Building Reproducible Analytical Pipelines

Master of Data Science, University of Luxembourg - 2025

Bruno Rodrigues

2025-09-15

Table of contents

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

- 2025/09/04 - 4 hours,
- 2025/09/11 - 4 hours,
- 2025/09/14 - 4 hours,
- 2025/09/02 - 4 hours,
- 2025/09/05 - 2 hours,
- 2025/09/09 - 5 hours,
- 2025/09/16 - 4 hours,
- 2025/09/19 - 3 hours,

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. In this course we will not focus too much on how to create automated reports or web apps (but I'll give an introduction to these, don't worry) but our focus will be 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, 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 dependency management tools);
- To be written with an open source programming language (nocode tools like Excel are by default non-reproducible because they can't be used non-interactively, and which is why we are going to use the R programming language);
- The project needs to be run on an open source operating system (thankfully, we can deal with this without having to install and learn to use a new operating system, thanks to Docker);
- Data and the paper/report need obviously to be accessible as well, if not publicly as is the case for research, then within your company.

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.

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}`, `{tidyr}`, `{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)

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

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

That being said, the reality of data science is that the future is becoming more and more polyglot. Data products are evermore complex, and necessity are 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. We are going first to learn how to package this dataset alongside some functions to make it clean. 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;
- Bonus points: pipeline is dockerized, or uses Nix, and/or uses Github Actions to run? 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.

Jargon

There's some jargon that is helpful to know when working with R. Here's a non-exhaustive list to get you started:

- 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;
- 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 constrast 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.

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)

License

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
- Install Nix
- Use `{rix}` to generate `default.nix` files
- Build cross-language environments for data work or software development

1.2 Why Reproducibility? Why Nix? *(2h)*

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 fullfill 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 native system dependencies (like `libxml2`, `pandoc`, or `curl`), and require users to "rebuild" their environments from partial metadata. Docker helps but introduces overhead, security challenges, and complexity.

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 the end also as a build automation tool).

What's a package manager? If you're not a Linux user, you may not know. 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) 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 and the R packages that we require for our analysis (or any other programming language that we need). 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.

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

The idea of `{rix}` is for you to declare the environment you need using the provided `rix()` function. `rix()` is the package's main function and 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 for each other, but can still interact with your system's files, unlike with Docker where a volume must be mounted. Environments can also interact with the software installed on your computer through the usual means, which 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

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:

```
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.

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.

Do not use your operating system's package manager to install Nix. Instead, simply open a terminal and run the following line (on Windows, if you cannot or have decided not to activate systemd, then you have to append `--init none` to the command. You can find more details about this on The Determinate Nix Installer page):

```
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!

1.2.6 Temporary shells

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

Open a terminal and run:

```
which sl
```

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 *(2h)*

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 package 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/2025-04-
  ↪  {};

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. 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
```

```
library(rix)
```

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

```
rix(
  date = "2025-06-02",
  r_pkgs = c("dplyr", "ggplot2"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "positron",
  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 2nd of June 2025 on CRAN. This will also add Python 3.13 and the `ploars` and `great-tables` Python packages as they were at the time in `nixpkgs` (more on this later). Finally, this also add the Positron IDE, which is a fork of VS Code for data science. This is just an example, and you can use another IDE if you wish. See this vignette for learning how to setup your IDE with Nix.

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, pin the version of Nixpkgs used. Replace `<nixpkgs>` with a fixed import:

```
let
  pkgs = import (fetchTarball
   ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/2025-06
   ↪  {};
in
pkgs.mkShell {
  buildInputs = [ pkgs.r pkgs.rPackages.dplyr ];
}
```

This avoids unexpected updates and lets others reproduce your environment exactly.

1.4 Configuring your IDE

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 RStudio, VS Code and Positron.

The following are the setups we recommend you use to work using an IDE and Nix environments. To be recommended, a setup should:

- be easy to setup;
- work the same on any operating system;
- not require any type of special maintenance.

Regardless of your operating system, a general-purpose editor such as VS Code (or Codium), Emacs, or Neovim meets the above requirements. Recent releases of Positron also work quite well. (Note: Neovim is not covered here due to lack of experience—PRs welcome!) However, some editors perform better on certain platforms.

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.

1.4.1 Recommended setup on macOS

On macOS, RStudio will only be available through Nix and only for versions 4.4.3 or more recent, or after the 2025-02-28 if you're using dates. For older versions of R or dates, RStudio is not available for macOS through Nix so you cannot use it. As such, we recommend either VS Code (or Codium) or Positron for older dates or versions. Emacs or Neovim are also good options. See

the relevant sections below to set up any of these editors. We also recommend to install the editor on macOS directly, and configure it to interact with Nix shells, instead of using Nix to install the editor, even though it does take some more effort to configure.

