BIODIVERSITY BUILDING BLOCKS FOR POLICY

D3.1 Quality requirements for software

28/02/2024

Author(s): Pieter Huybrechts, Maarten Trekels, Laura Abraham, Peter Desmet

Funded by the European Union

Prepared under contract from the European Commission
Grant agreement No. 101059592
EU Horizon Europe Research and Innovation Action

Project acronym:	**B3**
Project full title:	**Biodiversity Building Blocks for policy**
Project duration:	01.03.2023 – 31.08.2026 (42 months)
Project coordinator:	Dr. Quentin Groom, Agentschap Plantentuin Meise (MeiseBG)
Call:	HORIZON-CL6-2021-GOVERNANCE-01
Deliverable title:	Quality requirements for software
Deliverable n°:	D3.1
WP responsible:	WP3
Nature of the deliverable:	R: Document, Report
Dissemination level:	Public
Licence of use:	Creative Commons Attribution 4.0 International
Lead partner:	EV INBO
Recommended citation:	Huybrechts, P., Trekels, M., Abraham, L., Desmet, P. (2024). ***Quality requirements for software***. B3 project deliverable D3.1.
Due date of deliverable:	Month n°12
Actual submission date:	Month n°12

Deliverable status:

Version	Status	Date	Author(s)
1.0	Final/Draft	28 February 2024	Pieter Huybrechts (EV INBO), Maarten Trekels (MeiseBG), Laura Abraham (MeiseBG), Peter Desmet (EV INBO)

Table of contents

Key takeaway messages **6**
Executive summary **6**
Non-technical summary **6**
List of abbreviations **6**
1. Introduction **7**
2. Guidelines and requirements **8**
Code repositories 8
Create a repository 8
Set the copyright holder 10
Ignore Mac .DS_Store files 10
Add a CITATION.cff file 10
Add topics 11
Hide irrelevant tabs 11
Invite collaborators 11
Extend your README.md file 11
Setup your local environment, contribute code and collaborate 11
The README file 12
Format 12
Title 12
Badges 12
Description 13
Installation instructions 13
Examples or usage instructions 13
README files for data 13
Code collaboration 14
Add a Code of conduct 14
Enable notifications 15
Follow the GitHub flow 15
Protect the main branch 15
Contributing guide 16
Report issues 16
Local development 17
Versioning 18
Semantic versioning 18
Git commits 18
GitHub releases 19
Data products 19
Changelog 19
R 20

RStudio projects 21
Dependencies 21
Code style 22
Testing 22
Using testthat in practise 22
Testing figures and plots 24
Check your R code 24
R functions 25
How to split a script into functions 27
Naming functions 28
Function arguments 28
Documenting functions 29
R packages 31
Naming your package 33
Creating metadata for your package 33
Console messages 33
README 34
Adding badges to the README file 34
Documentation website 35
DESCRIPTION and authorship 36
CITATION 36
LICENSE 37
Examples 37
Dependencies 38
R analysis code 40
Python 42
Repository structure 43
Virtual environments 43
Dependencies 43
Code style 44
Use Explicit code 44
One statement per line 44
Line breaks with binary operations 44
Check your code against PEP 8 45
Testing 45
Packages 45
Documentation 45
Continuous integration with GitHub actions 46
Tutorials 47
Documenting software and code in B3 47
Creating a new tutorial 47

Writing your tutorial 48
Acknowledgements 49
References 50

Key takeaway messages

- Software produced by B3 partners should be open, functional, portable, and developed using best practices.
- This document lists the necessary requirements to achieve those goals. It also provides (links to) tutorials to help software developers meet those requirements.
- Special attention is given to software metadata (e.g. README, CITATION) and collaborative development (e.g. Github flow, Code of conduct).
- Many of the requirements focus on R, as it is the language in which most software will be developed in B3.
- Following the Data Management Plan (D1.3) all newly developed software will be open sourced under an MIT licence to encourage reuse.

Executive summary

This document provides quality requirements for the development of software within the B3 project. It covers maintaining and versioning software on GitHub, software metadata, guidelines for the R (general, packages and analyses) and Python programming languages, and how to provide user-friendly tutorials. The goal of the guidelines is to ensure the quality, openness, portability and reusability of the code produced within the project.

Non-technical summary

This document lists the expectations for software produced by B3 partners. Those requirements include practical instructions, examples and recommendations. By following the requirements, B3 partners will create high-quality software that can be used on several platforms.

List of abbreviations

API	Application Programming Interface
EBV	Essential Biodiversity Variable
EU	European Union
FAIR	Findable Accessible Interoperable and Reusable
GBIF	Global Biodiversity Information Facility
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSON-LD	JSON Linked Data
MIT	Massachusetts Institute of Technology
ORCID	Open Researcher and Contributor ID
PEP 8	Style Guide for Python Code
PIP	Package Installer for Python
R	Programming language for statistical computing and data visualization
RFC	Request for Comments
XML	Extensible Markup Language

1. Introduction

This document specifies high-level requirements for software, computational tools and resources developed for B3 (referred to further as only “software”) to ensure that the produced software meets the intended quality, openness, portability and reusability.

