

UA Repository

Publishing and Archiving

Item Type	Book chapter
Authors	Plomp, Esther
Citation	Plomp, E. (2025). Publishing and archiving. In: Cooper, N., & Hsing, P.-Y. (Eds.). Guide to Reproducible Code. British Ecological Society. https://doi.org/10.5281/zenodo.16421733
DOI	https://doi.org/10.5281/zenodo.16421733
Rights	Attribution-ShareAlike 4.0 International
Download date	2026-03-12 01:47:16
Item License	http://creativecommons.org/licenses/by-sa/4.0/
Link to Item	https://hdl.handle.net/20.500.14473/1824



.....
Better
Science
Guides

Reproducible code

Through science we can



Contents

Introduction	3
A simple reproducible project workflow	6
Organising projects for reproducibility	7
Programming	13
Code review	29
Reproducible notebooks	35
Version control	41
Publishing and archiving	49
Further resources	55
Acknowledgements	55



Editors

Natalie Cooper, Natural History Museum, London, UK.
Pen-Yuan Hsing, University of Bristol, UK.

Contributors (2025)

Batool Almarzouq, University of Liverpool, UK.
Selina Baldauf, Freie Universität Berlin, Germany.
Nilanjan Chatterjee, Senckenberg Biodiversity and Climate Research Centre, Germany.
Esther Plomp, University of Aruba, Aruba.
Tanya Strydom, University of Sheffield, UK.
Elina Takola, Helmholtz-Zentrum für Umweltforschung, Germany.
Zuzanna Zagrodzka, University of Sheffield and University of Manchester, UK.

Original contributors (2017)

Mike Croucher, Laura Graham, Tamora James, Anna Krystalli, François Michonneau.

Copyright © British Ecological Society and authors, 2025



This Guide is published under the Creative Commons
Attribution-ShareAlike 4.0 International license (CC BY-SA 4.0).
<https://creativecommons.org/licenses/by-sa/4.0/>

British Ecological Society

britishecologicalsociety.org

DOI (print version):

<https://doi.org/10.5281/zenodo.16421732>

DOI (Quarto notebook):

<https://doi.org/10.5281/zenodo.17294767>

GitHub repository:

<https://github.com/BES-Guide/reproducible-code>

Part of the
**BES Better
Science
Guides**



Introduction

Natalie Cooper, Natural History Museum, London, UK

Pen-Yuan Hsing, University of Bristol, UK

The way we do science is changing. Datasets are getting bigger, analyses are getting more complex and governments, funding agencies and the scientific method itself demand more transparency and accountability in research. One way to deal with these changes is to make our research more reproducible, especially our code. Here, reproducibility means being able to run the same analysis (e.g. code) with the same data and get the same results. Although many of us now write code to perform our analyses, it is often not very reproducible. We have all come back to a piece of work we have not looked at for a while and had no idea what our code was doing or which of the many “final_analysis” scripts truly was the final analysis! Unfortunately, the number of tools for reproducibility and all the jargon can leave new users feeling overwhelmed, with no idea how to start making their code more reproducible. So, we have put together this guide to help. This guide to **Reproducible Code** covers the basic tools and information you need to start making your code more reproducible. Most examples are in R and Python, with a few in Julia, but the tips should apply to any programming language. Doing everything described here all at once can be hard, especially if this is your first attempt at making your code more reproducible. But do not be discouraged. Instead, challenge yourself to add just one more aspect to each of your projects. Remember, partially reproducible research is much better than completely non-reproducible research.

Open science and FAIR research software

Writing and sharing reproducible code is a part of a wider set of good practices referred to as open science. Open science practices ensure that the processes and outputs of science are shared so that anyone can use, study, change or build upon them and continue to share what they've learned with others. Just as we build upon past knowledge to conduct our research, we have a responsibility to the scientific process to share what we learn with those who come after us. These ideals have been formalised in the [UNESCO Recommendation on Open Science](#), ratified in 2021.

Introduction

To help with this, there is a set of principles called **FAIR** (**F**indable, **A**ccessible, **I**nteroperable and **R**eusable) for making research outputs, such as software code, more transparent and reproducible. The FAIR principles for Research Software¹ state that:

1. Software and its associated metadata is **Findable** for both humans and machines. Software should have a persistent, long-lasting reference such as a digital object identifier (DOI). Different versions of the software should be assigned distinct identifiers.
2. Software and its metadata is **Accessible** via an access protocol. Accessible is not always the same as “open”: the software can be shared under restricted access with a clear protocol in place that could provide access to it. The metadata should always be accessible, even when the software is no longer available.
3. Software is **Interoperable** with other software by exchanging data and/or metadata and/or through interaction via application programming interfaces (APIs), ideally using domain-relevant community standards. Interoperable software is shared in open formats that can be opened without proprietary software. Software should also include references to other research objects.
4. Software is both usable (can be executed) and **Reusable** (can be understood, modified, built upon, or incorporated into other software). For this, the software is well documented (using for example a README) and assigned a license.

It is important to think about the FAIR principles throughout your project. By applying what you learn in this guide, you will make your code not only more open and reproducible, but more FAIR as well! You can also check out the [five recommendations for FAIR software](#), or use [Howfairis](#) to assess and improve research software’s adherence to the FAIR principles.

1 Chue Hong, N. P., et al. (2022). FAIR Principles for Research Software (FAIR4RS Principles) (1.0). Zenodo. <https://doi.org/10.15497/RDA00068>

Introduction

Invisible labour. Scientific research is often a team effort, but the specialised labour behind computer programming (and good data management) is often invisible, uncosted and unpaid. We stress the importance of appreciating the scale of work needed to write, manage, archive and share reproducible code and the need to plan and budget for this labour when developing research projects. There is now increasing recognition of Research Software Engineering (RSE) in academic research. For more information, check out the work by the [Software Sustainability Institute](#) and the [Society of Research Software Engineering](#).

Generative “AI” tools and programming. Generative “AI” tools are purported to help generate, explain, comment, translate, debug, optimise and test code². Any use of generative “AI” should be **transparent, accountable and acknowledged**. Check the editorial policies of journals before submission to ensure you are using the most up-to-date guidance (e.g. the British Ecological Society (BES) [Editorial Policies](#)).

We avoid directly recommending these tools in this guide for the following reasons:

1. The appearance of intelligence of these tools risks causing “illusions of understanding”³.
2. They reduce reproducibility. Large language models (LLMs) like ChatGPT are proprietary black boxes designed with a certain degree of inbuilt randomness, which makes it impossible to reproduce their outputs.
3. They can have massive environmental costs (including energy and water consumption).
4. The obscurity of their source and function can lead to the exploitation of hidden workers, deepening the global divide and displacing (not replacing) labour and preventing proper attribution of credit.

2 Cooper, N., et al. (2024). Harnessing large language models for coding, teaching and inclusion to empower research in ecology and evolution. *Methods in Ecology and Evolution*, 15, 1757–1763. <https://doi.org/10.1111/2041-210X.14325>

3 Messeri, L., et al. (2024). Artificial intelligence and illusions of understanding in scientific research. *Nature*, 627, 49–58. <https://doi.org/10.1038/s41586-024-07146-0>

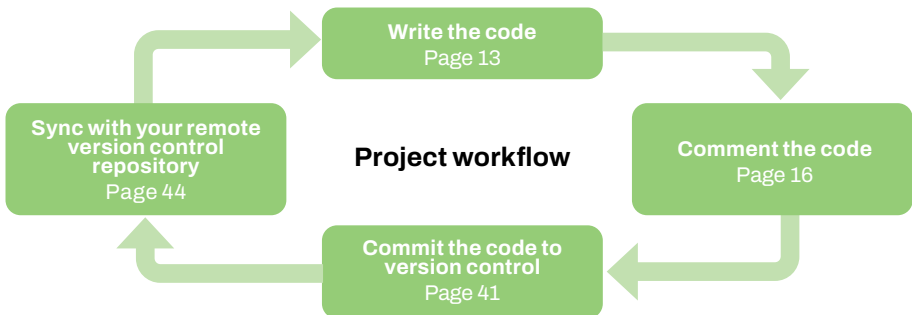
A simple reproducible project workflow

Before you start

- Think about the FAIR principles. Where will you publish/archive the code? How can you ensure your work is FAIR? Page 4
- Choose a project folder structure. Page 10
- Choose a file naming system. Page 12
- Choose a coding style and naming conventions. Page 13
- Install and set up your version control software (e.g. Git). Page 41
- OPTIONAL Set up an online version control account. Page 44

First steps

- Create a project folder and subfolders. Page 10
- Add a README. Page 10
- Add a LICENSE. Page 10
- Create a version control repository for the project. Page 41
- OPTIONAL Set up an online version control repository for your project. Page 44
- OPTIONAL Create a reproducible environment for your project. Page 26
- OPTIONAL Create Quarto notebooks for each set of analyses. Page 35



Code review Page 29

Preparing for publication/archiving

- Record the versions of software and packages you used (if you didn't already set up a reproducible environment). Page 25
- Update the README to include the full project work low. Page 50
- Check that your documentation makes sense and add more if needed. Page 51
- Make any private repositories public
- Archive your code to get a DOI. Page 49
- Don't forget to archive and document your data too! See the BES' Better Science Guide on [Data Management](#)⁴
- Ensure that the whole project is FAIR. Page 51

