Package Development Notes

Paolo Luciano Rivera

2018

Introduction

Developing a Package in `R` might seem like an impossible task or one that requires significant amount of coding experience. You'd be right. However, thanks to the people behind RStudio, specially our lord and savior *Sir. Hadley Wickham The Ultimate Hipster* this process has become significantly easier and can be done with minimal obscure `R` coding.

This book will mainly reference (copy tbh) his work on the book R Packages. Where he goes into a way deeper explanation on every aspect of Package Development. This notes, provide a high level explanation on how to build a package from scratch. The way it has been designed on RStudio, really simplifies most technical aspects and allows the user to focus on what's important, the actual `R` code and testing it.

Why develop a package?

Because coding a package makes your life way easier than just scripting. The main idea of developing an `R` package is not to build a ready to use product, it's more along the lines of a *framework* that will help every data scientist seamlessly integrate and test his or her code. Because we are lazy, most common task have been automated and because we're stupid documenting everything for your future self has been made easy. This way of coding might be a little over-the-top for some people that usually wants a quick answer, however, when you review your work several months in advance and you actually understand what's that vectorized auxiliary function is doing, it'll definitely pay off. Besides, a package is the "fundamental unit of shareable code" and it will make your life easier when you're trying to do something a bit more complex than doing a simple scatter plot.

To summarize, why develop a a package in `R` with RStudio?

- Because, it provides a framework for organizing and structuring code
- Because, it is the *basic unit of shareable code*
- Because it automates a lot of trivial things like: documenting the package, building top level files and auto-testing for bugs.
- Because version control is made simple
- Because it protects you from low-level coding mistakes and makes sure the package works
- Because `devtools` is one of the `R` packages out there

As you continue to work and develop packages, you'd might want to understand how these low-level details work. The best and most complete resource for it, is the official, but less friendly Writing R Extensions located in CRAN.

Getting started

What do you need?

- `R`
- RStudio
- A couple of packages:

  ```
  install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
  ```

- Low level development tools: RTools in windows, XCode on Mac and `r-base-dev`.

Now test that everything works by running

```
library(devtools)
has_devel() == TRUE
```

If it returns TRUE, you should be ready to go do some hardcore coding.

Let's now build a package

1. Open RStudio and click on *New Project.*
2. Name your package (hardest decision ever) and select where will it live. Tip: do it simple and don't use fancy characters.
3. ???
4. profit

Let's change some options for further documentation just not to break anything (we will review all of this later)) and *voilà*, you have a working package[1]. For real. Test that by changing the first function and pressing `Crtl+Shift+B` this is the shortcut to *build the package* (we will cover that later) and run the function `hello` in the terminal.

How to reopen my project? and start working on it again? Just open RStudio and reopen the *project* from the drop-down menu and/or open directly from the folder it is located at. Projects by themselves are very usefu tools for structuring code since they are *isolated* and they have handy features for bigger projects (`F2` for function editing (just like Excel! and `Ctrl + .` for the best search function out there (Press `Alt+Shift+K`) to view them all).

Now, click on the `pckname.Rproj` file, this allows us to work on the project options. Besides all the R stuff here is where I would integrate it with GitHub:

```
git init pck.dir
cd pck.dir
git status
git commit - m "First commit of First pck <3"
git add remote PaoloLuciano...
git push remote master
```

What's a package and what does it contain?

A package (in its source form) is a collection of files and directories with a very defined structure and documentation. The most relevant components of it are (in somewhat order of importance):

- **R Code**: specifically, a directory where all the functions live.
- **Package Metadata**: a file named `DESCRIPTION` that describes what is your package and what's required for it to work. It also contains the license and other boring stuff.
- **Documentation**: all the files that allow others (and future you) to understand what is actually happening and what does every function does. They should be as detailed as possible. This might be the most boring part of package development but `reoxygen2` makes it a walk in the park. Specially if you already know how to write LaTeX
- **Vignettes**: Think of this as as the home page Wikipedia for your package. It might even provide examples on how to use everything from the package.
- **Tests**: why not write automated test that ensures the package works as expected as you keep on developing it (or that others don't break your precious work).

