Reproducible Polyglot Data Science

Bruno Rodrigues

2025-12-31

Table of contents

Welcome! **1**

A modern, unified, and language-agnostic workflow for data science using Nix. 1

Preface **5**

1 Introduction **9**

1.1 Who is this book for? 9
1.2 What is the aim of this book? 10
1.3 Prerequisites 12
1.4 What actually is reproducibility? 14
1.4.1 Using open-source tools is a hard requirement 14
1.4.2 Hidden dependencies can hinder reproducibility 17
1.4.3 The requirements of a RAP 18
1.4.4 Are there different types of reproducibility? 19

2 The Nix Package Manager **25**

2.1 Introduction 25
2.2 Why Reproducibility? Why Nix? 26
2.2.1 Motivation: Reproducibility in Scientific and Data Workflows 26
2.2.2 Problems with Ad-Hoc Tools 26
2.2.3 Nix: A Declarative Solution 27

2.3 Important Concepts 28
2.3.1 Derivations 29
2.3.2 Dependencies of derivations 30
2.3.3 The Nix store and hermetic builds 31
2.3.4 Other key Nix concepts 32
2.4 Caveats 34
2.5 In summary 36

3 Setting Up Your Environment 37
3.1 Installing Nix 37
3.1.1 For Windows Users: WSL2 Prerequisites . 37
3.1.2 The Determinate Systems installer 38
3.2 Configuring the Cache 39
3.3 Verifying installation with Temporary Shells . . . 40
3.4 Configuring your IDE 42
3.4.1 Pre-requisites 42
3.4.2 direnv 43
3.4.3 In summary and next steps 44

4 Reproducible Development Environments with rix 47
4.1 Introduction 47
4.1.1 The Reproducibility Challenge 47
4.1.2 Component Closures: The Nix Approach . 48
4.1.3 The Polyglot Challenge 48
4.1.4 Enter rix 49
4.2 Transitioning to Nix-Managed R 49
4.2.1 Bootstrapping `{rix}` without a local R installation 50
4.3 The rix() Function 52
4.4 Choosing an R Version or Date 54
4.4.1 Using dates vs versions 55
4.4.2 The rstats-on-nix fork 55
4.5 Installing R Packages 56
4.5.1 From CRAN/Bioconductor 56

4.5.2 Installing archived versions 56
4.5.3 Installing from GitHub 57
4.5.4 Installing local packages 58
4.5.5 Why NOT to use install.packages() 59
4.6 System Tools and TexLive 60
4.6.1 Adding system packages 60
4.6.2 TexLive for literate programming 60
4.6.3 Python and Julia integration 61
4.6.4 Installing Python packages with uv (impure) 61
4.7 Building and Using Environments 64
4.7.1 Building with nix-build 64
4.7.2 Entering with nix-shell 64
4.7.3 Pure shells for complete isolation 65
4.7.4 Garbage collection 65
4.8 Converting renv Projects 66
4.8.1 Recommended workflow 66
4.8.2 Caveats 66
4.9 Summary 67
4.9.1 Quick Reference: Starting a New Project . 67

5 Functional Programming 69
5.1 Introduction: From Scripts to Functions 69
5.1.1 The Notebook Problem 70
5.1.2 The Functional Solution 72
5.1.3 FP vs OOP: Transformations vs Actors . 73
5.1.4 Why Does This Matter for Data Science? 74
5.2 Purity and Side Effects 75
5.2.1 Handling “Impure” Operations like Randomness 76
5.2.2 The OOP Caveat in Python 78
5.3 Functions: A Refresher and Beyond 79
5.3.1 The Basics 79
5.3.2 Higher-Order Functions 80

5.3.3 Closures: Functions That Remember . . . 81
5.3.4 Decorators (Python) 83
5.3.5 Tidy Evaluation in R 84
5.3.6 Anonymous Functions and Lambdas . . . 86
5.3.7 Partial Application 87
5.3.8 Immutability: Data That Does Not Change 88
5.4 The Functional Toolkit: Map, Filter, and Reduce 90
5.4.1 1. Mapping: Applying a Function to Each Element 90
5.4.2 2. Filtering: Keeping Elements That Match a Condition 92
5.4.3 3. Reducing: Combining All Elements into a Single Value 92
5.5 The Power of Composition 93
5.5.1 The Composition Challenge in Python . . 94
5.6 Handling Errors Functionally 95
5.6.1 In R with `purrr::safely()` and `purrr::possibly()` 95
5.6.2 In Python with try/except and Result patterns 97
5.7 When NOT to Use Functional Programming . . . 97
5.8 Summary 98

6 Building Reproducible Pipelines with rixpress 101
6.1 Introduction: From Scripts and Notebooks to Pipelines 101
6.1.1 The Evolution of Build Automation 101
6.1.2 The Separation Problem 102
6.1.3 The Imperative Approach: Make + Docker 102
6.1.4 rixpress: Unified Orchestration 105
6.2 What is rixpress? 107
6.3 Getting Started 109
6.3.1 Initialising a project 109
6.3.2 Defining the environment 109

6.3.3 Defining the pipeline . . . 110
6.4 Core Functions . . . 111
6.4.1 Defining derivations . . . 111
6.4.2 Building the pipeline . . . 112
6.4.3 Inspecting outputs . . . 112
6.4.4 Visualising the pipeline . . . 112
6.5 Polyglot Pipelines . . . 113
6.5.1 Method 1: Using language converters . . . 114
6.5.2 Method 2: Using universal data formats . 114
6.6 Caching and Incremental Builds . . . 115
6.7 Build Logs and Debugging . . . 116
6.8 ryxpress: The Python Interface . . . 117
6.9 Running Someone Else's Pipeline . . . 118
6.10 Exporting and Importing Artifacts . . . 118
6.11 Real-World Examples . . . 119
6.11.1 Example 1: Machine Learning with XGBoost . . . 119
6.11.2 Example 2: Reading Many Input Files . . 122
6.11.3 Example 3: Full Python→R→Quarto Workflow . . . 123
6.11.4 More Examples . . . 125
6.12 Summary . . . 126

7 Containerisation With Docker 127
7.1 Introduction . . . 127
7.1.1 Spatial vs Temporal Reproducibility . . . 127
7.1.2 The Best of Both Worlds . . . 128
7.1.3 Interactive Development vs Distribution . 129
7.2 Docker Essentials . . . 129
7.2.1 Installing Docker . . . 129
7.2.2 The Rocker Project and Image Registries . 131
7.2.3 Basic Docker Workflow . . . 133
7.3 Making Our Own Images . . . 135
7.3.1 A Minimal Dockerfile with Nix . . . 135

7.3.2 Splitting Into a Reusable Base Image . . . 137
7.4 Publishing Images on Docker Hub 139
7.4.1 Sharing Without Docker Hub 140
7.5 What If You Don't Use Nix? 140
7.6 Building Docker Images Directly with Nix 141
7.6.1 Why Skip the Dockerfile? 142
7.6.2 A Basic Example 142
7.6.3 Benefits Over Dockerfiles 144
7.6.4 Layered Images for Efficiency 144
7.6.5 When to Use This Approach 145
7.7 Dockerizing a rixpress Pipeline 145
7.7.1 Step 1: The Dockerfile 146
7.7.2 Step 2: The Export Script 146
7.7.3 Step 3: Build and Run 147
7.8 Summary 148
7.9 Further Reading 148

8 Robust Pipelines with Monads 149
8.1 The Problem: Decorated Functions Don't Compose 149
8.2 The Solution: Function Factories and Bind 151
8.3 The chronicler Package 152
8.4 The Maybe Monad: Handling Errors 153
8.5 Monads in Python: talvez 154
8.6 Building Robust Pipelines 155
8.7 Integrating with rixpress 157
8.8 Python Integration with talvez 159
8.9 Summary 160

References 161

Welcome!

A modern, unified, and language-agnostic workflow for data science using Nix.

This book is a complete reimagining of my previous work, "Building Reproducible Analytical Pipelines with R." If you're looking for that book, you can find it here. But if you're ready for the next step, you're in the right place.

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

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

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

The aim of this book is to teach you how to use some of the best practices from software engineering and DevOps to make your

projects robust, reliable and reproducible. It doesn't matter if you work alone, in a small or in a big team. It doesn't matter if your work gets (peer-)reviewed or audited: the techniques presented in this book will make your projects more reliable and save you a lot of frustration!

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

The book can be read for free on https://b-rodrigues.github.io/reproducible-data-science/ and you'll be able buy a DRM-free Epub or PDF on Leanpub[1] once there's more content.

This book is the culmination of my previous works. I started by writing a book focused on R, and then began working on a Python edition. During that process, I had a realization: tackling reproducibility one language at a time was solving the symptoms, not the root cause. The real solution needed to be universal, powerful, and capable of handling any language or tool we might need.

That universal solution is Nix.

[1] https://leanpub.com/

This book moves beyond language-specific tooling. It presents a holistic workflow where R, Python, and Julia are not competitors, but collaborators in a single, cohesive, and perfectly reproducible environment. We will cover:

- The Nix Philosophy: Why Nix is the ultimate tool for solving the "it works on my machine" problem, once and for all.
- Declarative Environments with `{rix}`: How to use a simple R interface to define exact, bit-for-bit reproducible software environments that include specific versions of R, Python, Julia, their packages, and any system-level dependencies.
- Polyglot Pipelines with `{rixpress}` (R) or `ryxpress` (Python): How to orchestrate complex analytical pipelines that seamlessly pass data between different languages, all managed by the Nix build system.
- Unit Testing and Functional Programming: Core principles for writing robust, testable, and maintainable code, no matter the language.
- Distribution and Automation: How to package your entire reproducible pipeline into a Docker container for easy sharing and automate your workflow with GitHub Actions.

While this is not a book for beginners (you should be familiar with at least one data-centric programming language before reading this), I will not assume that you have any knowledge of the tools discussed. But be warned, this book will require you to take the time to read it, and then type on your computer. Type *a lot*.

I hope that you will enjoy reading this book and applying the ideas in your day-to-day, ideas which hopefully should improve the reliability, traceability and reproducibility of your code.

If you find this book useful, don't hesitate to let me know! You can submit issues, suggest improvements, and ask questions on the book's Github repository.

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

[2]https://www.brodrigues.co/about/me/

Preface

Three years ago, I wrote a book with a straightforward premise: by borrowing a few key ideas from software engineering, people who analyse data could save themselves a great deal of frustration. The response to that book was more positive than I could have ever hoped for, and it confirmed a suspicion I had: we, as a community, are hungry for better ways to work.

That book, however, focused exclusively on the R ecosystem. A recurring question I received was, "This is great, but what about Python?" It was a fair question. The world of data science is not a monologue; it's a conversation between languages. So, I began what felt like the logical next step: writing a Python edition.

I mapped out the chapters and identified the equivalent tools, `pipenv` for dependency management, `ploomber` for pipelines, and started writing. But as I went deeper, a nagging feeling grew. I was solving the same problems all over again, just with a different set of tools. This feeling was compounded by the rapid churn within the Python ecosystem itself. How many package managers have been created to solve virtual environment management? As of writing, `uv` is all the rage, and while it may be here to stay, history suggests a new contender is always just around the corner.

This pointed to a larger issue. I am convinced that the future of data science is polyglot. An R user and a Python user, both following my original advice, would end up with reproducible

projects, but their workflows would be fundamentally incompatible. They couldn't easily share an environment or build a single pipeline that leverages the strengths of both languages. While companies like Posit have made excellent progress in making it easier to call Python from R, setting up a truly integrated development environment remains a challenge. And what if you wish to bring Julia, the other language of data analysis, into the fold? It is not as popular as Python or R, but it has its own distinct appeal and advantages.

I realised I was treating the symptoms, not the disease. The root problem wasn't "How do I make R reproducible?" or "How do I make Python reproducible?". The real challenge was the lack of a universal foundation that could handle *any* language, *any* tool, and *any* system dependency with absolute, bit-for-bit precision.

That's when I stopped writing the Python book.

The solution wasn't to create another language-specific guide but to find a tool that operated at a more fundamental level. That tool, I am now convinced, is **Nix**.

Nix is not just another package manager; it is a powerful, declarative system for building and managing software environments. It allows us to define the *entire* computational environment—from the operating system libraries up to the specific versions of our R and Python packages—in a single, simple text file. When you use Nix, the phrase "it works on my machine" becomes obsolete. It is replaced by the guarantee: "it builds identically, everywhere, every time"—with a few caveats that we will explore, of course.

This book is the result of that realisation. It is a complete reimagining of the original. We are moving away from language-specific patchworks and toward a unified, polyglot workflow. We

will use Nix as our bedrock, with the `{rix}` and `{rixpress}` R packages (or `ryxpress` for Python) serving as our friendly interface to its power. You will learn to build pipelines where R, Python, and Julia aren't just neighbours; they are collaborators, working together in a single, perfectly reproducible environment.

A note for Python-first users: do not be deterred by the fact that `{rix}` and `{rixpress}` are R packages: there is a Python version called `ryxpress` that will allow you to run your pipelines from an interactive Python session. You will be able to use them to define your environments (even integrating tools like `uv`) and orchestrate your pipelines, while doing all of your analytical work exclusively in Python. In this workflow, R simply becomes a convenient configuration language.

The core message from three years ago remains unchanged. You, as someone who writes code to analyse data, are a developer. Your work is important, and it deserves to be reliable. This book aims to give you the tools and the mindset to achieve that. The journey is more ambitious this time, but the payoff is far greater.

I hope you'll join me.

You can read this book for free online at https://b-rodrigues.github.io/reproducible-data-science/.

You can submit issues, suggest improvements, and ask questions on the book's Github repository.

1 Introduction

Just like the previous, R-focused edition of this book, this one will not teach you about machine learning, statistics, or visualisation.

The goal is to teach you a set of tools, practices, and project management techniques that should make your projects easier to reproduce, replicate, and retrace, with a focus on polyglot (multilingual) projects. I believe that with LLMs, polyglot projects will become increasingly common.

Before LLMs, if you were a Python user, you would avoid R like the plague, and vice versa. This is understandable: even though both are high-level scripting languages, and an R user could likely read and understand Python code (and vice versa), it is still a pain in the loins to set up another language just to run a few lines of code. Now, with LLMs, you can have a model generate the code, and depending on what you're doing, it's likely to be correct. However, setting up another language is still quite annoying. This is where Nix comes in. Nix makes adding another language to a project extremely simple.

1.1 Who is this book for?

This book is for anyone who uses raw data to build any type of output. This could be a simple quarterly report, in which

data is used for tables and graphs, a scientific article for a peer-reviewed journal, or even an interactive web application. The specific output doesn't matter, because the process is, at its core, always very similar:

- Get the data;
- Clean the data;
- Write code to analyse the data;
- Put the results into the final product.

This book assumes some familiarity with programming, particularly with the R and Python languages. I will not discuss Julia in great detail, as I am not familiar enough with it to do it justice. That being said, I will show you how to add Julia to a project and use it effectively if you need to.

1.2 What is the aim of this book?

The aim of this book is to make the process of analysing data as reliable, retraceable, and reproducible as possible, and to do this by design. This means that once you're done with the analysis, you're done. You don't want to spend time, which you often don't have anyway, to rewrite or refactor an analysis to make it reproducible after the fact. We both know this is not going to happen. Once an analysis is finished, it's on to the next one. If you need to rerun an older analysis (for example, because the data has been updated), you'll simply figure it out at that point, right? That's a problem for Future You. Hopefully, Future You will remember every quirk of your code, know which script to run at which point, which comments are outdated, and what features of the data need to be checked… You had better hope Future You is a more diligent worker than you are!

Going forward, I'm going to refer to a project that is reproducible as a "Reproducible Analytical Pipeline", or RAP for short. There are only two ways to create a RAP: either you are lucky enough to have someone on your team whose job is to turn your messy code into a RAP, or you do it yourself. The second option is by far the most common. The issue, as stated above, is that most of us simply don't do it. We are always in a rush to get to the results and don't think about making the process reproducible, because we assume it takes extra time that is better spent on the analysis itself. This is a misconception, for two reasons.