1.4.2 Recommended setup on Windows

On Windows, since you have to use Nix through WSL, your options are limited to editors that either:

- can be installed on Windows and interact with WSL, or
- can be launched directly from WSL.

We recommend to use an editor you can install directly on Windows and configure to interact nicely with WSL, and it turns out that this is mostly only VS Code (or Codium) or Positron. See this section to learn how to configure VS Code (or Codium) or Positron.

If you want to use RStudio, this is also possible but:

- RStudio should ideally be installed with Nix inside WSL;
- your version of Windows needs to support WSLg which should be fine on Windows 11 or the very latest Windows 10 builds. WSLg allows you to run GUI apps from WSL.

You should also be aware that there is currently a bug in the RStudio Nix package that makes RStudio ignore project-specific `.Rprofile` files, which can be an issue if you also have a system-level library of packages. Instead, you can sure the `.Rprofile` generated by `rix()` yourself or you can uninstall the system-level R and library of packages.

Furthermore, be aware that there is a bug in WSLg that prevents modifier keys like Alt Gr from working properly.

If you prefer Emacs or Neovim, then we recommend to install it in WSL and use it in command line mode, not through WSLg (so starting Emacs with the `-nw` argument).

1.4.3 Recommended setup on Linux

On Linux distributions, the only real limitation is that RStudio cannot interact with Nix shells (just like on the other operating systems), so if you want to use RStudio then you need to install it using Nix.

You should also be aware that there is currently a bug in the RStudio Nix package that makes RStudio ignore project-specific `.Rprofile` files, which can be an issue if you also have a system-level library of packages. Instead, you can sure the `.Rprofile` generated by `rix()` yourself or you can uninstall the system-level R and library of packages.

If you use another editor, just follow the relevant instructions below; the question you need to think about is whether you want to use Nix to install the editor inside of the development shell or if you prefer to install your editor yourself using your distribution's package manager, and configure it to interact with Nix shells. We recommend the latter option, regardless of the editor you choose.

1.4.4 RStudio

RStudio **must** be installed by Nix in order to *see* and use Nix shells. So you cannot use the RStudio already installed on your

computer to work with Nix shells. This means you need to set `ide = "rstudio"` if you wish to use RStudio.

You should also be aware that there is currently a bug in the RStudio Nix package that makes RStudio ignore project-specific `.Rprofile` files, which can be an issue if you also have a system-level library of packages. Instead, you can sure the `.Rprofile` generated by `rix()` yourself or you can uninstall the system-level R and library of packages.

1.4.4.1 RStudio on macOS

To use RStudio on macOS simply use `ide = "rstudio"`, but be aware that this will only work for R version 4.4.3 at least, or for a date on or after the 2025-02-28. If you don't need to work with older versions of R or older date, RStudio is an appropriate choice. Then, build the environment using `nix-build` and drop into the shell using `nix-shell`. Then, type `rstudio` to start RStudio. If you wish, you can even put the `rstudio` command in the shell hook to start it immediately as you run `nix-shell`.

1.4.4.2 RStudio on Linux or Windows

To use RStudio on Linux or Windows simply use `ide = "rstudio"`. Then, build the environment using `nix-build` and drop into the shell using `nix-shell`. Then, type `rstudio` to start RStudio.

If you plan to use RStudio on Ubuntu, then you need further configuration to make it work, because of newly introduced sandboxing features in Ubuntu 24.04. You will need to create an RStudio-specific AppArmor profile. To do so create this apparmor profile:

```
sudo nano /etc/apparmor.d/nix.rstudio
```

Populate it with:

```
profile nix.rstudio
/nix/store/*-RStudio-*-wrapper/bin/rstudio
flags=(unconfined) {
    userns,
}
```

Save it, load the profile and start RStudio:

```
sudo apparmor_parser -r /etc/apparmor.d/nix.rstudio
sudo systemctl reload apparmor
```

You can now start RStudio from the activated Nix shell.

On Windows, you need to have `WSLg` enabled, which should be the case on the latest versions of Windows. If you wish, you can even put the `rstudio` command in the shell hook to start it immediately as you run `nix-shell`.

