Research Ideas and Outcomes 12: e216899
doi: 10.3897/rio.12.e216899

OPEN ACCESS

Guidelines

# B-Cubed software development guide

Pieter Huybrechts‡, Ward Langeraert‡, Peter Desmet‡

‡ Research Institute for Nature and Forest (INBO), Brussels, Belgium

Corresponding author: Pieter Huybrechts ([email protected])

Reviewable v 1

Received: 24 Sep 2026 | Published: 19 Oct 2026

Citation: Huybrechts P, Langeraert W, Desmet P (2026) B-Cubed software development guide. Research Ideas and Outcomes 12: e216899. https://doi.org/10.3897/rio.12.e216899

## Abstract

This guide specifies high-level requirements for software, computational tools and resources developed for B-Cubed (referred to in the sections as “software”) to ensure that the produced software meets the intended quality, openness, portability and reusability.

## Keywords

data cubes, FAIR software, software quality standards, open software, software metadata, interoperability, b-cubed, software versioning, software testing

## Introduction

These requirements*[1] were carefully selected from numerous existing best practices and guidelines, and aim to promote a consistent **open source development** cycle that allows collaboration and reuse within and outside of the consortium. Emphasis is placed on standardized metadata (files) that make it easier for both humans and search engines to find the software, and thus to increase its discoverability and reuse. In this same vein, emphasis is placed on the **portability** of the produced software to make sure it is functional on different platforms now and in the future with minimal modifications. Following existing paradigms and design patterns makes the behaviour of the software more predictable and makes results easier to replicate. By following the recommendations in this document, interoperability between software packages can be achieved.

© Huybrechts P et al. This is an open access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

The sections include requirements, as well as hands-on instructions and examples. They cover topics such as code repositories and collaboration, and in-depth development best practices, including testing and documentation, for both the R programming language. At the head of every section an overview is offered [2]that summarizes the minimal requirements (MUST). The text of the section can include additional recommendations (SHOULD or RECOMMENDED).

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in RFC 2119 (Bradner 1997).

## Code repositories

All software code MUST be maintained on GitHub or GitLab. Code is maintained in a **repository**, which contains all files, discussions and version history related to a single software package or analysis.

### B-Cubed software requirements

- All software codes MUST be maintained on GitHub.
- An installable software tool MUST be maintained in its own repository.
- A repository MUST be public and be part of a GitHub organization (e.g. *b-cubed-eu*).
- A repository MUST contain a `.gitignore` file.
- A repository MUST contain a `LICENSE` file and be licenced under the MIT licence.
- A repository MUST contain a `README.md` file.
- A repository MUST contain a `CITATION.cff` file.

### Create a repository

You first need to define the scope of your repository. An installable software tool (R package, Python library, etc.) MUST be maintained in its own repository. For an analysis, choose a scope that is easy to manage and collaborate on.

It is RECOMMENDED that you start your repository before you write code. That way, you can follow best practices, others can contribute and all version history is captured from the start.

The easiest way to create a repository is in your browser:

1. Login to GitHub (https://github.com).
2. Go to https://github.com/orgs/b-cubed-eu/repositories. If you do not see a *New repository* button (green), then you are not yet invited to the B-Cubed organization on GitHub. Email your GitHub username to GitHub B-Cubed admin and wait with the following steps until you are invited.

3. Follow the Quickstart for repositories instructions and create a new repository at https://github.com/organizations/b-cubed-eu/repositories/new:
    1. Choose *b-cubed-eu* as owner. If that option is not available, see step 2.
    2. The repository name SHOULD be lowercase, dash-separated and short.
    3. The description SHOULD be a descriptive, one-sentence title (without period at the end), such as *R package to read and write Frictionless Data Packages*.
    4. The visibility MUST be set to *public*. This makes it easier to collaborate and reference files and code.
    5. Check *Add a README file*.
    6. You MUST select a `.gitignore` template (e.g. R, Python)
    7. You MUST select a licence and you MUST set it to *MIT License*. This conforms to the B-Cubed Data Management Plan.

If you already have your code (locally), follow About adding existing source code to GitHub. Using GitHub Desktop is the easiest option.

If your code is already on GitHub under a personal account, it MUST be transferred to the *b-cubed-eu* organization or your institution, if it has a well-established track record of maintaining code on GitHub. Contact the GitHub B-Cubed admin to gain the rights to transfer your repository to the *b-cubed-eu* organization.

Once you have created a repository, you SHOULD complete a number of additional steps.

## Set the copyright holder

1. Go to the `LICENSE` file.
2. Click the pencil icon.
3. Select *Choose a license template*.
4. Choose *MIT License*.
5. Select the *year* the software was started.
6. Set Full name to the institution where the maintainer of the software is employed (e.g. *Research Institute for Nature and Forest (INBO)*). When in doubt, leave as *B-Cubed*.
7. Commit the changes.

## Ignore Mac .DS_Store files

Mac operating systems create .DS_Store files to store attributes of a directory. These can clutter your repository and should be ignored.

1. Go to the `.gitignore` file.
2. Click the pencil icon.

3. Scroll to the bottom and add the following code (before the empty line): `# Mac OS .DS_Store`
4. Commit the changes.

## Add a CITATION.cff file

Repositories MUST contain a CITATION.cff file so users know how to cite the software. Its metadata also gets picked up when depositing a repository to Zenodo (see releases. For more information see What is a CITATION.cff file or GitHub's About CITATION files.

1. Go to the main page of your repository.
2. Click *Add file* then *Create new file*.
3. Name your file `CITATION.cff`.
4. An info box will appear, select *Insert example*.
5. Include the name and ORCID of the maintainers.
6. Remove the lines *doi* and *date-released*.
7. Commit the changes.

Note: this file can be updated later (manually or through functions).

Note: a `CITATION.cff` is different from the R-specific `CITATION` file (without an extension).

## Add topics

1. Follow the Classify with topics.
2. Add a number of topics, including the language (*r* and *rstats* or *python*), the type of software (e.g. *r-package*, *analysis*) and related subjects (e.g. *invasive-species*), cf. the section on GitHub repo topics in the rOpenSci Packages guide (rOpenSci et al. 2024).

## Hide irrelevant tabs

1. Go to the *Settings* tab.
2. In *Features*, turn off *Wikis* and *Projects*. These features will likely not be used.

## Invite collaborators

1. Contact the GitHub B-Cubed admin to indicate who you want to invite. The admin can then organize the collaborators in teams.
2. Follow the Invite collaborators instructions (these are for personal repositories, but many of the steps apply to organization repositories).
3. Type the GitHub name of the collaborator you want to add.
4. Indicate the rights (*Read*, *Triage*, *Write*, *Maintain*, or *Admin*).
5. The collaborator will receive an email invitation to collaborate.

### Extend your README.md file

See the README file sections.

### Setup your local environment, contribute code and collaborate

See the Code collaboration section.

## The README file

A README file communicates the most important information about your repository/ software. It will serve as a welcome sign for users, meaning that it will be the first and maybe most important piece of metadata that users will encounter. It often also serves as the landing page for a documentation website.

Maintainers SHOULD extend a README beyond its initial template when it was created as soon as possible, as it helps to define scope and expectations and facilitates collaboration. General guidance on writing a README can be found in GitHub's About READMEs or Make a README, while software languages (e.g. Python or R) often have specific instructions. Some suggestions for its contents are detailed in the sections below. See the README.md of the frictionless package (Desmet et al. 2025) as an example.

### B-Cubed software requirements

- The README file MUST be written in Markdown, unless the software language recommends otherwise.
- The README file MUST start with a title.
- The README file MUST include a brief introduction to the repository/software.

### Format

The README file MUST be written in Markdown (and therefore be named README.md), unless the software language recommends otherwise. Python for example recommends structured text (Reitz and Schlusser 2016). See the GitHub's Basic formatting syntax guide for more information on how to write Markdown.

### Title

A README MUST start with an H1 title with the (human-readable) name of the repository/ software. The title is generally the same as the name of the package, see Naming your package.

## Badges

Right below the title you can optionally show badges or shields to convey the current development status, link to a publication or archive, test coverage and more. Many GitHub workflows come with a status badge and static shields can easily be created using https://shields.io/badges. Vincent A. Cicirello provides a general overview in this blog post (Cicirello 2022).

Maintainers SHOULD at least include a repo status or lifecycle badge, to indicate the maturity/support for the repository/software. See repostatus.org for statuses or lifecycle Henry and Wickham 2023) and how to add it as a badge.