[1]If you want to feel a bit more hackery you can go: `devtools::create("path/to/package/pkgname")`

- **Namespace**: it ensures that if you creatively name your function `plot` it doesn't break all other `plot` functions out there.
- **Compiled Code**: want speed? Here you can put some more low level `C++`

What a package is is not

This is important: **a collection of scripts**. Everything that you code, **must** be a function, or should be something that creates objects (S3 and S4 classes). If you want to make scripts do theme somewhere else. As with almost all software, the code and structure we can edit and work on is very different than what a computer can actually use. So there's a certain *lifecycle* that packages go trough. Those are:

1. **Source**: a package on it's most raw form. It's just the directory we've been working with.
2. **Bundled**: package compressed into a single file they use extension `.tar.gz`. It's not very useful to be honest. If a package can be built, then it its good to go (at least the technical parts your code can be shit and still be able to build it).
3. **Binary**: the package you can actually distribute. It's a single file but quite different than a bundled one, a bundled package is platform (OS) specific.
4. **Installed**: binary package that has been decompressed into a library. When you run `install.packages(...)` you're usually downloading packages from `CRAN`, installing them and turning them into a library of functions that can be *loaded* into `R` .
5. **In-memory**: The package in it's most *pure* form where you can actually use it. A package must be loaded into memory and it's done using the traditional `library(...)` function.[2]

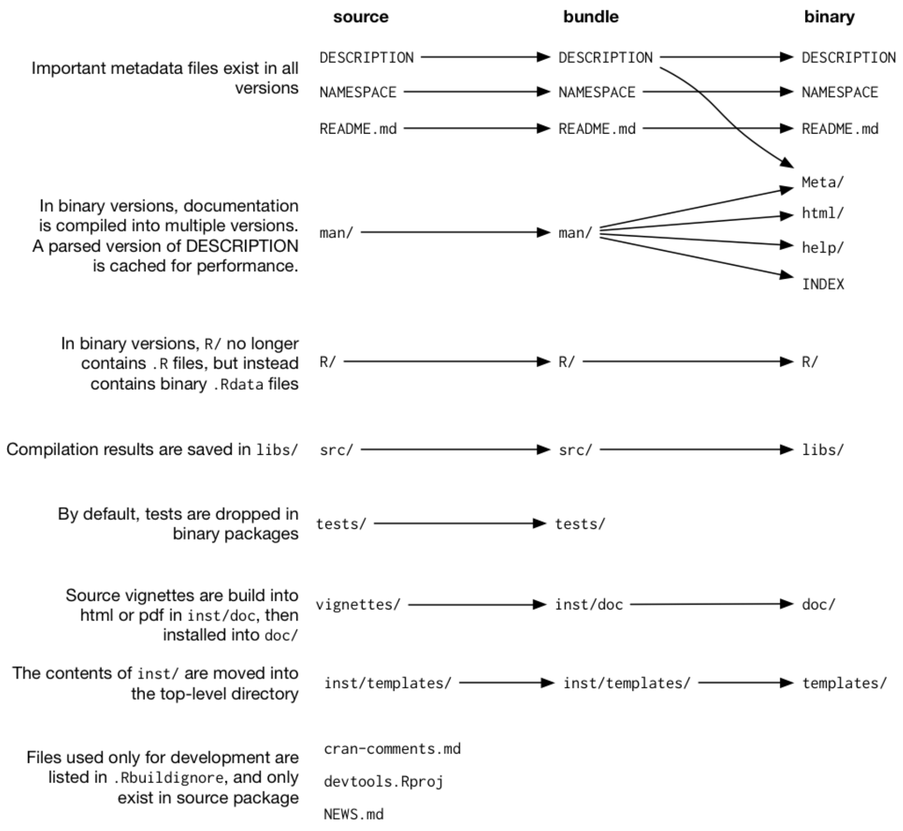
There are actually quite a few ways to install and use a package. One that I really like is: `devtools::install_github()` that automatically downloads a source package from Github, builds it and installs it. On a quick note, a library is different than a package in the sense that a library is simply a directory that contains installed packages. When developing a package, this distinction and further considerations regarding paths are really important and should be addressed in more detail.