On Linux and WSL, depending on your desktop environment, and for older versions of RStudio, you might see the following error message when trying to launch RStudio:

```
qt.glx: qglx_findConfig: Failed to finding matching
FBConfig for QSurfaceFormat(version 2.0, options
QFlags<QSurfaceFormat::FormatOption>(),
depthBufferSize -1, redBufferSize 1, greenBufferSize
1, blueBufferSize 1, alphaBufferSize -1,
stencilBufferSize -1, samples -1, swapBehavior
QSurfaceFormat::SingleBuffer, swapInterval 1,
colorSpace QSurfaceFormat::DefaultColorSpace,
profile  QSurfaceFormat::NoProfile)
```

```
Could not initialize GLX
Aborted (core dumped)
```

in this case, run the following before running RStudio:

```
export QT_XCB_GL_INTEGRATION=none
```

To use GitHub Copilot with RStudio, follow these instructions.

1.4.5 VS Code or Positron

Positron is a fork of VS Code made by Posit and tailored for data science. Henceforth, I will refer to both editors simply as *Code*.

The same instructions apply whether your host operating system is Linux, macOS or Windows. The first step is of course to install Code on your operating system using the usual means of installing software.

If you're on Windows, install Code on Windows, not in WSL. Code on Windows is able to interact with WSL seamlessly and before continuing here, please follow these instructions (it's mostly about installing the right extensions after having installed Positron).

On macOS, start by installing Code using the official `.dmg` installer. Start Code, and then the command palette using `COMMAND-SHIFT-P`. In the search bar, type `"Install 'positron' command in PATH"` and click on it: this will make it possible to start Positron from a terminal.

Once Code 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 Code. Follow the instructions for your operating system here but if you're using Windows, install `direnv` in WSL (even though you've just installed Code for Windows), so follow the instructions for whatever Linux distribution you're using there (likely Ubuntu), or use Nix to install `direnv` if you prefer (this is the way I recommend to install it on macOS, unless you already use `brew`):

```
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.

Then, in Code, install the direnv extension (and also the WSL extension if you're on Windows, as explained in the official documentation linked above!). Finally, add a file called `.envrc` and simply write the following two lines in it:

```
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 > 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. 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 Code should not prompt you anymore.

If you're on Windows, using Code like this is particularly interesting, because it allows you to install Code 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 VS Code to use GitHub Copilot, click here or for Positron click here.

1.5 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 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. So buckle up, because this chapter is not going to be easy.

Git is incredibly powerful, and 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 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.

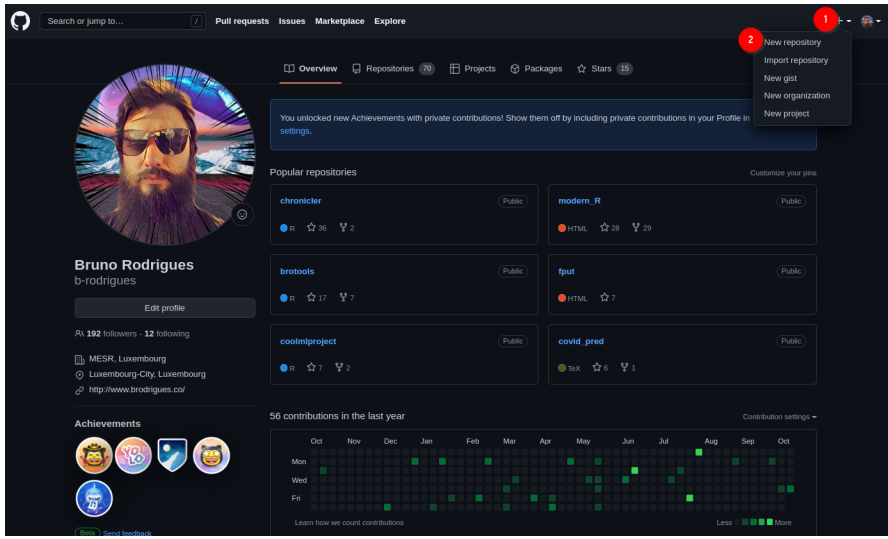
It would also be possible to install it with Nix, but because Git is also useful outside of development shells, it is better to have it installed at the level of your operating system.