## Description

Below the Title and the optional badges, a brief, title-less introduction MUST be provided, explaining the rationale and/or scope of the repository/software. A software package might initially limit its scope to only part of a bigger problem, and signal this in its description. For example, a package wrapping an API might only support reading from that API, not writing to it. Or an analysis library might only initially offer statistical functionality, but not any visualization of results.

## Installation instructions

While it can be very clear how to install a software package for those who were closely involved in writing them, the same is not always true for external users. Thus minimal instructions SHOULD be included in the README on how to install the software and its dependencies.

## Examples or usage instructions

Similar to the installation instructions at least one example of the functionality of the analysis workflow or software SHOULD be included in the README.

## README files for data

A repository can have additional README files beyond the one in the root. These typically serve as an introduction to a specific directory. These SHOULD NOT be used, as there are better ways to document code, but it can serve as a quick way to describe data files. See this guide by Cornell University Data Services or Dryad's best practices document for guidance. Better yet is to deposit your data elsewhere.

## Code collaboration

Open source software relies on **collaboration**. Participants in this process are not only developers, but anyone interacting with the software (code), such as maintainers, contributors, testers, users reporting issues, etc. To facilitate the collaboration process, it is good to adopt a number of community standards and best practices (see below).

For more information on open source software collaboration, see Finding ways to contribute to open source on GitHub (also useful for non-developers) and GitHub's Open source guides. See how well your repository is adopting community standards at *https://github.com/b-cubed-eu//community*.

### Add a Code of conduct

A code of conduct is a document that establishes expectations for behaviour from all software participants. Adopting and enforcing it can help to create a safe and positive working space.

All software MUST have a code of conduct, as a CODE_OF_CONDUCT.md file following the Contributor Covenant template. All participants to software MUST abide by its code of conduct.

To add a CODE_OF_CONDUCT.md:

1. Follow the Add a code of conduct instructions.
2. In step 5, choose the Contributor Covenant template.
3. Add as Contact method: [email protected] (this email is monitored by MeiseBG staff).
4. Before committing the changes, change the file name to .github/CODE_OF_CONDUCT.md.

Alternatively, you can complete these steps in R using: usethis::use_tidy_coc()

But update the default email address ([email protected]) in "Enforcement" to [email protected] before committing the file.

### Enable notifications

Notifications are (email) alerts of participant activity in a repository or issue thread you are subscribed to. They facilitate collaboration and relieve you from having to check into GitHub.com. Activities that can trigger a notification include issues, pull requests and releases. Commits do not trigger a notification, which is why the GitHub flow (i.e. pull requests) is recommended to inform collaborators of important changes. You also won't receive notifications for your own actions.

You are automatically subscribed to notifications based on your actions (like commenting on an issue) or the actions of others (like @mentioning or assigning you). You don't need

to be an official contributor to be notified; anyone can do so by clicking the Watch button on a repository homepage. Maintainers MUST watch the repository they maintain. If you receive too many notifications, you can control what events you want to be notified of.

When receiving a notification by email, click the view it on GitHub link at the bottom to interact. This generally provides better context and formatting options than your email client (see Report issues).

### Follow the GitHub flow

The GitHub flow is an easy-to-adopt practice for code collaboration that MUST be followed for all code contributions to B-Cubed software. It consists of making a **branch**, making changes, creating a **pull request**, addressing review comments, merging the pull request and deleting the branch. See GitHub flow for more information, including links to further documentation for all the steps.

### Protect the main branch

The main branch MUST contain the software code in a state that can be installed without issue. To ensure code contributions (via pull requests) are reviewed before these are merged into the main branch, you can configure your repository to do so:

1. Follow the Branch protection rule instructions.
2. For *Branch name pattern*, choose *main*.
3. *Require a pull request before merging* SHOULD be enabled, with the default *Require approvals*.

See also GitHub flow for working with branches.

### Contributing guide

Maintainers SHOULD clarify how participants can contribute to their software, by adding a contributing guide as a `CONTRIBUTING.md` file in the `.github` directory.

To add a `CONTRIBUTING.md` file:

1. Follow the Contributor guidelines instructions.
2. Copy/paste a template such as Peter Desmet's CONTRIBUTING.md or the Contributing to tidyverse.
3. Adapt where necessary.
4. Make sure the instructions do not contradict the Github flow.

Alternatively, you can complete these steps in R using: `usethis::use_tidy_contributing()`

Which will use the Contributing to tidyverse template.

### Report issues

While the GitHub flow lowers the barrier for making code contributions, it is useful (and saves you from writing unnecessary code) to interact with the maintainer(s) before suggesting changes. The easiest way to do so is by creating an issue.

Issues can be used to report and discuss a bug, idea or task. Issues are typically not used to ask for support in using the software. Anyone can create an issue or comment on it, and all participants watching the repository will get a notification. Once an issue is resolved (by fixing the bug, implementing the feature, or deciding not to act upon it) it can be closed. Closed issues are still accessible and can act as a history of decisions.

Writing a good issue takes skill, see this blog post (Weigel 2023) or the tidyverse code review guide (Vaughan 2023) for guidance, and follow the contributing guide.

Just like the README file, issues (and pull request) support **Markdown** formatting that can improve readability, link issues to code and other issues, and notify people. See the GitHub's Basic formatting syntax guide for more information.

As a maintainer, you can nudge participants in the right direction by providing an issue template. Follow the Configuring issue templates for your repository instructions to do so. Alternatively, you can complete these steps in R using: `usethis::use_tidy_issue_template()`

But update or remove the references to https://stackoverflow.com or https://community.rstudio.com before committing the file.

### Local development

While some contributions can be made directly in the browser (one file at a time), most software development will be done locally, in an environment where it can be run and tested. Git (and the GitHub flow) allow these changes to be synchronized. Rather than explaining how to use git, we recommend the use of GitHub Desktop to facilitate this process.

GitHub desktop is a visual interface that allows you to commit your changes (include file parts and multiple related files), push those to GitHub.com, pull changes from contributors, resolve merge conflicts, and switch branches. It works well next to other code editors such as R Studio. See the GitHub Desktop instructions to get started.

## Versioning

Code versioning (or version control) is an essential aspect of software development. It provides a mechanism to keep a detailed history of the changes that are made to the source code, as well as the decisions leading to those changes. This allows for a deeper understanding of the code and facilitates code audits/reviews. Code versioning also

serves as a backup and recovery mechanism of the code. In case of critical errors or functionality loss, it is possible to revert to a previous working release of the software. Finally, versioning is an important communication mechanism for users of the software, especially software using it as a dependency. It indicates what changes can potentially break or alter existing functionality, offering the users the option to adapt their code or use a previous version.

## Semantic versioning

Software MUST use semantic versioning, where the version number is of the form MAJOR.MINOR.PATCH. An example of version changes is presented in Table 1.

Table 1.
Example of version changes

| | |
|---|---|
| 0.1 | First release |
| 0.2 | Minor release |
| 0.2.1 | Critical bug fix |
| 0.3 | Minor release |
| 1.0-rc.1 | Release candidate for version 1.0 |
| 1.0 | First release of the public API (i.e. the collection of user facing functions) |
| 1.1 | Minor release |
| 1.1.1 | Critical bug fix |
| 1.2 | Minor release |

There is no semantic difference between *x.y.0* and *x.y*.

## Git commits

Versioning is built into git, where changes are expressed as commits. Try to create logical commits, where related changes are bundled together (and leave unrelated changes for a following commit). Document your commit with a concise commit message in the active tense (*Add Pieter as contributor*, Use *'invalid'* over *'incorrect'*, etc.) and where necessary, document the reasoning in the commit description.

## GitHub releases

Major and minor versions MUST have an associated GitHub release:

1. Follow the Manage releases instructions.
2. Use the semantic version number as the tag (e.g. *0.1*, *1.1.1*).

Starting from release 1.0, authors MUST also publish their releases on Zenodo. Zenodo and GitHub are integrated, allowing this publication to be automated. See this tutorial (Vanderhaeghe 2022) for details.

By default, Zenodo will use the repository metadata to populate the deposit metadata. This will be of limited quality (e.g. GitHub usernames as author names). If a `CITATION.cff` file is present (see Add a CITATION.cff file, Zenodo will use that to populate the metadata. This will be of much higher quality, but you will typically have to review and update some properties manually on Zenodo, such as adding funding information, updating author affiliation to ROR-linked entries, updating the notes to the changelog and adding the deposit to the B-Cubed community.