These requirements 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. A key aspect of such an ecosystem of packages is the interoperability of the data products they generate or consume, which will be specified in deliverable D3.3.

This document includes 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 and Python programming languages. The final chapter offers guidelines for the creation of tutorials for the produced software. At the head of every chapter an overview is offered that summarizes the minimal requirements (MUST as per RFC 2119). The text of the chapters can include additional recommendations (SHOULD, RECOMMENDED as per RFC 2119).

This guide will be further maintained at https://b-cubed-eu.github.io/documentation/.

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.

2. Guidelines and requirements

Code repositories

Lead author: Peter Desmet

All software code MUST be maintained on GitHub.
An installable software tool MUST be maintained in its own repository.
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.

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

Note: all steps below can be completed in the browser. For more information on GitHub terms, see the GitHub glossary.

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 B3 organization on GitHub. Email your GitHub username to GitHub B3 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:
    a. Choose `b-cubed-eu` as owner. If that option is not available, see step 2.
    b. The repository name SHOULD be lowercase, dash-separated and short.
    c. 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”

d. The visibility MUST be set to `public`. This makes it easier to collaborate and reference files and code.
e. Check `Add a README file`.
f. You MUST select a `.gitignore` template (e.g. R, Python)
g. You MUST select a licence and you MUST set it to `MIT License`. This conforms to the B3 Data Management Plan (Yovcheva et al. 2023).

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 B3 admin to gain the rights to transfer your repository to the `b-cubed-eu` organization.

Once you have created a repository (see Fig. 1), you SHOULD complete a number of additional steps.

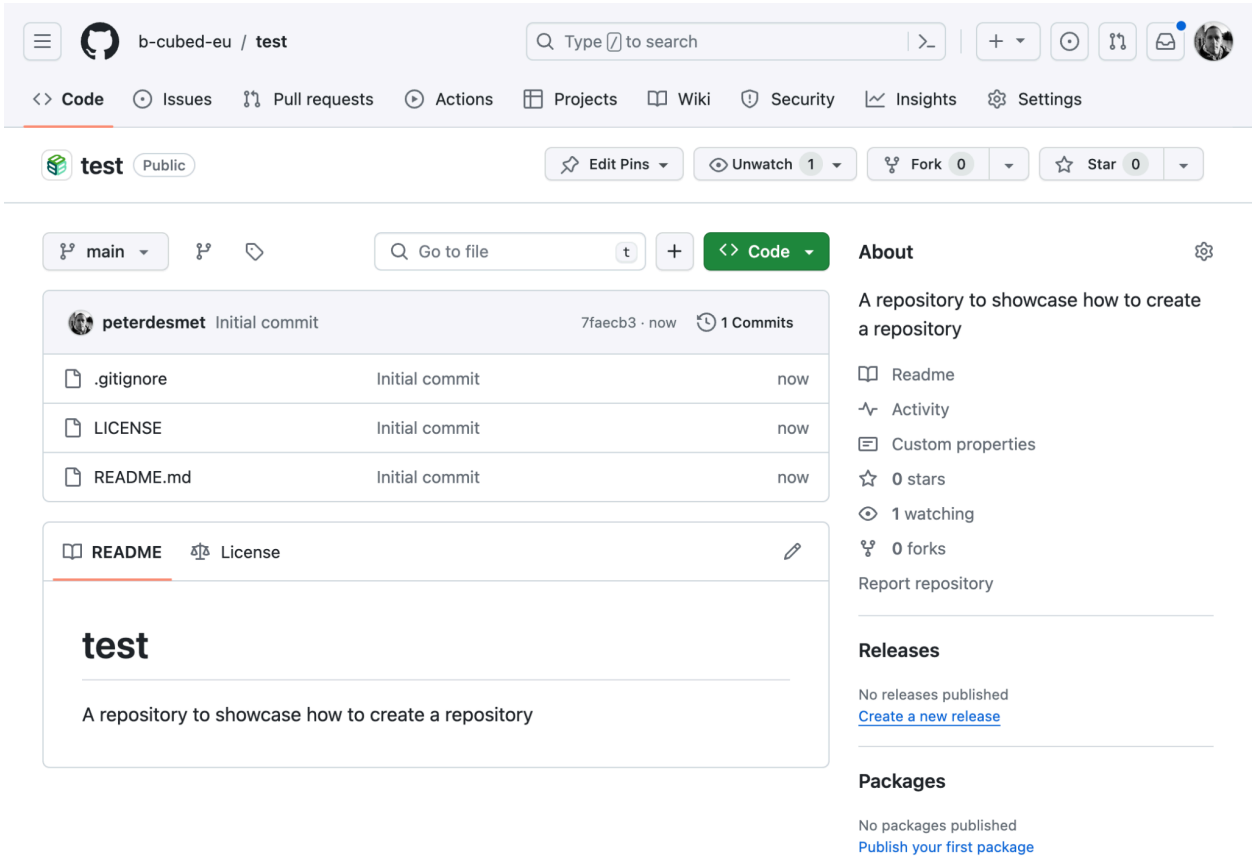


Figure 1: Screenshot of a newly created repository.

