rOpenSci Packages: Development, Maintenance, and Peer Review

rOpenSci software review editorial team (current and alumni): Brooke Anderson, Scott Chamberlain,

2023-09-19

Contents

```
@software{ropensci_2021_6619350,
  author       = {rOpenSci and
                  Anderson, Brooke and
                  Chamberlain, Scott and
                  DeCicco, Laura and
                  Gustavsen, Julia and
                  Krystalli, Anna and
                  Lepore, Mauro and
                  Mullen, Lincoln and
                  Ram, Karthik and
                  Ross, Noam and
                  Salmon, Maëlle and
                  Vidoni, Melina and
                  Riederer, Emily and
                  Sparks, Adam and
                  Hollister, Jeff},
  title        = {{rOpenSci Packages: Development, Maintenance, and
                   Peer Review}},
  month        = nov,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {0.7.0},
  doi          = {10.5281/zenodo.6619350},
  url          = {https://doi.org/10.5281/zenodo.6619350}
}
```

You can also read the PDF version of this book.

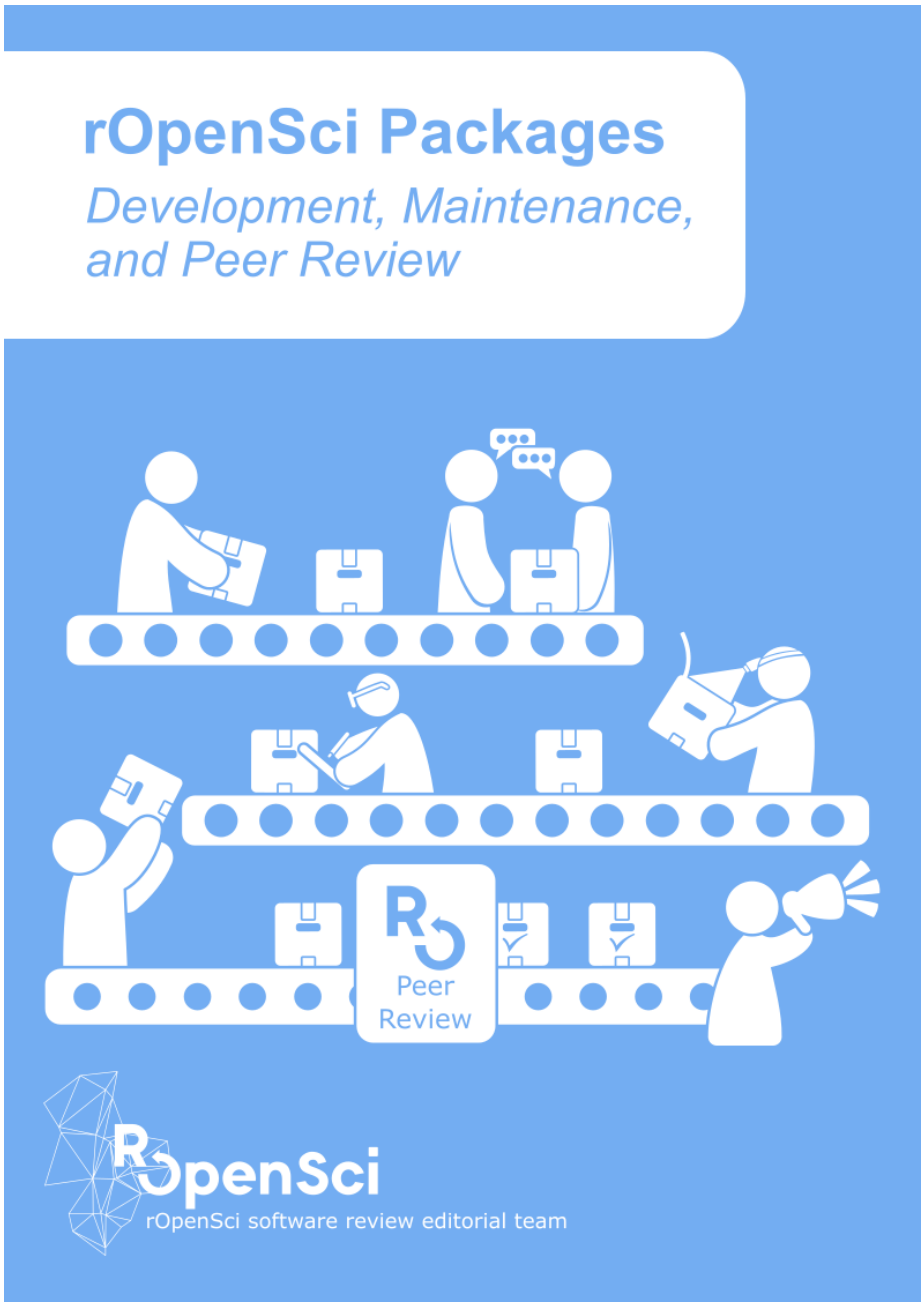


Figure 1: cover image

Preface

Welcome! This book is a guide for authors, maintainers, reviewers and editors of rOpenSci.

The first section of the book contains our guidelines for creating and testing R packages.

The second section is dedicated to rOpenSci's software peer review process: what it is, our policies, and specific guides for authors, editors and reviewers throughout the process. For *statistical software review*, refer to the project webpage and resources.

The third and last section features our best practice for nurturing your package once it has been onboarded: how to collaborate with other developers, how to document releases, how to promote your package and how to leverage GitHub as a development platform. The third section also features a chapter for anyone wishing to start contributing to rOpenSci packages.

We hope that you'll find the guide useful and clear, and welcome your suggestions in the issue tracker of the book. Happy R packaging!

The rOpenSci editorial team.

This book is a living document. You can view updates to our best practices and policies via the release notes.
You can cite this book using its Zenodo metadata and DOI.

```
@software{ropensci_2021_6619350,
  author       = {rOpenSci and
                  Anderson, Brooke and
                  Chamberlain, Scott and
                  DeCicco, Laura and
                  Gustavsen, Julia and
                  Krystalli, Anna and
                  Lepore, Mauro and
                  Mullen, Lincoln and
                  Ram, Karthik and
                  Ross, Noam and
                  Salmon, Maëlle and
```

```
                  Vidoni, Melina and
                  Riederer, Emily and
                  Sparks, Adam and
                  Hollister, Jeff},
  title        = {{rOpenSci Packages: Development, Maintenance, and
                   Peer Review}},
  month        = nov,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {0.7.0},
  doi          = {10.5281/zenodo.6619350},
  url          = {https://doi.org/10.5281/zenodo.6619350}
}
```

If you want to contribute to this book (suggestions, corrections) please refer to the GitHub repository in particular the contributing guidelines. Thanks!

We are thankful for all authors, reviewers and guest editors for helping us improve the system and this guide over the years. Thanks also to the following persons who made contributions to this guide and its previous incarnations: Katrin Leinweber, John Baumgartner, François Michonneau, Christophe Dervieux, Lorenzo Busetto, Ben Marwick, Nicholas Horton, Chris Kennedy, Mark Padgham, Jeroen Ooms, Sean Hughes, Jan Gorecki, Joseph Stachelek, Dean Attali, Julia Gustavsen, Nicholas Tierney, Rich FitzJohn, Tiffany Timbers, Hilmar Lapp, Miles McBain, Bryce Mecum, Jonathan Carroll, Carl Boettiger, Florian Privé, Stefanie Butland, Daniel Possenriede, Hadley Wickham, Mauro Lepore, Matthew Fidler, Luke McGuinness, Aaron Wolen, Indrajeet Patil, Kevin Wright, Will Landau, Hugo Gruson, Hao Ye, Sébastien Rochette. Please tell us if we forgot to acknowledge your contribution!

Part I

Building Your Package

Chapter 1

Packaging Guide

rOpenSci accepts packages that meet our guidelines via a streamlined Software Peer Review process. To ensure a consistent style across all of our tools we have written this chapter highlighting our guidelines for package development. Please also read and apply our chapter about continuous integration (CI). Further guidance for after the review process is provided in the third section of this book starting with a chapter about collaboration.

We strongly recommend that package developers read Hadley Wickham and Jenny Bryan's concise but thorough book on package development which is available for free online. Our guide is partially redundant with other resources but highlights rOpenSci's guidelines.

To read why submitting a package to rOpenSci is worth the effort to meet guidelines, have a look at reasons to submit.

1.1 Package name and metadata

1.1.1 Naming your package

- We strongly recommend short, descriptive names in lower case. If your package deals with one or more commercial services, please make sure the name does not violate branding guidelines. You can check if your package name is available, informative and not offensive by using the `available` package. In particular, do *not* choose a package name that's already used on CRAN or Bioconductor.

- A more unique package name might be easier to track (for you and us to assess package use) and search (for users to find it and to google their questions). Ob-

viously a *too* unique package name might make the package less discoverable (e.g. it might be an argument for naming your package geojson).

- Find other interesting aspects of naming your package in this blog post by Nick Tierney, and in case you change your mind, find out how to rename your package in this other blog post of Nick's.

1.1.2 Creating metadata for your package

We recommend you to use the `codemetar` package for creating and updating a JSON CodeMeta metadata file for your package via `codemetar::write_codemeta()`. It will automatically include all useful information, including GitHub topics. CodeMeta uses Schema.org terms so as it gains popularity the JSON metadata of your package might be used by third-party services, maybe even search engines.

1.2 Platforms

- Packages should run on all major platforms (Windows, macOS, Linux). Exceptions may be granted packages that interact with system-specific functions, or wrappers for utilities that only operate on limited platforms, but authors should make every effort for cross-platform compatibility, including system-specific compilation, or containerization of external utilities.

1.3 Package API

1.3.1 Function and argument naming

- Functions and arguments naming should be chosen to work together to form a common, logical programming API that is easy to read, and auto-complete.
  - Consider an `object_verb()` naming scheme for functions in your package that take a common data type or interact with a common API. `object` refers to the data/API and `verb` the primary action. This scheme helps avoid namespace conflicts with packages that may have similar verbs, and makes code readable and easy to auto-complete. For instance, in **stringi**, functions starting with `stri_` manipulate strings (`stri_join()`, `stri_sort()`, and in **googlesheets** functions starting with `gs_` are calls to the Google Sheets API (`gs_auth()`, `gs_user()`, `gs_download()`).
- For functions that manipulate an object/data and return an object/data of the same type, make the object/data the first argument of the function so as to enhance compatibility with the pipe operator (%>%).

- We strongly recommend `snake_case` over all other styles unless you are porting over a package that is already in wide use.
- Avoid function name conflicts with base packages or other popular ones (e.g. `ggplot2`, `dplyr`, `magrittr`, `data.table`)
- Argument naming and order should be consistent across functions that use similar inputs.
- Package functions importing data should not import data to the global environment, but instead must return objects. Assignments to the global environment are to be avoided in general.

1.3.2 Console messages

- Use `message()` and `warning()` to communicate with the user in your functions. Please do not use `print()` or `cat()` unless it's for a `print.*()` or `str.*()` methods, as these methods of printing messages are harder for the user to suppress.

1.3.3 Interactive/Graphical Interfaces

If providing graphical user interface (GUI) (such as a Shiny app), to facilitate workflow, include a mechanism to automatically reproduce steps taken in the GUI. This could include auto-generation of code to reproduce the same outcomes, output of intermediate values produced in the interactive tool, or simply clear and well-documented mapping between GUI actions and scripted functions. (See also "Testing" below.)

The `tabulizer` package e.g. has an interactive workflow to extract tables, but can also only extract coordinates so one can re-run things as a script. Besides, two examples of shiny apps that do code generation are https://gdancik.shinyapps.io/shinyGEO/, and https://github.com/wallaceEcoMod/wallace/.

1.4 Code Style

- For more information on how to style your code, name functions, and R scripts inside the `R/` folder, we recommend reading the code chapter in The R Packages book. We recommend the `styler` package for automating part of the code styling. We suggest reading the Tidyverse style guide.
- You can choose to use = over <− as long you are consistent with one choice within your package. We recommend avoiding the use of −> for assignment within a package. If you do use <− throughout your package, and you also

use `R6` in that package, you'll be forced to use = for assignment within your `R6Class` construction - this is not considered an inconsistency because you can't use <- in this case.

1.5 CITATION file

- If your package does not yet have a CITATION file, you can create one with `usethis::use_citation()`, and populate it with values generated by the `citation()` function.
- If you archive each release of your GitHub repo on Zenodo, add the Zenodo top-level DOI to the CITATION file.
- If one day **after** review at rOpenSci you publish a software publication about your package, add it to the CITATION file.
- Less related to your package itself but to what supports it: if your package wraps a particular resource such as data source or, say, statistical algorithm, remind users of how to cite that resource via e.g. `citHeader()`. Maybe even add the reference for the resource.

As an example see nasapower CITATION file that refers to both the manual and a paper. All it lacks is a Zenodo DOI for the manual – although most users would probably end up citing the JOSS paper.

```
citHeader("While nasapower does not redistribute the data in any way,\n",
          "we encourage users to follow the requests of the POWER\n",
          "Project Team:\n",
          "\n",
          "'When POWER data products are used in a publication, we\n",
          "request the following acknowledgment be included:\n",
          "These data were obtained from the NASA Langley Research\n",
          "Center POWER Project funded through the NASA Earth Science\n",
          "Directorate Applied Science Program.'\n",
          "\n",
          "To cite nasapower in publications, please use:")
citEntry(
  entry = "Article",
  author = as.person("Adam H Sparks"),
  title = "nasapower: A NASA POWER Global Meteorology, Surface Solar Energy and Climat
  doi = "10.21105/joss.01035",
  year = 2018,
  month = "oct",
  publisher = "The Open Journal",
  volume = 3,
  number = 30,
```

```
  pages = 1035,
  journal = "The Journal of Open Source Software",
  textVersion = paste("Sparks, Adam (2018). nasapower: A NASA POWER Global Meteorology, Surface S
year <- sub("-.*", "", meta$Date)
note <- sprintf("R package version %s", meta$Version)
bibentry(bibtype = "Manual",
         title = "{nasapower}: NASA-POWER Data from R",
         author = c(person("Adam", "Sparks")),
         year = year,
         note = note,
     url = "https://CRAN.R-project.org/package=nasapower")
 textVersion = paste0("Adam H Sparks, (", year, ").",
  " nasapower: A NASA POWER Global Meteorology, Surface Solar Energy and Climatology Data Client
  note, ".",
  " https://CRAN.R-project.org/package=nasapower")
```

- You could also create and store a `CITATION.cff` thanks to the cffr package. It also provides a GitHub Action workflow to keep the `CITATION.cff` file up-to-date.

1.6 README

- All packages should have a README file, named `README.md`, in the root of the repository. The README should include, from top to bottom:
  - The package name.
  - Badges for continuous integration and test coverage, the badge for rOpenSci peer-review once it has started (see below), a repostatus.org badge, and any other badges (e.g. R-universe). If the README has many more badges, you might want to consider using a table for badges, see this example, that one and that one. Such a table should be more wide than high.
  - Short description of goals of package, with descriptive links to all vignettes (rendered, i.e. readable, cf the documentation website section) unless the package is small and there's only one vignette repeating the README.
  - Installation instructions using e.g. the remotes package, pak package, or R-universe.
  - Any additional setup required (authentication tokens, etc).
  - Brief demonstration usage.
  - If applicable, how the package compares to other similar packages and/or how it relates to other packages.

  - Citation information i.e. Direct users to the preferred citation in the README by adding boilerplate text "here's how to cite my package". See e.g. ecmwfr README.

If you use another repo status badge such as a lifecycle badge, please also add a repostatus.org badge. Example of a repo README with two repo status badges.

- Once you have submitted a package and it has passed editor checks, add a peer-review badge via

```
[![](https://badges.ropensci.org/<issue_id>_status.svg)](https://github.com/ropensci/s
```

where issue_id is the number of the issue in the software-review repository. For instance, the badge for `rtimicropem` review uses the number 126 since it's the review issue number. The badge will first indicated "under review" and then "peer-reviewed" once your package has been onboarded (issue labelled "approved" and closed), and will link to the review issue.

- If your README has many badges consider ordering them in an html table to make it easier for newcomers to gather information at a glance. See examples in `drake` repo and in `qualtRics` repo. Possible sections are
  - Development (CI statuses cf CI chapter, Slack channel for discussion, repostatus)
  - Release/Published (CRAN version and release date badges from METACRAN, CRAN checks API badge, Zenodo badge)
  - Stats/Usage (downloads e.g. download badges from METACRAN) The table should be more wide than it is long in order to mask the rest of the README.
- If your package connects to a data source or online service, or wraps other software, consider that your package README may be the first point of entry for users. It should provide enough information for users to understand the nature of the data, service, or software, and provide links to other relevant data and documentation. For instance, a README should not merely read, "Provides access to GooberDB," but also include, "…, an online repository of Goober sightings in South America. More information about GooberDB, and documentation of database structure and metadata can be found at *link*".
- We recommend not creating `README.md` directly, but from a `README.Rmd` file (an R Markdown file) if you have any demonstration code. The advantage of the `.Rmd` file is you can combine text with code that can be easily updated whenever your package is updated.
- Consider using `usethis::use_readme_rmd()` to get a template for a `README.Rmd` file and to automatically set up a pre-commit hook to ensure that `README.md` is always newer than `README.Rmd`.

- Extensive examples should be kept for a vignette. If you want to make the vignettes more accessible before installing the package, we suggest creating a website for your package.
- Add a code of conduct and contribution guidelines.
- See the `gistr` README for a good example README to follow for a small package, and `bowerbird` README for a good example README for a larger package.

1.7 Documentation

1.7.1 General

- All exported package functions should be fully documented with examples.
- If there is potential overlap or confusion with other packages providing similar functionality or having a similar name, add a note in the README, main vignette and potentially the Description field of DESCRIPTION. Example in rtweet README, rebird README.
- The package should contain top-level documentation for `?foobar`, (or `?`foobar-package`` if there is a naming conflict). Optionally, you can use both `?foobar` and `?`foobar-package`` for the package level manual file, using `@aliases` roxygen tag. `usethis::use_package_doc()` adds the template for the top-level documentation.
- The package should contain at least one **HTML** vignette providing a substantial coverage of package functions, illustrating realistic use cases and how functions are intended to interact. If the package is small, the vignette and the README may have very similar content.
- As is the case for a README, top-level documentation or vignettes may be the first point of entry for users. If your package connects to a data source or online service, or wraps other software, it should provide enough information for users to understand the nature of the data, service, or software, and provide links to other relevant data and documentation. For instance, a vignette intro or documentation should not merely read, "Provides access to GooberDB," but also include, "…, an online repository of Goober sightings in South America. More information about GooberDB, and documentation of database structure and metadata can be found at *link*". Any vignette should outline prerequisite knowledge to be able to understand the vignette upfront.

The general vignette should present a series of examples progressing in complexity from basic to advanced usage.

- Functionality likely to be used by only more advanced users or developers might be better put in a separate vignette (e.g. programming/NSE with dplyr).

- The README, the top-level package docs, vignettes, websites, etc., should all have enough information at the beginning to get a high-level overview of the package and the services/data it connects to, and provide navigation to other relevant pieces of documentation. This is to follow the principle of *multiple points of entry* i.e. to take into account the fact that any piece of documentation may be the first encounter the user has with the package and/or the tool/data it wraps.

- The vignette(s) should include citations to software and papers where appropriate.

- If your package provides access to a data source, we require that DESCRIPTION contains both (1) A brief identification and/or description of the organisation responsible for issuing data; and (2) The URL linking to public-facing page providing, describing, or enabling data access (which may often differ from URL leading directly to data source).

- Only use package startup messages when necessary (function masking for instance). Avoid package startup messages like "This is foobar 2.4-0" or citation guidance because they can be annoying to the user. Rely on documentation for such guidance.

- You can choose to have a README section about use cases of your package (other packages, blog posts, etc.), example.

1.7.2 roxygen2 use

- We request all submissions to use `roxygen2` for documentation. `roxygen2` is an R package that automatically compiles `.Rd` files to your `man` folder in your package from simple tags written above each function.