See the b3doc R package (Govaert et al. 2025) as an example of a well-documented Zenodo deposit.

If you prefer to fully automate the setup, you can add a `.zenodo.json` file in the root of the repository. In this file you can define Zenodo-specific metadata that will be used when a release is created. Note that for B-Cubed software, you must include the `*b3*` community and the `*101059592*` grant ID, otherwise the GitHub–Zenodo integration will silently fail:

```
{ "communities": [ { "identifier": "b3" } ], "grants": [ { "id":
"101059592" } ]}
```

Further details on integrating GitHub and Zenodo using a .zenodo.json file are available in this tutorial (Vanderhaeghe 2022) and in the Zenodo developer documentation.

Always verify the Zenodo record after the first release to ensure the integration behaves as expected.

## Data products

The purpose of this section is to outline the requirements for software and scripts that are developed within the B-Cubed project. Data products are out-of-scope. However, many of the principles mentioned in this document can be applied to data products as well. For more details, we refer to the project deliverable D3.3 FAIR data products andinteroperability with the Essential Biodiversity Variables Data Portal (Estupinan-Suarez et al. 2025).

## Changelog

To communicate and explain version changes, each repository SHOULD have a changelog. This changelog SHOULD be expressed as a NEWS.md file for R code (see the rOpenSci recommendations, rOpenSci et al. 2024).

In R you can create a `NEWS.md` file using: `usethis::use_news_md()`

## R

R is a programming language for statistical computing and data visualization.

- General guidance on how to use and write R can be found in Adler 2012 and Crawley 2012.
- For R packages, we refer to the rOpenSci Packages guide (rOpenSci 2021).
- For analysis code, we refer to the Software Carpentry's fundamentals for reproducible scientific analysis in R (Zimmerman et al. 2019). More resources are listed on AGU's Introduction to Open Science (Vrouwenvelder et al. 2024). The British Ecological Society guide on reproducible code (Cooper et al. 2017) and Stoudt et al. 2021 list principles, while Sandve et al. 2013 has rules for reproducible data analysis. Specifically for Data Science we recommend R for Data Science (Wickham et al. 2023a).
- For more information on the use of RMarkdown, see R Markdown: The Definitive Guide (Xie et al. 2023) and Guidance for AGU Authors: R Script(s)/Markdown (Erdmann et al. 2021).

### RStudio projects

R code is run in a specific context, with an associated working directory, history, etc. If that context is undefined or too broad, it can create conflicts between projects or make it hard for others to run your code.

To solve this, software MUST make the context explicit by including an RStudio project file (file with `.Rproj` extension) in the root of the repository to make the context explicit. This file will set your and everyone's working directory at the root of the repository. In addition, software MUST only use relative paths starting at that project root to refer to files and MUST NOT use absolute paths. The implications of using absolute paths are described in the British Ecological Society guide on reproducible code (Cooper et al. 2017). R code should strive to be as portable as possible, for example by never referring to a drive letter, network location or storage mounting point.

Further benefits of RStudio Projects are described in this section of the R Packages book (Wickham and Bryan 2015). Software carpentry provides a guide on project management with Rstudio (Zimmerman et al. 2019).

### Dependencies

Please refer to the rOpenSci recommendations (rOpenSci et al. 2024) regarding dependencies.

Some recommendations for common use cases:

- HTTP requests: httr2 (Wickham 2025), curl (Ooms 2025a), crul (Chamberlain 2025)

- Parsing JSON: jsonlite (Ooms 2025b)
- Parsing XML: xml2 (Wickham et al. 2025a)
- Spatial data: sf (Pebesma 2018, Pebesma and Bivand 2023, Pebesma 2025), do note that rgdal (Bivand et al. 2022), rgeos (Bivand and Rundel 2020) and maptools (Bivand and Lewin-Koh 2019) are being deprecated, we thus advise against using sp (Pebesma 2025, Pebesma and Bivand 2005, Bivand et al. 2013, Pebesma and Bivand 2025). terra (Hijmans 2025) is preferred over raster (Hijmans 2025b), as it is being retired (see this blogpost by Nowosad (2023)). More information about the migration can be found on the r-spatial website (Bivand and Pebesma 2023), and this blogpost (Pebesma and Bivand 2022) about the retirement of sp, rgdal, maptools and rgeos.

In general, it is recommended to use packages from the tidyverse over base R functions in cases where the tidyverse alternative has significant advantages. For example, the case of `readr::read_csv()` over `base::read.table()`. The readr (Wickham et al. 2024a) alternative is faster, has better error handling, and is easier to use. However, certainly a case (Chamberlain 2018) can be made for having as few dependencies as possible, and wrapping your own functions around base to get around certain limitations. The question on when exactly you should take a dependency, depends on the context. The R Packages book (Wickham and Bryan 2015) offers some guidance in this matter. Leek (2015) has written a blog post on his decision process: How I decide when to trust an R package and the tidyverse (Pedersen 2022) makes a case for not using internal functions from dependencies.

Adding a dependency to your `DESCRIPTION` file is easy using usethis: `usethis::use_package("dplyr")`

Refer to the rOpenSci recommendations for more suggestions.

## Code style

*Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread*. --- R for Data Science

A number of useful packages exist to help you stick to the tidyverse style. To automatically modify your code to adhere to the recommendations, you can make use of styler (Müller and Walthert 2024) which also exists as a plug-in for RStudio. To check your code for issues, you can use a liter, a popular choice for R is lintr (Hester et al. 2025). More information regarding code style can be found in the rOpenSci Packages guide (rOpenSci et al. 2024) in the section on code style, the tidyverse style guide (The tidyverse team and Wickham 2025). Hadley Wickham (Wickham et al. 2023a) also offers some insight in his workflow when it comes to code style.

## Testing

Have you ever written any code that turned out to not really do what you wanted it to do? Made a change to a helper function that introduced bugs in other functions or scripts using it? Or found yourself running the same little ad hoc tests in the console time and time again to see if a function is behaving as expected? These are all signs that you could benefit from using automated testing.

By writing tests that check the major functionality of your software, you are ensuring that changes along the line don't break existing functionality. And that updates to underlying dependencies didn't have unexpected consequences.

A general overview of the how and why of testing R code is found in R Packages book chapter testing basics (Wickham and Bryan 2015). The rOpenSci Packages guide (rOpenSci et al. 2024) offers some helpful advice regarding tests in the section on testing. Other interesting resources include the blogpost by Lynch (2018) on why good developers write bad tests, the documentation of testthat (Wickham 2025b, Wickham 2011) and covr (Hester 2023). And rOpenSci also offers a book on HTTP testing (Chamberlain and Salmon 2024). For more information on unit testing in general, you might find "Unit Testing Principles, Practices, and Patterns" by Khorikov (2020) a good resource.

### Using testthat in practice

Start using testthat for an existing R project by running: `usethis::use_testthat()`

Which will create `tests/testthat/` and `tests/testthat.R`, and adds the testthat package to the *Suggests* field.

Creating a test for an existing function is automated via usethis:

```
# Explicitly refer to the file we want to test
usethis::use_test("filename_to_test")
# Or automatically if the file is already open and active in Rstudio
usethis::use_test()
```

While it is a good idea to regularly run your tests locally, it is also a good idea to automate this in the form of continuous integration:

```
# Runs R CMD CHECK which includes running all test
usethis::use_github_action("check-standard")
# Calculate the code coverage and report
usethis::use_github_action("test-coverage")
```

You can also add badges to your README page to signal your R CMD CHECK status (which includes your unit tests) and test coverage:

```
usethis::use_github_action("check-standard", badge = TRUE)
usethis::use_github_action("test-coverage", badge = TRUE)
```

Tests can then be run by:

```
# For a package
devtools::test()
# Or else, make sure your functions are loaded and then
testthat::test_dir("tests/testthat")
```

### Testing figures and plots

The goal of unit testing is to compare the output of a function to some expectation or expected value, however, for some outputs this isn't very practical. One example is binary outputs such as figures or plots. While testthat offers a solution for this in the form of snap shots (and this is certainly a very powerful and useful feature), these snapshot tests are very sensitive to minute changes.

vdiffr (Henry et al. 2024) is a package that forms an extension to testthat, it converts your visual outputs to reproducible svg files that are then compared as testthat snapshots. This offers some relief, but might still result in false positive test failures. After all, if your plotting library changes its rendering slightly, the test will fail.