Next, configure git:

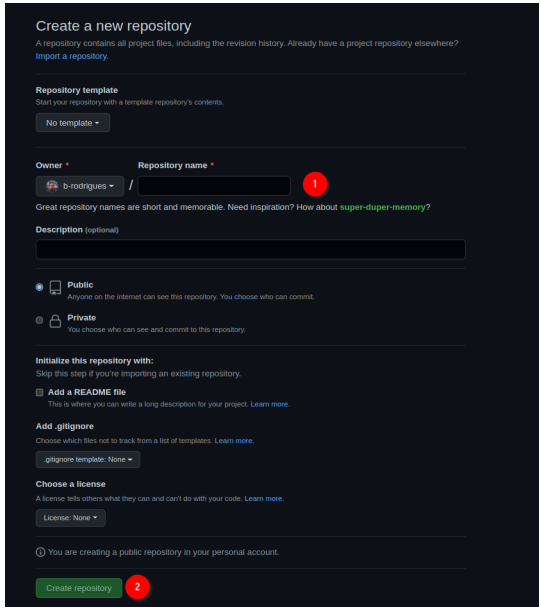
```
git config --global user.name "Your Name"
git config --global user.email
 ↪  "your.email@example.com"
```

2.3 Setting up a repo

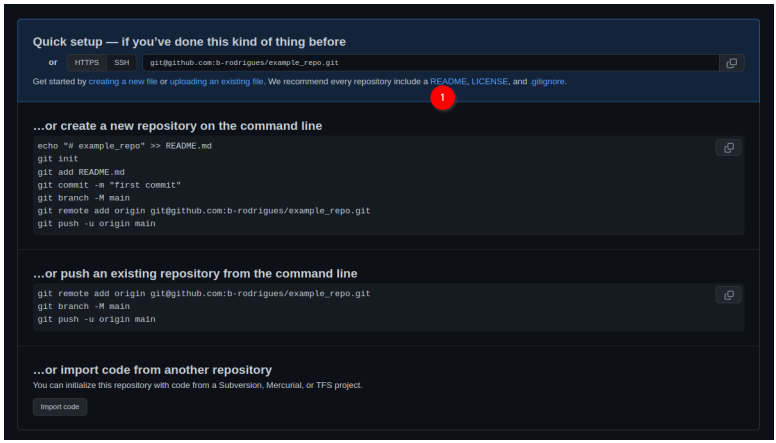
Ok so now that Git is installed, we can actually start using it. First, let's start by creating a new repository on github.com. As I've mentioned in the introductory paragraph, Git will allow you to interact with github.com, and you'll see in what ways soon enough. For now, login to your github.com account,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



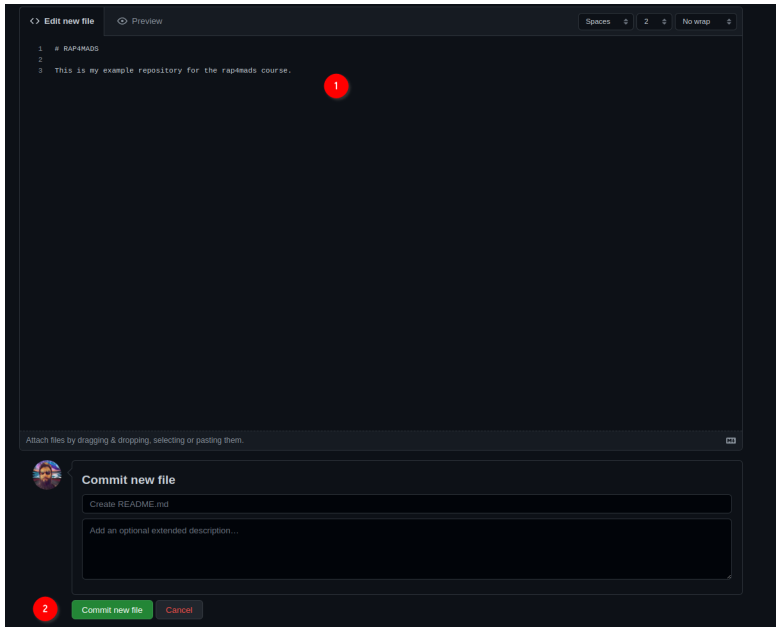
In the next screen, choose a nice name for your repository and ignore the other options, they're not important for now. Then click on 'Create repository':



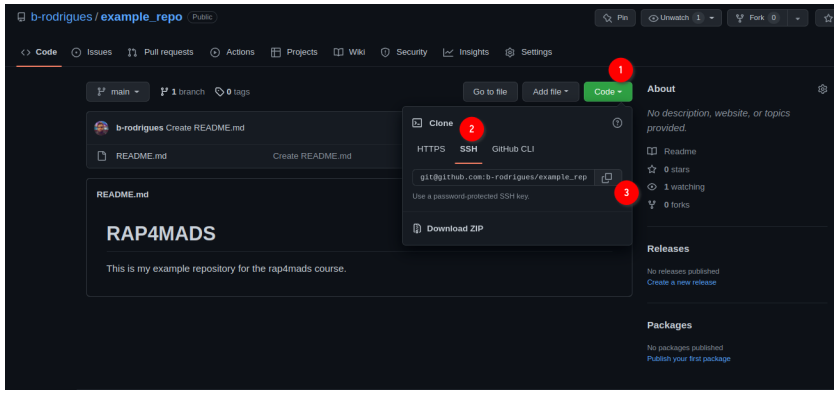
Ok, we're almost done with the easy part. The next screen tells us we can start interacting with the repository. For this, we're first going to click on 'README':