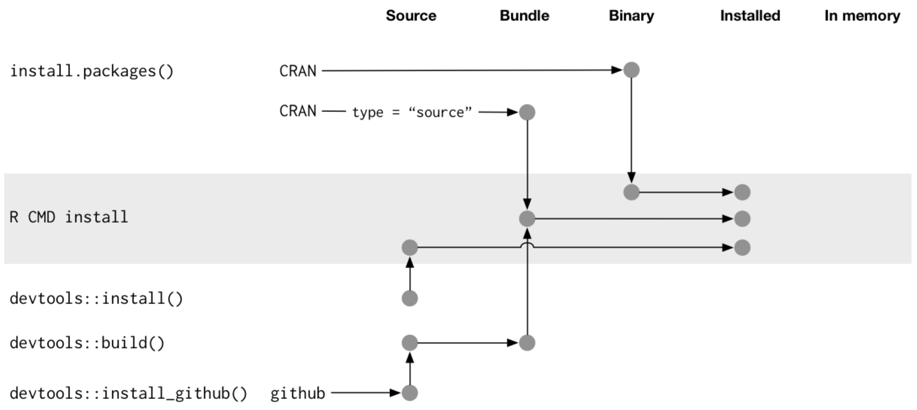
Now what?

Now, now you code. You already know that part. However there are a few important details. First All code must go into the smartly named `R\` sub-directory of your package. All good practice coding tips should be applied to this section, a few good practice tips:

- Function names should be lowercase
- Don't use _ for anything but S3 and S4 methods (OOP)
- Space words and symbols
- RStudio can autoformat stuff for you
- `<-` is superior to `=` for assignment.
- Name files accordingly and comment the code (different than documenting it)

[2]It's different to use `load()` and `require`


source
bundle
binary
Important metadata files exist in all versions
DESCRIPTION
NAMESPACE
README.md
In binary versions, documentation is compiled into multiple versions. A parsed version of DESCRIPTION is cached for performance.
man/
Meta/
html/
help/
INDEX
In binary versions, R/ no longer contains .R files, but instead contains binary .Rdata files
R/
Compilation results are saved in libs/
src/
libs/
By default, tests are dropped in binary packages
tests/
Source vignettes are build into html or pdf in inst/doc, then installed into doc/
vignettes/
inst/doc
doc/
The contents of inst/ are moved into the top-level directory
inst/templates/
templates/
Files used only for development are listed in .Rbuildignore, and only exist in source package
cran-comments.md
devtools.Rproj
NEWS.md



One of the main differences between a package and a the regular code (scripts) most of us are used to write (specially in a statistical language), is that, in a script, the code is run when loaded. In a package code is run when built, therefore, the package should only create objects. The reason is that a package is meant to be used multiple times and to be as flexible as possible. **Never run code at the top level of a package**, for example, never run `library(...)` inside of a package (we will cover this next).

Description file and integration with other packages

When developing a package, you need to think that other people are going to use it. The fundamental idea is to share the package even though if you only use it yourself, this mindset will help you become a better coder overall. One of the main building blocks of this mentality is the `DESCRIPTION` file. This file is simply a text file (DCF extension) that contains all of the *Metadata* of the package. You can view it's contents directly in the RStudio interface or in any text editor of choice. It contains details about the package and your contact information for when you get famous.