A final option is to use a public accessor of your plotting library, for example ggplot2 (Wickham et al. 2025c, Wickham 2016) offers a number of these assessors that allow you to test specific parts of every layer. This post (Pedersen 2022) on the tidyverse blog offers more insight on why you might want to go about testing this way.

## Check your R code

There are several packages to check how well your R code is following best practices (from which many of the requirements in this document are derived):

- pkgcheck (Padgham et al. 2025): follows rOpenSci recommendations.
- checklist (Onkelinx 2025): works for R packages and analyses, follows INBO recommendations.
- lintr (Hester et al. 2025): performs static code analysis to highlight possible problems, including good practises and syntax.
- styler (Müller and Walthert 2024): can automatically format code according to the tidyverse style guide.
- goodpractice (Padgham et al. 2025b) informs about different good practices for packages.

- dupree (Hyde 2020) identifies sections of code that are very similar or repeated.

## R functions

There are a number of advantages to wrapping existing code into functions, as put by Tierney (2023) in his excellent blog post on how to get better at R:

*I don't think I can overstate this, but learning how to write functions changed how I think about code and how I think about solving problems.* --- Nicholas Tierney

This same blog post also contains a simple example of how to turn existing code into a function. A more in depth description can be found in the section on functions in R for Data Science (Wickham et al. 2023a).

To summarize the why, you should use functions to:

- Create more readable code by placing difficult to understand code into functions.
- Avoid errors when reusing (copy/pasting) code multiple times over the same analysis.

However, using functions is not without its pitfalls. But many issues can be avoided by sticking to some ground rules:

Functions need to be self-contained, the reasoning behind is explained well in the section "writing functions" in the British Ecological Society guide on reproducible code (Cooper et al. 2017). Practically this means:

- A function SHOULD NOT rely on data from outside of the function whenever possible.
- A function SHOULD NOT manipulate data outside of the function, thus it MUST NOT make changes to objects in the global environment. If you are importing data from the system to R, return an object rather than modifying the global environment (as is also explained in the tidyverse style guide, The tidyverse team and Wickham 2025).
- If it is necessary to make changes to data outside of the function, create a new file rather than making changes to an existing one. Functions that create new files MUST make this clear in their name, a good example is starting the function with write, for example `write_csv()` from readr (Wickham et al. 2024a).

Keeping your functions separate from analysis code can improve the readability of the analysis, and ease the maintenance of the functions. It is considered best practice to place every function in its own `.R` file, and to name this file after the function. As described in the R Packages book (Wickham and Bryan 2015). You can create such a file using: `usethis::use_r("function-name")`

And start your function using the included code snippet in RStudio.

A sure way to confuse users is for a function to return a different output with the same inputs/arguments; this is the case when some of the inputs are implicit. For example, an option or locale setting, the system time or a local datafile that might have changed. Using implicit arguments can lead to difficult to trace behaviours across different computers, and makes a function more difficult to read. A notable exception is when there is an element of randomisation in the output. In this case it is a good idea to allow for setting the seed as a function argument to make this behaviour clear for the user and to allow for reproducible results. For more information about this, read this excellent section in the tidyverse design principles (Wickham 2025c).

*One of the best ways to improve your reach as a data scientist is to write functions*. --- R for Data Science

Further reading:

- An excellent overview of how functions actually work in R can be found in the section on functions in R for Data Science (Wickham et al. 2023a), the rest of the section also includes an excellent overview of best practices.
- Nicholas Tierney provides an easy to follow example of how functions can make your life easier and how to get started with writing them in this blogpost (Tierney 2023).
- Principles and strategies that come in handy when writing functions (and packages) are summarized in the tidyverse design principles (Wickham 2025c).
- Berkeley offers an introduction to functions in its Introduction to the R Language presentation (Peng 2020a, Peng 2020b, Peng 2015).
- Software Carpentry has an excellent course page on R functions (Zimmerman et al. 2019).

## How to split a script into functions

The process of taking an existing script and converting it into a collection of functions that make the workflow more flexible, easier to maintain and more efficient, is an example of code refactoring.

An often repeated principle in refactoring and software development in general is DRY (Don't Repeat Yourself), and while there are certainly situations (Bartholomae 2020) where you should repeat yourself (see also AHA programming (Dodds 2020), arguments (Lynch 2018, Khorikov 2020b) for repeating yourself in unit tests), avoiding repetition makes your code easier to maintain and understand. Functions are the most obvious tool we have to avoid repetition, with the equally important benefit that they can offer serious documentation benefits and can make it easier for existing software to be used flexibly in the future.

Looking at an existing script, it is useful to consider what every part actually does (rubber duck debugging can be a useful technique in this). These logical sections and their substeps are good starting points.

Encapsulate repeated code blocks, or logical subsections that perform a single task (especially if they do it multiple times) into functions, and place objects that can influence the output as arguments. This can be a bit of a judgement call, but things like input file paths, output file paths, filters on the data such as taxonomy or time, number of bootstraps, random seeds etc. make for ideal argument choices. Often an object keeps being passed from section to section undergoing transformations on the way, and finally resulting in some output. If this is the case, the data object MUST be the first argument. For more guidance on this step, refer to the section on arguments.

The Research Institute for Nature and Forest (INBO) has a coding club session (Oldoni 2023) on functions that has practical exercises on how to turn an existing script into functions, and even finally a package. You can find this session here. Jennifer Bryan presented on "code smells" in 2018 during the useR conference (Bryan 2018). Code smells are a useful tool to identify parts of code that contain bad practices and are good candidates for refactoring. This presentation is available on YouTube.

## Naming functions

From the tidyverse style guide:

*There are only two hard things in Computer Science: cache invalidation and naming things.* --- Phil Karlton

Use verbs to name functions whenever possible, this is a clear indication that a function *does* something, in contrast to other objects. For more guidance please refer to the tidyverse style guide section on functions (The tidyverse team and Wickham 2025). Keep in mind that the name of the function should describe what it does as closely as possible.

If you find this difficult, consider if your function isn't doing too much. Ideally a function should only do one thing, and only return one thing.

## Function arguments

Consistent naming of arguments across functions greatly improves user friendliness. For guidance on object naming, please refer to this section of the tidyverse style guide (The tidyverse team and Wickham 2025).

In the same vein, it is best practice to place the most important arguments first, because these will be used first. This practice is covered by the tidyverse design principles (Wickham 2025c). Doing this, also signals to the user what arguments they should minimally provide. It is also a good idea to never provide defaults for required arguments, and always provide defaults for optional arguments, as covered by this tidyverse design principle. This pattern communicates to users which arguments are required, and which ones are not, without having to read the documentation.

Similarly, functions that return objects or data of the same type as their input, MUST place this input as their first argument. This also ensures that functions are as compatible with pipes as possible (in base R |> or Bache and Wickham (2022)'s magrittr %>%).

Other tidyverse design principles regarding function arguments:

- Keep defaults short and sweet
- Enumerate possible options
- Prefer a enum, even if only two choices

### Documenting functions

Functions that are well written can be considered partially self documenting, their name is an indication of what they do and their arguments tell the user what is expected and in what shape. However, apart from this, adding additional information will make it much easier for your future self and others to reuse your code. R comes with this functionality built in in the form of `.Rd` files in the man/ folder. Instead of creating these files manually, this additional documentation MUST be written in the form of roxygen2 (Wickham et al. 2024b) documentation, which takes the form of commented out text right above your function. The `.Rd` files are then rendered whenever you run: `devtools::document()`

It is recommended that you add at least one example of the basic functionality of your function in the roxygen2 documentation. A very minimal example that uses roxygen2 documentation is the `fct_rev()` function from forcats (Wickham 2023):

```
#' Reverse order of factor levels
#'
#' This is sometimes useful when plotting a factor.
#'
#' @param f A factor (or character vector).
#' @export
#' @examples
#' f <- factor(c("a", "b", "c"))
#' fct_rev(f)
fct_rev <- function(f) {
  f <- check_factor(f)
  lvls_reorder(f, rev(lvls_seq(f)))
}
```

An additional advantage of this system is that every function will automatically get its own page on your documentation website. A screenshot of the webpage that was created for the function above is shown in Fig. 1