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):

```
Unset
# 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.
8. 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 rOpensci (2021).

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 B3 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 chapter.

Setup your local environment, contribute code and collaborate

See the Code collaboration chapter.

The README file

Lead author: Pieter Huybrechts

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.

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 & Oldoni 2022) as an example.

Note: all steps below can be completed in the browser. For more information on GitHub terms, see the GitHub glossary.

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. 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.

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 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

Lead author: Peter Desmet

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.
Maintainers MUST watch the repository they maintain.
Code contributions MUST follow the GitHub flow.
The main branch MUST contain the software code in a state that can be installed without issue.

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/<your-repo>/community`

Note: all steps below can be completed in the browser. For more information on GitHub terms, see the GitHub glossary.

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`: b-cubedsupport@meisebotanicgarden.be (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:

```
Unset
usethis::use_tidy_coc()
```

But update the default email address (codeofconduct@posit.co) in “Enforcement” to b-cubedsupport@meisebotanicgarden.be 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 B3 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 with the GitHub flow.

Alternatively, you can complete these steps in R using:

```
Unset
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 (BES 2019).

Writing a good issue takes skill, see this blog post or the tidyverse code review guide 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:

```
Unset
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

Lead author: Maarten Trekels

Software MUST use semantic versioning.
Major and minor versions MUST have an associated GitHub release.
Starting from version 1.0, all releases MUST be published to Zenodo.

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 could be the following:

```
Unset
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 for 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 for details.

Data products

The purpose of this document is to outline the requirements for software and scripts that are developed within the B3 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 upcoming deliverable “D3.3 Guidelines on the FAIR and open depositing of data products to ensure that B3 data cubes are compatible with the EBV Data Portal and other outlets for data cube dissemination”.

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).

In R you can create a NEWS.md file using:

```
Unset
usethis::use_news_md()
```

R

Lead author: Pieter Huybrechts

R code MUST be placed in the `R/` directory of the repository.
Data files included in the repository MUST be placed in the `data/` directory.
Repositories containing R code MUST include a project file (file with `.Rproj` extension) in the root.
R code MUST refer to files using relative paths and MUST NOT use absolute paths.
R code MUST NOT make use of the packages sp, rgdal, maptools, raster or rgeos but SHOULD use sf and/or terra.
R code MUST follow the rOpenSci recommendations regarding commonly used dependencies.
Dependencies on other packages MUST be declared in a DESCRIPTION file.
R code written MUST follow the tidyverse style guide.
R code MUST NOT make use of the right side assignment operator `->`.
All R code MUST reach a test coverage of at least 75% calculated using covr.
Unit tests MUST be implemented using the testthat package.
Shiny apps SHOULD make use of shinytest.
Unit tests MUST include the name of the R file they are testing.

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 (2010) and Crawley (2012).
- For R packages, we refer to the rOpenSci 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. A number of publications list principles (Stoudt et al. 2021, Croucher et al. 2017) and rules (Sandve et al. 2013) for reproducible data analysis. Specifically for Data Science we recommend R for Data Science (Wickham et al. 2023).
- For more information on the use of RMarkdown, see R Markdown: The Definitive Guide (Xie et al. 2018) and Guidance for AGU Authors: R Script(s)/Markdown.

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 (BES 2019). 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. Software carpentry provides a guide on project management with Rstudio.

Dependencies

Please refer to the rOpenSci recommendations regarding dependencies.

Some recommendations for common use cases:

- HTTP requests: httr2, curl, crul
- Parsing JSON: jsonlite
- Parsing XML: xml2
- Spatial data: sf, do note that rgdal, rgeos and maptools are being deprecated, we thus advise against using sp. terra is preferred over raster, as it is being retired (see this blogpost). More information about the migration can be found on the r-spatial website, and this blogpost 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 alternative is faster, has better error handling, and is easier to use. However, certainly a case 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 handbook (Wickham & Bryan 2023) offers some guidance in this matter. Jeff Leek has written a blog post on his decision process: "How I decide when to trust an R package" and the tidyverse makes a case for not using internal functions from dependencies.

Adding a dependency to your `DESCRIPTION` file is easy using usethis:

```
Unset
usethis::use_package("dplyr")
```

Refer to the rOpenSci recommendations for common scaffolding 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 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. More information regarding code style can be found in rOpenSci (2021) in the header on code style, the tidyverse style guide. Hadley (Wickham et al. 2023) 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 (Wickham & Bryan 2023) in chapter testing basics. rOpenSci (2021) offers some helpful advice regarding tests in the section on testing. Other interesting resources include the blogpost by Michael Lynch on why good developers write bad tests, the documentation of testthat and covr. And rOpenSci (Chamberlain & Salmon 2024) also offers a book on HTTP testing. 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 practise

Start using testthat for an existing R project by running:

```
Unset
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:

```
Unset
# 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:

```
Unset
# Runs R CMD CHECK which includes running all tests
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:

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

Tests can then be run by:

```
Unset
# 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 snapshots (and this is certainly a very powerful and useful feature), these snapshot tests are very sensitive to minute changes.

vdiffr 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 offers a number of these assessors that allow you to test specific parts of every layer. This post 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. 2023): follows rOpenSci recommendations.
- checklist (Onkelinx 2023): works for R packages and analyses, follows INBO recommendations.
- lintr (Hester et al. 2023): performs static code analysis to highlight possible problems, including good practises and syntax.
- styler (Müller & Walthert 2023): can automatically format code according to the tidyverse style guide.
- goodpractice (Marks et al. 2022) informs about different good practices for packages.
- dupree (Hyde 2024) identifies sections of code that are very similar or repeated.

