

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?	13
1.4.1 Using open-source tools to build a RAP is a hard requirement	14
1.4.2 There are hidden dependencies that can hinder the reproducibility of a project . .	16
1.4.3 The requirements of a RAP	17
1.5 Are there different types of reproducibility? . . .	19
References	25

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

Welcome!

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 Leanpub¹ 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}`: 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.

Welcome!

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 bio².

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

Preface

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 reimaging 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 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. 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 **[YOUR BOOK'S URL HERE]**.

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

1 Introduction

This book will not teach you about machine learning, statistics or visualisation.

The goal is to teach you a set of tools, practices and project management techniques that should make your projects easier to reproduce, replicate and retrace. These tools and techniques can be used right from the start of your project at a minimal cost, such that once you're done with the analysis, you're also done with making the project reproducible. Your projects are going to be reproducible simply because they were engineered, from the start, to be reproducible.

There are two main ideas in this book that you need to keep in mind at all times:

- DRY: Don't Repeat Yourself;
- WIT: Write IT down.

DRY WIT is not only the best type of humour, it is also the best way to write reproducible analytical pipelines.

1.1 Who is this book for?

This book is for anyone that uses raw data to build any type of output based on that raw data. This can be a simple quarterly

1 Introduction

report for example, in which the data is used for tables and graphs, or a scientific article for a peer reviewed journal or even an interactive web application. It doesn't matter, because the process is, at its core, always very similar:

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

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

1.2 What is the aim of this book?

The aim of this book is to make the process of analysing data as reliable, retraceable, and reproducible as possible, and do this by design. This means that once you're done with the analysis, you're done. You don't want to spend time, which you often don't have anyways, to rewrite or refactor an analysis and make it reproducible after the fact. We both know that this is not going to happen. Once an analysis is done, it's time to go to the next analysis. And if you need to rerun an older analysis (for example, because the data got updated), then you'll simply figure it out at that point, right? That's a problem for future you, right? Hopefully, future you will remember every quirk of your code and know which script to run at which point in the

1.2 *What is the aim of this book?*

process, which comments are outdated and can be safely ignored, what features of the data need to be checked (and when they need to be checked), and so on... You better hope future you is a more diligent worker than you!

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

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

The second reason is that an analysis is never, ever, a one-shot. Only the most simple things, like pulling out a number from some data base may be a one-shot. And even then, chances are that once you provide that number, you'll be asked to pull out a variation of that number (for example, by disaggregating by one or several variables). Or maybe you'll get asked for an update to that number in six months. So you will learn very quickly to keep that SQL query in a script somewhere to make sure that you provide a number that is consistent. But what about more complex analyses? Is keeping the script enough? Keeping the

1 Introduction

script is already a good start of course. The problem is that very often, there is no script, or not a script for each step of the analysis.

I've seen this play out many times in many different organisations. It's that time of the year again, we have to write a report. 10 people are involved, and just gathering the data is already complicated. Some get their data from Word documents attached to emails, some from a website, some from a report from another department that is a PDF... I remember a story that a senior manager at my previous job used to tell us: once, a client put out a call for a project that involved helping them setting up a PDF scraper. They periodically needed data from another department that came in PDFs. The manager asked what was, at least from our perspective, an obvious question: why can't they send you the underlying data from that PDF in a machine readable format? They had never thought to ask. So my manager went to that department, and talked to the people putting that PDF together. Their answer? "Well, we could send them the data in any format they want, but they've asked us to send the tables in a PDF format".

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

1.3 Prerequisites

You should be comfortable with the Python programming language. This book will assume that you have been using Python for some projects already, and want to improve not only your knowledge of the language itself, but also how to successfully

1.4 What actually is reproducibility?

manage complex projects. Ideally, you should know about packages, how to install them, you should have written some functions already, know about loops and have some basic knowledge of data structures like lists. While this is not a book on visualisation, we will be making some graphs using the `plotnine` package, so if you're familiar with that, that's good. If not, no worries, visualisation, data munging or data analysis is not the point of this book. Chapter 2, *Before we start* should help you gauge how easily you will be able to follow this book.

Ideally, you should also not be afraid of not using Graphical User Interfaces (GUIs). While you can follow along using an IDE like VS Code, I will not be teaching any features from any program with a GUI. This is not to make things harder than they should be (quite the contrary actually) but because interacting graphically with a program is simply not reproducible. So our aim is to write code that can be executed non-interactively by a machine. This is because one necessary condition for a workflow to be reproducible and get referred to as a RAP, is for the workflow to be able to be executed by a machine, automatically, without any human intervention. This is the second lesson of building RAPs: there should be no human intervention needed to get the outputs once the RAP is started. If you achieve this, then your workflow is likely reproducible, or can at least be made reproducible much more easily than if it requires some special manipulation by a human somewhere in the loop.