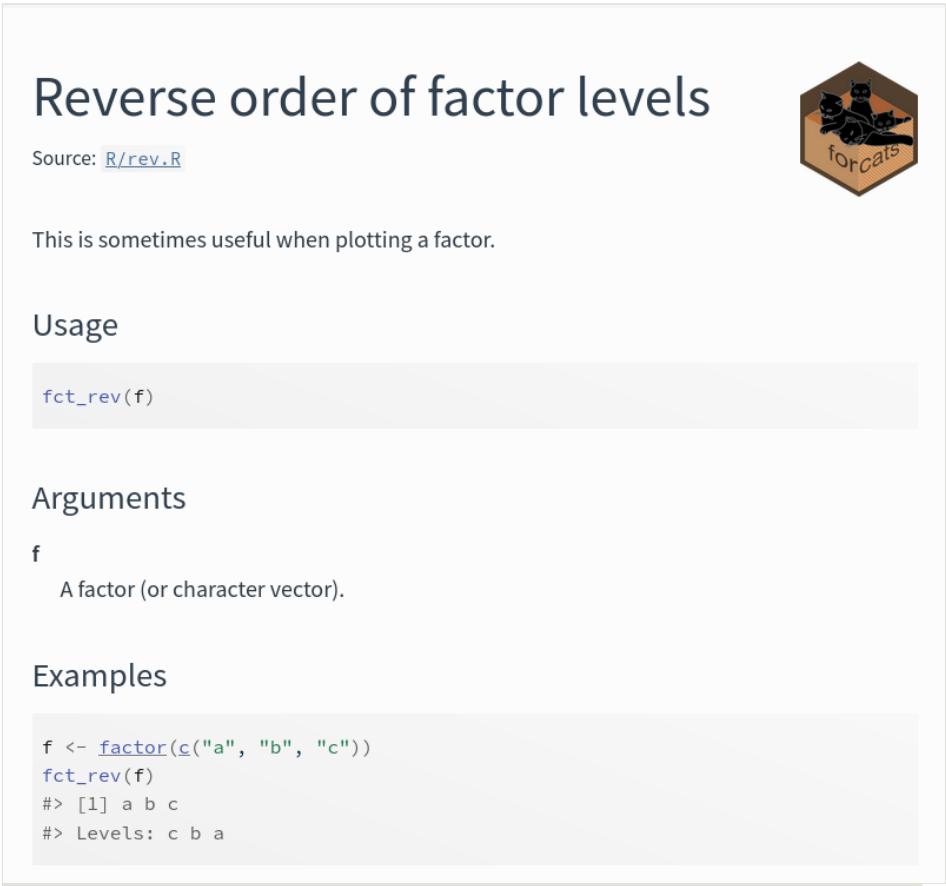


Figure 1.

Screenshot of the online documentation of the forcats function `fct_rev()` (licensed under MIT, Wickham 2023).

If you are new to documenting functions, have a look at the chapter on function documentation in the R Packages book (Wickham and Bryan 2015). There is also the getting started page of roxygen2 (Wickham et al. 2024b), and finally the rOpenSci Packages guide (rOpenSci et al. 2024) offers some advice in the section about documentation.

## R packages

Hadley Wickham and Jennifer Bryan have written an excellent guide on R Packages (Wickham and Bryan 2015), that comes highly recommended. This document goes through all the required steps to creating a package. More advanced is the R projects manual on writing R extensions (R Core Team et al. 2025). Hadley Wickham has included

sections on functional and object oriented programming in his book Advanced R (Wickham 2019b) that might come in useful.

A lot of the tooling around R packages is also useful for R analysis code formatted as a script. However, while it might look intimidating at first, authoring an R package isn't nearly as difficult as it might seem. Below there is an included example of R commands that set up an R package, and the required documentation, create a first function, tests for that function, update the documentation and run the package tests. All of this can be done equally well for a script, but this requires a lot more manual work.

```
# Starting a minimal R package --------
## Setup everything we need in a single line, isn't usethis amazing?
usethis::create_package("packagename")
## We will be using GIT for our version control
usethis::use_git()
# Further repository setup according to guidelines --------
## Add an MIT licence file
usethis::use_mit_license(copyright_holder = "institution name")
## Tell the world how to contribute
usethis::use_tidy_coc()
usethis::use_tidy_contributing()
## Write a README, in Rmd format if we want to show off our code
usethis::use_readme_rmd()
## Let's use github actions for some automation
usethis::use_github_action("check-standard", badge = TRUE)
usethis::use_github_action("test-coverage", badge = TRUE)
# Let's write our first function --------
usethis::use_r("cool_function_name")
## And add a test for it too!
usethis::use_testthat()
usethis::use_test()
## Update the documentation
devtools::document()
## Run all the tests
devtools::test()
```

### Naming your package

Naming a package or analysis script can be difficult. rOpenSci et al. (2024) offers a number of recommendations on this topic. To check if your package name is available, you can use the available package (Ganz et al. 2023), which can also inform you about possible other interpretations of the name, including possibly offensive ones.

```
available::available("mycoolpkgname")
```

Tierney (2018) also gives an interesting overview of trends in naming packages. Xie (2017) makes an excellent case for easy to type names without too many case changes.

## Creating metadata for your package

The CodeMeta project defines a `codemeta.json` metadata file (in JSON-LD format) that helps machines interpret information about your package. This is useful because it can ease the attribution, discoverability and reuse of your code beyond the tools already present in the R ecosystem. A `codemeta.json` makes it more likely someone will find your software who doesn't know where to look for it, and that you'll get credit for it when it is reused by allowing different metadata standards to be translated into each other via codemeta. The CodeMeta project makes a strong case for its inclusion in repositories. And so does rOpenSci (Boettiger and Salmon 2025).

Creating such a file is also very easy as it can be generated from the information already present in your `README`, `DESCRIPTION` and `CITATION` files. From the root of your package run: `codemetar::write_codemeta()`

## Console messages

Sometimes a package needs to communicate directly with its user; this is usually done through either `message()`, `warning()` or `stop()`. The rOpenSci Packages guide (rOpenSci et al. 2024) advises against using `print()` or `cat()` because these kinds of messages are much more difficult for the user to suppress. Additionally, these kinds of messages are also more difficult to write good tests for.

Apart from base R, the package cli (Csárdi et al. 2025) comes recommended for its many useful tools regarding good looking command line interfaces. Functions from cli also offer some advantages when used in assertions within functions over the popular assertthat (Wickham 2019) and `stopifnot()` from base. Please refer to the documentation of `cli_abort()` here. A practical example of how you could use cli instead of assertthat can be observed in this commit on the frictionless R package (Desmet et al. 2025).

## README

For general instructions, see the README file section. The `README` file for R packages largely takes the same form as the one required for all repositories. Additionally, a number of useful tools are available to you as a developer to create a great `README`.

If you don't have a README yet, you can create one with usethis: `usethis::use_readme_md()`

If you want to include code and its output in your `README`, you can instead create a `README.Rmd` in R markdown, and then render that to a traditional `README.md` file. Again, you can create one with usethis: `usethis::use_readme_rmd()`

This will not only create a `README.Rmd`, but also add some lines to `.Rbuildignore` and create a Git pre-commit hook to help remind you to keep `README.Rmd` and `README.md` synchronized. After you've made changes to `README.Rmd`, remember to update `README.md` by running: `devtools::build_readme()` via the devtools package (Wickham et al. 2022).

### Adding badges to the README file

Usethis (Wickham et al. 2024b) includes some useful functions you can use to add badges to your `README` file, for example for the lifecycle (Henry and Wickham 2023) of your software: `usethis::use_lifecycle_badge(stage = "stable")`

Some other functions within usethis (Wickham et al. 2024b) will also allow you to add a badge to your `README`, for example you can advertize your code coverage using the test-coverage action: `usethis::use_github_action("test-coverage", badge = TRUE)`

## Documentation website

A documentation website allows (potential) users to learn about your package and its functionality without having to install it first. Luckily, prior knowledge of web development is not needed to create a documentation website for R packages. It can be generated automatically with pkgdown (Wickham et al. 2025), which will pull the information you already included in the README file and function documentation. The introduction page of pkgdown describes its basic use (the documentation website of pkgdown was created with pkgdown). Here's how to get started:

```
# Run once to configure package to use pkgdown
usethis::use_pkgdown()
# Run to build the website
pkgdown::build_site()
```

By default, your website will include a homepage and a function reference, but there are several more pages you can add. The most useful ones are articles (called "vignettes") to explain a certain functionality or workflow in a more tutorial-like fashion:

```
# Create a vignette
usethis::use_vignette("vignette-title")
```

Since vignettes are included in the source code; users can also consult them when offline (in RStudio):

```
# Loading a vignette from dplyr (works offline too)
library(dplyr)
vignette("in-packages")
# This is the same page as https://dplyr.tidyverse.org/articles/in-
packages.html
```

Since we are using GitHub to host our code, deploying a website is fairly straightforward:

```
usethis::use_pkgdown_github_pages()
```

Neal Richardson posted a step-by-step guide to using pkgdown on his website (Richardson 2017). rOpenSci et al. (2024) also offers some guidance in their chapter on pkgdown.