The first is that employing the techniques we will discuss in this book won't actually take much time. As you will see, they are not things you "add on top of" the analysis, but are part of the analysis itself, and they will also help with managing the project. Some of these techniques, especially testing, will even save you time and headaches.

The second reason is that an analysis is never a one-shot. Only the simplest tasks, like pulling a single number from a database, might be. Even then, chances are that once you provide that number, you'll be asked for a variation of it (for example, disaggregated by one or several variables). Or perhaps you'll be asked for an update in six months. You will quickly learn to keep that SQL query in a script somewhere to ensure consistency. But what about more complex analyses? Is keeping the script enough? It is a good start, of course, but very often, there is no single script, or a script for each step of the analysis is missing.

I've seen this play out many times in many different organisations. It's that time of the year again, and a report needs to be written. Ten people are involved, and just gathering the data

is already complicated. Some get their data from Word documents attached to emails, some from a website, some from a PDF report from another department. I remember a story a senior manager at my previous job used to tell: once, a client put out a call for a project that involved setting up a PDF scraper. They periodically needed data from another department that only came in PDFs. The manager asked what was, at least from our perspective, an obvious question: "Why can't they send you the underlying data in a machine-readable format?" They had never thought to ask. So, my manager went to that department and talked to the people putting the PDF together. Their answer? "Well, we could send them the data in any format they want, but they've asked for the tables in a PDF."

So the first, and probably most important, lesson here is: when starting to build a RAP, make sure you talk with all the people involved.

1.3 Prerequisites

You should be comfortable with the command line. This book will not assume any particular Integrated Development Environment (IDE), so most of what I'll show you will be done via the command line. That said, I will spend some time helping you set up a data science-focused IDE, Positron, to work seamlessly with this workflow. The command line may be over 50 years old, but it is not going anywhere. In fact, thanks to the rise of LLMs, it seems to be enjoying a resurgence. Since these models generate text, it is far simpler to ask one for a shell command to solve a problem than to have it produce detailed instructions on where to click in a graphical user interface (GUI). Knowing your way around the command line is also essential for working with

modern data science infrastructure: continuous integration platforms, Docker, remote servers… they all live in the terminal. So, if you're not at all familiar with the command line, you might need to brush up on the basics. Don't worry, though; this isn't a book about the intricacies of the Unix command line. The commands I'll show you will be straightforward and directly applicable to the task at hand.

This means however that if you are using Windows, first of all, why? and second of all, you will have to set up Windows Subsystem for Linux. This is because there is no native Nix implementation for Windows, and so we need to run the Linux version through WSL. Don't worry though, it's not that hard, and you can then use an IDE from Windows to work with the environments managed by Nix, in a very seamless way (but seriously, consider, if you can, switching to Linux. How can you tolerate ads (ADS!) in the start menu).

Ideally, you should be comfortable with either R or Python. This book will assume that you have been using at least one of these languages for some projects and want to improve how you manage complex projects. You should know about packages and how to install them, have written some functions, understand loops, and have a basic knowledge of data structures like lists. While this is not a book on visualisation, we will be making some graphs as well.

Our aim is to write code that can be executed non-interactively. This is because a necessary condition for a workflow to be reproducible and to qualify as a RAP is for it to be executed by a machine, automatically, without any human intervention. This is the second lesson of building RAPs: there should be no human intervention needed to get the outputs once the RAP has started. If you achieve this, then your workflow is likely reproducible, or can at least be made so much more easily than if

it requires special manipulation by a human somewhere in the loop.

1.4 What actually is reproducibility?

A reproducible project means that it can be rerun by anyone at zero (or very minimal) cost. But there are different levels of reproducibility, which I will discuss in the next section. Let's first outline the requirements that a project must meet to be considered a RAP.

1.4.1 Using open-source tools is a hard requirement

Open source is a hard requirement for reproducibility. No ifs, ands, or buts. I'm not just talking about the code you wrote for your research paper or report; I'm talking about the entire ecosystem you used to write your code and build the workflow.

Is your code open? Good. Or is it at least available to others in your organisation, in a way that they could re-execute it if needed? Also good.

But is it code written in a proprietary program, like STATA, SAS, or MATLAB? Then your project is not reproducible. It doesn't matter if the code is well-documented, well-written, and available on a version control system. The project is simply not reproducible. Why?

Because on a long enough time horizon, there is no way to re-execute your code with the exact same version of the proprietary language and operating system that were used when the project

was developed. As I'm writing these lines, MATLAB, for example, is at version R2025a. Buying an older version may not be simple. I'm sure if you contact their sales department, they might be able to sell you an older version. Maybe you can even re-download older versions you've already purchased from their website. But maybe it's not that straightforward. Or maybe they won't offer this option in the future. In any case, if you search for "purchase old version of Matlab," you will see that many researchers and engineers have this need. ::: {.content-hidden when-format="pdf"}

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

:::

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

And if you're running old code written for version, say, R2008a, there's no guarantee that it will produce the exact same results on version R2025a. That's without even mentioning the toolboxes (if you're not familiar with them, they're MATLAB's equivalent of packages or libraries). These evolve as well, and there's no guarantee that you can purchase older versions. It's also likely that newer toolboxes cannot even run on older versions of MATLAB.

Let me be clear: what I'm describing here for MATLAB could also be said for any other proprietary programs still commonly (and unfortunately) used in research and statistics, like STATA,

SAS, or SPSS. Even if some of these vendors provide ways to run older versions of their software, the fact that you have to rely on them for this is a barrier to reproducibility. There is no guarantee they will provide this option forever. Who can guarantee that these companies will even be around forever? More likely, they might shift from a program you install on your machine to a subscription-based model.

For just 199€ a month, you can execute your SAS (or whatever) scripts on the cloud! Worried about data confidentiality? No problem, data is encrypted and stored safely on our secure servers! Run your analysis from anywhere and don't worry about your cat knocking coffee over your laptop! And if you purchase the pro licence, for an additional 100€ a month, you can even execute your code in parallel!

Think this is science fiction? There is a growing and concerning trend for vendors to move to a Software-as-a-Service model with monthly subscriptions. It happened to Adobe's design software, and the primary reason it hasn't yet happened for data analytics tools is data privacy concerns. Once those are deemed "solved," I would not be surprised to see a similar shift.

1.4.2 Hidden dependencies can hinder reproducibility

Then there's another problem. Let's suppose you've written a thoroughly tested and documented workflow and made it available on GitHub (and let's even assume the data is freely available and the paper is open access). Or, if you're in the private sector, you've done all of the above, but the workflow is only available internally.

Let's further assume that you've used R, Python, or another open-source language. Can this analysis be said to be reproducible? Well, if it ran on a proprietary operating system, then the conclusion is: your project is not fully reproducible.

This is because the operating system the code runs on can also influence the outputs. There are particularities in operating systems that may cause certain things to work differently. Admittedly, this is rarely a problem in practice, but it does happen[1], especially if you're working with high-precision floating-point arithmetic, as you might in the financial sector, for instance.

Thankfully, as you will see in this book, there is no need to change your operating system to deal with this issue.

1.4.3 The requirements of a RAP

So where does that leave us? For a project to be truly reproducible, it has to respect the following points:

- Source code must be available and thoroughly tested and documented (which is why we will be using Git and GitHub).
- All dependencies must be easy to find and install (we will deal with this using dependency management tools).
- It must be written in an open-source programming language (no-code tools like Excel are non-reproducible by default because they can't be used non-interactively, which is why we will be using languages like R, Python, and Julia).

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

- The project needs to run on an open-source operating system (we can deal with this without having to install and learn a new OS, thanks to tools like Docker).
- The data and the final report must be accessible—if not publicly, then at least within your company. This means the concept of "scripts and/or data available upon request" belongs in the bin.

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

1.4.4 Are there different types of reproducibility?

Let's take one step back. We live in the real world, where constraints outside of our control can make it impossible to build a true RAP. Sometimes we need to settle for something that might not be perfect, but is the next best thing.

In what follows, let's assume the code is tested and documented, so we will only discuss the pipeline's execution.

The *least* reproducible pipeline would be something that works, but only on your machine. This could be due to hardcoded paths

that only exist on your laptop. Anyone wanting to rerun the pipeline would need to change them. This should be documented in a README, which we've assumed is the case. But perhaps the pipeline only runs on your laptop because the computational environment is hard to reproduce. Maybe you use software, even open-source software, that is not easy to install (anyone who has tried to install R packages on Linux that depend on `{rJava}` knows what I'm talking about).

A better, though still imperfect, pipeline would be one that could be run more easily on any similar machine. This could be achieved by avoiding hardcoded absolute paths and by providing instructions to set up the environment. For example, in Python, this could be as simple as providing a `requirements.txt` file that lists the project's dependencies, which could be installed using `pip`:

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

Doing this helps others (or Future You) install the required packages. However, this is not enough, as other software on your system, outside of what `pip` manages, can still impact the results.

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

The take-away message is that relying on the language itself being stable over time is not a sufficient condition for reproducibility. We have to set up our code in a way that is *explicitly* reproducible by dealing with the versions of the language itself.

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

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

By "reproducibility is on a continuum," I mean that you can set up your project to take none, one, two, three, four, or all of the preceding items into consideration.

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

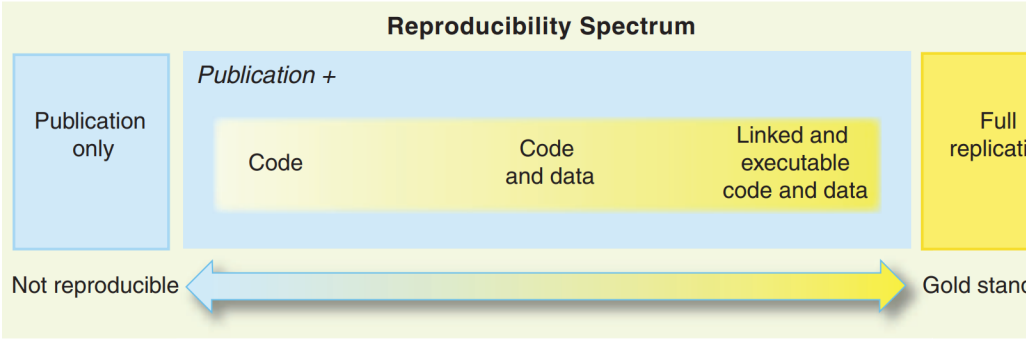

Figure 1.3: The reproducibility spectrum from Peng's 2011 paper.

Let me finish this introduction by discussing the last item on the list: hardware architecture. In 2020, Apple changed the hardware architecture of their computers. Their new machines no longer use Intel CPUs, but instead Apple's own proprietary architecture (Apple Silicon) based on the ARM specification. Concretely, this means that binary packages built for Intel-based Apple computers cannot run on their new machines, at least not without a compatibility layer. If you have a recent Apple Silicon Mac and need to install old packages to rerun a project (and we will learn how to do this), they need to be compiled to work on Apple Silicon first. While a compatibility layer called Rosetta 2 exists, my point is that you never know what might come in the future. The ability to compile from source is important because it requires the fewest dependencies outside of your control. Relying on pre-compiled binaries is not future-proof, which is another reason why open-source tools are a hard requirement for reproducibility.

For you Windows users, don't think that the preceding paragraph does not concern you. It is very likely that Microsoft will

push for OEM manufacturers to build more ARM-based computers in the future. There is already an ARM version of Windows, and I believe Microsoft will continue to support it. This is because ARM is much more energy-efficient than other architectures, and any manufacturer can build its own ARM CPUs by purchasing a license—a very interesting proposition from a business perspective.

It is also possible that we will move towards more cloud-based computing, though I think this is less likely than the hardware shift. In that case, it is quite likely that the actual code will be running on Linux servers that are ARM-based, due to energy and licensing costs. Here again, if you want to run your historical code, you'll have to compile old packages and programming language versions from source.

Ok, this might all seem incredibly complicated. How on earth are we supposed to manage all these risks and balance the immediate need for results with the future need to rerun an old project? And what if rerunning it is never needed?

As you shall see, this is not as difficult as it sounds, thanks to Nix and the tireless efforts of the `nixpkgs` maintainers who work to build truly reproducible packages.

Let's dive in!

2 The Nix Package Manager

2.1 Introduction

Nix is a package manager that can be installed on your computer, regardless of the operating system. If you are familiar with the Ubuntu Linux distribution, you have likely used `apt-get` to install software. On macOS, you may have used `homebrew` for similar purposes. Nix functions in a comparable way but has many advantages over classic package managers, as it focuses on reproducible builds and downloads packages from `nixpkgs`, currently the largest software repository[1].

In this chapter, we will explore the critical need for environment reproducibility in modern workflows. We will see why ad-hoc tools often fail, and how Nix's declarative approach and "component closures" provide a robust solution. We will also cover the core concepts of Nix—derivations, the store, and hermetic builds—that make this possible.

[1] https://repology.org/repositories/graphs

2.2 Why Reproducibility? Why Nix?

2.2.1 Motivation: Reproducibility in Scientific and Data Workflows

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

- Ensure the required version of your programming language (R, Python, etc.) is installed.
- Ensure the required versions of all packages are installed.
- Ensure all necessary system dependencies are installed (for example, a working Java installation for the `{rJava}` R package on Linux).
- Ensure you can install all of this on 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 fulfil all the requirements to build reproducible projects.

The current consensus for tackling the first three points is often a mixture of tools: Docker for system dependencies, `{renv}` or `uv` for package management, and tools like the R installation manager (`rig`) for language versions. As for the last point, hardware architecture, the only way out is to be able to compile the software for the target platform. This involves a lot of moving parts and requires significant knowledge to get right.

2.2.2 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 underly-

ing OS, do not capture system-level 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, and just adding it to your project doesn't make it reproducible if you don't explicitly take some precautionary steps.

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

2.2.3 Nix: A Declarative Solution

With Nix, we can handle all of these challenges with a single tool.

The first advantage of Nix is that its repository, `nixpkgs`, is humongous. As of this writing, it contains over 120,000 pieces of software, including the *entirety of CRAN and Bioconductor*. This means you can use Nix to handle everything: R, Python, Julia, their respective packages, and any other software available through `nixpkgs`, making it particularly useful for polyglot pipelines.

The second and most crucial advantage is that Nix allows you to install software in (relatively) isolated environments. When you start a new project, you can use Nix to install a project-specific version of R and all its packages. These dependencies are used only for that project. If you switch to another project, you switch to a different, independent environment. But this also means that all the dependencies of R and R packages, plus all of their dependencies and so on get installed as well. Your project's development environment will not depend on anything

outside of it (well, there are some caveats which we will explore as we move on).

This is similar to `{renv}`, but the difference is profound: you get not only a project-specific library of R packages but also a project-specific R version and all the necessary system dependencies. For example, if you need `{xlsx}`, Nix automatically figures out that Java is required and installs and configures it for you, without any intervention.

What's more, you can *pin* your project to a specific revision of the `nixpkgs` repository. This ensures that every package Nix installs will always be at the exact same version, regardless of when or where the project is built. The environment is defined in a simple plain-text file, and anyone using that file will get a byte-for-byte identical environment, even on a different operating system.

2.3 Important Concepts

Before we start using Nix, it is important to spend some time learning about some Nix-centric concepts, starting with the *derivation*.

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 (and "inputs" here is meant in a very broad sense; for example, configuration flags are also inputs!).
- 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 explain these three points in more detail.

2.3.1 Derivations

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

```
let
  pkgs = import (fetchTarball
   ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/2025-
   ↪  {};
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
  '';
}
```

Without going into too much detail, this code uses `awk`, a common Unix data processing tool, to filter the `mtcars.csv` file. As you can see, a significant amount of boilerplate is required for this simple operation. However, this approach is completely reproducible: the dependencies are declared and pinned to a specific version of the `nixpkgs` repository. The only thing that could make this small pipeline fail is if the `mtcars.csv` file is not provided to it.

