rOpenSci Packages: Development, Maintenance, and Peer Review

rOpenSci software review editorial team: Brooke Anderson, Scott Chamberlain, Anna Krystalli, Lincoln Mullen, Karthik Ram, Noam Ross, Maëlle Salmon, Melina Vidoni

2019-05-23

Contents

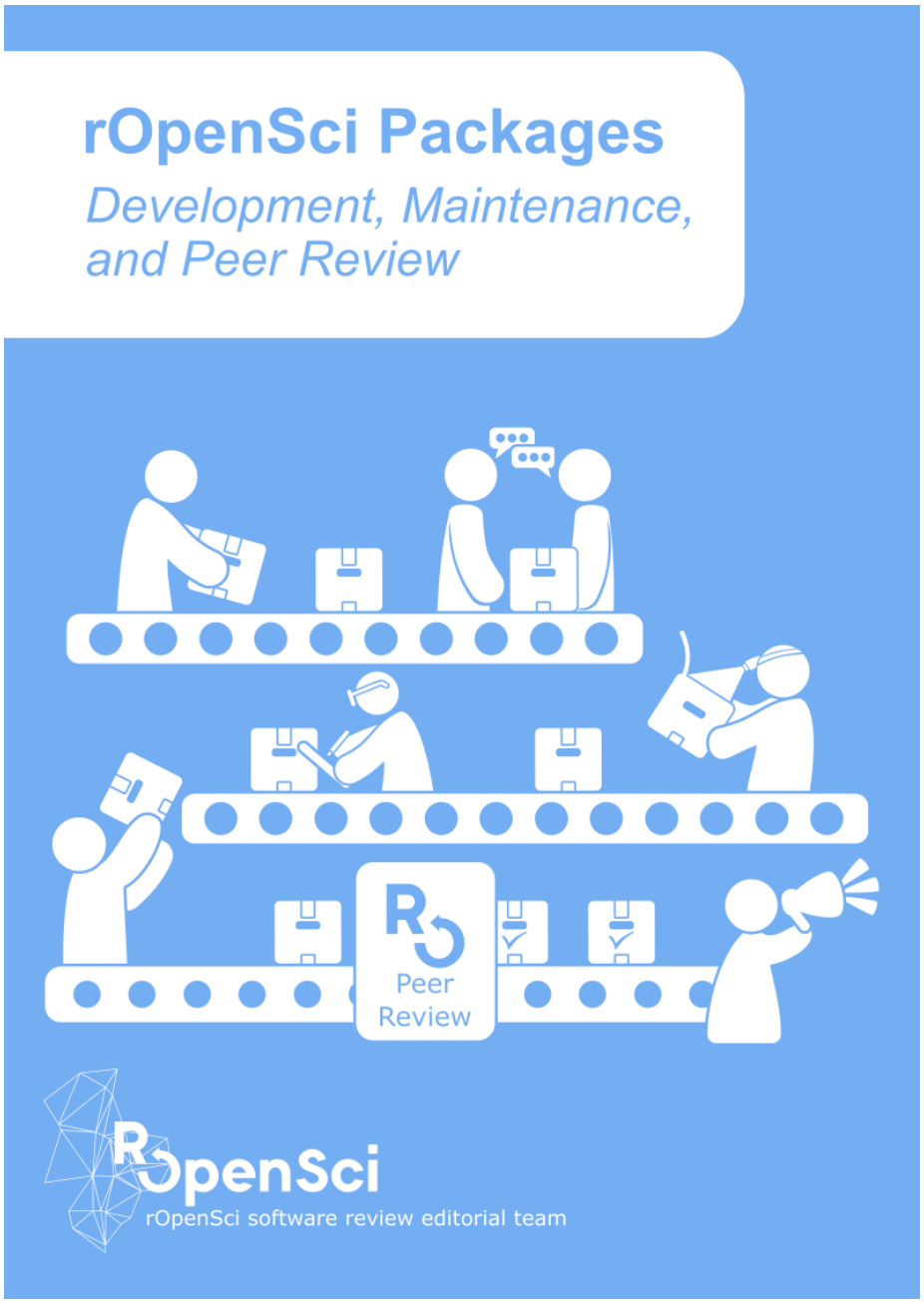

Figure 1: cover image

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.

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.

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. 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's concise but thorough book on package development which is available for free online (and print). 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

1.4 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.