- More information on using roxygen2 documentation is available in the R packages book.

- If you were writing Rd by hand, the Rd2roxygen package contains functions to convert Rd to roxygen documentation.

- One key advantage of using `roxygen2` is that your `NAMESPACE` will always be automatically generated and up to date.

- All functions should document the type of object returned under the `@return` heading.

- Documentation should support user navigation by including useful cross-links between related functions and documenting related functions together in groups or in common help pages. The `@seealso` and `@family` tags (the latter of which automatically creates "See also" links and can help group functions together on pkgdown sites), are recommended for this purpose. See the "manual" section of The R Packages book and the "function grouping" section of the present chapter for more details.

- You could re-use documentation pieces (e.g. details about authentication, related packages) across the vignettes/README/man pages. A possible approach for that is the use of R Markdown fragments, relying on knitr use of child documents so you can store the re-used parts in a folder in man/, and call them from both the README and the vignette as well as in man pages by using the `@includeRmd` tag available from `roxygen 0.7.0`.

- Add `#' @noRd` to internal functions.

- If you prefer not to clutter up code with extensive documentation, you can place further examples outside of the R script and use the `@example` tag; and you place further documentation in files in a `man-roxygen` folder in the root of your package, and those will be combined into the manual file by the use of `@template <file name>`, for example.

  - Put any documentation for an object in a `.R` file in the `man-roxygen` folder (at the root of your package). For example, this file. Link to that template file from your function (e.g.) with the `@template` keyword (e.g.). The contents of the template will be inserted when documentation is built into the resulting `.Rd` file that users will see when they ask for documentation for the function.
  - Note that if you are using markdown documentation, markdown currently doesn't work in template files, so make sure to use latex formatting.
  - In most cases you can ignore templates and `man-roxygen`, but there are two cases in which leveraging them will greatly help:
    1. When you have a lot of documentation for a function/class/object separating out certain chunks of that documentation can keep your `.R` source file tidy. This is especially useful when you have a lot of code in that `.R` file. On the other hand, it means the docs are not readable "in-source" since they're in another file.
    2. When you have the same documentation parts used across many `.R` functions it's helpful to use a template. This reduces duplicated text, and helps prevent mistakenly updating documentation for one function but not the other. Using a template file for a parameter documentation is an alternative to defining the parameter for one function and using `@inheritParams` for other functions using the same parameter.

- Starting from `roxygen2` version 7.0.0, `R6` classes are officially supported. See the roxygen2 docs for details on how to document `R6` classes.

1.7.3 URLs in documentation

This subsection is particularly relevant to authors wishing to submit their package to CRAN. CRAN will check URLs in your documentation and does not allow redirect status codes such as 301. You can use the urlchecker package to reproduce these checks and, in particular, replace URLs with the URLs they redirect to. Others have used the option to escape some URLs (change `<https://ropensci.org/>` to `https://ropensci.org/`, or `\url{https://ropensci.org/}` to `https://ropensci.org/`.), but if you do so, you will need to implement some sort of URL checking yourself to prevent them from getting broken without your noticing. Furthermore, links would not be clickable from local docs.

1.8 Documentation website

We recommend creating a documentation website for your package using `pkgdown`. Neal Richardson wrote a good tutorial to get started with `pkgdown`, and unsurprisingly `pkgdown` has its own documentation website.

There are a few elements we'd like to underline here.

1.8.1 Automatic deployment of the documentation website

You only need to worry about automatic deployment of your website until approval and transfer of your package repo to the ropensci organization; indeed, after that a pkgdown website will be built for your package after each push to the GitHub repo. You can find the status of these builds at `https://dev.ropensci.org/job/package_name`, e.g. for `magick`; and the website at `https://docs.ropensci.org/package_name`, e.g. for `magick`. The website build will use your pkgdown config file if you have one, except for the styling that will use the `rotemplate` package. The resulting website will have a local search bar. Please report bugs, questions and feature requests about the central builds at https://github.com/ropensci/docs/ and about the template at https://github.com/ropensci/rotemplate/.

If your package vignettes need credentials (API keys, tokens, etc.) to knit, you might want to precompute them since credentials cannot be used on the docs server.

Before submission and before transfer, you could use the approach documented by `pkgdown` or the `tic` package for automatic deployment of the package's

website. This would save you the hassle of running (and remembering to run) `pkgdown::build_site()` yourself every time the site needs to be updated. First refer to our chapter on continuous integration if you're not familiar with continuous integration. In any case, do not forget to update all occurrences of the website URL after transfer to the ropensci organization.

1.8.2 Grouping functions in the reference

When your package has many functions, use grouping in the reference, which you can do more or less automatically.

If you use `roxygen2` above version 6.1.1, you should use the `@family` tag in your functions documentation to indicate grouping. This will give you links between functions in the local documentation of the installed package ("See also" section) *and* allow you to use the `pkgdown` `has_concept` function in the config file of your website. Non-rOpenSci example courtesy of `optiRum`: family tag, `pkgdown` config file and resulting reference section. To customize the text of the cross-reference title created by roxygen2 (`Other {family}:`), refer to roxygen2 docs regarding how to provide a `rd_family_title` list in `man/roxygen/meta.R`.

Less automatically, see the example of `drake` website and associated config file.

1.8.3 Branding of authors

You can make the names of (some) authors clickable by adding their URL, and you can even replace their names with a logo (think rOpenSci… or your organisation/company!). See `pkgdown` documentation.

1.8.4 Tweaking the navbar

You can make your website content easier to browse by tweaking the navbar, refer to `pkgdown` documentation. In particular, note that if you name the main vignette of your package "pkg-name.Rmd", it'll be accessible from the navbar as a `Get started` link instead of via `Articles > Vignette Title`.

1.8.5 Mathjax

Once your package is transferred and it gets a website using our `pkgdown` template, if you want to use Mathjax you'll need to specify it in the `pkgdown` config file like so:

```
template:
  params:
    mathjax: true
```

1.8.6 Package logo

To use your package logo in the pkgdown homepage, refer to `usethis::use_logo()`. If your package doesn't have any logo, the rOpenSci docs builder will use rOpenSci logo instead.

1.9 Authorship

The `DESCRIPTION` file of a package should list package authors and contributors to a package, using the `Authors@R` syntax to indicate their roles (author/creator/contributor etc.) if there is more than one author, and using the comment field to indicate the ORCID ID of each author, if they have one (cf this post). See this section of "Writing R Extensions" for details. If you feel that your reviewers have made a substantial contribution to the development of your package, you may list them in the `Authors@R` field with a Reviewer contributor type (`"rev"`), like so:

```
person("Bea", "Hernández", role = "rev",
comment = "Bea reviewed the package (v. X.X.XX) for rOpenSci, see <https://github.
```

Only include reviewers after asking for their consent. Read more in this blog post "Thanking Your Reviewers: Gratitude through Semantic Metadata". Please do not list editors as contributors. Your participation in and contribution to rOpenSci is thanks enough!

1.9.1 Authorship of included code

Many packages include code from other software. Whether entire files or single functions are included from other packages, rOpenSci packages should follow the CRAN *Repository Policy*:

> The ownership of copyright and intellectual property rights of all components of the package must be clear and unambiguous (including from the authors specification in the DESCRIPTION file). Where code is copied (or derived) from the work of others (including from R itself), care must be taken that any copyright/license statements are preserved and authorship is not misrepresented.
>
> Preferably, an 'Authors@R' field would be used with 'ctb' roles for the authors of such code. Alternatively, the 'Author' field should list these authors as contributors.
>
> Where copyrights are held by an entity other than the package authors, this should preferably be indicated via 'cph' roles in the

'Authors@R' field, or using a 'Copyright' field (if necessary referring to an inst/COPYRIGHTS file).

Trademarks must be respected.

1.10 Licence

The package needs to have a CRAN or OSI accepted license. For more explanations around licensing, refer to the R packages book.

1.11 Testing

- All packages should pass `R CMD check`/`devtools::check()` on all major platforms.
- All packages should have a test suite that covers major functionality of the package. The tests should also cover the behavior of the package in case of errors.
- It is good practice to write unit tests for all functions, and all package code in general, ensuring key functionality is covered. Test coverage below 75% will likely require additional tests or explanation before being sent for review.
- We recommend using testthat for writing tests. Strive to write tests as you write each new function. This serves the obvious need to have proper testing for the package, but allows you to think about various ways in which a function can fail, and to *defensively* code against those. More information.
- Tests should be easy to understand. We suggest reading the blog post *"Why Good Developers Write Bad Unit Tests"* by Michael Lynch.
- Packages with Shiny apps should use a unit-testing framework such as `shinytest` to test that interactive interfaces behave as expected.
- For testing your functions creating plots, we suggest using vdiffr, an extension of the testthat package; or testthat snapshot tests.
- If your package interacts with web resources (web APIs and other sources of data on the web) you might find the HTTP testing in R book by Scott Chamberlain and Maëlle Salmon relevant. Packages helping with HTTP testing (corresponding HTTP clients):
  - httptest2 (httr2);
  - httptest (httr);
  - vcr (httr, crul);

  - webfakes (httr, httr2, crul, curl).

- testthat has a function `skip_on_cran()` that you can use to not run tests on CRAN. We recommend using this on all functions that are API calls since they are quite likely to fail on CRAN. These tests should still run on continuous integration. Note that from testthat 3.1.2 `skip_if_offline()` automatically calls `skip_on_cran()`. More info on CRAN preparedness for API wrappers.
- If your package interacts with a database you might find dittodb useful.
- Once you've set up continuous interaction (CI), use your package's code coverage report (cf this section of our book) to identify untested lines, and to add further tests.
- Even if you use continuous integration, we recommend that you run tests locally prior to submitting your package, as some tests are often skipped (you may need to set `Sys.setenv(NOT_CRAN="true")` in order to ensure all tests are run). In addition, we recommend that prior to submitting your package, you use MangoTheCat's **goodpractice** package to check your package for likely sources of errors, and run `spelling::spell_check_package()` to find spelling errors in documentation.

1.12 Examples

- Include extensive examples in the documentation. In addition to demonstrating how to use the package, these can act as an easy way to test package functionality before there are proper tests. However, keep in mind we require tests in contributed packages.
- You can run examples with `devtools::run_examples()`. Note that when you run R CMD CHECK or equivalent (e.g., `devtools::check()`) your examples that are not wrapped in `\dontrun{}` or `\donttest{}` are run. Refer to the summary table in roxygen2 docs.
- To safe-guard examples (e.g. requiring authentication) to be run on CRAN you need to use `\dontrun{}`. However, for a first submission CRAN won't let you have all examples escaped so. In this case you might add some small toy examples, or wrap example code in `try()`. Also refer to the `@exampleIf` tag present, at the time of writing, in roxygen2 development version.
- In addition to running examples locally on your own computer, we strongly advise that you run examples on one of the continuous integration systems. Again, examples that are not wrapped in `\dontrun{}` or `\donttest{}` will be run, but for those that are you can configure your continuous integration builds to run them via R CMD check arguments `--run-dontrun` and/or `--run-donttest`.

1.13 Package dependencies

- Use `Imports` instead of `Depends` for packages providing functions from other packages. Make sure to list packages used for testing (`testthat`), and documentation (`knitr`, `roxygen2`) in your `Suggests` section of package dependencies (if you use `usethis` for adding testing infrastructure via `usethis::use_testthat()` or a vignette via usethis::use_vignette(), the necessary packages will be added to `DESCRIPTION`). If you use any package in the examples or tests of your package, make sure to list it in `Suggests`, if not already listed in `Imports`.

- If your (not Bioconductor) package depends on Bioconductor packages, make sure the installation instructions in the README and vignette are clear enough even for an user who is not familiar with the Bioconductor release cycle.

  - Should the user use `BiocManager` (recommended)? Document this.
  - Is the automatic installation of Bioconductor packages by `install.packages()` enough? In that case, mention that the user needs to run `setRepositories()` if they haven't set the necessary Bioconductor repositories yet.
  - If your package depends on Bioconductor after a certain version, mention it in DESCRIPTION and in the installation instructions.

- Specifying minimum dependencies (e.g. `glue (>= 1.3.0)` instead of just `glue`) should be a conscious choice. If you know for a fact that your package will break below a certain dependency version, specify it explicitly. But if you don't, then no need to specify a minimum dependency. In that case when a user reports a bug which is explicitly related to an older version of a dependency then address it then. An example of bad practice would be for a developer to consider the versions of their current state of dependencies to be the minimal version. That would needlessly force everyone to upgrade (causing issues with other packages) when there is no good reason behind that version choice.

- For most cases where you must expose functions from dependencies to the user, you should import and re-export those individual functions rather than listing them in the `Depends` fields. For instance, if functions in your package produce `raster` objects, you might re-export only printing and plotting functions from the **raster** package.

- If your package uses a *system* dependency, you should

  - Indicate it in DESCRIPTION;
  - Check that it is listed by `sysreqsdb` to allow automatic tools to install it, and submit a contribution if not;

  - Check for it in a `configure` script (example) and give a helpful error message if it cannot be found (example). `configure` scripts can be challenging as they often require hacky solutions to make diverse system dependencies work across systems. Use examples (more here) as a starting point but note that it is common to encounter bugs and edge cases and often violate CRAN policies. Do not hesitate to ask for help on our forum.

- Consider the trade-offs involved in relying on a package as a dependency. On one hand, using dependencies reduces coding effort, and can build on useful functionality developed by others, especially if the dependency performs complex tasks, is high-performance, and/or is well vetted and tested. On the other hand, having many dependencies places a burden on the maintainer to keep up with changes in those packages, at risk to your package's long-term sustainability. It also increases installation time and size, primarily a consideration on your and others' development cycle, and in automated build systems. "Heavy" packages - those with many dependencies themselves, and those with large amounts of compiled code - increase this cost. Here are some approaches to reducing dependencies:

  - Small, simple functions from a dependency package may be better copied into your own package if the dependency if you are using only a few functions in an otherwise large or heavy dependency. (See *Authorship* section above for how to acknowledge original authors of copied code.) On the other hand, complex functions with many edge cases (e.g. parsers) require considerable testing and vetting.

    * An common example of this is in returning tidyverse-style "tibbles" from package functions that provide data. One can avoid the modestly heavy **tibble** package dependency by returning a tibble created by modifying a data frame like so:

      ```
      class(df) <- c("tbl_df", "tbl", "data.frame")
      ```

      (Note that this approach is not universally endorsed.)

  - Ensure that you are using the package where the function is defined, rather than one where it is re-exported. For instance many functions in **devtools** can be found in smaller specialty packages such as **sessioninfo**. The %>% function should be imported from **magrittr**, where it is defined, rather than the heavier **dplyr**, which re-exports it.

  - Some dependencies are preferred because they provide easier to interpret function names and syntax than base R solutions. If this is the primary reason for using a function in a heavy dependency, consider wrapping the base R approach in a nicely-named internal function in your package. See e.g. the rlang R script providing functions with a syntax similar to purrr functions.

  - If dependencies have overlapping functionality, see if you can rely on only one.

  - More dependency-management tips can be found in this post by Scott Chamberlain.

1.14 Recommended scaffolding

- For HTTP requests we recommend using curl, crul, httr or httr2 over RCurl. If you like low level clients for HTTP, curl is best, whereas crul or httr are better for higher level access. crul is maintained by rOpenSci. We recommend the rOpenSci maintained packages webmockr for mocking HTTP requests, and vcr for caching HTTP requests in package tests.
- For parsing JSON, use jsonlite instead of rjson or RJSONIO.
- For parsing, creating, and manipulating XML, we strongly recommend xml2 for most cases. You can refer to Daniel Nüst's notes about migration from XML to xml2.
- For spatial data, the sp package should be considered deprecated in favor of sf, and the packages rgdal, maptools, and rgeos will be retired by the end of 2023. We recommend use of the spatial suites developed by the r-spatial and rspatial communities. See this GitHub issue for relevant discussions.

1.15 Version Control

- Your package source files have to be under version control, more specifically tracked with Git. You might find the gert package relevant, as well as some of usethis Git/GitHub related functionality; you can however use git as you want.
- Make sure to list "scrap" such as `.DS_Store` files in .gitignore. You might find the gitignore package relevant.
- A later section of this book contains some git workflow tips.

1.16 Miscellaneous CRAN gotchas

This is a collection of CRAN gotchas that are worth avoiding at the outset.

- Make sure your package title is in Title Case.
- Do not put a period on the end of your title.
- Do not put 'in R' or 'with R' in your title as this is obvious from packages hosted on CRAN. If you would like this information to be displayed on your website nonetheless, check the `pkgdown` documentation to learn how to override this.

- Avoid starting the description with the package name or "This package …".
- Make sure you include links to websites if you wrap a web API, scrape data from a site, etc. in the `Description` field of your `DESCRIPTION` file. URLs should be enclosed in angle brackets, e.g. `<https://www.r-project.org>`.
- In both the `Title` and `Description` fields, the names of packages or other external software must be quoted using single quotes (e.g., *'Rcpp' Integration for the 'Armadillo' Templated Linear Algebra Library*).
- Avoid long running tests and examples. Consider `testthat::skip_on_cran` in tests to skip things that take a long time but still test them locally and on continuous integration.
- Include top-level files such as `paper.md`, continuous integration configuration files, in your `.Rbuildignore` file.

1.16.1 CRAN checks

Once your package is on CRAN, it will be regularly checked on different platforms. Failures of such checks, when not false positives, can lead to the CRAN team's reaching out. You can monitor the state of the CRAN checks via

- the `foghorn` package.
- the CRAN checks API maintained by rOpenSci, that provides badges.

1.17 Bioconductor gotchas

If you intend your package to be submitted to, or if your package is on, Bioconductor, refer to Bioconductor packaging guidelines.

1.18 Further guidance

- If you are submitting a package to rOpenSci via the software-review repo, you can direct further questions to the rOpenSci team in the issue tracker, or in our discussion forum.
- Before submitting a package use the **goodpractice** package (`goodpractice::gp()`) as a guide to improve your package, since most exceptions to it will need to be justified. E.g. the use of `foo` might be generally bad and therefore flagged by `goodpractice` but you had a good reason to use it in your package.
- Read, incorporate, and act on advice from the *Collaboration Guide* chapter.

1.18.1 Learning about package development

1.18.1.1 Books

- Hadley Wickham and Jenny Bryan's *R packages* book is an excellent, readable resource on package development which is available for free online (and print – link to former version by Hadley Wickham as the new version is not published yet as of June 2022).
- Writing R Extensions is the canonical, usually most up-to-date, reference for creating R packages.
- *Mastering Software Development in R* by Roger D. Peng, Sean Kross, and Brooke Anderson.
- *Advanced R* by Hadley Wickham

1.18.1.2 Tutorials

- Hilary Parker's famous blog post *Writing an R package from scratch* or its updated version by Tomas Westlake that shows how to do the same more efficiently using `usethis`.
- this workflow description by Emil Hvitfeldt.
- This pictorial by Matthew J Denny.

1.18.1.3 Blogs

- R-hub blog.
- Some posts of the rOpenSci blog e.g. "How to precompute package vignettes or pkgdown articles".
- Package Development Corner section of rOpenSci newsletter.
- Some posts of the tidyverse blog e.g. "Upgrading to testthat edition 3".

1.18.1.4 MOOCs

There is a Coursera specialization corresponding to the book by Roger Peng, Sean Kross and Brooke Anderson, with a course specifically about R packages.

Chapter 2

Continuous Integration Best Practices

This chapter summarizes our guidelines about continuous integration after explaining what continuous integration is.

Along with last chapter, it forms our guidelines for Software Peer Review.

2.1 Why use continuous integration (CI)?

All rOpenSci packages must use one form of continuous integration. This ensures that all commits, pull requests and new branches are run through `R CMD check`. rOpenSci packages' continuous integration must also be linked to a code coverage service, indicating how many lines are covered by unit tests.

Both test status and code coverage should be reported via badges in your package README.

