R Packages: A Concise Reference

Ryan Price
@ryapric

Why?

“Isn’t package development only for if I want to put my code on CRAN/PyPI/PGXN/etc?”

Why?

“Isn’t package development only for if I want to put my code on CRAN/PyPI/PGXN/etc?”

Why?

Packaging your workflow makes even your single-task-focused work more reproducible, deployable, and robust

Why?

- Reduce development time
  - Good IDEs handle most of the boring work for you through hotkeys, command searches, etc.
    - You *thought* you liked RStudio… *just you wait*
  - Stop wasting time with ad-hoc testing, data validation, etc.

Why?

- Deliver more robust solutions & recommendations
  - Modularizing your code into functions/methods allows for much more flexibility than most tutorials give credit for
    - *It's what packages are designed around*
  - Be able to gracefully handle when incoming data changes, or when new/future functionality is requested/stubbed out

Why?

- Become equipped with the skills to develop your own open-source projects, or contribute to existing projects
  - All extensible FOSS languages require packaging your code in order to distribute it through the formal channels, subject to approval by their governing bodies.
  - Whether that's something you care about, or has never crossed your mind, it can be a very rewarding experience, both to you and to the community at large.

How (do we get there)?

- My goal is to help you to understand:
  - The native R build system, and why packages are an important part of it
  - Why packaging your code is important in any workflow
  - Explore test suites
  - How to embrace modularity
    - Ashley Home Furniture --> top-shelf IKEA furniture

How (do we get there)?

- My goal is to help you to understand:
  - How to develop a mental framework where you follow these approaches as second-nature
    - You only have to do this a few times before you'll start to punish yourself for not starting with a package-based workflow, and you'll notice how easy it is

How (do we get there)?

- … And all without `"Hello World"/print(x)` examples
  - That garbage drives me **INSANE**

The R Packaging System

The best resource for authoring R packages is Hadley Wickham's book on the subject. It's available freely online.

For a boilerplate directory to get you started faster, please visit my package boilerplate repository.

The R Packaging System

R packages are perhaps the most rigorous implementation of a packaging system that you will encounter.

They depend heavily on proper formatting, often down to the whitespace used and specific documentation provided.

However, these strict requirements will help you appreciate the vast amount of checks that R performs on packages, and you may even find yourself upset that other tools don't have these checks.

The R Packaging System

Additionally, R provides something that no other software (that I've encountered) provides: `R CMD check`, which is a remarkably thorough and powerful command-line tool that checks all sorts of things about your package: documentation consistency, unit test results, conflicts in the overall namespace, code formatting, and tons of other tidbits.

Getting Started

- To build R packages along with this guide, you will need:
  - R >= v3.0
  - The RStudio IDE
  - The `devtools`, `roxygen2`, and `testthat` R packages
  - OS-specific:
    - Windows: the RTools bundle, available as a download from CRAN
    - Linux-based/macOS: it depends; watch for warnings. Definitely C, C++, and Fortran compilers, but these are installed along with R on Ubuntu, for example
  - ***Patience***

Developing From Scratch with RStudio

- To make an example R package structure, do the following in RStudio:
  - Click the Project icon at the top right → New Project → New Directory → R Package
  - Give it a name (cannot contain underscores, spaces, or other special characters, but *can* contain periods), and be sure to check “Create a git repository” at the bottom.
- You will have some placeholder files & folders generated for you

Developing From Scratch with RStudio

- Once the Project opens, also do the following:
  - Make sure the `devtools`, `roxygen2`, and `testthat` packages are installed
  - Go to Tools → Project Options → Build Tools
    - Check the box “Generate documentation with Roxygen”, and then click on the “Configure” button next to it
    - Check *all the options in this menu*

Note About a Clean Workspace

A clean workspace is of paramount importance when writing code that works the same every time it is run. If you've ever solved some mysterious problem by closing, and then reopening your software environment, you might understand this.

Make sure all polluting options are turned ***off*** in RStudio's Global Options → General menu, such as "Save/Restore .Rdata".

Note About a Clean Workspace

RStudio provides a keyboard shortcut to entirely clear your active workspace, wiping your environment & namespace clean. ***You should run this every single time you change something, or want to re-run some bit of code***. None of that first-line `rm(list = ls())` garbage.

The shortcut is `Ctrl` + `Shift` + `F10`

Package Metadata: DESCRIPTION File

The **DESCRIPTION file** (all caps) contains the metadata about your package. It serves to inform both an end-user as well as the R build system of this metadata. This file sits at the root of the package directory.

Information such as the package name, description, authors, dependencies, and other valuable data are available in the DESCRIPTION.