1.4.1 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.*()` method, as these methods of printing messages are harder for the user to suppress.

1.4.2 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/

1.5 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 Hadley's book. We recommend the `styler` package for automating part of the code styling.
- 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 inconsistency beause you can't use <- in this case.

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. 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
  - 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

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.
- 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
- 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`.
- *After* a package is accepted but before transfer, the rOpenSci footer should be added to the bottom of the README file with the following markdown line:

```
[![ropensci_footer](http://ropensci.org/public_images/github_footer.png)](https://ropensci.org)
```

- Add a code of conduct and contribution guidelines, cf this section of the book.
- 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

- All exported package functions should be fully documented with examples.
- 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.
- 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.
- We recommend using the `@family` tag in the documentation of functions to allow their grouping in the documentation of the installed package and potentially in the package's website, see this section of Hadley Wickham's book and this section of the present chapter for more details.
- 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 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 can have the same 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 (i.e. programming/NSE with dplyr).
- The vignette(s) should include citations to software and papers where appropriate.
- 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.

- Add `#' @noRd` to internal functions.
- 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.
- If you prefer not to clutter up code with extensive documentation, place further documentation/examples 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.
    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.

1.8 Documentation website

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

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

1.8.1 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 `roxygen` above version 6.0.1.9000 (as of July 2018, development version to be installed via `remotes::install_github("klutometis/roxygen")`) 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.

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

1.8.2 Automatic deployment of the documentation website

You could use the `tic` package for automatic deployment of the package's website, see this example repo. 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/Travis.

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 and this example in the wild: `pkgdown` config file, resulting website.

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 for ropensci, see <https://github.com/ropensci/
```

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. As of June 2018 you need to use `roxygen2` dev version for the list of authors in the package-level documentation to be compiled properly with the "rev" role (because this is a MARC role not included yet in `royxgen2` CRAN version from February 2017).

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

1.10 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.
- 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.
- Once you've set up CI, use your package's code coverage report (cf this section of our book) to identify untested lines, and to add further tests.
- `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 will still run on Travis.
- 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.

1.11 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.

- In addition to running examples locally on your own computer, we strongly advise that you run examples on one of the CI systems, e.g. Travis-CI. Again, examples that are not wrapped in `\dontrun{}` or `\donttest{}` will be run, but for those that are you can add `r_check_args: "--run-dontrun"` to run examples wrapped in `\dontrun{}` in your `.travis.yml` (and/or `--run-donttest` if you want to run examples wrapped in `\donttest{}`).

1.12 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 any package in the examples or tests of your package, make sure to list it in `Suggests`, if not already listed in `Imports`.

- 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. 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.
  - If dependencies that overlapping functionality, see if you can rely on only one.
  - More dependency-management tips can be found in this post by Scott Chamberlain.

1.13 Recommended scaffolding

- For HTTP requests we recommend using curl, crul, or httr 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.

1.14 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.
- 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.
- 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 Travis.
- Include top-level files such as `paper.md`, `.travis.yml` in your `.Rbuildignore` file.

1.15 Further guidance

- Hadley Wickham's *R Packages* is an excellent, readable resource on package development which is available for free online (and print).
- Writing R Extensions is the canonical, usually most up-to-date, reference for creating R packages.
- 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.

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.

2.2 How to use continuous integration?

The `usethis` package offers a few functions for continuous integration setup, see the documentation.

Details will be provided below for different services.

2.3 Which continuous integration service(s)?

Different continuous integration services will support builds on different operating systems.

R packages should have CI for all platforms 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 platforms, and most often not too much hassle.

2.3.1 Travis CI (Linux and Mac OSX)

Travis offers continuous integration for Linux and Mac OSX. Set it up using `usethis::use_travis()`.

R is now a natively supported language on Travis-CI, making it easier than ever to do continuous integration. See R Packages and Julia Silge's Beginner's Guide to Travis-CI for R for more help.

To customize your build, have a look at Travis' docs and at existing Travis config files over rOpenSci's ropensci GitHub organization, e.g. this one for the `spelling` package.

2.3.1.1 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. You can enable this on Travis by adding the following to your `.travis.yml` configuration file:

```
r:
  - oldrel
  - release
  - devel
```

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. See this more complex example of a Travis configuration where pkgdown documentation is built during the release R version testing only.

2.3.1.2 Minimizing build times on Travis

Please use these tips to minimize build time on Travis especially after your package project gets transferred to ropensci Travis account:

- Cache installation of packages. Add `cache: packages` at the beginning of the config file. Example in the wild. It'll already be in the config file if you set Travis up using `usethis::use_travis()`.
- Enable auto-cancellation of builds.
- If you have a matrix to test your package on several Travis systems, place any `after_success` or `before_deploy` commands within one single matrix build to avoid them being executed on every matrix build.

2.3.2 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`

2.3.3 Circle CI

Circle CI is used, for example, by rOpenSci package `bomrang` as an alternative to Travis CI.

2.4 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.

2.5 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.

2.6 Make more of your CI builds: tic

The `tic` package facilitates deployment tasks for R packages tested by Travis CI, AppVeyor, or any other CI tool of your choice, cf e.g. our suggestion to build and deploy the documentation website of your package via Travis. Actually this book uses `tic` for deployment.

2.7 Make more of your CI builds: Use Travis to deploy websites

If your package has a website built by `pkgdown`, you can use Travis to update and deploy the website on each build. See the help on `pkgdown::deploy_site_github()`. If your `.travis.yml` has a matrix specifying multiple Travis builds, you should put the `before_deploy` and `deploy` commands within a single matrix entry to avoid all of them rebuilding and deploying your website. See an example here.

Chapter 3

Package Development Security Best Practices

This chapter isn't written yet. It should mention the elements outlined in this bookdown (repo) started at the unconf 2017, or have a link to this book/a similar book when written.

Also interesting are these unconf projects:

- the `notary` package
- the security-related projects of unconf18

3.1 Misc

We recommend you secure your GitHub account with 2FA.

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.

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 are also distributed via our drat repository and Docker images, and listed in our task views.
- rOpenSci **packages can be cross-listed** with other repositories such as CRAN and BioConductor.
- rOpenSci packages that contain a short accompanying paper can, after review, be automatically submitted to the Journal of Open-Source Software for fast-tracked publication.

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, click here to fill out a 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 Editors and reviewers

4.5.1 Associate editors

rOpenSci's Software Peer Review process is run by:

- Noam Ross, EcoHealth Alliance
- Scott Chamberlain, rOpenSci
- Karthik Ram, rOpenSci
- Maëlle Salmon, rOpenSci
- Lincoln Mullen, George Mason University
- Anna Krystalli, University of Sheffield RSE
- Melina Vidoni, INGAR CONICET-UTN
- Brooke Anderson, Colorado State University

4.5.2 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 · Alison Appling · Zebulun Arendsee · Taylor Arnold · Dean Attali · Mara Averick · Suzan Baert · James Balamuta · David Bapst · Joëlle Barido-Sottani · Cale Basaraba · John Baumgartner · Marcus Beck · Gabe Becker · Jason Becker · Dom Bennett · Kenneth Benoit · Aaron Berdanier · Carl Boettiger · Ben Bond-Lamberty · Alison Boyer · Jenny Bryan · Lorenzo Busetto · Jorge Cimentada · Jon Clayden · Will Cornwell · Ildiko Czeller · Laura DeCicco · Christophe Dervieux · Amanda Dobbyn · Jasmine Dumas · Remko Duursma · Mark Edmondson · Paul Egeler · Evan Eskew · Manuel Fernandez · Rich FitzJohn · Robert Flight · Zachary Foster · Auriel Fournier · Carl Ganz · Duncan Garmonsway · Sharla Gelfand · Duncan Gillespie · David Gohel · Laura Graham · Charles Gray · Corinna Gries · Julia Gustavsen · W Kyle Hamilton · Ivan Hanigan · Jeff Hanson · Ted Hart · Nujcharee Haswell · Verena Haunschmid · Rafael Pilliard Hellwig · Bea Hernandez · Jim Hester · Peter Hickey · Roel M. Hogervorst · Jeff Hollister · Kelly Hondula · Sean Hughes · Brandon Hurr · Najko Jahn · Tamora D James · Max Joseph · Krunoslav Juraic · Soumya Kalra · Michael Kane · Andee Kaplan · Hazel Kavılı · Os Keyes · Michael Koontz · Bianca Kramer · Anna Krystalli · Will Landau · Erin LeDell · Thomas Leeper · Stephanie Locke · Robin Lovelace · Julia Stewart Lowndes · Tim Lucas · Andrew MacDonald · Jesse Maegan · Tristan Mahr · Ben Marwick · Miles McBain · Lucy D'Agostino McGowan · Amelia McNamara · Elaine McVey · Bryce Mecum · Francois Michonneau · Jessica Minnier · Priscilla Minotti · Paula Moraga · Ross Mounce · Lincoln Mullen · Matt Mulvahill · Dillon Niederhut · Rory Nolan · Jakub Nowosad · Daniel Nüst · Paul Oldham · Samantha Oliver · Jeroen Ooms · Philipp Ottolinger · Mark Padgham · Marina Papadopoulou · Edzer Pebesma · Thomas Lin Pedersen · Etienne Racine · Nistara Randhawa · David Ranzolin · Neal Richardson · Emily Riederer · tyler rinker · Emily Robinson · Xavier Rotllan-Puig · Bob Rudis · Edgar Ruiz · Kent Russel · Francisco Rodriguez Sanchez · Alicia Schep · Marco Sciaini · Heidi Seibold · Julia Silge · Margaret Siple · Peter Slaughter · Mike Smith · Tuija Sonkkila · Gaurav Sood · Adam Sparks · Joseph Stachelek · Irene Steves · Michael Sumner · Sarah Supp · Filipe Teixeira · Andy Teucher · Jennifer Thompson · Joe Thorley · Tiffany Timbers · Tim Trice · Ted Underwood · Kevin Ushey · Josef Uyeda · Frans van Dunné · Mauricio Vargas · Remi Vergnon · Claudia Vitolo · Ben Ward · Elin Waring · Rachel Warnock · Leah Wasser · Stefan Widgren · Luke Winslow · David Winter · Kara Woo · Bruna Wundervald · Lauren Yamane · Taras Zakharko · Hao Zhu · Chava Zibman · Naupaka Zimmerman

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 here.

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 below. If you are unsure whether your package fits into one of these 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, but trained models that act as utilities (e.g., for optical character recognition), may qualify. (Examples: **tabulizer**, **robotstxt**, **genbankr**)

- **scientific software wrappers**: Packages that wrap 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. 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**)

- **database access**: Bindings and wrappers for generic database APIs (Example: **rrlite**)
- **data munging**: Packages for processing data from formats above. This area does not include broad data manipulations tools such as **reshape2** or **tidyr**, but rather tools for handling data in specific scientific formats. (Example: **plateR**)
- **data deposition**: Packages that support deposition of data into research repositories, including data formatting and metadata generation. (Example: **EML**)
