- **HTTPS (Password):** Like using a password to unlock a door. You have to type it in frequently.
- **SSH (Key):** Like having a special key that unlocks the door automatically. You set it up once, and it grants you access without needing to re-enter a password.

We will create a pair of digital keys: a **public key** that we will give to GitHub, and a **private key** that will stay on our computer. When we try to connect, GitHub will use our public key to check if we have the matching private key, proving our identity.

Let's generate our SSH key pair. We'll use the modern and highly secure Ed25519 algorithm. Open your terminal (or WSL2 on Windows) and run the following command, replacing the email with the one you used for GitHub:

2 Version Control with Git

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

You will be prompted with a few questions. Here is what you'll see and how to answer:

```
# Press Enter to accept the default file location
> Enter a file in which to save the key
  ↵  (/home/your_username/.ssh/id_ed25519): [Press
  ↵  Enter]

# You can optionally set a passphrase.
> Enter passphrase (empty for no passphrase): [Press
  ↵  Enter]
> Enter same passphrase again: [Press Enter]
```

What about the passphrase? A passphrase adds an extra layer of security. If someone were to steal your computer, they still couldn't use your SSH key without knowing the passphrase. However, you would have to type it every time you interact with GitHub. For this course, it is fine to leave it empty for convenience by simply pressing **Enter**.

After running the command, two files have been created in a hidden directory in your home folder called `.ssh`:

1. `id_ed25519`: This is your **private key**. **NEVER share this file with anyone or upload it anywhere**. It must remain secret on your computer.
2. `id_ed25519.pub`: This is your **public key**. The `.pub` stands for “public”. This is the key you can safely share and will upload to GitHub in the next step.

Note for Older Systems: If the `ssh-keygen` command gives an error about `ed25519` being an “invalid option”, your system might be too old to support it. In that rare case, you can use the older RSA algorithm instead: `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`

Now that we have our key pair, our next task is to give the public key to GitHub. Let’s display the public key:

```
cat ~/.ssh/id_ed25519.pub
```

Copy the entire output (starting with `ssh-rsa` and ending with your email).

Go to GitHub.com, click on your profile picture, then Settings → SSH and GPG keys → New SSH key. Paste your public key and give it a descriptive title.

Let’s test the connection:

```
ssh -T git@github.com
```

You should see a message confirming successful authentication.

2.6 Your first commit

Let’s create a simple script and add some code to it (in what follows, all the code is going to get written into files using the command line, but you can also use your text editor to do it):

2 Version Control with Git

```
echo 'print("Hello, Git!")' > hello.py
```

Or create a more complex example:

```
cat > analysis.R << 'EOF'  
# Load data  
data(mtcars)  
  
# Create a simple plot  
plot(mtcars$mpg, mtcars$hp,  
      xlab = "Miles per Gallon",  
      ylab = "Horsepower",  
      main = "MPG vs Horsepower")  
EOF
```

Now let's check the status of our repository:

```
git status
```

You'll see that Git has detected new untracked files. Let's add them to the staging area:

```
git add .
```

The `.` adds all files in the current directory. You can also add specific files:

```
git add analysis.R
```

Let's check the differences before committing:

2.7 Understanding Git workflow commands

```
git diff --staged
```

This shows what changes are staged for commit. Now let's commit with a descriptive message:

```
git commit -m "Add initial analysis script with  
↪ basic plot"
```

Let's check our commit history:

```
git log --oneline
```

Finally, push our changes to GitHub:

```
git push origin main
```

2.7 Understanding Git workflow commands

Here are the essential Git commands you'll use daily:

Checking status and differences:

```
git status          # Show working directory  
↪ status  
git diff           # Show unstaged changes  
git diff --staged  # Show staged changes  
git diff HEAD~1    # Compare with previous  
↪ commit
```

2 Version Control with Git

Adding and committing:

```
git add filename          # Stage specific file
git add .                 # Stage all changes
git commit -m "message"  # Commit with message
git commit -am "msg"      # Add and commit tracked
                         ↵ files
```

Working with remote repositories:

```
git push origin main     # Push to main branch
git pull origin main     # Pull latest changes
git fetch                # Download changes without
                         ↵ merging
```

Viewing history:

```
git log                  # Show detailed commit
                         ↵ history
git log --oneline        # Show abbreviated history
git log --graph          # Show branching history
git show commit-hash     # Show specific commit
                         ↵ details
```

2.8 Working with commit history

Let's explore how to work with previous versions. First, let's make another change:

```
echo '# This is a new line' >> analysis.R  
git add analysis.R  
git commit -m "Add comment to analysis script"
```

View the commit history:

```
git log --oneline
```

To view a previous version without changing anything:

```
git checkout <commit-hash>  
cat analysis.R # View the file at that point in  
    ↵ time
```

You'll be in “detached HEAD” state. To return to the latest version:

```
git checkout main
```

To permanently revert a commit (creates a new commit that undoes changes):

```
git revert <commit-hash>
```

2.9 Collaborating and handling conflicts

Let's set up collaboration. Have a colleague invite you to their repository, or invite someone to yours. On GitHub, go to Settings → Manage access → Invite a collaborator.

Once you're both collaborators, try this workflow:

2 Version Control with Git

1. Both of you clone the repository
2. One person makes changes and pushes:

```
echo 'library(ggplot2)' > new_analysis.R  
git add new_analysis.R  
git commit -m "Add ggplot2 analysis"  
git push origin main
```

3. The other person attempts to push their own changes:

```
echo 'data(iris)' > new_analysis.R  
git add new_analysis.R  
git commit -m "Add iris analysis"  
git push origin main # This will fail!
```

You'll get an error like `! [rejected] main -> main (non-fast-forward)`. This sounds scary, but it's Git's safe way of telling you: “**The remote repository on GitHub has changes that you don't have on your computer. I'm stopping you from pushing because you would overwrite those changes.**”

To solve this, you must first pull the changes from the remote repository and combine them with your local work. Git gives you two primary ways to do this: **merging** and **rebasing**.

2.9.1 Strategy 1: Merging (The Default)

If you just run `git pull`, Git will perform a *merge*. It looks at the remote changes and your local changes and creates a new, special “merge commit” to tie the two histories together.

Imagine the history looks like this:

- Your colleague pushed commit D.
- You worked locally and created commit C.



A `git pull` (which is `git fetch + git merge`) will result in this:



The history is now non-linear. While this accurately records that two lines of work were merged, it can clutter up the project history with many “Merge branch ‘main’...” commits, making it harder to read.

2.9.2 Strategy 2: Rebasing (The Cleaner Way)

The second strategy is to *rebase*. Rebasing does something clever. It says: “Let me temporarily put your local changes aside. I’ll download the latest remote changes first. Then, I’ll take your changes and re-apply them one-by-one on top of the new remote history.”

Using the same scenario:

- Start: C (Your local work) / A---B
---D (Remote work on GitHub)
- Running `git pull --rebase` does this:
 1. It “unplugs” your commit C.

2 Version Control with Git

2. It fast-forwards your `main` branch to include D.
 3. It then “re-plays” your commit C on top of D, creating a new commit C'.
- The final result is a clean, single, linear history:
`A---B---D---C'` (Your work is now on top)

Your project’s history now looks like you did your work *after* your colleague, even if you did it at the same time. This makes the log much easier to read and understand.

For its clean, linear history, **rebasing is the preferred method in many professional workflows, and it’s the one we will use.**

Now, let’s do it. To pull the remote changes and place your local commits on top, run:

```
git pull --rebase origin main
```

If there are no conflicts, Git will automatically complete the rebase. Your local work will now be neatly stacked on top of the remote changes, and your `git push` will succeed.

If there are conflicts, Git will pause the rebase process and tell you which files have conflicts. This happens when you and a collaborator changed the same lines in the same file.

```
git status # Shows "You are currently rebasing."  
↪ and lists conflicted files
```

Your job is to be the surgeon. Open the conflicted files (e.g., `analysis.R`). You will see Git’s conflict markers:

2.9 Collaborating and handling conflicts

```
<<<<< HEAD
# This is my version of the code
data(iris)
=====
# This is their version from the server
data(mtcars)
>>>>> a1b2c3d... Add mtcars analysis
```

Manually edit the file to resolve the conflict. You must delete the <<<<<, =====, and >>>>> markers and decide what the final, correct version of the code should be. For example:

```
# I decided to keep both datasets for now
data(iris)
data(mtcars)
```

Once you have fixed the file and saved it, you need to tell Git you're done:

```
# Mark the conflict as resolved
git add conflicted-file.R

# Continue the rebase process
git rebase --continue
```

Git will continue applying your commits one by one. If you have another conflict, repeat the process. Once the rebase is complete, you can finally push your work.

Finally, push your changes:

2 Version Control with Git

```
git push origin main
```

This time, it should succeed.

2.10 Working with branches

Branches allow you to work on features without affecting the main codebase:

```
# Create and switch to a new branch  
git checkout -b feature-new-plots  
  
# Or use the newer syntax  
git switch -c feature-new-plots
```

List all branches:

```
git branch
```

Work on your feature:

```
echo 'boxplot(mtcars$mpg ~ mtcars$cyl)' >>  
  ↵ analysis.R  
git add analysis.R  
git commit -m "Add boxplot analysis"
```

Push the branch to GitHub:

```
git push origin feature-new-plots
```

Switch back to main and merge your feature:

```
git checkout main
git merge feature-new-plots
```

If you're done with the branch, delete it:

```
git branch -d feature-new-plots          # Delete
  ↵ locally
git push origin --delete feature-new-plots # Delete
  ↵ on GitHub
```

2.11 Advanced workflow with branches

For more complex workflows, you might want to keep branches separate and use pull requests on GitHub instead of direct merging:

```
# Create feature branch
git checkout -b feature-advanced-stats
echo 'summary(lm(mpg ~ hp + wt, data = mtcars))' >>
  ↵ analysis.R
git add analysis.R
git commit -m "Add linear regression analysis"
git push origin feature-advanced-stats
```

Then go to GitHub and create a Pull Request from the web interface. This allows for code review before merging.

2.12 Essential daily workflow

Here's the typical daily workflow:

1. Start your day: Pull latest changes

```
git pull origin main
```

2. Create a feature branch:

```
git checkout -b feature-description
```

3. Work and commit frequently:

```
# Make changes  
git add .  
git commit -m "Descriptive commit message"
```

4. Push your branch:

```
git push origin feature-description
```

5. When feature is complete: Merge or create pull request

```
git checkout main  
git pull origin main # Get latest changes  
git merge feature-description  
git push origin main
```

2.13 A Better Way to Collaborate: Trunk-Based Development

The “Essential Daily Workflow” you just learned is a great start, but it leaves one important question unanswered: how long should a feature branch live? Days? Weeks? Months?

A common mistake for new teams is to let branches live for a very long time. A data scientist might create a branch called **feature-big-analysis**, work on it for three weeks, and then try to merge it back into **main**. The result is often what’s called “merge hell”: **main** has changed so much in three weeks that merging the branch back in creates dozens of conflicts and is a painful, stressful process.

To avoid this, many professional teams use a workflow called **Trunk-Based Development (TBD)**. The philosophy is simple but powerful:

All developers integrate their work back into the main branch (the “trunk”) as frequently as possible—at least once a day.

This means that feature branches are incredibly **short-lived**. Instead of a single, massive feature branch that takes weeks, you create many tiny branches that each take a few hours or a day at most.

The goal is to keep the **main** branch constantly updated with the latest code from everyone on the team. This has huge benefits:

- **Fewer Merge Conflicts:** Because you are merging small changes frequently, the chance of conflicting with a teammate’s work is dramatically lower.

2 Version Control with Git

- **Easier Code Reviews:** Reviewing a small change that adds one function is much easier and faster than reviewing a 1,000-line change that refactors an entire analysis.
- **Continuous Integration:** Everyone is working from the most up-to-date version of the project, which reduces integration problems and keeps the project moving forward.

2.13.1 How to Work with Short-Lived Branches

But how can you merge something back into `main` if the feature isn't finished? The `main` branch must **always be stable and runnable**. You can't merge broken code.

The first way to solve this issue is to use feature flags.

A feature flag is just a simple variable (like a TRUE/FALSE switch) that lets you turn a new, unfinished part of the code on or off. This allows you to merge the code into `main` while keeping it “off” until it's ready.

Imagine you are adding a new, complex plot to `analysis.R`, but it will take a few days to get right.

```
# At the top of your analysis.R script
# --- Configuration ---
use_new_scatterplot <- FALSE # Set to FALSE while in
  ↵ development

# ... lots of existing, working code ...

# --- New Feature Code ---
if (use_new_scatterplot) {
  # All your new, unfinished, possibly-buggy
    ↵ plotting code goes here.
```

2.13 A Better Way to Collaborate: Trunk-Based Development

```
# It won't run as long as the flag is FALSE.  
library(scatterplot3d)  
scatterplot3d(mtcars$mpg, mtcars$hp, mtcars$wt)  
}
```

With this `if` block, you can safely merge your changes into `main`. The new code is there, but it won't execute and won't break the existing analysis. Other developers can pull your changes and won't even notice. Once you've finished the feature in subsequent small commits, the final change is just to flip the switch: `use_new_scatterplot <- TRUE`.

The second strategy is to *stack* pull requests. This is useful when a feature is too big for one small change, but it can be broken down into a logical sequence of steps. For example, to add a new analysis, you might need to:

1. Add a new data cleaning function.
2. Use that function to process the data.
3. Generate a new plot from the processed data.

Instead of putting all this in one giant Pull Request (PR), you can “stack” them. A stacked PR is a PR that is based on another PR branch, not on `main`.

Here's the workflow:

1. **Create the first branch** from `main` for the first step.

```
git switch -c add-cleaning-function  
# ...do the work, commit, and push...
```

Create a Pull Request on GitHub for this branch (`add-cleaning-function -> main`).

2 Version Control with Git

2. Create the second branch *from the first branch.*

This is the key step.

```
git switch -c process-the-data
# ...do the work that DEPENDS on the cleaning
  ↵  function...
```

Create a new PR for this branch. On GitHub, when you create the PR, **manually change the base branch from `main` to `add-cleaning-function`.** Now this PR only shows the changes for step 2.

Your team can now review and approve `add-cleaning-function` first. Once it's merged into `main`, you go to your `process-the-data` PR on GitHub and change its base back to `main`. It will now be ready to merge after a quick update.

This approach breaks down large features into small, logical, reviewable chunks, keeping your development velocity high while adhering to the TBD philosophy.

By embracing short-lived branches, feature flags, and stacked PRs, you can make collaboration smoother, less stressful, and far more productive.

2.14 Contributing to someone else's repository

To contribute to repositories you don't have write access to:

1. **Fork the repository** on GitHub (click the Fork button)
2. **Clone your fork:**

2.14 Contributing to someone else's repository

```
git clone  
  ↳ git@github.com:yourusername/original-repo-name.git  
cd original-repo-name
```

3. Add the original repository as upstream:

```
git remote add upstream  
  ↳ git@github.com:originalowner/original-repo-name.git
```

4. Create a feature branch:

```
git checkout -b fix-issue-123
```

5. Make changes and commit:

```
# Make your changes  
git add .  
git commit -m "Fix issue #123: describe what you  
  ↳ fixed"
```

6. Push to your fork:

```
git push origin fix-issue-123
```

7. Create a Pull Request on GitHub from your fork to the original repository

This workflow is fundamental for contributing to open source projects and collaborating in professional environments.

The command line approach to Git gives you complete control and understanding of the version control process, making you a more effective developer and collaborator.

2.15 Working with LLMs and Git: Managing AI-Generated Changes

When working with Large Language Models (LLMs) like GitHub Copilot, ChatGPT, or Claude to generate or modify code, it's crucial to review changes carefully before committing them. Git provides excellent tools for examining and selectively accepting or rejecting AI-generated modifications.

2.15.1 The LLM workflow with Git

Here's a recommended workflow when using LLMs to modify your code:

1. Always commit your working code first:

```
git add .
git commit -m "Working state before LLM
    ↵ modifications"
```

2. Apply LLM suggestions to your files (copy-paste, or use tools that directly modify files)
3. Review changes chunk by chunk using Git's tools
4. Selectively accept or reject changes
5. Commit accepted changes with descriptive messages

2.15.2 Examining LLM changes

After an LLM has modified your files, use Git to see exactly what changed:

```
# See all modified files  
git status  
  
# See all changes at once  
git diff  
  
# See changes in a specific file  
git diff analysis.R  
  
# See changes with more context (10 lines  
#   ↵ before/after)  
git diff -U10 analysis.R
```

For a more visual review, you can use Git's word-level diff:

```
# Show word-by-word changes instead of line-by-line  
git diff --word-diff analysis.R  
  
# Show character-level changes  
git diff --word-diff=color --word-diff-regex=.
```

2.15.3 Interactive staging: Accepting changes chunk by chunk

Git's interactive staging feature (`git add -p`) is perfect for reviewing LLM changes. It lets you review each “hunk” (chunk of changes) individually:

2 Version Control with Git

```
git add -p
```

This will show you each chunk of changes and prompt you with options: - y - stage this hunk - n - do not stage this hunk - q - quit; do not stage this hunk or any remaining ones - a - stage this hunk and all later hunks in the file - d - do not stage this hunk or any later hunks in the file - s - split the current hunk into smaller hunks - e - manually edit the current hunk - ? - print help

2.15.4 Example: Reviewing LLM changes to an R script

Let's say an LLM modified your `analysis.R` file. Here's how to review it:

```
# First, see what files were modified
git status

# Review the changes
git diff analysis.R
```

You might see output like:

```
@@ -1,8 +1,12 @@
 # Load required libraries
-library(ggplot2)
+library(ggplot2)
+library(dplyr)
+library(tidyr)
```

2.15 Working with LLMs and Git: Managing AI-Generated Changes

```
# Load data
data(mtcars)
+mtcars <- mtcars %>%
+  mutate( efficiency = ifelse(mpg > 20, "High",
+    "Low"))

-# Create a simple plot
-plot(mtcars$mpg, mtcars$hp)
+# Create an improved plot with ggplot2
+ggplot(mtcars, aes(x = mpg, y = hp, color =
+  efficiency)) +
+  geom_point(size = 3) +
+  theme_minimal()
```

Now use interactive staging to review each change:

```
git add -p analysis.R
```

Git will show you each hunk and ask what to do. For example:

```
@@ -1,2 +1,4 @@
 # Load required libraries
 library(ggplot2)
+library(dplyr)
+library(tidyr)
 Stage this hunk [y,n,q,a,d,s,e,?]?
```

You might decide:

- y if you want the additional libraries

2 Version Control with Git

- `n` if you think they're unnecessary
- `s` to split this into smaller chunks if you want only one library

2.15.5 Advanced chunk management

Sometimes hunks are too large. Use `s` to split them:

```
# When prompted with a large hunk
Stage this hunk [y,n,q,a,d,s,e,?]?
```

If Git can't split automatically, use `e` to manually edit:

```
Stage this hunk [y,n,q,a,d,s,e,?]?
```

This opens your editor where you can:

- Remove lines you don't want (delete the entire line)
- Keep lines by leaving them as-is
- Lines starting with `+` are additions
- Lines starting with `-` are deletions
- Lines starting with `(space)` are context

2.15.6 Creating meaningful commits after LLM review

After selectively staging changes, commit with descriptive messages:

```
# Commit the staged changes
git commit -m "Add dplyr and efficiency
    categorization

- Added dplyr for data manipulation
- Created efficiency category based on mpg > 20
- LLM suggested changes reviewed and approved"

# If there are remaining unstaged changes you want
    to reject
git checkout -- analysis.R # Revert unstaged
    changes
```

2.15.7 Working with multiple files modified by LLM

When an LLM modifies multiple files, review them systematically:

```
# See all changed files
git status

# Review each file individually
git diff analysis.R
git diff data_processing.R
git diff visualization.R

# Use interactive staging for each file
git add -p analysis.R
git add -p data_processing.R
# ... etc
```

2 Version Control with Git

Or stage all changes interactively at once:

```
git add -p
```

2.15.8 Handling LLM-generated new files

When an LLM creates entirely new files:

```
# See new files
git status

# Review new file content
cat new_functions.R

# Add if you approve
git add new_functions.R

# Or ignore if you don't want it
echo "new_functions.R" >> .gitignore
```

2.15.9 Using Git to compare LLM suggestions

Create a branch to safely experiment with LLM suggestions:

```
# Create a branch for LLM experiments
git checkout -b llm-suggestions

# Apply LLM changes
# ... make modifications ...
```

```
# Commit the LLM suggestions
git add .
git commit -m "LLM suggestions for code improvement"

# Compare with original
git diff main..llm-suggestions

# If you like some but not all changes, cherry-pick
# specific commits
git checkout main
git cherry-pick --no-commit <commit-hash>
git add -p # Selectively stage parts of the
# cherry-picked changes
git commit -m "Selected improvements from LLM
# suggestions"
```

2.15.10 Best practices for LLM + Git workflow

1. Always commit working code before applying LLM suggestions
2. Never blindly accept all LLM changes - review each modification
3. Use descriptive commit messages that mention LLM involvement
4. Test code after accepting LLM suggestions before final commit
5. Keep LLM-generated changes in separate commits for easier tracking
6. Use branches for experimental LLM suggestions
7. Document why you accepted or rejected specific suggestions

2.15.11 Example complete workflow

```
# 1. Save current working state
git add .
git commit -m "Working analysis script before LLM
    ↪ optimization"

# 2. Apply LLM suggestions (manually copy-paste or
    ↪ use tools)
# ... LLM modifies your files ...

# 3. Review all changes
git status
git diff

# 4. Interactively stage only the changes you want
git add -p

# 5. Commit approved changes
git commit -m "LLM improvements: added data
    ↪ validation and error handling

Reviewed and approved:
- Input validation for data loading
- Error handling for missing values
- Improved variable naming

Rejected:
- Overly complex optimization that hurt readability"

# 6. Discard remaining unwanted changes
git checkout .
```

```
# 7. Test the code  
Rscript analysis.R # or python script.py  
  
# 8. Push if everything works  
git push origin main
```

This workflow ensures you maintain full control over your code-base while benefiting from LLM assistance, with complete traceability of what changes were made and why.