Package Metadata: DESCRIPTION File

```
Package: stlfedR
Title: Dummy Package to Demonstrate Package-Based Development Workflow
Description:
    Longer description of package. Can span multiple lines.
Version: 0.1.0
Date: 2019-12-05
Authors@R:
  c(person("Ryan", "Price", email = "ryapric@gmail.com", role = c("cre", "aut")))
License: MIT + file LICENSE
Depends: R (>= 3.4.0)
Imports:
  data.table (>= 1.11.8),
  forecast (>= 8.4),
  glue (>= 1.3.0),
  here (>= 0.1),
  httr (>= 1.4.0),
  lubridate (>= 1.7.4),
  RSQLite (>= 2.1.1)
Suggests:
  devtools (>= 2.0.0),
  knitr (>= 1.19),
  rmarkdown (>= 1.8),
  testthat (>= 2.0.0)
URL: https://github.com/ryapric/pbdw
BugReports: https://github.com/ryapric/pbdw/issues
RoxygenNote: 6.1.1
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
```

Package Namespace: NAMESPACE File

The **NAMESPACE file** (all caps) contains metadata about what functionality is imported & exported from your package. It defines your package's *namespace* when used called via `library(mypkg)`.

The NAMESPACE file needs to be very specifically formatted, and while it's not hard to understand the formatting, you should leave the dynamic generation of the NAMESPACE file to the `roxygen2` package, which we'll see later.

Package Namespace: NAMESPACE File

So, if you generated package files via “New Project → Package” in RStudio, *immediately delete the NAMESPACE file that it made for you*, so `roxygen2` can make a managed one.

Package Files & Folders

The rest of your package's root directory will contain a few special dotfiles (.gitignore, .Rbuildignore, CI files, etc.), and then the package files themselves stuffed into folders.

Package Files & Folders

- These folders are most commonly:
  - R
    - The main application code files go in here
  - man
    - Documentation output
  - inst
    - “Installed files”, e.g. main scripts, supplemental data, etc.
  - tests
    - Contains files that run unit & integration tests

Main Code: R Folder

Your top-level R folder is where all your function & object definitions go, into various `.R` files.

Note that these are ***not scripts!*** They should ***only*** contain function or method definitions, global config objects, etc.

Documenting Your Work

As mentioned previously, R is ***very picky*** about things like documentation. In fact, your package checks will fail if your function, method, etc. documentation is not ***exactly*** as R expects.

This used to be a prohibitive nightmare, as R documentation is written entirely in a LaTeX-alike typeset (`Rd` format). The development of the `roxygen2` package has significantly lowered this barrier to entry, as it automatically generates the `Rd` for you based on human-readable docstrings you write.

Documenting Your Work

Roxygen is very deep, and could have a talk all its own; so the easy way to generate the docstring skeleton is to first write a function, place your cursor inside the body, and hit `Ctrl + Alt + Shift + R` on your keyboard.

This will prepopulate the docstring skeleton directly above your function definition, and you can edit it from there. You can differentiate Roxygen comments from regular comments by the trailing apostrophe, i.e. `#'` vs. `#`

We will see more of Roxygen docblocks when we start the workshop portion, but here's its full documentation (under "Vignettes").

Documenting Your Work

```r
#' Add
#'
#' Adds `x` and `y` together.
#'
#' @param x,y Values to add together
#' @return The `numeric` sum of `x` and `y`
#' @export
add <- function(x, y) {
    x + y
}
```

Unit Testing

Now that you've written some functions, and documented them, you'll want to write some *unit tests*. Unit testing your software ensures that the smallest bits of functionality ("units") behave exactly as expected.

Depending on the software being written, unit testing doesn't need to be universally exhaustive, per se; but your tests should cover *every use case* you can think of at the time of writing, both how your functions should ***and should not*** behave.

Test Frameworks

Many software provide a wide array of choices for *test frameworks*. These are libraries that make writing *unit tests* easier (and more fun!).

Perhaps fortunately, the R ecosystem only has one test framework that most people use, `testthat`, which makes writing unit tests both easy & human-readable.

We will see how `testthat` works in the workshop, but here's its full documentation.

Test Frameworks

```
context("Arithmetic operators")

test_that('add(x, y) returns the sum of x and y', {
  expect_equal(add(1, 1), 2)
})
```

R's Build System

Now that your tests are finally written, your package is ready to be tested, built, checked, and/or installed!

RStudio makes the test, build, check, and install processes very easy: they're just keyboard shortcuts! `Ctrl` + `Shift` + `<key>`, where `<key>` is:

- Test -- `T`
- Install (by building first) -- `B`
- Check -- `E`

R’s Build System

But, how do you run these commands when RStudio’s not around to do them for you?

RStudio’s hotkeys are just a wrapper around system & R commands, so the functionality of the build system is accessible via the command line.

R’s Build System

The command-line set of instructions are provided by running `R CMD <command>`, where `<command>` is one of:

- `build` -- builds a distributable `.tar.gz` file of your package
- `check` -- runs the full R check system on the zipped tarball
- `INSTALL` (note the all-caps) -- installs the package locally

Unit tests are run as part of the `check` command, but you can run them from the command line via: `Rscript -e 'devtools::test()'`

Top-Level Documentation

Don’t forget a README.md!!!

READMEs are long-form, high-level documentation on how your package works, how to use it, common use cases, gotchas, etc. It’s critically important, but many developers counterintuitively scoff at the idea of such documentation. If your code is missing a README, **YOUR CODE IS NOT DELIVERABLE**.

The R Packaging System

Interactive workshop: Your First R Package!