1.4 What actually is reproducibility?

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

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

1.4.1 Using open-source tools to build a RAP is a hard requirement

Open source is a hard requirement for reproducibility.

No ifs nor buts. And I'm not only talking about the code you typed for your research paper/report/analysis. I'm talking about the whole ecosystem that you used to type your code and build the workflow.

Is your code open? That's good. Or is it at least available to other people from your organisation, in a way that they could re-execute it if needed? Good.

But is it code written in a proprietary program, like STATA, SAS or MATLAB? Then your project is not reproducible. It doesn't matter if this code is well documented and written and available on a version control system (internally to your company or open to the public). This project is just not reproducible. Why?


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

1.4 What actually is reproducibility?

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

Old version of matlab

[Follow](#) 4 views (last 30 days)

 on 29 Nov 2018 STAFF MVP on 29 Nov 2018 Vote | 1 [Link](#)


Hallo after a few years we need to use again an old program written with matlab R12 6.0.0.88. We don't find the installation CD, can we buy again this old version of the program? Thanks best regard

0 Comments

[Sign in to comment.](#)

[Sign in to answer this question.](#)

Answers (1)

 STAFF MVP on 29 Nov 2018 Vote | 0 [Link](#)

Have you tried running the old program on a more recent release of MATLAB?

MATLAB 6.0 (R12) is **eighteen years old** (released in November 2000) and I think it highly unlikely you'll be able to get it working on a new operating system. The [Windows system requirements](#) lists several Windows versions on which that release was supported, the **newest** of which was Windows ME which was released in September 2000. Microsoft ended mainstream support for this OS in 2003 and ended extended support in July 2006 according to [Wikipedia](#).

0 Comments

[Sign in to comment.](#)

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

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

1 Introduction

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

And let me be clear, what I'm describing here with MATLAB could also be said for any other proprietary programs still commonly (unfortunately) used in research and in statistics (like STATA, SAS or SPSS). And even if some, or even all, of the editors of these proprietary tools provide ways to buy and run older versions of their software, my point is that the fact that you have to rely on them for this is a barrier to reproducibility, and there is no guarantee they will provide the option to purchase older versions forever. Also, who guarantees that the editors of these tools will be around forever? Or, and that's more likely, that they will keep offering a program that you install on your machine instead of shifting to a subscription based model?

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

Think this is science fiction? Google "SAS cloud" to see SAS's cloud based offering.

1.4.2 There are hidden dependencies that can hinder the reproducibility of a project

Then there's another problem: let's suppose you've written a nice, thoroughly tested and documented workflow, and made it available on Github (and let's even assume that the data is available for people to freely download, and that the paper is

1.4 What actually is reproducibility?

open access). Or, if you're working in the private sector, you did everything above as well, the only difference being that the workflow is only available to people inside the company instead of being available freely and publicly online.

Let's further assume that you've used R or Python, or any other open source programming language. Could this study/analysis be said to be reproducible? Well, if the analysis ran on a proprietary operating system, then the conclusion is: your project is not reproducible.

This is because the operating system the code runs on can also influence the outputs that your pipeline builds. There are some particularities in operating systems that may make certain things work differently. Admittedly, this is in practice rarely a problem, but it does happen¹, especially if you're working with very high precision floating point arithmetic like you would do in the financial sector for instance.

Thankfully, there is no need to change operating systems to deal with this issue, and we will learn how to use Docker to safeguard against this problem.

1.4.3 The requirements of a RAP

So where does that leave us? Basically, for something to be truly reproducible, it has to respect the following bullet points:

- Source code must obviously be available and thoroughly tested and documented (which is why we will be using Git and Github);

¹<https://github.com/numpy/numpy/issues/9187>

1 Introduction

- All the dependencies must be easy to find and install (we are going to deal with this using dependency management tools);
- To be written with an open source programming language (nocode tools like Excel are by default non-reproducible because they can't be used non-interactively, and which is why we are going to use the Python programming language);
- The project needs to be run on an open source operating system (thankfully, we can deal with this without having to install and learn to use a new operating system, thanks to Docker);
- Data and the paper/report need obviously to be accessible as well, if not publicly as is the case for research, then within your company. This means that the concept of “scripts and/or data available upon request” belongs in the trash.



