Developing an R package: a tutorial

Going further with your R package development

Ghislain Durif

July 2021

CNRS – IMAG (Montpellier, France)

Getting started

Additional R packages to help you create R packages

- `testthat`: to implement automatic tests of your functions
- `remotes`: to install package from anywhere (integrated in `devtools`)
- `rmarkdown` and `knitr`: to create detailed documentation materials and notebooks (code showcase)
- `pkgdown` to create a website for your package

Setup your environment

- install additional R packages providing development tools: `testthat`, `remotes`[1]

```
install.packages(c(”testthat”))
```

[1] not necessary if you already installed `devtools`

Additional references regarding R programming

- Hadley Wickham **book**: *Advanced R* (web version and sources)

Digression: Good practice for software development and programming (not just in R)

Good practice (1)

- The code should be **human readable**[2] and **easily understandable** (use comments, code presentation and formatting)
  - Experiment: read your (5 weeks/months/years) old codes, are you sure that you will understand it? (worst with code written by others)
- Use a **versioning system** (e.g. `git` ↗) to manage your code evolution/version and for collaborative development

[2] being machine readable is necessary for the code to work but not sufficient

Good practice (2)

- Implement **automatic tests** (e.g. unit tests) for each new function/module/etc. (and not afterward) to **verify your implementation and results** and avoid breaking your code[3]

- Use **continuous integration**[4]: to automatically run build, check, tests as your package development progresses (e.g. commit after commit if you are using a versioning system like `git`)

[3] never trust yourself, you will implement bugs

[4] software forge offers such service like `gitlab CI/CD` or `github` actions

Good practice (3)

- Write a **documentation** for your code/package/library, including explained code showcases/demos
- **Publish** your source codes (preferably on a software forge), so that other can continue your work, especially when you move on to other projects, carreer path
- **Archive** your source codes (because your software forge or webpage can disappear)

Software forge (1)

An online server and/or website offering code/software development and management functionality

- versioning
- collaborative work and planning
- issue, feedback, bug reports, feature requests
- software release/publication
- continuous integration
- possibility to get a publication identification like a DOI[5]
- etc.

[5] eventually externally with Zenodo, c.f. later

Software forge (2)

Examples of software forge

- `gitlab`: **free and open-source `git` forge hosting software** (different hosts are available: in the academic world[6] or abroad[7])
- `github`: **very popular**[8] `git` forge with gratis and commercial solutions to host development projects (maybe more simple to reach outside the french academic community)
- other: `bitbucket`

Discontinued forges: `gitorious`, `Google code`, `Inria Gforge` (It happens!)

[6] e.g. `https://plmlab.math.cnrs.fr`, `https://gitlab.inria.fr`, etc.

[7] e.g. `https://gitlab.com`

[8] but owned by Microsoft

Archive your code (publication $\neq$ archiving)

- What happens if your software forge (or the webpage where you host your code) disappear ?
- The Software Heritage initiative
  - “Our ambition is to collect, preserve, and share all software that is publicly available in source code form. On this foundation, a wealth of applications can be built, ranging from cultural heritage to industry and research.”
  - Simple deposit procedure from a software forge[9]

[9] See https://archive.softwareheritage.org/save/

Get a DOI for your code with Zenodo

- a DOI[10] to facilitate your software identification and citation (e.g. in publication using it)
- Upload your codes to Zenodo and get a unique DOI for the current version (possible integration with `github` to directly generate identification for the different versions of your code)
- Possible to identify codes, datasets, creative contents
- More at `https://help.zenodo.org/features/` and in the FAQ

[10] Digital Object Identifier

Test your functions

Implement automatic tests with testthat (1)

- `testthat` provides a seamless (and user-friendly) workflow to implement automatic tests for your package
- **Good practice:**
  - write tests when you code functions (and not after)
  - as soon as you create/modify a function, verify if tests are passing

Implement automatic tests with testthat (2)

- To enable `testthat` in your package:
  `usethis::use_testthat()`
- To create a test case:
  `usethis::use_test()`
  (e.g. `usethis::use_test(”feature1”)`)
- Test file example:

```
test_that(”multiplication works”, {
    res <- my_multiplication(2, 2)
    expect_equal(res, 4)
})
```