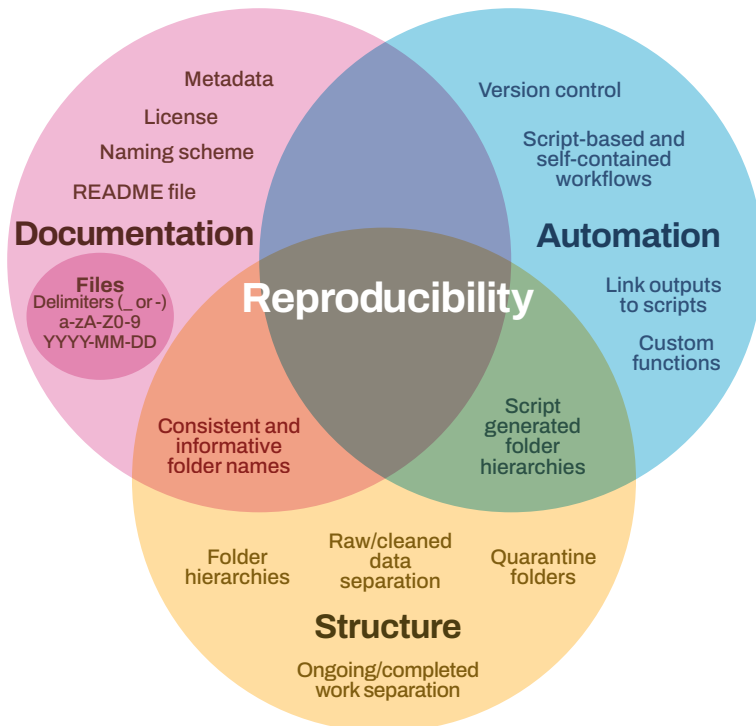
4 Data Management Guide, (2018), Better Science Guides, British Ecological Society.

Organising projects for reproducibility

Elina Takola, Helmholtz-Zentrum für Umweltforschung, Germany

The fundamental idea behind a robust reproducible analysis is a clean repeatable script-based workflow (the sequence of tasks from the start to the end of a project) that links raw data through to clean data and to final analysis outputs. Here, we define raw data as information collected through observations, experiments, or generated by instruments or devices. Raw data refers to materials that cannot be recreated once lost (e.g., worksheets, notes, lab notebooks). Most analyses will be re-run many times before they are finished (and perhaps again throughout the review process) so the smoother and more automated the workflow, the easier, faster and more robust the process of repeating will be. In other words, you should build a reproducibility ecosystem (Figure 1). These principles apply to whatever programming language or analytical tools you are using.

Figure 1: **The reproducibility ecosystem.**



Organising projects for reproducibility

Principles of a good analysis workflow

- Start your analysis from copies of your raw data.
- Clean, merge, transform (etc.) data in scripts (not manually). Raw data can be used as an input for the analyses, but the output should be saved separately and not replace the raw data.
- Split your workflow (scripts) into logical thematic units. For example, you might separate your code into scripts that (i) load, merge and clean data, (ii) analyse data and (iii) produce outputs like figures and tables.
- Eliminate code duplication by packaging useful code into custom functions (see **Programming**). Add comments for more clarity; explain expected inputs and outputs of formulas, what they do and why.
- Document your code and data as comments in your scripts or by producing separate documentation (see **Programming and Reproducible notebooks**).
- Keep any intermediary outputs generated by your workflow separate from raw data.
- Decide on an appropriate repository or platform to publish and/or archive your code prior to the analysis (See **Publishing and archiving**).
- Use structured folders (for example R Projects, Workspaces in RStudio or R packages such as workflowR) to keep things organised.

Organising and documenting workflows

The simplest and most effective way of documenting your workflow – its inputs and outputs – is through good file system organisation and informative, consistent naming of analysis materials. The name and location of files should be as informative as possible on what a file contains, why it exists and how it relates to other files in the project. These principles extend to all files in your project and are intimately linked to good research data management, such as described in the BES Better Science Guide on [Data Management](#)⁴

4 Data Management Guide, (2018), Better Science Guides, British Ecological Society.

Organising projects for reproducibility

What's the difference between a folder and a directory? Nothing! We can use the words *folder* and *directory* interchangeably as they mean the same thing. In this guide we use the word *folder* when talking about organising files and *directory* when talking about programming. Note that R and Python always refer to folders as directories. But don't let this confuse you!

File system structure

It's best to keep all files related to a specific project within a single folder. Integrated development environments (IDEs) such as RStudio and VS Code, offer a great way to keep the workflows self-contained and portable. IDEs can use file paths to data sources or scripts and remain valid even when projects are transferred between computers or shared with collaborators (see **Programming**). An exception can be code associated with numerous projects that may live in its own folder (you could also consider making this into a package; see [R Packages \(2e\)](#) for advice on making packages in R).

There is no single best way to organise a file system. The key is to make sure the structure of folders and location of files is **consistent, informative, clearly described and works for you**.

Here's an example of a basic project folder structure:

- The **data** folder contains all input data (and metadata) used in the analysis. This folder should include the raw data (as original files or in a sub-folder named **raw-data**). You might also have a **cleaned-data** folder.
- The **manuscript** or **deliverable** folder contains the deliverable (optional).
- The **figs** folder contains figures generated by the analysis.
- The **output** folder contains any type of intermediate or output files.
- The **analyses** folder contains analysis scripts.
- The **functions** folder contains functions with function definitions.
- The **reports** folder contains files that document the analysis or report on results (see **Reproducible notebooks**).

Organising projects for reproducibility

Folder structure

- Be consistent. It is important that once you have decided on a naming scheme for folders, you stick to it. If you can, agree on a naming scheme at the start of your research project. Being consistent may be more important than following a particular naming scheme.
- Structure folders hierarchically. Start with a limited number of folders for the broader topics and create more specific folders within these, as and when they are required.
- Include a README file to describe the project and provide some basic orientation around project files. Good naming of files should take care of the rest. Remember to describe the naming scheme in the README file (see **Publishing and archiving**).
- Include an appropriate license (see **Publishing and archiving**).
- Quarantine inputs and materials given to you by others (i.e. keep it in a separate folder). Rename and repurpose into your own file system, recording what has been done. Ensure provenance of inputs is tracked.
- Keep track of ideas, discussions and decisions about analysis. An **info** folder where you store important emails or idea drafts could work. Otherwise, many online version control (see **Version control**) hosting services (such as GitHub) have issues management features which provide a lot of useful functionality for this purpose.
- Separate ongoing and completed work. Keep older documents separately from current ones.

RStudio versus Positron. [Positron](#) is a new IDE designed for use with both R and Python. It is made by the company Posit which also distributes RStudio. At the time of writing, Positron is still in beta testing so we don't refer to it in the guide, but it is possible that many RStudio users will switch to Positron in future. Posit has stressed that maintenance and development of RStudio will continue, but if you regularly use R and Python you might want to check Positron out.

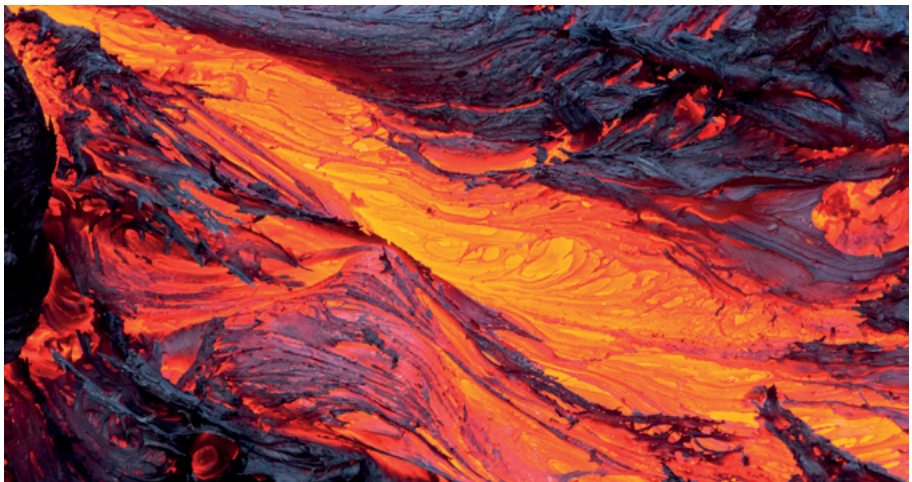
Organising projects for reproducibility

Informative, consistent naming

The rules of effective naming extend to all files, folders and even objects in your analysis (see **Programming**). Naming conventions help make the contents and relationships among elements of your analysis understandable, searchable and organised. For example, informative, consistent naming can allow you to easily find data generated during a specific period, data generated by a particular script, analysis outputs generated using specific values of a parameter and objects generated by a specific function.