2.16 Hands-on Exercises

2.16.1 Exercise 1: Setup and First Commit (skip if you followed along)

Goal: To confirm that Git is installed and configured correctly, and to practice the fundamental `clone`, `add`, `commit`, and `push` cycle.

1. On your GitHub account, create a new **public** repository named `data-analysis-project`.
2. Initialize the repository with a `README.md` file directly on GitHub.
3. On your local machine, navigate to a suitable projects directory (e.g., `~/Documents/projects`).
4. Clone your new repository to your local machine using the SSH URL.
5. Create a new file in the local repository called `hello.py` that contains a single line: `print("Hello, Git!")`.
6. Use the command line to stage the new `hello.py` file.

2 Version Control with Git

7. Commit the staged file with the message “Add initial Python script”.
8. Push your commit to the `main` branch on GitHub.
9. Verify that the `hello.py` file appears in your repository on the GitHub website.

2.16.1.1 Exercise 2: Exploring History

Goal: To practice inspecting the project’s history and learn how to revert changes.

1. In your local `data-analysis-project` repository, modify the `README.md` file by adding a new line: “This is my project for analyzing interesting data.”
2. Commit this change with the message “Update README with project description”.
3. Next, create a new file named `analysis.R` and add the following code:
`R # Initial data exploration
data(iris) summary(iris)`
4. Commit this new file with the message “Add initial analysis for iris dataset”.
5. Use `git log --oneline` to view your commit history. You should see at least three commits.
6. Find the commit hash for your “Update README with project description” commit.
7. Now, create a new commit that **reverts** the README change. Use the `git revert <commit-hash>` command. A text editor will open for the revert commit message; you can leave the default message and just save and close it.
8. Push your changes to GitHub. On the GitHub history for the `main` branch, you should now see a new “Revert...”

commit, and the `README.md` file should no longer contain the project description.

2.16.1.2 Exercise 3: Working with Branches

Goal: To understand the workflow of creating a feature branch to work in isolation and then merging it back into the main branch.

1. In your local repository, create a new branch called `feature-visualization`.
2. Switch to the `feature-visualization` branch.
3. In the `analysis.R` file, add the following code at the end of the file to create a plot: `R # Add a simple plot of the data plot(iris$Sepal.Length, iris$Sepal.Width, main="Iris Sepal Dimensions")`
4. Commit this change to the `feature-visualization` branch with the message “Feat: Add sepal dimensions plot”.
5. Push the `feature-visualization` branch to GitHub.
6. Switch back to the `main` branch on your local machine.
7. Merge the `feature-visualization` branch into `main`.
8. Push the `main` branch to see the merged changes on GitHub.
9. (Optional) Delete the `feature-visualization` branch both locally and on the remote repository.

2.16.1.3 Exercise 4: Resolving Conflicts

Goal: To learn how to handle the common “non-fast-forward” error by using `git pull --rebase` and resolving a merge conflict.

2 Version Control with Git

1. On the GitHub website, navigate to your `analysis.R` file and use the “Edit” button to change the comment `# Initial data exploration to # Initial exploration of the Iris dataset.`. Commit this change directly on GitHub.
2. Now, on your **local machine** (without pulling the remote changes yet), edit the **exact same line** in your `analysis.R` file to be `# R-Script for Iris Data`.
3. Stage and commit this change locally with the message “Refactor: Update analysis script comment”.
4. Try to push your change using `git push origin main`. The push will be rejected because your local history has diverged from the remote’s history.
5. Fetch the remote changes and re-apply your local commit on top by running `git pull --rebase origin main`. This will trigger a merge conflict.
6. Open `analysis.R` in a text editor. You will see the conflict markers (`<<<<<`, `=====`, `>>>>>`).
7. Resolve the conflict by deleting the markers and choosing a final version for the line (e.g., `# R-Script for the initial exploration of the Iris dataset`).
8. After saving the file, stage the resolved file using `git add analysis.R`.
9. Continue the rebase with `git rebase --continue`.
10. Finally, push your successfully rebased and conflict-free history to GitHub.

2.16.1.4 Exercise 5: Reviewing AI-Generated Code

Goal: To practice the safe workflow for reviewing and selectively accepting code suggested by an LLM using interactive staging.

1. First, commit any outstanding work in your repository to

ensure you have a clean state.

2. Pretend an LLM has suggested an “improvement” to your `hello.py` file. Manually edit the file to look like this:

```
import sys

# A function to greet
def greet(name):
    """A function that prints a greeting."""
    print(f"Hello, {name}!") # The main change
    ↵ we want
    # The LLM also added this logging, which we
    ↵ don't want
    print("Function finished execution.",
    ↵ file=sys.stderr)

if __name__ == '__main__':
    greet("Git")
```

3. Use `git diff` to see the changes. Notice the unwanted `print` statement to `sys.stderr`.
4. Instead of staging the whole file, use interactive staging: `git add -p hello.py`.
5. Git will show you the chunk of changes. When prompted `Stage this hunk [y,n,q,a,d,s,e,?]?`, press `e` to manually edit the hunk.
6. Your text editor will open with the patch. Delete the line `+print("Function finished execution.", file=sys.stderr)`. Save and close the editor.
7. You have now staged only the parts of the change you approved. Commit the staged changes with a message like “Feat: Refactor `hello.py` into a function”.

2 Version Control with Git

8. Run `git status`. You will see that `hello.py` is still listed as modified. Use `git diff` to see the remaining change (the one you rejected).
9. Discard the unwanted change permanently with `git checkout -- hello.py`. Your working directory is now clean, and only the approved changes are in your commit history.

2.16.1.5 Exercise 6: Contributing to another Project (Fork & Pull Request)

Goal: To simulate contributing to an open-source project by forking a repository and submitting a Pull Request. For this exercise, use this repo to contribute to.

1. With your browser, navigate to the GitHub repository you want to contribute to (your partner's or instructor's).
2. Click the **Fork** button in the top-right corner to create a copy of the repository under your own account.
3. Clone **your fork** to your local machine.
4. Add the original repository as a new remote called `upstream`. You can do this with: `git remote add upstream git@github.com:original-owner/original-repo-name`.
5. Create a new branch for your contribution, for example, `fix-spelling-error`.
6. Make a small, helpful change. For example, find a typo in their `README.md` and fix it. Or add a new file `CONTRIBUTORS.md` and add your name.
7. Commit your change with a clear commit message.
8. Push your branch to **your fork** (`origin`), not `upstream`.
9. On GitHub, navigate to your forked repository. You should see a banner prompting you to “Contribute” and “Open a pull request”.

2.16 Hands-on Exercises

10. Open a Pull Request. Ensure the base repository is the original project's **main** branch and the head repository is your **fix-spelling-error** branch. Write a short title and description for your change and submit it. The owner of the original repository can now review your work.

3 Functional Programming



What you'll learn by the end of this chapter:

- Why functional programming is crucial for reproducible, testable, and collaborative data science.
- How to write self-contained, “pure” functions in both R and Python.
- How to use functional concepts like `map`, `filter`, and `reduce` to replace error-prone loops.

3 Functional Programming

- How writing functions makes your code easier to review, debug, and even generate with LLMs.

3.1 Introduction: From Scripts to Functions

So far, we've established two pillars of reproducible data science:

1. **Reproducible Environments (with Nix):** Ensuring everyone has the *exact same tools* (R, Python, system libraries) to run the code.
2. **Reproducible History (with Git):** Ensuring everyone has the *exact same version* of the code and can collaborate effectively.

Now we turn to the third and arguably most important pillar: **writing reproducible code itself**. A common way to start a data analysis is by writing a script: a sequence of commands that are executed from top to bottom.

```
# R script example
library(dplyr)
data(mtcars)

heavy_cars <- filter(mtcars, wt > 4)
mean_mpg_heavy <- mean(heavy_cars$mpg)
print(mean_mpg_heavy)
```

```
# Python script example
import pandas as pd
mtcars = pd.read_csv("mtcars.csv") # Assume the file
    ↵ exists
```

```
heavy_cars = mtcars[mtcars['wt'] > 4]
mean_mpg_heavy = heavy_cars['mpg'].mean()
print(mean_mpg_heavy)
```

This works, but it has a hidden, dangerous property: state. The script relies on variables like `heavy_cars` existing in the environment, making the code hard to reason about, debug, and test. If scripting with state is a crack in the foundation of reproducibility, then using computational notebooks is a gaping hole.

Notebooks like Jupyter introduce an even more insidious form of state: the cell execution order. You can execute cells out of order, meaning the visual layout of your code has no relation to how it actually ran. This is a recipe for non-reproducible results and a primary cause of the “it worked yesterday, why is it broken today?” problem.

The solution to this chaos is to embrace a paradigm that minimizes state: Functional Programming (FP). Instead of a linear script, we structure our code as a collection of self-contained, predictable functions. To support this, we will work exclusively in plain text files (`.R`, `.py`), which enforce a predictable, top-to-bottom execution, and use literate programming (using Quarto). The power of FP comes from the concept of purity, borrowed from mathematics. A mathematical function has a beautiful property: for a given input, it always returns the same output. `sqrt(4)` is always 2. Its result doesn’t depend on what you calculated before or on a random internet connection. Our Nix environments handle the “right library” problem; purity handles the “right logic” problem. Our goal is to write our analysis code with this same level of rock-solid predictability.

3 Functional Programming

To appreciate what FP brings, it helps to contrast it with **object-oriented programming (OOP)** which is arguably the dominant paradigm in many software systems. OOP grew out of large industrial projects where different teams or departments managed different parts of a system. Each team could define the behavior of its component (an “object”) and expose a limited interface to the rest of the organization. The core idea was *message passing*: you send a message to an object, asking it to perform an action, without needing to know how it works internally. OOP organizes computation around *who does what*, a network of objects communicating with each other and managing their own internal state.

Functional programming, by contrast, organizes computation around *how data changes*. It replaces a network of interacting objects with a flow of transformations: data goes in, data comes out, and nothing else changes in the process. Rather than objects that carry both data and behavior, FP favors pure functions that take data as input and return new data as output. The emphasis shifts from *actors* to *transformations* — from modeling the world as a set of entities that act, to modeling it as a sequence of transformations applied to values.

This shift is especially powerful in data science. Analyses are naturally expressed as pipelines of transformations (cleaning, filtering, aggregating, modeling) where each step should depend only on its input, not on hidden state elsewhere in the program. Functional programming aligns perfectly with this workflow. Pure functions make results reproducible, because the same inputs always yield the same outputs. Immutability prevents accidental side effects on shared data. And because transformations can be composed, tested, and reused independently, FP encourages modular, maintainable analysis code. Finally, because pure functions are self-contained, they can be easily

parallelized, making it easier to scale analyses across datasets or machines.

In short, OOP helps us build complex systems by modeling *actors* that exchange messages, while FP helps us build robust analyses by composing *transformations* of data. For data science, where clarity, reproducibility, and data flow matter most, the functional approach offers the stronger foundation.

3.1.1 Why Does This Matter for Data Science?

Adopting a functional style brings massive benefits that directly connect to our previous chapters:

1. **Unit Testing is Now Possible:** You can't easily test a 200-line script. But you *can* easily test a small function that does one thing. Does `calculate_mean_mpg(data)` return the correct value for a sample dataset? This makes your code more reliable.
2. **Code Review is Easier (Git Workflow):** As we saw in the Git chapter, reviewing a small, self-contained change is much easier than reviewing a giant, sprawling one. A Pull Request that just adds or modifies a single function is simple for your collaborators to understand and approve.
3. **Working with LLMs is More Effective:** It's difficult to ask an LLM to "fix my 500-line analysis script." It's incredibly effective to ask, "Write a Python function that takes a pandas DataFrame and a column name, and returns the mean of that column, handling missing values. Also, write three `pytest` unit tests for it." Functions provide the clear boundaries and contracts that LLMs excel at working with.

3 Functional Programming

4. **Readability and Maintainability:** Well-named functions are self-documenting.

```
starwars %>%
  group_by(species) %>%
  summarize(mean_height = mean(height))
```

is instantly understandable. The equivalent `for` loop is a puzzle you have to solve.

3.2 Purity and Side Effects

A **pure function** has two rules:

1. It only depends on its inputs. It doesn't use any "global" variables defined outside the function.
2. It doesn't change anything outside of its own scope. It doesn't modify a global variable or write a file to disk. This is called having "no side effects."

Consider this "impure" function in Python:

```
# IMPURE: Relies on a global variable
discount_rate = 0.10

def calculate_discounted_price(price):
    return price * (1 - discount_rate) # What if
    ↵ discount_rate changes?

print(calculate_discounted_price(100))
# > 90.0
discount_rate = 0.20 # Someone changes the state
print(calculate_discounted_price(100))
```

```
# > 80.0 -- Same input, different output!
```

The pure version passes *all* its dependencies as arguments:

```
# PURE: All inputs are explicit arguments
def calculate_discounted_price_pure(price, rate):
    return price * (1 - rate)

print(calculate_discounted_price_pure(100, 0.10))
# > 90.0
print(calculate_discounted_price_pure(100, 0.20))
# > 80.0
```

Now the function is predictable and self-contained.

3.2.1 Handling “Impure” Operations like Randomness

Some operations, like generating random numbers, are inherently impure. Each time you run `rnorm(10)` or `numpy.random.rand(10)`, you get a different result.

The functional approach is not to avoid this, but to *control* it by making the source of impurity (the random seed) an explicit input.

In R, the `{withr}` package helps create a temporary, controlled context:

3 Functional Programming

```
library(withr)

# This function is now pure! For a given seed, the
#   output is always the same.
pure_rnorm <- function(n, seed) {
  with_seed(seed, {
    rnorm(n)
  })
}

pure_rnorm(n = 5, seed = 123)
pure_rnorm(n = 5, seed = 123)
```

In Python, `numpy` provides a more modern, object-oriented way to handle this, which is naturally functional:

```
import numpy as np

# Create a random number generator instance with a
#   seed
rng = np.random.default_rng(seed=123)

# Now, calls on this 'rng' object are deterministic
#   within its context
print(rng.standard_normal(5))

# If we re-create the same generator, we get the
#   same numbers
rng2 = np.random.default_rng(seed=123)
print(rng2.standard_normal(5))
```

The key is the same: the “state” (the seed) is explicitly managed, not hidden globally.

However, this introduces a concept from another programming paradigm: **Object-Oriented Programming (OOP)**. The `rng` variable is not just a value; it's an *object* that bundles together data (its internal seed state) and methods that operate on that data (`.standard_normal()`). This is called **encapsulation**. This is a double-edged sword for reproducibility. On one hand, it's a huge improvement over hidden global state. On the other, the `rng` object itself is now a stateful entity. If we called `rng.standard_normal(5)` a second time, it would produce different numbers because its internal state would have been mutated by the first call.

In a purely functional world, we would avoid creating such stateful objects. However, in the pragmatic world of Python data science, this is often unavoidable. Core libraries like `pandas`, `scikit-learn`, and `matplotlib` are fundamentally object-oriented. You create `DataFrame` objects, model objects, and plot objects, all of which encapsulate state. Our guiding principle, therefore, must be one of careful management: **use functions for the flow and logic of your analysis, and treat objects from libraries as values that are passed between these functions**. Avoid building your own complex classes with hidden state for your data pipeline. A pipeline composed of functions (`df2 = clean_data(df1); df3 = analyze_data(df2)`) is almost always more transparent and reproducible than an object-oriented one (`pipeline.load(); pipeline.clean(); pipeline.analyze()`).

3.2.2 Can We Make This Truly Pure?

This naturally raises this next question: can we force this `numpy` example to be truly pure? A pure function cannot have side effects, which means it cannot mutate the `rng` object's internal

3 Functional Programming

state. To achieve this, our function must take the generator's current state as an explicit input and return a tuple containing both the desired random numbers **and** the new, updated state of the generator.

Let's write a wrapper function that does exactly this:

```
import numpy as np

def pure_standard_normal(generator_state,
    ↵ n_samples):
    """
    A pure function to generate standard normal
    ↵ random numbers.

    Args:
        generator_state: The current state of a
    ↵ numpy BitGenerator.
        n_samples: The number of samples to
    ↵ generate.

    Returns:
        A tuple containing (random_numbers,
    ↵ new_generator_state).
    """

    # 1. Create a temporary generator instance from
    ↵ the input state
    temp_rng =
    ↵ np.random.Generator(np.random.PCG64(generator_state))

    # 2. Generate the numbers (this mutates the
    ↵ *temporary* generator)
    numbers = temp_rng.standard_normal(n_samples)
```

```

# 3. Extract the new state from the temporary
    ↵ generator
new_state = temp_rng.bit_generator.state

# 4. Return both the result and the new state
return (numbers, new_state)

# --- How to use this pure function ---

# 1. Get an initial state from a seed
initial_state = np.random.PCG64(123).state

# 2. First call: provide the state, get back numbers
    ↵ and a *new* state
first_numbers, state_after_first_call =
    ↵ pure_standard_normal(initial_state, 5)
print("First call results:", first_numbers)

# 3. Second call: MUST use the new state from the
    ↵ previous call
second_numbers, state_after_second_call =
    ↵ pure_standard_normal(state_after_first_call, 5)
print("Second call results:", second_numbers)

# Proof of purity: If we re-use the initial state,
    ↵ we get the exact same "first" result
proof_numbers, _ =
    ↵ pure_standard_normal(initial_state, 5)
print("Proof call results:", proof_numbers)

```

As you can see, this is now 100% pure and predictable. The function `pure_standard_normal` will always produce the same output tuple for the same input tuple.

3 Functional Programming

3.2.2.1 Is This Feasible in Practice?

While this is a powerful demonstration of functional principles, it is often not practical for day-to-day data science in Python. Manually passing the `state` variable from one function to the next throughout an entire analysis script (`state_1`, `state_2`, `state_3...`) would be extremely verbose and cumbersome.

The key takeaway is understanding the trade-off. The object-oriented approach (`rng = np.random.default_rng(seed=123)`) is a pragmatic compromise. It encapsulates the state in a predictable way, which is a vast improvement over hidden global state, even if it's not technically “pure”. If you have to use Python: **treat stateful library objects like `rng` as values that are created once with a fixed seed and passed into your pure analysis functions.** This gives you 99% of the benefit of reproducibility with a fraction of the complexity.

This difference in the “feel” of functional composition between R’s pipe and Python’s method chaining is no accident; it reflects the deep-seated design philosophies of each language. This context is crucial for understanding why certain patterns feel more “natural” in each environment. R’s lineage traces back to the S language, which was itself heavily influenced by Scheme, a dialect of Lisp and a bastion of functional programming. Consequently, treating data operations as a series of function transformations is baked into R’s DNA. The entire Tidyverse ecosystem, with its ubiquitous pipe, is a modern implementation of this functional heritage.

Python, in contrast, was designed with a different set of priorities, famously summarized in its Zen: “There should be one—and preferably only one—obvious way to do it.” Its creator, Guido van Rossum, historically argued that explicit for loops and list comprehensions were more readable and “Pythonic”

than functional constructs like map and lambda. He was so committed to this principle of one clear path that he even proposed removing these functions from the language entirely at one point.

R is fundamentally a functional language that has acquired object-oriented features, while Python is a quintessential object-oriented language with powerful functional capabilities. Recognizing this history helps explain why a chain of functions feels native in R, while method chaining on objects is the default in pandas and polars. My goal in this course is for you to master the functional paradigm so you can apply it effectively in either language, leveraging the native strengths of each.

3.3 Writing Your Own Functions

Let's learn the syntax. The goal is always to encapsulate a single, logical piece of work.

3.3.0.1 In R

R functions are first-class citizens. You can assign them to variables and pass them to other functions.

```
# A simple function
calculate_ci <- function(x, level = 0.95) {
  # Calculate the mean and standard error
  se <- sd(x, na.rm = TRUE) / sqrt(length(x))
  mean_val <- mean(x, na.rm = TRUE)

  # Calculate the confidence interval bounds
```

3 Functional Programming

```
alpha <- 1 - level
lower <- mean_val - qnorm(1 - alpha/2) * se
upper <- mean_val + qnorm(1 - alpha/2) * se

# Return a named vector
# the `return()` statement is not needed at the
#   ↵ end
# but can be useful for early returning a result
c(mean = mean_val, lower = lower, upper = upper)
}

# Use it
data <- c(1.2, 1.5, 1.8, 1.3, 1.6, 1.7)
calculate_ci(data)
```

For data analysis, you'll often want to write functions that work with data frames and column names. The `{dplyr}` package uses a special technique called “tidy evaluation” for this.

```
library(dplyr)

# A function that summarizes a column in a dataset
summarize_variable <- function(dataset,
  ↵ var_to_summarize) {
  dataset %>%
    summarise(
      n = n(),
      mean = mean({{ var_to_summarize }}, na.rm =
        ↵ TRUE),
      sd = sd({{ var_to_summarize }}, na.rm = TRUE)
    )
}
```

```
# The {{ }} (curly-curly) syntax tells dplyr to use
#   the column name
# passed into the function.
starwars %>%
  group_by(species) %>%
  summarize_variable(height)
```

This is incredibly powerful for creating reusable analysis snippets. To learn more, read about programming with `{dplyr}` here.