## DESCRIPTION and authorship

The `DESCRIPTION` file includes, among others things, a list of all authors and contributors to the package. Apart from information about the authors, it also includes vital metadata about the version, licence and purpose of the included code. This file thus forms an important piece of metadata for the code it describes. This is also another place to refer to any external resources, such as the github repository where users can file bug reports or URLs to any external web APIs that may be called.

To uniquely identify the contributors to the software, it is very useful to include ORCID identifiers under the Authors in the description. The benefits of this are described on this rOpenSci blog post (Salmon 2018). An example:

```
Authors@R: person( "Pieter", "Huybrechts", email = "[email protected]",
role = c("aut", "cre"), comment = c(ORCID = "0000-0002-6658-6062"))
```

Further guidance on editing DESCRIPTION files can be found in the chapter on package metadata in the R Packages book (Wickham and Bryan 2015). A more detailed overview can be found in the R project manual on writing R extensions (R Core Team et al. 2025).

### CITATION

R packages commonly include a `CITATION` file (no extension) that provides information about how the package should be cited. See the CITATION file section in the rOpenSci Packages guide for guidance. This file can be created with:

```
usethis::use_citation()
```

And users can retrieve its information with:

```
citation("package-name")
```

All repositories MUST also include a `CITATION.cff` file (see Add a CITATION.cff file. You can keep it in sync with the `CITATION` file using a GitHub action provided by the cffr (Hernangómez 2021) package.

rOpenSci offers a useful blog post (LaZerte 2021) on how to cite R and R packages that is a good read for both software authors and users.

### LICENSE

As described in the Create a repository section, all software produced in the context MUST be licenced under the MIT licence. The copyright holder of the software will be the institution that will be maintaining the package, not the authors of the package.

Adding this LICENSE file is easy with usethis, Wickham et al. (2024b) (take care to immediately set the copyright holder, as it will default to the package authors):

```
usethis::use_mit_license(copyright_holder = "institution name")
```

This function will also set the License field in the DESCRIPTION file. For more information on package licensing, refer to the section on licensing in the R Packages book (Wickham and Bryan 2015).

### Examples

Examples show users how to use your software, and will often be the first thing people look at when they have trouble reusing your code. Thus including them not only fills an educational niche, but also provides a nice piece of documentation.

All functions intended to be used by users (i.e. public functions) MUST have an example in their roxygen2 (Wickham et al. 2024a) documentation. But even for analysis code or workflows, including an example can be very helpful. More complex examples (or example workflows) SHOULD be included in a vignette. Every repository that contains R code MUST at least have one vignette.

Creating a vignette is automated by usethis (Wickham et al. 2024), Keep in mind this will not work if you don't have a DESCRIPTION file:

```
usethis::use_vignette("vignette-title")
```

## Dependencies

Dependencies are other packages your package relies on. Those need to be defined in the DESCRIPTION file, so that they are automatically installed when a user instals your package. You can use usethis (Wickham et al. 2024) to add a dependency to your DESCRIPTION:

```
usethis::use_package("package-to-depend-on")
```

This will add a package to the *Imports* section of the `DESCRIPTION` file. The function also allows you to set it to *Suggests* instead, or to declare a minimum package version. Declaring a minimum version of a dependency isn't usually necessary and should only be done as a continuous choice. For more guidance on the tradeoffs and decisions around dependencies, read the section on package dependencies in the rOpenSci Packages guide (rOpenSci et al. 2024).

The difference between *Imports* and *Depends* is that while both are installed together with the package, *Depends* are also attached to the global environment, thus opening the door to all kinds of trouble. For example, if your package Depends on dplyr (Wickham et al. 2023) it will overwrite the stats function `filter()` which is loaded by default, because dplyr includes a `filter()` of its own. This kind of namespace conflict should be handled with caution, and avoided whenever possible. Code written by the user should behave as they expect, regardless of the order in which they load packages. Dependencies declared in *Imports* are not attached, thus avoiding this problem entirely. This principle and several other good practices are described in the section on package dependencies in the rOpenSci Packages guide (rOpenSci et al. 2024).

When calling a function from a dependency, the dependency MUST be explicitly mentioned using `package::function()`. This makes it easier for collaborators to understand your code and it helps when searching for functions of a specific dependency:

```
# Bad
my_function <- function(file) {read_csv(file)}
# Good
my_function <- function(file) {readr::read_csv(file)}
```

For dependency recommendations, see the dependencies section in the R section.

## R analysis code

An important note is that most R analysis scripts could be wrapped as a package. This has many advantages:

- Packages provide a better structure.
- Packages are easier to install and use.
- Packages allow for better documentation.
- It is much easier for others to reuse your work.
- There are a lot of tools that can help you make your work more reproducible that work better in the context of an R package.
- Within B-Cubed and the wider R community there are people ready to help, so if you've been waiting for an opportunity to learn: this is it.

Creating an R package might seem like a huge step if you haven't done it before, and while there is a learning curve, it really isn't nearly as hard as it seems. All of this to say, please don't be afraid to start an R package instead of an analysis script as part of your analysis workflow.

For more information on packages, refer to the R packages section.

An R analysis script/project can be started from scratch via usethis (Wickham et al. 2024c):

```
usethis::create_project("myprojectname")
```

This automates a number of steps:

- It creates a new directory for your project to live in.
- It sets the RStudio active project to the new folder.
- It creates a new subdirectory R/ for R code to live in.
- It creates an `.Rproj` file.
- It adds `.Rproj.user` to `.gitignore`.
- And finally it opens your new project in a new RStudio window.

As a next step you could initiate git:

```
usethis::use_git()
```

## Acknowledgements

We would like to thank the following people for their valuable insights, feedback and discussion in relation to this publication: Damiano Oldoni, Floris Vanderhaeghe, Laura Abraham, Maarten Trekels, Maëlle Salmon and Thierry Onkelinx.

## Funding program

This work was supported by the B3 (Biodiversity Building Blocks for policy) project, which has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No. 101059592.Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

## Conflicts of interest

The authors have declared that no competing interests exist.

## References

- Adler J (2012) R in a Nutshell. 2. O'Reilly URL: https://www.oreilly.com/library/view/r-in-a/9781449358204/
- Bache SM, Wickham H (2022) magrittr: A Forward-Pipe Operator for R. URL: https://magrittr.tidyverse.org
- Bartholomae D (2020) Moist code — Why code should not be completely DRY. https://startup-cto.medium.com/moist-code-why-code-should-not-be-completely-dry-1f06f2d31c31. Accessed on: 2025-8-05.
- Bivand R, Pebesma E, Gomez-Rubio V (2013) Applied spatial data analysis with R. 2. Springer, New York. [ISBN 978-1-4614-7618-4] https://doi.org/10.1007/978-1-4614-7618-4
- Bivand R, Lewin-Koh N (2019) maptools: Tools for Handling Spatial Objects. 0.9-9. URL: https://CRAN.R-project.org/package=maptools
- Bivand R, Rundel C (2020) rgeos: Interface to Geometry Engine - Open Source ('GEOS'). 0.5-5. URL: https://CRAN.R-project.org/package=rgeos
- Bivand R, Keitt T, Rowlingson B (2022) rgdal: Bindings for the 'Geospatial' Data Abstraction Library. 1.6.2. URL: https://CRAN.R-project.org/package=rgdal
- Bivand R, Pebesma E (2023) Progress on R-spatial evolution, Apr 2023. https://r-spatial.org/r/2023/04/10/evolution3.html. Accessed on: 2025-8-01.
- Boettiger C, Salmon M (2025) codemetar: Generate 'CodeMeta' Metadata for R Packages. 0.3.6. URL: https://docs.ropensci.org/codemetar
- Bradner S (1997) Key words for use in RFCs to Indicate Requirement Levels. Internet Engineering Task Force Best Current Practice 14 (2119). https://doi.org/10.17487/RFC2119
- Bryan J (2018) Code Smells and Feels. rstd.io/code-smells. Accessed on: 2025-8-05.
- Chamberlain S (2018) Limiting dependencies in R package development. https://recology.info/2018/10/limiting-dependencies/. Accessed on: 2025-8-05.
- Chamberlain S, Salmon M (2024) HTTP testing in R. rOpenSci Books URL: https://books.ropensci.org/http-testing/
- Chamberlain S (2025) crul: HTTP Client. 1.6.0. URL: https://docs.ropensci.org/crul/
- Cicirello V (2022) Badges - TL;DR for your repository's README. https://dev.to/cicirello/badges-tldr-for-your-repositorys-readme-3oo3. Accessed on: 2025-8-01.