Examples of bad vs better file names

Bad	Better
<code>01.R</code>	<code>01_download-data.R</code>
<code>abc.py</code>	<code>02_clean_data_functions.py</code>
<code>fig1.png</code>	<code>fig1_scatterplot-bodymass-v-brainmass.png</code>
<code>IUCN metadata.txt</code>	<code>2016-12-01_IUCN_reptile_shapefile_metadata.txt</code>



Organising projects for reproducibility

Key principles of good file naming

Good file names should:

1. Be machine readable

- Avoid spaces, punctuation, accented characters and case sensitivity. More specifically, stick to “a-zA-Z0-9_” characters.
- Use periods/full stops for file type only (i.e. **.csv**).
- Use delimiters consistently, i.e. “_” to separate metadata to be extracted as strings later on and “-” instead of spaces or vice versa but don’t mix. This makes names regular expression-friendly (see https://simple.wikipedia.org/wiki/Regular_expression if you don’t know what we mean by this), i.e. easy to match and search programmatically and easy to analyse. Note that Python cannot use “-” so you need to use “_” for file names.

2. Be human readable

- Ensure file names include informative descriptions of their contents.
- Adapt the concept of the **slug** to link outputs with the scripts in which they are generated. A slug in this context is something short that you could stick on to any file name to make it clear what analysis/paper/report it refers to, e.g. **bird-beaks_analysis.R** or **bird-beaks_results.csv**, where **bird-beaks** is the slug.

3. Work well with default ordering

- Starting file names with a number helps.
- For data, this might be a date allowing chronological ordering.
- Make sure to use **YYYY-MM-DD** date format (year-month-date, which follows the ISO 8601 standard⁵; to avoid confusion between differing local dating conventions and to ensure files are sorted chronologically.
- For scripts, you could use a number indicating the position of the scripts in the analysis sequence e.g. **01_download-data.R**
- Make sure you left pad single digit numbers with a zero or you’ll end up with this:

```
10_final-fig-for-publication.png
1_figure1.png
2_figure2.png
```

5 https://en.wikipedia.org/wiki/ISO_8601 accessed 15th August 2025

Programming

Selina Baldauf, Freie Universität Berlin, Germany

Tanya Strydom, University of Sheffield, UK

The main goal of adopting good programming practices is to make your code readable, maintainable and reproducible. Additionally, good programming practices are crucial in collaborative projects to work efficiently and seamlessly with others.



The mindset of good practice programming

While writing code, imagine how someone else (or future you) will see the project for the first time. Will they be able to understand and use it? The goal of writing reproducible code is to ensure that the answer to this question is “yes.” You can practically test this by sending your project to a colleague and asking them to try to understand and run the code.

There are many general and language-specific guidelines and tips to write readable, maintainable and reproducible code. We list the most essential ones below.

Structured workflow and readable code

Have a consistent programming style

Consistency improves readability and maintainability. It helps others to quickly understand your project’s logic and workflows.

Follow a style guide

Every programming language has a style guide for things like indentation, spacing and naming conventions for variables and functions. Check the style guide for the languages you use to get an overview.

There are tools to help you enforce a consistent style across a codebase:

- **Formatters:** Can auto-format your script to eliminate inconsistencies.
- **Linters:** Can analyse your code for errors, defects and stylistic issues and list areas for improvement.

Programming



Using formatters and linters

The details of this depend on your combination of IDE and programming language. Just search for the right formatters and linters for your setup.

R and RStudio

- **Auto-format:** Open the command palette (Ctrl/Cmd + Shift + P) and search for “format”. Use “Reformat Current Document” to auto-format your code. Toggle “Reformat documents on save” for convenience. You can also choose your own “Code formatter”, but the default option “Styler”, which applies the tidyverse style guide, is already a very good option.
- **Lint:** Install the `lintr` package and search for “lint” options in the command palette. Use “Lint current file” or “Lint current package” to list style problems.
- **Air:** A new package called `Air` is currently in development⁶ as a code formatter and language server.

Python and VS Code

- **Auto-format:** Open the command palette (Ctrl/Cmd + Shift + P) and search for “Format Document”. You can also enable “Format on Save” in VS Code settings and select a default formatter in the VS Code settings.
- **Lint:** Install a linter like `pylint` or `flake8` and configure it in VS Code settings.

Julia and VS Code

- **Auto-format:** Open the command palette (Ctrl/Cmd + Shift + P) and search for “Format Document”. You can also enable “Format on Save” in VS Code settings and select a default formatter in the VS Code settings. `JuliaFormatter` is packaged within the Julia extension and has ‘sane’ defaults, but a user can also specify their own style configurations.
- **Lint:** The Julia extension is by default statically linted; you can modify this behavior in your workplace settings.

6 <https://posit-dev.github.io/air/> accessed 15th August 2025

Programming

Have a naming convention

Just like when naming your files (see **Organising projects for reproducibility**), use clear and descriptive names for files, variables, functions and modules. The goal of a good naming convention is that it is immediately clear to the reader what is behind any file or object. For objects in your codebase, you can follow these tips:

- **Concise and descriptive:** Variable names are usually nouns and function names are verbs.
- **Avoid conflicts:** Don't use names of existing variables or functions unless you are intentionally extending or overriding them (e.g. when developing an R package or a new method in Julia).
- **Use consistent capitalisation rules:** Each language community has a preferred naming style (e.g., **snake_case** for Python and R, **lowerCamelCase** for JavaScript).
- **Develop rules and document them:** You can develop your own conventions where it is useful. This can include when and how to use abbreviations or rules on how you name your functions (e.g. you might want to prefix all your helper functions with `zzz_` to mark them as helper functions).



Examples of good and bad object names

Bad names do not reveal what is behind the variable/function and could be misleading.

`temp, datal, data_function, my_function`

Good names are human-readable and tell the user what is behind the objects.

`temperature_readings, user_data, read_data, run_binomial_model`

Programming

It is good practice to establish and follow a naming convention throughout a project. It also helps to document your conventions and naming logic in the README file of your project. This way, it is easy for collaborators to read and understand your code but also to contribute using the same style.

Comment your code

An easy win for making code more readable and reproducible is the liberal and effective use of comments to provide context for human readers, which are ignored by the computer during execution of code. One good principle to adhere to is to comment on the ‘why’ rather than the ‘what’. The code itself tells the reader what is being done; it is far more important to document the reasoning behind a particular section of code.

You can use inline comments for short explanations or block comments that span multiple lines to summarise sections of code or provide detailed explanations. Although different languages have different ways of denoting a comment Julia, Python and R all use a `#` at the start of a line to denote that it is a comment as opposed to code.

Don't forget to update your comments as your code evolves to avoid outdated or misleading information.

Structure your scripts

Structure your scripts in a consistent and logical way so that readers can orient themselves easily in your codebase.

Here are some things to consider:

- **Split long scripts:** Make scripts do just one thing. If needed, you can import multiple scripts that you need for your analysis (e.g. using source in R, import in Python, include in Julia).
- **Use a standardised header:** Include essential information like the purpose, authors, contact information, license, etc. of the script.
- **Initialise at the top:** Load all libraries, define global variables and paths, and read all data in one block at the top instead of throughout the script.
- **Use section headers:** Guide readers through your scripts with section headers. In many IDEs you can navigate these sections using a script outline or collapse different sections.

Programming

- **Create a script template:** Create a new script from a pre-structured template where you can fill out the relevant information. In RStudio, you can check out code snippets that allow you to easily load template code and script sections in your scripts.⁷



Example structure of R script for data analysis

```
# Purpose: Analyze climate data
# Author: Jane Doe, John Doe
# License: GPL-3.0-or-later
# Contact: jane.doe@email.com
# Date: 2025-02-20

# Load libraries -----
library(tidyverse)

# Define global variables -----
rain_data_path <- "data/temperature_readings.csv"
temp_data_path <- "data/rainfall_readings.csv"

# Load data -----
temperature_data <- read_csv(temp_data_path)
rainfall_data <- read_csv(rain_data_path)

# Data processing -----

# Analysis -----
# Output results -----
```

⁷ <https://support.posit.co/hc/en-us/articles/204463668-Code-Snippets-in-the-RStudio-IDE>
accessed 15th August 2025

Programming

Modular and functional code

One of the core principles in software development is DRY (*Don't Repeat Yourself*), reduce any repetitive patterns or duplicates in your code in favour of creating modular and referenceable code. Although it may seem simpler to just copy and paste the same code over again when performing repetitive tasks, it means that every time you need to change something (or fix an error), you need to change it in every place the code has been copied to. Functions are a simple way to avoid this, as they allow you to break your code into modules, which allows you to repeat the same task in a standardised (and documented) manner. DRY functions are designed to execute a specific task and ensure a data analysis is correct. See **Defensive programming** for more information.

Writing your first function

Documentation

When documenting your code, never assume that the reader knows the basics of what is going on; strive to explain things to a layperson. Document how a function works, what it does and how to use it. It is useful to think about creating two 'levels' of documentation. The first level is documentation that allows developers/collaborators to understand *what* the code does (this you can do with comments inside the function; see **Comment your code**). The second level is documentation for those who will use the function and need to know *how* to use it. Python and Julia allow you to add 'docstrings' directly to a function but external tools such as [doxygen](#) or [roxygen2](#) can create more complex documentation. Typically function descriptions are 'exported' with a function, act as metadata and are searchable; comments however are not and have to be viewed by looking at the original source code. Additionally, it may be useful to provide higher-level documentation as to how the functions integrate and work together (e.g. using a README file).