`tests` sub-directory structure after initialization

```
tests
+-- testthat
+-- testthat.R
```

`tests` sub-directory structure after creating a test

```
tests
+-- testthat
|   +-- test-feature1.R
+-- testthat.R
```

Implement automatic tests with testthat (3)

- Tests use `expect_XX()` functions to verify conditions of any type on any R expression
- Unit tests (i.e. test regarding a single function/functionality) can be grouped into `test_that(”id”, {})` chunks
- You have to enumerate and write yourself all test cases
- More details at `https://r-pkgs.org/tests.html` and `testthat` exhaustive tour

test

Verify that the **test you write are passing** (e.g. your code is doing what you want)

- `devtools::test()`
- in Rstudio interface (Build panel - More - Test package[11])
- **Note:** tests will be run during package check

[11] keyboard shortcut: `CTRL + SHIFT + T`

Sharing (your code) is caring

Publish and distribute your package

- Others can use your work, collaborate with you to improve it (collaborative development)
- Many repositories: the CRAN (official), bioconductor (bioinformatics-oriented package repository)
- the `remotes` package (exported by `devtools`) can be used to install packages stored almost anywhere on the Internet (`CRAN`, `bioconductor`, `git` forges, etc.) or locally

CRAN

- Strict policy to accept a package (**READ IT!**)

- Pipeline
  1. `devtools::build()` (or `R CMD build`)
  2. `devtools::check()` (or `R CMD check --as-cran`)
  3. upload it[12] to `https://cran.r-project.org/submit.html`

- `devtools::release()` can help you to prepare the release (i.e. the version of your package that will be publish)

[12] in bundle state

Reverse dependencies

- **Important**: if you are releasing a new version of existing package, it is your responsibility to check that it does not break downstream dependencies[13] (i.e. all packages that list your package in the `Depends`, `Imports`, `Suggests` or `LinkingTo` fields)

- `usethis::use_revdep()` to enable the `revdepcheck` package that can help you in that task

[13] called “reverse dependencies”

git

- versioning system: see the `official website` and the `book`
  - manage evolution of your code
  - branch-base system for production/development code cohabitation
  - decentralized system: if you lose your remote, you do not lose the project history
  - easy to distribute (with `git clone`) and to move from remote to remote
- Command line tool or possible to manage everything from R/Rstudio:
  - `usethis::use_git()` to initialize a repository in your project
  - Git panel in Rstudio to manage your local repository and interact with remote (ssh key generation, etc.)
- More detail at `https://r-pkgs.org/git.html`

Distribute your package on a git repository

To install packages hosted on:

- `github`: `remotes::install_github()`
- any `git` forge: `remotes::install_git()`

Possibility to specify the branch, the sub-directory where to find the package, etc.

```
remotes::install_github(”RcppCore/Rcpp”)

remotes::install_git(
    ”https://github.com/getkeops/keops”,
    subdir = ”rkeops”, branch = ”dev”, args=”--recursive”
)
```

Organize your package project

- Package root directory = Rstudio project/git repository root directory (default behavior when using `usethis::create_package()` or Rstudio new project package)
- The package root directory is a sub-directory of the Rstudio project/git repository
  - you can specify the path to your package directory to `devtools` functions
  - Rstudio project setup: Tools - Project Options - Build tools - Package directory

Advanced documentation

Writing a “vignette”

- A document[14] presenting/detailing your package (or a functionality in your package), included in the package (and visible on CRAN)
- Written in a markup language: `Rmarkdown`[15] to integrate R code chunks, or LaTeX or Markdown
- To create a vignette: `usethis::use_vignette(”my-vignette”)`
- Possible to write multiple vignettes (e.g. `Rcpp` package)
- **Rendering** (in pdf/html/etc.) with the package `knitr`

[14] See `https://r-pkgs.org/vignettes.html`

[15] See also this cheat sheet

Create a website

- Create and build a *standardized* website for your package with `pkgdown`[16]
- Hostable on Github or Gitlab pages, or on your own webpage
- To create the website template: `usethis::use_pkgdown()`
- To build the website[17] (e.g. generate the HTML source): `pkgdown::build_site()`
- More details in the `pkgdown` vignette