- **reproducibility**: Tools that facilitate reproducible research. This includes packages that facilitate use of version control, provenance tracking, automated testing of data inputs and statistical outputs, citation of software and scientific literature. It does not include general tools for literate programming (e.g., R markdown extensions not under the previous topics). (Example: **assertr**)

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: **rgeospatialquality**, **osmplotr**).
- **text analysis**: We are currently *piloting* a sub-specialty area for text analysis that includes implementation of statistical/ML methods for analyzing or extracting text data. This does not include packages with new methods, but only implementation or wrapping of previously published methods. As this is a pilot, the scope for this topic is not fully defined and we are still developing a reviewer base and process for this area. Please open a pre-submission inquiry if you are considering submitting a package that falls under this topic.

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.

Here are some types of packages we are unlikely to accept:

- Packages that wrap or implement statistical or machine learning methods. We are not organized so as to review the correctness of these methods. (But see "text analysis", above)
- Exploratory data analysis packages that visualize or summarize data.
- General workflow or package development support packages

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.

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 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.

- Consult our policies see if your package meets our criteria for fitting into our suite and is not overlapping 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.
- Read and follow our packaging style guide and reviewer's guide to ensure your package meets our style and quality criteria.
  - If you would like your package to also be submitted to Journal of Open-Source Software (JOSS), it should include a `paper.md` file describing the package. More detail on JOSS's requirements can be found at their website.
  - If you choose this option you should *not* submit your package to JOSS separately. It will be evaluated by JOSS based on the rOpenSci review.
- 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.
  - 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 if the package is at least *Maturing*.
  - At the submission stage, all major functions should be stable enough to be fully documented and tested.
  - 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.
  - Your package will continue to evolve after review, this book provides guidance about the topic.

- 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.
- Next, open a new issue in the software review repository and fill out the template.
- 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 certain issues. *When submitting a package please make sure your GitHub notification settings make it unlikely you will miss a comment.*
- 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. Here is an author response example. We encourage ongoing conversations between authors and reviewers. See the reviewing guide for more details.
- 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

Note that when installing the package to review it, you should use the `dependencies = TRUE` argument of `devtools::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 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 withing 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 package's logs on its continuous integration services (Travis-CI, Codecov, etc.)
- Running `devtools::check()` and `devtools::test()` on the package to find any errors that may be missed on the author's system.
- Using the **covr** package to examine the extent of test coverage.
- Using the **goodpractice** package (`goodpractice::gp()`) to identify likely sources of errors and style issues. Most exceptions will need to be justified by the author in the particular context of their package.
- Using `spelling::spell_check_package()` (and `spelling::spell_check_files("README.Rmd")`) to find spelling errors.

7.1.2 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."

7.1.3 Helper package for reviewers

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("ropenscilabs/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.

7.1.4 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.

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 responsibilities of the Editor-in-Chief, of any editor in charge of a submission, and how to respond to an out-of-scope submission.

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 EiC Responsibilities

The EiC serves for 3 months or a time agreed to by all members of the editorial board. The EiC plays the following roles

- Watches all issues posted to the software-review repo.
- 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 rejects due to being too busy or because of conflicting interests.
- Raises scope/overlap issue with all editors if they see an ambiguous case. This may also be done by handling editors (see below). To initiate discussion, this is posted to the rOpenSci Slack software review channel, tagging all editors.
- Responds to pre-submission inquiries and `meta` issues posted to the software-review and software-review-meta repos, similarly pinging channel if discussion is needed. Any editor should feel free to step in on these. See this section about how to respond to out-of-scope (pre-) submissions.
- Responds to referrals from JOSS or other publication partners.
- Monitors pace of review process and reminds other editors to move packages along as needed.

8.2 Handling Editor's Checklist

8.2.1 Upon submission:

- Tag issue with `1/editor-checks` tag and assign a main editor. Please strive to finish the checks and start looking for reviewers within 5 working days.
- 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 associated editor Anna Krystalli
- Check that template has been properly filled out.
- Check against policies for fit and overlap. Initiate discussion via Slack #onboarding 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.
- Run automated tests: `spelling::spell_check_package()`, `goodpractice::gp()` (most exceptions will need to be justified by the author in the particular context of their package.), `devtools::spell_check()`. Run `covr::package_coverage()` using `NOT_CRAN` if needed, as well. Check that documentation is generated using `roxygen2`, not by hand (this isn't part of automatic tests yet). Report relevant outputs in the issue thread.
- 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 both Travis and AppVeyor 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.
- If initial checks show major gaps, request changes before assigning reviewers.
- 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 reviewers:

8.2.2.1 Tasks

- Switch numbered tag to `2/seeking-reviewers`.
- Ask author to add a rOpenSci review badge to their README. Badge URL is `https://badges.ropensci.org/<issue_id>_status.svg`. Full link should be:

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

- 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 a due date 3 weeks after all reviewers have been found.
- Once two or more reviewers are found, assign reviewers by tagging in the issue with the following format:

```
Reviewer: @githubname1
Reviewer: @githubname2
Due date: YYYY-MM-DD
```

- Switch numbered tag `to 3/reviewers-assigned` once reviewers are assigned.
- Invite authors and reviewers to rOpenSci Slack if they aren't on this Slack already.
- Update Airtable database: Add one of the review record numbers associated with the package to the `Reviews` field of each reviewer's record. Insert new `Reviewers` record if required.

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
- 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.

8.2.2.3 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 all reviews being submitted, change the review status tag to `4/review-in-awaiting-changes` to update the reminder bot.
- Update Airtable database: Add the link to the review comment to the `review_url` field and the number of review hours to the `review_hours` field of each review record.
- 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`.

8.2.4 After review:

- Change the status tag to `6/approved`.
- You can use the comment template.
- Add review/er information to the review database.
- If authors intend to submit to CRAN, direct them to the section about CRAN gotchas and offer to provide support through this process.

- Ask authors to migrate to `ropensci`
  - Create a two-person team in rOpenSci's "ropensci" GitHub organization, named for the package, with yourself and the package author as members.
  - Have the author transfer the repository to `ropensci`
  - Go to the repository settings in rOpenSci's "ropensci" GitHub organization and give the author "Admin" access to the repository.
- Ask author to:
  - Add rOpenSci footer to README `[![ropensci_footer](https://ropensci.org/public_images/ropensci`
  - Add a CodeMeta file by running `codemetar::write_codemeta()` (`codemetar` GitHub repo)
  - Change any needed links, such those for CI badges
  - Re-activate CI services
    - For Travis, activating the project in the ropensci account should be sufficient
    - For AppVeyor, tell the author to update the GitHub link in their badge, but do not transfer the project: AppVeyor projects should remain under the authors' account. The badge is `[![AppVeyor Build Status](https://ci.appveyor.com/api/projects/status/github/ropensci/pkgname?br`
    - For Codecov, the webhook may need to be reset by the author.
- Add a "peer-reviewed" topic to the repo.
- Close the software-review issue.

8.2.5 For joint JOSS submissions:

- After repo is transferred to ropensci and admin rights assigned, have author generate a new release with a DOI.
- Ask author to submit package via http://joss.theoj.org/papers/new
- Watch for paper to pop up at http://joss.theoj.org/papers, then add the following comment to the submission thread:

```
This submission has been accepted to rOpenSci. The review
thread can be  found at [LINK TO SOFTWARE REVIEW ISSUE]
```

8.2.6 Package promotion:

- Ask authors to write either a blog post or a tech-notes post for the package, as appropriate, and ping Stefanie Butland, rOpenSci community manager.
- Alert maintainers of appropriate task views
- Direct the author to the chapters of the guide about package releases, marketing and GitHub grooming.

8.3 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.

Part III

Maintaining Packages

Chapter 9

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.

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.

9.1 Make your repo contribution and collaboration friendly

9.1.1 Code of conduct

We require that you use a code of conduct such as the Contributor Covenant in developing your package. You should document your code of conduct in a `CODE_OF_CONDUCT.md` file in the package root directory, and link to this file from the `README.md` file. `usethis::use_code_of_conduct()` will add the Contributor Covenant template to your package. Use `usethis` > 1.3.1.

9.1.2 Contributing guide

We have templates for issue, pull request and contributing guidelines that you can find in this GitHub repository along with a helper function to insert them into your repository.

You can tweak them a bit depending on your workflow and package. For example, make sure contributors have instructions in CONTRIBUTING.md for running local tests if not trivial. CONTRIBUTING.md can also contain some details about how you acknowledge contributions (see this section) and the roadmap of your package (cf this example).

We encourage you to direct feedback that is not a bug report or a feature request to rOpenSci 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.

You can use the `lintr` package's own bot, lintr-bot, via continuous integration, to get comments if a commit or PR deteriorates the code style of your package. See this link for guidance.

9.1.3 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.

9.2 Working with collaborators

9.2.1 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.

9.2.2 Working with collaborators (including yourself)

You could implement the "gitflow" philosophy as explained by Amanda Dobbyn in this blog post.

One particular aspect of working with collaborators is reviewing pull requests. Even if not adopting gitflow it might make sense for repo collaborators to make PRs and have them reviewed, and in general PRs by external developers will need to be assessed Sometimes you'll be fine just reading the changes and trusting Continuous integration. Sometimes you'll need more exploration and even extension of the PR in which case we recommend reading "Explore and extend a pull request" in happy-gitwithr.com.

9.2.3 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 for ropensci, see <https://github.com/ropensci/software-r
```