Keeping things modular

A function should perform a single action, not rely on objects from outside the function and not change objects outside the function. So, if a function does too many jobs, split it. This is useful to keep in mind in instances when you have similar analyses that share 90% of the same code. Here, it makes sense to write a function that does the 90% and keep the 10% difference external to the function.

Programming

Have a consistent naming convention

Consistency is key to making your work easier for others to understand and follow. When introducing functions into your workflow, make sure that you are consistent with how you name and describe them. Because functions perform an action on an object, having a combination of verb and noun in the name makes sense. Having a consistent design pattern makes your code easier to understand and serves as a template for further development.



Example

Let's say we have two functions: one initialises an object as a number and the other as a character. Bad practice would be to name them as follows:

- `int_char()`
- `numInit()`

Instead, opt for consistency in representing the 'initialise' action and the verb-noun order:

- `init_num()`
- `init_char()`

Note that there is no right and wrong in terms of the words, order, or case used; just make sure that it is consistent and that it is clearly documented.

Defensive programming

Defensive programming is all about anticipating errors and writing robust code. The aim is to ensure that when your code fails, it does so with well-defined errors and rests on the idea that we expect our code to fail. By applying a defensive programming philosophy (and adding checks and tests into the code), you can find unexpected behaviour sooner. Although this initially means more work, it will make debugging the code a lot easier later.

Programming

Checking the behaviour of a function

One aspect of a function you can check is that it is behaving as expected. This can include ensuring the data types are correct (e.g., is the input a number and not a character?), or testing the boundaries of the data (e.g., asserting that two dataframes are the same dimensions or contain the same variables). Since a function only does what is specified, it is important to specify what it should not do. It is also useful to include a defined error message should those checks fail, as it makes it easier to try and correct the error.



Writing checks and good error messages

Knowing that a check fails is already a good starting point but writing an error message that explains how the check is failing is even better. If the error message has additional information, it will probably give us an idea as to what is happening that is resulting in the unexpected behaviour.

For example, let's say a function is designed to count items by summing a vector. The input data should be a vector of integers because it is a count; however, it is also possible to sum a vector of floats. This means that if you were to input a vector of floats, the function would still be able to sum the numbers; however, this is not the specified job of the function. Adding a check that asserts that the input vector is an integer is a way to prevent unintended misuse of a function.

The logical check would be to see if `typeof(input_vector) == Integer` and then throw an error if this clause is not met. Although having an error message of "input data is not of type integer" is already informative, it is useful to add some additional info, such as exactly what the input data type is e.g., "input data is not of type integer but rather type float."

Testing the output of a function

By testing your code, you can catch edge cases and ensure that functions are working as intended and expected, even when users are using functions in unexpected ways. While the idea of developing tests may feel excessive when starting out with programming, it is valuable to be aware of these principles as they provide a conceptual basis from which you can develop code that meets the expectations associated with conducting 'good' science.

Programming

Unit tests

Unit testing focuses on testing individual codebase functions to ensure that they are doing what they should be and meet the specified requirements in a formalised and automated manner. At a high level, the aim of unit tests is to make sure that the underlying maths/logic of your function is correct. This can be done by inputting a value into the function for which you know what the output is and testing if the output that the function gives you is the same.



Writing and running unit tests

Most programming languages have packages that will help with executing a test run. Usually, this involves creating a separate directory where you can write your tests as well as where the testing workflows are hosted. Tests are typically run in a new language process, where the package itself and any test-specific dependencies are made available.

R

The `testthat` package⁸ is the commonly used testing framework and visually shows a pass, fail, or error for your tests. It easily integrates in your existing workflow, allowing for informal testing or the building of more 'complex' test suites.

Python

It is possible to write basic tests using `assert` to test if a statement evaluates to TRUE. For writing more complex tests the `unittest` module provides the flexibility to write more nuanced tests (assertions).

Julia

Julia allows you to write basic tests using the `@test` macro and will test that the expression evaluates to TRUE. The `Pkg.jl` has a framework for building testing suites that are run when compiling a project or package.

8 <https://testthat.r-lib.org/> accessed 15th August 2025

Programming

Integration tests

Integration tests are more about ensuring that the parts fit into the whole. So, going back to the data analysis example, you want to make sure that the output from your data cleaning function can seamlessly act as the input for our data analysis function. Alternatively, you might want to run integration tests when you are introducing new features (functions) to your project and need to ensure that these do not break or alter the behaviour of your existing workflow.

Test-driven development

Test-driven development (TDD) is an approach to software development whereby tests are written before the code to identify the desired behaviour of the system. You write a small test that defines the desired functionality, write the minimum code necessary to pass that test and refactor the code to improve structure and performance. This ensures the reliability of your code by predefining the parameters and expected outputs before you even start programming up the project.

Debugging and logging

Debugging is the process of finding and resolving errors in code. Code that has well-thought-out checks and error messages should be easy to debug, as problems are already identified and isolated. Creating logs is a comprehensive way to document the behaviour of the entire workflow. Logs are usually created by an automated workflow that runs through and records events or messages (that you have specified) as it goes. This allows you to diagnose and troubleshoot issues. Log messages can also give information on the state of the workflow. Unless you are developing extremely complex workflows or packages, it might make sense to only log errors to aid in the debugging process.

Programming



Debugging with an IDE

Generally speaking the IDE (e.g. RStudio⁹ or VS Code¹⁰) you choose to use will have some form of a ‘debug mode’ that will allow you to run the code until a specified breakpoint (the point where you suspect the problem is arising) and look at and/or walk through the code, step-by-step at that point.

Reproducible code

To ensure your code is reproducible, document the exact versions of all packages, libraries, software and potentially your operating system and hardware, alongside the code and data. Below are some basic tips to ensure others can run your code and obtain the same results.



Test reproducibility

If you are unsure whether your project is reproducible, send it to a colleague or test it on a different machine.

Write portable code

To improve the portability of your code, avoid absolute paths and use relative paths instead, ensuring that the script is run from the project root folder.

```
# Absolute path: Exists only on your machine  
absolute_path <- "C:/Users/my_name/project_folder/data/species_dat.csv"  
  
# Relative path: Exists within the project  
relative_path <- "data/species_dat.csv"
```

You can immediately see the problem with absolute paths is that they only exist on one machine, while relative paths exist within the project no matter how the machine's folders are organised.

9 <https://docs.posit.co/ide/user/ide/guide/code/debugging.html> accessed 15th August 2025

10 <https://code.visualstudio.com/docs/debugtest/debugging> accessed 15th August 2025

Programming



Avoid setwd() in R

Use RStudio projects to automatically set the working directory to the project directory. Use the [here](#) package¹¹ to construct paths relative to the project root:

```
# A relative path built with the here package  
project_path <- here::here("data", "species_dat.csv")
```

So, if your project's root folder is called "fish-jaws" and it lives on your Documents, the output for this would look something like: **Documents/fish-jaws/data/species_dat.csv** depending on your operating system.



11 <https://here.r-lib.org/> accessed 15th August 2025

Programming

Dependency management

Documenting and managing dependencies are essential for reproducibility because the software changes over time. If you, for example, wrote your code with a recent version of an R package and gave it to someone who has not upgraded recently, they may not be able to run your code, or they might get different results.

Dependency management can be done in a lot of ways. Below, you will find three levels of complexity to document the dependencies for your projects.

1. Show packages that you used

The simplest approach is to document all your dependencies in a file that you add to your project.



Find dependencies of your project

- **R**: use `devtools::session_info()` to get a nicely printed table of all dependencies. Add this information to your project (e.g. in a README file).
- **Python**: you can use `pip freeze` to list all installed packages and their versions. Save this information to a `requirements.txt` file.
- **Julia**: you can use `Pkg.status()` to list all installed packages and their versions. Save this information to a `Project.toml` file.

2. Use a project local library

Create a local library with the packages used in the project. This way, users don't have to use their globally installed software that might have a different version, they can use the local project library.