R functions

Lead author: Pieter Huybrechts

Functions MUST NOT make changes to the global environment.
Functions that create or overwrite files MUST have a name that makes this clear such as `write_*`.
Repeated code MUST be placed in functions.
Functions MUST be named consistently across a package/analyses.
Functions MUST use snake_case for their name.
Functions MUST contain a verb as part of their name.
Exported functions in packages MUST have roxygen2 documentation.
Functions in analysis scripts MUST have roxygen2 documentation.
Functions in packages MUST have `@return` and `@examples`.
The output of a function MUST only depend on its arguments (inputs).
Each function MUST be stored in a separate .R file, except for helper functions.
Helper functions MUST be placed in `R/utils.R`.
Arguments MUST be named consistently across functions that use similar inputs.
Function arguments MUST be ordered from most important (and required) to least important (and optional).
If a function returns an object or data of the same type as its input, this argument MUST be in the first position.
Optional arguments MUST have default values, while required arguments MUST NOT have defaults.

There are a number of advantages to wrapping existing code into functions, as put by Nicholas Tierney 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. 2023).

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 (BES 2019). 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).
- 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.

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 & Bryan 2023). You can create such a file using:

```
Unset
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.

> *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. 2023), the rest of the chapter 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.
- Principles and strategies that come in handy when writing functions (and packages) are summarized in the tidyverse design principles.
- Berkeley offers an introduction to functions in its Introduction to the R Language presentation.
- Software Carpentry has an excellent course page on R functions.

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 where you should repeat yourself (see also AHA programming, arguments 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 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. 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 heading on functions. 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.

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. 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 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 documentation, which takes the form of commented out text right above your function. The `.Rd` files are then rendered whenever you run:

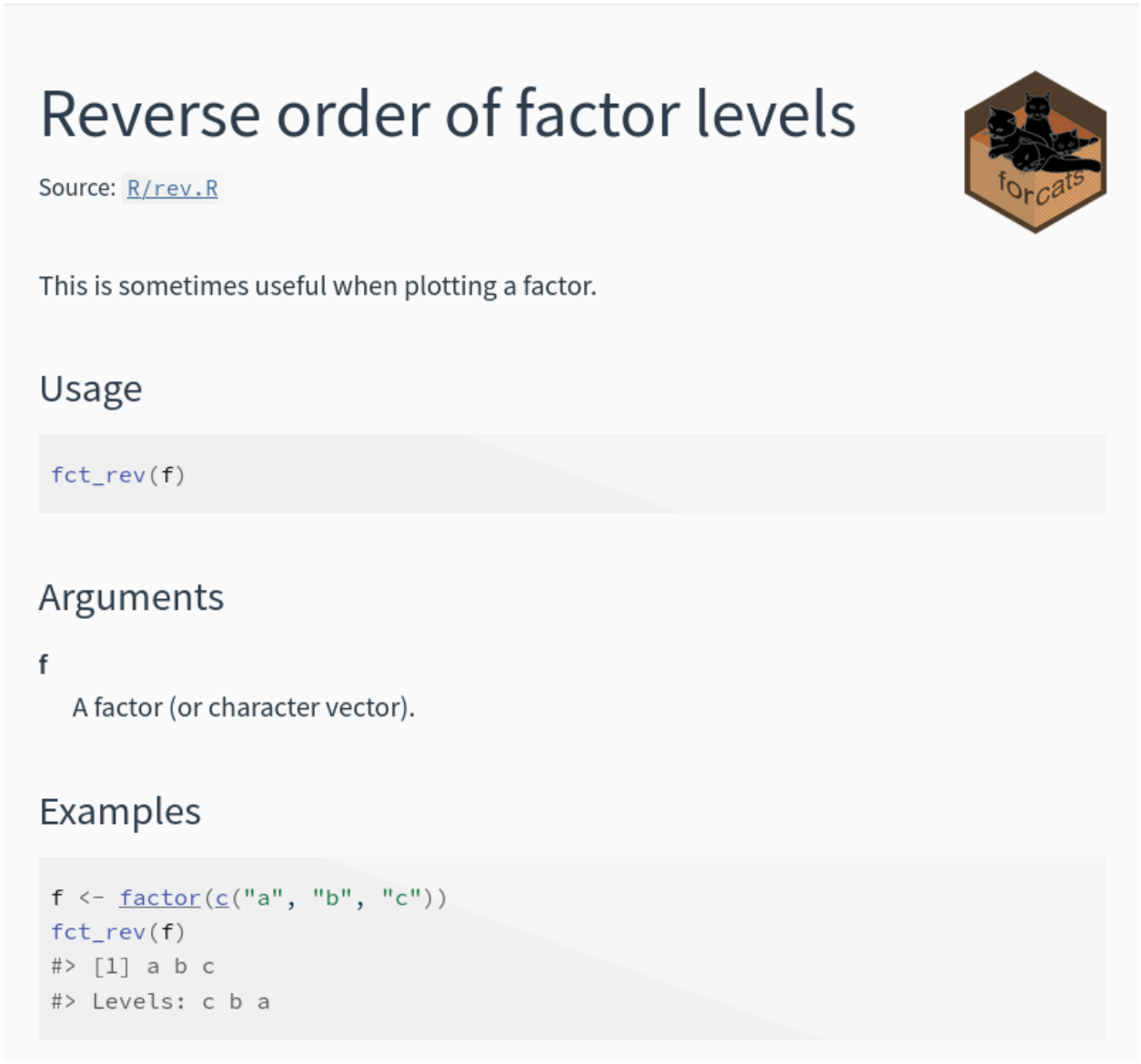
```
Unset
devtools::document()
```

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:

```
Unset
#' 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 Figure 2.



Reverse order of factor levels

Source: R/rev.R

This is sometimes useful when plotting a factor.

Usage

```
fct_rev(f)
```

Arguments

f

A factor (or character vector).

Examples

```
f <- factor(c("a", "b", "c"))
fct_rev(f)
#> [1] a b c
#> Levels: c b a
```

Figure 2: Screenshot of the online documentation of the forcats function `fct_rev()`.

If you are new to documenting functions, have a look at the chapter on function documentation in R packages (Wickham & Bryan 2023). There is also the getting started page of roxygen2, and finally rOpenSci (2021) offers some advice in the section about documentation.

R packages

Lead author: Pieter Huybrechts

R Packages MUST work on all major platforms: Windows, Linux and Mac.
R packages MUST include a `codemeta.json` in their repository.
R packages MUST pass R CMD CHECK without ERRORs.
Code included in a package MUST NOT use `print()` or `cat()`.
R packages MUST adhere to the tidyverse style guide.
Exported functions in R packages MUST be covered by a `testthat` unit test.
The package title MUST be available on CRAN.
The title of an R package MUST be in `Title Case`.
The title of an R package MUST NOT end in a period (`.`).
R packages MUST have a documentation website produced by pkgdown.
All authors MUST also include an ORCID identifier in the R authors `comment` field in the `DESCRIPTION` file.
The copyright holder (the institute that will be maintaining the software) MUST be added in the `Authors` field of the DESCRIPTION file.
The DESCRIPTION file MUST contain a URL in the `BugReports` field to the issues page of the repository.
All repositories that include R code MUST have at least one vignette with examples demonstrating its use.
Packages must declare their dependencies in the `DESCRIPTION` file.
Packages MUST NOT use `Depends` but instead MUST use `Imports` or `Suggests` to declare dependencies in the `DESCRIPTION` file.
When calling a function from a dependency, the dependency MUST be explicitly mentioned using `package::function()`.

Hadley Wickham and Jennifer Bryan have written an excellent guide on R Packages (Wickham & Bryan 2023), 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. Hadley Wickam (2019) has included sections on functional and object oriented programming in his book advanced R 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.

```
Unset
# Starting a minimal R package --------

## Setup everything we need in a single line, isn't usethis amazing?
usethis::create_package("packagetitle")

## 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()

## 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 offers a number of recommendations on this topic. To check if your package name is available, you can use the available package, which can also inform you about possible other interpretations of the name, including possibly offensive ones.

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

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

Creating metadata for your package

The codemeta project defines a metadata file: codemeta.json (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.

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:

```
Unset
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()`. rOpenSci (2021) 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 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 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.

README

See the README chapter. 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:

```
Unset
usethis::use_readme_md()
```

If you want to include code and its output in your REAMDE, 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:

```
Unset
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:

```
Unset
devtools::build_readme()
```

Adding badges to the README file

Usethis includes some useful functions you can use to add badges to your README file, for example for the lifecycle of your software:

```
Unset
usethis::use_lifecycle_badge(stage = "stable")
```

Some other functions within usethis will also allow you to add a badge to your README, for example you can advertize your code coverage using the test-coverage action:

```
Unset
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, 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:

```
Unset
# 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:

```
Unset
# 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):

```
Unset
# 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:

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

Neal Richardson posted a step by step guide on using pkgdown on his website. rOpenSci (2021) 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 which are described on this rOpenSci blog post. An example:

```
Unset
Authors@R: person(
 "Pieter", "Huybrechts",
 email = "pieter.huybrechts@inbo.be",
 role = c("aut", "cre"),
 comment = c(ORCID = "0000-0002-6658-6062")
)
```

Further guidance on editing `DESCRIPTION` files can be found in R packages (Wickham & Bryan 2023) in the chapter on package metadata. A more detailed overview can be found in the R project manual on writing R extensions.

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 of rOpenSci (2021) for guidance. This file can be created with:

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

And users can retrieve its information with:

```
Unset
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 package.

rOpenSci offers a useful blog post 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 chapter, 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 (take care to immediately set the copyright holder, as it will default to the package authors):

```
Unset
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 R packages book (Wickham & Bryan 2023) section on licensing.

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 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, Keep in mind this will not work if you don't have a DESCRIPTION file:

```
Unset
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 to add a dependency to your `DESCRIPTION`:

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

This will add a package to the `Imports` section of the `DESCRIPTION`. 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 rOpensci (2021).