- Cooper N, Hsing P, Croucher M, Graham L, James T, Krystall A, Michonneau F (2017) Reproducable Code. 1. British Ecological Society URL: https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf
- Crawley M (2012) The R Book. John Wiley & Sons [ISBN 9781118448908] https://doi.org/10.1002/9781118448908
- Csárdi G, Wickham H, Müller K, Brüggemann S, Software P (2025) *cli*: Helpers for Developing Command Line Interfaces. 3.6.3. URL: https://cli.r-lib.org
- Desmet P, Oldoni D, Huybrechts P, Govaert S, Research Institute for Nature and Forest (INBO) (2025) frictionless: Read and Write Frictionless Data Packages. 1.2.1. Release date: 2025-5-23. URL: https://doi.org/10.32614/CRAN.package.frictionless
- Dodds K (2020) AHA Programming. https://kentcdodds.com/blog/aha-programming. Accessed on: 2025-8-05.
- Erdmann C, Meyer M, Little J, Hondula K, Stachelek J, Oleksy I, Brousil M, Claborn K, Mesman J, Dennis T (2021) Guidance for AGU Authors: R Script(s)/Markdown. Advancing Earth and Space Science https://doi.org/10.5281/zenodo.5647997
- Estupinan-Suarez L, Quoss L, Langer C, Oceguera E, Pereira H (2025) D3.3 FAIR data products and interoperability with the Essential Biodiversity Variables Data Portal. https://b-cubed.eu/storage/app/uploads/public/68e/4f8/3ae/68e4f83aecb88828604715.pdf
- Ganz C, Csárdi G, Hester J, Lewis M, Tatman R, Posit Software, PBC (2023) available: Check if the Title of a Package is Available, Appropriate and Interesting. 1.1.0. URL: https://github.com/r-lib/available
- Govaert S, Desmet P, Huybrechts P, Langeraert W (2025) b3doc: Tools for the B-Cubed Documentation Website. v0.2.0. EU Open Research Repository. Release date: 2025-8-20. URL: https://doi.org/10.5281/zenodo.15649519
- Henry L, Wickham H (2023) *lifecycle*: Manage the Life Cycle of your Package Functions. 1.0.4. URL: https://lifecycle.r-lib.org/
- Henry L, Pedersen TL, Luciani TJ, Decorde M, Lise V (2024) vdiffr: Visual Regression Testing and Graphical Diffing. 1.0.8. URL: https://vdiffr.r-lib.org/
- Hernangómez D (2021) cffr: Generate Citation File Format Metadata for R Packages. Journal of Open Source Software 6 (67): 3900. https://doi.org/10.21105/joss.03900
- Hester J (2023) covr: Test Coverage for Packages. 3.6.4. URL: https://covr.r-lib.org
- Hester J, Angly F, Hyde R, Chirico M, Ren K, Rosenstock A, Patil I (2025) lintr: A 'Linter' for R Code. 3.2.0. URL: https://lintr.r-lib.org
- Hijmans R (2025a) terra: Spatial Data Analysis. 1.8-61. URL: https://rspatial.org/
- Hijmans R (2025b) raster: Geographic Data Analysis and Modeling. 3.6-32. URL: 10.32614/CRAN.package.raster
- Huybrechts P, Trekels M, Abraham L, Desmet P (2024) D3.1 Quality requirements for software. https://purews.inbo.be/ws/portalfiles/portal/150763204/D3.1_2024_Quality_requirements_for_software.pdf. Accessed on: 2025-8-31.
- Hyde R (2020) dupree: Identify Duplicated R Code in a Project. 0.3.0. 10.32614/CRAN.package.dupree. URL: https://github.com/russHyde/dupree
- Khorikov V (2020a) Unit Testing Principles, Practices, and Patterns. Manning, 304 pp. URL: https://www.manning.com/books/unit-testing [ISBN 9781617296277]
- Khorikov V (2020b) DRY vs DAMP in Unit Tests. https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/. Accessed on: 2025-8-05.

- LaZerte S (2021) How to Cite R and R Packages. https://ropensci.org/blog/2021/11/16/how-to-cite-r-and-r-packages/. Accessed on: 2025-8-01.
- Leek J (2015) How I decide when to trust an R package. https://simplystatistics.org/posts/2015-11-06-how-i-decide-when-to-trust-an-r-package/. Accessed on: 2025-8-05.
- Lynch M (2018) Why Good Developers Write Bad Unit Tests. https://mtlynch.io/good-developers-bad-tests/. Accessed on: 2025-8-05.
- Müller K, Walthert L (2024) styler: Non-Invasive Pretty Printing of R Code. 1.10.3. URL: https://styler.r-lib.org
- Nowosad J (2023) Upcoming changes to popular R packages for spatial data: what you need to do. www.r-bloggers.com/2023/06/upcoming-changes-to-popular-r-packages-for-spatial-data-what-you-need-to-do/. Accessed on: 2025-5-01.
- Oldoni D (2023) Functions in R. https://coding-club.inbo.be/sessions/20230926_functions_in_r.html#1. Accessed on: 2025-8-05.
- Onkelinx T (2025) checklist: A Thorough and Strict Set of Checks for R Packages and Source Code. Version 0.4.2. INBO https://doi.org/10.5281/zenodo.4028303
- Ooms J (2025a) *curl*: A Modern and Flexible Web Client for R. 6.99999. URL: https://jeroen.r-universe.dev/curl
- Ooms J (2025b) jsonlite: A Simple and Robust JSON Parser and Generator for R. 2.0.0. URL: https://arxiv.org/abs/1403.2805
- Padgham M, Salmon M, Wujciak-Jens J, Kim E (2025a) pkgcheck: rOpenSci Package Checks. 0.1.2.207. URL: https://docs.ropensci.org/pkgcheck/
- Padgham M, Marks K, Bortoli Dd, Csardi G, Frick H, Jones O, Alexander H (2025b) goodpractice: Advice on R Package Building. 1.0.5. URL: https://github.com/ropensci-review-tools/goodpractice
- Pebesma, Bivand (2005) Classes and methods for spatial data in R. R News 5 (2): 9-13. URL: https://CRAN.R-project.org/doc/Rnews/
- Pebesma E (2018) Simple Features for R: Standardized Support for Spatial Vector Data. The R Journal 10 (1): 439-446. https://doi.org/10.32614/RJ-2018-009
- Pebesma E, Bivand R (2022) R-spatial evolution: retirement of rgdal, rgeos and maptools. https://r-spatial.org/r/2022/04/12/evolution.html. Accessed on: 2025-8-01.
- Pebesma E, Bivand R (2023) Spatial Data Science: With applications in R. Chapman and Hall/CRC https://doi.org/10.1201/9780429459016
- Pebesma E (2025) sf: Simple Features for R. 1.0-21. 10.32614/CRAN.package.sf. URL: https://r-spatial.github.io/sf/
- Pebesma E, Bivand R (2025) sp: Classes and Methods for Spatial Data. 2.2-0. URL: https://edzer.github.io/sp/
- Pedersen TL (2022) Playing on the same team as your dependency. https://www.tidyverse.org/blog/2022/09/playing-on-the-same-team-as-your-dependecy/. Accessed on: 2025-8-05.
- Peng R (2015) Overview and History of R. https://www.youtube.com/watch?v=STihTnVSZnI. Accessed on: 2025-8-05.
- Peng R (2020a) Introduction to the R Language: Functions. https://rdpeng.github.io/Biostat776/lecture-functions.html. Accessed on: 2025-8-05.
- Peng R (2020b) Introduction to the R Language: Functions (presentation). https://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/functions.pdf. Accessed on: 2025-8-05.

