Reproducible Polyglot Data Science

Bruno Rodrigues

2025-12-31

Table of contents

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." If you're looking for the R-specific version of 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*:

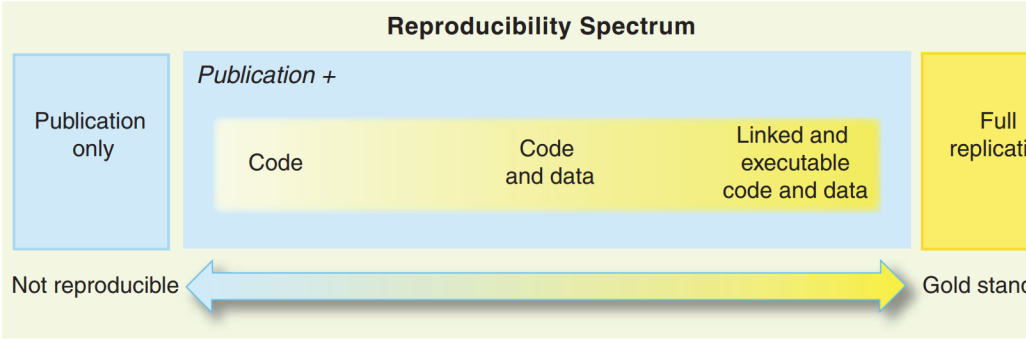

Figure 1.3: The reproducibility spectrum from Peng's 2011 paper.

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].

What makes Nix particularly useful for reproducible projects? Why not use another package manager? Wouldn't we achieve the same thing?

To answer that, let's start from the beginning. To ensure a project is reproducible, you need to deal with at least four challenges:

- Ensure the required version of your programming language (R, Python, etc.) is installed.
- Ensure the required versions of all packages are installed.

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

- 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.

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.

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.

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.

2.2 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.*

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

- All inputs to a derivation must be explicitly declared.

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

The next sections explain these three points in more detail.

2.2.1 Derivations

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

```
let
  pkgs = import (fetchTarball
   ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/2025-04
   ↪  {};
in
pkgs.stdenv.mkDerivation {
  name = "filtered_mtcars";
  buildInputs = [ pkgs.gawk ];
  dontUnpack = true;
  src = ./mtcars.csv;
  installPhase = ''
    mkdir -p $out
    awk -F',' 'NR==1 || $9=="1" { print }' $src >
 ↪  $out/filtered.csv
  '';
}
```

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` 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.

2.2.2 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.

2.2.3 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-mtcars_tail`. 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.

2.2.4 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—you can just reuse the result. Nix does the same with binary caches: every build is identified by a unique cryptographic hash of its inputs. That means a prebuilt package fetched from `cache.nixos.org` (or your own cache) 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.

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).

With these extra concepts in mind, we can now wrap up how Nix ties everything together before moving on to installing it.

2.3 In summary

Nix generates a derivation from a Nix expression through a process called instantiation. During this process, Nix resolves all inputs and computes a unique cryptographic hash from the contents of the derivation and its entire dependency graph. This ensures that even the smallest change results in a distinct derivation.

Once instantiated, the derivation is built in a hermetic environment where only explicitly declared dependencies are available. If a pre-built binary is available through the official Nix cache (or through another cache you might have added), it gets fetched instead, since building it locally would have resulted in **exactly** the same binary anyways. This makes the build entirely deterministic. After a successful build, Nix stores the output in the Nix store under a unique path determined by its hash. This process is extremely precise: even changing a comma to a pipe as a separator in a CSV file will result in a different hash, because one of the inputs changed.

The key takeaway is that Nix is a complex tool because it solves a complex problem: ensuring complete reproducibility across different environments and over time. Throughout this book, we will use the `{rix}` and `{rixpress}` (or `ryxpress` for Python) packages to make Nix more accessible, allowing you to benefit from its power without having to master all its complexities.

In the next chapter, we will learn how to install Nix, and use `{rix}` to set up our first reproducible development environments!

3 Reproducible Development Environments with rix

References

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