R packages should have CI for all operating systems (Linux, Mac OSX, Windows) when they contain:

- Compiled code
- Java dependencies
- Dependencies on other languages
- Packages with system calls
- Text munging such as getting people's names (in order to find encoding issues).

- Anything with file system / path calls

In case of any doubt regarding the applicability of these criteria to your package, it's better to add CI for all operating systems. Most CI services standards setups for R packages allow this with not much hassle.

2.2 Which continuous integration service(s)?

There are a number of continuous integration services, including standalone services (CircleCI, AppVeyor), and others integrated into code hosting or related services (GitHub Actions, GitLab, AWS Code Pipeline). Different services support different operating system configurations.

GitHub Actions is a convenient option for many R developers who already use GitHub as it is integrated into the platform and supports all needed operating Systems. There are actions supported for the R ecosystem, as well and first-class support in the {usethis} package. All packages submitted to rOpenSci for peer review are checked by our own `pkgcheck` system, described further in the Guide for Authors. These checks are also provided as a GitHub Action in the `ropensci-review-tools/pkgcheck-action` repository. Packages authors are encouraged to use that action to confirm prior to submission that a package passes all of our checks. See our blog post for more information.

usethis supports CI setup for other systems, though these functions are soft-deprecated. rOpenSci also supports the circle package, which aids in setting up CircleCI pipelines, and the tic package for building more complicated CI pipelines.

2.2.0.1 Testing using different versions of R

We require that rOpenSci packages are tested against the latest, previous and development versions of R to ensure both backwards and forwards compatibility with base R.

Details of how to run tests/checks using different versions of R locally can be found in the R-hub vignette on running Local Linux checks with Docker.

You can fine tune the deployment of tests with each versions by using a testing matrix.

If you develop a package depending on or intended for Bioconductor, you might find biocthis relevant.

2.2.0.2 Minimizing build times on CI

You can use these tips to minimize build time on CI:

- Cache installation of packages. The default r-lib/actions do this.

2.2.1 Travis CI (Linux and Mac OSX)

We recommend moving away from Travis.

2.2.2 AppVeyor CI (Windows)

For continuous integration on Windows, see R + AppVeyor. Set it up using `usethis::use_appveyor()`.

Here are tips to minimize AppVeyor build time:

- Cache installation of packages. Example in a config file. It'll already be in the config file if you set AppVeyor CI up using `usethis::use_appveyor()`.
- Enable rolling builds.

We no longer transfer AppVeyor projects to ropensci AppVeyor account so after transfer of your repo to rOpenSci's "ropensci" GitHub organization the badge will be `[![AppVeyor Build Status](https://ci.appveyor.com/api/projects/status/github/ropensci/pkgname?bra`

2.2.3 Circle CI (Linux and Mac OSX)

Circle CI is used, for example, by rOpenSci package `bomrang` as continuous integration service.

2.3 Test coverage

Continuous integration should also include reporting of test coverage via a testing service such as Codecov or Coveralls. See the README for the **covr** package for instructions, as well as `usethis::use_coverage()`.

If you run coverage on several CI services the results will be merged.

2.4 Even more CI: OpenCPU

After transfer to rOpenSci's "ropensci" GitHub organization, each push to the repo will be built on OpenCPU and the person committing will receive a notification email. This is an additional CI service for package authors that allows for R functions in

packages to be called remotely via https://ropensci.ocpu.io/ using the opencpu API. For more details about this service, consult the OpenCPU help page that also indicates where to ask questions.

2.5 Even more CI: rOpenSci docs

After transfer to rOpenSci's "ropensci" GitHub organization, a pkgdown website will be built for your package after each push to the GitHub repo. You can find the status of these builds at `https://dev.ropensci.org/job/package_name`, e.g. for `magick`; and the website at `https://docs.ropensci.org/package_name`, e.g. for `magick`. The website build will use your pkgdown config file if you have one, except for the styling that will use the `rotemplate` package. Please report bugs, questions and feature requests about the central builds at https://github.com/ropensci/docs/ and about the template at https://github.com/ropensci-org/rotemplate/.

Chapter 3

Package Development Security Best Practices

This work-in-progress chapter includes guidance about managing secrets in packages and links for further reading.

3.1 Miscellaneous

We recommend the article Ten quick tips for staying safe online by Danielle Smalls and Greg Wilson.

3.2 GitHub access security

- We recommend you secure your GitHub account with two-factor (authentication) 2FA. It is *compulsory* for all ropensci GitHub organization members and outside collaborators so make sure to enable it before your package is approved.
- We also recommend you regularly check who has access to your package repository, and that you prune any unused access (such as from former collaborators).

3.3 https

- If the web service your package wraps has either https or http, opt for https.

3.4 Secrets in packages

This section contains guidance for when you develop a package interacting with a web resource requiring credentials (API keys, tokens, etc.). Also refer to the `httr` vignette about sharing secrets.

3.4.1 Secrets in packages and user protection

Say your package needs an API key for making requests on behalf of users of your package.

- In your package documentation, guide the user so the API key doesn't end up in the .Rhistory/script of users of your package.
  - Encourage the use of environment variables to store the API key (or even remove the possibility to pass it as an argument to the functions?). You could link to this intro to startup files and `usethis::edit_r_environ()`.
  - Or your package could depend on, or encourage the use of, `keyring` to help user store variables in the specific OS' credential stores (more secure than .Renviron): i.e. you'd create a function for setting the key, and have another one for retrieving the key; or the user would write `Sys.setenv(SUPERSECRETKEY = keyring::key_get("myservice"))` at the beginning of their script.
  - Do not print the API key even in verbose mode in any message, warning, error.
- In the GitHub issue template, it should be stated not to share any credentials. If an user of your package accidentally shares credentials in an issue, make sure they're aware of that so they can revoke the key (i.e. ask them explicitly in an answer whether they realized they shared their key).

3.4.2 Secrets in packages and development

You'll need to protect your secrets as you protect secrets of users, but there's more to take into account and keep in mind.

3.4.2.1 Secrets and recorded requests in tests

If you use `vcr` or `httptest` in tests for caching API responses, you need to make sure the recorded requests / fixtures do not contain secrets. Refer to `vcr` security guidance and `httptest` guidance "Redacting and Modifying Recorded Requests",

and inspect your recorded requests / fixtures before committing them the first time to be sure you got the setup right.

`vcr` being an rOpenSci package, you can post any question you might have to rOpenSci forum.

3.4.2.2 Share secrets with CI services

Now, you might need to share secrets with continuous integration services.

You could store API keys as environment variables / secrets, referring to the docs of the CI service.

For more details and workflow advice, refer to the gargle article "Managing tokens securely" and the security chapter of the HTTP testing in R book.

Document the steps you made in CONTRIBUTING.md so you, or say a new maintainer, can remember how to do that next time.

3.4.2.3 Secrets and collaborations

What about pull requests from external contributors? Tests using your secrets will fail unless you use some sort of mocked/cached response, so you might want to skip them depending on the context. For instance, in your CI account you could create an environment variable called `THIS_IS_ME` and then skip tests based on the presence of this variable. This obviously means the PR checks by the CI are not exhaustive, so you'll need to check out the PR locally to run all tests.

Document the behavior of your package for external PRs in CONTRIBUTING.md for the sake of people making PRs and of people reviewing them (you in a few weeks, and other authors of the package).

3.4.3 Secrets and CRAN

On CRAN, skip any tests (`skip_on_cran()`) and examples (`dontrun`) requiring credentials.

Precompute vignettes requiring credentials.

3.5 Further reading

Useful security resources:

- rOpenSci community call “Security for R” (recording, slides, etc. see in particular the list of resources);
- the security-related projects of unconf18;
- the `notary` package;
- `gargle` article “Managing tokens securely”

Part II

Software Peer Review of Packages

Chapter 4

Software Peer Review, Why? What?

This chapter contains a general intro to our software peer review system for packages, reasons to submit a package, reasons to volunteer as a reviewer, why our reviews are open, and acknowledgements of actors of the system.

Our system has recently been expanded to statistical software peer-review.

If you use our standards/checklists etc. when reviewing software elsewhere, do tell the recipients (e.g. journal editors, students, internal code review) that they came from rOpenSci, and tell us in our public forum, or privately by email.

4.1 What is rOpenSci Software Peer Review?

rOpenSci's suite of packages is partly contributed by staff members and partly contributed by community members, which means the suite stems from a great diversity of skills and experience of developers. How to ensure quality for the whole set? That's where software peer review comes into play: packages contributed by the community undergo a transparent, constructive, non adversarial and open review process. For that process relying mostly on volunteer work, associate editors manage the incoming flow and ensure progress of submissions; authors create, submit and improve their package; reviewers, two per submission, examine the software code and user experience. This blog post written by rOpenSci editors is a good introduction to rOpenSci software peer review Other blog posts about review itself and reviewed packages can be find via the "software-peer-review" tag on rOpenSci blog.

You can recognize rOpenSci packages that have been peer-reviewed via a green "peer-reviewed" badge in their README, linking to their reviews (cf this example);

and via a blue comment icon near their description on rOpenSci packages page, also linking to the reviews.

Technically, we make the most of GitHub infrastructure: each package review process is an issue in the ropensci/software-review GitHub repository. For instance, click here to read the review thread of the `ropenaq` package: the process is an ongoing conversation until acceptance of the package, with two external reviews as important milestones. Furthermore, we use GitHub features such as the use of issue templates (as submission templates), and labelling which we use to track progress of submissions (from editor checks to approval).

4.2 Why submit your package to rOpenSci?

- First, and foremost, we hope you submit your package for review **because you value the feedback**. We aim to provide useful feedback to package authors and for our review process to be open, non-adversarial, and focused on improving software quality.
- Once aboard, your package will continue to receive **support from rOpenSci members**. You'll retain ownership and control of your package, but we can help with ongoing maintenance issues such as those associated with updates to R and dependencies and CRAN policies.
- rOpenSci will **promote your package** through our webpage, blog, and social media. Packages in our suite also get a documentation website that is automatically built and deployed after each push.
- rOpenSci **packages can be cross-listed** with other repositories such as CRAN and BioConductor.
- rOpenSci packages that are in scope for the Journal of Open-Source Software and add the necessary accompanying short paper, would, at the discretion of JOSS editors, benefit from a fast-tracked review process.
- If you write one, rOpenSci will **promote gitbooks related to your package**: the source of such books can be transferred to the `ropensci-books` GitHub organisation for books to be listed at books.ropensci.org.

4.3 Why review packages for rOpenSci?

- As in any peer-review process, we hope you choose to review **to give back to the rOpenSci and scientific communities.** Our mission to expand access to scientific data and promote a culture of reproducible research is only possible through the volunteer efforts of community members like you.
- Review is a two-way conversation. By reviewing packages, you'll have the chance to **continue to learn development practices from authors and other reviewers**.

- The open nature of our review process allows you to **network and meet colleagues and collaborators** through the review process. Our community is friendly and filled with supportive members expert in R development and many other areas of science and scientific computing.
- To volunteer to be one of our reviewers, fill out this short form providing your contact information and areas of expertise. We are always looking for more reviewers with both general package-writing experience and domain expertise in the fields where packages are used.

4.4 Why are reviews open?

Our reviewing threads are public. Authors, reviewers, and editors all know each other's identities. The broader community can view or even participate in the conversation as it happens. This provides an incentive to be thorough and provide non-adversarial, constructive reviews. Both authors and reviewers report that they enjoy and learn more from this open and direct exchange. It also has the benefit of building a community. Participants have the opportunity to meaningfully network with new peers, and new collaborations have emerged via ideas spawned during the review process.

We are aware that open systems can have drawbacks. For instance, in traditional academic review, double-blind peer review can increase representation of female authors, suggesting bias in non-blind reviews. It is also possible reviewers are less critical in open review. However, we posit that the openness of the review conversation provides a check on review quality and bias; it's harder to inject unsupported or subjective comments in public and without the cover of anonymity. Ultimately, we believe that having direct and public communication between authors and reviewers improves quality and fairness of reviews.

Furthermore, authors and reviewers have the ability to contact privately the editors if they have any doubt or question.

4.5 How will users know a package has been reviewed?

- Your package README will feature a peer-review badge linking to the software review thread.
- Your package will get a `docs.ropensci.org` docs website that you can link from DESCRIPTION.
- Your package repo will be transferred to the rOpenSci organization.
- If reviewers agree to be listed in DESCRIPTION, their metadata will mention the review.

4.6 Editors and reviewers

4.6.1 Associate editors

rOpenSci's Software Peer Review process is run by:

- Noam Ross, EcoHealth Alliance
- Karthik Ram, rOpenSci
- Maëlle Salmon, rOpenSci
- Mark Padgham, rOpenSci
- Anna Krystalli, University of Sheffield RSE
- Melina Vidoni, RMIT University (School of Science)
- Mauro Lepore, 2 Degrees Investing Initiative
- Laura DeCicco, USGS
- Julia Gustavsen, Agroscope
- Emily Riederer, Capital One
- Adam Sparks, Department of Primary Industries and Regional Development
- Jeff Hollister, US Environmental Protection Agency

4.6.2 Reviewers

We are grateful to the following individuals who have offered up their time and expertise to review packages submitted to rOpenSci.

Sam Albers · Toph Allen · Kaique dos S. Alves · Brooke Anderson · Alison Appling · Denisse Fierro Arcos · Zebulun Arendsee · Taylor Arnold · Al-Ahmadgaid B. Asaad · Dean Attali · Mara Averick · Suzan Baert · James Balamuta · Vikram Baliga · David Bapst · Joëlle Barido-Sottani · Allison Barner · Cale Basaraba · John Baumgartner · Marcus Beck · Gabriel Becker · Jason Becker · Dom Bennett · Ken Benoit · Aaron Berdanier · Fred Boehm · Carl Boettiger · Will Bolton · Ben Bond-Lamberty · Anne-Sophie Bonnet-Lebrun · Alison Boyer · Abby Bratt · François Briatte · Eric Brown · Julien Brun · Jenny Bryan · Lukas Burk · Lorenzo Busetto · Mario Gavidia Calderón · Brad Cannell · Kevin Cazelles · Scott Chamberlain · Cathy Chamberlin · Jennifer Chang · Pierre Chausse · Jorge Cimentada · Nicholas Clark · Chase Clark · Jon Clayden · Will Cornwell · Nic Crane · Enrico Crema · Ildiko Czeller · Tad Dallas · Kauê de Sousa · Christophe Dervieux · Amanda Dobbyn · Jasmine Dumas · Remko Duursma · Mark Edmondson · Paul Egeler · Evan Eskew · Salvador Fernandez · Alexander Fischer · Kim Fitter · Robert M Flight · Sydney Foks · Stephen Formel · Zachary Stephen Longiaru Foster · Auriel Fournier · Carl Ganz · Duncan Garmonsway · Jan Laurens Geffert · Sharla Gelfand · Monica Gerber · Duncan Gillespie · David Gohel · A. Cagri gokcek · Guadalupe Gonzalez · Rohit Goswami · Laura Graham · Charles Gray · Matthias Grenié · Corinna Gries · Hugo Gruson · W Kyle Hamilton · Ivan Hanigan · Jeffrey Hanson · Rayna Harris · Ted Hart · Nujcharee Haswell · Verena Haunschmid · Stephanie Hazlitt · Andrew Heiss · Max Held · Anna Hepworth · Bea Hernandez · Jim Hester · Peter Hickey · Roel Hogervorst · Kelly Hondula · Allison Horst · Sean Hughes · James

Hunter · Brandon Hurr · Ger Inberg · Christopher Jackson · Najko Jahn · Tamora D James · Mike Johnson · Will Jones · Max Joseph · Megha Joshi · Krunoslav Juraic · Soumya Kalra · Zhian N. Kamvar · Michael Kane · Andee Kaplan · Tinula Kariyawasam · Hazel Kavılı · Jonathan Keane · Christopher T. Kenny · Os Keyes · Eunseop Kim · Aaron A. King · Michael Koontz · Bianca Kramer · Will Landau · Sam Lapp · Erin LeDell · Thomas Leeper · Sam Levin · Lisa Levinson · Stephanie Locke · Marion Louveaux · Robin Lovelace · Julia Stewart Lowndes · Tim Lucas · Muralidhar, M.A. · Andrew MacDonald · Jesse Maegan · Tristan Mahr · Paula Andrea Martinez · Joao Martins · Ben Marwick · Claire Mason · Miles McBain · Lucy D'Agostino McGowan · Amelia McNamara · Elaine McVey · Bryce Mecum · François Michonneau · Mario Miguel · Helen Miller · Beatriz Milz · Jessica Minnier · Priscilla Minotti · Nichole Monhait · Kelsey Montgomery · Paula Moraga · Ross Mounce · Athanasia Monika Mowinckel · Lincoln Mullen · Matt Mulvahill · Maria Victoria Munafó · David Neuzerling · Dillon Niederhut · Rory Nolan · Kari Norman · Jakub Nowosad · Matt Nunes · Daniel Nüst · Joseph O'Brien · Paul Oldham · Samantha Oliver · Dan Olner · Jeroen Ooms · Luis Osorio · Philipp Ottolinger · Mark Padgham · Marina Papadopoulou · Edzer Pebesma · Thomas Lin Pedersen · Marcelo S. Perlin · Rafael Pilliard-Hellwig · Rodrigo Neto Pires · Lindsay Platt · Nicholas Potter · Etienne Racine · Manuel Ramon · Nistara Randhawa · David Ranzolin · Quentin Read · Neal Richardson · tyler rinker · Emily Robinson · David Robinson · Alec Robitaille · Francisco Rodriguez-Sanchez · Julia Romanowska · Xavier Rotllan-Puig · Bob Rudis · Edgar Ruiz · Kent Russel · Michael Sachs · Sheila Saia · Alicia Schep · Klaus Schliep · Clemens Schmid · Patrick Schratz · Collin Schwantes · Marco Sciaini · Heidi Seibold · Julia Silge · Margaret Siple · Peter Slaughter · Mike Smith · Tuija Sonkkila · Øystein Sørensen · Jemma Stachelek · Christine Stawitz · Irene Steves · Kelly Street · Matt Strimas-Mackey · Alex Stringer · Michael Sumner · Chung-Kai Sun · Sarah Supp · Emi Tanaka · Jason Taylor · Filipe Teixeira · Andy Teucher · Jennifer Thompson · Joe Thorley · Tiffany Timbers · Tan Tran · Tim Trice · Utku Turk · Kyle Ueyama · Ted Underwood · Adithi R. Upadhya · Kevin Ushey · Josef Uyeda · Frans van Dunné · Mauricio Vargas · Remi Vergnon · Jake Wagner · Ben Ward · Elin Waring · Rachel Warnock · Leah Wasser · Lukas Weber · Marc Weber · Karissa Whiting · Stefan Widgren · Anna Willoughby · Saras Windecker · Luke Winslow · David Winter · Sebastian Wójcik · Witold Wolski · Kara Woo · Marvin N. Wright · Bruna Wundervald · Lauren Yamane · Emily Zabor · Taras Zakharko · Hao Zhu · Chava Zibman · Naupaka Zimmerman · Jake Zwart · NA · Bri · Flury · NA · Pachá · Rich · Claudia · Jasmine · Lluís · NA · gaurav

We are also grateful to the following individuals who have served as guest editors.

Ana Laura Diedrichs · Francisco Rodriguez-Sanchez · Hao Zhu

Chapter 5

Software Peer Review policies

This chapter contains the policies of rOpenSci Software Peer Review.

In particular, you'll read our policies regarding software peer review itself: the review submission process including our conflict of interest policies, and the aims and scope of the Software Peer Review system. This chapter also features our policies regarding package ownership and maintenance.

Last but not least, you'll find the code of conduct of rOpenSci Software Peer Review.