Programming



Create project local libraries

In **R**, you can use the **renv** package¹² to manage dependencies. Initialise **renv** in your project:

```
install.packages("renv")
renv::init()
```

In **Python**, you can use **pip** and **venv** to manage dependencies. Specifically, **venv** (a standard package shipped with **Python 3**¹³) supports lightweight 'virtual environments' that hosts its own set of independent packages. Here's the command for creating a virtual environment and install packages within that environment with **pip** (which are listed in a file named requirements.txt):

```
Python -m venv env
source env/bin/activate
# On Windows use `env\Scripts\activate`
pip install -r requirements.txt
```

In **Julia**, you can use the built-in package manager. Create a **Project.toml** file and activate the environment:

```
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

3. Use a container

A more advanced approach is to use containers, such as **Docker** or **Podman**, to encapsulate your entire environment. This ensures that your code runs in the exact same environment, regardless of the host system. Containers take more steps to set up but are especially useful for reproducing results when the analyses behind them require software packages that can be difficult to install.

12 <https://rstudio.github.io/renv/articles/renv.html> accessed 15th August 2025

13 <https://docs.python.org/3/library/venv.html> accessed 15th August 2025

Programming

For R, the [Rocker](#) project helps you provide container images with popular R software and optimized libraries pre-installed.

Namespace conflicts

Using multiple packages can result in namespace conflicts, where different packages have functions with the same name. This can lead to unexpected behaviour in your code. It is good practice to prefix functions with the package name to avoid this and make the dependency explicit.



Example of avoiding namespace conflicts

In **R**, both **dplyr** and **plyr** have a function called **summarise**. You can make R use the **dplyr** version either by using **dplyr::** or the **use** function:

```
dplyr::summarize(data, mean_value = mean(value))
```

OR

```
use("dplyr", c("summarise"))
```

In **Python**, if both **pandas** and **numpy** have a function called **mean**:

```
import pandas as pd
```

```
import numpy as np
```

```
mean_value = pd.DataFrame.mean(data)
```

```
mean_array = np.mean(array)
```

In **Julia**, if both **DataFrames** and **Statistics** have a function called **mean**:

```
using DataFrames
```

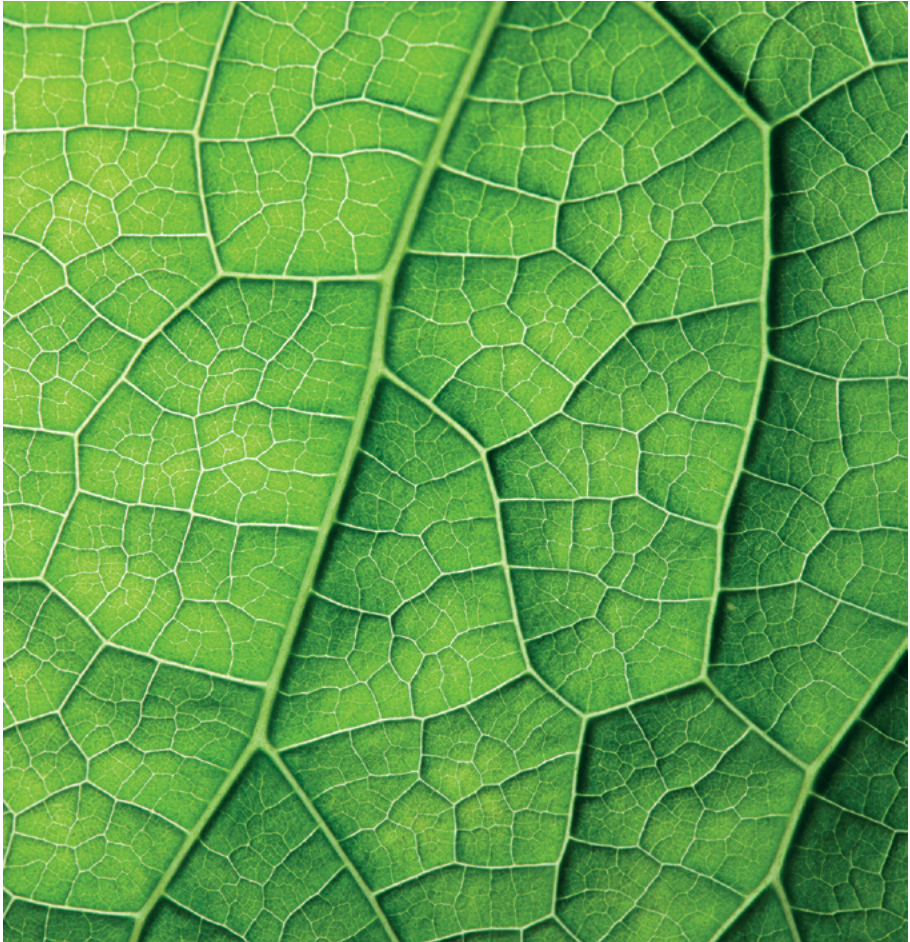
```
using Statistics
```

```
mean_value = Statistics.mean(data)
```

Programming

Other considerations

Set a seed: A random seed is a number used to start a random number generator. Explicitly setting a random seed ensures reproducibility as the random number generation will start at the same point. This is important when your code involves random number generation, which often occurs in some statistical modelling. For this, in R you can use `set.seed(123)` or use the `withr` package, in Python you can use `random.seed(123)` in Python and in Julia you can use `Random.seed!(123)`.



Code review

Zuzanna Zagrodzka, University of Sheffield and University of Manchester, UK

Code review is crucial for research reliability, transparency and reproducibility. It improves research workflow and clarity, helps identify errors, and enhances accessibility^{14,15}.

How to conduct code review

When reviewing code, it is important to check if it runs without errors, is well-documented and follows best programming practice to remain reproducible in the long term. **The 4Rs of Code Review** provide helpful guidance, particularly for code accompanying publications, but the same principles apply to stand-alone code or software. Additionally, **the FAIR** principles¹⁶ support maintaining quality, transparency and reusability, which are essential for long-term accessibility and reliability of code (see **Introduction**).

Remember, code review is also an opportunity to support learning and foster a positive, collaborative culture. To make your code review as helpful as possible (i) start with high-level feedback before diving into the details; (ii) give code examples to illustrate suggestions; (iii) keep comments open, respectful and directed at the code, not the author; and (iv) tie suggestions to principles like readability, maintainability, or performance rather than personal preference wherever possible.

14 Fernández-Juricic, E. (2021). Why sharing data and code during peer review can enhance behavioral ecology research. *Behavioral Ecology and Sociobiology*, 75 (7), 103. <https://doi.org/10.1007/s00265-021-03036-x>

15 Ivimey- Cook, E.R., et al. (2023). Implementing code review in the scientific workflow: Insights from ecology and evolutionary biology. *Journal of Evolutionary Biology*, 36 (10), 1347–1356. <https://doi.org/10.1111/jeb.14230>

16 Barker, M., et al. (2022). Introducing the FAIR Principles for research software. *Scientific Data*, 9 (1), 622. <https://doi.org/10.1038/s41597-022-01710-x>

Code review

The 4Rs of code review

1. **Reported: Does the code match what is described in the manuscript/description?**

For code linked to a publication, this means that the code matches the methodology in the manuscript, ensuring it performs as stated. Any discrepancies, such as differing model specifications, must be identified and all relevant packages and version numbers should be specified for reproducibility. For stand-alone code, this means ensuring that the code aligns with the software stated purpose and functions as described.

2. **Run: Does the code execute successfully?**

Even if the code aligns with the methodology, it must be executable without errors. Missing dependencies, typos, or lack of data can prevent execution. Providing data snippets or simulated data when the original data is not available is recommended.

3. **Reliable: Is the code producing accurate results?**

Code can run without errors yet still produce incorrect, reproducible results due to conceptual or programming mistakes. Thorough checks of intermediate outputs and using testing tools to catch hidden errors, especially as code complexity increases, are necessary to ensure code reliability (see **Defensive programming**).

4. **Reproducible: Do rerun results match previous results?**

Whether reviewing code linked to a paper or stand-alone code, the goal is to ensure that rerunning the code produces consistent results. Some variation is expected with stochastic methods, but numerical and statistical conclusions should remain stable within a reasonable tolerance. Reproducibility is assessed by comparing results using metrics like confidence interval overlap, direction and significance of effects or percentage error. Minor differences that don't change conclusions are acceptable, but reproduced results should closely match reported outputs to ensure reliability.

While the **4Rs** ensure that the code is functional and reliable within a research context, the **FAIR** principles help make it Findable, Accessible, Interoperable and Reusable (see **Introduction**) ensuring high technical quality and reuse

Code review

by both humans and machines. Without compliance with these principles, the code may not be easily found and accessed by reviewers or reused by other researchers.

Ensuring the FAIR principles

Documentation (reusable, accessible; see Publishing and archiving)

- Does the code include a README with clear installation and usage instructions?
- Is there a CITATION.cff or equivalent file to facilitate proper citation?

Version control (findable, accessible, reusable; see Version control)

- Is the code stored in a publicly accessible repository like GitHub, GitLab, or Codeberg?
- Does the repository include a CONTRIBUTING.md file with guidelines for contributors?
- Are issues and discussions actively managed via an issue tracker (e.g., GitHub Issues, GitLab Issues)?

License (accessible, reusable; see Publishing and archiving)

- Does the code have an appropriate license for reuse? Use choosealicense.com to choose one.
- Is the license clearly stated in a LICENSE file in the repository?

Archiving (findable, accessible; see Publishing and archiving)

- Has the code been assigned a DOI for long-term accessibility? Platforms like Zenodo, FigShare or Software Heritage provide DOI assignments.

Interoperability and reusability (interoperable, reusable)

- Does the software use open standards and widely accepted formats to ensure compatibility?
- Are there clear instructions for installing dependencies and running the code?
- Does the project include an application programming interface (API) or guidelines for extending the code?

Code review

Testing (reusable, interoperable; see **Programming**) - *when appropriate*

- Does the repository include tests?

Community registry (findable, accessible, reusable) - *when appropriate*

- Is the software published in a community repository?
- Common registries:
 - R: CRAN, rOpenSci, Bioconductor.
 - Python: PyPI, Anaconda Cloud Gallery.

When to conduct code review

Peer-reviewed code fosters organised workflows, enhancing research transparency and reusability. It also helps train students and early-career researchers (and senior researchers too!) in best programming practices. Code review can take place before publication, during formal review and after publication when reusing someone else's code and identifying errors. It can also take place as a routine part of developing code, even if the end goal is not a publication.

- **Pre-submission peer review:** Pre-submission code review should become a standard practice for each research team. It can be effectively implemented by establishing a code review group within research teams and using platforms like GitHub for collaboration and discussion.
- **Peer review during submission:** Many journals (including the BES journal *Methods in Ecology and Evolution*) now require code to be submitted alongside data for peer review¹⁷. Additionally, some journals in ecology and evolution, such as *The American Naturalist*, have data editors who specifically review the supporting code and data. Providing access to code during peer review has multiple benefits: it allows reviewers to verify that the code produces the reported results and ensures statistical analyses align with the study's design. To keep code review anonymous, authors can upload their code to platforms like Figshare, the Open Science Framework (OSF), or Zenodo and provide an anonymised link to reviewers.

17 <https://doi.org/10.32942/X2492Q> accessed 15th August 2025

Code review

- **Post-publication peer review:** After a study is published, researchers reusing the code may find errors. If they do, they should reach out to the original authors. Many journals encourage authors to correct mistakes, often facilitated through project issues or direct communication. This process helps prevent the spread of errors and enhances the reliability of published findings.



Reproducible notebooks

Batool Almarzouq, University of Liverpool, UK

A reproducible notebook is a digital document that combines text, code and results (like figures and tables) in one place. The key feature is reproducibility: anyone can run the notebook again and get the same results, as all the steps, data and code are included together. This is especially important in science and data analysis, where others need to verify or build upon your work.

Why use a reproducible notebook?

Using a reproducible notebook makes your work much clearer and easier to understand for both yourself and others. These notebooks let you show every step you took to get your results, from the raw data all the way to the final charts and tables (see **Organising projects for reproducibility**). Because your code and explanations are together in one place, anyone reading your notebook can see exactly how you did your analysis, which makes your work more trustworthy and transparent. If you ever need to update your data or fix a mistake, you can change the data or code, and the notebook will automatically update all the results and figures for you. This saves time and helps avoid errors.

Using reproducible notebooks also reduces mistakes introduced when copying and pasting between different software programs. They help keep your results and models synchronised, so you always know which code produced which output. You can also give readers more insight into your research process by including details about different approaches or analyses you tried before reaching your results. These extra details can be added as supplementary material or tracked in your version control system (see **Version control**), making your work even more open and robust.

You can create reproducible notebooks using tools like Quarto and Jupyter. Both are language agnostic, meaning you can use R, Python, Julia and other languages in the same document. In this section, we will focus on Quarto.

What is Quarto?

Quarto is a modern, open source tool for creating reproducible notebooks and technical documents. You can mix text (written in Markdown), code (Python, R, Julia, etc.) and outputs like plots and tables. Quarto documents can be turned into polished reports, websites, PDFs, slides and more, all from a single source file.

Reproducible notebooks

Quarto helps you keep your code, results and explanations together so others can see exactly how you did your analysis and can easily reproduce or build on your work.

Getting started with Quarto

You can use Quarto in many IDEs, including [RStudio](#) and [VS Code](#). In both RStudio and VS Code you create a new file with the `.qmd` extension to start writing your notebook. You can write text and code in this file. In RStudio you can toggle between the “Source” and “Visual” options using buttons near the top of the file. “Source” shows the notebook in Markdown format, “Visual” shows you what the rendered final file will look like. In VS Code you can use the “Quarto: Preview” command or press `Ctrl+Shift+K` to render and view your output as HTML, PDF, or Word.

Setting up your document (YAML Block)

Every Quarto document starts with a YAML block at the very top, which is used to set important information and options for your document. In the YAML block, you can add information like the document’s title, author and date, as well as specify the output format^{18,19,20}. You can also include details like the author’s affiliation, a subtitle, keywords, table of contents, bibliography files for citations and custom settings for how code and figures are displayed. YAML can handle simple values, and more complex settings like nested options for output formats or author details²¹.

The YAML block is surrounded by three dashes at the top and bottom:

```
---  
title: "My First Quarto Document"  
author: "Your Name"  
date: "2025-04-21"  
format: html  
---
```

18 <https://quarto-tdg.org/yaml.html> accessed 15th August 2025

19 <https://quarto.org/docs/authoring/front-matter.html> accessed 15th of August 2025

20 <https://rpubs.com/drgregmartin/1266674> accessed 15th August 2025

21 <https://quarto-tdg.org/yaml.html> accessed 15th August 2025

Reproducible notebooks

Writing text with Markdown

Below the YAML block, you write your main content using Markdown, a simple way to add formatting to your text, like headings, bold, italics and lists, using plain characters. Markdown helps keep your writing readable and easy to edit.

Figure 2: **A Markdown file with the left side displaying the raw Markdown syntax and the right side showing how it appears when rendered. The screenshot was rendered from <https://markdownlivepreview.com/>.**

```
1 # Markdown syntax
2
3 ## Headers
4
5 # This is a Heading h1
6 ## This is a Heading h2
7 ##### This is a Heading h6
8
9 ## Emphasis
10
11 *This text will be italic*
12
13 **This text will be bold**
14
15 You can combine them
16
17 ## Lists
18
19 ### Unordered
20
21 * Item 1
22 * Item 2
23 * Item 2a
24 * Item 2b
25     * Item 3a
26     * Item 3b
27
28 ### Ordered
29
30 1. Item 1
31 2. Item 2
32 3. Item 3
33     1. Item 3a
34     2. Item 3b
```

Markdown syntax

Headers

This is a Heading h1

This is a Heading h2

This is a Heading h6

Emphasis

This text will be italic

This text will be bold

*You **can** combine them*

Lists

Unordered

- Item 1
- Item 2
- Item 2a
- Item 2b
 - Item 3a
 - Item 3b

Ordered

1. Item 1
2. Item 2
3. Item 3
 - i. Item 3a
 - ii. Item 3b

Reproducible notebooks

Adding code chunks

A code chunk is a special block where you write code, which Quarto can run to show the results in your document. Code chunks start and end with three backticks (```) and the language you are using:

```
```Python
import matplotlib.pyplot as plt
plt.plot([1,1],t.show())
```
```

You can also add options to your code chunk. For example, adding ```echo: false``` at the top of the chunk will hide the code and show only the result (such as the plot). This is useful if you want your readers to see just the output, not the code itself.

Code chunk refers to a section of your document where you write and run code whereas `echo` controls whether the code is shown (```echo: true```) or hidden (```echo: false```) in the final document.

Inserting figures and images

When your code creates a plot or image, Quarto automatically includes it as a figure. You can add a caption, a short description under the figure, and a label, a name you use to refer to the figure later, by adding lines at the top of your code chunk. For example:

```
```
| label: fig-simple
| fig-cap: "A simple line plot"
| fig-alt: "Here is where to put alt text"
plt.plot([1,1],t.show())
```
```

This will show the plot with the caption “A simple line plot” underneath; you can refer to this figure elsewhere in your document using its label.

Reproducible notebooks

Creating tables

You can also create tables from your data using code. Quarto will display the table in your document, and you can add captions and labels just like with figures. For example, if you are using Python with the **pandas** library, you can create a small data table and show it in your notebook like this:

```
---  
#| label: tbl-sample  
#| tbl-cap: "Example of a simple data table"  
import pandas as pd  
  