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



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:



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:

```
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):

```
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:

```
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
```

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:

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)' > iris_analysis.R
git add iris_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`.

```
      C (Your local work)
     /
A---B ---D (Remote work on GitHub)
```

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

```
      C-------E (Merge commit)
     /       /
A---B-------D
```

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. 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:

```
<<<<<<< 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:

```
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. * **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.
  # 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. `bash 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. **Create the second branch *from the first branch*.** This is the key step. `bash 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:**

```
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:

```
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)

 # 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 - `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,?]? s
```

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

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

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 codebase while benefiting from LLM assistance, with complete traceability of what changes were made and why.

3 Functional Programming: The Cornerstone of Reproducible Analysis

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" func-

tions in both R and Python. * How to use functional concepts like `map`, `filter`, and `reduce` to replace error-prone loops. * 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.

3.1.1 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.
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:

```
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 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.

3.2.2.1 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.

```r
# 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
  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

# 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.

- `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.

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.

3.4.1.2 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.

```python
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.

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

```r
# 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.

```
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
print(total_sum)
```

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.

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
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()
# )
```

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.

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

R version (`utils.R`):

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

Python version (`utils.py`):

```python
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.

```r
# 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 `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)
```

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.

4.4 Testing as a Design Tool

Here is where testing becomes a superpower. What happens if we try to normalize a vector where all the elements are the same? The standard deviation will be 0, leading to a division by zero!

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]`
    # 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**. This is great! 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.

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.

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.

R (`my_functions.R`):

```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`):

```python
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_functions.py`). 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`):

```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`):

```python
# 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.

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 Building Reproducible Pipelines with Nix and {rixpress}

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.

5.1 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 `{rix}` to ensure everyone uses the exact same versions of R, Python, 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

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 limitaton 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

(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.

5.2 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`.

5.2.1 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
```

5.2.2 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`.

5.2.3 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='|')"
  ),
  # Note: polars must be converted to pandas for
   ↪  reticulate
  rxp_py(
    name = mtcars_pl_am,
    py_expr = "mtcars_pl.filter(polars.col('am') ==
     ↪  1).to_pandas()"
```

```
),
...
```

The next derivation is defined using `rxp_py()` which runs Pyton code. As you can see, the `py_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.

5.2.3.1 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:

```
rxp_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:

```
rxp_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.

5.2.3.2 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,
    py_expr = "mtcars_pl.filter(polars.col('am') ==
     ↪  1)",
    additional_files = "functions.py",
    serialize_function = "serialize_to_json" # Use
     ↪  our Python helper
  ),

  rxp_r(
    name = mtcars_head,
    expr = my_head(mtcars_pl_am),
    additional_files = "functions.R",
    unserialize_function = "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.

5.2.4 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,
    py_expr = "mtcars_pl.filter(polars.col('am') ==
     ↪  1).to_pandas()"
  ),
  rxp_py(
    name = mtcars_pl_am,
    py_expr = "mtcars_pl.filter(polars.col('am') ==
     ↪  1)",
    additional_files = "functions.py",
    serialize_function = "serialize_to_json" # Use
     ↪  our Python helper
  ),

  rxp_r(
    name = mtcars_head,
    expr = my_head(mtcars_pl_am),
```

```
    additional_files = "functions.R",
    unserialize_function = "jsonlite::fromJSON" #
     ↪ Use R's jsonlite
  ),
) |>
  rixpress()
```

The very last function, `rixpress()` 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. Once it's 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)
```

You can also only generate the required code, but not run the pipeline yet, by setting `build = FALSE` in `rixpress()`.

5.3 Caching

First, visualize your pipeline's dependency graph:

```
# You'll need to firts generate the required files
 ↪  by running
# `rixpress(...)` or `rixpress(..., build = FALSE)`
 ↪  first
# Then you can visualize the graph
rxp_ggdag()
```

This will show you a clear, unambiguous graph of your workflow.

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.

5.4 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.

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_z9y8x7w6v5u4t3s2r1q0p9o8n7m6l5k
 ↪  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.

5.5 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.

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