FS4R

Job Spijker e.a.

2026-01-07

Table of contents

Preface

This guide provides practical recommendations and best practices to help you write good research software, make it Findable, Accessible, Interoperable, and Reusable (FAIR). Following FAIR principles increases the visibility, reusability, and reproducibility of your software. Using these guidance you are also supporting reproducible science and collaboration within your research community.

Status

This guide is currently under development. The content is a work in progress and will be continuously improved. If you have any suggestions or comments, please open an issue on GitHub or contact the authors.

To Do

Since this is the preface and the first part of the book, it is logical place to put an annotated outline of the rest of the book. This will help to structure the book and to see what is missing.

Please note, this is an extensive list and maybe somewhat ambitious. During the development of this book we might drop a few subjects (or squeze them in other chapters) to keep the book manageable.

1. Introduction (this chapter)
    - Purpose and scope
    - Who is this book for?
2. Project Planning
    - Overview of the software/project life cycle
    - Defining objectives and requirements
    - Data Management Plan (DMP)
    - Stakeholder identification
3. Version Control & Initial Setup (versioncontrol.qmd)

- Creating a Git repository
- Essential files: README, LICENSE, .gitignore, etc.
- Collaboration and branching strategies

4. Conceptual Design

   - Developing the conceptual/data model
   - Mapping workflow steps (import, clean, transform, model, visualize, report)
   - Flow diagrams (e.g., Mermaid)

5. Data Management

   - Organizing and storing data
   - Data formats and naming conventions
   - Data sharing and archiving policies

6. Implementation & Coding Standards (goodcoding.qmd)

   - Writing clean, efficient, maintainable code
   - Applying style guides and conventions
   - Structuring codebase (modules, scripts, notebooks)

7. Documentation (documentation.qmd)

   - Project documentation (README, CHANGELOG, CONTRIBUTING)
   - Code documentation (comments, docstrings, notebooks)
   - Keeping documentation up-to-date

8. Reproducibility & Environment Management (reproducibility.qmd)

   - Definitions: reproducibility, replicability, reuse
   - Managing environments (Conda, Docker, renv)
   - Ensuring analyses can be reliably repeated (notebooks, scripts, dependencies)

9. Testing & Validation (testing.qmd)

   - Unit, regression, and integration tests
   - Validation and verification of results
   - Automated testing tools and workflows

10. Deployment, Maintenance & Collaboration

    - Sharing code and results (repositories, archives)
    - Licensing and citation
    - Persistent identifiers (DOIs)

    - Updating, maintaining, and extending the project

- Team communication and best practices

1 Introduction

Working with data and models, using to create valuable and impactful information is a complex task. It requires a combination of technical skills, domain knowledge, and an understanding of the broader context in which the data and models are used.

The effictive and responsible use of data and models is crucial for researchers. It must me clear how resulta are obtained, what the limitations are, and how they can be reproduced. This is where the Fair Software for Research (FS4R) book comes in. The knowlesge of this book is based on standing practices in data science and data software engineering,

This book is written to help you to apply these standards to your work in a hands on way. We will not explain each standard in detail but only briefly mention or link them where relevant. What we will do is to show you best practices about data science and data engineering that will help you to improve your research and make your work reproducible, transparent, reusable and reliable.

This book is writen for data scientist, data modellers, data analysts, data managers and anybody else working with data. The book is collaborative effort and anybody is welcome to contribute.

> **Note**
>
> TODO: explain how to contribute

1.1 How to use this book?

While this is a book, it is not meant to be read from cover to cover. It a is a collection of guidances and recipes that you can use when you are doing your research. You can read the book in any order or just pick the chapters that are relevant to you. Each chapter is self contained and can be read independently. However, the order of the chapters is logical for somebody who is new to data science or the way of working with data and models.

2 Summary

In summary, this book has no content whatsoever.

3 You are not alone

In the era of computational science, the roles of scientific modeller and data scientist are inseparable from that of a software developer. Off-the-shelf commercial software rarely addresses the unique needs of every specialized research model, so scientists often find themselves building or adapting code to fit their specific requirements.

Like many in the field, you may have developed your programming skills out of necessity rather than through formal education. Most academic programs offer only minimal training in programming—typically just enough for basic data analysis or simple simulations, but not for developing complex and maintainable software.

This means you are often solely responsible for creating, debugging, maintaining, and expanding your computational models. If you're fortunate, you are part of a supportive team where you can share experiences, frustrations, and discuss different modelling approaches, but many researchers face these technical challenges on their own.

As a result, researchers often end up working in a highly individualistic manner, prioritizing scientific discoveries and the advancement of knowledge over the craft of software engineering. The primary focus becomes achieving results and fulfilling research objectives, since those are the things you are hired for. While this approach can be innovative and agile, the resulting code is often difficult to maintain, scale or share. This reinforces the habit of working in isolation rather than prioritizing collaboration or best practices in software development.