`Nix` builds the `filtered.csv` output file in two steps: it first generates a *derivation* from this expression, and only then does it build the output. For clarity, I will refer to code like the example above as a *derivation* rather than an expression, to avoid confusion with the concept of an *expression* in R.

The goal of the tools we will use in this book, `{rix}` and `{rixpress}` (or `ryxpress` if you prefer using Python), is to help you create pipelines from such derivations without needing to learn the Nix language itself, while still benefiting from its powerful reproducibility features.

2.3.2 Dependencies of derivations

Nix requires that the dependencies of any derivation be explicitly listed and managed by Nix itself. If you are building an output that requires Quarto, then Quarto must be explicitly listed as an input, even if you already have it installed on your system. The same applies to Quarto's dependencies, and their dependencies, all the way down. To run a linear regression with R, you essentially need Nix to build the entire universe of software that R depends on first.

In Nix terms, this complete set of packages is what its author, Eelco Dolstra, refers to as a *component closure*:

> The idea is to always deploy component closures: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. That is, we must always deploy a set of components that is closed under the 'depends on' relation.

(*Nix: A Safe and Policy-Free System for Software Deployment*, Dolstra et al., 2004).

Figure 4 of Dolstra et al. (2004)

In the figure, `subversion` depends on `openssl`, which itself depends on `glibc`. Similarly, if you write a derivation to filter `mtcars`, it requires an input file, R, `{dplyr}`, and all of their respective dependencies. All of these must be managed by Nix. If any dependency exists "outside" this closure, the pipeline will only *work on your machine*, defeating the purpose of reproducibility.

2.3.3 The Nix store and hermetic builds

When building derivations, their outputs are saved into the **Nix store**. Typically located at `/nix/store/`, this folder contains all the software and build artefacts produced by Nix.

For example, the output of a derivation might be stored at a path like `/nix/store/81k4s9q652jlka0c36khpscnmr8wk7jb-filtered`. The long cryptographic hash uniquely identifies the build output and is computed based on the content of the derivation and all its inputs. This ensures that the build is fully reproducible.

As a result, building the same derivation on two different machines will yield the same cryptographic hash. You can substitute the built artefact with the derivation that generates it

one-to-one, just as in mathematics, where writing $f(2)$ is the same as writing 4 for the function $f(x) := x^2$.

To guarantee that derivations always produce identical outputs, builds must occur in an isolated environment known as a **hermetic sandbox**. This process ensures that the build is unaffected by external factors, such as the state of the host system. This isolation extends to environment variables and even network access. If you need to download data from an API, for example, it will not work from within the build sandbox. This may seem restrictive, but it makes perfect sense for reproducibility: an API's output can change over time.

For a truly reproducible result, you should obtain the data once, version it, and use that archived data as an input to your analysis.

2.3.4 Other key Nix concepts

We've covered derivations, dependencies, closures, the Nix store, and hermetic builds. That's the core of what makes Nix tick. But there are a few more concepts worth knowing about before we move on:

- **Purity**: Nix tries very hard to keep builds "pure": the output only depends on what you explicitly list as inputs. If a build script tries to reach out to the internet or read some random file on your machine, Nix will block it. That can feel restrictive at first, but it's what guarantees reproducibility.
- **Binary caches**: You don't always need to build everything yourself. Think back to the math analogy: if you already know that $f(2) = 4$, there's no need to compute

it again; just reuse the result! Nix does the same with binary caches: because every build is identified by its unique cryptographic hash made from its inputs, this means a prebuilt package fetched from `cache.nixos.org` (or your own cache you may want to set up) is bit-for-bit identical to what you would have built locally. This is why Nix is both reproducible and fast.

- **Garbage collection**: Since Nix never overwrites anything in the store, old packages can pile up. Running `nix-store --gc` will run the garbage collector to free up space.
- **Overlays**: If you want to tweak a package or add your own without forking all of `nixpkgs`, you can use overlays. They let you extend or override existing definitions in a clean, composable way.
- **Flakes**: The newer way to define and pin Nix projects. Flakes make it easier to share and reuse Nix setups across machines and repositories. There is a lot of discussion around flakes, as they're officially still considered not stable, even though they've been widely adopted by the community. But don't worry, this is not something you'll need to think about for this book.

Together, these features explain why Nix isn't *just another package manager*. It's more like a framework for reproducible environments that can scale from a single project to an entire operating system (called NixOS[2]).

As a little sidenote: I want also to highlight that Nix can even be used to declaratively and reproducibly configure your operating system (be it NixOS, macOS or other Linux distributions)

[2] https://nixos.org/

using a tool that integrates with it called `homemanager`.[3] This is outside the scope of this book, but I wanted to highlight it, as it's extremely powerful. What this means in practice is that you could write a whole Nix expression that not only downloads and configures software, but even sets up users with their specific software, and preferences like wallpapers, colour schemes and so on.

2.4 Caveats

While Nix is powerful, there are some limitations and practical hurdles to be aware of if you plan to use it for actual work:

- **Hardware acceleration**: On non-NixOS systems, it can be difficult to set up GPU acceleration (CUDA, ROCm, OpenCL). Drivers are tightly coupled to the host kernel and libraries, while Nix builds aim for strict isolation. On NixOS this integration is smoother, but on macOS or other Linux distros you may encounter limitations or extra manual steps.
- **macOS-specific issues**: Reproducibility is harder to achieve on macOS (both Intel and Apple Silicon) than on Linux. Nix packages often rely on Apple system frameworks (e.g., CoreFoundation, Security) that live outside the Nix store and compromise hermetic builds[4]. The macOS sandbox is also weaker than Linux's and sometimes leaks system tools like Xcode or Rosetta into builds[5]. Hydra cache coverage is thinner for Darwin

[3] https://github.com/nix-community/home-manager
[4] https://github.com/NixOS/nixpkgs/issues/67166
[5] https://discourse.nixos.org/t/nix-macos-sandbox-issues-in-nix-2-4-and-later/17475

platforms, especially Apple Silicon, so cache misses and local builds are much more common[6]. Finally, pinned environments can break after macOS or Xcode updates because of changes in system libraries or compiler flags[7].

That said, in practice most packages build fine on macOS, and the ecosystem continues to improve. When reproducibility problems do appear, the simplest fix is often just to use another nearby `nixpkgs` revision for pinning that is known to work. This makes the situation less fragile than it might first appear. Another solution would be to use Docker to deploy the Nix environment and use that as a dev container. All of this will be discussed in detail in this book.

- **Steep learning curve**: The Nix language and ecosystem (flakes, overlays, derivations) can be conceptually difficult if you come from traditional package managers. Even basic customizations require some ramp-up time. This was my main motivation to write `{rix}`, `{rixpress}` (for R) and `ryxpress` (for Python).
- **Disk space and builds**: Because Nix never mutates software in place, the store can accumulate large amounts of data. Cache misses sometimes force local builds, which can be slow and resource-intensive. However, it is of course possible to empty the Nix store to recover disk space, and it is also possible to set up your own project cache if you wish so. Setting up your own cache will also be something that we will explore in this book.

These caveats don't diminish Nix's strengths but highlight that its guarantees are strongest on Linux (and thus WSL), and es-

[6] https://www.reddit.com/r/NixOS/comments/17uxj6q/how_does_nix_package_m
[7] https://github.com/NixOS/nix/issues/11679

pecially on NixOS. On macOS, reproducibility is possible but sometimes requires extra work, a bit of flexibility, and occasionally picking a different `nixpkgs` snapshot.

2.5 In summary

Nix makes it possible to *actually* build software reproducibly. To achieve this, it introduces several core concepts that are quite specific, and definitely worth taking the time to understand.

In the next chapter, we will learn how to install Nix, configure `cachix`, and set up Positron for a seamless development experience.

3 Setting Up Your Environment

In this chapter, we will install Nix, configure the `rstats-on-nix` binary cache to speed up installations, and set up our development environment (Positron, VS Code, or Emacs) to work seamlessly with reproducible Nix environments.

3.1 Installing Nix

3.1.1 For Windows Users: WSL2 Prerequisites

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

```
wsl --install
```

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

I recommend activating `systemd` in Ubuntu WSL2, mainly because this supports users other than `root` running Nix. To set this up, please follow this official Ubuntu blog entry:

```
# in WSL2 Ubuntu shell

sudo -i
nano /etc/wsl.conf
```

This will open `/etc/wsl.conf` in 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.

3.1.2 The Determinate Systems installer

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, run this inside WSL, and after the prerequisites listed above):

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

Just follow the instructions on screen, and in no time Nix will be available on your machine!

3.2 Configuring the Cache

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

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

then use the cache:

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

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

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

```
warning: ignoring the client-specified setting
'trusted-public-keys', because it is a restricted
setting and you are not a trusted user
warning: ignoring untrusted substituter
'https://rstats-on-nix.cachix.org', you are not a
trusted user.
Run `man nix.conf` for more information on the
`substituters` configuration option.
warning: ignoring the client-specified setting
'trusted-public-keys', because it is a restricted
setting and you are not a trusted user
```

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

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

then simply add this line in the file:

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

where `YOURUSERNAME` is your current login user name.

3.3 Verifying installation with Temporary Shells

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

Open a terminal and run:

```
which sl
```

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!

Temporary shells are quite useful, especially if you want to simply run a command using some tool that you only need infrequently. But this is not how we are going to be using Nix.

3.4 Configuring your IDE

3.4.1 Pre-requisites

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

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

On Windows, you need to install Positron on Windows, not inside WSL.

If you want to use RStudio, you can, but you will need to install it through Nix: an RStudio installed through the usual means for your system is not going to be able to interact with Nix! This is a limitation of RStudio and there is currently no workaround. If you want to use RStudio, just skip the rest of the chapter, as it's irrelevant to you. Later, when we use `{rix}` to set up our first Nix development environment, I'll tell you what to do to use RStudio.

Other editors work well with Nix too. Emacs users can use the envrc or emacs-direnv packages to automatically load Nix environments. Neovim users can use direnv.nvim. The key is that any editor with `direnv` support will work with the setup described below.

3.4.2 direnv

Once Positron is installed, you need to install a piece of software called `direnv`: `direnv` will automatically load Nix shells when you open a project that contains a `default.nix` file in an editor.[1] It works on any operating system and many editors support it, including Positron. If you're using Windows, install `direnv` in WSL (even though you've just installed Positron for Windows). To install `direnv` run this command:

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

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

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

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

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

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

Then, in Positron, install the direnv extension. Finally, add a file called `.envrc` and simply write the following two lines in it

[1] A `default.nix` file is a file that contains the specification of our development environments, and is the file that will be automatically generated by `{rix}`

(this `.envrc` file should be in the same folder as your project's `default.nix`):

```
use nix
mkdir $TMP
```

in it. On Windows, *remotely connect to WSL* first, but on other operating systems, simply open the project's folder using `File > 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 Positron and open the project folder again: now it should work every time.

3.4.3 In summary and next steps

For a new project, simply repeat this process:

- Generate the project's `default.nix` file (we will see how in the next chapter);
- Build it using `nix-build`;
- Create an `.envrc` and write the two lines from above in it;
- Open the project's folder in Positron and click allow when prompted;

- Restart the extension and Positron if necessary.

Another option is to create the `.envrc` file and write the two required lines, then open a terminal, navigate to the project's folder, and run `direnv allow`. Doing this before opening Positron should not prompt you anymore.

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

4 Reproducible Development Environments with rix

4.1 Introduction

Now that we have Nix installed, and our IDE configured we can actually tackle the reproducibility puzzle.

4.1.1 The Reproducibility Challenge

Reproducibility in research and data science exists on a continuum. At one end, authors might only describe their methods in prose. Moving along the spectrum, they might share code, then data, and finally what we call a *computational environment*: the complete set of software required to execute an analysis.

Even when researchers share code and data, they rarely specify the full software stack: the exact version of R, all package versions, and crucially, the system-level dependencies. Yet differences in any of these can lead to divergent results from the same code.

Tools like `{renv}` address part of the puzzle: they capture R package versions in a lockfile. But `{renv}` does not manage the

R version itself (you need `rig` for that), and neither handles system libraries. If `{sf}` requires GDAL 3.0 but your system has 2.4, `{renv}` can't help. And if your project uses both R and Python? Now you're coordinating multiple package managers, each with its own configuration.

4.1.2 Component Closures: The Nix Approach

This is where Nix shines. Nix deploys *component closures*: when you install a package, Nix also installs all its dependencies, their dependencies, and so on. Think of it like packing for a trip—traditional package managers assume you'll find essentials at your destination, while Nix packs everything you need.

As the original Nix paper explains:

> The idea is to always deploy component closures: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. Since closures are self-contained, they are the units of complete software deployment.

This means when you install `{sf}` through Nix, you automatically get the correct versions of GDAL, GEOS, and PROJ—no manual system configuration needed.

4.1.3 The Polyglot Challenge

Modern data science is increasingly polyglot. Research shows that data scientists use, on average, nearly two programming languages in their work, with R and Python being the most common combination. Python dominates machine learning, R

excels at statistical modelling, and Julia offers high-performance numerics. Projects increasingly combine these strengths.

This creates a reproducibility challenge: a project using R, Python, and Quarto requires coordinating multiple package managers. Nix solves this by providing a unified framework for all languages and system tools.

4.1.4 Enter rix

However, Nix has a steep learning curve. Its functional programming language can be daunting for researchers focused on their analysis, not system administration.

`{rix}` bridges this gap. It's an R package that generates Nix expressions from intuitive R function calls. You describe *what* you want, and `{rix}` figures out *how* to express it in Nix. The workflow is simple:

1. **Declare** your environment using `rix()`
2. **Build** the environment using `nix-build`
3. **Use** the environment using `nix-shell`

This chapter covers everything you need to know to create project-specific, reproducible development environments for your R projects.

4.2 Transitioning to Nix-Managed R

Now that Nix is installed, I strongly recommend uninstalling any system-wide R installation and removing the packages in your user library (typically found in `~/R` on Linux or `~/Library/R` on macOS). From this point forward, let Nix handle everything. If

you are using Windows, you can keep your Windows-specific R installation, since Nix will not interfere with it (remember, Nix is installed inside WSL and Positron will automatically load Nix environments installed in WSL as well).

If you are not ready to take this step, you can continue using your local R library. The `{rix}` package makes efforts not to interfere with your existing setup, and the `.Rprofile` it generates helps keep Nix environments isolated. However, for the cleanest experience and to avoid subtle conflicts, I recommend fully committing to Nix.

4.2.1 Bootstrapping `{rix}` without a local R installation

But if R is uninstalled, how do you use `{rix}` to generate environments? The answer lies in the temporary shells we saw in the previous chapter. Use Nix itself to bootstrap a temporary R session with `{rix}` available.

Running the following line in a terminal will drop you into an interactive R session:

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

This gives you a temporary shell with R and `{rix}` ready to use. From here, you can generate project-specific Nix expressions.