5.1 Review process

- For a package to be considered for the rOpenSci suite, package authors must initiate a request on the ropensci/software-review repository.
- Packages are reviewed for quality, fit, documentation, clarity and the review process is quite similar to a manuscript review (see our packaging guide and reviewing guide for more details). Unlike a manuscript review, this process will be an ongoing conversation.
- Once all major issues and questions, and those addressable with reasonable effort, are resolved, the editor assigned to a package will make a decision (accept, hold, or reject). Rejections are usually done early (before the review process begins, see the aims and scope section), but in rare cases a package may also be not onboarded after review & revision. It is ultimately editor's decision on whether or not to reject the package based on how the reviews are addressed.
- Communication between authors, reviewers and editors will first and foremost take place on GitHub, although you can choose to contact the editor by email or Slack for some issues. When submitting a package, please make sure your GitHub notification settings make it unlikely you will miss a comment.

- The author can choose to have their submission put on hold (editor applies the holding label). The holding status will be revisited every 3 months, and after one year the issue will be closed.
- If the author hasn't requested a holding label, but is simply not responding, we should close the issue within one month after the last contact intent. This intent will include a comment tagging the author, but also an email using the email address listed in the DESCRIPTION of the package which is one of the rare cases where the editor will try to contact the author by email.
- If a submission is closed and the author wishes to re-submit, they'll have to start a new submission. If the package is still in scope, the author will have to respond to the initial reviews before the editor starts looking for new reviewers.

5.1.1 Publishing in other Venues

- We strongly suggest submitting your package for review *before* publishing on CRAN or submitting a software paper describing the package to a journal. Review feedback may result in major improvements and updates to your package, including renaming and breaking changes to functions. We do not consider previous publication on CRAN or in other venues sufficient reason to not adopt reviewer or editor recommendations.
- Do not submit your package for review while it or an associated manuscript is also under review at another venue, as this may result on conflicting requests for changes.

5.1.2 Conflict of interest for reviewers/editors

Following criteria are meant to be a guide for what constitutes a conflict of interest for an editor or reviewer. The potential editor or reviewer has a conflict of interest if:

- The authors with a major role are from the potential reviewer/editor's institution or institutional component (e.g., department)
- Within in the past three years, the potential reviewer/editor has been a collaborator or has had any other professional relationship with any person on the package who has a major role
- The potential reviewer/editor serves as a member of the advisory board for the project under review
- The potential reviewer/editor would receive a direct or indirect financial benefit if the package is accepted
- The potential reviewer/editor has significantly contributed to a competitor project.
- There is also a lifetime COI for the family members, business partners, and thesis student/advisor or mentor.

In the case where none of the associate editors can serve as editor, an external guest editor will be recruited.

5.2 Aims and Scope

rOpenSci aims to support packages that enable reproducible research and managing the data lifecycle for scientists. Packages submitted to rOpenSci should fit into one or more of the categories outlined either below. Statistical software may also be submitted for peer review, for which we have a separate set of guidelines and standards. The categories below are for general, and not statistical, software, while the remainder of this chapter applies to both kinds of software. If you are unsure whether your package fits into one of the general or statistical categories, please open an issue as a pre-submission inquiry (**Examples**).

As this is a living document, these categories may change through time and not all previously onboarded packages would be in-scope today. For instance, data visualization packages are no longer in-scope. While we strive to be consistent, we evaluate packages on a case-by-case basis and may make exceptions.

Note that not all rOpenSci projects and packages are in-scope or go through peer review. Projects developed by staff or at conferences may be experimental, exploratory, address core infrastructure priorities and thus not fall into these categories. Look for the peer-review badge - see below - to identify peer-reviewed packages in the rOpenSci repository.

5.2.1 Package categories

- **data retrieval**: Packages for accessing and downloading data from online sources with scientific applications. Our definition of scientific applications is broad, including data storage services, journals, and other remote servers, as many data sources may be of interest to researchers. However, retrieval packages should be focused on data *sources* / *topics*, rather than *services*. For example a general client for Amazon Web Services data storage would not be in-scope. (Examples: **rotl**, **gutenbergr**)
- **data extraction**: Packages that aid in retrieving data from unstructured sources such as text, images and PDFs, as well as parsing scientific data types and outputs from scientific equipment. Statistical/ML libraries for modeling or prediction are typically not included in this category, nor are code parsers. Trained models that act as utilities (e.g., for optical character recognition), may qualify. (Examples: **tabulizer** for extracting tables from PDF documents, **genbankr** for parsing files from GenBank, **treeio** for phylogentic reading in phylogentic tree files, **lightr** for parsing files from spectroscopic instruments))

- **data munging**: Packages for processing data from formats above. This area does not include broad data manipulations tools such as **reshape2** or **tidyr**, or tools for extracting data from R code itself. Rather, it focuses on tools for handling data in specific scientific formats generated from scientific workflows or exported from scientific instruments. (Examples: **plateR** for reading in data structured as plate maps for scientific instruments, or **phonfieldwork** for processing annotated audio files for phonics research)

- **data deposition**: Packages that support deposition of data into research repositories, including data formatting and metadata generation. (Example: **EML**)

- **data validation and testing**: Tools that enable automated validation and checking of data quality and completeness as part of scientific workflows. (Example: **assertr**)

- **workflow automation**: Tools that automate and link together workflows, such as build systems and tools to manage continuous integration. Does not include general tools for literate programming. (e.g., R markdown extensions not under the previous topics). (Example: **drake**)

- **version control**: Tools that facilitate the use of version control in scientific workflows. Note that this does not include all tools that interact with online version control services (e.g., GitHub), unless they fit into another category. (Example: **git2rdata**)

- **citation management and bibliometrics**: Tools that facilitate managing references, such as for writing manuscripts, creating CVs or otherwise attributing scientific contributions, or accessing, manipulating or otherwise working with bibliometric data. (Example: **RefManageR**)

- **scientific software wrappers**: Packages that wrap non-R utility programs used for scientific research. These programs must be specific to research fields, not general computing utilities. Wrappers must be non-trivial, in that there must be significant added value above simple `system()` call or bindings, whether in parsing inputs and outputs, data handling, etc. Improved installation process, or extension of compatibility to more platforms, may constitute added value if installation is complex. This does not include wrappers of other R packages or C/C++ libraries that can be included in R packages. We strongly encourage wrapping open-source and open-licensed utilities - exceptions will be evaluated case-by-case, considering whether open-source options exist. (Examples: **babette**, **nlrx**)

- **field and laboratory reproducibility tools**: Packages that improve reproducibility of real-world workflows through standardization and automation of field and lab protocols, such as sample tracking and tagging, form and data sheet generation, interfacing with laboratory equipment or information systems, and executing experimental designs. (Example: **baRcodeR**)

- **database software bindings**: Bindings and wrappers for generic database APIs (Example: **rrlite**)

In addition, we have some *specialty topics* with a slightly broader scope.

- **geospatial data**: We accept packages focused on accessing geospatial data, manipulating geospatial data, and converting between geospatial data formats. (Examples: **osmplotr**, **tidync**).
- **text data**: We include packages that process and manage text and language data. This is limited to text processing/munging/management (e.g., tokenization, stemming, conversion to and between structured text formats, text metadata and repository access, etc.). Machine-learning and packages implementing NLP analysis algorithms should be submitted under *statistical* software peer review. The scope for this topic is not fully defined, please open a pre-submission inquiry if you are considering submitting a package that falls under this topic. (Example: **tokenizers**)

5.2.2 Other scope considerations

Packages should be *general* in the sense that they should solve a problem as broadly as possible while maintaining a coherent user interface and code base. For instance, if several data sources use an identical API, we prefer a package that provides access to all the data sources, rather than just one.

Packages that include interactive tools to facilitate researcher workflows (e.g., shiny apps) must have a mechanism to make the interactive workflow reproducible, such as code generation or a scriptable API.

For packages that are not in the scope of rOpenSci, we encourage submitting them to CRAN, BioConductor, as well as other R package development initiatives (e.g., cloudyr), and software journals such as JOSS, JSS, or the R journal, depending on the current scopes of those journals.

Note that the packages developed internally by rOpenSci, through our events or through collaborations are not all in-scope for our Software Peer Review process.

5.2.3 Package overlap

rOpenSci encourages competition among packages, forking and re-implementation as they improve options of users overall. However, as we want packages in the rOpenSci suite to be our top recommendations for the tasks they perform, we aim to avoid duplication of functionality of existing R packages in any repo without significant improvements. An R package that replicates the functionality of an existing R package may be considered for inclusion in the rOpenSci suite if it significantly improves on alternatives in any repository (RO, CRAN, BioC) by being:

- More open in licensing or development practices
- Broader in functionality (e.g., providing access to more data sets, providing a greater suite of functions), but not only by duplicating additional packages
- Better in usability and performance
- Actively maintained while alternatives are poorly or no longer actively maintained

These factors should be considered *as a whole* to determine if the package is a significant improvement. A new package would not meet this standard only by following our package guidelines while others do not, unless this leads to a significant difference in the areas above.

We recommend that packages highlight differences from and improvements over overlapping packages in their README and/or vignettes.

We encourage developers whose packages are not accepted due to overlap to still consider submittal to other repositories or journals.

5.3 Package ownership and maintenance

5.3.1 Role of the rOpenSci team

Authors of contributed packages essentially maintain the same ownership they had prior to their package joining the rOpenSci suite. Package authors will continue to maintain and develop their software after acceptance into rOpenSci. Unless explicitly added as collaborators, the rOpenSci team will not interfere much with day to day operations. However, this team may intervene with critical bug fixes, or address urgent issues if package authors do not respond in a timely manner (see the section about maintainer responsiveness).

5.3.2 Maintainer responsiveness

If package maintainers do not respond in a timely manner to requests for package fixes from CRAN or from us, we will remind the maintainer a number of times, but after 3 months (or shorter time frame, depending on how critical the fix is) we will make the changes ourselves.

The above is a bit vague, so the following are a few areas of consideration.

- Examples where we'd want to move quickly:
  - Package `foo` is imported by one or more packages on CRAN, and `foo` is broken, and thus would break its reverse dependencies.

  - Package `bar` may not have reverse dependencies on CRAN, but is widely used, thus quickly fixing problems is of greater importance.

- Examples where we can wait longer:
  - Package `hello` is not on CRAN, or on CRAN, but has no reverse dependencies.
  - Package `world` needs some fixes. The maintainer has responded but is simply very busy with a new job, or other reason, and will attend to soon.

We urge package maintainers to make sure they are receiving GitHub notifications, as well as making sure emails from rOpenSci staff and CRAN maintainers are not going to their spam box. Authors of onboarded packages will be invited to the rOpenSci Slack to chat with the rOpenSci team and the greater rOpenSci community. Anyone can also discuss with the rOpenSci community on the rOpenSci discussion forum.

Should authors abandon the maintenance of an actively used package in our suite, we will consider petitioning CRAN to transfer package maintainer status to rOpenSci.

5.3.3 Quality commitment

rOpenSci strives to develop and promote high quality research software. To ensure that your software meets our criteria, we review all of our submissions as part of the Software Peer Review process, and even after acceptance will continue to step in with improvements and bug fixes.

Despite our best efforts to support contributed software, errors are the responsibility of individual maintainers. Buggy, unmaintained software may be removed from our suite at any time.

5.3.4 Package removal

In the unlikely scenario that a contributor of a package requests removal of their package from the suite, we retain the right to maintain a version of the package in our suite for archival purposes.

5.4 Ethics, Data Privacy and Human Subjects Research

rOpenSci packages and other tools are used for a variety of purposes, but our focus is on tools for research. We expect that tools will enable ethical use by research practitioners, who are obligated to adhere to ethical codes such Declaration of Helsinki and The Belmont Report. Researchers bear responsibility for their use of software,

but software developers must consider the ethical use of their products, and developers themselves adhere to ethical codes for computer professionals such as those expressed by IEEE and ACM. rOpenSci contributors often play both the role of both researcher and developer.

We ask that software developers place themselves in researchers' role and consider the requirements of an ethical workflow using authors' software. Given the variation and degree of flux of ethical approaches for Internet-based analyses, judgement calls rather than recipes are required. The Ethical Guidelines of The Association of Internet Researchers provides a robust framework and we encourage authors, editors, and reviewers to use this in evaluating their work. In general, adherence to legal or regulatory minimum requirements may not be sufficient, though these (e.g., GDPR), may be relevant. Package authors should direct users to relevant resources for the ethical use of the software.

Some packages, due to the nature of data they handle, may be determined by editors to require enhanced scrutiny. For these, editors may require additional (or reduced) functionality, and robust documentation, defaults, and warnings to direct users to relevant ethical practices. The following topics may merit enhanced scrutiny:

- ***Vulnerable populations***: Authors of packages and workflows that deal with information related to vulnerable populations bear responsibility to protect them from likely harms.
- ***Personally identifiable or sensitive data***: The release of personally identifiable or sensitive data is potentially harmful. This includes "reasonably re-identifiable" data - which a motivated individual could trace back to the owner or creator even if the data are anonymized. This includes both cases where identifiers (e.g., name, date of birth) are available as part of data, and also if unique pseudonyms/screen names are linked with full-text posts, through which one can link back individuals through cross-reference with other data sets.

While the best response to ethical concerns will be context-specific, these general guidelines should be followed by packages where the challenges above arise:

- Packages should adhere to data source's terms of use, as expressed in website Terms and Conditions, "robots.txt" files, privacy policies, and other relevant restrictions, and link to them prominently in package documentation. Packages should provide or document functionality to adhere to such restrictions (e.g., scrape from only allowed endpoints, use appropriate rate limiting in code, examples, or vignettes). Note that while Terms and Conditions, Privacy Policies, etc., may not provide sufficient bounds on ethical usage, they can provide an outer bound.
- A key tool in addressing the risks posed in studying vulnerable populations or using personally identifiable data is ***informed consent***. Package authors

should support users' acquisition of informed consent when relevant. This may include providing links to data source's preferred method of acquiring consent, contact information of data providers (e.g. forum moderators), documentation of informed consent protocols, or getting pre-approval for general uses of a package.

Note that consent is not implicitly granted just because data are accessible. Accessible data are not necessarily public, as different persons and contexts have different normative expectations of privacy (see work by Social Data Lab).

- Packages accessing personally identifiable information should take special care to follow security best practices (e.g., exclusive use of secure internet protocols, strong mechanisms for storing credentials, etc.).

- Packages that access or handle personally identifiable or sensitive data should enable, document, and demonstrate workflows for de-identification, secure storage, other best practices to minimize risk of harm.

As standards for data privacy and research continue to evolve, we welcome input from authors on considerations specific to their software and supplemental documentation such as approval from university ethics review boards. These may be attached to issue threads of package submissions or pre-submission inquiries, or conveyed directly to editors if needed. General suggestions may be filed as issues in this book's repository.

5.4.1 Resources

The following resources may be helpful for researchers, package authors, editors and reviewers in addressing ethical questions related to privacy and research software.

- The Declaration of Helsinki and The Belmont Report provide fundamental principles for ethical practice by researchers.
- Several organizations provide guidance on how to translate these principles into the context of internet research. These include the Ethical Guidelines of The Association of Internet Researchers, the NESH Guide to Internet Research Ethics, and BPS' Ethics Guidelines for Internet-Mediated Research. Anabo et al (2019) provide a helpful overview of these.
- The Social Media Lab provides a high-level overview with data on normative expectations of privacy and use on social forums.
- Bechmann A., Kim J.Y. (2019) Big Data: A Focus on Social Media Research Dilemmas. In: Iphofen R. (eds) Handbook of Research Ethics and Scientific Integrity. https://doi.org/10.1007/978-3-319-76040-7_18-1
- Chu, K.-H., Colditz, J., Sidani, J., Zimmer, M., & Primack, B. (2021). Re-evaluating standards of human subjects protection for sensitive

health data in social media networks. Social Networks, 67, 41–46. https://dx.doi.org/10.1016/j.socnet.2019.10.010
- Lomborg, S., & Bechmann, A. (2014). Using APIs for Data Collection on Social Media. The Information Society, 30(4), 256–265. https://dx.doi.org/10.1080/01972243.2014.915276
- Flick, C. (2016). Informed consent and the Facebook emotional manipulation study. *Research Ethics*, *12*(1), 14–28. https://doi.org/10.1177/1747016115599568
- Sugiura, L., Wiles, R., & Pope, C. (2017). Ethical challenges in online research: Public/private perceptions. *Research Ethics*, *13*(3–4), 184–199. https://doi.org/10.1177/1747016116650720
- Taylor, J., & Pagliari, C. (2018). Mining social media data: How are research sponsors and researchers addressing the ethical challenges? Research Ethics, 14(2), 1–39. https://doi.org/10.1177/1747016117738559
- Zimmer, M. (2010). "But the data is already public": on the ethics of research in Facebook. Ethics and Information Technology, 12(4), 313–325. https://dx.doi.org/10.1007/s10676-010-9227-5

5.5 Code of Conduct

rOpenSci's community is our best asset. Whether you're a regular contributor or a newcomer, we care about making this a safe place for you and we've got your back. We have a Code of Conduct that applies to all people participating in the rOpenSci community, including rOpenSci staff and leadership and to all modes of interaction online or in person. The Code of Conduct is maintained on the rOpenSci website.

Chapter 6

Guide for Authors

This concise guide presents the software peer review process for you as a package author.

6.1 Planning a Submission (or a Pre-Submission Enquiry)

- Do you expect to maintain your package for at least 2 years, or to be able to identify a new maintainer?
- Consult our policies see if your package meets our criteria for fitting into our suite and does not overlap with other packages.
  - If you are unsure whether a package meets our criteria, feel free to open an issue as a pre-submission inquiry to ask if the package is appropriate.
- Please consider the best time in your package's development to submit. Your package should be sufficiently mature so that reviewers are able to review all essential aspects, but keep in mind that review may result in major changes.
  - We strongly suggest submitting your package for review *before* publishing on CRAN or submitting a software paper describing the package to a journal. Review feedback may result in major improvements and updates to your package, including renaming and breaking changes to functions.
  - Do not submit your package for review while it or an associated manuscript is also under review at another venue, as this may result in conflicting requests for changes.
- Please also consider the time and effort needed to respond to reviews: think about your availability or that of your collaborators in the next weeks and

months following a submission. Note that reviewers are volunteers, and we ask that you respect their time and effort by responding in a timely and respectful manner.
- If you use repostatus.org badges (which we recommend), submit when you're ready to get an *Active* instead of *WIP* badge. Similarly, if you use lifecycle badges, submission should happen when the package is at least *Maturing*.
- For any submission or pre-submission inquiry the README of your package should provide enough information about your package (goals, usage, similar packages) for the editors to assess its scope without having to install the package. Even better, set up a pkgdown website for allowing more detailed assessment of functionality online.

  - At the submission stage, all major functions should be stable enough to be fully documented and tested; the README should make a strong case for the package.
  - Your README file should strive to explain your package's functionality and aims, assuming readers have little to no domain knowledge. All technical tems, including references to other software, should be clarified.

- Your package will continue to evolve after review, the chapter on *Package evolution* provides guidance about the topic.

6.2 Preparing for Submission

- Read and follow our packaging style guide, reviewer's guide to ensure your package meets our style and quality criteria.
- Feel free to ask any questions about the process, or your specific package, in our Discussion Forum.
- All submissions are automatically checked by our pkgcheck system to ensure packages follow our guidelines. All authors are expected to have run the main `pkgcheck` function locally to confirm that the package is ready to be submitted. Alternatively, an even easier way to ensure a package is ready for submission is to use the `pkgcheck` GitHub Action to run `pkgcheck` as a GitHub Action, as described in our blog post.
- If there are any aspects of `pkgcheck` which your package is unable to pass, please explain reasons in your submission template.
- If you feel your package is in scope for the Journal of Open-Source Software (JOSS), do not submit it to JOSS consideration until after the rOpenSci review process is over: if your package is deemed in scope by JOSS editors, only the accompanying short paper would be reviewed, (not the software that will have been extended reviewed by rOpenSci by that time). Not all rOpenSci packages will meet the criteria for JOSS.