3.3.0.2 In Python

Python's syntax is similar, using the `def` keyword. Type hints are a best practice for clarity.

```
import pandas as pd
import numpy as np

# A function to summarize a column in a DataFrame
def summarize_variable_py(dataset: pd.DataFrame,
                           var_to_summarize: str) -> pd.DataFrame:
    """Calculates summary statistics for a given
       column."""
    summary = dataset.groupby('species').agg(
        n=(var_to_summarize, 'size'),
        mean=(var_to_summarize, 'mean'),
        sd=(var_to_summarize, 'std'))
    .reset_index()
    return summary
```

3 Functional Programming

```
# Load data (assuming starwars.csv exists)
# starwars_py = pd.read_csv("starwars.csv")
# summarize_variable_py(starwars_py, 'height')
```

3.4 The Functional Toolkit: Map, Filter, and Reduce

Once you start thinking in functions, you'll notice common patterns emerge. Most `for` loops can be replaced by one of three core functional concepts: **mapping**, **filtering**, or **reducing**. These operations are handled by “higher-order functions”—functions that take other functions as arguments. Mastering them is key to writing elegant, declarative code.

3.4.1 1. Mapping: Applying a Function to Each Element

The pattern: You have a list of things, and you want to perform the same action on each element, producing a new list of the same length.

This is the most common replacement for a `for` loop. Instead of manually iterating and storing results, you just state your intent: “map this function over this list.”

3.4.1.1 In R with `purrr::map()`

The `{purrr}` package is the gold standard for functional programming in R. The `map()` family is its workhorse.

3.4 The Functional Toolkit: Map, Filter, and Reduce

- `map()`: Always returns a list.
- `map_dbl()`: Returns a vector of doubles (numeric).
- `map_chr()`: Returns a vector of characters (strings).
- `map_lgl()`: Returns a vector of logicals (booleans).
- `map_dfr()`: Returns a data frame by row-binding the results.

In base R, we have `lapply()`, `vapply()`, `apply()`, but the `{purrr}` functions provide a more homogenous interface.

Example: Calculate the mean of every column in a data frame.

```
library(purrr)

# The classic for-loop way (verbose and clunky)
# Allocate an empty vector with the right size
means_loop <- vector("double", ncol(mtcars))

for (i in seq_along(mtcars)) {
  means_loop[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

print(means_loop)

# The functional way with map_dbl()
means_functional <- map_dbl(mtcars, mean, na.rm =
  TRUE)

print(means_functional)
```

The `map()` version is not just shorter; it's safer. You can't make an off-by-one error, and you don't have to pre-allocate `means_loop`. The code clearly states its purpose.

3 Functional Programming

3.4.1.2 In Python with List Comprehensions and map()

Python's most idiomatic tool for mapping is the **list comprehension**, which we saw earlier. It's concise and highly readable.

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
# > [1, 4, 9, 16, 25]
```

Python also has a built-in `map()` function, which returns a “map object” (an iterator). You usually wrap it in `list()` to see the results. It’s most useful when you already have a function defined.

```
def to_upper_case(s: str) -> str:
    return s.upper()

words = ["hello", "world"]
upper_words = list(map(to_upper_case, words))
# > ['HELLO', 'WORLD']
```

3.4.2 2. Filtering: Keeping Elements That Match a Condition

The pattern: You have a list of things, and you want to keep only the elements that satisfy a certain condition. The condition is defined by a function that returns TRUE or FALSE.

3.4.2.1 In R with purrr::keep() or purrr::discard()

`keep()` retains elements where the function returns `TRUE`. `discard()` does the opposite. The base function is `Filter()`.

Example: From a list of data frames, keep only the ones with more than 100 rows.

```
# setup: create a list of data frames
df1 <- data.frame(x = 1:50)
df2 <- data.frame(x = 1:200)
df3 <- data.frame(x = 1:75)
list_of_dfs <- list(a = df1, b = df2, c = df3)

# The functional way to filter the list
large_dfs <- keep(list_of_dfs, ~ nrow(.x) > 100)
print(names(large_dfs))
```

3.4.2.2 In Python with List Comprehensions

List comprehensions have a built-in `if` clause that makes filtering incredibly natural.

```
numbers = [1, 10, 5, 20, 15, 30]

# Keep only numbers greater than 10
large_numbers = [n for n in numbers if n > 10]
# > [20, 15, 30]
```

Python also has a built-in `filter()` function, which, like `map()`, returns an iterator.

3 Functional Programming

```
def is_even(n: int) -> bool:  
    return n % 2 == 0  
  
numbers = [1, 2, 3, 4, 5, 6]  
even_numbers = list(filter(is_even, numbers))  
# > [2, 4, 6]
```

3.4.3 3. Reducing: Combining All Elements into a Single Value

The pattern: You have a list of things, and you want to iteratively combine them into a single summary value. You start with an initial value and repeatedly apply a function that takes the “current total” and the “next element.”

This is the most complex of the three but is powerful for things like summing, finding intersections, or joining a list of data frames.

3.4.3.1 In R with purrr::reduce()

Example: Find the total sum of a vector of numbers.

```
# reduce() will take the first two elements (1, 2),  
#   ↪ apply `+` to get 3.  
# Then it takes the result (3) and the next element  
#   ↪ (3), applies `+` to get 6.  
# And so on.  
total_sum <- reduce(c(1, 2, 3, 4, 5), `+`)  
  
# This is equivalent to 1 + 2 + 3 + 4 + 5
```

3.4 The Functional Toolkit: Map, Filter, and Reduce

```
print(total_sum)
```

The base R function is called `Reduce()`.

A more practical data science example: find all the column names that are common to a list of data frames.

```
# Get the column names of each df in the list
list_of_colnames <- map(list_of_dfs, names)
print(list_of_colnames)

# Use reduce with the `intersect` function to find
#   common elements
common_cols <- reduce(list_of_colnames, intersect)
print(common_cols)
```

3.4.3.2 In Python with `functools.reduce`

The `reduce` function was moved out of the built-ins and into the `functools` module in Python 3 because it's often less readable than an explicit `for` loop for simple operations like summing (well, at least according to Python users...). However, it's still the right tool for more complex iterative combinations.

```
from functools import reduce
import operator

numbers = [1, 2, 3, 4, 5]

# Use reduce with the addition operator to sum the
#   list
```

3 Functional Programming

```
total_sum_py = reduce(operator.add, numbers)
# > 15

# You can also use a lambda function
total_product = reduce(lambda x, y: x * y, numbers)
# > 120
```

3.5 The Power of Composition

The final, beautiful consequence of a functional style is **composition**. You can chain functions together to build complex workflows from simple, reusable parts. This is exactly what the pipe operators (`|>` in R, `%>%` from `{magrittr}`) and method chaining (the `.` in pandas) are designed for.

This R code is a sequence of function compositions:

```
starwars %>%
  filter(!is.na(mass)) %>%
  select(species, sex, mass) %>%
  group_by(sex, species) %>%
  summarise(mean_mass = mean(mass), .groups =
    ~ "drop")
```

This is equivalent to `summarise(group_by(select(filter(starwars, ...))))`. The pipe makes it readable.

The same idea applies in Python with pandas:

```
# (starwars_py
#   .dropna(subset=['mass'])
#   .filter(items=['species', 'sex', 'mass'])
#   .groupby(['sex', 'species'])
#   ['mass'].mean()
#   .reset_index()
# )
```

The issue with *method chaining* though, is that this only works within the methods that are available for `pandas.DataFrame` objects. You could apply another function using `pandas.DataFrame.apply()` function, but you can't pipe functions from different packages like you could in R (more on this in the next subsection).

Each step is a function that takes a data frame and returns a new, transformed data frame. By combining `map`, `filter`, and `reduce` with this compositional style, you can express complex data manipulation pipelines without writing a single `for` loop. This makes your code more declarative, less prone to bugs, and easier to reason about—a perfect fit for a reproducible workflow.

3.5.1 The Challenge of Composition in Python

The difficulty of function composition in Python, as you've noted, stems from fundamental differences in its primary programming paradigm compared to R. While both languages are powerful and versatile, their core designs influence how naturally they support the seamless chaining of functions. The crux of the matter lies in Python's primarily encapsulated

3 Functional Programming

Object-Oriented Programming (OOP) versus R’s functional OOP and its use of polymorphic functions.

3.5.1.1 Python’s Encapsulated OOP: Methods Belong to Objects

As already mentioned earlier in the chapter, Python is predominantly an object-oriented language where data and the functions that operate on that data are bundled together into objects. This concept is known as **encapsulation**. A class defines the blueprint for an object, and the functions defined within a class are called methods. These methods are intrinsically tied to the object’s class and are typically invoked using dot notation (.), as seen in the `pandas` example.

This tight coupling of methods to specific object types is the main reason why fluid composition can be challenging. Method chaining, while elegant, is limited to the methods that have been explicitly defined for a particular class. To apply a function from a different library or a user-defined function that isn’t a method of the object, you often need to use workarounds like `apply()` in `pandas`, which can break the intuitive flow of a composition chain.

Furthermore, while Python supports functional programming concepts, they are not always as central to the language’s design as its OOP features. For instance, `lambda` functions in Python are limited to a single expression, which can make defining complex on-the-fly functions cumbersome.

3.5.1.2 R's Functional OOP: Functions are Polymorphic and Independent

In contrast, R was designed with a strong emphasis on functional programming. Its approach to object-orientation is described as **functional OOP**. In this paradigm, methods are not encapsulated within class definitions. Instead, functions are often “generic,” meaning they can behave differently depending on the class of the object passed to them. This is a form of **polymorphism**.

This design choice has a profound impact on composition. Because functions are not strictly owned by objects, they can be more easily and flexibly combined. The pipe operators in R (`|>` and `%>%`) are a testament to this, allowing for the creation of highly readable and complex data manipulation pipelines by passing data through a series of independent functions. Each function takes data as an input and returns a transformed version, which is then passed to the next function in the chain.

3.5.1.3 The Core Distinction: “Has-a” vs. “Is-a” and its Impact on Composition

The principle of “favor composition over inheritance” is a well-known software design guideline. Inheritance models an “is-a” relationship (a Dog is an `Animal`), while composition models a “has-a” relationship (a Car has an `Engine`).

- **Python's encapsulated OOP** often encourages the use of inheritance, where a class inherits methods from a parent class. While powerful, this can lead to rigid hierarchies.

3 Functional Programming

- R’s functional approach naturally favors a compositional style. You build complex operations by combining simpler, single-purpose functions. This aligns well with the “has-a” model, where a data analysis pipeline “has a” filtering step, a selection step, and a summarization step.

In essence, Python’s strength lies in creating well-defined, encapsulated objects with specific behaviors. R’s strength, particularly in data analysis, is in its ability to fluidly combine and apply functions to data. This makes the “compositional style” a more natural fit for the R ecosystem. While Python can achieve similar results, it often requires more deliberate effort to break out of the strict object-method paradigm to achieve the same level of compositional freedom.

3.6 Conclusion: Functions as the Bedrock of Reproducibility

This chapter has laid the groundwork for the most critical pillar of reproducible data science: writing reproducible code. We have moved beyond the ephemeral, state-dependent nature of scripts and computational notebooks to embrace the discipline and predictability of **Functional Programming**.

By treating functions as our primary unit of work, we unlock a cascade of benefits. **Pure functions**, which guarantee the same output for the same input, form the core of this approach. They are transparent, easy to reason about, and, most importantly, **testable**. When we encounter inherently “impure” operations like random number generation, we’ve learned to control the impurity by making the source of impurity (in this example the

random seed) an explicit and managed input, rather than a hidden global state.

We've replaced verbose and error-prone `for` loops with the powerful functional trio of **map**, **filter**, and **reduce**. These higher-order functions allow us to express complex data manipulations declaratively, stating *what* we want to do rather than detailing *how* to do it. This leads to code that is not only more concise but also less prone to bugs.

Finally, we explored **composition**, the elegant chaining of these simple functions to build sophisticated analysis pipelines. We saw how this concept manifests differently in R and Python, a direct reflection of their core design philosophies. R's functional heritage makes composition via the pipe (`%>%` or `|>`) a natural and seamless experience. Python's object-oriented nature favors method chaining on objects like pandas DataFrames, a powerful but more constrained form of composition.

Understanding this distinction is key to becoming an effective data scientist in any language. By mastering the functional paradigm, you are not just learning a new coding style; you are adopting a new way of thinking. You are building a foundation for code that is robust, easy to review, simple to debug, and truly reproducible—the ultimate goal of any serious analytical project.

3.7 Exercises

The following exercises will help you solidify your understanding of functional programming concepts in both R and Python. Use built-in datasets like `iris` or `mtcars` for R, and you can load them into pandas DataFrames for the Python exercises.

3.7.1 1. From Impure to Pure

Goal: Refactor a function that relies on global state into a pure function.

- **R:** The following function filters `mtcars` to find cars with a miles-per-gallon (MPG) above a certain threshold, but the threshold is a global variable. Rewrite it so that it becomes a pure function.

```
# Impure function
mpg_threshold <- 20

get_efficient_cars_impure <- function(dataset)
  {
    dplyr::filter(dataset, mpg > mpg_threshold)
  }

# Your task: Create a pure function
`get_efficient_cars_pure` # that takes the dataset and the threshold as
# arguments.
# Then, call it with a threshold of 25.
```

- **Python:** Do the same for the Python equivalent. Rewrite the impure function into a pure one (you can find the `mtcars` dataset as a csv here).

```
import pandas as pd
mtcars = pd.read_csv("path/to/mtcars.csv")

# Impure function
mpg_threshold <- 20

def get_efficient_cars_impure(df):
```

```

    return df[df['mpg'] > mpg_threshold]

# Your task: Create a pure function
    ↵ `get_efficient_cars_pure`
# that takes the DataFrame and the threshold as
    ↵ arguments.
# Then, call it with a threshold of 25.

```

3.7.2 2. Mapping

Goal: Use mapping to apply a function to multiple elements of a list or data frame.

- **R:** Using the `iris` dataset, calculate the number of distinct values for each of its four numeric columns (`Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`).
- **Hint:** Use `purrr::map_int()` and the `n_distinct()` function from `dplyr`.
- **Python:** You are given a list of strings. Use a list comprehension to create a new list containing the length of each string.

```

words = ["functional", "programming", "is",
    ↵ "powerful"]
# Your task: create a list `word_lengths`  

    ↵ containing [10, 11, 2, 8]

```

3 Functional Programming

3.7.2.1 3. Filtering

Goal: Use filtering to select elements from a list based on a condition.

- **R:** You have a list of vectors. Use `purrr::keep()` to select only the vectors whose mean is greater than 5.

```
list_of_vectors <- list(  
  a = c(1, 2, 9),  
  b = c(8, 8, 9),  
  c = c(1, 1, 2)  
)  
# Your task: create a list `high_mean_vectors`  
  ↵  that contains only vectors a and b.
```

- **Python:** You have a list of dictionaries, where each dictionary represents a product. Use a list comprehension with an if clause to filter for products that are on sale.

```
products = [  
  {'name': 'Laptop', 'price': 1200,  
   ↵  'on_sale': False},  
  {'name': 'Mouse', 'price': 25, 'on_sale':  
   ↵  True},  
  {'name': 'Keyboard', 'price': 75,  
   ↵  'on_sale': True}  
]  
# Your task: create a list `sale_items`  
  ↵  containing only the mouse and keyboard  
  ↵  dicts.
```

3.7.2.2 4. Reducing

Goal: Use a reduce operation to aggregate a list into a single value.

- **R:** You are given three small data frames. Use `purrr::reduce()` and a `dplyr::full_join()` to combine them into a single data frame.

```
df1 <- data.frame(id = c("a", "b"), val1 = c(1,
    ↵ 2))
df2 <- data.frame(id = c("a", "c"), val2 = c(3,
    ↵ 4))
df3 <- data.frame(id = c("b", "c"), val3 = c(5,
    ↵ 6))
list_of_dfs <- list(df1, df2, df3)

# Your task: use reduce to join them all by the
    ↵ 'id' column.
```

- **Python:** Given a list of lists (a 2D matrix), use `functools.reduce` to “flatten” it into a single list.

```
from functools import reduce
matrix = [[1, 2, 3], [4, 5], [6]]
# Your task: use reduce to produce the list [1,
    ↵ 2, 3, 4, 5, 6]
# Hint: `operator.add` can concatenate lists.
```

3.7.2.3 5. Composition Challenge: A Reusable Analysis Function

Goal: Write a reusable function that encapsulates a common data analysis task.

3 Functional Programming

- **R:** Write a single function named `summarize_by_group` that takes three arguments: a data frame (`dataset`), a categorical column to group by (`grouping_var`), and a numeric column to summarize (`summary_var`). The function should return a summarized data frame with the count, mean, and standard deviation for the `summary_var` within each group. Use `dplyr` and the `{ }` syntax. Test it on the `iris` dataset by summarizing `Sepal.Length` grouped by `Species`.
- **Python:** Write the equivalent function in Python named `summarize_by_group_py`. It should take a pandas DataFrame, a `grouping_var` name (string), and a `summary_var` name (string). Use `.groupby()` and `.agg()` to produce the same summary table (count, mean, sd). Test it on the penguins dataset by summarizing `body_mass_g` grouped by `species`.

4 Unit Testing: The Safety Net for Your Code



What you'll learn by the end of this chapter:

- What unit tests are and why they are essential for reliable data analysis.
- How to write and run unit tests for your functions in both R (with `{testthat}`) and Python (with `pytest`).

- How to use testing to improve the design and robustness of your code.
- How to leverage LLMs to accelerate test writing and embrace your role as a code reviewer.

4.1 Introduction: Proving Your Code Works

I hope you are starting to see the pieces of our reproducible workflow coming together. We now have:

1. **Reproducible Environments (Nix):** The correct tools for everyone.
2. **Reproducible History (Git):** The correct version of the code for everyone.
3. **Reproducible Logic (Functional Programming):** A philosophy for writing clean, predictable, and self-contained code.

This brings us to the final, crucial question: **How do we *prove* that our functions actually do what we claim they do?**

The answer is **unit testing**. A unit test is a piece of code whose sole job is to check that another piece of code, a “unit”, works correctly. In our functional world, the “unit” is almost always a single function. This is why we spent so much time on FP in the previous chapter. Small, pure functions are not just easy to reason about; they are incredibly easy to test.

Writing tests is your contract with your collaborators and your future self. It’s a formal promise that your function, `calculate_mean_mpg()`, given a specific input, will always produce a specific, correct output. It’s the safety net that

catches bugs before they make it into your final analysis and the tool that gives you the confidence to refactor and improve your code without breaking it.

4.2 The Philosophy of a Good Unit Test

So, what should we test? Writing good tests is a skill, but it revolves around answering a few key questions about your function. For any function you write, you should have tests that cover:

- **The “Happy Path”:** Does the function return the expected, correct value for a typical, valid input?
- **Bad Inputs:** Does the function fail gracefully or throw an informative error when given garbage input (e.g., a string instead of a number, a data frame with the wrong columns)?
- **Edge Cases:** How does the function handle tricky but valid inputs? For example, what happens if it receives an empty data frame, a vector with `NA` values, or a vector where all the numbers are the same?

Writing tests forces you to think through these scenarios, and in doing so, almost always leads you to write more robust and well-designed functions.

4.3 Unit Testing in Practice

Let’s imagine we’ve written a simple helper function to normalize a numeric vector (i.e., scale it to have a mean of 0 and a

4 Unit Testing: The Safety Net for Your Code

standard deviation of 1). We'll save this in a file named `utils.R` or `utils.py`.

R version (`utils.R`):

```
normalize_vector <- function(x) {  
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)  
}
```

Python version (`utils.py`):

```
import numpy as np  
  
def normalize_vector(x):  
    return (x - np.nanmean(x)) / np.nanstd(x)
```

Now, let's write tests for it.

4.3.1 Testing in R with `{testthat}`

In R, the standard for unit testing is the `{testthat}` package. The convention is to create a `tests/testthat/` directory in your project, and for a script `utils.R`, you would create a test file named `test-utils.R`.

Inside `test-utils.R`, we use the `test_that()` function to group related expectations.