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

1.5 Are there different types of reproducibility?

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

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

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

So a least worse pipeline would be one that could be run more easily on any similar machine to yours. This could be achieved by not using hardcoded absolute paths, and by providing instructions to set up the environment. For example, in the case of Python, this could be as simple as providing a `requirements.txt` file that lists the dependencies of the project, and which could be easily install using `pip`:

1 Introduction

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

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

1.5 Are there different types of reproducibility?

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

The take-away message is that counting on the language itself being stable through time as a sufficient condition for reproducibility is not enough. We have to set up the code in a way that it actually is reproducible and explicitly deal with versions of the language itself.

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

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

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

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

1 Introduction

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

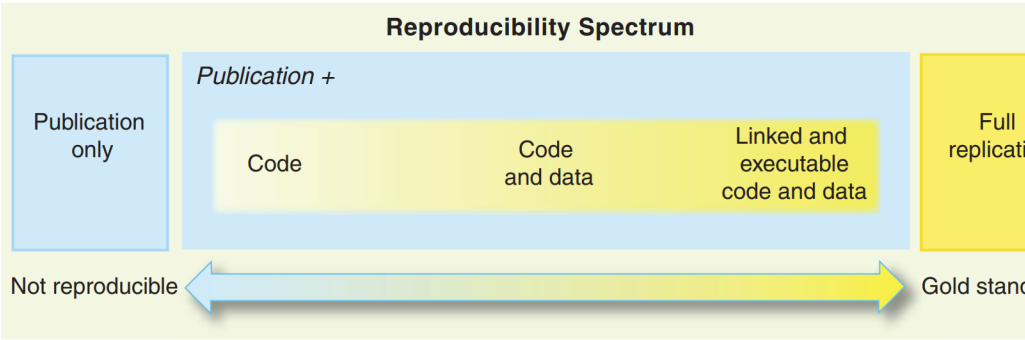


Figure 1.3: The reproducibility spectrum from Peng’s 2011 paper.

Let me just finish this introduction by discussing the last item on the previous list: hardware architecture. You see, Apple has changed the hardware architecture of their computers recently. Their new computers don’t use Intel based hardware anymore, but instead Apple’s own proprietary architecture (Apple Silicon) based on the ARM specification. And what does that mean concretely? It means that all the binary packages that were built for Intel based Apple computers cannot run on their new computers (at least not without a compatibility layer). Which means that if you have a recent Apple Silicon Macbook and need to install old packages to rerun a project (and we will learn how to do this later in the book), these need to be compiled to work on Apple Silicon first. Now I have read about a compatibility layer called Rosetta which enables to run binaries compiled for the Intel architecture on the ARM architecture, and maybe this works well with older Python and package binaries compiled for Intel architecture. Maybe, I don’t know. But my point is that you never know what might come in the future, and thus

1.5 Are there different types of reproducibility?

needing to be able to compile from source is important, because compiling from source is what requires the least amount of dependencies that are outside of your control. Relying on binaries is not future-proof (and which is again, another reason why open-source tools are a hard requirement for reproducibility).

And for you Windows users, don't think that the preceding paragraph does not concern you. I think that it is very likely that Microsoft will push in the future for OEM manufacturers to build more ARM based computers. There is already an ARM version of Windows after all, and it has been around for quite some time, and I think that Microsoft will not kill that version any time in the future. This is because ARM is much more energy efficient than other architectures, and any manufacturer can build its own ARM cpus by purchasing a license, which can be quite interesting from a business perspective. For example in the case of Apple Silicon cpus, Apple can now get exactly the cpus they want for their machines and make their software work seamlessly with it (also, further locking in their users to their hardware). I doubt that others will pass the chance to do the same.

Also, something else that might happen is that we might move towards more and more cloud based computing, but I think that this scenario is less likely than the one from before. But who knows. And in that case it is quite likely that the actual code will be running on Linux servers that will likely be ARM based because of energy and licensing costs. Here again, if you want to run your historical code, you'll have to compile old packages and R versions from source.

Ok, so this might seem all incredibly complicated. How on earth are we supposed to manage all these risks and balance the immediate need for results with the future need of rerunning an

1 Introduction

old project? And what if rerunning this old project is not even needed in the future?

This is where this book will help you. By employing the techniques discussed in this book, not only will it be very easy and quick to set up a project from the ground up that is truly reproducible, the very fact of building the project this way will also ensure that you avoid mistakes and producing results that are wrong. It will be easier and faster to iterate and improve your code, to collaborate, and ultimately to trust the results of your pipelines. So even if no one will rerun that code ever again, you will still benefit from the best practices presented in this book. Let's dive in!

References

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