6.3 The Submission Process

- Software is submitted for review by opening a new issue in the software review repository and filling out the template.
- The template begins with a section which includes several HTML-styled variables (`<!---variable--->`). These are used by our `ropensci-review-bot`, and must be left in place, with values filled between the indicated start and end points, like this:

```
<!---variable--->insert value here<!---end-variable>
```

- Communication between authors, reviewers and editors will first and foremost take place on GitHub so that the review thread can serve as a full record of the review. You may choose to contact the editor by email or Slack if private consultation is needed (e.g., asking how to respond to a reviewer question). Do not contact reviewers off-thread without asking them in the GitHub thread whether they agree to it.
- *When submitting a package please make sure your GitHub notification settings make it unlikely you will miss a comment.*
- Packages are automatically checked on submission by our `pkgcheck` system, which will confirm whether or not a package is ready to be reviewed.

6.4 The Review Process

- An editor will review your submission within 5 business days and respond with next steps. The editor may assign the package to reviewers, request that the package be updated to meet minimal criteria before review, or reject the package due to lack of fit or overlap.
- If your package meets minimal criteria, the editor will assign 1-3 reviewers. They will be asked to provide reviews as comments on your issue within 3 weeks.
- We ask that you respond to reviewers' comments within 2 weeks of the last-submitted review, but you may make updates to your package or respond at any time. Your response should include a link to the updated NEWS.md of your package. Here is an author response example. We encourage ongoing conversations between authors and reviewers. See the reviewing guide for more details.
- Any time package changes are likely to alter the results of the automated `pkgcheck` checks, authors can request a re-check with the command, `@ropensci-review-bot check package`.
- Please notify us immediately if you are no longer able to maintain your package or to respond to reviews. You will then be expected to either retract a submission, or to find alternative package maintainers. You can also discuss maintenance issues in the rOpenSci slack workspace.

- Once your package is approved, we will provide further instructions about the transfer of your repository to the rOpenSci repository.

Our code of conduct is mandatory for everyone involved in our review process.

Chapter 7

Guide for Reviewers

Thanks for accepting to review a package for rOpenSci! This chapter consists of our guidelines to prepare, submit and follow up on your review.

You might contact the editor in charge of the submission for any question you might have about the process or your review.

Please strive to complete your review within 3 weeks of accepting a review request. We will aim to remind reviewers of upcoming and past due dates. Editors may assign additional or alternate reviewers if a review is excessively late.

If you use our standards/checklists etc. when reviewing software elsewhere, do tell the recipients (e.g. journal editors, students, internal code review) that they came from rOpenSci, and tell us in our public forum, or privately by email.

7.1 Preparing your review

All submissions trigger a detailed report on package structure and functionality, generated by our `pkgcheck` package. If the package has changed substantially since the last checks, you may request a re-check with the command `@ropensci-review-bot check package`. Note that when installing the package to review it, you should use the `dependencies = TRUE` argument of `remotes::install()` to make sure you have all dependencies available.

7.1.1 General guidelines

To review a package, please begin by copying our review template (or our review template in Spanish) and using it as a high-level checklist. In addition to checking off the minimum criteria, we ask that you provide general comments addressing the following:

- Does the code comply with general principles in the Mozilla reviewing guide?
- Does the package comply with the rOpenSci packaging guide?
- Are there improvements that could be made to the code style?
- Is there code duplication in the package that should be reduced?
- Are there user interface improvements that could be made?
- Are there performance improvements that could be made?
- Is the documentation (installation instructions/vignettes/examples/demos) clear and sufficient? Does it use the principle of *multiple points of entry* i.e. takes into account the fact that any piece of documentation may be the first encounter the user has with the package and/or the tool/data it wraps?
- Were functions and arguments named to work together to form a common, logical programming API that is easy to read, and autocomplete?
- If you have your own relevant data/problem, work through it with the package. You may find rough edges and use-cases the author didn't think about.

Please be respectful and kind to the authors in your reviews. Our code of conduct is mandatory for everyone involved in our review process. We expect you to submit your review within 3 weeks, depending on the deadline set by the editor. Please contact the editor directly or in the submission thread to inform them about possible delays.

We encourage you to use automated tools to facilitate your reviewing. These include:

- Checking the initial package report generated by our `@ropensci-review-bot`.
- Checking the package's logs on its continuous integration services (GitHub Actions, Codecov, etc.)
- Running `devtools::check()` and `devtools::test()` on the package to find any errors that may be missed on the author's system.
- Seeing whether tests' skipping is justified (e.g. `skip_on_cran()` tests that do real API requests vs. skipping all tests on one operating system).

Reviewers may also re-generate package check results from `@ropensci-review-bot` at any time by issuing the single comment in a review issue: `@ropensci-review-bot check package`.

7.1.2 Off-thread interactions

If you interact with the package authors and talked about the review outside a review thread (in chats, DMs, in-person, issues in the project repository), please make sure that your review captures and/or links to elements from these conversations that are relevant to the process.

7.1.3 Experience from past reviewers

First-time reviewers may find it helpful to read (about) some previous reviews. In general you can find submission threads of onboarded packages here. Here are a few chosen examples of reviews (note that your reviews do not need to be as long as these examples):

- `rtika` review 1 and review 2
- `NLMR` review 1 and review 2
- `bowerbird` pre-review comment, review 1, review 2.
- `rusda` review (from before we had a review template)

You can read blog posts written by reviewers about their experiences via this link. In particular, in this blog post by Mara Averick read about the "naive user" role a reviewer can take to provide useful feedback even without being experts of the package's topic or implementation, by asking themselves *"What did I think this thing would do? Does it do it? What are things that scare me off?"*. In another blog post Verena Haunschmid explains how she alternated between using the package and checking its code.

As both a former reviewer and package author Adam Sparks wrote this "[write] a good critique of the package structure and best coding practices. If you know how to do something better, tell me. It's easy to miss documentation opportunities as a developer, as a reviewer, you have a different view. You're a user that can give feedback. What's not clear in the package? How can it be made more clear? If you're using it for the first time, is it easy? Do you know another R package that maybe I should be using? Or is there one I'm using that perhaps I shouldn't be? If you can contribute to the package, offer."

7.1.4 Helper package for reviewers

If working in RStudio, you can streamline your review workflow by using the `pkgreviewr` package created by associated editor Anna Krystalli. Say you accepted to review the `refnet` package, you'd write

```
remotes::install_github("ropensci-org/pkgreviewr")
library(pkgreviewr)
pkgreview_create(pkg_repo = "embruna/refnet",
                 review_parent = "~/Documents/workflows/rOpenSci/reviews/")
```

and

- the GitHub repo of the `refnet` package will be cloned.
- a review project will be created, containing a notebook for you to fill, and the review template.

7.1.5 Feedback on the process

We encourage you to ask questions and provide feedback on the review process on our forum.

7.2 Submitting the Review

- When your review is complete, paste it as a comment into the package software-review issue.
- Additional comments are welcome in the same issue. We hope that package reviews will work as an ongoing conversation with the authors as opposed to a single round of reviews typical of academic manuscripts.
- You may also submit issues or pull requests directly to the package repo if you choose, but if you do, please comment about them and link to them in the software-review repo comment thread so we have a centralized record and text of your review.
- Please include an estimate of how many hours you spent on your review afterwards.

7.3 Review follow-up

Authors should respond within 2 weeks with their changes to the package in response to your review. At this stage, we ask that you respond as to whether the changes sufficiently address any issues raised in your review. We encourage ongoing discussion between package authors and reviewers, and you may ask editors to clarify issues in the review thread as well.

You'll use the approval template.

Chapter 8

Guide for Editors

Software Peer Review at rOpenSci is managed by a team of editors. We are piloting a system of a rotating Editor-in-Chief (EiC).

This chapter presents the responsabilities of the Editor-in-Chief, of any editor in charge of a submission, how to respond to an out-of-scope submission and how to manage a dev guide release.

If you're a guest editor, thanks for helping! Please contact the editor who invited you to handle a submission for any question you might have.

8.1 Editors' responsibilities

- In addition to handling packages (about 4 a year), editors weigh in on group editorial decisions, such as whether a package is in-scope, and determining updates to our policies. We generally do this through Slack, which we expect editors to be able to check regularly.
- We also rotate Editor-in-Chief responsibilities (first-pass scope decisions and assigning editors) amongst the board about quarterly.
- You do not have to keep track of other submissions, but if you do notice an issue with a package that is being handled by another editor, feel free to raise that issue directly with the other editor, or post the concern to editors-only channel on slack. Examples:
  - You know of an overlapping package, that hasn't been mentioned in the process yet.
  - You see a question to which you have an expert answer that hasn't been given after a few days (e.g. you know of a blog post tackling how to add images to package docs).

  - Concerns related to e.g. the speed of the process should be tackled by the editor-in-chief so that's who you'd turn to for such questions.

8.2 Handling Editor's Checklist

8.2.1 Upon submission:

- If you're the EiC or the first editor to respond, assign an editor with a comment of `@ropensci-review-bot assign @username as editor`. This will also add tag `1/editor-checks` to the issue.
- Submission will automatically generate package check output from ropensci-review-bot which should be examined for any outstanding issues (most exceptions will need to be justified by the author in the particular context of their package.). If you want to re-run checks after any package change post a comment `@ropensci-review-bot check package`.
- After automatic checks are posted, use the editor template to guide initial checks and record your response to the submission. You can also streamline your editor checks by using the `pkgreviewr` package created by associate editor Anna Krystalli. Please strive to finish the checks and start looking for reviewers within 5 working days.
- Check that template has been properly filled out.
- Check against policies for fit and overlap. Initiate discussion via Slack #software-review channel if needed for edge cases that haven't been caught by previous checks by the EiC. If reject, see this section about how to respond.
- Check that mandatory parts of template are complete. If not, direct authors toward appropriate instructions.
- For packages needing continuous integration on multiple platforms (cf criteria in this section of the CI chapter) make sure the package gets tested on multiple platforms (having the package built on several operating systems via GitHub Actions for instance).
- Wherever possible when asking for changes, direct authors to automatic tools such as `usethis` and `styler`, and to online resources (sections of this guide, sections of the R packages book) to make your feedback easier to use. Example of editor's checks.
- Ideally, the remarks you make should be tackled before reviewers start reviewing.
- If initial checks show major gaps, request changes before assigning reviewers. If the author mentions changes might take time, apply the holding label via typing `@ropensci-review-bot put on hold`. You'll get a reminder every 90 days (in the issue) to check in with the package author(s).
- If the package raises a new issue for rOpenSci policy, start a conversation in Slack or open a discussion on the rOpenSci forum to discuss it with other editors (example of policy discussion).

8.2.2 Look for and assign two reviewers:

8.2.2.1 Tasks

- Comment with `@ropensci-review-bot seeking reviewers`.
- Use the email template if needed for inviting reviewers
  - When inviting reviewers, include something like "if I don't hear from you in a week, I'll assume you are unable to review," so as to give a clear deadline when you'll move on to looking for someone else.
- Assign reviewers with `@ropensci-review-bot assign @username as reviewer`. `add` can also be used instead of `assign`, and `to reviewers` (plural) instead of `as reviewer` (single). The following is thus also valid: `@ropensci-review-bot add @username to reviewers`. One command should be issued for each reviewer. If needed later, remove reviewers with `@ropensci-review-bot remove @username from reviewers`.
- If you want to change the due date for a review use `@ropensci-review-bot set due date for @username to YYYY-MM-DD`.

8.2.2.2 How to look for reviewers

8.2.2.2.1 Where to look for reviewers? As a (guest) editor, use

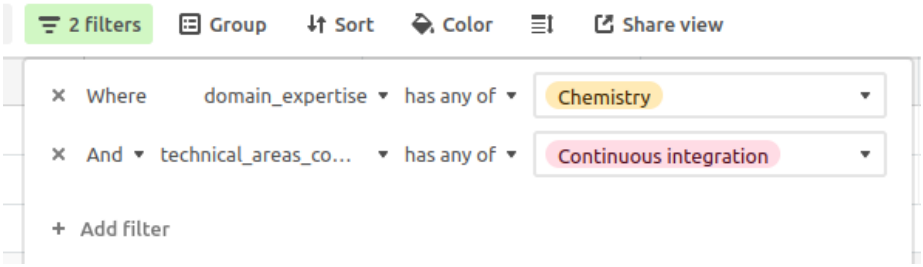
- the potential suggestions made by the submitter(s), (although submitters may have a narrow view of the types of expertise needed. We suggest not using more than one of suggested reviewers);
- the Airtable database of reviewers and volunteers (see next subsection);
- and the authors of rOpenSci packages.

When these sources of information are not enough,

- ping other editors in Slack for ideas,
- look for users of the package or of the data source/upstream service the package connects to (via their opening issues in the repository, starring it, citing it in papers, talking about it on Twitter).
- You can also search for authors of related packages on r-pkg.org.
- R-Ladies has a directory specifying skills and interests of people listed.
- You might tweet about the reviewer search.

8.2.2.2.2 Tips for reviewer search in Airtable Your tools for searching reviewers in the Airtable are

- filters (please remove them once you're done), see example below.



- creating a duplicated view, depending on your Airtable skills.

Have a look at the `last_time_contacted` and `last_answer` columns before contacting someone! If someone recently refused because they were busy, it might be best to abstain, whereas someone who refused because of a COI could be contacted again without waiting too long.

8.2.2.2.3 Storing information from the reviewer search in Airtable For people listed in the Airtable, if you contact them about a review please update the `last_time_contacted` column, and enter the category corresponding to their answer in `last_answer`.

Only add people to Airtable if they accept to review (otherwise, people should volunteer themselves by filling the Airtable form). Do not enter any information other than GitHub username, name and email address yourself. Point reviewers to the Airtable form.

8.2.2.2.4 Criteria for choosing a reviewer Here are criteria to keep in mind when choosing a reviewer. You might need to piece this information together by searching CRAN and the potential reviewer's GitHub page and general online presence (personal website, Twitter).

- Has not reviewed a package for us within the last 6 months.
- Some package development experience.
- Some domain experience in the field of the package or data source
- No conflicts of interest.
- Try to balance your sense of the potential reviewer's experience against the complexity of the package.
- Diversity - with two reviewers both shouldn't be cis white males.
- Some evidence that they are interested in openness or R community activities, although blind emailing is fine.

Each submission should be reviewed by *two* package reviewers. Although it is fine for one of them to have less package development experience and more domain

knowledge, the review should not be split in two. Both reviewers need to review the package comprehensively, though from their particular perspective. In general, at least one reviewer should have prior reviewing experience, and of course inviting one new reviewer expands our pool of reviewers.

8.2.3 During review:

- Check in with reviewers and authors occasionally. Offer clarification and help as needed.
- In general aim for 3 weeks for review, 2 weeks for subsequent changes, and 1 week for reviewer approval of changes.
- Upon each review being submitted,
  - Write a comment thanking the reviewer with your words;
  - Record the review via typing a new comment `@ropensci-review-bot submit review <review-url> time <time in hours>`. E.g. for the review https://github.com/ropensci/software-review/issues/329#issuecomment-809783937 the comment would be `@ropensci-review-bot submit review https://github.com/ropensci/software-review/issues/329#issuecomment-809783 time 4`.
- If the author stops responding, refer to the policies and/or ping the other editors in the Slack channel for discussion. Importantly, if a reviewer was assigned to a closed issue, contact them when closing the issue to explain the decision, thank them once again for their work, and make a note in our database to assign them to a submission with high chances of smooth software review next time (e.g. a package author who has already submitted packages to us).
- Upon changes being made, change the review status tag to `5/awaiting-reviewer-response`, and request that reviewers indicate approval with the reviewer approval template.

8.2.4 After review:

- `@ropensci-review-bot approve <package-name>`
- *If the original repository owner opposes transfer, add a line with its address to this repos list to ensure the package gets included in rOpenSci package registry.*
- Nominate a package to be featured in an rOpenSci blog post or tech note if you think it might be of high interest. Please note in the software review issue one or two things the author could highlight, and tag `@ropensci/blog-editors` for follow-up.
- If authors maintain a gitbook that is at least partly about their package, contact an rOpenSci staff member so they might contact the authors about transfer to the `ropensci-books` GitHub organisation.

8.2.5 Package promotion:

- Direct the author to the chapters of the guide about package releases, marketing and GitHub grooming.

8.3 EiC Responsibilities

The EiC serves for 3 months or a time agreed to by all members of the editorial board. The EiC is entitled to taking scope and overlap decisions as independently as possible (but can still request help/advice). In details, the EiC plays the following roles

- Watches all issues posted to the software-review repo (either subscribe to repo notifications on GitHub, or watch the `#software-peer-review-feed` channel on Slack).
- Tags issue with `0/editorial-team-prep`
- Assigns package submissions to other editors, including self, to handle. Mostly this just rotates among editors, unless the EiC thinks an editor is particularly suited to a package, or an editor declines handling the submission due to being too busy or because of conflicting interests.

  ```
  @ropensci-review-bot assign @username as editor
  ```

- Monitors pace of review process and reminds other editors to move packages along as needed.
- On assuming EiC rotation, reviews status of current open reviews and reminds editors to respond or update status as needed.
- Responds to issues posted to the software-review-meta repo
- Makes decisions on scope/overlap for pre-submission inquiries, referrals from JOSS or other publication partners, and submissions if they see an ambiguous case (This last case may also be done by handling editors (see below)). To initiate discussion, this is posted to the rOpenSci Slack editors-only channel along with a small summary of what the (pre-)submitted/referred submission is about, what doubts the EiC has i.e. digesting information a bit. If after one day or two the EiC feels they haven't received enough answers, they can ping all editors.
  - Any editor should feel free to step in on these. See this section about how to respond to out-of-scope (pre-) submissions.
  - After explaining the out-of-scope decision, write an issue comment `@ropensci-review-bot out-of-scope`.
- Requests a new EiC when their rotation is up (set a calendar reminder ahead of your expected end date and ask for volunteers in the editors' Slack channel)

8.3.1 Asking for more details

In some cases online documentation is sparse. Minimal README, no pkgdown website make assessment harder. In that case please ask for more details: even if the package is deemed out-of-scope, the package docs will have gotten better so we are fine asking for these efforts.

Example text

```
Hello <username> and many thanks for your submission.

We are discussing whether the package is in scope and need a bit more information.

Would you mind adding more details and context to the README?
After reading it someone with little domain knowledge should have been informed about the aim, go

<optional>
If a package has overlapping functionality with other packages, we require it to demonstrate in t
</optional>
```

8.3.2 Inviting a guest editor

After discussion with other editors the EiC might invite a guest editor to handle a submission (e.g. if submission volume is large, if all editors have a conflict of interest, if specific expertise is needed, or as a trial prior to inviting a person to join the editorial board).

When inviting a guest editor,

- Ask about conflicts of interest using the same phrasing as for reviewers,
- Give a link to the guide for editors.

If the person said yes (yay!),

- Make sure they enabled 2FA for their GitHub account,
- Invite them to the `ropensci/editors` team and to the ropensci organization,
- Once they've accepted this repo invitation, assign the issue to them,
- Ensure they're (already) invited to rOpenSci Slack workspace,
- Add their name to the Airtable guest-editor table (so their names might appear in this book and in the software-review README).

After the review process is finished (package approved, issue closed),

- Thank the guest editor again,
- Remove them from the `ropensci/editors` team (but not from the ropensci organization).

8.4 Responding to out-of-scope submissions

Thank authors for their submission, explain the reasons for the decision, and direct them to other publication venues if relevant, and to the rOpenSci discussion forum. Use wording from Aims and scope in particular regarding the evolution of scope over time, and the overlap and differences between unconf/staff/software-review development.

Examples of out-of-scope submissions and responses.

8.5 Answering reviewers' questions

Reviewers might ask for feedback on e.g. the tone of their review. Beside pointing them at general guidance in this guide, asking editors / opening an issue when such guidance is lacking, here are some review examples that might be useful.