```
# In file: tests/testthat/test-utils.R  
  
# First, we need to load the function we want to  
#   test
```

```
source("../utils.R")

library(testthat)

test_that("Normalization works on a simple vector
         ↳ (the happy path)", {
  # 1. Setup: Create input and expected output
  input_vector <- c(10, 20, 30)
  expected_output <- c(-1, 0, 1)

  # 2. Action: Run the function
  actual_output <- normalize_vector(input_vector)

  # 3. Expectation: Check if the actual output
  #    ↳ matches the expected output
  expect_equal(actual_output, expected_output)
})

test_that("Normalization handles NA values
         ↳ correctly", {
  input_with_na <- c(10, 20, 30, NA)
  expected_output <- c(-1, 0, 1, NA)

  actual_output <- normalize_vector(input_with_na)

  # We need to use expect_equal because it knows how
  #    ↳ to compare NAs
  expect_equal(actual_output, expected_output)
})
```

The `expect_equal()` function checks for near-exact equality. `{testthat}` has many other `expect_*`() functions, like `expect_error()` to check that a function fails correctly, or

4 Unit Testing: The Safety Net for Your Code

`expect_warning()` to check for warnings.

4.3.2 Testing in Python with pytest

In Python, the de facto standard is `pytest`. It's incredibly simple and powerful. The convention is to create a `tests/` directory, and your test files should be named `test_*.py`. Inside, you just write functions whose names start with `test_` and use Python's standard `assert` keyword.

```
# In file: tests/test_utils.py

import numpy as np
from utils import normalize_vector # Import our
    ↵ function

def test_normalize_vector_happy_path():
    # 1. Setup
    input_vector = np.array([10, 20, 30])
    expected_output = np.array([-1.0, 0.0, 1.0])

    # 2. Action
    actual_output = normalize_vector(input_vector)

    # 3. Expectation
    # For floating point numbers, it's better to
    ↵ check for "close enough"
    assert np.allclose(actual_output,
        ↵ expected_output)

def test_normalize_vector_with_nas():
    input_with_na = np.array([10, 20, 30, np.nan])
    expected_output = np.array([-1.0, 0.0, 1.0,
        ↵ np.nan])
```

```
actual_output = normalize_vector(input_with_na)

# `np.allclose` doesn't handle NaNs, but
#   `np.testing.assert_allclose` does!
np.testing.assert_allclose(actual_output,
                           expected_output)
```

Because this isn't a package though, but a simple project with scripts, you also need to create another file called `pytest.ini`, which will tell `pytest` where to find the tests:

```
[pytest]
# Discover tests in the tests/ directory
testpaths = tests/

# Include default patterns and your hyphenated
#   pattern
python_files = test_*.py *_test.py test-*.py

# Adds the root directory to the pythonpath, without
#   this
# it'll be impossible to import normalize_vector()
#   from
# utils.py
pythonpath = .
```

To run your tests, you simply navigate to your project's root directory in the terminal and run the command `pytest`. It will automatically discover and run all your tests for you. That being said, you will get an error:

4 Unit Testing: The Safety Net for Your Code

```
>         assert np.allclose(actual_output,
  ↵  expected_output)
E         assert False
E           +   where False = <function allclose at
  ↵  0x7f0e81c959f0>(array([-1.22474487,  0.
  ↵  1.22474487]), array([-1.,  0.,  1.])    def
  ↵  test_normalize_vector_with_nas():
      input_with_na = np.array([10, 20, 30,
  ↵  np.nan])
      expected_output = np.array([-1.0, 0.0, 1.0,
  ↵  np.nan])

      actual_output =
  ↵  normalize_vector(input_with_na)

      # `np.allclose` doesn't handle NaNs, but
      ↵  `np.testing.assert_allclose` does!
>      np.testing.assert_allclose(actual_output,
  ↵  expected_output)
E      AssertionError:
E      Not equal to tolerance rtol=1e-07, atol=0
E
E      Mismatched elements: 2 / 4 (50%)
E      Max absolute difference among violations:
  ↵  0.22474487
E      Max relative difference among violations:
  ↵  0.22474487
E      ACTUAL: array([-1.224745,  0.,
  ↵  1.224745,        nan])
E      DESIRED: array([-1.,  0.,  1., nan])

tests/test_utils.py:25: AssertionError
=====
  ↵  short test summary info
  ↵  =====
```

```

FAILED
↳ tests/test_utils.py::test_normalize_vector_happy_path
↳ - assert False
FAILED
↳ tests/test_utils.py::test_normalize_vector_with_nas
↳ - AssertionError:
=====
↳ 2 failed in 0.15s
↳ =====

```

You don't encounter this issue in R and understanding why reveals how valuable unit tests can be. (Hint: how does the implementation of the `sd` function, which computes the standard deviation, differ between R and NumPy?)

4.4 Testing as a Design Tool

Testing can also help you with programming, by thinking about edge cases. For example, what happens if we try to normalize a vector where all the elements are the same?

Let's write a test for this edge case first.

`pytest` version:

```

# tests/test_utils.py
def test_normalize_vector_with_zero_std():
    input_vector = np.array([5, 5, 5, 5])
    actual_output = normalize_vector(input_vector)
    # The current function will return `[nan, nan,
    ↳   nan, nan]`
```

4 Unit Testing: The Safety Net for Your Code

```
# Let's assert that we expect a vector of zeros
    ↵ instead.
assert np.allclose(actual_output, np.array([0,
    ↵ 0, 0, 0]))
```

If we run `pytest` now, this test will **fail**. Our test has just revealed a flaw in our function's design. This process is a core part of **Test-Driven Development (TDD)**: write a failing test, then write the code to make it pass.

Let's improve our function:

Improved Python version (`utils.py`):

```
import numpy as np

def normalize_vector(x):
    std_dev = np.nanstd(x)
    if std_dev == 0:
        # If std is 0, all elements are the mean. Return
        ↵ a vector of zeros.
        return np.zeros_like(x, dtype=float)
    return (x - np.nanmean(x)) / std_dev
```

Now, if we run `pytest` again, our new test will pass. We used testing not just to verify our code, but to actively make it more robust and thoughtful.

4.5 The Modern Data Scientist’s Role: Reviewer and AI Collaborator

In the past, writing tests was often seen as a chore. Today, LLMs make this process very easy.

4.5.1 Using LLMs to Write Tests

LLMs are fantastic at writing unit tests. They are good at handling boilerplate code and thinking of edge cases. You can provide your function to an LLM and give it a prompt like this:

Prompt: “Here is my Python function `normalize_vector`. Please write three `pytest` unit tests for it. Include a test for the happy path with a simple array, a test for an array containing `np.nan`, and a test for the edge case where all elements in the array are identical.”

The LLM will likely generate high-quality test code that is very similar to what we wrote above. This is a massive productivity boost. However, this introduces a new, critical role for the data scientist: **you are the reviewer**.

An LLM does not *write* your tests; it *generates a draft*. It is your professional responsibility to:

1. **Read and understand** every line of the test code.
2. **Verify** that the `expected_output` is actually correct.
3. **Confirm** that the tests cover the cases you care about.
4. **Commit** that code under your name, taking full ownership of it.

“A COMPUTER CAN NEVER BE HELD ACCOUNTABLE THEREFORE A COMPUTER MUST NEVER MAKE A MANAGEMENT DECISION” – IBM Training Manual, 1979.

If I ask you why you did something, and your answer is something to the effect of “I dunno, the LLM generated it”, be glad we’re not in the USA where I could just fire you, because that’s what I’d do.

4.5.2 Testing and Code Review

This role as a reviewer is central to modern collaborative data science. When a teammate (or your future self) submits a Pull Request on GitHub, the tests are your first line of defense. A PR that changes logic but doesn’t update the tests is a major red flag. A PR that adds a new feature without adding any tests should be rejected until tests are included.

Even as a junior member of a team, one of the most valuable contributions you can make during a code review is to ask: “This looks great, but what happens if the input is NA? Could we add a test for that case?” This moves the quality of the entire project forward.

By embracing testing, you are not just writing better code; you are becoming a better collaborator and a more responsible data scientist.

4.5.3 A Note on Packaging and Project Structure

Throughout this chapter, we’ve focused on testing individual functions within a simple project structure (`utils.R` and

`tests/test-utils.R`). This is the fundamental skill. It’s important to recognize, however, that this entire process becomes even more streamlined and robust when your code is organized into a formal **package**.

Packaging your code provides a standardized structure for your functions, documentation, and tests. It solves many logistical problems automatically: testing frameworks know exactly where to find your source code without needing manual `source()` or `from utils import ...` statements, and tools can easily run all tests with a single command. It also makes your code installable, versionable, and distributable, which is the ultimate form of reproducibility.

In chapter 7, we will learn some packaging basics for Python and R.

4.5.4 Hands-On Exercises

For these exercises, create a project directory with a `tests/` subdirectory. Place your function code in a script in the root directory (e.g., `my_functions.R` or `my_functions.py`) and your test code inside the `tests/` directory (e.g., `tests/test_my_functions.R` or `tests/test_my_functions.py`).

4.5.4.1 Exercise 1: Testing the “Happy Path”

The median of a list of numbers is a common calculation. However, the logic is slightly different depending on whether the list has an odd or even number of elements. Your task is to test both of these “happy paths.”

Here is the function in R and Python.

4 Unit Testing: The Safety Net for Your Code

R (`my_functions.R`):

```
calculate_median <- function(x) {  
  sorted_x <- sort(x)  
  n <- length(sorted_x)  
  mid <- floor(n / 2)  
  
  if (n %% 2 == 1) {  
    # Odd number of elements  
    return(sorted_x[mid + 1])  
  } else {  
    # Even number of elements  
    return(mean(c(sorted_x[mid], sorted_x[mid +  
      ↴ 1])))  
  }  
}
```

Python (`my_functions.py`):

```
import numpy as np  
  
def calculate_median(x):  
  sorted_x = np.sort(np.array(x))  
  n = len(sorted_x)  
  mid = n // 2  
  
  if n % 2 == 1:  
    # Odd number of elements  
    return sorted_x[mid]  
  else:  
    # Even number of elements  
    return (sorted_x[mid - 1] + sorted_x[mid]) / 2.0
```

Your Task:

1. Create a test file (`test-my_functions.R` or `tests/test_my_function.R`)
2. Write a test that checks if `calculate_median` gives the correct result for a vector with an **odd** number of elements (e.g., `c(10, 20, 40)`).
3. Write a second test that checks if `calculate_median` gives the correct result for a vector with an **even** number of elements (e.g., `[1, 2, 8, 10]`).

4.5.4.2 Exercise 2: Testing Edge Cases and Expected Errors

The geometric mean is another way to calculate an average, but it has strict requirements: it only works with non-negative numbers. This makes it a great candidate for testing edge cases and expected failures.

R (`my_functions.R`):

```
calculate_geometric_mean <- function(x) {  
  if (any(x < 0)) {  
    stop("Geometric mean is not defined for negative  
        numbers.")  
  }  
  return(prod(x)^(1 / length(x)))  
}
```

Python (`my_functions.py`):

```
import numpy as np

def calculate_geometric_mean(x):
    if np.any(np.array(x) < 0):
        raise ValueError("Geometric mean is not defined
                         for negative numbers.")
    return np.prod(x)**(1 / len(x))
```

Your Task:

Write three tests for this function:

1. A “happy path” test with a simple vector of positive numbers (e.g., `c(1, 2, 4)`) should result in 2).
2. An **edge case** test for a vector that includes 0. The expected result should be 0.
3. An **error test** that confirms the function fails correctly when given a vector with a negative number.
 - In R, use `testthat::expect_error()`.
 - In Python, use `pytest.raises()`. Example: `with pytest.raises(ValueError): your_function_call()`

4.5.4.3 Exercise 3: Test-Driven Development (in miniature)

Testing can help you design better functions. Here is a simple function that is slightly flawed. Your task is to use testing to find the flaw and fix it.

R (`my_functions.R`):

```
# Initial flawed version
find_longest_string <- function(string_vector) {
  # This will break on an empty vector!
  string_vector[which.max(nchar(string_vector))]
}
```

Python (`my_functions.py`):

```
# Initial flawed version
def find_longest_string(string_list):
  # This will break on an empty list!
  return max(string_list, key=len)
```

Your Task:

1. **Part A:** Write a simple test to prove the function works for a standard case (e.g., `c("a", "b", "abc")`) should return `"abc"`.
2. **Part B:** Write a new test for an **empty input** (`c()` or `[]`). Run your tests. **This test should fail with an error.**
3. **Part C:** Modify the original `find_longest_string` function in your source file to handle the empty input gracefully (e.g., it could return `NULL` in R, or `None` in Python).
4. Run your tests again. Now all tests should pass. You have just completed a mini-cycle of Test-Driven Development!

4.5.4.4 Exercise 4: The AI Collaborator

One of the most powerful uses of LLMs is to accelerate the creation of tests. Your job is to act as the senior reviewer for the code an LLM generates.

4 Unit Testing: The Safety Net for Your Code

Here is a simple data cleaning function in Python.

Python (`my_functions.py`):

```
import pandas as pd

def clean_sales_data(df: pd.DataFrame) ->
    pd.DataFrame:
    """
    Cleans a raw sales DataFrame.
    - Renames 'ts' column to 'timestamp'.
    - Converts 'timestamp' column to datetime objects.
    - Ensures 'sale_value' is a numeric type.
    """
    if 'ts' not in df.columns:
        raise KeyError("Input DataFrame must contain a
                       'ts' column.")

    df = df.rename(columns={'ts': 'timestamp'})
    df['timestamp'] = pd.to_datetime(df['timestamp'])
    df['sale_value'] = pd.to_numeric(df['sale_value'])
    return df
```

Your Task:

- 1. Prompt your LLM:** Copy the function above and give your LLM a prompt like this: > “You are a helpful assistant writing tests for a Python data science project. Here is a function. Please write a `pytest` test file for it. Include a test for the happy path where everything works correctly. Also, include a test that verifies the function raises a `KeyError` if the ‘ts’ column is missing.”
- 2. Act as the Reviewer:**

- Create a new test file (`tests/test_data_cleaning.py`) and paste the LLM’s response.
- Read every line of the generated test code. Is the logic correct? Is the `expected_output` data frame what you would actually expect?
- Run the tests using `pytest`. Do they pass? If not, debug and fix them. It is your responsibility to ensure the final committed code is correct.
- Add a comment at the top of the test file describing one thing the LLM did well and one thing you had to change or fix (e.g., `# LLM correctly set up the test for the KeyError, but I had to correct the expected data type in the happy path test.`).

5 A Short Intro to Packaging Your Code in R and Python



What you'll learn by the end of this chapter:

- Why organizing your code into a formal package is the ultimate form of reproducibility and reusability.

- How to create, document, and test a basic R package using `{devtools}` and `{usethis}`.
- How to create, document, and test a modern Python package using `uv` and `pytest`.
- How to install your own packages directly from GitHub, enabling you to share your tools with colleagues and your future self.

5.1 Introduction: Why Bother Packaging?

So far, we have built a robust workflow based on three pillars: reproducible environments with Nix, reproducible history with Git, and reproducible logic with functional programming. We've organized our code into functions, which are a massive improvement over messy scripts.

The final, logical step in this journey is to treat our collection of functions not just as a set of helper scripts, but as a formal **package**. A package is more than just a folder of code; it's a self-contained, distributable, and installable unit of software that bundles together code, data, documentation, and tests.

You might think, “I’m a data scientist, not a software engineer. Isn’t this overkill?” The answer is a definitive **no**. Packaging your code, even for an internal analysis project, provides enormous benefits:

1. **Reusability:** Instead of copying and pasting your `clean_data()` function from project to project, you can simply `import mypackage` or `library(mypackage)` and use a single, trusted version.

5.2 Part 1: Creating an R Package with `{usethis}` and `{devtools}`

2. **Distribution & Collaboration:** How do you share your work with a colleague? Emailing a zip file of scripts is a recipe for disaster. Sending them a single command—`devtools::install_github("my_repo")`—is robust and professional.
3. **Documentation:** Packaging forces you into a standardized way of documenting your functions. This makes your code understandable to others and, more importantly, to yourself six months from now.
4. **Testing:** A package provides a formal framework for running unit tests, ensuring that your functions work as expected and giving you the confidence to make changes without breaking things.
5. **Dependency Management:** A package explicitly declares all of its dependencies (e.g., “this package needs `dplyr` version 1.1.0 or newer”). This solves a huge source of reproducibility errors.

In this chapter, we will walk through the process of creating a simple package in both R and Python. The goal is not to become an expert package developer, but to understand the structure and benefits so you can apply this powerful “packaging mindset” to all your future projects.

5.2 Part 1: Creating an R Package with `{usethis}` and `{devtools}`

The R community has developed an outstanding set of tools that make package development incredibly streamlined. The two essential packages are:

- **{devtools}**: Provides core development tools like `install()`, `test()`, and `check()`.
- **{usethis}**: A workflow package that automates all the boilerplate. It creates files, sets up infrastructure, and guides you through the process.

Let's build a package called `cleanR`, which will contain a function to standardize column names. Create folder called `cleanR` and `cd` into it.

5.2.1 Step 1: Project Setup

First, make sure you have the necessary tools installed. Create a Nix environment that contains the following R packages: `{devtools}`, `{usethis}` and `{roxygen2}`.

Drop into this Nix shell, and let `{usethis}` create the package structure for you. From your R console, run:

```
usethis::create_package("~/Documents/projects/cleanR")
```

This will create a new `cleanR` directory with all the necessary files and subdirectories. It will also open a new RStudio session for that project. The key components are:

- **R/**: This is where your R source code files will live.
- **DESCRIPTION**: A metadata file describing your package, its author, license, and dependencies.
- **NAMESPACE**: A file that declares which functions your package exports for users and which functions it imports from other packages. **You should never edit this file by hand.** `{roxygen2}` will manage it for you.

5.2.2 Step 2: Write and Document a Function

Let's create our function. `{usethis}` helps with this too:

```
usethis::use_r("clean_names")
```

This creates a new file `R/clean_names.R` and opens it for editing. Let's add our function, including special comments for documentation. These `#'` comments are used by the `{roxygen2}` package to automatically generate the official documentation.

```
# In R/clean_names.R

#' Clean and Standardize Column Names
#'
#' This function takes a data frame and returns a
#<-- new data frame with
#' cleaned-up column names (lowercase, with
#<-- underscores instead of spaces
#' or periods).
#'
#' @param df A data frame.
#' @return A data frame with standardized column
#<-- names.
#' @export
#' @examples
#' messy_df <- data.frame("First Name" = c("Ada",
#<-- "Bob"), "Last.Name" = c("Lovelace", "Ross"))
#' clean_names(messy_df)
clean_names <- function(df) {
  old_names <- names(df)
  new_names <- tolower(old_names)
  new_names <- gsub("[ .]", "_", new_names)
```

```
  names(df) <- new_names  
  return(df)  
}
```

The key tags here are:

- `@param`: Describes a function argument.
- `@return`: Describes what the function returns.
- `@export`: This is crucial. It tells R that you want this function to be available to users when they load your package with `library(cleanR)`.
- `@examples`: Provides runnable examples that will appear in the help file.

Now, run the magic command to process these comments:

```
devtools::document()
```

This updates the NAMESPACE file and creates the help file (`man/clean_names.Rd`). You can now see your function's help page with `?clean_names`.

5.2.3 Step 3: Add Unit Tests

A package without tests is a package waiting to break. `{usethis}` makes setting up tests trivial.

```
usethis::use_testthat() # Sets up the  
  ↵ tests/testthat/ directory  
usethis::use_test("clean_names") # Creates  
  ↵ tests/testthat/test-clean_names.R
```

5.2 Part 1: Creating an R Package with `{usethis}` and `{devtools}`

Now, edit the test file to add your expectations.

```
# In tests/testthat/test-clean_names.R
test_that("clean_names works with spaces and
  ↵ periods", {
  messy_df <- data.frame("First Name" = c("A"),
  ↵ "Last.Name" = c("B"))
  cleaned_df <- clean_names(messy_df)

  expected_names <- c("first_name", "last_name")

  expect_equal(names(cleaned_df), expected_names)
})

test_that("clean_names handles already clean names",
  ↵ {
  clean_df <- data.frame(a = 1, b = 2)
  # The function should not change anything
  expect_equal(names(clean_names(clean_df)), c("a",
  ↵ "b"))
})
```

To run all the tests for your package, use:

```
devtools::test()
```

5.2.4 Step 4: Check and Install

The final step before sharing is to run the official R CMD check, the gold standard for package quality. This command runs all tests, checks documentation, and looks for common problems.