However, when building simple packages, the most important role of the `DESCRIPTION` file is to make it play well with other packages. It clearly states which other packages need to be loaded with you package for it to work properly. To add packages you need to run:

```
usethis::use_package("ggplot2")
```

And when adding functions from other packages to yours, always specify from which package is each function. For example:

```
simple.plot <- function(n){
data <- data.frame(x = rnorm(n,1,2), y = runif(n,5,6))
ggplot2::ggplot(data = data, aes(x = x, y = y)) + geom_point() +
geom_smooth(method = "loess")
}
```

The `::` operator ensures that R searches for the function on the correct `NAMESPACE`. This has to do with the *lexical scoping* `R` is so good at. There can be several issues with the path functions and variables are referring to and this will ensure they play along nicely. The `NAMESPACE` file in your package also contains information about what objects are actually declared inside of your package. This can get tricky, but for now, we will allow `devtools` to handle that, when a package is built, the namespace is automatically updated. However, trying to run this function will not work due to problems with the absolute `NAMESPACE`. We will fix this by documenting our functions!

Documenting your package, the right way to do it

Documenting you code is important, and `R` can make the task simple, efficient and helpful. This, is with the help of a `roxygen2` package (also developed by H. Wickham). The options we changed at the beginning help us integrate this package into out workstream (and not break things). For now we will delete a couple of files, this because `reoxygen2` will not overwrite the `NAMESPACE` file and the original `.Rd` files but will create new ones that allow us to work seamlessly. The `roxygen2` package first defines a clear structure for commenting functions that's built into RStudio. A good way to quickly document functions is to use the shortcut: `Ctrl + Alt + Shift + R` to build a documentation skeleton for your function.

All documentation files created by `roxygen2` have the extension `.Rd` and are documents that somewhat resemble a LaTeXfile. They will be stored on the `man/` directory. However, RStudio can render those files in HTTP and present them on a readable way; the useful `?func` fetches the documentation page for that specific package or function and presents it on the console. That's why `CRAN` help documentation looks so shitty (but not DataCamp) .

All *comments* in `roxygen2` start with a `#'`. All lines before a function are called a block. Each block has **tags** attached to it that stat with a `@`. However, before using tags, there are three special lines (paragraphs) that `roxygen` parses automatically and have speciall meanings. They are:

- **Title**: first. line of the documentation.
- **Description**: Paragraph explaining what the function does.
- **Details**: self explanatory

There are a lot of tags and they, will tell `roxygen2` what are you trying to specify. Roughly speaking the ones you'll use the most are:

- `@params` Created automatically for every input/or parameter on your function where you specify the data type, representation, etc. All parameters should be documented
- `@inheritParams` to inherit parameters for other functions (do repeat yourself)
- `@examples` To teach the user how to call the function
- `@export` To put the function in the `NAMESPACE` so that it can be used by others (think of this as public and private functions)
- `@import` To import from other packages
- `@return` To specify the output of the function
- `@seealso` For further resources like web pages
- `@family` For grouping functions.

So now we can fix the `NAMESPACE` from the package by adding the line `@import ggplot2` to our previous function. Note that there are other types of more general types of documentation such as *vignettes* and the `DESCRIPTION` file itself. You can add a lot of other fancy Latexy stuff to the documentation like links, footnotes, code, lists and math so feel free to be as explicit as you can.

Conclusion

And that's pretty much it. You can now have working packages within minutes and ones that actually enable you to focus on what's most important, the content of the package.
A quick summary of all the steps (for this version of R, RStudio, devtools, etc.) is:

1. Open RStudio and click on *New Project*.
2. Name your package and select where will it live.
3. Change options for documenting with `roxygen2`
4. Delete `\man` directory and `NAMESPACE` file
5. Edit details of `DESCRIPTION` file
6. Code first function
7. `Ctrl + Alt + Shift + R` document function
8. `Ctrl + Shift + D` to build
9. Initialize Github
10. Repeat until something breaks