- tough-but-constructive example: the part of this review suggesting a re-write of the vignette: ropensci/software-review#191 (comment).
- the `slopes` package, which ended up being fundamentally redesigned in response to the reviews. All reviews/reviewers were at all times entirely constructive, which seems to have played a major role in motivating the authors to embark on such a major overhaul. Comments such as, *"this package does not …"* or *"has not …"* were invariably followed by constructive suggestions for what could be done (there are, for example, several in one of the first reviews).
- tic reviews politely expressed reservations: https://github.com/ropensci/software-review/issues/305#issuecomment-504762517 and https://github.com/ropensci/software-review/issues/305#issuecomment-508271766
- bowerbird useful "pre-review" that resulted in a package split before the actual reviews.

8.6 Managing a dev guide release

If you are in charge of managing a release of the very book you are reading, use the book release guidance as an issue template to be posted in the dev guide issue tracker, and do not hesitate to ask questions to other editors.

8.6.1 Dev guide governance

For very small amendments to the dev guide, no PR review is needed. For larger amendments, request review from at least a few editors (if none participated in the

discussion related to the amendment, request a review from all of them on GitHub, and in the absence of any reaction merge after a week).

Two weeks before a dev guide release, once the PR from dev to master **and the release blog post** are ready for review, all editors should be pinged by GitHub ("review request" on the PR from dev to master) and Slack, but the release doesn't need all of them to explicitly approve the release.

8.6.2 Blog post about a release

The blog post about a release will be reviewed by editors, and one of `@ropensci/blog-editors`.

8.6.2.1 Content

Refer to the general rOpenSci blogging guidance, and the more specific guidance below.

First example of such a post; second example.

The blog post should mention all important items from the changelog organized in (sub)sections: e.g. a section about big change A, another one about big change B, and one about smaller changes lumped together. Mention the most important changes first.

For each change made by an external contributor, thank them explicitly using the information from the changelog. E.g.

```
[Matt Fidler](https://github.com/mattfidler/)
amended our section on Console messages [ropensci/dev_guide#178](https://github.com/ropensci/dev_
```

At the end of the post, mention upcoming changes by linking to open issues in the issue tracker, and invite readers to contribute to the dev guide by opening issues and participating in open discussions. Conclusion template:

```
In this post we summarized the changes incorporated into our book ["rOpenSci Packages: Developmen
We are grateful for all contributions that made this release possible.
We are already working on updates for our next version, such as ISSUE1, ISSUE2.
Check out the [the issue tracker](https://github.com/ropensci/dev_guide/issues/) if you'd like to
```

8.6.2.2 Authorship

The editor writing the post is first author, other editors are listed by alphabetical order.

Chapter 9

Editorial management

Guidance for managing the editorial team.

9.1 Recruiting new editors

Recruiting new editors and maintaining a sufficient and well-balanced editorial board is a responsibility of the Software Review Lead, with support and advice from the editorial board.

Steps:

- Start a private channel for discussion (so that it does not have a history in the editors channel that future editors will join, which could be awkward).
- Ping editors to be sure they get a notification as this is an important topic.
- Wait for a majority of editors to chime in before inviting someone. Leave them one week to respond.

9.2 Inviting a new editor

- Candidates might start by being guest editors. When inviting them as guest editor, invite them as you would invite a guest editor for other reasons.
- If a candidate is guest editor first, assess how the process went after the submission is tackled. Asked other editors for their advice again.
- Send an email.

```
We would like to invite you to join the rOpenSci editorial board as a full member. [SPE
We think you would make a wonderful addition to the team.

[IF GUEST EDITOR:You are familiar with the editor's role as you've been a guest editor]
We ask that editors make an informal commitment of serving for two years, and re-evalua
On a short-term basis, any editor can decline to handle a package or say, "I'm pretty b

In addition to handling packages, editors weigh in on group editorial decisions, such a
We generally do this through Slack, which we expect editors to be able to check regular
We have editorial board calls annually.
We also rotate Editor-in-Chief responsibilities (first-pass scope decisions and assigni
You'll have the opportunity to enter this rotation once you have been on the board for
Some of us also take on bigger projects to improve the peer-review process, though this

We hope that you'll join the board!
It's an exciting time for peer-review at rOpenSci.

Please give this some thought, ask us any questions you have, and let us know whether y

Best,
[EDITOR], on behalf of the rOpenSci Editorial Board
```

9.3 Onboarding a new editor

- Inform rOpenSci community manager so that
  - editors are added to the rOpenSci website.
  - an introduction blog post can be put together.
- If they haven't already done so as guest editors, request that the new editor turn on two-factor authentication (2FA) for GitHub.
- Invite editors to the rOpenSci GitHub organization as member, as a member of the `editors` team and the `data-pkg-editors` or `stats-board` sub-team, as appropriate. This will give them appropriate permissions and allow them to get team-specific notifications.
- Editors need access to the AirTable database of software review.
- Editors need access to the private editors channel in rOpenSci Slack workspace (and to the Slack workspace in general if they didn't previously, in that case ask rOpenSci community manager).
- Post a welcome message in the channel, pinging all editors.
- In the Slack workspace they need to be added to the editors team so that `@editors` will ping them too.

- We add editors' names to
  - dev_guide authors list
  - dev_guide chapter introducing software review (at two locations in this file, as editors and a bit below to remove them from the reviewers list)
  - software-review README (in two places in this file as well) Both the dev_guide and software-review README are automatically knit via continuous integration.
- Add editors to https://github.com/orgs/ropensci/teams/editors/members

9.4 Offboarding an editor

- Thank them for their work!
- Remove them from the editors-only channel and the editors Slack team.
- Remove them from https://github.com/orgs/ropensci/teams/editors/members and sub-team.
- Inform rOpenSci community manager or some other staff emember so that they might be moved to alumni team members on the website.
- Remove their access to the Airtable workspace.
- Remove them from
  - dev_guide chapter introducing software review (at two locations in this file, as editors and a bit below to remove them from the reviewers list)
  - software-review README (in two places in this file as well) Both the dev_guide and software-review README are automatically knit via continuous integration.

Part III

Maintaining Packages

Chapter 10

Collaboration Guide

Having contributors will improve your package, and if you onboard some of them as package authors with write permissions to the repo, your package will be more sustainably developed. It can also be very enjoyable to work as a team!

This chapter contains our guidance for collaboration, in a section about making your repo contribution- and collaboration-friendly by infrastructure (code of conduct, contribution guidelines, issue labels); and a section about how to collaborate with new contributors, in particular in the context of the rOpenSci's "ropensci" GitHub organization.

Besides these mostly technical tips, it is important to remember to be kind, and to take others' perspective into account especially when their priorities differ from yours.

10.1 Make your repo contribution and collaboration friendly

10.1.1 Code of conduct

After transfer to our GitHub organization, rOpenSci Code of Conduct will apply to your project. Please add this text to the README

```
Please note that this package is released with a [Contributor
Code of Conduct](https://ropensci.org/code-of-conduct/).
By
contributing to this project, you agree to abide by its terms.
```

And delete the package current code of conduct if there was one.

10.1.2 Contributing guide

You can use issue, pull request and contributing guidelines. Having a contributing file as `.github/CONTRIBUTING.md` or `docs/CONTRIBUTING.md` is compulsory. An easy way to insert a template for a contributing guide is the `use_tidy_contributing()` function from the `usethis` package, which inserts this template as `.github/CONTRIBUTING.md`. A more extensive example is this template by Peter Desmet, or the comprehensive GitHub wiki pages for the mlr3 package. These and other templates will generally need to be modified for use with an rOpenSci package, particularly by referring and linking to our Code of Conduct, as described elsewhere in this book. Modifying a generic contributing guide to add a personal touch also tends to make it look less generic and more sincere. Personal preferences in a contributing guide include:

- Style preferences? You might however prefer to make style a configuration (of lintr, styler) or to fix code style yourself especially if you don't use a popular code style like the tidyverse coding style.
- Infrastructure like roxygen2?
- Workflow preferences? Issue before a PR?
- A scope statement, like in the skimr package?
- Sandbox account creation? Mocking in tests? Linking to external docs?

rOpenSci further encourages contributing guides to include a lifecycle statement clarifying visions and expectations for the future development of your package, like in this example. Statistical packages are required to have a lifecycle statement, as specified in *General Statistical Stanards* G1.2. That links provides a template for a simple lifecycle statement. CONTRIBUTING.md files can also describe how you acknowledge contributions (see this section).

We encourage you to direct feedback that is not a bug report or a feature request to rOpenSci forum, after making sure you'd see such questions on the forum. Users can use the forum to ask questions about use and report their use cases, and you can subscribe to individual categories and tags to receive notifications about your package. Feel free to mention this in the docs of your package and/or the contributing guidelines/issue template. Please direct your users to tag posts with the package name.

Once a pull request is closer to being merged, you could use a GitHub Actions PR workflow to style the code with styler.

10.1.3 Issue management

By using GitHub features around issues you can help potential contributors find them, and make your roadmap public.

10.1.3.1 Issue labelling

You can use labels such as “help wanted” and “good first issue” to help potential collaborators, including newbies, find your repo. Cf GitHub article. You can also use the “Beginner” label. See examples of beginner issues over all ropensci repos.

10.1.3.2 Pinning issues

You can pin up to 3 issues by repository that will then appear at the top of your issue tracker as nice issue cards. It can help advertise what your priorities are.

10.1.3.3 Milestones

You can create milestones and assign issues to them, which help see what you plan for the next version of your package for instance.

10.2 Working with collaborators

First thing first: keep in touch with your GitHub repository!

- do not forget to **watch your GitHub repository** to be notified of issues or pull requests (alternatively, if you work in bursts, maybe add the information to the contributing guide).
- do not forget to push updates you have locally.
- disable failing tests if you cannot fix them as they create noise in PRs that can puzzle new contributors.

10.2.1 Onboarding collaborators

There’s no general rOpenSci rule as to how you should onboard collaborators. You should increase their rights to the repo as you gain trust, and you should definitely acknowledge contributions (see this section).

You can ask a new collaborator to make PRs (see following section for assessing a PR locally, i.e. beyond CI checks) to dev/master and assess them before merging, and after a while let them push to master, although you might want to keep a system of PR reviews… even for yourself once you have team mates!

A possible model for onboarding collaborators is provided by Jim Hester in his `lintr` repo.

If your problem is *recruiting* collaborators, you can post an open call like Jim Hester's on Twitter, GitHub, and as an rOpenSci package author, you can ask for help in rOpenSci slack and ask rOpenSci team for ideas for recruiting new collaborators.

10.2.2 Working with collaborators (including yourself)

Branches are cheap. Use them extensively when developing features, testing out new ideas, fixing problems.

One of the branches is the default / main branch, where, if you follow trunk-based development, you "merge small, frequent updates". See also GitHub flow and GitLab flow docs. You might want to pair the frequent incrementing of version numbers (in `DESCRIPTION`). One particular aspect of working with collaborators is reviewing pull requests, with some useful guidance in:

- The Art of Giving and Receiving Code Reviews (Gracefully), by Alex Hill
- GitHub documentation about PR reviews

For making and reviewing pull requests we recommend exploring usethis functions.

10.2.3 Be generous with attributions

If someone contributes to your repository consider adding them in DESCRIPTION, as contributor ("ctb") for small contributions, author ("aut") for bigger contributions. Traditionally when citing a package in a scientific publication only "aut" authors are listed, not "ctb" contributors; and on `pkgdown` websites only "aut" names are listed on the homepage, all authors being listed on the authors/ page.

At a minimum consider adding the name of contributors near the feature/bug fix line in NEWS.md.

We recommend your being generous with such acknowledgements, because it is a nice thing to do and because it will make folks more likely to contribute again to your package or other repos of the organization.

As a reminder from our packaging guidelines if your package was reviewed and you feel that your reviewers have made a substantial contribution to the development of your package, you may list them in the `Authors@R` field with a Reviewer contributor type (`"rev"`), like so:

```
person("Bea", "Hernández", role = "rev",
comment = "Bea reviewed the package (v. X.X.XX) for rOpenSci, see <https://github.com/ropensc
```

Only include reviewers after asking for their consent. Read more in this blog post "Thanking Your Reviewers: Gratitude through Semantic Metadata". Note that 'rev' will raise a CRAN NOTE unless the package is built using R v3.5. Make sure you use `roxygen2`'s latest CRAN version.

Please do not list editors as contributors. Your participation in and contribution to rOpenSci is thanks enough!

10.2.4 Welcoming collaborators to rOpenSci

If you give someone write permissions to the repository,

- please contact a staff member so that this new contributor might get **invited to rOpenSci's "ropensci" GitHub organization** (instead of being an outside collaborator)
- please contact rOpenSci's community manager or another staff member so that this new contributor might get get **invited to rOpenSci Slack workspace**.

10.3 Further resources

- rOpenSci community call Set Up Your Package to Foster a Community.
- For re-using kind and usual answers, consider GitHub's saved replies.

Chapter 11

Changing package maintainers

This chapter presents our guidance for taking over maintenance of a package.

11.1 Do you want to give up maintenance of your package?

We have a call for contributors section in our newsletter that comes out every two weeks. The section is called *Call For Contributors*. In that section we highlight packages looking for new maintainers. If you are looking to leave the role of maintainer of your package, get in touch with us and we can highlight your package in our newsletter. We've been very successful thus far, finding new maintainers for six out of six packages.

11.2 Do you want to take over maintenance of a package?

We have a call for contributors section in our newsletter that comes out every two weeks. The section is called *Call For Contributors*. In that section we highlight packages looking for new maintainers. If you are not subscribed to the newsletter already, it's a good idea to subscribe to get notified when there's a package looking for a new maintainer.

11.3 Taking over maintenance of a package

- Add yourself as the new maintainer in the DESCRIPTION file, with `role = c("aut", "cre")`, and make the former maintainer `aut` only.
- Make sure to change maintainer to your name anywhere else in the package, while retaining the former maintainer as an author (e.g, package level manual file, CONTRIBUTING.md, CITATION, etc.)
- The Collaboration Guide has guidance about adding new maintainers and collaborators
- Packages that have been archived or orphaned on CRAN don't need permission of the previous maintainer to be taken over on CRAN. In these cases do get in touch with us so we can offer any help as needed.
- If the package has not been archived by CRAN and there is a maintainer change, have the old maintainer email CRAN and put in writing who the new maintainer is. Make sure to mention that email about the maintainer change when you submit the first new version to CRAN. If the old maintainer is unreachable or will not send this email get in touch with rOpenSci staff.
- If the previous maintainer is reachable, scheduling a meeting will help you get the "lay of the land"

11.3.1 FAQ for new maintainers

- There are a few unresolved issues from the package that I don't know how to fix. Whom may I ask for help?

  It depends; here's what to do in different scenarios:

  - if the old maintainer can be contacted: reach out to them, and ask for help;
  - rOpenSci slack: good for getting help on specific or general problems, see the #package-maintenance channel;
  - rOpenSci discussion forum: this forum is a good option, feel free to ask any questions there;
  - rOpenSci staff: feel free to get in touch with one of us via email/pinging us on GitHub issues, we'll be happy to help;
  - of course there's general R help too if that suits your needs: RStudio community forum, StackOverflow, Twitter #rstats, etc.

- How much can/should you change in the package?

  For general help on changing code in a package, see the Package evolution section.

  When thinking though this, there are many considerations.

  How much time do you have to spend on the package? If you have very limited time, it'd be best to focus on the most critical tasks, whatever those are for the

package in question. If you have ample amount of time, your goals can be larger in scope.

How mature is the package? If the package is mature, many people likely have code that depends on the package's API (i.e., the exported functions, and their parameters). In addition, if there are packages that depend on your package on CRAN, then you need to check that whatever changes you make don't break those packages. The more mature the package is, the more careful you need to be about making changes, especially with the names of exported functions, their parameters, and the exact structure of what exported functions return. Changes can be more easily made that only affect internals of the package.

11.4 Tasks for rOpenSci staff

As an organization, rOpenSci is interested in making sure packages in our suite are available as long as they are useful to the community. As maintainers need to move on, we will in most cases try to get a new maintainer for each package. To these ends, the following tasks are the responsibility of rOpenSci staff.

- If a repository hasn't seen any activity (commits, issues, pull requests) in quite a long time it may simply be a mature package with little need for changes/etc., so take this into account.
- Current maintainer has not responded to issues/pull requests in many months, via any of emails, GitHub issues, or Slack messages:
  - Make contact and see what the situation is. They may say they'd like to step down as maintainer, in which case look for a new maintainer
- Current maintainer is completely missing/not responding
  - If this happens we will try to contact the maintainer for up to one month. However, if updating the package is urgent, we may use our admin access to make changes on their behalf.
- Put a call out in the "Call for Contributors" section of the rOpenSci newsletter for a new maintainer - open an issue in the newsletter repo.

Chapter 12

Releasing a package

Your package should have different versions over time: snapshots of a state of the package that you can release to CRAN for instance. These versions should be properly *numbered*, *released* and *described in a NEWS file*. More details below.

Note that you could streamline the process of updating NEWS and versioning your package by using the `fledge` package.

12.1 Versioning

- We strongly recommend that rOpenSci packages use semantic versioning. A detailed explanation is available in the description chapter.

12.2 Releasing

- Using `usethis::use_release_issue()` and `devtools::release()` will help you remember about more checks.
- Git tag each release after every submission to CRAN. more info
- CRAN does not like too frequent updates. That said, if you notice a major problem one week after a CRAN release, explain it in cran-comments.md and try releasing a newer version.

12.3 News file

A NEWS file describing changes associated with each version makes it easier for users to see what's changing in the package and how it might impact their workflow. You must add one for your package, and make it easy to read.

- It is mandatory to use a `NEWS` or `NEWS.md` file in the root of your package. We recommend using `NEWS.md` to make the file more browsable.
- Please use our example NEWS file as a model. You can find a good NEWS file in the wild in the `taxize` package repo for instance.
- If you use `NEWS`, add it to `.Rbuildignore`, but not if you use `NEWS.md`
- Update the news file before every CRAN release, with a section with the package name, version and date of release, like (as seen in our example NEWS file):

```
foobar 0.2.0 (2016-04-01)
=========================
```

- Under that header, put in sections as needed, including: `NEW FEATURES`, `MINOR IMPROVEMENTS`, `BUG FIXES`, `DEPRECATED AND DEFUNCT`, `DOCUMENTATION FIXES` and any special heading grouping a large number of changes. Under each header, list items as needed (as seen in our example NEWS file). For each item give a description of the new feature, improvement, bug fix, or deprecated function/feature. Link to any related GitHub issue like `(#12)`. The `(#12)` will resolve on GitHub in Releases to a link to that issue in the repo.
- After you have added a `git tag` and pushed up to GitHub, add the news items for that tagged version to the Release notes of a release in your GitHub repo with a title like `pkgname v0.1.0`. See GitHub docs about creating a release.
- New CRAN releases will be tweeted about automatically by roknowtifier and written about in our biweekly newsletter but see next chapter about marketing about how to inform more potential users about the release.
- For more guidance about the NEWS file we suggest reading the tidyverse NEWS style guide.

Chapter 13

Marketing your package

We will help you promoting your package but here are some more things to keep in mind.

- If you hear of an use case of your package, please encourage its author to post the link to our discussion forum in the Use Cases category, for a tweet from rOpenSci and possible inclusion in the rOpenSci biweekly newsletter. We also recommend you to add a link to the use case in a "use cases in the wild" section of your README.
- When you release a new version of your package or release it to CRAN for the first time,
  - Make a pull request to R Weekly with a line about the release under the "New Releases" section (or "New Packages" for the first GitHub/CRAN release).
  - Tweet about it using the "#rstats" hashtag and tag rOpenSci! This might help with contributor/user engagement. Example.
  - Consider submitting a short post about the release to rOpenSci tech notes. Contact rOpenSci Community Manager, (e.g. via Slack or info@ropensci.org). Refer to the guidelines about contributing a blog post).
  - Submit your package to lists of packages such as CRAN Task Views, and rOpenSci non-CRAN Task Views.
- If you choose to market your package by giving a talk about it at a meetup or conference (excellent idea!) read this article of Jenny Bryan's and Mara Averick's.