```
devtools::check()
```

If your package passes with 0 errors, 0 warnings, and 0 notes, you are in great shape. If your package raises NOTES or WARNINGS during the check phase, you can *most of the time* safely ignore these, especially if the package is only intended for internal usage. However, I would recommend that you still take care of the WARNINGS at the very least.

As a next step, you could edit the DESCRIPTION file. This is where you will list yourself as the package author, list the dependencies of the package and so on. I won't get into detail here, but learning how to edit the DESCRIPTION file is important for actual package development (especially listing dependencies is key).

To use your package within a project, the simplest way is to host it on GitHub or build a .tar.gz file and install it locally.

5.2.5 Step 5: Install from GitHub

If you can publicly host your package, hosting it on GitHub is a good way to easily share your code, and install the package in your projects without needed to publish it on CRAN.

1. Create a new, empty repository on GitHub (e.g., `cleanR`).
2. In your local project, follow the instructions GitHub provides to link your local repository and push your code. This usually involves commands like:

5.2 Part 1: Creating an R Package with `{usethis}` and `{devtools}`

```
git remote add origin
  ↳ git@github.com:yourusername/cleanR.git
git branch -M main
git push -u origin main
```

3. Now, anyone (including you on a different machine) can install your package with a single command (**if you don't use Nix**):

```
# You might need to install {remotes} first
# install.packages("remotes")
remotes::install_github("yourusername/cleanR")
```

If you want to create an environment using Nix that includes this package, use the `git_pkgs` argument of `rix::rix()` to generate the right `default.nix` file.

Congratulations, you have created and shared a fully functional R package!

5.2.6 Step 5bis: Install it locally

If you can't share your package on GitHub, the alternative is to build it locally using:

```
devtools::build()
```

which will create a `.tar.gz` package. You can then install it using either `devtools::install_local()` if you don't use Nix or the `local_r_pkgs` argument of `rix::rix()`.

5.3 Part 2: Creating a Minimal Python Package with uv

The Python packaging ecosystem is rapidly modernizing. While we use Nix to manage our overall environment, we still need to define the metadata and structure for our Python package. We will use **uv**, an extremely fast and modern tool, for one specific purpose: initializing our project's configuration file. We will **not** use **uv** to manage a virtual environment, as Nix already handles that for us (unless you absolutely want to: however, you should then make sure that **uv** itself is being managed by Nix to ensure reproducibility).

Let's build a Python package called `pyclean`, the equivalent of our R package.

5.3.1 Step 1: Project Setup with uv

First, ensure **uv** is installed in your Nix environment:

```
library(rix)

rix(
  date = "2025-10-07",
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("pytest", "pandas")
  ),
  system_pkgs = "uv",
  ide = "none",
  project_path = ".",
  overwrite = TRUE
```

5.3 Part 2: Creating a Minimal Python Package with uv

```
)
```

Then, create a directory for your new package and initialize it:

```
mkdir pyclean  
cd pyclean  
uv init --bare
```

The `--bare` flag is perfect for our Nix workflow. It creates only the essential `pyproject.toml` file without creating a virtual environment or extra directories. This leaves us with a clean slate.

Now, we must create the source and test directories manually. We'll use the standard `src` layout:

```
mkdir -p src/pyclean  
mkdir tests  
touch src/pyclean/__init__.py
```

Your project structure should now look like this (check it using the `tree` command):

```
pyclean/  
    pyproject.toml  
    src/  
        pyclean/  
            __init__.py  
    tests/
```

5.3.2 Step 2: Write a Function and Declare Dependencies

Let's create our `clean_names` function inside a new file, `src/pyclean/formatters.py`.

```
# In src/pyclean/formatters.py
import pandas as pd

def clean_names(df: pd.DataFrame) -> pd.DataFrame:
    """Clean and standardize column names of a
    → DataFrame.

    Args:
        df: The input pandas DataFrame.

    Returns:
        A pandas DataFrame with standardized column
    → names.
    """
    new_df = df.copy()
    new_cols = {col: col.lower().replace(" ",
    → "_").replace(".", "_") for col in
    → new_df.columns}
    new_df = new_df.rename(columns=new_cols)
    return new_df
```

To make this function easily importable, we expose it in `src/pyclean/__init__.py`:

```
# In src/pyclean/__init__.py
from .formatters import clean_names
```

```
__version__ = "0.1.0"
```

Next, we must declare our dependencies by manually editing `pyproject.toml`. We need `pandas` for our function and `pytest` for our tests.

```
# In pyproject.toml
[project]
name = "pyclean"
version = "0.1.0"
description = "A simple package to clean data."
dependencies = [
    "pandas>=2.0.0",
]

[project.optional-dependencies]
test = [
    "pytest",
]

[tool.pytest.ini_options]
pythonpath = [
    "src"
]
```

The `pythonpath = ["src"]` line is the key. Without it, you'd first need to install your `pyclean` library in editable mode using `pip` before running the tests. By adding this block, simply running `pytest` from the command line will work.

5.3.3 Step 3: Add Unit Tests

Create a new test file, `tests/test_formatters.py`, and add your tests.

```
# In tests/test_formatters.py
import pandas as pd
from pyclean import clean_names

def test_clean_names_happy_path():
    messy_df = pd.DataFrame({"First Name": ["Ada"],  
    ↵ "Last.Name": ["Lovelace"]})
    cleaned_df = clean_names(messy_df)
    expected_cols = ["first_name", "last_name"]
    assert list(cleaned_df.columns) == expected_cols

def test_clean_names_is_idempotent():
    clean_df = pd.DataFrame({"first_name": ["a"],  
    ↵ "last_name": ["b"]})
    still_clean_df = clean_names(clean_df)
    assert list(still_clean_df.columns) ==
        ↵ list(clean_df.columns)
```

Since your Nix environment provides all the tools, you can run tests directly from your terminal:

```
pytest
```

5.3.4 Step 4: Build and Install

To package your code, you need a build tool. It turns out that uv bundles a build tool with it, so we only need to call `uv build`:

5.3 Part 2: Creating a Minimal Python Package with uv

```
# In your terminal, from the root of the 'pyclean'  
    ↵ project  
uv build
```

This creates a `dist/` directory containing a source distribution (`.tar.gz`) and a compiled wheel (`.whl`). The wheel is the modern standard for distribution.

Outside of a Nix shell, to use your package during development, you can install it in “editable” mode. This creates a link to your source code, so any changes you make are immediately reflected without needing to reinstall.

```
# Install the package and its test dependencies  
pip install -e .[test]
```

But we are working from a Nix shell. Instead, we will simply edit our `default.nix` to update the `PYTHONPATH` environment variable, so our package can easily be found. If you look at the `default.nix` file of the course you’ve been using, you’ll see the following at the bottom:

```
shellHook = ''  
  export PYTHONPATH=$PWD/pyclean/src:$PYTHONPATH  
'';
```

(you may need to adapt the path depending on where you’re developing the package). With this, dropping into the Nix shell, starting the Python interpreter and then typing `import pyclean` will work without any issues.

5.3.5 Step 5: Install from GitHub

Sharing via GitHub is the most common way to distribute packages that aren't on the official Python Package Index (PyPI):

1. Create a new, empty repository on GitHub.
2. Push your local project to the remote repository.
3. Now, anyone can install your package directly from GitHub using pip, which is smart enough to find and process your `pyproject.toml` file: `bash pip install git+https://github.com/yourusername/pyclean.git`

For Nix environments, add this to your `default.nix`:

```
pyclean = pkgs.python313Packages.buildPythonPackage
rec {
  pname = "pyclean";
  version = "0.1.0";
  src = pkgs.fetchgit {
    url = "https://github.com/b-rodrigues/pyclean";
    rev =
      "174d4d482d400536bb0d987a3e25ae80cd81ef3c";
    sha256 =
      "sha256-xTYydkuduPpZsCXE2fv5qZCnYYCRoNfpV71QBM3LMSg=";
  };
  pyproject = true;
  propagatedBuildInputs = [
    pkgs.python313Packages.pandas
    pkgs.python313Packages.setuptools ];
  # Add more dependencies to propagatedBuildInputs
  # as needed
};
```

You need to add the `rev`, which corresponds to the commit that want, and the `sha256`. To find the right `sha256`, start with an

5.4 Conclusion: The Packaging Mindset in the Age of AI

empty one (`sha256 = "";`) and try to build the package. The error message will give you the right `sha256`. Also note that this isn't the most idiomatic way to build a Python package for Nix, but it's good enough for our purposes.

Finally, add `pyclean` to the `buildInputs` of the shell:

```
buildInputs = [ rpkgs pyconf pyclean tex
    ↵ system_packages github_pkgs ];
```

This process is naturally more involved than simply calling `pip install`, but it has the advantage of being entirely reproducible.

5.4 Conclusion: The Packaging Mindset in the Age of AI

You have now successfully created, tested, documented, and shared a basic package in both R and Python. While there is much more to learn about advanced package development, you have already mastered the most important part: the **packaging mindset**.

From now on, when you start a new analysis project, think of it as a small, internal package.

- Put your reusable logic into functions.
- Place those functions in the `R/` or `mypackage/` source directory.
- Document them.
- Write a few simple tests to prove they work.

5 A Short Intro to Packaging Your Code in R and Python

- Manage dependencies formally in `DESCRIPTION` or `pyproject.toml`.

Adopting this structure will make your work more robust, easier to share, and fundamentally more reproducible. It is the bridge between writing one-off scripts and building reliable, professional data science tools.

This packaging mindset becomes even more powerful when you introduce a modern collaborator: the LLM. The structured, component-based nature of a package is the perfect way to interact with AI assistants.

A package provides a clear contract and a well-defined structure that LLMs thrive on. Instead of a vague prompt like, “Refactor my messy analysis script,” you can now make precise, targeted requests:

- “Here is my function `clean_names`. Please write three `pytest` unit tests for it, including one for the happy path, one for an empty DataFrame, and one for names that are already clean.”
- “Generate the roxygen2 documentation skeleton for this R function, including `@param`, `@return`, and `@examples` tags.”
- “I need a function in my `pyclean/utils.py` module that calculates the Z-score for a pandas Series. Please generate the function and its docstring.”

This synergy is a two-way street. Not only does the structure help you write better prompts, but LLMs excel at generating the very boilerplate that makes packaging robust. Tedious tasks like writing standard documentation headers, creating skeleton unit test files, or even generating a first draft of a function based on a clear description become near-instantaneous.

5.4 Conclusion: The Packaging Mindset in the Age of AI

This elevates your role from a writer of code to an **architect and a reviewer**. Your job is to design the components (the functions), prompt the LLM to generate the implementation, and then—most critically—use the testing framework you just built to rigorously verify that the AI-generated code is correct, efficient, and robust. You are the final authority, and the package structure gives you the tools to enforce quality control.

By combining the discipline of packaging with the power of LLMs, you lower the barrier to adopting best practices like comprehensive testing and documentation. This combination doesn't just make you faster; it makes you a more reliable and professional data scientist, capable of producing tools that are truly reproducible and built to last.

While a full guide to package development is beyond the scope of this course, it is the natural next step in your journey as a data scientist who produces reliable tools. When you are ready to take that step, here are the definitive resources to guide you:

- **For R:** The “R Packages” (2e) book by Hadley Wickham and Jennifer Bryan is the essential, comprehensive guide. It covers everything from initial setup with `{usethis}` to testing, documentation, and submission to CRAN. [Read it online here](#).
- **For Python:** The official **Python Packaging User Guide** is the place to start. For a more modern and streamlined approach that handles dependency management and publishing, many developers use tools like **Poetry** or **Hatch**.

Treating your data analysis project like a small, internal software package, complete with functions and tests, is a powerful mindset that will elevate the quality and reliability of your work.

Here are two new exercises to add to the end of the chapter.

5.5 Hands-on Exercises

5.5.1 Exercise 1: Build Your First Python Package with Nix

Goal: To build a complete, testable, and documented Python package and make it available in a reproducible Nix environment.

You will create a small Python package called `statstools`. This package will contain a single function that calculates descriptive statistics for a list of numbers.

Requirements:

1. **Project Structure:** Create the following directory structure for your package:

```
statstools/
    pyproject.toml
    default.nix
    src/
        statstools/
            __init__.py
            calculations.py
    tests/
        test_calculations.py
```

2. **Functionality:** In `src/statstools/calculations.py`, create a function `descriptive_stats(numbers: list) -> dict`. This function should take a list of numbers and return a dictionary containing the mean and standard deviation. Use the `numpy` library for the calculations.

3. **Documentation:** Write a clear docstring for your `descriptive_stats` function explaining what it does, its parameters (`Args`), and what it returns (`Returns`).
4. **Dependencies:** In your `pyproject.toml` file:
 - Define the project name as `statstools` and give it a version of 0.1.0.
 - Add `numpy` as a runtime dependency.
 - Add `pytest` as an optional dependency for testing.
 - Add the `[tool.pytest.ini_options]` block to set the `pythonpath` correctly so `pytest` can find your `src` directory.
5. **Unit Tests:** In `tests/test_calculations.py`, write at least two tests for your function using `pytest`:
 - A “happy path” test with a simple list like `[1, 2, 3, 4, 5]`.
 - A test with negative numbers and floats.
6. **GitHub Repository:** Push your completed package to a new public repository on GitHub.
7. **Nix Expression:** This is the most critical part. Create a `default.nix` file in the root of your project. This file should:
 - Fetch your package’s source code from your GitHub repository using `pkgs.fetchgit`. You will need to get the commit hash (`rev`) and the `sha256` hash (remember, you can get the correct `sha256` from the error message when you first try to build with an empty `sha256 = "";`).

5 A Short Intro to Packaging Your Code in R and Python

- Define your package using `pkgs.python3Packages.buildPython`. Ensure you set `pyproject = true`; and list its dependencies (`numpy`, `pytest`) in `propagatedBuildInputs`.
- Create a reproducible shell using `pkgs.mkShell` that includes both your `statstools` package and the Python interpreter.
- **Hint:** Look closely at the `default.nix` file from the main course repository for a complete example of how a Python package is fetched from GitHub and built.

To verify your work: * Run `pytest` from within the Nix shell to ensure all your tests pass. * Start a Python interpreter inside the Nix shell and successfully run `from statstools import descriptive_stats`.

5.5.2 Exercise 2: Build and Package an R Tool with Nix

Goal: To apply R's best practices for package development (`usethis`, `devtools`, `roxygen2`) and integrate the final product into a reproducible Nix environment.

You will create a small R package called `datasummary`. This package will provide a function to quickly summarize a data frame.

Requirements:

1. **Project Setup:** Use `usethis::create_package("datasummary")` to generate the standard R package structure.
2. **Functionality:**
 - Use `usethis::use_r("summarize_df")` to create a new file for your function.

- The function, `summarize_df(df)`, should take a data frame as input.
- It should return a new data frame with two columns: `column_name` and `missing_values`, showing the count of NAs in each column of the input data frame.
- This function will require the `{dplyr}` package. Use `usethis::use_package("dplyr")` to declare this dependency in your `DESCRIPTION` file.

3. Documentation:

- Use `roxygen2` comments to document your `summarize_df` function.
- Include `@param`, `@return`, and `@export` tags.
- Provide a working example in the `@examples` tag.
- Since your function will use functions from `{dplyr}`, add the appropriate `@importFrom` tags (e.g., `@importFrom dplyr %>% summarise_all`).
- Run `devtools::document()` to generate the documentation.

4. Unit Tests:

- Set up testing with `usethis::use_testthat()`.
- Create a test file with `usethis::use_test("summarize_df")`.
- In the test file, write at least one test using `test_that()` that creates a sample data frame with missing values and checks if your function returns the correct counts.
- Run `devtools::test()` to execute your tests.

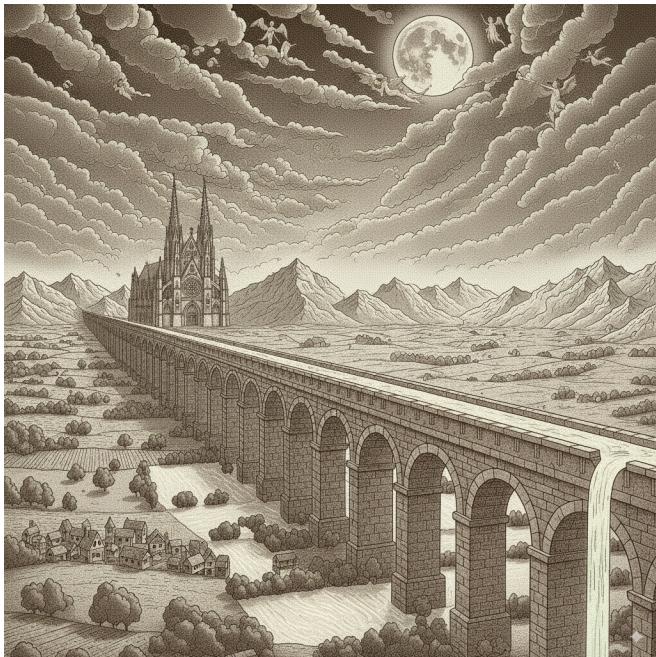
5. **Quality Check:** Run `devtools::check()` to ensure your package is free of errors, warnings, and notes.
6. **GitHub Repository:** Push your completed R package to a new public repository on GitHub.

7. **Nix Expression:** Create a `default.nix` file in the root of your `datasummary` project. This file must:

- Define a custom R environment.
- Define your `datasummary` package by using `buildRPackage`. Fetch the source from your GitHub repository using `pkgs.fetchgit` (you will need the `rev` and `sha256`).
- Make sure to list its R dependencies (like `{dplyr}` and its own dependencies) in `propagatedBuildInputs`.
- Create a final shell with `pkgs.mkShell` that drops you into an R session where your `datasummary` package is installed and available.
- **Hint:** Refer to the main course repository's `default.nix` file to see how R packages are defined with `buildRPackage` and included in the final R environment. Using `{nix}` is also an option.

To verify your work: * Drop into the Nix shell provided by your `default.nix`. * Start an R session. * Successfully run `library(datasummary)` and test your `summarize_df()` function on a data frame like `iris`.

6 Building Reproducible Pipelines with Nix and {riexpress}



What you'll have learned by the end of the chapter: how to orchestrate a fully reproducible, polyglot analytical pipeline using Nix as a build automation tool, and why this is a fundamentally

more robust approach than using computational notebooks or other common workflow tools.

6.1 Introduction: From Scripts and Notebooks to Pipelines

So far, we have learned about the 3 main necessary pillars for building reproducible pipelines:

1. **Define Reproducible Environments** with Nix and {trix} to ensure everyone uses the exact same versions of R, Python or Julia, and all system-level dependencies.
2. **Manage Reproducible History** with Git to track every change to our code and collaborate effectively.
3. **Write Reproducible Logic** with Functional Programming to create clean, testable, and predictable functions.

The last pillar is orchestration.

How do we take our collection of functions and data files and run them in the correct order to produce our final data product? This problem of managing computational workflows is not new, and a whole category of **build automation tools** has been created to solve it.

The original solution to this problem, dating back to the 1970s, is **make**. Created by Stuart Feldman at Bell Labs in 1976, **make** was born out of frustration. Feldman, working on his Fortran programs, was tired of the tedious and error-prone process of manually re-compiling only the necessary parts of his code after making a change. He designed **make** to read a **Makefile** that describes the dependency graph of a project. You tell it that `report.pdf` depends on `plot.png`. If you change the code that

6.1 Introduction: From Scripts and Notebooks to Pipelines

generates `plot.png`, `make` is smart enough to only re-run the steps needed to rebuild the plot and the final report. General-purpose tools like `waf` follow a similar philosophy.

The strength of these tools is their language-agnosticism, but their weakness is that they only track files and know nothing about the *software environment* needed to create those files. Another limitation of these generic tools is that they are **file-centric**. This means that *you* are responsible for manually handling all input and output. Your first script must explicitly save its result as `data.csv`, and your second script must explicitly load `data.csv`. This adds boilerplate code and creates a new surface for errors.

This is where a specialized tool like R's `{targets}` package shines. `{targets}` tracks dependencies between **R objects directly**, not just files. When you pass a data frame from one step to the next, `{targets}` automatically handles the **serialization** for you (serialization is the process of saving an object into a binary to disk) behind the scenes and loads it back when needed. This is a massive ergonomic improvement, allowing you to think in terms of data objects, not file paths.

The Python ecosystem, while rich in tools, lacks a single, dominant tool that offers the same lightweight, object-centric feel as `{targets}` for everyday analysis. Tools like **Snakemake** are powerful but often follow the `make` model of file-based I/O. Others like **Luigi** or **Airflow** are typically used for large-scale data engineering but can be overkill for a typical analytical project. This gap highlights the need for a solution that combines an ergonomic, object-passing interface with robust reproducibility.

Furthermore, all these tools, from `make` to `{targets}` to `Airflow`, separate workflow management from environment management. You use one tool to run the pipeline and another

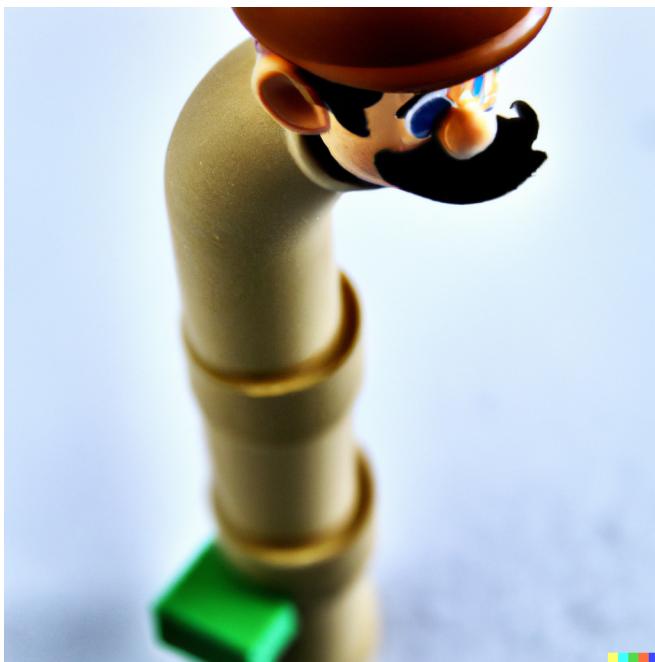
6 Building Reproducible Pipelines with Nix and {rixpress}

(like `conda`, Docker, or `{renv}`) to set up the software. But what if we could use a single, declarative system to manage *both*?

This is why we will also be using Nix for build automation. Nix is not just a package manager; it is a full-fledged build system. When Nix builds a pipeline, it controls the entire dependency graph, from your input data files all the way down to the C compiler used to build R itself. It unifies the “what to run and when” problem with the “what tools to use” problem into a single, cohesive framework.

However, writing build instructions directly in the Nix language can be complex. This is where `{rixpress}` comes in. It provides a user-friendly R interface, heavily inspired by `{targets}`, that lets us define our pipeline in familiar R code. `{rixpress}` then translates this into the necessary Nix expressions for us. We get the ergonomic, object-passing feel of `{targets}` with the unparalleled, bit-for-bit reproducibility of the Nix build system. It is the perfect tool to complete our reproducible workflow.

6.2 Our First Polyglot Pipeline



Let's start with a simple pipeline. Our goal will be to read the `mtcars` dataset, perform some initial filtering in Python with `{polars}`, pass the result to R for further manipulation with `{dplyr}`, and finally compile a Quarto document that presents the results.

First, let's create a new project directory. Inside, we'll bootstrap our project. If you're in a terminal, you can get a temporary shell with the necessary tools by running:

```
nix-shell --expr "$(curl -sL  
↳ https://raw.githubusercontent.com/ropensci/rix/main/inst,
```

Once inside this temporary shell, start R and run:

```
rixpress::rxp_init()
```

This handy function creates two essential plain text files: `gen-env.R` and `gen-pipeline.R`.

6.2.1 Step 0: Use Git

This might be the right time to start a Git repository. Either start by creating an empty project on GitHub, or start from your command line, locally:

```
git init
```

6.2.2 Step 1: Defining the Environment

Open `gen-env.R`. This is where we use `{rix}` to define the tools our pipeline needs.