[16] See also `https://github.com/r-lib/pkgdown`

[17] `README.md` become the homepage, `man` documentation are used to generate function references, and vignettes are rendered into `articles`

Continuous Integration

- Automate package testing and checking when you modify it
- Generally associated with a software forge
- See `usethis::use_gitlab_ci()` or `usethis::use_github_actions()`
- You define a set of actions (e.g. tests and checks) that are run after each commit, or before any pull/merge request (configurable)

Non R code

interfacing language

Rcpp: Seamless R and C++ Integration

- See the **Rcpp webpage** and the **introduction vignette**
- C++ API to use R types and R like functions[18] in C++
- Automatic export of C++ functions to R[19] in particular when creating/building a package
- Expose C++ functions and classes to R[20]
- Conversion from C++ to R and back[21]

[18] See the “Rcpp-sugar” vignette

[19] See the “Rcpp-attributes” vignette

[20] See the “Rcpp-modules” vignette

[21] See the “Rcpp-extending” vignette

Rcpp: compilation on the fly

In `convolve.cpp` file:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b
) {
    int na = a.size(), nb = b.size();
    int nab = na + nb - 1;
    NumericVector xab(nab);
    for (int i = 0; i < na; i++)
        for (int j = 0; j < nb; j++)
            xab[i + j] += a[i] * b[j];
    return xab;
}
```

Compilation on the fly in R:

```
sourceCpp(”convolve.cpp”)
convolveCpp(x, y)
```

Rcpp in a package (1)

- Create a Rcpp-based package template:

```
Rcpp::Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

- All `C++` codes should be in the `src` sub-directory
- Add the comment `// [[Rcpp::export]]` before every C++ functions that should be exported to R
- Add `LinkingTo: Rcpp` in `DESCRIPTION` file

Rcpp in a package (2)

- To generate the C++ to R wrappers: `devtools::load_all()`[22] or `devtools::build()` will call `Rcpp::compileAttributes()`[23]
- The files `src/RcppExports.cpp` and `R/RcppExports.R` are automatically created (or updated) and contain the code necessary to expose your C++ functions in R
- You C++ code will be compiled during your package installation

[22] Reminder: `CTRL + SHIFT + L`

[23] or you can call it yourself

Rcpp in a package (3)

- Compatible with `roxygen2` doc generation
- `Rcpp::compileAttributes()` converts `//'` C++ doc comment chunks to `#'` `roxygen2` doc comment chunks in the `R/RcppExports.R` file

```
#include <Rcpp.h>

using namespace Rcpp;

//' Do something
//' @author someone
//' @description
//' This function does something
//'
//' @param x An integer vector
//' @export
// [[Rcpp::export]]
void my_fun(IntegerVector a) {
    // do something...
}
```

The Rcpp ecosystem (1)

- `RcppEigen`: 'Rcpp' Integration for the `Eigen` Templated Linear Algebra Library
- `RcppArmadillo`: 'Rcpp' Integration for the `Armadillo` Templated Linear Algebra Library
- `RcppGSL`: Rcpp Integration for `GNU GSL` Vectors and Matrices
- `BH`: `Boost` C++ Header Files ("a set of libraries providing support for tasks and structures such as linear algebra, pseudo-random number generation, multi-threading, image processing, regular expressions, and unit testing")
- and more...

The Rcpp ecosystem (2)

How to use the previous C++ libraries in your package ?