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 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 rOpenSci (2021).

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:

```
Unset
# Bad
my_function <- function(file) {
  read_csv(file)
}
```

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

For dependency recommendations, see the general section on dependencies.

R analysis code

Lead author: Pieter Huybrechts

R analysis code MUST adhere to the proposed directory structure.
Data files MUST be placed in the `data` directory in the applicable subdirectory `raw`, `interim` or `processed`.
Any included files MUST adhere to the tidyverse style guide section on file names.
R code meant as an analysis workflow MUST be stored in `.Rmd` or `.R` format.

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 chapter.

An R analysis script/project can be started from scratch via usethis:

```
Unset
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:

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

Python

Lead author: Maarten Trekels

Code development MUST be done in a virtual environment.
The repository MUST contain a `requirements.txt` file.
Python code MUST adhere to the PEP 8 style guide.
Package and module names MUST be lowercase and short.
Class names MUST use CamelCase.
Indentation MUST be done using 4 spaces.
All Python code MUST reach a test coverage of at least 75% calculated using pytest-cov.
Unit tests MUST be implemented using the pytest package.
Documentation MUST be created using Sphinx.
Classes and functions MUST be documented using docstrings.

Many of the principles that are outlined in the chapters on R and R packages, also apply to writing Python code. In this chapter, we will outline some additional requirements for Python. A very good reference to Python programming can be found in The Hitchhicker's Guide to Python (Reitz & Schlusser 2016).

Repository structure

After creating a new code repository, the preferred structure for a repo named `sample` is as follows:

```
Unset
.gitignore
requirements.txt
README.md
LICENSE
Makefile
setup.py
pyproject.toml
sample/
    __init__.py
    core.py
    helpers.py
tests/
    tests_basic.py
    tests_advanced.py
docs/
    index.rst
    conf.py
    requirements.in
data/
    mydata.csv
CHANGE.md
CITATION.cff
.github/
    CODE_OF_CONDUCT.md
```

Virtual environments

You MUST use a virtual environment to develop your Python code. Virtual environments provide control over the version of Python and the installed packages. This will also make it easier to create a `requirements.txt`. There are several options to do this, but it is recommended to either use virtualenv or conda.

Dependencies

All Python projects MUST contain a `requirements.txt` file containing all dependencies of the code. A guide on the preparation of this requirements file can be found in this documentation. The requirements file guarantees that the code can be executed in a reproducible manner. Also, when creating a Python package, this allows PIP to install all dependencies together with the package.

Code style

Python code must adhere to the PEP 8 style guide for Python code. Some of the main elements that should be taken into consideration when writing code are outlined below.

Use Explicit code

The most explicit and straightforward way of coding is preferred. E.g.

```python
# Bad
def make_complex(*args):
    x, y = args
    return dict(**locals())

# Good
def make_complex(x, y):
    return {'x': x, 'y': y}
```

One statement per line

Although in some particular cases it might be reasonable to have multiple statements per line, in general this is bad practice to have more than one disjointed statement on one line:

```python
# Bad
print("one"); print("two")

# Good
print("one")
print("two")
```

Line breaks with binary operations

In order to improve the readability of the code, it is recommended to use line breaks before the binary operator:

```python
# Bad
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

```
# Good
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Check your code against PEP 8

It is recommended that each piece of Python code is checked using pycodestyle. Your code can be checked by using:

```
Python
pycodestyle --first yourcode.py
```

For more advanced usage of the package, please refer to its documentation website.

Testing

In general, testing should be performed on small units of functionality. As a good practice, it is recommended that each function has a corresponding test associated with. All testing MUST be performed using the pytest package. Several guidelines on using the package are available on its documentation website.

Another good practice is to include a test for each bug that is/was present in the code. In that case, please refer to the corresponding bug report in the test documentation.

Packages

Although there might be several use cases where it is sufficient to develop a Python module (single file), it is RECOMMENDED to package your code into a Python package. A comprehensive guide to creating a Python package can be found here. When adhering to the recommended repository structure, many of the requirements for a Python package are covered.

Documentation

Documentation for your packages MUST be created using Sphinx. Sphinx is a very powerful documentation generator tool which is widely used within the Python community.

The way Classes and functions are documented is using docstrings. A Sphinx docstring has the following structure:

```
Python
"""[Summary]

:param [ParamName]: [ParamDescription], defaults to [DefaultParamVal]
:type [ParamName]: [ParamType](, optional)
...
:raises [ErrorType]: [ErrorDescription]
...
:return: [ReturnDescription]
:rtype: [ReturnType]
"""
```

Continuous integration with GitHub actions

GitHub Actions SHOULD be used to test, build and release your Python packages. A step-by-step guide to publish your releases can be found here.

Tutorials

Lead author: Laura Abraham

Each package and analysis MUST have at least one tutorial.
Tutorials MUST be included in the B3 documentation website.
Tutorials MUST be written in English using literate programming documents.

To make software more welcoming to users, each package and analysis MUST have at least one tutorial guiding users through its main functionality. These tutorials MUST be included (copied or referenced) in the B3 documentation website.

Documenting software and code in B3

Tutorials MUST be written in English and presented as literate programming documents (e.g. Jupyter notebooks, RMarkdown or Quarto) to provide both narrative context and executable code snippets. The documentation website will be versioned and an automated testing mechanism will be set up to guarantee that provided documentation works for a specific release of the B3 toolbox. Ensure that your tutorial passes automated tests.

Creating a new tutorial

1. Create a new branch in https://github.com/b-cubed-eu/documentation following the Github flow.
2. Go to the `tutorials` folder in the documentation repository or use this link.
3. Click `Add file` and then `Create new file`.
4. Name your file `name-of-tutorial/index.md`. Use lowercase and dashes (`create-occurrence-cube/index.md`).
5. Start your Markdown file with front matter:

```
Unset
---
title: [Your tutorial title]
description: [Short description of your tutorial]
authors:
  - name: [Author name]
    orcid: [Author ORCID]
date: [YYYY-MM-DD]
categories: [category]
source: [url]
---
```

6. Replace `[Your tutorial title]` with the actual title of your tutorial, provide a description and fill in the author's information. You can include multiple authors by adding additional items under the `authors` field. The `date` field should be filled with the publication or last modification date and the `categories` field can be customised based on the content of your tutorial. The `source` is the URL of your tutorial if it is maintained elsewhere.
7. Commit the changes.

You now have a directory for your tutorial, which can contain any files (images, small datasets, reproducible notebook) related to your tutorial. The `index.md` will serve as the public page for your tutorial.

Writing your tutorial

You can write your tutorial directly in the `index.md`, but if it includes code snippets, it is RECOMMENDED to write it as a reproducible R Markdown, Quarto or Jupyter Notebook. This makes it easier to run and test (cf. a README.Rmd over a README.md).

Such files can then be rendered to HTML/Markdown, and will not only include the text and the code snippets, but also the results of running the code (example). That rendered HTML/Markdown can be copied to `index.md`, under the frontmatter.

As for the content of your tutorial:

- Clearly state the purpose of the tutorial.
- Include step-by-step instructions on how to install the software.
- Specify dependencies and system requirements if applicable.
- Detail how to use the software.
- Include at least one example and explain key features.
- Write in a clear and concise manner for a diverse audience.

Once your tutorial is ready, submit your branch as a pull request for review.

Acknowledgements

The authors would like to acknowledge the contributions of Damiano Oldoni who's insight has been helpful in the development of these guidelines. His ongoing commitment and dedication to open science and open source development have been invaluable to this work.

References

The reference list also includes all referenced software packages.

Adler J (2010). *R in a nutshell: A desktop quick reference*. O'Reilly Media, Inc.

Bache S, Wickham H (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.3. https://CRAN.R-project.org/package=magrittr

Bivand R, Pebesma E, Gomez-Rubio V (2013). *Applied spatial data analysis with R, Second edition*. Springer, NY. https://asdar-book.org

Bivand R, Keitt T, Rowlingson B (2023). *rgdal: Bindings for the 'Geospatial' Data Abstraction Library*. R package version 1.6-7. https://CRAN.R-project.org/package=rgdal

Bivand R, Lewin-Koh N (2023). *maptools: Tools for Handling Spatial Objects*. R package version 1.1-8. https://CRAN.R-project.org/package=maptools

Bivand R, Rundel C (2023). *rgeos: Interface to Geometry Engine - Open Source ('GEOS')*. https://r-forge.r-project.org/projects/rgeos/ https://libgeos.org http://rgeos.r-forge.r-project.org/index.html

British Ecological Society, Croucher M, Graham L, James T, Krystalli A, Michonneau F (2017). *Reproducible code*. https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf

Chamberlain S, Salmon M (2024). *HTTP testing in R*. rOpenSci. https://doi.org/10.5281/zenodo.10608847 https://books.ropensci.org/http-testing/

Chang W, Csárdi G, Wickham H (2023). *shinytest: Test Shiny Apps*. R package version 1.5.3. https://CRAN.R-project.org/package=shinytest

Crawley MJ (2012). *The R book*. John Wiley & Sons.

Csárdi G (2023). *cli: Helpers for Developing Command Line Interfaces*. R package version 3.6.2. https://CRAN.R-project.org/package=cli

Desmet P, Oldoni D (2022). *frictionless: Read and Write Frictionless Data Packages*. R package version 1.0.2. https://cran.r-project.org/package=frictionless

Ganz C, Csárdi G, Hester J, Lewis M, Tatman R (2022). *available: Check if the Title of a Package is Available, Appropriate and Interesting*. R package version 1.1.0. https://CRAN.R-project.org/package=available

Henry L, Pedersen T, Luciani T, Decorde M, Lise V (2023). *vdiffr: Visual Regression Testing and Graphical Diffing*. https://vdiffr.r-lib.org/, https://github.com/r-lib/vdiffr

Henry L, Wickham H (2023). *lifecycle: Manage the Life Cycle of your Package Functions*. R package version 1.0.4. https://CRAN.R-project.org/package=lifecycle

Hester J (2023). *covr: Test Coverage for Packages*. R package version 3.6.4. https://CRAN.R-project.org/package=covr