```
# In gen-env.R
library(rix)

# Define execution environment for our polyglot
  ↵ pipeline
rix(
  date = "2025-06-02",
  r_pkgs = c("dplyr", "quarto", "reticulate",
    ↵ "jsonlite"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "pyarrow", "pandas")
```

```
)  
git_pkgs = list(  
    package_name = "rixpress",  
    repo_url =  
        "https://github.com/b-rodrigues/rixpress",  
    commit = "HEAD"  
)  
ide = "none",  
project_path = ".",  
overwrite = TRUE  
)
```

Run this script (`source("gen-env.R")`) to generate the `default.nix` file that describes our complete environment. Now, exit the temporary shell, build your project environment with `nix-build`, and enter it with `nix-shell`.

6.2.3 Step 2: Defining the Pipeline

Now, open `gen-pipeline.R`. This plain text file is where we'll define the actual pipeline. `{rixpress}` offers several ways to pass data between languages.

A pipeline is a list of derivations. A derivation is defined using functions such as `rxp_r()`, `rxp_py()`, etc. Most of the time, we start by importing data. In this case, we will be importing a `.csv` file (which you can download here and save it in the `data/` folder) using `polars`:

```
# In gen-pipeline.R  
library(rixpress)
```

```
list(
  rxp_py_file(
    name = mtcars_pl,
    path = "data/mtcars.csv",
    read_function = "lambda x: polars.read_csv(x,
      ↵ separator='|')"
  ),
  ...
)
```

We use the `rxp_py_file()` function to define a derivation that reads in the `.csv` file using the `read_csv()` function from `polars`. When importing data using `rxp_py_file()` or (`rxp_r_file()`), the `read_function` argument must be a function of a single argument, the path to the data.

Next, we want to filter the dataset:

```
# In gen-pipeline.R
library(rixpress)

list(
  rxp_py_file(
    name = mtcars_pl,
    path = 'data/mtcars.csv',
    read_function = "lambda x: polars.read_csv(x,
      ↵ separator='|')"
  ),
  rxp_py(
    name = mtcars_pl_am,
    expr = "mtcars_pl.filter(polars.col('am') ==
      ↵ 1).to_pandas()"
  ),
)
```

```
...
```

The next derivation is defined using `rxp_py()` which runs Python code. As you can see, the `expr` argument is literal Python code, where `polars` is used to filter data and then convert the result to a `pandas` data frame.

To pass data to R, we have two methods available.

6.2.3.1 Method 1: Using Language-Specific Converters

The `rxp_r2py()` and `rxp_py2r()` functions are convenient wrappers that use the `{reticulate}` package behind the scenes to convert objects:

```
r xp_py2r(  
  name = mtcars_am_r,  
  expr = mtcars_pl_am  
,  
  ...
```

This converts the `mtcars_pl_am` data frame (which is a `pandas` data frame) into an R data frame using the R package `{reticulate}`.

We can then continue with an R derivation:

```
r xp_r(  
  name = mtcars_head,  
  expr = head(mtcars_am_r)  
,  
  ...
```

This works well, but it tightly couples your pipeline to `{reticulate}`'s conversion capabilities, which in some cases could be overkill.

6.2.3.2 Method 2: A lighter Approach with Universal Data Formats

A lighter and language-agnostic approach is to use a universal data format like **JSON**. This makes your pipeline more modular, as any language that can read and write JSON could be added in the future. `{rixpress}` supports this via the `serialize_function` and `unserialize_function` arguments.

Let's rewrite our pipeline to use JSON. First, we need simple helper functions in our project.

Create a script called `functions.py` that will contain all the Python helper functions we might need. In it, add:

```
# A function to save a polars DataFrame to a JSON
# file
def serialize_to_json(pl_df, path):
    with open(path, 'w') as f:
        f.write(pl_df.write_json())
```

Do the same for R functions, in `functions.R`:

```
# Just aliasing head for demonstration
my_head <- head
```

Now, we can update `gen-pipeline.R` to use these helpers:

```
library(rixpress)

list(
  .....

  rxp_py(
    name = mtcars_pl_am,
    expr = "mtcars_pl.filter(polars.col('am') ==
      ↵ 1)",
    additional_files = "functions.py",
    encoder = "serialize_to_json" # Use our Python
      ↵ helper
  ),

  rxp_r(
    name = mtcars_head,
    expr = my_head(mtcars_pl_am),
    additional_files = "functions.R",
    decoder = "jsonlite::fromJSON" # Use R's
      ↵ jsonlite
  ),
  ...
)
```

This approach works well in simple cases like passing data frames between languages, but may not work for more complex objects for which `{reticulate}` may have specialized code for conversion.

6.2.4 Step 3: Building and Inspecting the Pipeline

The complete pipeline will look like this:

```
library(rixpress)

list(
  rxp_py_file(
    name = mtcars_pl,
    path = 'data/mtcars.csv',
    read_function = "lambda x: polars.read_csv(x,
      ↴ separator='|')"
  ),
  # Note: polars must be converted to pandas for
  ↴ reticulate
  rxp_py(
    name = mtcars_pl_am,
    expr = "mtcars_pl.filter(polars.col('am') ==
      ↴ 1).to_pandas()"
  ),
  rxp_py(
    name = mtcars_pl_am,
    expr = "mtcars_pl.filter(polars.col('am') ==
      ↴ 1)",
    additional_files = "functions.py",
    encoder = "serialize_to_json" # Use our Python
      ↴ helper
  ),
  rxp_r(
    name = mtcars_head,
    expr = my_head(mtcars_pl_am),
```

```

additional_files = "functions.R",
decoder = "jsonlite::fromJSON" # Use R's
    ↵ jsonlite
),
) |>
rxp_populate()

```

The very last function, `rxp_populate()` takes a list of derivations as input and will translate the list of derivations into a `pipeline.nix` file and instruct Nix to build the entire pipeline. This will not build the pipeline (unless you set the `build` argument to `TRUE`), so to build the pipeline use `rxp_make()`. Once the build process is done, you can use `rxp_inspect()` to check which artifacts where built, and you can easily access the any of them:

```

# Check out all artifacts
rxp_inspect()

# Load the mtcars_head data frame into your R
    ↵ session
rxp_load("mtcars_head")

# You can now inspect it
head(mtcars_head)

```

6.3 Caching

First, visualize your pipeline's dependency graph:

6 Building Reproducible Pipelines with Nix and {rixpress}

```
# You'll need to first generate the required files
↳ by running
# The execution environment will need to include
↳ `ggplot2` and `ggdag`
# Then you can visualize the graph
r xp_ggdag()
```

This will show you a clear, unambiguous graph of your workflow. You can also use `r xp_trace()` for a text based view:

Now, modify a step. Open `gen-pipeline.R` and change the `my_head` function in `functions.R` to use `tail()` for example. Save the file and re-run `rixpress()`. Nix will detect that the data loading and Python filtering steps are unchanged and instantly use the cached results from the `/nix/store/`. It will **only** re-build the final R step that was affected by the change.

This is the incredible power of a proper build automation tool. The cognitive load of tracking what to re-run is gone. You are free to experiment, confident that the tool will efficiently and correctly rebuild only what is necessary.

6.4 Debugging and Working with Build Logs

But what happens to the *old* results? What if you want to compare the `head()` version of your data to the `tail()` version? This is where {rixpress}'s build logging becomes a superpower.

6.4 Debugging and Working with Build Logs

Every time you run `rixpress()`, a timestamped log of that specific build is saved in the `_rixpress/` directory. This is like having a Git history for your pipeline's *outputs*.

You can list all the past builds you've run:

```
rxp_list_logs()
#>
  ↵   filename      modification_time
#> 1
  ↵   build_log_20250602_143015_a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p
  ↵   2025-06-02 14:30:15
#> 2
  ↵   build_log_20250602_142500_z9y8x7w6v5u4t3s2r1q0p9o8n7m615l
  ↵   2025-06-02 14:25:00
```

Let's say the first log (...a1b2c3d...) is our new `tail()` run, and the second (...z9y8x7w...) is our original `head()` run. You can now pull the artifact from the *old* run directly into your current session for comparison:

```
# Load the result from the MOST RECENT build
new_result <- rxp_read("mtcars_head")

# Load the result from the PREVIOUS build by
  ↵ matching part of its log name
old_result <- rxp_read("mtcars_head", which_log =
  ↵ "z9y8x")

# Now you can compare them!
new_result
old_result
```

This is an incredibly powerful debugging and validation tool. You can go back in time to inspect the state of any output from any previous pipeline run, as long as it's still in the Nix store. This provides a safety net and traceability that is simply absent in a notebook-based workflow.

6.5 Running Someone Else's Pipeline: The Ultimate Test of Reproducibility

Imagine a collaborator wants to run your pipeline. If you had sent them a Jupyter notebook, they would face a series of questions: “Which version of Python did you use? What packages do I need? In what order do I run the cells? What is this variable that's used but never defined?”

With our Nix-based workflow, the process is radically simpler and more robust. All they need to do is:

1. `git clone` your repository (which, unlike a notebook, has a clean, readable history).
2. Run `nix-build`, then `nix-shell` in the project directory.
3. Start an R session, and build the pipeline by running the `gen-pipeline.R` script, or by running `rxp_make()`.

That's it. Nix reads your `default.nix` and `pipeline.nix` files and builds the *exact* same environment and the *exact* same data product, bit-for-bit. It solves all the problems we identified with other approaches: it controls the language versions, the operating system libraries, and all dependencies in one unified, declarative system.

You now have the knowledge to build robust, efficient, polyglot, and truly reproducible analytical pipelines. By abandoning

the chaos of notebooks for production work and embracing the structured, automatable world of plain text files and build automation, your work becomes more reliable, more scalable, and fundamentally more scientific.

6.6 Hands-On Exercises

Check out the `rixpress_demos` GitHub repository and, by taking inspiration from them, write a pipeline that uses the packages you wrote for the exercises of Chapter 6 *Packages*.

7 Containerization With Docker



What you'll have learned by the end of the chapter: build self-contained, truly reproducible analytical pipelines thanks to Docker.

7.1 Introduction

Up until now, we've been using Nix as a powerful tool for creating reproducible development environments directly on our machines. Nix gives us fine-grained control over every package and dependency in our project, ensuring bit-for-bit reproducibility. However, when it comes to distributing a *data product*, another technology, Docker, is incredibly popular.

While Nix manages dependencies for an application that runs on a host operating system, Docker takes a different approach: it packages an application *along with* a lightweight operating system and all its dependencies into a single, portable unit called a **container**. This container can then run on any machine that has Docker installed, regardless of its underlying OS.

The idea is to not only deliver the source code for our data products, but also include it inside a complete package that contains not only R and the required libraries, but also the necessary components of the operating system itself (which will usually be a flavor of Linux, like Ubuntu). This approach solves the “it works on my machine” problem in a very direct way.

For rebuilding a data product, a single command can be used which will pull the Docker **image** from a registry, start a **container**, build the data product, and stop.

If you've never heard of Docker before, this chapter will provide the basic knowledge required to get started. Let's start by watching this very short video that introduces the core concepts.

In a sense, Docker can be seen as a lightweight virtual machine running a Linux distribution (usually Ubuntu) that you can interact with using the command line. This also means that familiarity with Linux distributions will make using Docker easier. Thankfully, there is a very large community of Docker users

who also use R. This community is organized as the Rocker Project and provides a very large collection of `Dockerfiles` to get started easily. As you saw in the video above, `Dockerfiles` are simple text files that define a Docker image, from which you can start a container.

While Nix and Docker are often seen as competing tools for environment management, they can be used together effectively by leveraging their respective strengths. A powerful pattern is to use Nix *inside* a Docker container. In this setup, you start with a minimal base Docker image that has Nix installed. Then, you use Nix to declaratively build the precise, bit-for-bit reproducible development environment within the image. Docker’s role then shifts from environment provisioning to simply being a portable, universal runtime for this Nix-managed environment, making it excellent for deployment.

This approach contrasts with using Docker alone for reproducibility. While many attempt this, it’s not Docker’s core strength. Achieving a reproducible `docker build` often requires “abusing” Docker’s features—pinning base image hashes, freezing system package versions, and using specific package manager snapshots—because Docker was designed for creating portable runtime containers, not for guaranteeing reproducible builds. Its true reproducibility promise is that a specific, pre-built image will always launch an identical container, not that building the same `Dockerfile` twice will yield an identical image.

7.2 Docker essentials

7.2.1 Installing Docker

The first step is to install Docker. You'll find the instructions for Ubuntu here, for Windows here (read the system requirements section as well!) and for macOS here (make sure to choose the right version for the architecture of your Mac, if you have an M1 Mac use *Mac with Apple silicon*).

After installation, it might be a good idea to restart your computer, if the installation wizard does not invite you to do so. To check whether Docker was installed successfully, run the following command in a terminal (or on the desktop app on Windows):

```
docker run --rm hello-world
```

This should print the following message:

```
Hello from Docker!
This message shows that your installation appears to
↪ be working correctly.
```

```
To generate this message, Docker took the following
↪ steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image
   ↪ from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from
   ↪ that image which runs the
```

executable that produces the output you are
↳ currently reading.
4. The Docker daemon streamed that output to the
↳ Docker client, which sent it
to your terminal.

To try something more ambitious, you can run an
↳ Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a
↳ free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

If you see this message, congratulations, you are ready to run Docker. If you see an error message about permissions, this means that something went wrong. If you're running Linux, make sure that your user is in the Docker group by running:

```
groups $USER
```

you should see your username and a list of groups that your user belongs to. If a group called `docker` is not listed, then you should add yourself to the group by following these steps.

7.2.2 The Rocker Project and image registries

When running a command like:

7 Containerization With Docker

```
docker run --rm hello-world
```

what happens is that an image, in this case `hello-world` gets pulled from a so-called *registry*. A registry is a storage and distribution system for Docker images. Think of it as a GitHub for Docker images, where you can push and pull images, much like you would with code repositories. The default public registry that Docker uses is called Docker Hub, but companies can also host their own private registries to store proprietary images. When you execute a command like `docker run`, the Docker daemon first checks if the image is present on your local machine. If not, it connects to the configured registry, downloads the required image layers, and then assembles them to run the container.

Many open source projects build and distribute Docker images through Docker Hub, for example the Rocker Project.

The Rocker Project is instrumental for R users that want to use Docker. The project provides a large list of images that are ready to run with a single command. As an illustration, open a terminal and paste the following line:

```
docker run --rm -e PASSWORD=yourpassword -p
← 8787:8787 rocker/rstudio
```

Once this stops running, go to `http://localhost:8787/` and enter `rstudio` as the username and `yourpassword` as the password. You should login to a RStudio instance: this is the web interface of RStudio that allows you to work with R from a server. In this case, the *server* is the Docker container running the image. Yes, you've just pulled a Docker image containing Ubuntu with a fully working installation of RStudio web!

(If you cannot connect to `http://localhost:8787`, try with the following command:

```
docker run --rm -ti -d -e PASSWORD=yourpassword -p
    ↳ 8787:8787 --network="host" rocker/rstudio
```

)

Let's open a new script and run the following lines:

```
data(mtcars)

summary(mtcars)
```

You can now stop the container (by pressing `CTRL-C` in the terminal). Let's now rerun the container... (with the same command as before) you should realize that your script is gone! This is the first lesson: whatever you do inside a container will disappear once the container is stopped. This also means that if you install the R packages that you need while the container is running, you will need to reinstall them every time. Thankfully, the Rocker Project provides a list of images with many packages already available. For example to run R with the `{tidyverse}` collection of packages already pre-installed, run the following command:

```
docker run --rm -ti -e PASSWORD=yourpassword -p
    ↳ 8787:8787 rocker/tidyverse
```

If you compare it to the previous command, you see that we have replaced `rstudio` with `tidyverse`. This is because `rocker/tidyverse` references an image, hosted on Docker

7 Containerization With Docker

Hub, that provides the latest version of R, RStudio server and the packages from the `{tidyverse}`. You can find the image hosted on Docker Hub here. There are many different images, and we will be using the *versioned* images made specifically for reproducibility. For now, however, let's stick with the `tidyverse` image, and let's learn a bit more about some specifics.

7.2.3 Basic Docker workflow

You already know about running containers using `docker run`. With the commands we ran before, your terminal will need to stay open, or else, the container will stop. Starting now, we will run Docker commands in the background. For this, we will use the `-d` flag (`d` as in *detach*), so let's stop the container one last time with CTRL-C and rerun it using:

```
docker run --rm -d -e PASSWORD=yourpassword -p
    ↵ 8787:8787 rocker/tidyverse
```

(notice `-d` just after `run`). You can run several containers in the background simultaneously. You can list running containers with `docker ps`:

```
docker ps
CONTAINER ID        IMAGE               COMMAND      CREATED
    ↵   STATUS            PORTS
    ↵   NAMES
c956fbeebebc       rocker/tidyverse     "/init"      3
    ↵   minutes ago      Up 3 minutes
    ↵   0.0.0.0:8787->8787/tcp,  :::8787->8787/tcp
    ↵   elastic_morse
```

The running container has the ID `c956fbbeebcb`. Also, the very last column, shows the name of the running container. This is a label that you can change. For now, take note of ID, because we are going to stop the container:

```
docker stop c956fbbeebcb
```

After Docker is done stopping the running container, you can check the running containers using `docker ps` again, but this time no containers should get listed. Let's also discuss the other flags `--rm`, `-e` and `-p`. `--rm` removes the container once it's stopped. Without this flag, we can restart the container and all the data and preferences we saved will be restored. However, this is dangerous because if the container gets removed, then everything will get lost, forever. We are going to learn how to deal with that later. `-e` allows you to provide environment variables to the container, so in this case the `$PASSWORD` variable. `-p` is for setting the port at which your app is going to get served. Let's now rerun the container, but by giving it a name:

```
docker run -d --name my_r --rm -e
  ↵ PASSWORD=yourpassword -p 8787:8787
  ↵ rocker/tidyverse
```

Notice the `--name` flag followed by the name we want to use, `my_r`. We can now interact with this container using its name instead of its ID. For example, let's open an interactive bash session. Run the following command:

```
docker exec -ti my_r bash
```

7 Containerization With Docker

You are now inside a terminal session, inside the running container! This can be useful for debugging purposes. It's also possible to start R in the terminal, simply replace `bash` by `R` in the command above.

Finally, let's solve the issue of our scripts disappearing. For this, create a folder somewhere on your computer (host). Then, rerun the container, but this time with this command:

```
docker run -d --name my_r --rm -e  
    ↵ PASSWORD=yourpassword -p 8787:8787 -v  
    ↵ /path/to/your/local/folder:/home/rstudio/scripts:rw  
    ↵ rocker/tidyverse
```

where `/path/to/your/local/folder` should be replaced to the folder you created. You should now be able to save the scripts inside the `scripts/` folder from RStudio and they will appear in the folder you created.

7.2.4 Making our own images

To create our own images, you can start from an image provided by an open source project like Rocker, or you can start from the base Ubuntu or Alpine Linux images. These images are bare-bones compared to the ones from Rocker, but as a consequence they are very lightweight, which in some cases can be important. For the remainder of the course, we are going to start from a base Ubuntu image, and use Nix to add our software stack.

The snippet below is a minimal `Dockerfile` that shows exactly this:

```
FROM ubuntu:latest

RUN apt update -y

RUN apt install curl -y

# We don't have R nor {rix} in this image, so we can
# bootstrap it by downloading
# the default.nix file that comes with {rix}. You
# can also download it beforehand
# and then copy it to the Docker image
RUN curl -O
    ↳ https://raw.githubusercontent.com/ropensci/rix/main/inst