Against this backdrop, what can improving software skills offer to the lone, individualistic, modeller beyond abstract "best practices"? The answer is simple: developing robust coding skills directly leads to more impactful, efficient, and personally rewarding research. Because, you are not alone!

Moving beyond an individualistic mindset and embracing a more collective approach can be truly transformative—both for you and the community. Sharing your knowledge and code, collaborating with peers, do more than contribute to the broader field; they accelerate your own learning and success. Working openly with others exposes you to new perspectives, more efficient techniques, and innovative solutions to complex problems.

Collaboration also enhances the robustness and reliability of your research. Colleagues and collaborators can spot errors, suggest improvements, and help validate your results, increasing the credibility and impact of your work. By joining forces with others, you gain access to a broader range of expertise, making it easier to tackle ambitious projects that might be overwhelming alone.

Furthermore, being part of a collaborative community opens doors to valuable networking opportunities, potential partnerships, and new projects. It fosters a culture of mutual support, where knowledge is shared and challenges are overcome collectively. Ultimately, seeing yourself as part of a larger, interconnected network not only advances your own research but also propels the progress of the entire field. In embracing collaboration, you find that the rewards—professional, intellectual, and personal—are far greater than the sum of individual efforts. If only the others would just understand your code.

But it is not all about how you benefit. Perhaps the most immediate and personal incentive for improving coding skills is the profound benefit to "your future self." Every modeller has experienced the agony of revisiting old code, only to be perplexed by cryptic variable names, tangled logic, or the dreaded "final_v2_reallyfinal.py." By adopting better practices—such as clear naming, meaningful comments, version control, and tidy organization—you transform code from a messy trail to a clear roadmap. This means that when deadlines loom, manuscripts need revision, or new questions arise, you can rapidly understand and extend your own work rather than restarting from scratch. Clean, well-documented code is a gift to yourself, sparing you hours of confusion and frustration later.

It's easy to think of coding skills as simply the ability to write functional code, but truly effective scientific modelling goes much further. Good coding practice isn't just about the code you write; it's about how you organize, track, and communicate your work. Good documentation, for example, helps others—and your future self—understand what your code does and why you made certain choices. Version control allows you to experiment freely while keeping an organized history of your changes, making it easy to revert mistakes or collaborate with others. Testing, on the other hand, ensures your model behaves as expected, giving you confidence in your results.

While writing code is an essential starting point, these practices are what transform code from a personal tool into a robust, reliable, and valuable resource for scientific progress. If you focus only on getting your code to run, you might achieve results in the short term, but investing time in these broader skills pays dividends in reproducibility, efficiency, and collaboration.

4 Reproducible Research

Reproducible research is the practice of organizing and documenting data, code, and methods so that others can independently recreate the same results from the same materials.

In recent years, the integrity and reliability of scientific research have faced growing scrutiny across disciplines. Increasingly, the scientific community recognizes that progress hinges not only on the novelty of results, but also on the transparency and reproducibility of the research underlying those results. Reproducibility—the capability for others to independently confirm results using the same data, code, and methods—is no longer optional. It is the minimum standard for reliable scientific work (Peng (2011)).

Too often, scientific findings cannot be replicated because the underlying data or computational steps are inaccessible or poorly documented. This erodes trust and stalls the advancement of knowledge. The The Turing Way Community et al. (2019) offers a practical manifesto for change, outlining steps researchers can take to embed reproducibility into their everyday practice. These principles, while simple in concept, are transformative in their effect.

A cornerstone of reproducible research is automation: using scripts and workflow management tools to carry out analyses reduces manual errors and makes the research process transparent. As described in Chapter 7 and Chapter 8, automating analysis not only streamlines work, but also enables reliable testing and validation of results.

Equally vital is thorough documentation. Documenting your choices—how data were obtained and cleaned, which methods and tools were used—ensures transparency and supports both your future self and your colleagues. This principle connects directly to Chapter 5, where practical guidance is offered for keeping project and code documentation up-to-date.

Finally, robust version control (see Chapter 6) should become a staple in every researcher's toolkit. By tracking changes to code and documentation, version control systems like Git support collaborative work and make it easier to identify and correct errors introduced during development. This practice also complements team workflows described in Deployment, Maintenance & Collaboration.

5 Documentation

TBD

6 Version Control

TBD

7 Good Coding Practices

Good code is important, it is like good writing. If you are reading a story wich is written incoherently, it takes you much longer to read and to understand, if you can understand it anyway. The problem with bad code is that it can actually works and produce the right result. And if the result counts, you might get lured into the swamp of bad code. If you, your collegue, or your future self has to make changes to the code, that swamp can turn out to be a nightmare. The code is hard to understand, hard to maintain, and hard to extent or reuse. This leads to frustration, errors, and wasted time.