For example, navigate to an empty directory for a new project:

```
mkdir my-project
cd my-project
```

Then start R:

```
R
```

Load `{rix}` and generate an expression:

```
library(rix)

rix(
  date = "2025-04-11",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

This writes a `default.nix` file to your project directory. The expression defines a shell with R, `{dplyr}`, and `{ggplot2}` as they were on the 11th of April 2025 on CRAN. The `ide` argument is set to `"none"` because we configured Positron in the previous chapter rather than having Nix manage an IDE.

I recommend saving this code in a file called `gen-env.R` in your project directory. If you later need to add packages or change the R version, simply edit `gen-env.R`, run it to regenerate `default.nix`, and rebuild with `nix-build`. This keeps your environment definition readable and versioned alongside your project.

You can now exit this temporary session (type `q()` in R, then `exit` in the shell) and build your new environment:

```
nix-build
```

Once built, use `nix-shell` to enter your project's environment. This workflow of bootstrapping `{rix}` via a temporary shell, generating a `default.nix`, and then building it is the pattern you will follow for every new project.

This is also the reason why there is no Python version of `{rix}`: you can bootstrap a Python environment using `{rix}` in the same way, by only temporarily having the R interpreter available through the Nix shell.

4.3 The rix() Function

The `rix()` function is the heart of the package. It generates a `default.nix` file—a Nix expression that defines your development environment. Here are its main arguments:

- `r_ver`: the version of R you need (or use `date` instead)
- `date`: a date corresponding to a CRAN snapshot
- `r_pkgs`: R packages to install from CRAN/Bioconductor
- `system_pkgs`: system tools like `quarto` or `git`
- `git_pkgs`: R packages to install from GitHub
- `local_r_pkgs`: local `.tar.gz` packages to install
- `tex_pkgs`: TexLive packages for literate programming
- `ide`: which IDE to configure (`"rstudio"`, `"code"`, `"positron"`, `"none"`). Since we've configured Positron in the previous chapter, you'll want to set this to `"none"`. If you want to use RStudio, then set it to `"rstudio"`. This will install RStudio using Nix and make it available from the development shell. If you set it to `"code"` or

`"positron"`, this will then install VS Code or Positron using Nix.
- `project_path`: where to save the `default.nix` file
- `overwrite`: whether to overwrite an existing `default.nix`

Let's create our first environment. Suppose you need R with `{dplyr}` and `{ggplot2}`:

```
library(rix)

rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "rstudio",
  project_path = ".",
  overwrite = TRUE
)
```

This generates two files:

1. **`default.nix`**: The Nix expression defining your environment
2. **`.Rprofile`**: Created by `rix_init()` (called automatically), this file prevents conflicts with any system-installed R packages

The `.Rprofile` is important: it ensures that packages from your user library don't get loaded into the Nix environment, and it redefines `install.packages()` to throw an error, because you should never install packages that way in a Nix environment.

4.4 Choosing an R Version or Date

The `r_ver` argument (or alternatively `date`) controls which version of R and which package versions you'll get. Here's a summary of the options:

`r_ver` / `date`	Intended Use	R Version	Package Versions
`"latest-upstream"`	New project, versions don't matter	Current/previous	Up to 6 months old
`"4.4.2"` (or similar)	Reproduce old project or start new	Specified version	Up to 2 months old
`date = "2024-12-14"`	Precise CRAN snapshot	Current at that date	Exact versions from date
`"bleeding-edge"`	Develop against latest CRAN	Always current	Always current
`"frozen-edge"`	Latest CRAN, manual updates	Current at generation	Current at generation
`"r-devel"`	Test against R development version	R-devel	Always current

To see which R versions are available:

```
available_r()
```

To see which dates are available for snapshotting:

```
available_dates()
```

4.4.1 Using dates vs versions

Using a specific date is often the best choice for reproducibility. When you specify a date, you get the exact state of CRAN on that day:

```
rix(
  date = "2024-12-14",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

I find that this is often easier and clearer than using an R version. Be careful though, as using a date doesn't reflect the state of PyPi on that date. So if you also need Python packages, the versions of packages for Python that will be provided are the ones available from `nixpkgs` on that day (but more on this later).

4.4.2 The rstats-on-nix fork

When you use a specific R version or date (rather than `"latest-upstream"`), `{rix}` uses our `rstats-on-nix` fork of `nixpkgs` rather than the upstream repository. This fork:

- Snapshots CRAN more frequently
- Offers newer R releases faster than the official channels
- Includes many fixes, especially for Apple Silicon (even though as time goes by, this is becoming much rarer, because Nix is getting better and more support for Apple Silicon)

4.5 Installing R Packages

4.5.1 From CRAN/Bioconductor

The simplest case—just list package names in `r_pkgs`:

```
rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2", "tidyr", "readr"),
  ide = "none",
  project_path = "."
)
```

Both CRAN and Bioconductor packages can be specified this way.

4.5.2 Installing archived versions

Need a specific old version of a package? Use the `@` syntax:

```
rix(
  r_ver = "4.2.1",
  r_pkgs = c("dplyr@0.8.0", "janitor@1.0.0"),
  ide = "none",
  project_path = "."
)
```

This will install `{dplyr}` version 0.8.0 from the CRAN archives. Note that archived packages are built from source, which may fail for packages requiring compilation. Thus I recommend you use a date on which that specific version of `{dplyr}` was current.

4.5.3 Installing from GitHub

For packages on GitHub, use the `git_pkgs` argument with a list containing the package name, repository URL, and commit hash:

```
rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2"),
  git_pkgs = list(
    list(
      package_name = "housing",
      repo_url =
       ↪ "https://github.com/rap4all/housing/",
      commit =
       ↪ "1c860959310b80e67c41f7bbdc3e84cef00df18e"
    )
  ),
```

```
  ide = "none",
  project_path = "."
)
```

Always specify a commit hash, not a branch name. This ensures reproducibility: branch names can change, but commits are immutable.

If the R package lives in a subfolder of the repository, append the subfolder to the URL:

```
git_pkgs = list(
  package_name = "BPCells",
  repo_url = "https://github.com/bnprks/BPCells/r",
   ↪  # Note the /r suffix
  commit =
   ↪  "16faeade0a26b392637217b0caf5d7017c5bdf9b"
)
```

Note that this will install the package from source, so if it's a package that requires specific system dependencies, you will need to specify them manually (Nix takes care of this for you for packages that are available through `nixpkgs`, but not for ad-hoc packages from other sources).

4.5.4 Installing local packages

For local `.tar.gz` archives, place them in the same directory as your `default.nix` and use `local_r_pkgs`:

```
rix(
  r_ver = "4.3.1",
  local_r_pkgs = c("mypackage_1.0.0.tar.gz"),
  ide = "none",
  project_path = "."
)
```

Just like with packages from Git, note that this will install the package from source, so if it's a package that requires specific system dependencies, you will need to specify them manually (Nix takes care of this for you for packages that are available through `nixpkgs`, but not for ad-hoc packages from other sources).

4.5.5 Why NOT to use install.packages()

It's crucial to understand: **never call `install.packages()` from within a Nix environment**. Here's why:

1. **Declarative environments**: If you install packages imperatively, your `default.nix` no longer matches your actual environment.
2. **Leaking packages**: Packages installed via `install.packages()` go to your user library, not the Nix environment. They'll be visible to *all* Nix shells, breaking isolation.
3. **Reproducibility**: The whole point of Nix is that your environment is fully defined by the `default.nix`. Ad-hoc installations defeat this.

Instead, add packages to your `rix()` call and rebuild the environment.

4.6 System Tools and TexLive

4.6.1 Adding system packages

Need command-line tools like Quarto or Git? Add them via `system_pkgs`:

```
rix(
  r_ver = "latest-upstream",
  r_pkgs = c("quarto"),
  system_pkgs = c("quarto", "git"),
  ide = "none",
  project_path = "."
)
```

Note that here we install both the R `{quarto}` package and the `quarto` command-line tool—they're different things!

To find available packages, search at search.nixos.org.

4.6.2 TexLive for literate programming

For PDF output from Quarto, R Markdown, or Sweave, you need TexLive packages:

```
rix(
  r_ver = "latest-upstream",
  r_pkgs = c("quarto"),
  system_pkgs = "quarto",
  tex_pkgs = c("amsmath", "framed", "fvextra"),
  ide = "none",
```

```
  project_path = "."
)
```

This installs the `scheme-small` TexLive distribution plus the specified packages.

4.6.3 Python and Julia integration

`{rix}` can also add Python or Julia to your environment:

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  py_conf = list(
    py_version = "3.12",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "none",
  project_path = "."
)
```

This creates a polyglot environment with R, Python, and the specified packages for each.

4.6.4 Installing Python packages with uv (impure)

Not all Python packages (or versions of packages) are available through Nix. Unlike CRAN, PyPI doesn't get automatically mirrored—individual packages must be packaged by volunteers.

If a Python package you need isn't in nixpkgs, you can use `uv` as an escape hatch.

This approach is also useful when collaborating with colleagues who use `uv` but haven't adopted Nix yet. `uv` is 10–100x faster than `pip` and generates a lock file for improved reproducibility.

The idea is to install `uv` in your shell (but not Python or Python packages through Nix):

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  system_pkgs = c("uv"),
  ide = "none",
  project_path = "."
)
```

Then use `uv` from within your shell. We recommend:

1. Specify Python packages in a `requirements.txt` file with explicit versions (e.g., `scanpy==1.11.4`)
2. Set up a shell hook to automatically configure the virtual environment

Here's a complete example with a shell hook:

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  system_pkgs = c("uv"),
  shell_hook = "
    if [ ! -f pyproject.toml ]; then
      uv init --python 3.13.5
```

```
    fi
    uv add --requirements requirements.txt
    alias python='uv run python'
  ",
  ide = "none",
  project_path = "."
)
```

After running `nix-shell`, `uv` initialises a Python project with the specified version and installs packages from `requirements.txt`. This happens each time you enter the shell, but `uv` caches everything so it's nearly instant after the first run.

4.6.4.1 Troubleshooting wheel issues

When using wheels (pre-compiled Python packages), you may encounter errors like:

```
ImportError: libstdc++.so.6: cannot open shared
object file
```

This happens because wheels expect certain libraries in certain locations. Add this to your shell hook to fix it:

```
shellHook = ''
  export LD_LIBRARY_PATH="${pkgs.lib.makeLibraryPath
 ↪  (with pkgs; [
    zlib gcc.cc glibc stdenv.cc.cc
  ])}":$LD_LIBRARY_PATH
  # ... rest of your hook
'';
```

If this seems complicated: yes, it is. This is exactly the kind of problem Nix aims to solve. When possible, prefer Python packages included in nixpkgs.

4.7 Building and Using Environments

4.7.1 Building with nix-build

Once you have a `default.nix`, build the environment:

```
nix-build
```

This downloads/builds all required packages and creates a `result` symlink in your project directory. The `result` file prevents the environment from being garbage-collected.

4.7.2 Entering with nix-shell

To use the environment interactively:

```
nix-shell
```

You'll drop into a shell where R and all your packages are available. Type `R` to start an R session. If you're using RStudio (and specified `ide = "rstudio"` in the call to `rix()`) then type `rstudio` to launch RStudio. If instead you configured Positron, (or any other IDE), open Positron, and open the project folder that contains the `default.nix` (make sure you also have an `.envrc` there for `direnv` to load the environment directly).

You can even run scripts directly without entering the shell:

```
nix-shell default.nix --run "Rscript analysis.R"
```

This is quite useful on CI/CD platforms.

4.7.3 Pure shells for complete isolation

By default, `nix-shell` can still see programs installed on your system. This is quite important to understand: building the environment happens in an isolated, hermetic sandbox, but when inside a shell, isolation is more porous and it is possible to use other systems tools. For example, you don't need to install `git` inside the Nix development environment: you can just keep using the `git` executable already available on your system.

But it is possible to run the environment with increased isolation:

```
nix-shell --pure
```

This hides everything not explicitly included in your environment.

4.7.4 Garbage collection

Nix never deletes old packages automatically. To clean up:

1. Delete the `result` symlink that will appear in your project's folder after calling `nix-build`
2. Run `nix-store --gc`

This removes all packages that are no longer referenced by any environment.

4.8 Converting renv Projects

If you have existing projects using `{renv}`, the `renv2nix()` function can help you migrate:

```
renv2nix(
  renv_lock_path = "path/to/project/renv.lock",
  project_path = "path/to/new_nix_project"
)
```

4.8.1 Recommended workflow

1. Copy the `renv.lock` file to a new, empty folder
2. Run `renv2nix()` pointing to that folder
3. Build the environment with `nix-build`

Do **not** convert in the same folder as the original `{renv}` project—the generated `.Rprofile` will conflict with `{renv}`'s `.Rprofile`.

4.8.2 Caveats

- Package versions may not match exactly due to how Nix handles snapshotting
- If the `renv.lock` lists an old R version but recent packages, use the `override_r_ver` argument to specify a more appropriate R version

4.9 Summary

Let's step back and recap why we have gone through all this trouble.

Traditional tools like `{renv}` or Python's `venv` only capture part of the reproducibility puzzle. They track package versions but not the language version itself, nor system-level dependencies like GDAL or Java. This means your project can still break on a different machine, or even on your own machine after a system update.

Nix solves this by managing *everything*: R, Python, all packages, and all system dependencies. When you define an environment with `{rix}`, you get a complete, self-contained specification that anyone can use to recreate the exact same environment, on any machine, at any point in the future.

4.9.1 Quick Reference: Starting a New Project

Here is the workflow you will follow for every new project:

1. **Create an empty folder** for your project:

```
mkdir my-project
cd my-project
```

2. **Write a `gen-env.R` file** with your environment definition:

```
library(rix)

rix(
  date = "2025-04-11",
  r_pkgs = c("dplyr", "ggplot2"),
```

```
  ide = "none",  # or "rstudio" if you prefer
  project_path = ".",
  overwrite = TRUE
)
```

3. **Add a `.envrc` file** (only needed for Positron or VS Code, skip if using RStudio):

```
use nix
mkdir $TMP
```

4. **Bootstrap `{rix}` and generate `default.nix`**:

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

   Then in R:

```
source("gen-env.R")
```

   Exit R with `q()` and the shell with `exit`.

5. **Build the environment**:

```
nix-build
# you could also run `direnv allow` to allow
 ↪  direnv to load the environment
# automatically and build the environment on
 ↪  first use
```

6. **Start working**: Open the folder in Positron (which will load the environment via `direnv`), or run `nix-shell` followed by `rstudio` if you set `ide = "rstudio"`.

That's it. Your project is now reproducible. Anyone with Nix installed can clone your repository, run `nix-build`, and get the exact same environment you have.

5 Functional Programming

In this chapter, we will see why functional programming is crucial for reproducible, testable, and collaborative data science. We will compare how to write self-contained, "pure" functions in both R and Python, and how to use functional concepts like `map`, `filter`, and `reduce` to replace error-prone loops. Finally, we will discuss how writing functions makes your code easier to review, debug, and even generate with LLMs.

5.1 Introduction: From Scripts to Functions

In the previous chapter, we learned how to create reproducible development environments with `{rix}`. We can now ensure everyone has the *exact same tools* (R, Python, system libraries) *to run our code.

We could have stopped there: after all, we have now reproducible environments, as many as we need for each of our projects. But having the right tools is only half the battle. Now we turn to **writing reproducible code itself**. A common way to start a data analysis is by writing a script: a sequence of commands executed from top to bottom.

```r
# 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
# Python script example
import pandas as pd
mtcars = pd.read_csv("mtcars.csv")
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. Of course, this is a simple example that is only 4 lines long. But what if you have several scripts, each hundreds of lines long? Making sense of that is not easy, if you use the coding style above especially.

5.1.1 The Notebook Problem

If scripting with state is a crack in the foundation of reproducibility, then using computational notebooks is a gaping hole. I will not make many friends in the Python community with the following paragraphs, but that's because only truth hurts.

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.

In a famous talk from JupyterCon 2018, Joel Grus (a research engineer at the Allen Institute for AI) played the role of the "heel" at a conference dedicated to the very tool he was criticising.[1] His central thesis is that data science code should follow software engineering best practices, and Jupyter notebooks actively discourage those practices.

His biggest complaint is **hidden state and out-of-order execution**. The state of the variables depends on the *execution history*, not the order of the code on the screen. You can delete a cell that defined a variable, but that variable still exists in memory. This leads to unreproducible results where the code works right now (because of hidden state in memory) but will fail if you restart the kernel and run top-to-bottom. Worse, it confuses beginners who do not understand why their code works one minute and breaks the next.

Notebooks also **discourage modular code**. Because it is difficult to import code from one notebook into another, users tend to write massive, monolithic scripts, copy and paste the same code blocks into multiple notebooks, and avoid creating functions or modules that can be tested and reused.