- R Core Team, DebRoy S, Trapletti A (2025) Writing R Extensions. 4.5.1. Release date: 2025-6-13. URL: https://cran.rstudio.com/doc/manuals/r-release/R-exts.html
- Reitz K, Schlusser T (2016) The Hitchhiker's Guide to Python: Best Practices for Development. 1. O'Reilly, 336 pp. [ISBN 978-1491933176]
- Richardson N (2017) Getting Down with pkgdown. https://enpiar.com/2017/11/21/getting-down-with-pkgdown/. Accessed on: 2025-8-01.
- rOpenSci, Anderson B, Chamberlain S, DeCicco L, Gustavsen J, Krystalli A, Lepore M, Mullen L, Ram K, Ross N, Salmon M, Vidoni M, Riederer E, Sparks A, Hollister J (2024) rOpenSci Packages: Development, Maintenance, and Peer Review. URL: https://doi.org/10.5281/zenodo.10797633
- Salmon M (2018) Distinguish yourself in CRAN person() with ORCID. https://ropensci.org/blog/2018/10/08/orcid/. Accessed on: 2025-8-01.
- Sandve GK, Nekrutenko A, Taylor J, Hovig E (2013) Ten Simple Rules for Reproducible Computational Research. PLoS Computational Biology 9 (10). https://doi.org/10.1371/journal.pcbi.1003285
- Stoudt S, Vásquez V, Martinez C (2021) Principles for data analysis workflows. PLOS Computational Biology 17 (3). https://doi.org/10.1371/journal.pcbi.1008770
- The tidyverse team, Wickham H (2025) Tidyverse style guide. https://style.tidyverse.org/. Accessed on: 2025-8-05.
- Tierney N (2018) Naming Things. https://www.njtierney.com/post/2018/06/20/naming-things/. Accessed on: 2025-8-05.
- Tierney N (2023) How to Get Good with R? https://www.njtierney.com/post/2023/11/10/how-to-get-good-with-r/. Accessed on: 2025-8-05.
- Vanderhaeghe F (2022) Set up Zenodo - GitHub integration. https://tutorials.inbo.be/tutorials/git_zenodo/. Accessed on: 2025-8-01.
- Vaughan D (2023) Tidyteam code review principles. https://code-review.tidyverse.org/. Accessed on: 2025-8-01.
- Vrouwenvelder K, Hanson S, Sedora B (2024) Lesson Materials: Introduction to Open Science. Advancing Earth and Space Science https://doi.org/10.5281/zenodo.10957625
- Weigel BH (2023) How to Write a Good Issue: Tips for Effective Communication in Open Source. https://dev.to/opensauced/how-to-write-a-good-issue-tips-for-effective-communication-in-open-source-5443. Accessed on: 2025-8-01.
- Wickham H (2011) testthat: Get Started with Testing. The R Journal 3: 5-10. URL: https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf
- Wickham H, Bryan J (2015) R Packages. 2. O'Reilly, 202 pp. URL: https://r-pkgs.org/
- Wickham H (2016) ggplot2: Elegant Graphics for Data Analysis. Springer, New York. URL: https://ggplot2.tidyverse.org [ISBN 978-3-319-24277-4]
- Wickham H (2019a) assertthat: Easy Pre and Post Assertions. Release date: 2019-3-21. URL: https://doi.org/10.32614/CRAN.package.assertthat
- Wickham H (2019b) Advanced R. CRC, 588 pp. URL: https://adv-r.hadley.nz/ [ISBN 0815384572]
- Wickham H, Hester J, Chang W, Bryan J, RStudio (2022) *devtools*: Tools to Make Developing R Packages Easier. 2.4.5. URL: https://devtools.r-lib.org/
- Wickham H (2023) forcats: Tools for Working with Categorical Variables (Factors). 1.0.0. URL: https://forcats.tidyverse.org/
- Wickham H, Çetinkaya-Rundel M, Grolemund G (2023a) R for Data Science. 2. O'Reilly, 576 pp. URL: https://r4ds.hadley.nz/ [ISBN 978-1492097402]

- Wickham H, François R, Henry L, Müller K, Vaughan D (2023b) dplyr: A Grammar of Data Manipulation. 1.1.4. CRAN. Release date: 2023-11-17. URL: https://doi.org/10.32614/CRAN.package.dplyr
- Wickham H, Hester J, Bryan J (2024a) *readr*: Read Rectangular Text Data. 2.1.5. URL: https://readr.tidyverse.org
- Wickham H, Danenberg P, Csárdi G, Eugster M (2024b) roxygen2: In-Line Documentation for R. 7.3.2. URL: https://roxygen2.r-lib.org/
- Wickham H, Bryan J, Barrett M, Teucher A, Posit Software (2024c) usethis: Automate Package and Project Setup. 3.1.0. Release date: 2024-11-26. URL: https://doi.org/10.32614/CRAN.package.usethis
- Wickham H (2025a) httr2: Perform HTTP Requests and Process the Responses. 1.2.1. URL: https://httr2.r-lib.org
- Wickham H (2025b) testthat: Unit Testing for R. 3.2.3. URL: 10.32614/CRAN.package.testthat
- Wickham H (2025c) Tidy design principles. https://design.tidyverse.org/. Accessed on: 2025-8-05.
- Wickham H, Hester J, Ooms J (2025a) xml2: Parse XML. 1.3.8. URL: https://xml2.r-lib.org
- Wickham H, Hesselberth J, Salmon M, Roy O, Brüggemann S, Posit Software (2025b) *pkgdown: Make Static HTML Documentation for a Package*. 2.1.3. 10.32614/CRAN.package.pkgdown. Release date: 2025-5-25. URL: https://pkgdown.r-lib.org/
- Wickham H, Chang W, Henry L, Pedersen TL, Takahashi K, Wilke C, Woo K, Yutani H, Dunnington D, Brand Tvd (2025c) ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics. 3.5.2. URL: 10.32614/CRAN.package.ggplot2
- Xie Y (2017) One Little Thing To Consider When Naming Things (Software). https://yihui.org/en/2017/12/typing-names/. Accessed on: 2025-8-05.
- Xie Y, Allaire JJ, Grolemund G (2023) R Markdown: The Definitive Guide. 1. Chapman and Hall/CRC, New York, 338 pp. [ISBN 9781138359444] https://doi.org/10.1201/9781138359444
- Zimmerman N, Wilson G, Silva R, Ritchie S, Michonneau F, Oliver J, Dashnow H, Boughton A, Teucher A, Mawdsley D, MacDonald A, Rice T, Emonet R, Daigle R, Mills B, Bolker B, Penrose S, Sloggett C, Blischak J, Moore TE, Mawdsley D, Arnold J, Bridges D, Becker EA, Riva GVD, Ing-Simmons L, Research Bazaar, Bekolay T, Piaskowski J, Sze M, Hadley MJ, Hejazi N, Mayer F, Leinweber K, Deer L, Lesniak N, Burge OR, Martinez PA, Conrado AC, Jankevics A, Ashander J, Duckles J, Zappia L, Burle M, Mitchell N, Bouchet P, Harris RM, Renaut S, Sparks A, Attali D, Tyre D, Morrison E, McDonald G, Bar I, Mickley J, McDevitt-Irwin J, Koziar K, Samuk K, Marwaha K, Chatzidimitriou K, Chang L, Kardish M, Potter N, Boersch-Supan P, Funkhouser SA, Magle T, waiteb5, Khoja AA, Lee A, Berlanga-Taylor A, Srinath A, bippuspm, Beheim B, butterflyskip, Harris D, Oliveira DR, Balamuta J, Quan J, Woo K, Hertweck K, Ottoboni K, Weitemier K, Nederbragt L, Lonsdale A, Johnston LW, Frassl M, Dunning M, Donovan M, Clark M, Jackson M, Cadzow M, Raghupathy N, Sélem N, Bachant P, Banaszkiewicz P, Barnes R, Bagchi R, Brosda S, Munro S, Lavrentovich S, Rossum TV, Kelly TC, Hillis V, West KA, Takemon Y, Wright T, Zimmerman N, Oliver J, Mawdsley D (2019) Software Carpentry: R for Reproducible Scientific Analysis. v2019.06.1. Software Carpentry. Release date: 2019-6-01. URL: https://doi.org/10.5281/zenodo.3265163

## Endnotes

[*1] This guide was first created as B-Cubed project deliverable D3.1 Quality requirements for software (Huybrechts et al. 2024). While it was written for B-Cubed software, the suggestions and recommendations are general enough to be used by anyone who wants to improve their research software code.