It takes time to learn how to write a good, concise, and easy to understand story. If you really master it it can even become an art. The same holds for coding. So it also takes time and effort to learn how to write good code. And the best way to learn is by doing it, learn from others, and get regular feedback on your code.

This chapter is not a simple 'x steps to success' guide. It is a collection of good practices and guidelines which you can implement in your own work. Do not try to implement all at once, but pick one or two practices at a time and try how they work for you. Then, over time, your coding skills will improve. But before we start, we need to know what we aim for: what is good code?

7.1 The defintion of good code

What is good code? That is code which is easy to read and understand. A good metric to evaluate the code is the 'time till understanding': How long does it take for your collegue to read through your code and uderstand what is does (**boswell_Foucher?**). And to understand code it means:

- that you can make changes
- that you can spot errors or inconsistencies

- that you can extend the code, or reuse the code, to new use cases,
- and that you understand how this code interacts with other code

Writing better code means that you are minimizing the metric 'time till understanding'. When it is good enough is a matter of definition. In theory this means that the metric is equal to time needed for a single, top to bottom, read through of the code. In practice this is not

always possible, because some code is inherently complex. Actually measuring the time till understanding is also not recommended, because it will very a lot between different people with differen expertise levels. It is, however, a good goal to keep in mind. Both when writing code but also when reviewing code.

7.2 Good coding practices

7.2.1 Keep it small

A book is not written as a single huge chapter, but is broken down in smaller chapters and sections. The same holds for code. If you break down your code in smaller chunks, scripts of files, it becomes much easier to read and understand. Especially if these chunks of code can allso be related to the conceptual model of your workflow (see chapter about conceptual models). If the reader known the context of the chunk of code it is much easier to understand what it does.

The same goes for sentences in writing. Long sentences are hard to understand. Short sentences are easier to read. So do't try to be superefficient and put as much operations in one single line but try to split it up

EXAMPLE code

EXAMPLE R code, pipe operators

7.2.2 formating and style guidens

The easiest part in writing good code is apllying a style guide to your code. A style guide is a set of rules how to format your code. It includes things like indentation, spacing, line breaks, naming conventions, and other aspects that contribute to the overall readability of the code.

Examples of generic style guides are 'tidystyle' [ref] for R and 'PEP 8' [ref] for Python. These guides are often extended with local style guides wich contains conventions like file naming, folder structures, and other intitute spefific rules.

For most generic style guides, linters are avialable. These are tools which can be integrated in your code editor of EDI and check automagically your code against the style guide. It is like an autmoated spelling check in a text editor. Most linters can also autoformat your code, so you do not have to worry about the details of the style guide yourself.

TODO: give examples/refs Example of linters:

- R: lintr, styler
- Python: pylint, black, flake8

Exampe of editor settings:

- R: Rstudio settings for lintr and styler
- Python: VSCode settings for pylint and black

Example, formatted / unformatted code

Working with a style guide and a linter can be hard in the beginning. Especially if you are allready programming for some time using your own code style. You have to 'unlearn' your old habbits and get used to the new style. While this can be frustrating in the beginning, it will pay off later because you will be producing more consistent code. If you and your colleagues use the same style, it also means that it is easier to read eachother code. Reducing the 'time till understanding'

7.2.3 Naming functions and variables

It is important to use meaningful names for variables and functions because it makes your code easier to read and understand, both for yourself and for others who may work with your code in the future. Clear names describe what a variable represents or what a function does, which helps prevent confusion and reduces the chances of making mistakes. Good naming also makes it easier to maintain and update your code over time.

TODO: Something about scope (see quote)

TODO: See 'art of readable code, chap 2'

7.2.4 Keep it simple

Simple and straightforwarder code

DOnt; make the reader think too musch

7.2.5 Don't repeat yourself (DRY)

write functions document function (use genAI)

7.2.6 Single responsibility

Each function should do one thing and do it well

7.2.7 Comments

See chapter about documentation

7.2.8 Error handling

Check inputs and throw error when needed catch errors, not unexpected results

7.2.9 refactoring

not without a test

7.2.10 Reviewing

how to review (should this be a separate chapter?)

8 Testing and Validation

TBD

References

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27. https://doi.org/10.1126/science.1213847.

The Turing Way Community, The Turing Way, Becky Arnold, Louise Bowler, Sarah Gibson, Patricia Herterich, Rosie Higman, Anna Krystalli, Alexander Morley, Martin O'Reilly, and Kirstie Whitaker. 2019. "The Turing Way: A Handbook for Reproducible Data Science." Zenodo. https://doi.org/10.5281/ZENODO.3233986.