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. As of June 2018 you need to use `roxygen2` dev version for the list of authors in the package-level documentation to be compiled properly with the "rev" role (because this is a MARC role not included yet in `royxgen2` CRAN version from February 2017).

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

9.2.4 Welcoming collaborators to rOpenSci

If you give someone write permissions to the repository, please contact one of the editors or Stefanie Butland so that this new contributor might

- get invited to rOpenSci's "ropensci" GitHub organization (instead of being outside collaborators)
- get invited to rOpenSci Slack workspace.

Chapter 10

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.

10.1 Versioning

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

10.2 Releasing

- Using `devtools::release()` will help you remember about more checks. See this issue for a more thorough checklist to go through when releasing a new package version.
- Git tag each release after every submission to CRAN. more info

10.3 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 11

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 post the link to our discussion forum in the Use Cases category and tag Scott Chamberlain (`@sckott`) for inclusion in the rOpenSci biweekly newsletter (or tag Scott on Slack). 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! Example.
  - Consider submitting a technote about the release to rOpenSci technotes blog. Contact Stefanie Butland, rOpenSci community manager, (e.g. via Slack or stefanie@ropensci.org). The guidelines about contributing a blog post can be found here).
  - 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 12

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.

12.1 Make your repository more discoverable

12.1.1 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.

12.1.2 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.

More info about GitHub linguist overrides here.

12.2 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 13

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.

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

13.1 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.

13.2 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 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.

13.3 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
```

13.4 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.

Chapter 14

Contributing Guide

This chapter outlines how you can contribute to the rOpenSci project as a user or developer.

So you want to contribute to rOpenSci? Fantastic! First of all, maybe contributing code or documentation to a package is not the (only) way you'll want to get involved. Check out our community page to find out all the ways to participate in the project, from reading about our packages to volunteering to review them.

We strive to make contributing to our suite a constructive and positive experience and we welcome participation that adheres to our code of conduct.

If you wish to contribute to this guide itself, please report any suggestions via this GitHub repository.

14.1 Why contribute to rOpenSci packages?

In general, as explained by Kara Woo in her talk at the CascadiaR conference, contributing to R packages allows you to make things work the way you want (by adding some functionality to your favorite package), can lead to opportunities and allows you to learn about package development.

Specifically at rOpenSci, we strive to make contributing a good experience. This is embedded in our mission: we are creating social infrastructure through a welcoming and diverse community and building capacity of software users and developers and fostering a sense of pride in their work.

14.2 Non code contributions

14.2.1 Reporting use cases for our packages or guidelines

It's valuable to both users and developers of a package to see how it has been used "in the wild". If you see a use case of one of our packages, please add it to our public forum in the Use Cases category. We've created a template to help. We'll tweet about it, and might add it to the rOpenSci biweekly newsletter. You might also let the package author know by opening an issue in the GitHub repository for that package, perhaps suggesting they create a "use cases in the wild" section in the README, and maybe even tag them in a tweet about the use case. This goes a long way to encouraging people to keep up development knowing there are others who appreciate and build on their work. See examples of README's listing use cases in the wild here and here

We would also love to hear about examples in which rOpenSci resources or guidelines have been used. Have you implemented better security practices, like signing your commits, based on recommendations in a Community Call, or used our standards/checklists etc. when reviewing software elsewhere? Do tell people (e.g. journal editors, students, internal code review group, users) that they came from rOpenSci, and tell us in our public forum, or privately by email.

14.2.2 Reporting issues or Requesting features

If you already have a package in mind, head to its issue tracker to report a bug or a feature request you might have. E.g. if you have a feature request for `magick`, the place to fill it is https://github.com/ropensci/magick/issues

When reporting a bug try to include a reproducible example ("reprex") to help the package author help you.

14.2.3 Asking other questions

You can use our discussion forum to ask questions related to the usage of our packages, even when you don't know yet which of our packages could help you.

For more guidance about where to ask questions related to rOpenSci packages, refer to Scott Chamberlain's talk in this community call or the corresponding slidedeck.

14.3 Code contributions

14.3.1 Prerequisites for package development

Before contributing to one of our packages, you might want to read a bit more about package development in general and our guidelines.

14.3.1.1 Learning about package development

14.3.1.1.1 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

14.3.1.1.2 Books

- Hadley Wickham's *R packages* book
- *Writing R extensions*, the official CRAN guide
- *Mastering Software Development in R* by Roger D. Peng, Sean Kross, and Brooke Anderson
- *Advanced R* by Hadley Wickham
- *Testing R code* by Richard Cotton

14.3.1.1.3 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.

14.3.1.2 Reading about our guidelines

See the whole first section of this book: Building Your Package

14.3.2 Finding where to contribute code or documentation

Our packages are developed on GitHub. We have two organizations:

- https://github.com/ropensci for more mature packages, including some developed by rOpenSci staff and some developed by community members whose packages have gone through software peer review
- https://github.com/ropenscilabs for experiments and packages that are not ready for release yet.

If you already know what package you'd like to contribute to, have a look at its issue tracker and comment in one of the issues to ask the maintainer whether they'd be interested in your help. E.g. if you like `taxize`, the place to look is https://github.com/ropensci/taxize/issues

If you're not sure which package you'd like to contribute to, you can start by browsing our packages page. To browse issues targeted at beginners you could try the `contributr` Shiny app by Lucy D'Agostino McGowan or use an advanced GitHub search such as this one that gets all issues from "ropensci" or "ropenscilabs" organizations that are open and labelled "beginner".

Still not sure how you can contribute? Tell us about your expertise and interests and we'll try to match you with a package that could use your help.

Appendix A

NEWS

A.1 0.2.0

- 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.2 0.1.5

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

A.3 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.4 place-holder 0.0.1

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

Appendix B

Review template