Chapter 14

GitHub Grooming

rOpenSci packages are currently in their vast majority developed on GitHub. Here are a few tips to leverage the platform in a section about making your repo more discoverable and a section about marketing your own GitHub account after going through peer review.

14.1 Make your repository more discoverable

14.1.1 GitHub repo topics

GitHub repo topics help browsing and searching GitHub repos, and are digested by `codemetar` for rOpenSci registry keywords.

We recommend:

- Adding “r”, “r-package” and “rstats” as topics to your package repo.
- Adding any other relevant topics to your package repo.

We might make suggestions to you after your package is onboarded.

14.1.2 GitHub linguist

GitHub linguist will assign a language for your repo based on the files it contains. Some packages containing a lot of C++ code might get classified as C++ rather than R packages, which is fine and shows the need for the “r”, “r-package” and “rstats” topics.

We recommend overriding GitHub linguist by adding or modifying a .gitattributes to your repo in two cases:

- If you store html files in non standard places (not in docs/, e.g. in vignettes/) use the documentation overrides. Add `*.html linguist-documentation=true` to .gitattributes (Example in the wild)
- If your repo contains code you haven't authored, e.g. JavaScript code, add `inst/js/* linguist-vendored` to .gitattributes (Example in the wild)

This way the language classification and statistics of your repository will more closely reflect the source code it contains, as well as making it more discoverable. Notably, if linguist does not correctly recognize your repo as containing mainly R code, your package won't appear in search results with the `language:R` filter. Similarly, your repo cannot be listed among the trending R repos.

More info about GitHub linguist overrides here.

14.2 Market your own account

- As the author of an onboarded package, you are now a member of rOpenSci's "ropensci" GitHub organization. By default, organization memberships are private; see how to make it public in GitHub docs.
- Even after your package repo has been transferred to rOpenSci, you can pin it under your own account.
- In general we recommend adding at least an avatar (which doesn't need to be your face!) and your name to your GitHub profile.

Chapter 15

Package evolution - changing stuff in your package

This chapter presents our guidance for changing stuff in your package: changing parameter names, changing function names, deprecating functions, and even retiring and archiving packages.

This chapter was initially contributed as a tech note on rOpenSci website by Scott Chamberlain; you can read the original version here.

15.1 Philosophy of changes

Everyone's free to have their own opinion about how freely parameters/functions/etc. are changed in a library - rules about package changes are not enforced by CRAN or otherwise. Generally, as a library gets more mature, changes to user facing methods (i.e., exported functions in an R package) should become very rare. Libraries that are dependencies of many other libraries are likely to be more careful about changes, and should be.

15.2 The lifecycle package

This chapter presents solutions that do not require the lifecycle package but you might still find it useful. We recommend reading the lifecycle documentation.

15.3 Parameters: changing parameter names

Sometimes parameter names must be changed for clarity, or some other reason.

A possible approach is check if deprecated arguments are not missing, and stop providing a meaningful message.

```
foo_bar <- function(x, y) {
    if (!missing(x)) {
        stop("use 'y' instead of 'x'")
    }
    y^2
}

foo_bar(x = 5)
#> Error in foo_bar(x = 5) : use 'y' instead of 'x'
```

If you want to be more helpful, you could emit a warning but automatically take the necessary action:

```
foo_bar <- function(x, y) {
    if (!missing(x)) {
        warning("use 'y' instead of 'x'")
        y <- x
    }
    y^2
}

foo_bar(x = 5)
#> 25
```

Be aware of the parameter `...`. If your function has `...`, and you have already removed a parameter (lets call it `z`), a user may have older code that uses `z`. When they pass in `z`, it's not a parameter in the function definition, and will likely be silently ignored – not what you want. Instead, leave the argument around, throwing an error if it used.

15.4 Functions: changing function names

If you must change a function name, do it gradually, as with any other change in your package.

Let's say you have a function `foo`.

```
foo <- function(x) x + 1
```

However, you want to change the function name to `bar`.

Instead of simply changing the function name and `foo` no longer existing straight away, in the first version of the package where `bar` appears, make an alias like:

```
#' foo - add 1 to an input
#' @export
foo <- function(x) x + 1

#' @export
#' @rdname foo
bar <- foo
```

With the above solution, the user can use either `foo()` or `bar()` – either will do the same thing, as they are the same function.

It's also useful to have a message but then you'll only want to throw that message when they use the old function, e.g.,

```
#' foo - add 1 to an input
#' @export
foo <- function(x) {
    warning("please use bar() instead of foo()", call. = FALSE)
    bar(x)
}

#' @export
#' @rdname foo
bar <- function(x) x + 1
```

After users have used the package version for a while (with both `foo` and `bar`), in the next version you can remove the old function name (`foo`), and only have `bar`.

```
#' bar - add 1 to an input
#' @export
bar <- function(x) x + 1
```

15.5 Functions: deprecate & defunct

To remove a function from a package (let's say your package name is `helloworld`), you can use the following protocol:

- Mark the function as deprecated in package version `x` (e.g., `v0.2.0`)

In the function itself, use `.Deprecated()` to point to the replacement function:

```
foo <- function() {
    .Deprecated("bar")
}
```

There's options in `.Deprecated` for specifying a new function name, as well as a new package name, which makes sense when moving functions into different packages.

The message that's given by `.Deprecated` is a warning, so can be suppressed by users with `suppressWarnings()` if desired.

Make a man page for deprecated functions like:

```
#' Deprecated functions in helloworld
#'
#' These functions still work but will be removed (defunct) in the next version.
#'
#' \itemize{
#'  \item \code{\link{foo}}: This function is deprecated, and will
#'  be removed in the next version of this package.
#' }
#'
#' @name helloworld-deprecated
NULL
```

This creates a man page that users can access like `?`helloworld-deprecated`` and they'll see in the documentation index. Add any functions to this page as needed, and take away as a function moves to defunct (see below).

- In the next version (`v0.3.0`) you can make the function defunct (that is, completely gone from the package, except for a man page with a note about it).

In the function itself, use `.Defunct()` like:

```
foo <- function() {
    .Defunct("bar")
}
```

Note that the message in `.Defunct` is an error so that the function stops whereas `.Deprecated` uses a warning that let the function proceed.

In addition, it's good to add `...` to all defunct functions so that if users pass in any parameters they'll get the same defunct message instead of a `unused argument` message, so like:

```
foo <- function(...) {
    .Defunct("bar")
}
```

Without `...` gives:

```
foo(x = 5)
#> Error in foo(x = 5) : unused argument (x = 5)
```

And with `...` gives:

```
foo(x = 5)
#> Error: 'foo' has been removed from this package
```

Make a man page for defunct functions like:

```
#' Defunct functions in helloworld
#'
#' These functions are gone, no longer available.
#'
#' \itemize{
#'  \item \code{\link{foo}}: This function is defunct.
#' }
#'
#' @name helloworld-defunct
NULL
```

This creates a man page that users can access like `` ?`helloworld-defunct` `` and they'll see in the documentation index. Add any functions to this page as needed. You'll likely want to keep this man page indefinitely.

15.5.1 Testing deprecated functions

You don't have to change the tests of deprecated functions until they are made defunct.

- Consider any changes made to a deprecated function. Along with using `.Deprecated` inside the function, did you change the parameters at all in the deprecated function, or create a new function that replaces the deprecated function, etc. Those changes should be tested if any made.
- Related to above, if the deprecated function is simply getting a name change, perhaps test that the old and new functions return identical results.

- `suppressWarnings()` could be used to suppress the warning thrown from `.Deprecated`, but tests are not user facing, so it is not that bad if the warning is thrown in tests, and the warning could even be used as a reminder to the maintainer.

Once a function is made defunct, its tests are simply removed.

15.6 Archiving packages

Software generally has a finite lifespan, and packages may eventually need to be archived. Archived packages are archived and moved to a dedicated GitHub organization, ropensci-archive. Prior to archiving, the contents of the README file should be moved to an alternative location (such as "README-OLD.md"), and replaced with minimal contents including something like the following:

```
# <package name>

[![Project Status: Unsupported](https://www.repostatus.org/badges/latest/unsupported.s
[![Peer-review badge](https://badges.ropensci.org/<issue_number>_status.svg)](https://

This package has been archived. The former README is now in [README-old](<link-to-READ
```

The repo status badge should be "unsupported" for formerly released packages, or "abandoned" for former concept or WIP packages, in which case the badge code above should be replaced with:

```
[![Project Status: Abandoned](https://www.repostatus.org/badges/latest/abandoned.svg)]
```

An example of a minimal README in an archived package is in ropensci-archive/monkeylearn. Once the README has been copied elsewhere and reduced to minimal form, the following steps should be followed:

- ☐ Close issues with a sentence explaining the situation and linking to this guidance.
- ☐ Archive the repository on GitHub (also under repo settings).
- ☐ Transfer the repository to ropensci-archive, or request an rOpenSci staff member to transfer it (you can email `info@ropensci.org`).

Archived packages may be unarchived if authors or a new person opt to resume maintenance. For that please contact rOpenSci. They are transferred to the ropenscilabs organization.

Chapter 16

Package Curation Policy

This chapter summarizes a proposed curation policy for rOpenSci's ongoing maintenance of packages developed as part of rOpenSci activities and/or under the rOpenSci GitHub organization. This curation policy aims to support these goals:

- Ensure packages provided by rOpenSci are up-to-date and high quality
- Provide clarity as to the development status and and review status of any software in rOpenSci repositories
- Manage maintenance effort for rOpenSci staff, package authors, and volunteer contributors
- Provide a mechanism to gracefully sunset packages while maintaining peer-review badging

Elements of infrastructure described below needed for implementation of the policy are in some cases partly built and in other cases not yet begun. We aim to adopt this policy in part to prioritize work on these components.

16.1 The package registry

- The rOpenSci package registry is a central listing of R packages currently or formerly maintained or reviewed by rOpenSci. It contains essential package metadata including development and review status, and will be the source of data for display on websites, badges, etc. It will allow this listing to be maintained independently of package or infrastructure hosting platforms.

16.2 Staff-maintained packages

Staff-maintained packages are developed and maintained by rOpenSci staff as part of rOpenSci projects. These packages may also be peer-reviewed packages, but are not necessarily peer reviewed. Many are infrastructure packages that fall out of scope for peer review.

- Staff-maintained packages will be listed in the registry with tag "staff_maintained" and listed on rOpenSci's packages web page or similar venues with tag "staff-maintained"
- These packages will be stored in the "ropensci" GitHub organization
- Staff-maintained packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross).
- When the packages fail checks, rOpenSci staff will endeavor to fix changes, prioritizing packages based on user base (downloads), reverse dependencies, or strategic goals.
- On a biannual or annual basis, rOpenSci will review all packages that have been failing for over a month to determine whether to transfer them to the "ropensci-archive" GitHub organization.
- Packages consistently failing and without an ongoing plan to return to active maintenance will move to "archive" status. When archived, staff packages will move to the "ropensci-archive" repository (to be created) and and gain the "archived" type in the registry. They will not be built on rOpenSci system.
- Archived packages will not be displayed by default on the packages web page. A special tab of packages pages will display these with `"type": "archived"` that were either peer-reviewed or staff-maintained.
- Archived packages can be unarchived when the old or a new maintainer is willing to address the problems and wants to revive the package. For that please contact rOpenSci. They are transferred to the ropenscilabs organization.

16.3 Peer-reviewed packages

Peer-reviewed packages are those contributed to the rOpenSci by the community and have passed through peer review. They need to be in-scope at the time of submission to be reviewed.

- Upon acceptance, these peer-reviewed packages are transferred from the author's GitHub to the "ropensci" GitHub organization

- Peer-reviewed packages will be in the registry tagged as “peer-reviewed” and have a peer-reviewed badge in their README.
- Peer-reviewed packages will be listed on rOpenSci’s web page or similar venues with tag “peer-reviewed”
- Peer-reviewed packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross).
- If packages are on CRAN, package authors can choose to subscribe to the notifications of CRAN checks API.
- Annually or bi-annually, rOpenSci staff will review packages in a failing state or have been failing for extended periods, and contact the authors to determine ongoing maintenance status and expected updates. Based on this exchange, rOpenSci may opt to retain the package’s current status with the expectation of an updates, contribute support or seek a new maintainer, or transfer the package to “archived” status.
- Based on user base (measured by downloads), reverse dependencies, or rOpenSci strategic goals, rOpenSci staff may support failing packages via PRs reviewed by package authors, or direct changes (if authors are unresponsive for approximately a month). rOpenSci will also provide support to package authors on request, both by staff and community volunteers, based on time available.
- At the author’s request, or if authors are unresponsive to inquiries for approximately a month, rOpenSci may seek a new maintainer for select peer-reviewed packages it deems have high community value based on user base/downloads, reverse dependencies, or rOpenSci strategic goals.
- When archived, these packages will move from the “ropensci” GitHub organization to the “ropensci-archive” organization (or author GitHub accounts if desired), following transfer guidance. They will gain the “archived” type in the registry. They will retain “peer-reviewed” tags and badges. They will not be built on rOpenSci system.
- Archived packages will not be displayed by default. A special tab of packages pages will `display these with`“type”: “archived”‘ that were either peer-reviewed or staff-maintained.

16.4 Legacy acquired packages

“Legacy” packages are packages not created or maintained by rOpenSci staff and not peer reviewed, but are under the rOpenSci GitHub organization(s) due to historical reasons. (Prior to establishing the peer review process and its scope, rOpenSci absorbed packages from various developers without well-defined criteria.)

- rOpenSci will transfer legacy packages back to author organizations and repositories. If authors are uninterested, we will transfer them to the "ropensci-archive" repository following transfer guidance. If packages are in-scope, rOpenSci will inquire if authors would like to submit them to the Software Review process.
- Legacy packages will not be listed in the package registry.
- Exceptions may be made for packages that are vital parts of the R and/or rOpenSci package ecosystem which are actively monitored by staff.

16.5 Incubator packages

"Incubator" packages are in-development packages created by staff or community members as part of community projects, such as those created at unconferences

- Incubator packages will live in the "ropenscilabs" organization.
- Incubator packages will appear in the package registry with the "incubator" tag
- Incubator packages will not appear on the website by default, but packages pages will include an "experimental packages" tab.
- Incubator packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross). The docs will indicate clearly the package is experimental.
- Biannually or annually, rOpenSci will contact incubator maintainers about repositories at least three months old, inquiring about development status and author preferences for migration to peer-review, ropensci-archive, or to author organizations. Based on response, package will be migrated immediately, peer review will be initiated, or migration will be deferred to the next review. Incubator packages will be migrated to ropensci-archive by default after one year, following transfer guidance.
- Archived incubator packages will gain the "archived" type.

16.5.1 Incubator non-R-packages

- The "incubator" organization will also include non-R-package projects.
- These projects will not be listed in the registry or appear on a web page, and will not be automatically built.

- Migration policy will be the same as for R packages, with appropriate migration locations (e.g., ropensci-books)
- If archived, non-R-packages will move to "ropensci-archive" following transfer guidance.

16.6 Books

rOpenSci books are long-form documentation, often bookdown-formatted, related to rOpenSci packages, projects, or themes, created by both rOpenSci staff and community members.

- Books will live in the "ropensci-books" organization
- Books will be hosted at books.ropensci.org
- Books may be mature or in-development, but must have minimal outlines/content before migrating into "ropensci-books" (e.g. from "ropensci-labs").
- The authorship and development status of a book should be clearly described on its home page and README.
- rOpenSci may provide badges or templates (e.g., "In development," "Community Maintained,") for authors to use on book home pages in the future

Chapter 17

Contributing Guide

This chapter describes our Contributing Guide that outlines how you can make code and non-code contributions to the rOpenSci project.

So you want to contribute to rOpenSci? Fantastic! We developed the rOpenSci Community Contributing Guide to welcome you to rOpenSci and help you recognize yourself as a potential contributor. It will help you figure out what you might gain by giving your time, expertise, and experience, match your needs with things that will help rOpenSci's mission, and connect you with resources to help you along the way.

Our staff and community actively foster a welcoming environment where users and developers from different backgrounds and skill levels learn, share ideas and innovate together openly through shared norms and shared software. Participation in all rOpenSci activities is supported by our Code of Conduct.

We welcome code and non-code contributions from new and seasoned coders at any career stage, and in any sector. You don't have to be a developer! Maybe you want to **spend 30 minutes** sharing your package use case in our public forum or reporting a bug, **one hour** learning by attending a Community Call, **five hours** reviewing an R package submitted for open peer review, **or maybe you want to make an ongoing commitment** to help maintain a package.

What are some benefits of contributing?

- Connect with a community who shares your interest in making science more open
- Learn from people outside your domain who use R with challenges similar to yours
- Ask and answer new research questions by getting to know new software tools and allies

- Feel confident and supported in your efforts to write code and develop software
- Gain visibility for your open source work
- Improve the software you use or build
- Level up your R skills and help others level up theirs
- Level up your writing skills
- Get more exposure for your package

Consult our Contributing Guide and browse "What brings you here?" to find which *I want to …* statements fit you best and choose your path! To help you recognize yourself, we've grouped these into: Discover; Connect; Learn; Build; Help. For each category, we list examples of what those contributions might look like and we link to our resources for the details you need.

Appendix A

NEWS

A.1 0.8.0

- 2022-06-03, Remove former references to now-archived “rodev” package
- 2022-05-30, Advise that reviewers can also directly call `@ropensci-review-bot check package`
- 2022-05-27, Add Mark Padgham to list of authors
- 2022-05-27, Add `devguider::prerelease_checklist` item to pre-release template (#463)
- 2022-05-13, Align version number in DESCRIPTION file with actual version (#443)
- 2022-05-13, Update guidelines for CONTRIBUTING.md (#366, #462)
- 2022-05-09, Add section on authorship of included code, thanks to `@KlausVigo` (#388).
- 2022-05-09, Remove mention of ‘rev’ role requiring R v3.5
- 2022-05-05, Move all scripts from local `inst` directory to `ropensci-org/devguider` pkg.
- 2022-05-03, Update package archiving guidance to reduce README to minimal form.
- 2022-04-29, Advise that authors can directly call `@ropensci-review-bot check package`.
- 2022-04-29, Describe `pkgcheck-action` in CI section.

- 2022-04-29, Update scope in policies section to include statistical software.
- 2022-04-29, Add `prelease.R` script to open pre-release GitHub issue & ref in appendix.
- 2022-04-26, Add GitHub 2FA recommendation to package security.
- 2022-03-29, Remove references to Stef Butland, former community manager.
- 2022-03-28, Add comments on submission planning about time commitment.
- 2022-03-24, Remove approval comment template (coz it's automatically generated by the bot now).
- 2022-03-21, rephrase CITATION guidance to make it less strict. Also mentions CITATION.cff and the cffr package.
- 2022-03-08, add links to blogs related to package development (#389).
- 2022-02-17, update redirect instructions (`@peterdesmet`, #387).
- 2022-02-14, link to Michael Lynch's post Why Good Developers Write Bad Unit Tests.
- 2022-02-14, mention more packages for testing like dittodb, vcr, httptest, httptest2, webfakes.
- 2022-01-10, make review templates R Markdown files (`@Bisaloo`, #340).
- 2022-01-14, update guidance on CI services (#377)
- 2022-01-11, update guidance around branches, with resources suggested by `@ha0ye` and `@statnmap`.
- 2022-01-10, divide author's guide into sub-sections, and add extra info including `pkgcheck`.
- 2021-11-30, adds links to examples of reviews, especially tough but constructive ones (with help from `@noamross`, `@mpadge`, #363).
- 2021-11-19, add recommended spatial packages to scaffolding section (software-review-meta#47)
- 2021-11-18, update advice on grouping functions for pkgdown output (#361)

A.2 0.7.0

- 2021-11-04, add mentions of stat software review to software review intro and to the first book page (#342).
- 2021-11-04, mention pkgcheck in the author guide (**?**, #343).
- 2021-11-04, add editors' responsibilities including Editor etiquette for commenting on packages on which you aren't handling/reviewing (**?**, #354).
- 2021-11-04, give precise examples of tools for installation instructions (remotes, pak, R-universe).
- 2021-11-04, add more bot guidance (less work for editors).
- 2021-10-07, add guidance for editorial management (recruiting, inviting, onboarding, offboarding editors).
- 2021-09-14, add a requirement that there is at least one *HTML* vignette.
- 2021-09-03, add some recommendations around git. (**?**, #341)
- 2021-07-14, clarify the categories data extraction and munging by adding examples. (**?**, #337)
- 2021-05-20, add guidance around setting up your package to foster a community, inspired by the recent rOpenSci community call. (with help from `@Bisaloo`, #289, #308)
- 2021-04-27, no longer ask reviewers to ask covr as it'll be done by automatic tools, but ask them to pay attention to tests skipped.
- 2021-04-02, add citation guidance.
- 2021-04-02, stop asking reviewers to run goodpractice as this is part of editorial checks.
- 2021-03-23, launched a new form for reviewer volunteering.
- 2021-02-24, add guidance around the use of `@ropensci-review-bot`.