# The next 4 lines install Nix inside Docker. See
# the Determinate Systems installer's
# documentation
RUN curl --proto '=https' --tlsv1.2 -sSf -L
    ↳ https://install.determinate.systems/nix | sh -s
    -- install linux \
    --extra-conf "sandbox = false" \
    --init none \
    --no-confirm

# Adds Nix to the path, as described by the
# Determinate Systems installer's documentation
ENV PATH="${PATH}:/nix/var/nix/profiles/default/bin"
ENV user=root

# Set up rstats-on-nix cache
# Thanks to the rstats-on-nix cache, precompiled
# binary packages will
# be downloaded instead of being compiled from
# source
```

7 Containerization With Docker

```
RUN mkdir -p /root/.config/nix && \  
    echo "substituters = https://cache.nixos.org  
        ↵ https://rstats-on-nix.cachix.org" >  
        ↵ /root/.config/nix/nix.conf && \  
    echo "trusted-public-keys =  
        ↵ cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGsp  
        ↵ rstats-on-nix.cachix.org-1:vdiiVgocg6WeJrODIqdprZRUrhi  
        ↵ >> /root/.config/nix/nix.conf  
  
# Copy a script to generate the environment of  
    ↵ interest using {rix}  
COPY generate_env.R .  
  
# This will overwrite the default.nix we downloaded  
    ↵ previously with a new  
# expression generated from running `generate_env.R`  
RUN nix-shell --run "Rscript generate_env.R"  
  
# We now build the environment  
RUN nix-build  
  
# Finally, we run `nix-shell`. This will get  
    ↵ executed when running  
# containers from this image. You can of course put  
    ↵ anything in here  
CMD nix-shell
```

This can seem quite complicated, but if you take the time to read the comments, you'll see that it's actually quite simple.

Every **Dockerfile** starts with a **FROM** statement. This means that this **Dockerfile** will use the **ubuntu:latest** image as a starting point.

We start off from the `ubuntu:latest` image: you might read online that this is not a good practice, and that instead one should use a stable image, for example `ubuntu:24.04` which will always use version 24.04 of Ubuntu. This is true **IF** you don't use Nix. But since we are using Nix to set up the reproducible development environment, we can use `ubuntu:latest`: our development environment will always be exactly the same, thanks to Nix.

Then, every command we wish to run starts with a `RUN` statement. We install and configure Nix, copy an R script to generate the environment (we could also copy an already generated `default.nix` instead) and then build the environment. Finally, we finish by running `nix-shell` when executing a container which is the command prepended with `CMD`.

This image actually does two things:

- a first step which consists in setting up Nix inside Docker;
- a second step which consists in setting up our project-specific Nix development environment.

Because the first step is generic, we will split up this in two stages.

First, create a new Dockerfile in a separate directory, with a new Git repo so that you can commit and push it (later in the book we will set up continuous integration to build and publish this image automatically):

```
# Stage 1 - Base with Nix and rstats-on-nix cache
FROM ubuntu:latest AS nix-base

RUN apt update -y && apt install -y curl
```

7 Containerization With Docker

```
# Install Nix via Determinate Systems installer
RUN curl --proto '=https' --tlsv1.2 -sSf -L
    https://install.determinate.systems/nix | sh -s
    -- install linux \
    --extra-conf "sandbox = false" \
    --init none \
    --no-confirm

ENV PATH="/nix/var/nix/profiles/default/bin:${PATH}"
ENV user=root

# Configure Nix binary cache
RUN mkdir -p /root/.config/nix && \
    echo "substituters = https://cache.nixos.org
        https://rstats-on-nix.cachix.org" >
    /root/.config/nix/nix.conf && \
    echo "trusted-public-keys =
        cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGsp
        rstats-on-nix.cachix.org-1:vdiiVgocg6WeJr0DIqdprZRUrhi" >> /root/.config/nix/nix.conf
```

Commit and push. Then, we need to build this image once, and tag it:

```
docker build -t nix-base:latest .
```

This image is now available on our machines under the tag `nix-base:latest`, and we can refer to it for any of our projects. For a new project, simply reuse it like so:

```

FROM nix-base:latest

COPY generate_env.R .

RUN curl -O
  https://raw.githubusercontent.com/ropensci/rix/main/inst
RUN nix-shell --run "Rscript generate_env.R"
RUN nix-build

CMD ["nix-shell"]

```

The issue with this approach is that now you have created a dependency between the two Dockerfiles which you need to manage. I would recommend the second approach only if you can push the first image with the Nix base on a registry (either public or a private one from your company). Later in this chapter we will publish the first image.

In the same folder than the second Dockerfile, add the required `generate_env.R` script:

```

library(rix)

rix(
  date = "2025-08-04",
  r_pkgs = c("dplyr", "ggplot2"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE

```

7 Containerization With Docker

```
)
```

This will setup an environment for our project. Let's stop here, and build the image:

```
docker build -t my-project .
```

and now run a container:

```
docker run -it --rm --name my-project-container
↪ my-project
```

This should drop you in an interactive Nix shell running inside Docker! As Docker is more popular than Nix, in particular in enterprise settings, this makes sharing development environments easier.

Remember, anything you do in this container will be lost after you stop it. So if you want to use it to work interactively on files, you should mount a volume:

```
docker run --rm --name my-project-container -v
↪ /path/to/your/local/project-folder/workspace:/workspace:rw
↪ -w /workspace my-project
```

This will mount a folder called `workspace` inside a running Docker container that will map to a folder called `workspace` on your current project folder. The `-w /workspace` flag sets the working directory inside the container to `/workspace`, so any commands you run will execute from there. This acts as a kind

of tunnel between the two; any file put there will be available and editable on the other side.

While this is good to know, I don't recommend using Docker to work interactively. Use Nix for this instead, and use Docker to then deploy whatever product you've been working on once you're done.

Before moving on to actually build projects using Docker, let's first publish the base Nix image on Docker Hub to easily re-use it across projects.

7.2.5 Publishing images on Docker Hub

If you want to share Docker images through Docker Hub, you first need to create a free account. A free account gives you unlimited public repositories. If you want to make your images private, you need a paid account. For our purposes though, a free account is more than enough. In the next section, we will discuss how you can build new images upon other images without using Docker Hub.

We will be uploading the image `nix-base` to Docker Hub.

Now is the right moment to talk about the `docker images` command. This will list all the images available on your computer. You should see something like this:

REPOSITORY	TAG	IMAGE ID	CREATED
↳ SIZE			
<code>nix-base</code>	latest	d3764d067534	2 days
↳ ago	1.61GB		
<code>dev_env_r</code>	latest	92fcf973ba42	2 days
↳ ago	1.42GB		

7 Containerization With Docker

```
raps_ubuntu_r      latest    7dababdf3c7ee   4 days
↳ ago   1.04GB
rocker/tidyverse   4.2.2     545e4538a28a   3 weeks
↳ ago   2.19GB
rocker/r-ver       4.2.2     08942f81ec9c   3 weeks
↳ ago   824MB
```

Take note of the image id of the `nix-base` image (second line), we will use it to push our image to Docker Hub. Also, don't be alarmed by the size of the images, because this is a bit misleading. Different images that use the same base (so here Ubuntu), will reuse "layers" such that they don't actually take up the size that is printed by `docker images`. So if images A and B both use the same version of Ubuntu as a base, but image A has RStudio installed and B also RStudio but Python as well, most of the space that A and B take up will be shared. The only difference will be that B will need a little bit more space for Python.

You can also list the running containers with `docker container ls` (or `docker ps`). If a container is running you should see something like this:

```
CONTAINER ID  IMAGE           COMMAND      CREATED
545e4538a28a  rocker/tidyverse  "/init"    3
↳ minutes ago

STATUS        PORTS
↳ NAMES
Up 3 minutes  0.0.0.0:8787->8787/tcp,
↳ :::8787->8787/tcp  elastic_morse
```

You can stop the container by running `docker stop CONTAINER`

ID. So, list the images again using `docker images`. Take note of the image id of the image you want to push to Docker Hub.

Now, log in to Docker Hub using `docker login` (yes, from your terminal). You will be asked for your credentials, and if log in is successful, you see a message **Log In Succeeded** in your terminal (of course, you need first to have an account on Docker Hub).

Now, you need to tag the image (this gives it a version number). So you would write something like:

```
docker tag IMAGE_ID
↪ your_username_on_docker_hub/your_image:version1
```

so in my case, it would be:

```
docker tag 92fcf973ba42 brodriguesco/nix-base:latest
```

Next, I need to push it using `docker push`:

```
docker push brodriguesco/nix-base:latest
```

You can go check your profile and your repositories, you should see your image there.

This image can now be used as a stable base for developing our pipelines. Here's how I can now use this base image for our project:

```
FROM brodriguesco/nix-base:latest
```

```
RUN mkdir ...
```

7 Containerization With Docker

Now I'm re-using the image that defines the development environment, and I can do so for as many projects as necessary. I would recommend putting a link to the base image as a comment just before the first `FROM`.

If you want to test this, you could delete all images and containers from your system. This way, when you build the image using the above Dockerfile, it will have to pull from Docker Hub. To delete all containers, start by using `docker system prune`. You can then delete all images using `docker rmi $(docker images -a -q)`. This should remove everything.

If you work for a company that has its own private registry, the process will be essentially the same, as it's just that Docker would have been configured to pull and push to the private registry instead.

In the next section, I'll explain to you how you can re-use base images like we just did, but without using Docker Hub, in case you cannot, or do not want, to rely on it.

7.2.6 Sharing a compressed archive of your image

If you can't upload the image on Docker Hub, you can still "save it" into a file and share that file instead (internally to your institution/company).

Run `docker save` to save the image into a file:

```
docker save nix-base > nix-base.tar
```

This will create a ``.tar`` file of the image. You can
↳ then compress this file

```
with an archiving tool if you want. If you're on
↳ Linux, you could do so in one
go (this will take some time):
```

```
```bash
docker save nix-base | gzip > nix-base.tgz
```

If you want to load this image, use `docker load`:

```
Uncompress it first
gzip -d nix-base.tgz

Load it
docker load < nix-base.tar
```

you should see an output like this:

```
202fe64c3ce3: Loading layer
↳ [=====] 80.33MB/80.33MB
e7484d5519b7: Loading layer
↳ [=====] 6.144kB/6.144kB
a0f5608ee4a8: Loading layer
↳ [=====] 645.4MB/645.4MB
475d1d69813f: Loading layer
↳ [=====] 102.9kB/102.9kB
d7963749937d: Loading layer
↳ [=====] 108.9MB/108.9MB
224a0042a76f: Loading layer
↳ [=====] 600MB/600MB
a75e978c1654: Loading layer
↳ [=====] 605.7kB/605.7kB
7efc10233531: Loading layer
↳ [=====] 1.474MB/1.474MB
```

```
Loaded image: nix-base:latest
```

or you can also use:

```
docker load -i nix-file.tar
```

to load the archive.

Since the image is available locally, it'll get used instead of pulling it from Docker Hub. So in case you cannot use Docker Hub, you could build the base images, compress them, and share them on your corporate network. Then, people can simply download them and load them and build new images on top of them.

So in summary, here's how you can share images with the world, your colleagues, or future you:

- Only share the Dockerfiles. Users need to build the images.
- Share images on Docker Hub. It's up to you if you want to share a base image with the required development environment, and then separate, smaller images for the pipelines, or if you want to share a single image which contains everything.
- Share images privately using a private registry, or by saving the image unto a file.

### 7.2.7 What if you don't use Nix?

Using Nix inside of Docker makes it very easy to setup an environment, but what if you can't use Nix for some reason? In this case, you would need to use other tools to install the right

R or Python packages to build your Docker image and it is likely that it's going to be more difficult. The main issue you will likely face is missing development libraries to successfully install R or Python packages. In this case, you will need to first install the right development library. For example, to install the and use the R `{stringr}` package, you will need to first install `libicu-dev`. Below is an example of how this may end up looking like:

```
FROM rocker/r-ver:4.5.1

RUN apt-get update && apt-get install -y \
 libglpk-dev \
 libxml2-dev \
 libcairo2-dev \
 libgit2-dev \
 default-libmysqlclient-dev \
 libpq-dev \
 libsasl2-dev \
 libsqlite3-dev \
 libssh2-1-dev \
 libxtst6 \
 libcurl4-openssl-dev \
 libharfbuzz-dev \
 libfribidi-dev \
 libfreetype6-dev \
 libpng-dev \
 libtiff5-dev \
 libjpeg-dev \
 unixodbc-dev \
 wget
```

A way to avoid that is to configure.

## 7 Containerization With Docker

Another issue that you will face is that building the image is not a reproducible process, only running containers is. To mitigate this issue you can use tagged images (like in the example above) or better yet, using a digest which you can find on Dockerhub:

```
FROM
```

```
 ↳ rocker/r-ver@sha256:1dbe7a6718b7bd8630addc45a32731624fb7b71
```

This will always pull exactly the same layers. However, this does not completely solve everything. At some point, that version of Ubuntu that you are using will be outdated, and it won't be able to download anything from repositories anymore. At that point, if you still need that image, you either need to store and keep it, or you will need to start using a newer image, and potentially have to update your code as well. Using Nix, you can stay on `ubuntu:latest`.

To summarise, if you can't use Nix inside of Docker, you will have to deal with the same issues you face when trying to setup environment on your computer.

### 7.3 Dockerizing a `{rixpress}` Pipeline

In the previous chapter, we learned how to build a fully reproducible, polyglot pipeline using Nix and `{rixpress}`. This workflow is perfect for development, ensuring that every run is bit-for-bit identical. However, what if you need to share your final data product with a collaborator or deploy it to a server where installing Nix is not an option?

This is the ideal use case for Docker. We can package our entire `{rixpress}` project (the Nix environment definition, the

pipeline logic, and all source files) into a single Docker image. This image can then be run by anyone with Docker installed, regardless of their host operating system or whether they have Nix. The user doesn't build the pipeline; they run the container, and the pre-built results are extracted. If they wish, they can "log-in" to an interactive session to access the environment within a running Docker container and "play around" with the data and code.

Let's take the `{rixpress}` project we created in the last chapter and dockerize it. Assume your project directory has the following structure:

```
.
 data/
 mtcars.csv
 gen-env.R
 gen-pipeline.R
 functions.R
 functions.py
 default.nix
 pipeline.nix
```

### 7.3.1 Step 1: The Dockerfile

Create a new file named `Dockerfile` in your project's root directory. We will use the `nix-base` image we built earlier as our foundation.

```
Use the base image with Nix pre-installed.
If you built the image locally, you can use its
local tag:
FROM nix-base:latest
```

## 7 Containerization With Docker

```
If you pushed it to Docker Hub, you (or a
↵ collaborator) can use that.
This is the more portable and recommended approach
↵ for sharing.
FROM your-username/nix-base:latest

Optionally
Set a working directory inside the container
WORKDIR /app

Copy all your project files into the image's
↵ working directory
COPY . .

Build the pipeline
This single command leverages the 'pipeline.nix'
↵ file.
Nix will first build the environment defined in
↵ 'default.nix',
then it will execute the pipeline defined in
↵ 'pipeline.nix'.
The results are stored immutably in the /nix/store
↵ inside the image.
Instead, you can also only copy the R files, and
↵ regenerate the
pipeline.nix file during the build process
RUN nix-build pipeline.nix

The CMD will define what happens when the
↵ container is run.
We will create a small R script to export the
↵ final artifact
```

```
from the Nix store to a mounted volume.
COPY export-results.R .
CMD ["Rscript", "export-results.R"]
```

This `Dockerfile` is elegant and concise because it delegates the heavy lifting of environment and pipeline management to Nix, which is its specialty.

### 7.3.2 Step 2: The Export Script

The `RUN nix-build` command has already executed our entire pipeline during the image build process. This is ideal when your data can be bundled with the image. The user running the container doesn't need to re-run the pipeline; they just need the output. (Note: If your data is too large or sensitive to be included in the image, an alternative approach is to remove the `RUN nix-build` step from the `Dockerfile` and instead execute the pipeline at runtime, using a mounted volume to provide the input data.)

In our case however, we can bundle the data in the image. The final artifacts, like our `mtcars_head` data frame, are now stored in the Nix store within the image. The user running the container doesn't need to re-run the pipeline; they just need the output.

Create a small R script named `export-results.R` to extract these results:

```
export-results.R
This script runs inside the container after the
pipeline has been built.
```

## 7 Containerization With Docker

```
Ensure {riexpress} is available to find the results
The environment is activated via the Nix build
 ↵ process.
library(riexpress)
library(jsonlite)

Define the directory where Docker will mount the
 ↵ output volume.
This path MUST match the target path in the
 ↵ `docker run -v` command.
output_dir <- "/output"
dir.create(output_dir, showWarnings = FALSE)

Read the final artifact from the completed
 ↵ pipeline
message("Reading target 'mtcars_head'...")
final_data <- rxp_read("mtcars_head")

Save the final data to the mounted directory in a
 ↵ universal format
You could also save the data in a `.csv` file
output_path <- file.path(output_dir,
 ↵ "mtcars_analysis_result.json")
write_json(final_data, output_path, pretty = TRUE)

message(paste("Successfully exported result to",
 ↵ output_path))
```

### 7.3.3 Step 3: Build and Run

With the `Dockerfile` and `export-results.R` in place, you can now build your self-contained data product.

1. **Build the Docker image:** Open a terminal in your project directory and run: `bash docker build -t my-reproducible-pipeline .`
2. **Run the container to get the results:** Now, anyone can get the result of your analysis with a single command. We will create a local `output` folder and mount it into the container. (In case you couldn't bundle the raw data into the image, then this is also how you would provide the data at run time. The pipeline would only be executed then).

```
Create a directory on your host machine to
receive the output
mkdir -p ./output

Run the container
docker run --rm --name my_pipeline_run \
-v "$(pwd)/output":/output \
my-reproducible-pipeline
```

After the container runs and exits, check your local `output` directory. You will find the `mtcars_analysis_result.json` file, containing the exact, reproducible result of your pipeline. The `docker run` command automatically executed the `CMD ["Rscript", "export-results.R"]` we defined in our `Dockerfile`, which extracted the pre-built artifact.

You have successfully packaged a complex, polyglot pipeline into a simple, portable Docker image. This workflow combines the

## 7 Containerization With Docker

best of both worlds: Nix's unparalleled power for creating reproducible builds and Docker's universal standard for distributing and running applications.

### 7.4 Further reading

- <https://www.statworx.com/content-hub/blog/wie-du-ein-r-skript-in-docker-ausfuehrst/> (in German, English translation: <https://www.r-bloggers.com/2019/02/running-your-r-script-in-docker/>)
- <https://colinfay.me/docker-r-reproducibility/>
- <https://jsta.github.io/r-docker-tutorial/>
- <http://haines-lab.com/post/2022-01-23-automating-computational-reproducibility-with-r-using-renv-docker-and-github-actions/>

### 7.5 Hands-on Exercises

#### 7.5.1 Level 1: Foundational Skills

These exercises focus on mastering the basic Docker commands and concepts.

##### Exercise 1: The Ephemeral Container and the Persistent Volume

The goal of this exercise is to solidify your understanding of how containers are ephemeral and how volumes provide persistence.

1. Run the `rocker/tidyverse` image in detached mode (`-d`), giving it a name (e.g., `my-rstudio`). Publish the port 8787 and set a password.
2. Connect to the RStudio instance in your browser. Create a new R script named `test_script.R` and save it in the default home directory (`/home/rstudio`). Inside the script, write a simple R command like `message("Hello from inside the container!")`.
3. Stop and remove the container using `docker stop` and `docker rm` (or just use `--rm` from the start).
4. Run a *new* container with the same command. Connect to RStudio again. Is your `test_script.R` still there? (It shouldn't be).
5. Now, create a folder on your local machine called `r_projects`.
6. Run the `rocker/tidyverse` container again, but this time, mount your `r_projects` folder as a volume to `/home/rstudio/projects` inside the container. The flag should look something like `-v "$(pwd)/r_projects":/home/rstudio/projects`.
7. Connect to RStudio, navigate to the `projects` folder, and create your `test_script.R` there.
8. Stop and remove the container. Check your local `r_projects` folder. Is the script there? (It should be). This demonstrates how volumes link your host machine to the container's filesystem.

## Exercise 2: The Container Inspector

This exercise is designed to get you comfortable with interacting with a running container from your terminal.

1. Run a basic `ubuntu:latest` container in detached mode (`-d`) and give it a name like `my-ubuntu-box`. Use the command `sleep 3600` to keep it running for an hour.

## 7 Containerization With Docker

- *Hint:* The full command would be `docker run -d --name my-ubuntu-box ubuntu:latest sleep 3600.`
2. Use `docker ps` to verify that your container is running.
  3. Use `docker exec` to get an interactive bash shell inside the `my-ubuntu-box` container.
  4. Once inside, run the following Linux commands: `ls -la`, `pwd`, `whoami`, and `cat /etc/os-release`. What do you observe?
  5. Still inside the container, use `apt-get update && apt-get install -y fortunes` to install a fun package. Run the `fortune` command.
  6. Exit the container's shell (type `exit`). Is the container still running? (It should be).
  7. Stop the container using `docker stop my-ubuntu-box`.

### 7.5.2 Level 2: Building and Distributing Images

These exercises focus on creating your own `Dockerfile` and sharing your work.

#### Exercise 3: Your First Custom R Image

Create a `Dockerfile` that builds a simple, non-Nix R image.