1. Install the corresponding R package (with `install.packages(”<pkg>”)`)
2. Add `LinkingTo: <pkg>` in your `DESCRIPTION` file
3. Add the comment `// Rcpp::depends(<pkg>)]]` when including the corresponding library in your C++ code, e.g.:

```
#include <RcppArmadillo.h>
// Rcpp::depends(RcppArmadillo)]]
```

4. Use the C++ corresponding library in a standard way in your C++ code

reticulate: R Interface to Python

CRAN page and webpage

- Calling Python from R (dedicated vignette)

```
library(reticulate)
scipy <- import(”scipy”)
scipy$amin(c(1,3,5,7))
```

- Conversion from R to Python matrix/array (dedicated vignette)
- Python code chunks in Rmarkdown (dedicated vignette])

Managing Python from R

- Python Version Configuration (dedicated vignette and help page)
- Use virtual environment with `reticulate::use_virtualenv()` and `reticulate::use_condaenv()`

Using Python code in an R package

- Using `reticulate` in a R package (dedicated vignette)
- Configuring Python dependencies of your R package (dedicated vignette)

Control your R environment

renv

https://rstudio.github.io/renv/articles/renv.html

packrat

https://github.com/rstudio/packrat/

https://rstudio.github.io/packrat/

Configuring R

- References: here and here
- Configure where you install packages and from where you load packages (i.e. in which directory on your system)
- Setup a default CRAN mirror for package installation
- Define default R objects, functions that will be available without additional file sourcing
- Modify R global options (see the functions `options()` and `getOption()` to check R global options)

.Renviron: configure the environment where R is run (1)

`.Renviron` = a file defining environment variables (as in `bash`) with the following syntax (**!!not R code!!**):

```
Key1=value1
Key2=value2
...
```

To edit your `.Renviron` file, you can use `usethis::edit_r_environ()`.

.Renviron: configure the environment where R is run (2)

- To modify the directory where packages are installed[24] and loaded from[25]: you can set[26] `R_LIBS_USER=/path/to/my/lib/dir` (useful to have project-specific package installation[27])
- Define environment variables (e.g. `MYVAR=5`) that will be available in R (with `Sys.getenv("MYVAR")`) or have an effect an your R code behavior

[24] by `install.package()`, `devtools::install()`, `remotes::install_from_xxx()`
[25] by `library()` or `require()`
[26] default value is ‘`R_LIBS_USER=~/R/%p/%v`
[27] to avoid package version conflict between project

Where storing the .Renviron file

R tries to use an `.Renviron` file in the following order:

1. in the working directory where R is started (if existing), e.g. in your RStudio project root directory
2. in your home directory (if existing)

Note: You can modify this behavior by setting (outside of R/RStudio[28]) the following environment variable: `R_ENVIRON_USER=/path/to/my/.Renviron`

Anyway: R has a global `Renviron.site` file that is read first. Using your own `.Renviron` file allows you to modify the default environment defined in this file.

[28] as in your bash environment

.Rprofile: configure and modify your R session

- `.Rprofile` = an R source file that will be run at R startup (after `.Renviron` was read)

- What for ?
  - define your own default R objects/functions
  - write a startup message
  - modify R global options
  - etc.

To edit your `.Rprofile` file, you can use `usethis::edit_r_profile()`.

.Rprofile: an example

```
# setup a default CRAN repository
options(repos = c(CRAN = ”https://cran.rstudio.org”))

# modify an option only in interactive mode
if(interactive()) {
    options(width = 120)
}
```

Note: interactive mode = as in R console[29] (in RStudio or in a terminal)

[29] versus script mode (like scripts run by `Rscript`)

Where storing the .Rprofile file

R tries to use an `.profile` file in the following order:

1. in the working directory where R is started (if existing), e.g. in your RStudio project root directory
2. in your home directory (if existing)

Note: You can modify this behavior by setting (outside of R/RStudio[30]) the following environment variable: `R_PROFILE_USER=/path/to/my/.Renviron`

Anyway: R has a global `Rprofile.site` file that is read first and using your own `.Rprofile` file allows you to modify the default R session defined in this file.

[30] as in your bash environment

.Renviron/.Rprofile and reproducibility

Attention: you should be careful that your code is usable without your `.Renviron` and `.Rprofile` files

- `.Renviron` and `.Rprofile` files are personal files, another user may configure its environment differently
- **Example:** charging packages or modifying (global or packages) options that have an impact on output values[31] in your `.Rprofile` file may affect the reproducibility of your code (i.e. the results can be different or you code can be broken without your `.Rprofile` file)

[31] e.g. `options(stringsAsFactors = FALSE)`

The end

Questions ?