```
## Package Review

*Please check off boxes as applicable, and elaborate in comments below.  Your review is not limit

- [ ] As the reviewer I confirm that there are no [conflicts of interest](#coi) for me to review

#### 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
- [ ] **Installation instructions:** for the development version of package and any non-standard
- [ ] **Vignette(s)** demonstrating major functionality that runs successfully locally
- [ ] **Function Documentation:** for all exported functions in R help
- [ ] **Examples** for all exported functions in R Help that run successfully locally
- [ ] **Community guidelines** including contribution guidelines in the README or CONTRIBUTING, a

>##### For packages co-submitting to JOSS
>
>- [ ] The package has an **obvious research application** according to [JOSS's definition](http:
>
>The package contains a `paper.md` matching [JOSS's requirements](http://joss.theoj.org/about#pap
>
>- [ ] **A short summary** describing the high-level functionality of the software
>- [ ] **Authors:**  A list of authors with their affiliations
>- [ ] **A statement of need** clearly stating problems the software is designed to solve and its
>- [ ] **References:** with DOIs for all those that have one (e.g. papers, datasets, software).

#### 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 guidelin

#### Final approval (post-review)

- [ ] **The author has responded to my review and made changes to my satisfaction. I re

Estimated hours spent reviewing:

- [ ] Should the author(s) deem it appropriate, I agree to be acknowledged as a package

---

### Review Comments
```

Appendix C

Editor's template

```
### Editor checks:

- [ ] **Fit**: The package meets criteria for [fit](policies.md#fit) and [overlap](policies.md#fi
- [ ] **Automated tests:** Package has a testing suite and is tested via Travis-CI or another CI
- [ ] **License:** The package has a CRAN or OSI accepted license
- [ ] **Repository:** The repository link resolves correctly
- [ ] **Archive** (JOSS only, may be post-review): The repository DOI resolves correctly
- [ ] **Version** (JOSS only, may be post-review): Does the release version given match the GitHu

---

#### Editor comments

---

Reviewers:
Due date:
```

Appendix D

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 p

The package, [PACKAGE] by [AUTHOR(S)], does [FUNCTION]. You can find it on GitHub here: [REPO LIN

If you accept, note that we ask reviewers to complete reviews in three weeks. (We've found it tak

Our [reviewers guide] details what we look for in a package review, and includes links to example

{IF APPLICABLE: The authors have also chosen to jointly submit their package to the Journal of Op

Are you able to review? If you can not, suggestions for alternate reviewers are always helpful. I
review at this time.

Thank you for your time.

Sincerely,

[EDITOR]

[reviewers guide]: https://ropensci.github.io/dev_guide/reviewerguide.html
[packaging guide]: https://ropensci.github.io/dev_guide/building.html
[template]: https://ropensci.github.io/dev_guide/reviewtemplate.html
[rOpenSci forum]: https://discuss.ropensci.org/
```

Appendix E

Approval comment template

````
Approved! Thanks <author(s) GitHub username(s)> for submitting and <reviewers' GithHub usernames>

To-dos:
- [ ] Transfer the repo to rOpenSci's "ropensci" GitHub organization under "Settings" in your rep
- [ ] Add the rOpenSci footer to the bottom of your README
"```
[![ropensci_footer](https://ropensci.org/public_images/ropensci_footer.png)](https://ropensci.org
- [ ] Fix any links in badges for CI and coverage to point to the ropensci URL. We no longer tran
- [ ] We're starting to roll out software metadata files to all ropensci packages via the Codemet
<IF JOSS>
- [ ] Activate Zenodo watching the repo
- [ ] Tag and create a release so as to create a Zenodo version and DOI
- [ ] Submit to JOSS using the Zenodo DOI.  We will tag it for expedited review.
<IF JOSS/>

Should you want to acknowledge your reviewers in your package DESCRIPTION, you can do so by makin

Welcome aboard! We'd love to host a blog post about your package - either a [short introduction t

We've started putting together a gitbook with our best practice and tips, [this chapter](https://
````

Appendix F

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
```