Then there is the **"works on my machine" problem**. Notebooks often lack clear dependency specifications, users hardcode file paths that only exist on their specific computer, and reusing someone else's work usually involves manually copying and pasting cells, which is error-prone.

[1] I don't like notebooks

Grus also argues that notebooks have **poor tooling compared to IDEs**. Actual text editors provide linting (identifying stylistic errors or unused variables), type checking, superior autocomplete, and the ability to run unit tests. All of these are difficult or impossible in notebooks.

Finally, **version control is a nightmare**. Notebooks are JSON files. If two people edit a notebook and try to merge their changes in Git, the diffs are unreadable blocks of JSON metadata, and merge conflicts are incredibly difficult to resolve. This encourages workflows where people email files back and forth rather than using proper version control.

What does Grus suggest instead? Write code in modules (`.py` or `.R` files) using a proper editor, write unit tests to ensure the code works, and use notebooks *only* for the final step: importing those modules to visualise the data or present the results. Ideally, the notebook should contain very little logic and mostly just function calls.

This is exactly the approach we will take in this book.

5.1.2 The Functional Solution

The solution is to embrace a paradigm that minimises state: **Functional Programming (FP)**. Instead of a linear script, we structure our code as a collection of self-contained, predictable functions.

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.

5.1.3 FP vs OOP: Transformations vs Actors

To appreciate what FP brings, it helps to contrast it with **Object-Oriented Programming (OOP)**, arguably the dominant paradigm in many software systems.

OOP organises computation around *who does what*: a network of objects communicating with each other and managing their own internal state. You send a message to an object, asking it to perform an action, without needing to know how it works internally.

Functional programming, by contrast, organises computation around *how data changes*. It replaces a network of interacting objects with a flow of transformations: data goes in, data comes out, and nothing else changes in the process.

This shift is especially powerful in data science:

- Analyses are naturally expressed as **pipelines of transformations** (cleaning, filtering, aggregating, modelling)
- Pure functions make results **reproducible**: same inputs always yield same outputs
- **Immutability** prevents accidental side effects on shared data
- Because transformations can be composed, tested, and reused independently, FP encourages **modular, maintainable** analysis code

There is also a structural reason why FP fits data science better than OOP. OOP excels when you have *many different types of objects*, each with a *small set of methods*. A graphical user interface, for example, has buttons, menus, windows, and dialogs, each responding to a few actions like `click()` or `resize()`. But data science is the opposite: we typically work with *a small number of data structures* (data frames, arrays, models) and apply *a large number of operations* to them (filtering, grouping, joining, summarising, plotting). FP handles this case naturally: adding a new function is trivial, while OOP would require modifying every class.

5.1.4 Why Does This Matter for Data Science?

Adopting a functional style brings massive benefits:

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.
2. **Code Review is Easier:** 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 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."
4. **Readability:** 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.

5.2 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
```

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

```
80.0
```

```
# > 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
```

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

```
80.0
```

```
# > 80.0
```

Now the function is predictable and self-contained.

5.2.1 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:

```r
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)
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839
0.12928774
```

```r
pure_rnorm(n = 5, seed = 123)  # Same result!
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839
0.12928774
```

In Python, `numpy` provides a more modern, object-oriented way:

```python
import numpy as np

# Create a random number generator instance with a
 ↪  seed
```

```
rng = np.random.default_rng(seed=123)
print(rng.standard_normal(5))
```

```
[-0.98912135 -0.36778665  1.28792526  0.19397442
0.9202309 ]
```

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

```
[-0.98912135 -0.36778665  1.28792526  0.19397442
0.9202309 ]
```

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

5.2.2 The OOP Caveat in Python

This introduces a concept from OOP: the `rng` variable is an *object* that bundles together data (its internal seed state) and methods (`.standard_normal()`). This is **encapsulation**.

This is a double-edged sword for reproducibility. The `rng` object is now a stateful entity. If we called `rng.standard_normal(5)` a second time, it would produce different numbers because its internal state was mutated.

Core Python libraries like `pandas`, `scikit-learn`, and `matplotlib` are fundamentally object-oriented. Our guiding principle must be:

> **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 than an OOP one (`pipeline.load(); pipeline.clean(); pipeline.analyze()`).

5.3 Functions: A Refresher and Beyond

You likely already know how to write functions in R and Python. This section serves as a quick refresher, but also introduces some concepts you may not have encountered: **higher-order functions**, **closures**, and **decorators**.

5.3.1 The Basics

In R, functions are first-class citizens. You assign them to variables and pass them around like any other value:

```r
calculate_ci <- function(x, level = 0.95) {
  se <- sd(x, na.rm = TRUE) / sqrt(length(x))
  mean_val <- mean(x, na.rm = TRUE)
  alpha <- 1 - level
  lower <- mean_val - qnorm(1 - alpha/2) * se
  upper <- mean_val + qnorm(1 - alpha/2) * se
  c(mean = mean_val, lower = lower, upper = upper)
}
```

In Python, the `def` keyword defines functions. Type hints are recommended:

```
def calculate_ci(x: list[float], level: float =
 ↪  0.95) -> dict:
    """Calculate confidence interval for a list of
     ↪  numbers."""
    import statistics
    import scipy.stats as stats
    n = len(x)
    mean_val = statistics.mean(x)
    se = statistics.stdev(x) / (n ** 0.5)
    alpha = 1 - level
    z = stats.norm.ppf(1 - alpha / 2)
    return {"mean": mean_val, "lower": mean_val - z
     ↪  * se, "upper": mean_val + z * se}
```

5.3.2 Higher-Order Functions

A **higher-order function** is a function that takes another function as an argument, returns a function, or both. This is the foundation of functional programming.

You have already seen examples: `map()`, `filter()`, and `reduce()` are all higher-order functions because they take a function as their first argument.

Here is a simple example in R:

```
apply_twice <- function(f, x) {
 f(f(x))
}
```

```
apply_twice(sqrt, 16)  # sqrt(sqrt(16)) = sqrt(4) =
 ↪  2
```

```
[1] 2
```

And in Python:

```
def apply_twice(f, x):
    return f(f(x))

apply_twice(lambda x: x ** 2, 2)  # (2^2)^2 = 16
```

```
16
```

5.3.3 Closures: Functions That Remember

A **closure** is a function that "remembers" variables from its enclosing scope, even after that scope has finished executing. This is useful for creating specialised functions. These are sometimes called *function factories*.

In R:

```
make_power <- function(n) {
  function(x) x^n
}

square <- make_power(2)
cube <- make_power(3)

square(4)  # 16
```

```
[1] 16
```

```
cube(4)    # 64
```

```
[1] 64
```

In Python:

```
def make_power(n):
    def power(x):
        return x ** n
    return power

square = make_power(2)
cube = make_power(3)

square(4)  # 16
```

```
16
```

```
cube(4)    # 64
```

```
64
```

The inner function “closes over” the variable `n`, preserving its value.

5.3.4 Decorators (Python)

Python has a special syntax for a common use of higher-order functions: **decorators**. A decorator wraps a function to extend its behaviour without modifying its code.

```
import time

def timer(func):
    """A decorator that prints how long a function
     ↪  takes to run."""
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end -
         ↪  start:.4f} seconds")
        return result
    return wrapper

@timer
def slow_sum(n):
    return sum(range(n))

slow_sum(1_000_000)
```

```
slow_sum took 0.0219 seconds
499999500000
```

```
# > slow_sum took 0.0312 seconds
```

The `@timer` syntax is equivalent to `slow_sum = timer(slow_sum)`. Decorators are widely used in Python frameworks (Flask, FastAPI, pytest) for logging, authentication, and caching.

If you find decorators elegant, you will appreciate the chapter on **monads** later in this book. Monads take the idea of wrapping and chaining functions further, providing a principled way to handle errors, missing values, and side effects in a purely functional style.

R does not have built-in decorator syntax, but you can achieve the same effect with higher-order functions:

```
timer <- function(f) {
  function(...) {
    start <- Sys.time()
    result <- f(...)
    end <- Sys.time()
    message(sprintf("Elapsed: %.4f seconds", end -
     ↪  start))
    result
  }
}

slow_sum <- timer(function(n) sum(seq_len(n)))
slow_sum(1e6)
```

```
Elapsed: 0.0000 seconds
```

```
[1] 500000500000
```

5.3.5 Tidy Evaluation in R

For data analysis, you will often want functions that work with column names. The `{dplyr}` package uses "tidy evaluation" with the `{ }` (curly-curly) syntax:

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from
'package:stats':

    filter, lag

The following objects are masked from
'package:base':

    intersect, setdiff, setequal, union
```

```
summarise_variable <- function(data, var) {
  data %>%
    summarise(
      n = n(),
      mean = mean({{ var }}, na.rm = TRUE),
      sd = sd({{ var }}, na.rm = TRUE)
    )
}

starwars %>%
  group_by(species) %>%
  summarise_variable(height)
```

```
# A tibble: 38 x 4
   species       n  mean    sd
   <chr>     <int> <dbl> <dbl>
 1 Aleena        1    79    NA
 2 Besalisk      1   198    NA
```

```
 3 Cerean          1  198   NA
 4 Chagrian        1  196   NA
 5 Clawdite        1  168   NA
 6 Droid           6  131.  49.1
 7 Dug             1  112   NA
 8 Ewok            1   88   NA
 9 Geonosian       1  183   NA
10 Gungan          3  209.  14.2
# i 28 more rows
```

The `{ var }` tells `dplyr` to treat `var` as a column name rather than a literal variable.

5.3.6 Anonymous Functions and Lambdas

Sometimes you need a quick, throwaway function that does not deserve a name. Both R and Python support **anonymous functions**.

In Python, the `lambda` keyword creates a small function in a single expression:

```python
squares = list(map(lambda x: x ** 2, [1, 2, 3, 4]))
squares
```

```
[1, 4, 9, 16]
```

In R, the base syntax is verbose, but `{purrr}` introduced the **formula shorthand** using `~` and `.x`:

```r
library(purrr)

# Full anonymous function
map_dbl(1:4, function(x) x^2)
```

```
[1]  1  4  9 16
```

```
# Formula shorthand (purrr style)
map_dbl(1:4, ~ .x^2)
```

```
[1]  1  4  9 16
```

R 4.1 also introduced a shorter base syntax with `\(x)`:

```
# Base R shorthand (R >= 4.1)
sapply(1:4, \(x) x^2)
```

```
[1]  1  4  9 16
```

Use anonymous functions when the logic is simple and a full function definition would be overkill.

5.3.7 Partial Application

Partial application means fixing some arguments of a function to create a more specialised version. This is closely related to function factories but works with existing functions rather than defining new ones.

In R, use `purrr::partial()`:

```
library(purrr)

# Create a function that always rounds to 2 decimal
 ↪  places
round2 <- partial(round, digits = 2)

round2(3.14159)  # 3.14
```

```
[1] 3.14
```

```
round2(2.71828)  # 2.72
```

```
[1] 2.72
```

In Python, use `functools.partial()`:

```
from functools import partial

# Create a function that always rounds to 2 decimal
 ↪  places
round2 = partial(round, ndigits=2)

round2(3.14159)  # 3.14
```

```
3.14
```

```
round2(2.71828)  # 2.72
```

```
2.72
```

Partial application is useful for creating callbacks, simplifying repetitive code, and making functions fit the signature expected by `map()` or similar.

5.3.8 Immutability: Data That Does Not Change

A core principle of functional programming is **immutability**: once data is created, it is never modified in place. Instead, transformations produce *new* copies of the data.

R has **copy-on-modify** semantics. When you "modify" a data frame, R actually creates a new copy:

```r
df <- data.frame(x = 1:3)
df2 <- df
df2$x[1] <- 99
df$x[1]  # Still 1, df was not mutated
```

```
[1] 1
```

Python, by contrast, has **mutable defaults**, which can surprise newcomers:

```python
import pandas as pd

df = pd.DataFrame({"x": [1, 2, 3]})
df2 = df  # This is a reference, not a copy!
df2.loc[0, "x"] = 99
df.loc[0, "x"]  # Now 99, df was mutated!
```

```
np.int64(99)
```

```python
# To avoid this, explicitly copy:
df2 = df.copy()
```

Immutability prevents entire classes of bugs where one part of your code unexpectedly modifies data used elsewhere. When using Python, be explicit about copying when you need independent data.

5.4 The Functional Toolkit: Map, Filter, and Reduce

Most `for` loops can be replaced by one of three core functional concepts: **mapping**, **filtering**, or **reducing**. These are "higher-order functions": functions that take other functions as arguments.

5.4.1 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.

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

The `{purrr}` package is the gold standard for functional programming in R:

- `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)

```
library(purrr)

# The classic for-loop way (verbose)
means_loop <- vector("double", ncol(mtcars))
for (i in seq_along(mtcars)) {
  means_loop[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
```

```
}

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

The `map()` version is not just shorter; it's safer. You can't make an off-by-one error.

5.4.1.2 In Python with List Comprehensions

Python's most idiomatic tool for mapping is the **list comprehension**:

```
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:

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

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

5.4.2 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.

5.4.2.1 In R with `purrr::keep()`

```r
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)

# Keep only data frames with more than 100 rows
large_dfs <- keep(list_of_dfs, ~ nrow(.x) > 100)
```

5.4.2.2 In Python with List Comprehensions

List comprehensions have a built-in `if` clause:

```python
numbers = [1, 10, 5, 20, 15, 30]
large_numbers = [n for n in numbers if n > 10]
# > [20, 15, 30]
```

5.4.3 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.

5.4.3.1 In R with `purrr::reduce()`

```r
# Sum all elements
total_sum <- reduce(c(1, 2, 3, 4, 5), `+`)

# Find common columns across multiple data frames
list_of_colnames <- map(list_of_dfs, names)
common_cols <- reduce(list_of_colnames, intersect)
```

5.4.3.2 In Python with `functools.reduce`

```python
from functools import reduce
import operator

numbers = [1, 2, 3, 4, 5]
total_sum = reduce(operator.add, numbers)
# > 15
```

5.5 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 R code is a sequence of function compositions:

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

The same idea in Python with pandas:

```python
(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.

5.5.1 The Composition Challenge in Python

Method chaining in pandas is elegant but limited to the methods defined for `DataFrame` objects. R's pipe operators (`|>` and `%>%`) are more flexible because functions are not strictly owned by objects, and they can be more easily combined.

This reflects the languages' different philosophies:

- **R**'s lineage traces back to Scheme (a Lisp dialect), making functional composition natural
- **Python** was designed around "one obvious way to do it," favouring explicit loops and method chaining over function composition

R is fundamentally a functional language that acquired OOP features, while Python is an OOP language with powerful functional capabilities. Recognising this helps you leverage the native strengths of each.

5.6 Handling Errors Functionally

What happens when a function in your pipeline fails? In imperative code, you might wrap everything in try/catch blocks. Functional programming offers a cleaner approach: functions that *capture* errors rather than throw them.

5.6.1 In R with `purrr::safely()` and `purrr::possibly()`

The {purrr} package provides wrappers that turn error-prone functions into safe ones:

```
library(purrr)

# A function that might fail
risky_log <- function(x) {
  if (x <= 0) stop("x must be positive")
  log(x)
}
```

```
# safely() returns a list with $result and $error
safe_log <- safely(risky_log)
safe_log(10)   # list(result = 2.302585, error =
 ↪  NULL)
```

```
$result
[1] 2.302585

$error
NULL
```

```
safe_log(-1)   # list(result = NULL, error =
 ↪  <error>)
```

```
$result
NULL

$error
<simpleError in .f(...): x must be positive>
```

```
# possibly() returns a default value on error
maybe_log <- possibly(risky_log, otherwise = NA)
map_dbl(c(10, -1, 5), maybe_log)  # c(2.30, NA,
 ↪  1.61)
```

```
[1] 2.302585       NA 1.609438
```

This lets your pipeline continue even when some elements fail, and you can inspect failures afterwards.

5.6.2 In Python with try/except and Result patterns

Python's traditional approach uses `try/except`:

```
def safe_log(x):
    try:
        if x <= 0:
            raise ValueError("x must be positive")
        import math
        return math.log(x)
    except ValueError:
        return None

results = [safe_log(x) for x in [10, -1, 5]]
# [2.302585, None, 1.609437]
```

For a more functional approach, libraries like `returns` provide a `Result` type that explicitly represents success or failure. We will explore this pattern further in the chapter on monads.

5.7 When NOT to Use Functional Programming

Functional programming is powerful, but it is not always the best choice. Here are situations where imperative or object-oriented code may be clearer:

1. **Complex stateful algorithms**: Some algorithms (like graph traversals or simulations) naturally require mutable

state. Forcing them into a functional style can make the code harder to read.

2. **Performance-critical inner loops**: Functional abstractions like `map()` introduce overhead. In tight loops where microseconds matter, a simple `for` loop may be faster, especially in Python.
3. **Interactive or exploratory work**: When exploring data interactively, it is fine to use scripts and notebooks to iterate quickly. The functional discipline matters most when code needs to be shared, tested, or maintained.
4. **When your team is not familiar**: Code is read more often than written. If your colleagues are not comfortable with functional patterns, overly abstract code becomes a liability.

The goal is not functional purity for its own sake, but **clarity and correctness**. Use functional techniques where they help, and step back to simpler approaches when they do not.

5.8 Summary

This chapter has laid the groundwork for writing reproducible code by embracing **Functional Programming**.

Key takeaways:

- **Pure functions** guarantee the same output for the same input, with no hidden dependencies on global state
- Make impure operations (like randomness) explicit by controlling the seed
- Replace error-prone `for` loops with **map, filter, and reduce**

- Use **composition** to build complex pipelines from simple, reusable functions
- In Python, treat stateful library objects as values passed between pure functions

Understanding the distinction between R's functional heritage and Python's OOP nature is key to becoming an effective data scientist in either language. By mastering the functional paradigm, you're building a foundation for code that is robust, easy to review, simple to debug, and truly reproducible.

In the next chapter, we'll put these principles into practice with **`{rixpress}`**, a package that leverages functional composition and Nix to build fully reproducible, polyglot analytical pipelines.

6 Building Reproducible Pipelines with rixpress

6.1 Introduction: From Scripts and Notebooks to Pipelines

So far, we have learned about reproducible development environments with Nix and `{rix}`. We can now create project-specific environments with precise versions of R, Python, and all dependencies. But there's one more piece to the puzzle: **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.

6.1.1 The Evolution of Build Automation

The original solution, dating back to the 1970s, is `make`. Created by Stuart Feldman at Bell Labs in 1976, `make` reads a `Makefile` that describes the dependency graph of a project. 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.

The strength of these tools is their language-agnosticism, but their weaknesses are twofold:

1. **File-centric**: You must manually handle all I/O. Your first script saves `data.csv`, your second loads it. This adds boilerplate and surfaces for error.
2. **Environment-agnostic**: They track files but know nothing about the *software environment* needed to create those files.

This is where R's **`{targets}`** package shines. It tracks dependencies between **R objects directly**, automatically handling serialisation. But `{targets}` operates within a single R session; for polyglot pipelines, you must manually coordinate via `{reticulate}`.

6.1.2 The Separation Problem

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 (Docker, `{renv}`) to set up the software.

This separation creates friction. Running `{targets}` inside Docker ensures reproducibility, but forces the entire pipeline into one monolithic environment. What if your Python step requires TensorFlow 2.15 but your R step needs reticulate with Python 3.9? You're stuck.

6.1.3 The Imperative Approach: Make + Docker

To illustrate this, consider the traditional setup for a polyglot pipeline. You'd need:

1. A `Dockerfile` to set up the environment
2. A `Makefile` to orchestrate the workflow
3. Wrapper scripts for each step

Here's what a `Makefile` might look like:

```
# Makefile for a Python → R pipeline

DATA_DIR = data
OUTPUT_DIR = output

$(OUTPUT_DIR)/predictions.csv: $(DATA_DIR)/raw.csv
 ↪  scripts/train_model.py
    python scripts/train_model.py
 ↪  $(DATA_DIR)/raw.csv $@

$(OUTPUT_DIR)/plot.png:
 ↪  $(OUTPUT_DIR)/predictions.csv
 ↪  scripts/visualise.R
    Rscript scripts/visualise.R $< $@

$(OUTPUT_DIR)/report.html: $(OUTPUT_DIR)/plot.png
 ↪  report.qmd
    quarto render report.qmd -o $@

all: $(OUTPUT_DIR)/report.html

clean:
    rm -rf $(OUTPUT_DIR)/*
```

This looks clean, but notice the procedural boilerplate required in each script. Your `train_model.py` must parse command-line arguments and handle file I/O:

```
# scripts/train_model.py
import sys
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

def main():
    input_path = sys.argv[1]
    output_path = sys.argv[2]

    # Load data
    df = pd.read_csv(input_path)

    # ... actual ML logic ...

    # Save results
    predictions.to_csv(output_path, index=False)

if __name__ == "__main__":
    main()
```

And your `visualise.R` script needs the same boilerplate:

```
# scripts/visualise.R
args <- commandArgs(trailingOnly = TRUE)
input_path <- args[1]
output_path <- args[2]

# Load data
predictions <- read.csv(input_path)

# ... actual visualisation logic ...

# Save plot
```

```
ggsave(output_path, plot)
```

The scientific logic is buried under file I/O scaffolding. And the environment? That's a separate 100+ line `Dockerfile` you must maintain.

6.1.4 rixpress: Unified Orchestration

`{rixpress}` solves this by using Nix not just as a package manager, but as the build automation engine itself. Each pipeline step is a **Nix derivation**: a hermetically sealed build unit.

Compare the same pipeline in `{rixpress}`:

```
library(rixpress)

list(
  rxp_py_file(
    name = raw_data,
    path = "data/raw.csv",
    read_function = "lambda x: pandas.read_csv(x)"
  ),

  rxp_py(
    name = predictions,
    expr = "train_model(raw_data)",
    user_functions = "functions.py"
  ),

  rxp_py2r(
    name = predictions_r,
    expr = predictions
```

```
  ),

  rxp_r(
    name = plot,
    expr = visualise(predictions_r),
    user_functions = "functions.R"
  ),

  rxp_qmd(
    name = report,
    qmd_file = "report.qmd"
  )
) |>
  rxp_populate()
```

And your `functions.py` contains *only* the scientific logic:

```
# functions.py
def train_model(df):
    # ... pure ML logic, no file I/O ...
    return predictions
```

The difference is stark:

Aspect	Make + Docker	rixpress
Files needed	Dockerfile, Makefile, wrapper scripts	gen-env.R, gen-pipeline.R, function files
I/O handling	Manual in every script	Automatic via encoders/decoders

Aspect	Make + Docker	rixpress
Dependencies	Explicit file rules	Inferred from object references
Environment	Separate Docker setup	Unified via `{rix}`
Expertise needed	Linux admin, Make syntax	R programming

This provides two key benefits:

1. **True Polyglot Pipelines**: Each step can have its own Nix environment. A Python step runs in a pure Python environment, an R step in an R environment, a Quarto step in yet another, all within the same pipeline.
2. **Deep Reproducibility**: Each step is cached based on the cryptographic hash of *all* its inputs: the code, the data, *and the environment*. Any change in dependencies triggers a rebuild. This is reproducibility at the build level, not just the environment level.

The interface is heavily inspired by `{targets}`, so you get the ergonomic, object-passing feel you're used to, combined with the bit-for-bit reproducibility of the Nix build system.

6.2 What is rixpress?

`{rixpress}` streamlines creation of *micropipelines* (small-to-medium, single-machine analytic pipelines) by expressing a pipeline in idiomatic R while delegating build orchestration to the Nix build system.

Key features:

- Define pipeline *derivations* with concise `rxp_*()` helper functions
- Seamlessly mix R, Python, Julia, and Quarto steps
- Reuse hermetic environments defined via `{rix}` and a `default.nix`
- Visualise and inspect the DAG; selectively read, load, or copy outputs
- Automatic caching: only rebuild what changed

Here is what a basic pipeline looks like:

```
library(rixpress)

list(
  rxp_r_file(
    mtcars,
    'mtcars.csv',
    \(x) read.csv(file = x, sep = "|")
  ),

  rxp_r(
    mtcars_am,
    filter(mtcars, am == 1)
  ),

  rxp_r(
    mtcars_head,
    head(mtcars_am)
  ),

  rxp_qmd(
    page,
    "page.qmd"
  )
```

```
) |>
  rxp_populate()
```

6.3 Getting Started

6.3.1 Initialising a project

If you're starting fresh, you can bootstrap a project using a temporary shell:

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

Once inside, start R and run:

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

This creates two essential files:

- **gen-env.R**: Where you define your environment with {rix}
- **gen-pipeline.R**: Where you define your pipeline with {rixpress}

6.3.2 Defining the environment

Open gen-env.R and define the tools your pipeline needs:

```
library(rix)

rix(
  date = "2025-10-14",
  r_pkgs = c("dplyr", "ggplot2", "quarto",
   ↪  "rixpress"),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Run this script to generate `default.nix`, then build and enter your environment:

```
nix-build
nix-shell
```

6.3.3 Defining the pipeline

Open `gen-pipeline.R` and define your pipeline:

```
library(rixpress)

list(
  rxp_r_file(
    name = mtcars,
    path = "data/mtcars.csv",
    read_function = \(x) read.csv(x, sep = "|")
  ),

  rxp_r(
```

```
    name = mtcars_am,
    expr = dplyr::filter(mtcars, am == 1)
  ),

  rxp_r(
    name = mtcars_head,
    expr = head(mtcars_am)
  )
) |>
  rxp_populate()
```

Running `rxp_populate()` generates a `pipeline.nix` file and builds the entire pipeline.

6.4 Core Functions

6.4.1 Defining derivations

`{rixpress}` provides several functions to define pipeline steps:

Function	Purpose
`rxp_r()`	Run R code
`rxp_r_file()`	Read a file using R
`rxp_py()`	Run Python code
`rxp_py_file()`	Read a file using Python
`rxp_qmd()`	Render a Quarto document
`rxp_py2r()`	Convert Python object to R
`rxp_r2py()`	Convert R object to Python

6.4.2 Building the pipeline

```
# Generate pipeline.nix only (don't build)
rxp_populate(build = FALSE)

# Build the pipeline
rxp_make()
```

6.4.3 Inspecting outputs

Because outputs live in `/nix/store/`, `{rixpress}` provides helpers:

```
# List all built artifacts
rxp_inspect()

# Read an artifact into R
result <- rxp_read("mtcars_head")

# Load an artifact into the global environment
rxp_load("mtcars_head")

# Copy an output file to current directory
rxp_copy("page")
```

6.4.4 Visualising the pipeline

```
# Static DAG plot
rxp_ggdag()

# Interactive network
rxp_visnetwork()

# Text-based trace
rxp_trace()
```

6.5 Polyglot Pipelines

One of `{rixpress}`'s strengths is seamlessly mixing languages. Here's a pipeline that reads data with Python's `polars`, processes it with R's `dplyr`, and renders a Quarto report:

```
library(rixpress)

list(
  rxp_py_file(
    name = mtcars_pl,
    path = "data/mtcars.csv",
    read_function = "lambda x: polars.read_csv(x,
     ↪  separator='|')"
  ),

  rxp_py(
    name = mtcars_filtered,
    expr = "mtcars_pl.filter(polars.col('am') ==
     ↪  1).to_pandas()"
  ),
```

```
  rxp_py2r(
    name = mtcars_r,
    expr = mtcars_filtered
  ),

  rxp_r(
    name = mtcars_head,
    expr = head(mtcars_r)
  ),

  rxp_qmd(
    name = report,
    qmd_file = "report.qmd"
  )
) |>
  rxp_populate()
```

6.5.1 Method 1: Using language converters

The `rxp_py2r()` and `rxp_r2py()` functions use `{reticulate}` to convert objects between languages:

```
rxp_py2r(
  name = mtcars_r,
  expr = mtcars_py
)
```

6.5.2 Method 2: Using universal data formats

For more control, use `encoder` and `decoder` arguments to serialize to formats like JSON:

```
# Python step: serialize to JSON
rxp_py(
  name = mtcars_json,
  expr = "mtcars_pl.filter(polars.col('am') == 1)",
  user_functions = "functions.py",
  encoder = "serialize_to_json"
),

# R step: deserialize from JSON
rxp_r(
  name = mtcars_head,
  expr = my_head(mtcars_json),
  user_functions = "functions.R",
  decoder = "jsonlite::fromJSON"
)
```

This approach makes your pipeline more modular: any language that can read JSON could be added in the future.

6.6 Caching and Incremental Builds

One of the most powerful features of using Nix for pipelines is automatic caching. Because Nix tracks all inputs to each derivation, it knows exactly what needs to be rebuilt when something changes.

Try this:

1. Build your pipeline with `rxp_make()`
2. Change one step in your pipeline
3. Run `rxp_make()` again

Nix will detect that unchanged steps are already cached and instantly reuse them. It only rebuilds the steps affected by your change.

6.7 Build Logs and Debugging

Every time you run `rxp_populate()`, a timestamped log is saved in the `_rixpress/` directory. This is like having a Git history for your pipeline's *outputs*.

```
# List all past builds
rxp_list_logs()

# Load artifact from current build
new_result <- rxp_read("mtcars_head")

# Load artifact from previous build
old_result <- rxp_read("mtcars_head", which_log =
 ↪  "z9y8x")

# Compare them
identical(new_result, old_result)
```

This is incredibly powerful for debugging and validation. You can go back in time to inspect any output from any previous pipeline run.

6.8 ryxpress: The Python Interface

If you prefer working in Python, `ryxpress` provides the same functionality. You still define your pipeline in R (since that's where `{rixpress}` runs), but you can build and inspect artifacts from Python.