data = {  
    "Species": ["Oak", "Pine", "Birch"],  
    "Height_m": [20, = pd.DataFrame(data)]  
df  
---
```

In this example, the table will appear in your document with the caption "Example of a simple data table". The label **tbl-sample** allows you to refer to this table elsewhere in your text.

Cross-referencing figures and tables

If you want to refer to a figure or table in your text, you use the label you assigned to it. In the table example above, the data table has the label **tbl-sample**. Writing "See Table **@tbl-sample** for a summary of the data" means that Quarto will automatically turn **@tbl-sample** into a clickable link to the table in your document. This process is called a **cross-reference**. Cross-referencing makes your document easier to navigate and helps readers quickly find the information you mention.

Adding citations

To refer to books, articles, or other sources, you can add citations in your Quarto document. You keep your references in a separate file, usually called `references.bib`, which uses the BibTeX format. For example, by writing a citation like **[@smith2020]** in your text Quarto will format it and add it to the reference list at the end of your document.

Reproducible notebooks

Software citation managers such as Zotero or Mendeley can help you collect and organize your references, and they provide options to export your citations as a BibTeX (**.bib**) file. If you want your references to follow the style of a specific journal, you can use a Citation Style Language (CSL) file. CSL files for most journals can be downloaded from the [Zotero Style Repository](#).



Version control

Nilanjan Chatterjee, Senckenberg Biodiversity and Climate Research Centre, Germany

Version control enables multiple users to collaborate efficiently by recording modifications, keeping a history of changes and allowing users to revert to previous versions when needed.

Why is version control important?

Version control is essential for research, software development and document management because it:

- Prevents data loss by maintaining backups of previous versions.
- Facilitates teamwork by allowing multiple users to work on the same project without overwriting each other's changes.
- Ensures consistency and organisation across different project versions.
- Enhances transparency and accountability by keeping a log of who made changes and when.

By using version control systems like [Git](#), teams can work on different features simultaneously, troubleshoot issues and integrate changes efficiently. This makes version control a fundamental tool for managing projects that involve continuous updates and collaboration.

Why can't I just use Dropbox? Cloud storage services (e.g. Nextcloud, Google Drive, OneDrive, Dropbox) offer access to a file's version history, backing up file versions for a limited time (usually 30 days) and allowing restoration of previous versions within this time frame. However, although these tools are easy to use, they are not a recommended route because there is limited control over revisions and they are not intended for long-term archiving. Instead, we strongly advise using a dedicated version control software.

Version control software

Version control software is designed to help you manage your file revisions. The software runs directly on your computer, allowing you to manage files within your local file system. You can also use version control software to interact with external copies of versioned files if you choose. Here we focus on the widely used open source software [Git](#).

Version control

Git offers flexibility in how you can use it for version control. You can use it as a stand-alone tool to manage files on your own computer and, optionally, you can use it to connect to an external service to archive your files. Git can be used directly on the commandline, through other IDE software, or via stand-alone graphical programs (the official Git website lists several options²²), so it is flexibly integrated into your normal workflow.

Git is a distributed version control system, which means that each user interacts with a stand-alone copy of the versioned files. This stand-alone copy is called a **repository** (often shortened to repo) and is usually a folder on your computer that contains all the work for a particular paper or project. Individual repositories synchronise with each other by exchanging information about what changes have been made. Multiple users can work on their local version in the same repository, known as **branches**. Various branching and merging options enable multiple team members to work on and develop different features simultaneously.

22 <https://git-scm.com/downloads/guis> accessed 15th August 2025

Version control



Getting started with Git: download and configure

To get started, you will need to download and [install Git](#).

Once installed, configure Git with your name and email address so that this information can be added to your version history. This can be done with the graphical programs or on the command line. To access the command line in Windows use Windows Terminal or Command Prompt, in Mac/Linux use Terminal. You can also use the Terminal tab in RStudio or VS Code in any operating system. Enter the following three lines of code.

```
git config --global user.name "Your Name"  
git config --global user.email your.email@domain.com  
git config --global credential.helper "store"
```

Once this is done, users can check if it has been configured correctly with:

```
git config --global --list
```

This should return the options you just entered.

Git has [excellent documentation](#) that will help you get started.



Getting started with Git: integrating Git with an IDE

Git can be easier to use with an IDE like RStudio or VS Code, or with a dedicated desktop app.

- **RStudio-Git Integration**

Detailed instructions for using Git with RStudio are at [Happy Git and GitHub for the useR](#).

- **VS Code-Git integration**

Full instructions for how to set up Git in VS Code are available at [Introduction to Git in VS Code](#). To enable Git in VS Code, go to **File > Preferences > Settings**. Type Git: Enabled in the search bar and make sure that the box is ticked. For further controls of the Git repository, click the Git option on the File Tab.

Version control

Version control repository hosting services

To facilitate collaboration, researchers often use version control repository hosting services online, such as GitHub, GitLab and Codeberg. One of the most popular is GitHub. For most users, the free of charge version of GitHub should be fine, but there is a subscription option to an educational plan for many colleges and universities (at the risk of being locked into GitHub)²³. Some repository hosting services require two factor authentication (2FA) and a Personal Access Token (PAT) to push things directly to your remote repositories. See the documentation for the service you are using for details.

Version control workflow

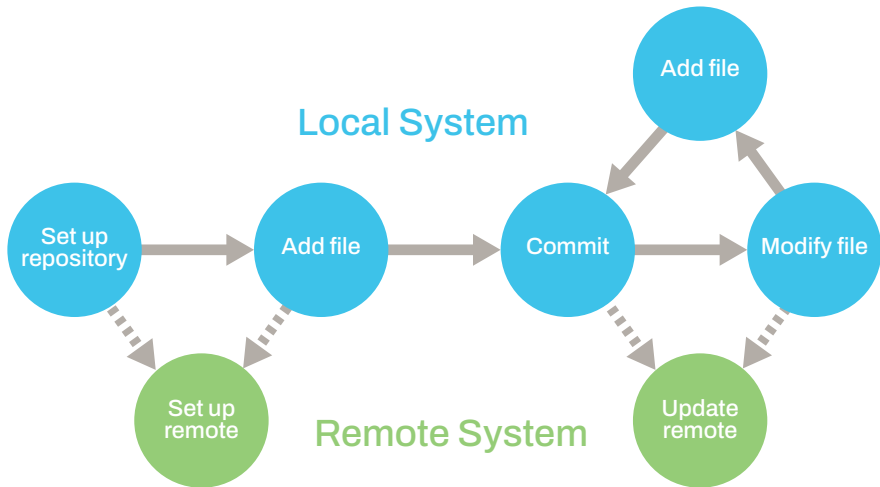
Understanding how to run a version control workflow can be tricky for new users. The official [online documentation](#) is excellent and provides screenshots and video examples. We recommend you refer to the documentation, but a typical version control workflow with Git consists of the following steps (see also Figure 3).

1. **Set up** a new repository/project on your computer and ask Git to track it.
2. **OPTIONAL** Set up a remote repository connected to the repository on your computer.
3. **Add** or **Stage** the file(s) that you want to include for version control.
4. **Commit** the initial file(s) with a suitable commit message.
5. **OPTIONAL Push** initial files to your remote repository.
6. Make changes to your file(s) and save.
7. **Add** or **Stage** the file(s).
8. **Commit** changes with a suitable commit message.
9. **OPTIONAL Push** changes to your remote repository.
10. Repeat steps 6-9.

23 https://education.github.com/discount_requests/application accessed 15th August 2025

Version control

Figure 3: A typical version control workflow showing the steps on local and remote systems.



Use Git to carry out these steps:

1. Setting up a repository

When you use Git, you interact with a set of versioned files called a **repository**. A Git repository is a directory on your computer with some hidden files for bookkeeping. You can create a Git repository from scratch when you create an RStudio/VS Code project. You can also clone Git repositories from existing repositories (such as those on GitHub) or create them directly using the command line. If you have set up a version-controlled project in RStudio/VS Code, the operations described below are available from the “Git” tab within the Environment/History pane in RStudio or “Git” Tab in VS Code.

2. Adding or staging files

Git does not automatically track all files in your project. You must **stage** or **add** files to let Git know which ones should be included in version control. **Staging/adding** a file means it is prepared to be saved as part of your version history in the next step (**committing**).

Version control

3. Creating a commit

A **commit** is a snapshot of the changes to the repository. It acts like a checkpoint in your project, allowing you to keep track of modifications over time. When you create a commit using Git you will be prompted to enter a **commit message**. The message can be used to provide a description of the changes that you have made. Each commit is assigned a unique identifier that is stored together with the date, author and commit message. This identifier can be used to specify a commit later.

4. Connecting to a remote repository - a local Git repository allows you to track changes to files, storing the details in your version history and allowing you to go back to previous file versions, if necessary. If you want to create an external copy of your version history and share code with others, you should consider linking your repository to a **remote** repository. Synchronising your local repository to the remote repository will archive your code and provide a centralised store for your project, making it easy for you to collaborate with other users, share your work or to access it from another computer.

You can set up a remote repository on a hosting service or host it on your own server. Public repositories are publicly viewable, but this does not mean anyone can directly make changes to them. Private repositories are accessible only to those who have been granted access. However, for very sensitive data, online repositories are not advisable as they do not always remain private. Git also uses a file called **.gitignore** which users can modify manually to ensure which files (tokens, passwords, very large data files etc.) are to be ignored for **staging/adding** or **committing** to the repository. You can also use the **usethis** function **git_vaccinate** to help with this (https://usethis.r-lib.org/reference/git_vaccinate.html).

5. Pushing to the remote repository

This step sends the details of any **commits** which have been made locally to the specified remote repository, which can act as a backup of your work or a resource to share with others. This step is optional; you can just work with Git locally on your computer.

Version control

Using version control

A Git version control workflow can be useful in several different ways.

Viewing your version history - the version history that you build up as you commit your work forms a record of the changes that you have made to your code or documents. This can help you to track a project's development, understand reasons for past programming decisions, highlight major revisions and identify where bugs were introduced.

Returning to a previous commit - the real power of Git will become clear when you need to return to a previous point in your revision history, for example to recreate figures for a paper or to run an analysis again. You can use the unique ID for a commit to tell Git to return your working directory to the state captured by that commit. You can then work with your code as it was at that point in time.

Using version control to collaborate - there are a few alternative models for collaborating when using version control. If you use a distributed system like Git, each individual collaborator works with their own repository, and you need to decide how to consolidate your work. A couple of possible workflows are:

1. Everyone connects their local repository to the same remote repository and must coordinate their changes. Changes to the central repository should be integrated into a user's local repository before they submit their own changes to the remote repository.
2. Individual contributors create a copy or **fork** of the main repository and use this as their remote repository. They must send a request to the maintainer of the original repository to incorporate their changes into the repository. This model allows a formalised review process before changes are integrated and is often used in open source projects.

There are many advanced features we have not included here. See [Git - Documentation](#) for more information.

Version control



Git quick reference

| | |
|-----------------------|--|
| Add/Stage | The process of selecting a file for inclusion in version history. |
| Branch | A separate set of changes to version history allowing users to work in parallel on the same files, or for one user to experiment with different solutions. |
| Commit | A snapshot of changes to be added to version history. |
| Commit message | User-specified description of the changes made in a commit. |
| Conflict | A problem arising when changes from different sources cannot be combined automatically. |
| Fork | A remote repository derived from another project that can be used for collaboration. |
| Merge | Combining changes that originate from different repositories or branches. |
| Pull | An action that synchronises the local repository with local changes. |
| Push | An action that synchronises the remote repository with local changes. |
| Remote | An external repository that can be synchronised with local changes. |
| Repository | A directory containing the files under version control. |
| Status | Displays the status of modified files in the working directory. |

Publishing and archiving

Esther Plomp, University of Aruba, Aruba

In many areas of research, academic papers form only a part of scholarly output, and they frequently do not contain enough information for reproducibility and detailed discourse. A more complete description of the research requires the publication of custom-written code, details on all software used (such as version information) and supporting data files.

In this section, we propose methods of code and data publication that are suitable for inclusion in the permanent scholarly record. This section also stresses that software should be cited on the same basis as other research outputs and accorded the same importance in the scholarly record as citations of other research products, providing scholarly credit and normative, legal attribution to all contributors to the software.

Publishing research software

To work towards satisfying the FAIR principles (see **Introduction**), we recommend the use of research data repositories such as Zenodo, Dryad or Figshare when publishing code and data. These data repositories provide a persistent digital object identifier (DOI) and long-term preservation of research outputs. Zenodo, developed and maintained by the European Organization for Nuclear Research (CERN), is free of charge for most usage up to 50 GB and is available to researchers from all research fields. Figshare is a commercial option with a free usage tier up to 20 GB. Both Figshare and Zenodo provide the option to version DOIs for the research output, allowing you to further develop software and update your repositories over time in a manner that they stay clearly connected. If used in association with GitHub some information is automatically recorded. **See the further resources section for this chapter for links on how to do this.** Many universities have institutional repositories that can also be used for code and data archiving or may require you to register your research outputs in other systems.

You can also publish your code or software packages (see [CRAN for R](#) and [PyPI for Python](#)) via software registries and software papers. See Chue Hong's "Doing Science in the Digital Age (a personal journey as a data explorer)"²⁴ for an overview of software publication options.

24 Chue Hong, N. (2021). Doing Science in the Digital Age (a personal journey as a data explorer). Figshare. Presentation. <https://doi.org/10.6084/m9.figshare.17094365.v1>

Publishing and archiving

Licensing

A **licence** is an explicit statement that grants certain uses of a work. It is critical to include a licence to allow people to reuse code when publishing it to an online repository. For text content (such as journal publications), Creative Commons licences are typically used. However, these are not suitable for research code, for which an open source license should be selected²⁵. Guides on how to choose appropriate licenses for your work have been published by the [Software Sustainability Institute](#) and [The Turing Way](#). It is important to license your work so that it is also machine readable, making licensing less ambiguous²⁶.

Attaching a license to your work is straightforward. Once you have chosen the licence you wish to use, add a text file called LICENSE or LICENSE.txt to your project that contains the license text. The website [choosealicense.com](#) contains simplified information about available licenses along with the full license text for you to copy and paste into your LICENSE or LICENSE.txt file. When using GitHub, you can also start with adding a license file to your repository, which will open license templates that you can use. We recommend that you use an existing license, rather than writing a custom license, as this prevents confusion about reuse. You should also check whether your organisation has a policy on publishing software and recommended licenses that you should use, and guidance on how to include information about the copyright holder.

README

A README file provides information about the software and is intended to help ensure that the software can be correctly interpreted and used, by yourself or others. A README file is generally the landing page of your software repository. See [Make a README](#) for more information.

25 <https://opensource.org/docs/osd> accessed 15th August 2025

26 <https://reuse.software> accessed 15th August 2025

Publishing and archiving

Citation

A software citation ideally contains the following information²⁷:

Example Software Citation (APA): Developer, A. A., Developer, B. B., & Developer, C. C. (yyyy) 1. Title of the software: Subtitle (Version #.#) 2. [Computer software] 3. Publisher 4. <https://URL>.

The CFFinit tool can be used to create a CITATION.cff²⁸. **See the further resources section for this chapter for more information about using this tool.** Citing the underlying code of a publication within the publication itself improves the visibility of research outputs. It is also important to cite any other software that you reused. References to the research software you created or reused should also be included in the Data/Code Availability Statement²⁹ in the research article.

Putting the FAIR principles into action

1. Get a DOI.

Share your software via a data repository or software paper.

2. Ensure that others can understand and execute your code.

Include sufficient metadata and documentation of the code, as well as a README file that introduces and explains the project. This includes information about how to use the code, its dependencies and citations of software you reused. Save the code in a format that can be opened by any text editor or IDE, such as .txt.

3. Add a LICENCE to enable possible reuse.

In the code repository, include a LICENSE file that outlines the reuse requirements.

27 Katz D. S, et al. (2021). Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved]. F1000Research, 9:1257. <https://doi.org/10.12688/f1000research.26932.2>

28 <https://citation-file-format.github.io/cff-initializer-javascript> accessed 15th August 2025

29 <https://book.the-turing-way.org/communication/citable/citable-cite#cm-citable-cite-data> accessed 15th August 2025

Publishing and archiving

4. Add a CITATION file to ensure credit, visibility and findability of your work.

In the code repository, include a CITATION.cff file that outlines how the code should be cited.

5. Linking your research outputs.

Cite the software in your research article (in citations and data/code availability statement) and refer to your research article in the code repository (for example, via your README file). Check all the references to related research outputs so all outputs are linked and findable.

For examples of how FAIR principles apply to code see **Code review**.

Figure 4: The five steps to FAIR research code





Further resources

Code Review

Software Quality Checklist:

<https://github.com/eurise-network/technical-reference/blob/v0.1/quality/software-checklist.rst>

Self-assessment for FAIR research software:

<https://fairsoftwarechecklist.net/v0.2/>

Software Quality Guidelines:

<https://github.com/CLARIAH/software-quality-guidelines/blob/v1.0/softwareguidelines.pdf>

Reproducible Notebooks

Tutorial: Hello, Quarto:

<https://quarto.org/docs/get-started/hello/vscode.html>

The Ultimate Markdown Cheat Sheet for Technical Writers and Documentation Engineers:

<https://medium.com/@kevinteaches/the-ultimate-markdown-cheat-sheet-for-technical-writers-541dbb9fd53c>

Version Control

Happy Git and GitHub for the user:

<https://happygitwithr.com/>

RStudio User Guide Release 2025.05.1: Version control:

<https://docs.posit.co/ide/user/ide/guide/tools/version-control.html>

Version control using RStudio:

<https://nceas.github.io/oss-lessons/version-control/4-getting-started-with-git-in-RStudio.html>

Reproducible research with R, RStudio and GitLab: Version control:

<https://matthieu-bruneaux.gitlab.io/guide-r-rstudio-git-gitlab/020-intro-version-control.html>

Visual Studio IDE: About Git in Visual Studio:

<https://learn.microsoft.com/en-us/visualstudio/version-control/git-with-visual-studio?view=vs-2022>

Further resources

Publishing and Archiving

GitHub Docs: Referencing and citing content:

<https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content>

How To Make Your GitHub Code Citable With Zenodo: pyOpenSci:

<https://www.youtube.com/watch?v=1pl4QU-7c98>

How to connect Figshare with your GitHub account:

<https://help.figshare.com/article/how-to-connect-figshare-with-your-github-account>

OSF Support: Connect GitHub to a project:

<https://help.osf.io/article/211-connect-github-to-a-project>

Dataverse Uploader Action:

<https://github.com/marketplace/actions/dataverse-uploader-action>

Intro to Zenodo: Advancing Open Science (Esther Plomp):

<https://www.youtube.com/watch?v=eChOfh8t04k>

How to create a CITATION.cff using cffinit:

<https://www.youtube.com/watch?v=zcgLIT5Qd4M>

How to update an existing CITATION.cff file using cffinit:

<https://www.youtube.com/watch?v=6Ik1onYbO3A>

TADA! Simple guidelines to improve code sharing:

<https://ecoevorxiv.org/repository/view/9806/>

Acknowledgements

This booklet was coordinated by Amelia Macho and Kate Harrison of the BES. We thank Edward R. Ivimey-Cook, Antonio J. Pérez-Luque, Luis D. Verde Arregoitia, Nemo Andrea, Carlos Martinez Ortiz, Bryan M Gee, Daniel Padfield, Emma Dunne, Michael D Catchen, Jiangyue Wang, Gbadamassi Gouvide Olawole Dossa and Harriet Rhodes for reviewing the second edition of this guide.



Thousands have
already joined our
global community

britishecologicalsociety.org

.....

Through science we can