Hester J, Angly F, Hyde R, Chirico M, Ren K, Rosenstock A, Patil I (2023). *lintr: A 'Linter' for R Code*. R package version 3.1.1. https://CRAN.R-project.org/package=lintr

Hijmans R (2023). *raster: Geographic Data Analysis and Modeling*. R package version 3.6-26, https://CRAN.R-project.org/package=raster

Hijmans R (2023). *terra: Spatial Data Analysis*. R package version 1.7-65, https://CRAN.R-project.org/package=terra

Hyde R (2024). *dupree: Identify Duplicated R Code in a Project*. R package version 0.3.0, https://russhyde.github.io/dupree/, https://CRAN.R-project.org/package=dupree

Khorikov V (2020). *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

Marks K, de Bortoli D, Csardi G, Frick H, Jones O, Alexander H (2022). *goodpractice: Advice on R Package Building*. R package version 1.0.4, https://CRAN.R-project.org/package=goodpractice

Müller K, Walthert L (2023). *styler: Non-Invasive Pretty Printing of R Code*. R package version 1.10.2, https://CRAN.R-project.org/package=styler

Onkelinx, T (2023) checklist: A Thorough and Strict Set of Checks for R Packages and Source Code. Version 0.3.5. https://inbo.github.io/checklist/

Padgham M, Salmon M, Wujciak-Jens J (2023). *pkgcheck: rOpenSci Package Checks*. https://docs.ropensci.org/pkgcheck/, https://github.com/ropensci-review-tools/pkgcheck

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 (2005). *Classes and methods for spatial data in R*. R News, 5(2), 9-13. https://CRAN.R-project.org/doc/Rnews/

Pebesma E, Bivand R (2023). *Spatial Data Science: With Applications in R*. Chapman and Hall/CRC. https://doi.org/10.1201/9780429459016

R Core Team (2022). *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

Rajlich VT, Bennett KH (2000). *A staged model for the software life cycle*. Computer, 33(7), 66-71. https://doi.org/10.1109/2.869374

Reitz K, Schlusser T (2016). *The Hitchhiker's guide to Python: best practices for development*. O'Reilly Media, Inc. https://docs.python-guide.org/

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 (2021). *rOpenSci Packages: Development, Maintenance, and Peer Review* (0.7.0). https://doi.org/10.5281/zenodo.6619350

Sandve GK, Nekrutenko A, Taylor J, Hovig E (2013). *Ten Simple Rules for Reproducible Computational Research*. PLoS Comput Biol, 9(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Stoudt S, Vásquez VN, Martinez CC (2021). *Principles for data analysis workflows*. PLoS Comput Biol, 17(3), e1008770. https://doi.org/10.1371/journal.pcbi.1008770

Wickham H (2011). *testthat: Get Started with Testing*. The R Journal, 3, 5-10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf

Wickham H (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham H (2019). *Advanced r*. CRC press. https://adv-r.hadley.nz/

Wickham H (2023). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 1.0.0. https://CRAN.R-project.org/package=forcats

Wickham H, Bryan J (2023). *R packages*. O'Reilly Media, Inc. https://r-pkgs.org/

Wickham H, Bryan J, Barrett M, Teucher A (2023). *Usethis: Automate Package and Project Setup*. R package version 2.2.2. https://CRAN.R-project.org/package=usethis

Wickham H, Çetinkaya-Rundel M, Grolemund G (2023). *R for data science*. O'Reilly Media, Inc. https://r4ds.hadley.nz/

Wickham H, Danenberg P, Csárdi G, Eugster M (2022). *roxygen2: In-Line Documentation for R*. R package version 7.2.3. https://CRAN.R-project.org/package=roxygen2

Wickham H, François R, Henry L, Müller K, Vaughan D (2023). *dplyr: A Grammar of Data Manipulation*. R package version 1.1.4. https://CRAN.R-project.org/package=dplyr

Wickham H, Hesselberth J, Salmon M (2022). *pkgdown: Make Static HTML Documentation for a Package*. R package version 2.0.7. https://CRAN.R-project.org/package=pkgdown

Wickham H, Hester J, Bryan J (2023). *readr: Read Rectangular Text Data*. R package version 2.1.4. https://CRAN.R-project.org/package=readr

Wickham H, Hester J, Chang W, Bryan J (2022). *Devtools: Tools to Make Developing R Packages Easier*. R package version 2.4.5. https://CRAN.R-project.org/package=devtools

Yovcheva N, Metodiev T, Stoev P, Ruffino FR, Castro FJ (2023). *Data Management Plan*. B3 project deliverable D1.3. https://b-cubed.eu/storage/app/uploads/public/64e/f45/6cd/64ef456cd4da1356663578.pdf

Xie Y, Allaire JJ, Grolemund G (2018). *R markdown: The definitive guide*. CRC Press. https://bookdown.org/yihui/rmarkdown/

Zimmerman N, Wilson G, Silva R, Ritchie S, Michonneau F, Oliver J, …, Takemon Y (2019, July). swcarpentry/r-novice-gapminder: *Software Carpentry: R for Reproducible Scientific Analysis*, June 2019 (Version v2019.06.1). http://doi.org/10.5281/zenodo.3265164