1. Create a new project folder. Inside, create a `Dockerfile`.
2. In the `Dockerfile`, start from `rocker/r-ver:4.5.1` (or any other versioned tag).
3. Add a `RUN` command to install the R package `{cowsay}` from CRAN.
4. Create an R script named `run.R` in the same folder. The script should contain:  
`r library(cowsay)  
say("Moo-ving to Docker!", by = "cow")`

5. In your `Dockerfile`, add a `COPY` command to copy `run.R` into the image (e.g., to `/home/run.R`).
6. Set the `CMD` to execute your script using `Rscript`.
7. Build the image with the tag `my-cowsay-app`.
8. Run a container from your new image. You should see the cow's message printed to your terminal.

### Exercise 4: Publish Your Nix Base

Take the `nix-base` image you created in the chapter and practice the full distribution workflow.

1. If you haven't already, build the `nix-base` image locally.
2. Create a free account on Docker Hub.
3. Log in to Docker Hub from your terminal using `docker login`.
4. Tag your `nix-base:latest` image with your Docker Hub username, e.g., `docker tag nix-base:latest your-username/nix-base:1.0`.
5. Push the image to Docker Hub using `docker push`.
6. To test that it works, remove your local copy of the image: `docker rmi your-username/nix-base:1.0` and `docker rmi nix-base:latest`.
7. Create a simple `Dockerfile` that starts with `FROM your-username/nix-base:1.0`. When you build this `Dockerfile`, Docker should pull the image you just pushed from Docker Hub.

### 7.5.3 Level 3: The Capstone Project

This exercise integrates all the concepts from the chapter into a complete, reproducible data product.

### Exercise 5: Package Your Own `{rrixpress}` Pipeline

## 7 Containerization With Docker

Take a `{rixpress}` pipeline (you can use the one from the previous chapter’s exercises or create a new one) and package it into a distributable Docker image.

1. Your project should contain all the necessary files: `gen-env.R`, `gen-pipeline.R`, your `default.nix` and `pipeline.nix`, and any data/function scripts.
2. Create a `Dockerfile` that uses your published `nix-base` image from Exercise 4 as its `FROM` source.
3. The `Dockerfile` should:
  - Set a working directory (e.g., `/app`).
  - `COPY` all your project files into the image.
  - `RUN` the `nix-build pipeline.nix` command to execute the pipeline *during the build process*.
  - Include an `export-results.R` script (you’ll need to write this) that saves one or more of your final pipeline artifacts to an `/output` directory.
  - Set the `CMD` to run your `export-results.R` script.
4. Build the final image with a descriptive tag (e.g., `my-final-analysis:latest`).
5. Run the image, mounting a local `output` folder to the container’s `/output` folder.
6. Verify that the final results (e.g., a plot, a CSV, or a JSON file) appear in your local `output` folder. **Congratulations, you’ve created a fully portable and reproducible data product!**

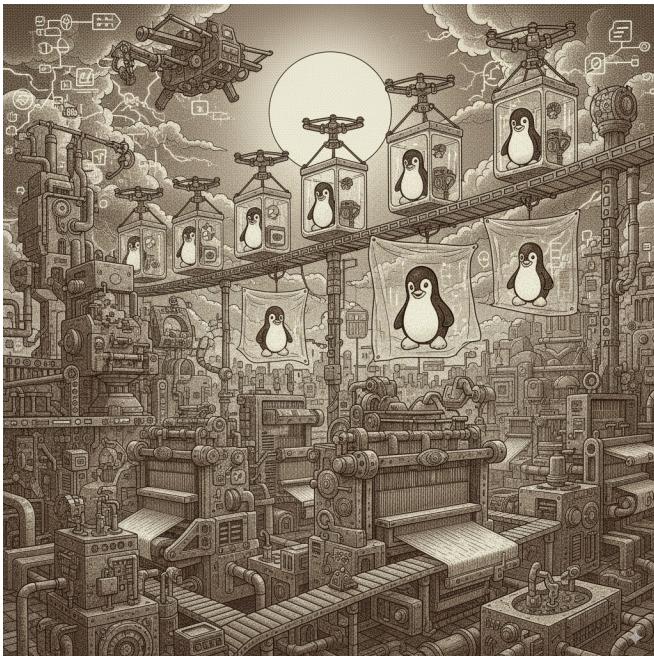
**Extra Challenge:** Modify your solution from Exercise 5 for a “big data” scenario where the input data cannot be included in the image. \* Your `Dockerfile` should **not** copy the data and should **not** run `nix-build` at build time. \* Instead, the `CMD` should execute the entire pipeline at runtime. You will need to figure out how to pass the paths for input data and output

## 7.5 Hands-on Exercises

results to your script. \* The `docker run` command will now need to mount *two* volumes: one for the input data and one for the output results.



# 8 Intro to CI/CD with Github Actions



What you'll have learned by the end of the chapter: very basic knowledge of Github Actions, but enough to run your RAP in the cloud.

## 8.1 Introduction

We are almost at the end; actually, we could have stopped at the end of the previous chapter. We have reached our goal; we are able to run pipeline in a 100% reproducible way. However, this requires some manual steps. And maybe that's not a problem; if your image is done, and users only need to pull it and run the container, that's not really a big problem. But you should keep in mind that manual steps don't scale. Let's imagine another context; let's suppose that you are part of a company and that you are part of a team that needs to quickly ship products to clients. Maybe several people contribute to the product using an internal version control solution (like a Gitlab instance that is deployed on the premises of your company). Maybe you even need to work on several products in the same day; you (and your teammates) should only be focusing writing code (and `Dockerfiles`)... your time and resources cannot get clogged by building images (which depending on what you're working on, can take quite some time). So ideally, we would want to automate this step. That is what we are going to learn in this chapter.

This chapter will introduce you to the basic ideas of CI/CD (Continuous Integration and Continuous Deployment/Delivery) and DevOps with Github Actions. Because we're using Git to trigger all the events and automate the whole pipeline, this can also be referred to as GitOps. What's Dev(Git)Ops? I think that the Atlassian page on DevOps makes a good job of explaining it. The bottom line is that DevOps makes it easy for developers to focus on coding, and makes it easy for them to ship data products. The core IT team provides the required infrastructure and tools to make this possible. GitOps is a variant of DevOps where the definition of the infrastructure is versioned, and can be changed by editing simple text files. Through events,

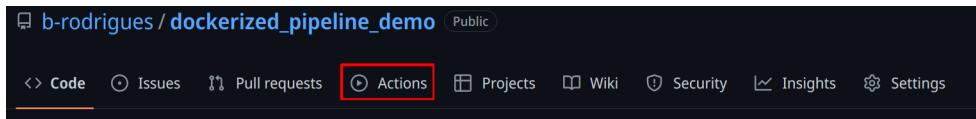
## 8.2 Getting your repo ready for Github Actions

such as pushing to the repository, new images can be built, or containers executed. Data products can then also be redeployed automatically. All the steps we've been doing manually, with one simple push! It's also possible, in the context of package development, to execute unit tests when code gets pushed to repo, or get documentation and vignettes compiled. This also means that you could be developing on a very thin client with only a text editor and git installed. Pushing to Github would then execute everything needed to have a package ready for sharing.

So our goal here is, in short, to do exactly the same as what we have been doing on our computer (so build an image, run a container, and get back 3 plots), but on Github.

## 8.2 Getting your repo ready for Github Actions

You should see an “Actions” tab on top of any Github repo:



This will open a new view where you can select a lot of available, ready to use actions. Shop around for a bit, and choose the right one (depending on what you want to do). You should know that there is a very nice repository with many actions for R. Once you're done choosing an action, a new view in which you can edit a file will open. This file will have the name of the chosen action, and have the `.yml` extension. This file will be automatically added to your repository, in the following path: `.github/workflows`.

## 8 Intro to CI/CD with Github Actions

Let's take a look at such a workflow file:

```
name: Hello world
on: [push]
jobs:
 say-hello:
 runs-on: ubuntu-latest
 steps:
 - run: echo "Hello from Github Actions!"
 - run: echo "This command is running from an
Ubuntu VM each time you push."
```

Let's study this workflow definition line by line:

```
name: Hello world
```

Simply gives a name to the workflow.

```
on: [push]
```

When should this workflow be triggered? Here, whenever something gets pushed.

```
jobs:
```

What is the actual things that should happen? This defines a list of actions.

```
 say-hello:
```

This defines the `say-hello` job.

```
 runs-on: ubuntu-latest
```

This job should run on an Ubuntu VM. You can also run jobs on Windows or macOS VMs, but this uses more compute minutes than a Linux VM (you have 2000 compute minutes for free per month).

## 8.2 Getting your repo ready for Github Actions

steps:

What are the different steps of the job?

- run: echo "Hello from Github Actions!"

First, run the command echo "Hello from Github Actions!". This command runs inside the VM. Then, run this next command:

- run: echo "This command is running from an Ubuntu VM each time you push."

Let's push, and see what happens on github.com:

If we take a look at the commit we just pushed, we see this yellow dot next to the commit name. This means that an action is running. We can then take a look at the output of the job, and see that our commands, defined with the `run` statements in the workflow file, succeeded and echoed what we asked them.

So, the next step is running our Docker image and getting back our plots. This next example can be found in this repository. This example doesn't use Nix, `{rix}` nor `{rixpress}`, but the point here is to show how a Docker image can be executed on GitHub Actions, and artifacts can be recovered. The process is always the same, regardless is inside the Docker image.

This is what our workflow file looks like:

```
name: Reproducible pipeline

on:
 push:
 branches: ["main"]
 pull_request:
 branches: ["main"]
```

## 8 Intro to CI/CD with Github Actions

```
jobs:

 build:

 runs-on: ubuntu-latest

 steps:
 - uses: actions/checkout@v5
 - name: Build the Docker image
 run: docker build -t my-image-name .
 - name: Docker Run Action
 run: docker run --rm --name
 my_pipeline_container -v
 /github/workspace/fig/:/home/graphs/:rw
 my-image-name
 - uses: actions/upload-artifact@v4
 with:
 name: my-figures
 path: /github/workspace/fig/
```

For now, let's focus on the `run` statements, because these should be familiar:

```
run: docker build -t my-image-name .
```

and:

```
run: docker run --rm --name my_pipeline_container -v
 /github/workspace/fig/:/home/graphs/:rw
 my-image-name
```

The only new thing here, is that the path has been changed to `/github/workspace/`. This is the home directory of your repository, so to speak. Now there's the `uses` keyword that's new:

## 8.2 Getting your repo ready for Github Actions

```
uses: actions/checkout@v5
```

This action checkouts your repository inside the VM, so the files in the repo are available inside the VM. Then, there's this action here:

```
- uses: actions/upload-artifact@v4
 with:
 name: my-figures
 path: /github/workspace/fig/
```

This action takes what's inside `/github/workspace/fig/` (which will be the output of our pipeline) and makes the contents available as so-called “artifacts”. Artifacts are the outputs of your workflow. In our case, as stated, the output of the pipeline. So let's run this by pushing a change, and let's take a look at these artifacts!

As you can see from the video above, a zip file is now available and can be downloaded. This zip contains our plots! It is thus possible to rerun our workflow in the cloud. This has the advantage that we can now focus on simply changing the code, and not have to bother with boring manual steps. For example, let's change this target in the `_targets.R` file:

```
tar_target(
 commune_data,
 clean_unemp(unemp_data,
 place_name_of_interest =
 c("Luxembourg", "Dippach",
 "Wiltz",
 "Esch/Alzette",
 "Mersch",
 "Dudelange"),
 col_of_interest = active_population)
)
```

## *8 Intro to CI/CD with Github Actions*

I've added "Dudelange" to the list of communes to plot. Let me push this change to the repo now, and let's take a look at the artifacts. The video below summarises the process:

As you can see in the video, the `_targets.R` script was changed, and the changes pushed to Github. This triggered the action we've defined before. The plots (artifacts) get refreshed, and we can download them. We see then that Dudelange was added in the `communes.png` plot!

It is also possible to “deploy” the plots directly to another branch, and do much, much more. I just wanted to give you a little taste of Github Actions (and more generally GitOps). The possibilities are virtually limitless, and I still can't get over the fact that Github Actions is free for public repositories.

## **8.3 Building a Docker image and pushing it to a registry**

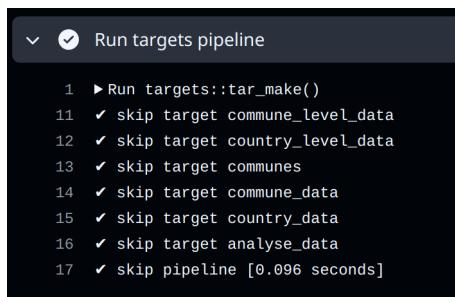
It is also possible to build a Docker image and have it made available on an image registry. You can see how this works on this repository. This images can then be used as a base for other RAPs, as in this repository. Why do this? Well because of “separation of concerns”. You could have a repository which builds in image containing your development environment: this could be an image with a specific version of R and R packages built with Nix. And then have as many repositories as projects that run RAPs using that development environment image as a basis. Simply add the project-specific packages that you need for each project.

## 8.4 Running a pipeline straight from Github Actions

Using Docker on Github Actions has the advantage that you can use the same image to develop locally on your computer, and then also on CI. However, you could also run the pipeline straight from a Github Actions runner, but it'll take some effort to set up the environment on CI. Take a look at the example from this repository.

The yaml file used in this action (which you can find [here](#)) was generated by running `targets::tar_github_actions()` and was then modified further, mostly to add the required development libraries to compile the needed R packages (under the `Install Linux system dependencies` step).

This action takes advantage of the included Github Actions cache to backup the targets from the pipeline, so they can also get skipped with subsequent runs:



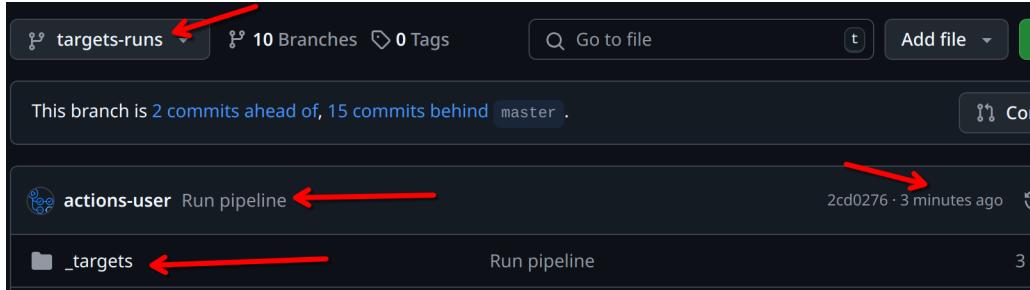
```
Run targets pipeline

1 ► Run targets::tar_make()
11 ✓ skip target commune_level_data
12 ✓ skip target country_level_data
13 ✓ skip target communes
14 ✓ skip target commune_data
15 ✓ skip target country_data
16 ✓ skip target analyse_data
17 ✓ skip pipeline [0.096 seconds]
```

This can also be achieved with Docker by mounting volumes, but requires more manual setup.

Another difference with Docker is that the outputs are not saved as an artifact, but instead get pushed to the `targets-runs` branch:

## 8 Intro to CI/CD with Github Actions



The previous examples assumed you didn't use Nix, but if you did, you can also run pipelines from inside Nix and with `{riexpress}`. Look at this `yaml` file from the `riexpress_demos` repository that shows how to run a `{riexpress}` pipeline on GitHub Actions link.

## 8.5 Running unit tests on Github Actions

Setting up your project as a package (or at least, the parts of your project that can be reused for others) as a package also has the advantage that it becomes very easy to run unit tests on CI. See for example the `{myPackage}` package that we developed together, in particular this file. This action runs on each push and pull request on Windows, Ubuntu and macOS:

```
on:
 push:
 branches: ["main"]
 pull_request:
 branches: ["main"]

jobs:
 rcmdcheck:
 runs-on: ${{ matrix.os }}
```

## 8.5 Running unit tests on Github Actions

```
strategy:
 matrix:
 os: [ubuntu-latest, windows-latest,
 macos-latest]
```

Several steps are executed, all using pre-defined actions from the `r-lib` project:

```
steps:
 - uses: actions/checkout@v4
 - uses: r-lib/actions/setup-r@v2
 - uses: r-lib/actions/setup-r-dependencies@v2
 with:
 extra-packages: any::rcmdcheck
 needs: check
 - uses: r-lib/actions/check-r-package@v2
```

An action such as `r-lib/actions/setup-r@v2` will install R on any of the supported operating systems without requiring any configuration from you. If you didn't use such an action, you would need to define three separate actions: one that would be executed on Windows, on Ubuntu and on macOS. Each of these operating-specific actions would install R in their operating-specific way.

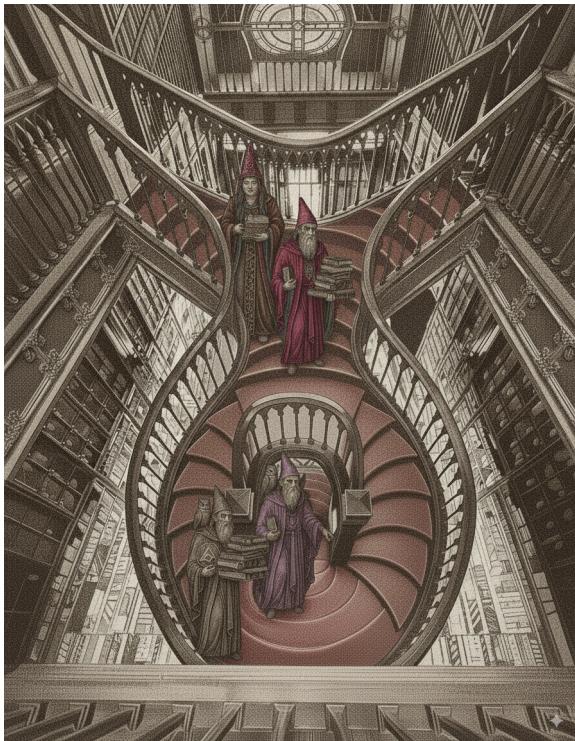
Check out the workflow results to see how the package could be improved here.

Here again, using Nix simplifies this process immensely. Look at this workflow file from `{rix}`'s repository here. Setting up the environment is much easier, as is running the actual test suite.

## 8.6 Further reading

- <http://haines-lab.com/post/2022-01-23-automating-computational-reproducibility-with-r-using-renv-docker-and-github-actions/>
- [https://orchid00.github.io/actions\\_sandbox/](https://orchid00.github.io/actions_sandbox/)
- <https://www.petefreitag.com/item/903.cfm>
- <https://dev.to/mihinduranasinghe/using-docker-containers-in-jobs-github-actions-3eof>

# 9 What else should you learn?



Here's a list of things I think would be nice for you to invest some time in, in no particular order.

## 9.1 Touch typing

One of the things I NEVER see discussed when talking “up-skilling” is improving your typing speed. According to a survey (which I’m sure is not statistically, nor scientifically sound, but still...) by onlinetyping.org (which you can find here, most back office workers (who spend all day typing) have a typing speed of 20 to 30 wpm (words per minute). According to this article by the Atlantic people write about 41638 words in email per year. You as programmers (yes, even if you’re focused on data, you’re a programmer) very surely type twice or thrice this amount of words per year. But let’s stay with 41638 words per year. That would translate to almost 28 days of non stop typing at a typing speed of 25 words per minute. Doubling to 50 wpm is actually quite easy, and reaching 70 is really doable. This could improve productivity, or better yet, make you go home earlier instead of working until 19h00 every day because you type like a snail.

You need to learn touch typing, meaning, typing without looking at your keyboard.

## 9.2 Vim

Yes, I think you should learn vim, or at the very least, your text editor of choice, by heart. You should know every keyboard shortcut and every possibility that your text editor offers. You should never touch the mouse when writing text. This is not just because of productivity, but also for your health. Grabbing the mouse to click one or twice, and then go back to typing, then go back to moving the mouse, etc, will destroy your shoulder. By keeping your hands on the keyboard at all times and minimizing mouse usage, you may be able to grow old healthy. Vim helps

with that because it is a modal text editor (and most editors actually ship a Vim-mode). Watch this video to get a quick introduction on Vim, and how to enable Vim mode in Vscode.

## **9.3 Statistical modeling**

Statistical modeling is crucial, and if you didn't major in stats, you very likely lack this knowledge. Here's a reading (and watching) list:

- Regression and other stories (has a free PDF)
- Statistical Rethinking 2022 (on youtube)
- Mostly harmless econometrics



# 10 Conclusion

## 10.1 Why bother

We're at the end of this course, which I hope you enjoyed. There is now yet another question we need to ask ourselves: is this worth it? Why should we bother with making our pipelines reproducible? I believe that there are two, fundamental, essential reasons.

The first one, is that time is finite, and working manually does not scale. Reproducible pipelines do take time to set up, but they allow us to win this time back once we start re-running them. Wasting time and resources running things manually (with the potential for introducing errors) is simply not acceptable. This freed up time can then be used to provide further value to your employer, yourself, and ideally your community as well.

The second reason, is that setting up RAPs is in itself an enjoyable activity, which requires the full depth and breadth of your skills. If you're working in science, there is the added benefit that by setting up a RAP you're doing actual science: providing a reproducible analysis where an hypothesis gets tested (an not writing papers).

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