To set up an environment with `ryxpress`:

```
rix(
  date = "2025-10-14",
  r_pkgs = c("rixpress"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("ryxpress", "rds2py", "biocframe",
    ↪  "pandas")
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Then from Python:

```
from ryxpress import rxp_make, rxp_inspect, rxp_load

# Build the pipeline
rxp_make()

# Inspect artifacts
rxp_inspect()

# Load an artifact
```

```
rxp_load("mtcars_head")
```

`ryxpress` handles the conversion automatically:

- Tries `pickle.load` first
- Falls back to `rds2py` for R objects
- Returns file paths for complex outputs

6.9 Running Someone Else's Pipeline

The ultimate test of reproducibility: can someone else run your pipeline?

With a Nix-based workflow, they need only:

1. `git clone` your repository
2. Run `nix-build && nix-shell`
3. Run `source("gen-pipeline.R")` or `rxp_make()`

That's it. Nix reads your `default.nix` and `pipeline.nix` files and builds the *exact* same environment and data product, bit-for-bit.

6.10 Exporting and Importing Artifacts

For CI/CD or sharing between machines:

```
# Export build products to a tarball
rxp_export_artifacts()

# Import on another machine before building
rxp_import_artifacts()
```

This speeds up continuous integration by avoiding unnecessary rebuilds.

6.11 Real-World Examples

The rixpress_demos repository contains many complete examples. Here are a few patterns worth studying.

6.11.1 Example 1: Machine Learning with XGBoost

This pipeline trains an XGBoost classifier in Python, then passes predictions to R for evaluation with `{yardstick}`:

```
library(rixpress)

list(
  # Load data as NumPy array
  rxp_py_file(
    name = dataset_np,
    path = "data/pima-indians-diabetes.csv",
    read_function = "lambda x: loadtxt(x,
     ↪  delimiter=',')"
```

```
),

# Split features and target
rxp_py(name = X, expr = "dataset_np[:,0:8]"),
rxp_py(name = Y, expr = "dataset_np[:,8]"),

# Train/test split
rxp_py(
  name = splits,
  expr = "train_test_split(X, Y, test_size=0.33,
   ↪  random_state=7)"
),

# Extract splits

rxp_py(name = X_train, expr = "splits[0]"),
rxp_py(name = X_test, expr = "splits[1]"),
rxp_py(name = y_train, expr = "splits[2]"),
rxp_py(name = y_test, expr = "splits[3]"),

# Train XGBoost model
rxp_py(
  name = model,
  expr = "XGBClassifier(use_label_encoder=False,
   ↪  eval_metric='logloss').fit(X_train,
   ↪  y_train)"
),

# Make predictions
rxp_py(name = y_pred, expr =
 ↪  "model.predict(X_test)"),

# Export predictions to CSV for R
```

```
  rxp_py(
    name = combined_df,
    expr = "DataFrame({'truth': y_test, 'estimate':
     ↪  y_pred})"
  ),

  rxp_py(
    name = combined_csv,
    expr = "combined_df",
    user_functions = "functions.py",
    encoder = "write_to_csv"
  ),

  # Compute confusion matrix in R
  rxp_r(
    combined_factor,
    expr = mutate(combined_csv, across(everything(),
     ↪  factor)),
    decoder = "read.csv"
  ),

  rxp_r(
    name = confusion_matrix,
    expr = yardstick::conf_mat(combined_factor,
     ↪  truth, estimate)
  )
) |>
  rxp_populate(build = FALSE)

# Adjust Python imports
adjust_import("import numpy", "from numpy import
 ↪  array, loadtxt")
adjust_import("import xgboost", "from xgboost import
 ↪  XGBClassifier")
```

```
adjust_import("import sklearn", "from
 ↪  sklearn.model_selection import
 ↪  train_test_split")
add_import("from pandas import DataFrame",
 ↪  "default.nix")

rxp_make()
```

This demonstrates:

- Python-heavy computation with XGBoost
- Custom serialization via `encoder`/`decoder`
- Adjusting Python imports with `adjust_import()` and `add_import()`
- Passing results to R for evaluation

6.11.2 Example 2: Reading Many Input Files

When you have multiple CSV files in a directory:

```
library(rixpress)

list(
  # R approach: read all files at once
  rxp_r_file(
    name = mtcars_r,
    path = "data",
    read_function = \(x) {
      readr::read_delim(list.files(x, full.names =
 ↪  TRUE), delim = "|")
    }
  ),
```

```
  # Python approach: custom function
  rxp_py_file(
    name = mtcars_py,
    path = "data",
    read_function = "read_many_csvs",
    user_functions = "functions.py"
  ),

  rxp_py(
    name = head_mtcars,
    expr = "mtcars_py.head()"
  )
) |>
  rxp_populate()
```

The key insight: `rxp_r_file()` and `rxp_py_file()` can point to a directory, and your `read_function` handles the logic.

6.11.3 Example 3: Full Python→R→Quarto Workflow

A complete pipeline that bounces data between languages and renders a report:

```
library(rixpress)

list(
  # Read with Python polars
  rxp_py_file(
    name = mtcars_pl,
```

```
  path = "data/mtcars.csv",
  read_function = "lambda x: polars.read_csv(x,
   ↪  separator='|')"
),

# Filter in Python, convert to pandas for
 ↪  reticulate
rxp_py(
  name = mtcars_pl_am,
  expr = "mtcars_pl.filter(polars.col('am') ==
   ↪  1).to_pandas()"
),

# Convert to R
rxp_py2r(name = mtcars_am, expr = mtcars_pl_am),

# Process in R
rxp_r(
  name = mtcars_head,
  expr = my_head(mtcars_am),
  user_functions = "functions.R"
),

# Back to Python
rxp_r2py(name = mtcars_head_py, expr =
 ↪  mtcars_head),

# More Python processing
rxp_py(name = mtcars_tail_py, expr =
 ↪  "mtcars_head_py.tail()"),

# Back to R
rxp_py2r(name = mtcars_tail, expr =
 ↪  mtcars_tail_py),
```

```
  # Final R step
  rxp_r(name = mtcars_mpg, expr =
   ↪  dplyr::select(mtcars_tail, mpg)),

  # Render Quarto document
  rxp_qmd(
    name = page,
    qmd_file = "my_doc/page.qmd",
    additional_files = c("my_doc/content.qmd",
     ↪  "my_doc/images")
  )
) |>
  rxp_populate()
```

Note the `additional_files` argument for `rxp_qmd()`: this includes child documents and images that the main Quarto file needs.

6.11.4 More Examples

The rixpress_demos repository includes:

- **jl_example**: Using Julia in pipelines
- **r_qs**: Using `{qs}` for faster serialization
- **python_r_typst**: Compiling to Typst documents
- **r_multi_envs**: Different Nix environments for different derivations
- **yanai_lercher_2020**: Reproducing a published paper's analysis

6.12 Summary

`{rixpress}` unifies environment management and workflow orchestration:

- **Define** pipelines with `rxp_*()` functions in familiar R syntax
- **Mix** languages freely: R, Python, Julia, Quarto
- **Build** with Nix for deterministic, cached execution
- **Inspect** outputs with `rxp_read()`, `rxp_load()`, `rxp_copy()`
- **Debug** with timestamped build logs
- **Share** reproducible pipelines via Git

The Python port `ryxpress` provides the same experience for Python-first workflows.

By embracing structured, plain-text pipelines over notebooks for production work, your analysis becomes more reliable, more scalable, and fundamentally more reproducible.

7 Containerisation With Docker

7.1 Introduction

Up until now, we've been using Nix as a powerful tool for creating reproducible development environments directly on our machines. Nix gives us fine-grained control over every package and dependency in our project, ensuring bit-for-bit reproducibility. However, when it comes to distributing a *data product*, another technology, Docker, is incredibly popular.

While Nix manages dependencies for an application that runs on a host operating system, Docker takes a different approach: it packages an application *along with* a lightweight operating system and all its dependencies into a single, portable unit called a **container**. This container can then run on any machine that has Docker installed, regardless of its underlying OS.

7.1.1 Spatial vs Temporal Reproducibility

To understand when and why to use Docker, it helps to distinguish between two types of reproducibility:

- **Spatial reproducibility**: The ability to execute an analysis identically across different machines *right now*. Docker excels here—a container runs the same way on your laptop, a colleague's workstation, and a cloud server.
- **Temporal reproducibility**: The ability to execute an analysis identically *over time*. Docker is weaker here. The imperative commands in a `Dockerfile` (like `apt-get update`) are non-deterministic—rebuilding the same file at different times can yield different images.

This distinction is crucial. Docker's true reproducibility promise is that a specific, pre-built image will always launch an identical container. It does *not* promise that building the same `Dockerfile` twice will yield an identical image.

As empirical research has shown, achieving deterministic builds requires systems designed for this purpose—like Nix's functional model, where identical inputs always produce identical outputs. Studies rebuilding historical Nix packages found bit-for-bit reproducibility rates exceeding 90%.

7.1.2 The Best of Both Worlds

This is why we combine Nix and Docker rather than choosing one or the other:

- **Nix** provides strong temporal reproducibility—the guarantee that an environment built today will be identical when rebuilt years later.
- **Docker** provides strong spatial reproducibility and universal distribution—the guarantee that a container runs the same everywhere Docker runs.

The pattern we'll use is simple: use Nix *inside* a Docker container. Start with a minimal base image that has Nix installed. Then, use Nix to declaratively build the precise environment within the image. Docker becomes a portable runtime for this Nix-managed environment, excellent for deployment to systems where Nix isn't installed.

7.1.3 Interactive Development vs Distribution

This approach also clarifies *when* to use each tool:

- **Use Nix directly** for interactive development. It integrates seamlessly with your IDE and filesystem. No volume mounts, no port forwarding, no graphical application headaches.
- **Use Docker** for distributing finished data products. Package your `{rixpress}` pipeline into an image that anyone can run with a single command.

If you've never heard of Docker before, this chapter will provide the basic knowledge required to get started. Let's start by watching this very short video that introduces the core concepts.

The Rocker Project provides a large collection of ready-to-use Docker images for R users.

7.2 Docker Essentials

7.2.1 Installing Docker

The first step is to install Docker. You'll find the instructions for Ubuntu here, for Windows here (read the system requirements

section as well!) and for macOS here (make sure to choose the right version for the architecture of your Mac, if you have an M1 Mac use *Mac with Apple silicon*).

After installation, it might be a good idea to restart your computer, if the installation wizard does not invite you to do so. To check whether Docker was installed successfully, run the following command in a terminal:

```
docker run --rm hello-world
```

This should print the following message:

```
Hello from Docker!
This message shows that your installation appears to
be working correctly.

To generate this message, Docker took the following
steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image
 from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from
 that image which runs the
    executable that produces the output you are
    currently reading.
 4. The Docker daemon streamed that output to the
 Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an
Ubuntu container with:
 $ docker run -it ubuntu bash
```

```
Share images, automate workflows, and more with a
free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

If you see this message, congratulations, you are ready to run Docker. If you see an error message about permissions, this means that something went wrong. If you're running Linux, make sure that your user is in the Docker group by running:

```
groups $USER
```

You should see your username and a list of groups that your user belongs to. If a group called `docker` is not listed, then you should add yourself to the group by following these steps.

7.2.2 The Rocker Project and Image Registries

When running a command like:

```
docker run --rm hello-world
```

what happens is that an image, in this case `hello-world`, gets pulled from a so-called *registry*. A registry is a storage and distribution system for Docker images. Think of it as a GitHub for Docker images, where you can push and pull images, much like you would with code repositories. The default public registry that Docker uses is called Docker Hub, but companies can also host their own private registries to store proprietary images.

Many open source projects build and distribute Docker images through Docker Hub, for example the Rocker Project.

The Rocker Project is instrumental for R users that want to use Docker. The project provides a large list of images that are ready to run with a single command. As an illustration, open a terminal and paste the following line:

```
docker run --rm -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/rstudio
```

Once this stops running, go to `http://localhost:8787/` and enter `rstudio` as the username and `yourpassword` as the password. You should login to a RStudio instance: this is the web interface of RStudio that allows you to work with R from a server. In this case, the *server* is the Docker container running the image. Yes, you've just pulled a Docker image containing Ubuntu with a fully working installation of RStudio web!

Let's open a new script and run the following lines:

```
data(mtcars)
summary(mtcars)
```

You can now stop the container (by pressing `CTRL-C` in the terminal). Let's now rerun the container… you should realise that your script is gone! This is the first lesson: whatever you do inside a container will disappear once the container is stopped. This also means that if you install the R packages that you need while the container is running, you will need to reinstall them every time.

Thankfully, the Rocker Project provides a list of images with many packages already available. For example to run R with the `{tidyverse}` collection of packages already pre-installed, run:

```
docker run --rm -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/tidyverse
```

7.2.3 Basic Docker Workflow

You already know about running containers using `docker run`. With the commands we ran before, your terminal will need to stay open, or else, the container will stop. Starting now, we will run Docker commands in the background using the `-d` flag (*d* as in *detach*):

```
docker run --rm -d -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/tidyverse
```

You can run several containers in the background simultaneously. List running containers with `docker ps`:

```
docker ps
CONTAINER ID   IMAGE              COMMAND   CREATED
 ↪  STATUS         PORTS                    NAMES
c956fbeeebcb   rocker/tidyverse   "/init"   3
 ↪  minutes ago   Up 3 minutes
 ↪  0.0.0.0:8787->8787/tcp   elastic_morse
```

Stop the container using its ID:

```
docker stop c956fbeeebcb
```

Let's discuss the other flags:

- `--rm`: Removes the container once it's stopped
- `-e`: Provides environment variables to the container (e.g., `PASSWORD`)
- `-p`: Sets the port mapping (host:container)
- `--name`: Gives the container a custom name

Run a container with a name:

```
docker run -d --name my_r --rm -e
 ↪  PASSWORD=yourpassword -p 8787:8787
 ↪  rocker/tidyverse
```

You can now interact with this container using its name:

```
docker exec -ti my_r bash
```

You are now inside a terminal session, inside the running container! This can be useful for debugging purposes.

Finally, let's solve the issue of our scripts disappearing. Create a folder somewhere on your computer, then run:

```
docker run -d --name my_r --rm -e
 ↪  PASSWORD=yourpassword -p 8787:8787 \
  -v
   ↪  /path/to/your/local/folder:/home/rstudio/scripts:rw
   ↪  rocker/tidyverse
```

You should now be able to save scripts inside the `scripts/` folder from RStudio and they will appear in the folder you created on your host machine.

7.3 Making Our Own Images

To create our own images, you can start from an image provided by an open source project like Rocker, or you can start from the base Ubuntu image. Since we are using Nix to set up the reproducible development environment, we can use `ubuntu:latest`—our development environment will always be exactly the same, thanks to Nix.

7.3.1 A Minimal Dockerfile with Nix

```
FROM ubuntu:latest

RUN apt update -y
RUN apt install curl -y

# Download the default.nix that comes with {rix}
RUN curl -O
 ↪  https://raw.githubusercontent.com/ropensci/rix/main/inst/

# Install Nix via Determinate Systems installer
RUN curl --proto '=https' --tlsv1.2 -sSf -L
 ↪  https://install.determinate.systems/nix | sh -s
 ↪  -- install linux \
  --extra-conf "sandbox = false" \
  --init none \
  --no-confirm

# Add Nix to the path
ENV PATH="${PATH}:/nix/var/nix/profiles/default/bin"
ENV user=root
```

```
# Configure rstats-on-nix cache
RUN mkdir -p /root/.config/nix && \
    echo "substituters = https://cache.nixos.org
     ↪ https://rstats-on-nix.cachix.org" >
     ↪ /root/.config/nix/nix.conf && \
    echo "trusted-public-keys =
     ↪ cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGsp
     ↪ rstats-on-nix.cachix.org-1:vdiiVgocg6WeJrODIqdprZRUrhi1
     ↪ >> /root/.config/nix/nix.conf

# Copy script to generate environment
COPY gen-env.R .

# Generate default.nix from gen-env.R
RUN nix-shell --run "Rscript gen-env.R"

# Build the environment
RUN nix-build

# Run nix-shell when container starts
CMD nix-shell
```

Every `Dockerfile` starts with a `FROM` statement specifying the base image. Then, every command starts with `RUN`. We install and configure Nix, copy an R script to generate the environment, and then build it. Finally, we run `nix-shell` when executing a container (the `CMD` statement).