A.3 0.6.0

- 2021-02-04, add guidance to enforce package versioning and tracking of changes through review (`@annakrystalli`, #305)
- 2021-01-25, add a translation of the review template in Spanish (`@Fvd`, `@maurolepore`, #303)

- 2021-01-25, the book has now better citation guidance in case you want to cite this very guide (@Bisaloo, #304).
- 2021-01-12, add some more guidance on escaping examples (#290).
- 2021-01-12, mention the lifecycle package in the chapter about package evolution (#287).
- 2021-01-12, require overlap information is put in documentation (#292).
- 2021-01-12, start using the bookdown::bs4_book() template.
- 2021-01-12, add a sentence about whether it is acceptable to push a new version of a package to CRAN within two weeks of the most recent version if you have just been made aware of, and fixed, a major bug (@sckott, #283)
- 2021-01-12, mention the HTTP testing in R book.
- 2021-01-12, mention testthat snapshot tests.
- 2021-01-12, remove mentions of Travis CI and link to Jeroen Ooms' blog post about moving away from Travis.
- 2021-01-12, update the package curation policy: mention a possible exception for legacy packages that are vital parts of the R and/or rOpenSci package ecosystem which are actively monitored by staff. (@noamross, #293)

A.4 0.5.0

- 2020-10-08, add help about link checking (@sckott, #281)
- 2020-10-08, update JOSS instructions (@karthik, #276)
- 2020-10-05, add links to licence resources (@annakrystalli, #279)
- 2020-10-05, update information about the contributing guide (@stefaniebutland, #280)
- 2020-09-11, make reviewer approval a separate template (@bisaloo, #264)
- 2020-09-22, add package curation policy (@noamross, #263)
- 2020-09-11, add more guidance and requirements for docs at submission (@annakrystalli, #261)
- 2020-09-14, add more guidance on describing data source in DESCRIPTION (@mpadge, #260)
- 2020-09-14, add more guidance about tests of deprecated functions (@sckott, #213)

- 2020-09-11, update the CI guidance (@bisaloo, @mcguinlu, #269)
- 2020-09-11, improve the redirect guidance (@jeroen, @mcguinlu, #269)

A.5 0.4.0

- 2020-04-02, give less confusing code of conduct guidance: the reviewed packages' COC is rOpenSci COC (@Bisaloo, @cboettig, #240)
- 2020-03-27, add section on Ethics, Data Privacy and Human Subjects Research to Policies chapter
- 2020-03-12, mention GitHub Actions as a CI provider.
- 2020-02-24, add guide for inviting a guest editor.
- 2020-02-14, add mentions of the ropensci-books GitHub organisation and associated subdomain.
- 2020-02-10, add field and laboratory reproducibility tools as a category in scope.
- 2020-02-10, add more guidance about secrets and package development in the security chapter.
- 2020-02-06, add guidance about Bioconductor dependencies (#246).
- 2020-02-06, add package logo guidance (#217).
- 2020-02-06, add one CRAN gotcha: single quoting software names(#245, @aaronwolen)
- 2020-02-06, improve guidance regarding the replacement of "older" pkgdown website links and source (#241, @cboettig)
- 2020-02-06, rephrase the EiC role (#244).
- 2020-02-06, remove the recommendation to add rOpenSci footer (https://github.com/ropensci/software-review-meta/issues/79).
- 2020-02-06, remove the recommendation to add a review mention to DESCRIPTION but recommends mentioning the package version when reviewers are added as "rev" authors.
- 2020-01-30, slightly changes the advice on documentation re-use: add a con; mention @includeRmd and @example; correct the location of Rmd fragments (#230).
- 2020-01-30, add more guidance for the editor in charge of a dev guide release (#196, #205).

- 2020-01-22, add guidance in the editor guide about not transferred repositories.
- 2020-01-22, clarify forum guidance (for use cases and in general).
- 2020-01-22, mention an approach for pre-computing vignettes so that the pkgdown website might get build on rOpenSci docs server.
- 2020-01-22, document the use of mathjax with rotemplate (`@Bisaloo`, #199).
- 2020-01-20, add guidance for off-thread interaction and COIs (`@noamross`, #197).
- 2020-01-20, add advice on specifying dependency minimum versions (`@karthik`, `@annakrystalli`, #185).
- 2020-01-09, start using GitHub actions instead of Travis for deployment.
- -2019-12-11, add note in Documentation sub-section of Packaging Guide section about referencing the new R6 support in roxygen2 (ropensci/dev_guide#189)
- 2019-12-11, add new CRAN gotcha about having 'in R' or 'with R' in your package title (`@bisaloo`, ropensci/dev_guide#221)

A.6 0.3.0

- 2019-10-03, include in the approval template that maintainers should include link to the docs.ropensci.org/pkg site (ropensci/dev_guide#191)
- 2019-09-26, add instructions for handling editors to nominate packages for blog posts (ropensci/dev_guide#180)
- 2019-09-26, add chapter on changing package maintainers (ropensci/dev_guide#128) (ropensci/dev_guide#194)
- 2019-09-26, update Slack room to use for editors (ropensci/dev_guide#193)
- 2019-09-11, update instructions in README for rendering the book locally (ropensci/dev_guide#192)
- 2019-08-05, update JOSS submission instructions (ropensci/dev_guide#187)
- 2019-07-22, break "reproducibility" category in policies into component parts. (ropensci/software-review-meta#81)
- 2019-06-18, add link to rOpenSci community call "Security for R" to security chapter.

- 2019-06-17, fix formatting of Appendices B-D in the pdf version of the book (bug report by `@IndrajeetPatil`, #179)
- 2019-06-17, add suggestion to use R Markdown hunks approach when the README and the vignette share content. (ropensci/dev_guide#161)
- 2019-06-17, add mention of central building of documentation websites.
- 2019-06-13, add explanations of CRAN checks. (ropensci/dev_guide#177)
- 2019-06-13, add mentions of the `rodev` helper functions where relevant.
- 2019-06-13, add recommendation about using `cat` for `str.*()` methods. RStudio assumes that `str` uses `cat`, if not when loading an R object the `str` prints to the console in RStudio and doesn't show the correct object structure in the properties. ([`@mattfidler`] (https://github.com/mattfidler/) #178)
- 2019-06-12, add more details about git flow.
- 2019-06-12, remove recommendation about `roxygen2` dev version since the latest stable version has what is needed. (`@bisaloo`, #165)
- 2019-06-11, add mention of usethis functions for adding testing or vignette infrastructure in the part about dependencies in the package building guide.
- 2019-06-10, use the new URL for the dev guide, https://devguide.ropensci.org/
- 2019-05-27, add more info about the importance of the repo being recognized as a R package by linguist (`@bisaloo`, #172)
- 2019-05-22, update all links eligible to HTTPS and update links to the latest versions of Hadley Wickham and Jenny Bryan's books (`@bisaloo`, #167)
- 2019-05-15, add book release guidance for editors. (ropensci/dev_guide#152)

A.7 0.2.0

- 2019-05-23, add CRAN gotcha: in the Description field of your DESCRIPTION file, enclose URLs in angle brackets.
- 2019-05-13, add more content to the chapter about contributing.
- 2019-05-13, add more precise instructions about blog posts to approval template for editors.
- 2019-05-13, add policies allowing using either <– or = within a package as long as the whole package is consistent.
- 2019-05-13, add request for people to tell us if they use our standards/checklists when reviewing software elsewhere.

- 2019-04-29, add requirement and advice on testing packages using `devel` and `oldrel` R versions on Travis.
- 2019-04-23, add a sentence about why being generous with attributions and more info about ctb vs aut.
- 2019-04-23, add link to Daniel Nüst's notes about migration from XML to xml2.
- 2019-04-22, add use of rOpenSci forum to maintenance section.
- 2019-04-22, ask reviewer for consent to be added to DESCRIPTION in review template.
- 2019-04-22, use a darker blue for links (feedback by `@kwstat`, #138).
- 2019-04-22, add book cover.
- 2019-04-08, improve formatting and link text in README (`@katrinleinweber`, #137)
- 2019-03-25, add favicon (`@wlandau`, #136).
- 2019-03-21, improve Travis CI guidance, including link to examples. (`@mpadge`, #135)
- 2019-02-07, simplify code examples in Package Evolution section (maintenance_evolution.Rmd file) (`@hadley`, #129).
- 2019-02-07, added a PDF file to export (request by `@IndrajeetPatil`, #131).

A.8 0.1.5

- 2019-02-01, created a .zenodo.json to explicitly set editors as authors.

A.9 First release 0.1.0

- 2019-01-23, add details about requirements for packages running on all major platforms and added new section to package categories.
- 2019-01-22, add details to the guide for authors about the development stage at which to submit a package.
- 2018-12-21, inclusion of an explicit policy for conflict of interest (for reviewers and editors).
- 2018-12-18, added more guidance for editor on how to look for reviewers.
- 2018-12-04, onboarding was renamed Software Peer Review.

A.10 place-holder 0.0.1

- Added a `NEWS.md` file to track changes to the book.

Appendix B

Review template

You can save this as an R Markdown file, or delete the YAML and save it as a Markdown file.

B.1 Package Review

Please check off boxes as applicable, and elaborate in comments below. Your review is not limited to these topics, as described in the reviewer guide

- **Briefly describe any working relationship you have (had) with the package authors.**

☐ As the reviewer I confirm that there are no conflicts of interest for me to review this work (if you are unsure whether you are in conflict, please speak to your editor *before* starting your review).

B.1.0.1 Documentation

The package includes all the following forms of documentation:

☐ **A statement of need:** clearly stating problems the software is designed to solve and its target audience in README
☐ **Installation instructions:** for the development version of package and any non-standard dependencies in README
☐ **Vignette(s):** demonstrating major functionality that runs successfully locally
☐ **Function Documentation:** for all exported functions
☐ **Examples:** (that run successfully locally) for all exported functions

- ☐ **Community guidelines:** including contribution guidelines in the README or CONTRIBUTING, and DESCRIPTION with `URL`, `BugReports` and `Maintainer` (which may be autogenerated via `Authors@R`).

B.1.0.2 Functionality

- ☐ **Installation:** Installation succeeds as documented.
- ☐ **Functionality:** Any functional claims of the software been confirmed.
- ☐ **Performance:** Any performance claims of the software been confirmed.
- ☐ **Automated tests:** Unit tests cover essential functions of the package and a reasonable range of inputs and conditions. All tests pass on the local machine.
- ☐ **Packaging guidelines**: The package conforms to the rOpenSci packaging guidelines.

Estimated hours spent reviewing:

- ☐ Should the author(s) deem it appropriate, I agree to be acknowledged as a package reviewer ("rev" role) in the package DESCRIPTION file.

B.1.1 Review Comments

Appendix C

Review template in Spanish

You can save this as an R Markdown file, or delete the YAML and save it as a Markdown file.

C.1 Revisión de un paquete

Por favor trata de marcar tantas casillas como te sea posible y elabora tus argumentos en comentarios abajo de cada una. Tu revisión no esta limitada a estos temas, tal como se describe en la guia para revisores (Reviewer Guide)

Por favor describe cualquier relación de trabajo que tengas/hayas tenido con los autores del paquete)

- ☐ Como revisor confirmo que no tengo conflictos de interés para poder hacer la revisión de este trabajo (si no estas segura si tienes un conflicto por favor entra en contacto con tu editor *antes* de arrancar con la revisión.

C.1.0.1 Documentación

El paquete incluye todos los siguiente tipos de documentación:

- ☐ **Una declaración de necesidades** que claramente describe las necesidades que el software esta diseñado a resolver y el public meta que busca atender en el archivo README
- ☐ **Instrucciones de instalación** de la versión en desarrollo del paquete incluyendo cualquier dependencia no-estándar en el archivo README

- ☐ **Viñeta(s)** demostrando la funcionalidad principal que ademas corren localmente
- ☐ **Documentación de las funciones** exportadas
- ☐ **Ejemplos** (que corren localmente) para todas las funciones exportadas
- ☐ **Directrices comunitarias** incluyendo una guia de contribución en el archivo README o el archivo CONTRIBUTING y un archivo DESCRIPTION que incluye `URL`, `BugReports` and `Maintainer` (todas en inglés por concenvión y para que puedan ser autogeneradas con `Authors@R`).

C.1.0.2 Funcionalidad

- ☐ **Instalación:** La instalación se completa con éxito tal como fue documentada.
- ☐ **Funcionalidad:** Toda afirmación de funcionalidad del software se confirma como existente.
- ☐ **Desempeño:** Toda afirmación de desempeño del software se confirma como alcanzada.
- ☐ **Pruebas automáticas:** Hay pruebas unitarias que cubren las funciones esenciales dentro del paquete con un rango razonable de entradas y condiciones. Todas las pruebas corren en la maquina local.
- ☐ **Directrices de empaque**: El paquete cumple con las directrices de empaque de rOpenSci.

Estimación de horas dedicadas a la revisión:

- ☐ Si la o las persona(s) autora(s) lo considera(n) apropiado, yo estoy de acuerdo con que me reconozcan como revisor del paquete (el rol “rev’) en la el archivo DESCRIPTION del paquete.

C.1.1 Comentarios de la revisión

Appendix D

Editor's template

D.0.1 Editor checks:

- ☐ **Documentation**: The package has sufficient documentation available online (README, pkgdown docs) to allow for an assessment of functionality and scope without installing the package. In particular,
    - ☐ Is the case for the package well made?
    - ☐ Is the reference index page clear (grouped by topic if necessary)?
    - ☐ Are vignettes readable, sufficiently detailed and not just perfunctory?
- ☐ **Fit**: The package meets criteria for fit and overlap.
- ☐ **Installation instructions:** Are installation instructions clear enough for human users?
- ☐ **Tests**: If the package has some interactivity / HTTP / plot production etc. are the tests using state-of-the-art tooling?
- ☐ **Contributing information**: Is the documentation for contribution clear enough e.g. tokens for tests, playgrounds?
- ☐ **License:** The package has a CRAN or OSI accepted license.
-

D.1 [ ] Project management: Are the issue and PR trackers in a good shape, e.g. are there outstanding bugs, is it clear when feature requests are meant to be tackled?

D.1.0.1 Editor comments

Appendix E

Review request template

Editors may make use of the e-mail template below in recruiting reviewers.

Dear [REVIEWER]

Hi, this is [EDITOR]. [FRIENDLY BANTER]. I'm writing to ask if you would be willing to review a package for rOpenSci. As you probably know, rOpenSci conducts peer review of R packages contributed to our collection in a manner similar to journals.

The package, [PACKAGE] by [AUTHOR(S)], does [FUNCTION]. You can find it on GitHub here: [REPO LINK]. We conduct our open review process via GitHub as well, here: [ONBOARDING ISSUE]

If you accept, note that we ask reviewers to complete reviews in three weeks. (We've found it takes a similar amount of time to review a package as an academic paper.)

Our reviewers guide details what we look for in a package review, and includes links to example reviews. Our standards are detailed in our packaging guide, and we provide a reviewer template for you to use. Please make sure you do not have a conflict of interest preventing you from reviewing this package. If you have questions or feedback, feel free to ask me or post to the rOpenSci forum.

Are you able to review? If you can not, suggestions for alternate reviewers are always helpful. If I don't hear from you within a week, I will assume you are unable to review at this time.

Thank you for your time.

Sincerely,

[EDITOR]

Appendix F

Reviewer approval comment template

F.1 Reviewer Response

F.1.0.1 Final approval (post-review)

- ☐ **The author has responded to my review and made changes to my satisfaction. I recommend approving this package.**

Estimated hours spent reviewing:

Appendix G

NEWS template

```
foobar 0.2.0 (2016-04-01)
=========================

### NEW FEATURES

  * New function added `do_things()` to do things (#5)

### MINOR IMPROVEMENTS

  * Improved documentation for `things()` (#4)

### BUG FIXES

  * Fix parsing bug in `stuff()` (#3)

### DEPRECATED AND DEFUNCT

  * `hello_world()` now deprecated and will be removed in a
    future version, use `hello_mars()`

### DOCUMENTATION FIXES

  * Clarified the role of `hello_mars()` vs. `goodbye_mars()`


### (a special: any heading grouping a large number of changes under one thing)

    * blablabla.
```

```
foobar 0.1.0 (2016-01-01)
=========================

### NEW FEATURES

  * released to CRAN
```

Appendix H

Book release guidance

Editors preparing for a release can run the `prelease.R` script in the `inst` directory of this repository to automatically open a GitHub issue with checkpoints for all current issues assigned to the upcoming release milestone, along with the following checklist. Before running the script, please manually check the assignment of issues to the milestone. This should be run one month prior to planned release.

H.1 Release book version

H.1.1 Repo maintenance between releases

- ☐ Look at the issue tracker for the dev guide and for software review meta for changes still to be made in the dev guide. Assign dev guide issues to milestones corresponding to versions, either the next one or the one after that, e.g. version 0.3.0. Encourage PRs, have them reviewed.

H.1.2 1 month prior to release

- ☐ Remind editors to open issues/PRs for items they want to see in the next version.
- ☐ Run the `devguide_prerelease()` function from the `devguider` package.
- ☐ Ask editors for any feedback you need from them before release.
- ☐ For each contribution/change make sure the NEWS in Appendix.Rmd were updated.

☐ Plan a date for release in communication with rOpenSci's Community Manager who will give you a date for publishing a blog post / tech note.

H.1.3 2 weeks prior to release

☐ Draft a blog post / tech note about the release with enough advance for editors and then Community Manager to review it (2 weeks). Example, General blog post instructions, specific instructions for release posts.

☐ Make a PR from the dev branch to the master branch, ping editors on GitHub and Slack. Mention the blog post draft in a comment on this PR.

H.1.4 Release

☐ Check URLs using the `devguide_urls()` function from the {devguider} package

☐ Check spelling using the `devguide_spelling()` function from the {devguider} package. Update the WORDLIST as necessary.

☐ Squash and merge the PR from dev to master.

☐ GitHub release, check Zenodo release.

- [ ] Re-build (for Zenodo metadata update in the book) or wait for daily build

☐ Re-create the dev branch

☐ Finish your blog post / tech note PR. Underline the most important aspects to be highlighted in tweets as part of the PR discussion.

Appendix I

How to set a redirect

I.1 Non GitHub pages site (e.g. Netlify)

Replace the content of the current website with a `index.html` and `404.html` files both containing:

```
<html>
<head>
<meta http-equiv="refresh" content="0;URL=https://docs.ropensci.org/<pkgname>/">
</head>
</html>
```

I.2 GitHub pages

You can setup the redirect from your main user gh-pages repository:

- create a new repository (if you don't have one yet): `https://github.com/<username>/<username>.github.io`.
- In this repository create a directory `<pkgname>` containing 2 files: a `index.html` and `404.html` file, which both redirect to the new location (see previous subsection).
- Test that `https://<username>.github.io/<pkgname>/index.html` now redirects.

Bibliography