7.3.2 Splitting Into a Reusable Base Image

Because the Nix installation step is generic, we can split this into two stages. First, create a base image with just Nix installed:

```
# nix-base/Dockerfile
FROM ubuntu:latest AS nix-base

RUN apt update -y && apt install -y curl

# Install Nix via Determinate Systems installer
RUN curl --proto '=https' --tlsv1.2 -sSf -L
 ↪  https://install.determinate.systems/nix | sh -s
 ↪  -- install linux \
  --extra-conf "sandbox = false" \
  --init none \
  --no-confirm

ENV PATH="/nix/var/nix/profiles/default/bin:${PATH}"
ENV user=root

# Configure Nix binary cache
RUN mkdir -p /root/.config/nix && \
    echo "substituters = https://cache.nixos.org
     ↪  https://rstats-on-nix.cachix.org" >
     ↪  /root/.config/nix/nix.conf && \
    echo "trusted-public-keys =
     ↪  cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQF
     ↪  rstats-on-nix.cachix.org-1:vdiiVgocg6WeJrODIqdprZRUr
     ↪  >> /root/.config/nix/nix.conf
```

Build and tag this image:

```
docker build -t nix-base:latest .
```

Now, for any project, simply reuse it:

```
FROM nix-base:latest

COPY gen-env.R .

RUN curl -O
 ↪  https://raw.githubusercontent.com/ropensci/rix/main/inst/ex
RUN nix-shell --run "Rscript gen-env.R"
RUN nix-build

CMD ["nix-shell"]
```

With `gen-env.R`:

```
library(rix)

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

Build and run:

```
docker build -t my-project .
docker run -it --rm --name my-project-container
 ↪  my-project
```

This drops you in an interactive Nix shell running inside Docker!

7.4 Publishing Images on Docker Hub

If you want to share Docker images through Docker Hub, you first need to create a free account. A free account gives you unlimited public repositories.

List all images on your computer:

```
docker images
```

Log in to Docker Hub:

```
docker login
```

Tag the image with your username:

```
docker tag IMAGE_ID your_username/nix-base:latest
```

Push the image:

```
docker push your_username/nix-base:latest
```

This image can now be used as a stable base for other projects:

```
FROM your_username/nix-base:latest

RUN mkdir ...
```

7.4.1 Sharing Without Docker Hub

If you can't upload to Docker Hub, you can save the image to a file:

```
docker save nix-base | gzip > nix-base.tgz
```

Load it on another machine:

```
gzip -d nix-base.tgz
docker load < nix-base.tar
```

7.5 What If You Don't Use Nix?

Using Nix inside of Docker makes it very easy to set up an environment, but what if you can't use Nix for some reason? In this case, you would need to use other tools to install the right R or Python packages and it is likely going to be more difficult. The main issue you will face is missing development libraries.

For example, to install and use the R `{stringr}` package, you will need to first install `libicu-dev`:

```
FROM rocker/r-ver:4.5.1

RUN apt-get update && apt-get install -y \
    libglpk-dev \
    libxml2-dev \
    libcairo2-dev \
    libgit2-dev \
    libcurl4-openssl-dev \
    # ... many more
```

Another issue is that building the image is not a reproducible process, only running containers is. To mitigate this, use tagged images or better yet, a digest:

```
FROM
 ↪ rocker/r-ver@sha256:1dbe7a6718b7bd8630addc45a32731624fb7b
```

This will always pull exactly the same layers. However, at some point, that version of Ubuntu will be outdated. Using Nix, you can stay on `ubuntu:latest`.

7.6 Building Docker Images Directly with Nix

So far, we've used Nix *inside* Docker containers. But there's an even more powerful approach: using Nix to build Docker images directly, bypassing `Dockerfile`s entirely.

Nix provides `dockerTools`, a set of functions that can create OCI-compliant container images. Because these images are

built through Nix's deterministic build system, they have stronger reproducibility guarantees than images built with `docker build`.

7.6.1 Why Skip the Dockerfile?

A `Dockerfile` is fundamentally imperative—it's a script of commands executed in order. Commands like `apt-get update` are non-deterministic by nature. Even with careful pinning, you're fighting the tool's design.

Nix's `dockerTools.buildImage` is declarative. You describe what should be in the image, and Nix figures out how to build it reproducibly. The resulting image is a pure function of its inputs.

7.6.2 A Basic Example

Here's a Nix expression that builds a Docker image containing R and some packages:

```
# docker-image.nix
let
  pkgs = import (fetchTarball

    ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/2025-
  ) {};

  # Define the R environment
  myR = pkgs.rWrapper.override {
    packages = with pkgs.rPackages; [
      dplyr
```

```
      ggplot2
      quarto
    ];
  };

in pkgs.dockerTools.buildImage {
  name = "my-r-env";
  tag = "latest";

  copyToRoot = pkgs.buildEnv {
    name = "image-root";
    paths = [ myR pkgs.quarto pkgs.coreutils
    ↪  pkgs.bash ];
    pathsToLink = [ "/bin" ];
  };

  config = {
    Cmd = [ "${pkgs.bash}/bin/bash" ];
    WorkingDir = "/work";
  };
}
```

Build and load it:

```
# Build the image (outputs a .tar.gz file)
nix-build docker-image.nix

# Load into Docker
docker load < result

# Run it
docker run -it --rm my-r-env:latest
```

7.6.3 Benefits Over Dockerfiles

Aspect	Dockerfile	Nix dockerTools
Reproducibility	Imperative, non-deterministic	Declarative, deterministic
Layer caching	Based on command order	Based on content hashes
Image size	Often includes unnecessary packages	Minimal—only what you specify
Composition	Copy-paste between files	Reuse Nix expressions

7.6.4 Layered Images for Efficiency

For faster CI builds, you can create layered images where the base environment is cached:

```
pkgs.dockerTools.buildLayeredImage {
  name = "my-analysis";
  tag = "latest";

  contents = [ myR pkgs.quarto ];

  config = {
    Cmd = [ "${myR}/bin/R" "--vanilla" ];
  };
}
```

`buildLayeredImage` creates separate layers for each package, so unchanged dependencies are cached between builds.

7.6.5 When to Use This Approach

- **CI/CD pipelines**: When you need reproducible image builds as part of automated workflows
- **Minimal images**: When image size matters and you want only what you need
- **Complex environments**: When the environment is already defined in Nix and you want to deploy it as a container

For simpler use cases, the "Nix inside Docker" approach from earlier sections may be more accessible.

7.7 Dockerizing a rixpress Pipeline

We can package our entire `{rixpress}` project into a single Docker image. This image can then be run by anyone with Docker installed, regardless of their host operating system or whether they have Nix.

Assume your project directory has:

```
.
  data/
     mtcars.csv
  gen-env.R
  gen-pipeline.R
  functions.R
  functions.py
  default.nix
  pipeline.nix
```

7.7.1 Step 1: The Dockerfile

```
FROM nix-base:latest
# Or: FROM your-username/nix-base:latest

WORKDIR /app

# Copy all project files
COPY . .

# Build the pipeline during image build
RUN nix-build pipeline.nix

# Export results when container runs
COPY export-results.R .
CMD ["nix-shell", "--run", "Rscript
 ↪  export-results.R"]
```

7.7.2 Step 2: The Export Script

```
# export-results.R
library(rixpress)
library(jsonlite)

output_dir <- "/output"
dir.create(output_dir, showWarnings = FALSE)

message("Reading target 'mtcars_head'...")
final_data <- rxp_read("mtcars_head")
```

```
output_path <- file.path(output_dir,
 ↪  "mtcars_analysis_result.json")
write_json(final_data, output_path, pretty = TRUE)

message(paste("Successfully exported result to",
 ↪  output_path))
```

7.7.3 Step 3: Build and Run

Build:

```
docker build -t my-reproducible-pipeline .
```

Run to get results:

```
mkdir -p ./output

docker run --rm --name my_pipeline_run \
  -v "$(pwd)/output":/output \
  my-reproducible-pipeline
```

Check your local `output` directory—you'll find the `mtcars_analysis_res` file containing the exact, reproducible result of your pipeline.

You have successfully packaged a complex, polyglot pipeline into a simple, portable Docker image. This workflow combines the best of both worlds: Nix's power for creating reproducible builds and Docker's universal standard for distributing and running applications.

7.8 Summary

Docker and Nix complement each other:

- **Docker** provides a universal runtime and distribution mechanism
- **Nix** provides bit-for-bit reproducible environment management
- **Together** they enable truly reproducible, portable data products

Key Docker concepts:

- **Images** are templates; **containers** are running instances
- Containers are ephemeral—use **volumes** for persistence
- Use **registries** (Docker Hub) to share images
- The **Rocker Project** provides ready-made R images

For reproducibility:

- Use `ubuntu:latest` + Nix rather than pinning specific OS versions
- Build a reusable `nix-base` image to share across projects
- Package `{rixpress}` pipelines for distribution

7.9 Further Reading

- Running Your R Script in Docker
- Docker & R Reproducibility
- R Docker Tutorial

8 Robust Pipelines with Monads

In the previous chapters, you learned functional programming fundamentals and how to build reproducible pipelines with `{rixpress}`. Now we'll add another layer of robustness: **monads**.

Monads might sound abstract, but they solve concrete problems involving **Logging** (tracing what each step does without cluttering your functions), **Error handling** (letting failures propagate gracefully instead of crashing), and **Composition** (keeping functions composable even when they need to do "extra" work).

By the end of this chapter, you'll know how to integrate `{chronicler}` and `talvez` into your pipelines for more robust, observable data workflows.

8.1 The Problem: Decorated Functions Don't Compose

Suppose you want your functions to provide logs. You might rewrite `sqrt()` like this:

```
my_sqrt <- function(x, log = "") {
  list(
    result = sqrt(x),
    log = c(log, paste0("Running sqrt with input ",
     ↪  x))
  )
}

my_log <- function(x, log = "") {
  list(
    result = log(x),
    log = c(log, paste0("Running log with input ",
     ↪  x))
  )
}
```

These functions now return lists with both the result and a log. But there's a problem—they don't compose:

```
# This works:
10 |> sqrt() |> log()

# This fails:
10 |> my_sqrt() |> my_log()
# Error: non-numeric argument to mathematical
 ↪  function
```

`my_log()` expects a number, but `my_sqrt()` returns a list. We've broken composition.

8.2 The Solution: Function Factories and Bind

A monad provides two things:

1. **A function factory** that decorates functions so they can provide additional output without rewriting their core logic
2. **A `bind()` function** that makes these decorated functions compose

Here's a simple function factory for logging:

```
log_it <- function(.f) {
  fstring <- deparse(substitute(.f))

  function(..., .log = NULL) {
    list(
      result = .f(...),
      log = c(.log, paste0("Running ", fstring, "
       ↪  with argument ", ...))
    )
  }
}

# Create decorated functions
l_sqrt <- log_it(sqrt)
l_log <- log_it(log)

l_sqrt(10)
#> $result
#> [1] 3.162278
#> $log
#> [1] "Running sqrt with argument 10"
```

Now we need `bind()` to make them compose:

```
bind <- function(.l, .f) {
  .f(.l$result, .log = .l$log)
}

# Now they compose!
10 |>
  l_sqrt() |>
  bind(l_log)

#> $result
#> [1] 1.151293
#> $log
#> [1] "Running sqrt with argument 10"
#> [2] "Running log with argument 3.16227766016838"
```

This pattern of a function factory plus `bind()` is the essence of a monad.

8.3 The chronicler Package

The `{chronicler}` package implements this pattern properly for R. It provides:

- `record()`: A function factory that decorates functions
- `bind_record()`: The bind operation
- Automatic logging of all operations

```
library(chronicler)

# Decorate functions
r_sqrt <- record(sqrt)
r_exp <- record(exp)
r_mean <- record(mean)

# Compose them
result <- 1:10 |>
  r_sqrt() |>
  bind_record(r_exp) |>
  bind_record(r_mean)

# View the result
result$value
#> [1] 5.187899

# View the log
read_log(result)
#> [1] "Complete log:"
#> [2] "  sqrt ran successfully"
#> [3] "  exp ran successfully"
#> [4] "  mean ran successfully"
```

8.4 The Maybe Monad: Handling Errors

Another common monad is **Maybe**, which handles computations that might fail. Instead of crashing, functions return either:

- `Just(value)` if successful
- `Nothing` if something went wrong

The `{chronicler}` package uses this under the hood:

```
r_sqrt <- record(sqrt)

# This works
r_sqrt(16)
#>   Value: Just
#> [1] 4

# This fails gracefully
r_sqrt("not a number")
#>   Value: Nothing
```

When `Nothing` is passed to a decorated function, it immediately returns `Nothing`—the error propagates through the pipeline without crashing.

8.5 Monads in Python: talvez

The same concepts exist in Python. The `talvez` package provides a Maybe monad:

```
from talvez import maybe, just

@maybe()
def parse_int(s: str) -> int:
    return int(s)

@maybe(ensure=lambda x: x != 0)
def reciprocal(n: int) -> float:
    return 1 / n
```

```
# Successful computation
result = (
    parse_int("25")
      .bind(reciprocal)
      .fmap(lambda x: x * 100)
)
print(result)             # Just(4.0)
print(result.get_or(-1)) # 4.0

# Failed computation
bad = (
    parse_int("not a number")
      .bind(reciprocal)
      .fmap(lambda x: x * 100)
)
print(bad)  # Nothing
```

The key operations are:

- `fmap(fn)`: Apply a pure function to the value inside the monad
- `bind(fn)`: Apply a function that itself returns a monad

8.6 Building Robust Pipelines

The real power of monads becomes apparent when you combine them with pipeline orchestration. Consider a typical data pipeline:

```
# Standard pipeline - fragile
raw_data |>
  basic_cleaning() |>
  recodings() |>
  filter_arrivals() |>
  make_monthly() |>
  make_plot()
```

If `recodings()` fails halfway through, the entire pipeline crashes. You get an error message, but no information about what succeeded before the failure.

Now imagine wrapping each function with `record()`:

```
library(chronicler)

# Robust pipeline with logging
r_basic_cleaning <- record(basic_cleaning)
r_recodings <- record(recodings)
r_filter_arrivals <- record(filter_arrivals)
r_make_monthly <- record(make_monthly)
r_make_plot <- record(make_plot)

result <- raw_data |>
  r_basic_cleaning() |>
  bind_record(r_recodings) |>
  bind_record(r_filter_arrivals) |>
  bind_record(r_make_monthly) |>
  bind_record(r_make_plot)

# Now you get:
# - The result (or Nothing if any step failed)
# - A complete log of which steps ran
```

```
# - Exactly where and why it failed
read_log(result)
```

This pattern transforms a fragile script into a robust, observable pipeline.

8.7 Integrating with rixpress

You can combine the power of `{chronicler}` with `{rixpress}` for even more robust pipelines. The key insight is that your `user_functions` can use `record()` internally:

```
# functions.R
library(chronicler)

# Create recorded versions of your functions
r_basic_cleaning <- record(function(data) {
  data |>
    select(contains("TIME"), contains("20")) |>
    pivot_longer(cols = contains("20"),
                 names_to = "date",
                 values_to = "passengers")
})

r_recodings <- record(function(data) {
  data |>
    mutate(tra_meas = fct_recode(tra_meas, ...)) |>
    mutate(passengers = as.numeric(passengers))
})

# Export a pipeline function that uses bind_record
```

```
process_aviation_data <- function(raw_data) {
  raw_data |>
    r_basic_cleaning() |>
    bind_record(r_recodings)
}
```

Then in your {rixpress} pipeline:

```
library(rixpress)

list(
  rxp_r_file(
    name = avia_raw,
    path = "data/avia.tsv",
    read_function = readr::read_tsv
  ),

  rxp_r(
    name = processed_data,
    expr = process_aviation_data(avia_raw),
    user_functions = "functions.R"
  ),

  # ... more steps
) |>
  rxp_populate()
```

The {rixpress} derivation caches the result while {chronicler} provides logging and error handling within the step.

8.8 Python Integration with talvez

The same pattern works in Python pipelines:

```
# functions.py
from talvez import maybe, chain, just

@maybe()
def basic_cleaning(df):
    return df.dropna().reset_index(drop=True)

@maybe()
def recodings(df):
    df['category'] =
 ↪  df['category'].map(category_mapping)
    return df

@maybe()
def filter_arrivals(df):
    return df[df['tra_meas'] == 'Arrivals']

def process_data(raw_df):
    """Returns Just(result) or Nothing with error
     ↪  handling."""
    return chain(
        just(raw_df),
        basic_cleaning,
        recodings,
        filter_arrivals
    )
```

Then in your `{rixpress}` pipeline:

```
rxp_py(
  name = processed_data,
  expr = "process_data(raw_data)",
  user_functions = "functions.py"
)
```

8.9 Summary

Monads add a layer of robustness to your pipelines:

- **chronicler** (`record()`, `bind_record()`): Logging and error handling for R
- **talvez** (`@maybe()`, `bind()`, `chain()`): Error handling for Python
- **Just/Nothing**: Graceful failure propagation without crashes
- **Composability preserved**: Even with extra capabilities, functions still compose

Combined with `{rixpress}`:

- Nix provides **hermetic, cached execution**
- Monads provide **logging and error handling**
- Together they give you **robust, reproducible, observable pipelines**

This completes our toolkit for reproducible data science: environments (Nix), functional code (FP), pipelines (rixpress), distribution (Docker), and robustness (monads).

References

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