Statistics, Visualization and more using R

R Packages: A Beginner's Guide

Bradley Mackay, Clemens Ehlich

30. Juni 2020

Summary

One of the biggest advantages of the R programming language is the extensive number of extension packages available which make tasks in R much easier. R packages are collections of functions and datasets and are mostly created by the community itself. As the title suggests, this short paper is about how to create your own little R package, because creating your own package has many advantages. Especially if you have to work on recurring tasks. For this we will first give a summary of important commands to work with packages. Then we will give an overview how packages are built and finally we will give a step by step guide how to create your first own package.

Master Data Science
PhD Linguistics
Universität Salzburg
clemens.ehlich@sbg.ac.at
bradley.mackay@sbg.ac.at

Inhaltsverzeichnis

1 Basics

1.1 What's a package

R packages are collections of functions and data sets. They extend the basic functionalities of the Language itself or add new ones. Packages are usually written in R code, but there are packages which extend this functionality to other languages for example, C++ (with the `rcpp` package (Eddelbuettel & Balamuta, 2017)). As a general rule of thumb, if there are any functions which you have copied and pasted from other R scripts more than twice, you should consider creating a package for those functions.

1.2 Why Packages?

Using packages makes your work-flow a lot easier and faster. Because for recurring tasks its not necessary to reinvent the wheel again and again. There are more than 19.000 packages on CRAN. For nearly every common problem a package exist, for example to load or visualize data. Packages form also interfaces to other software and their file formats (foreign, caffe, RQGIS), databases (RODBC, RPostgreSQL, etc), other programming languages (Rcpp, RPython) or webservices (Rfacebook, RGoogleAnalytics).

1.3 Where can I find packages?

To get an overview of which packages are already available you can use the website *https://www.rdocumentation.org/*. It lists nearly every package exists. You can also look at

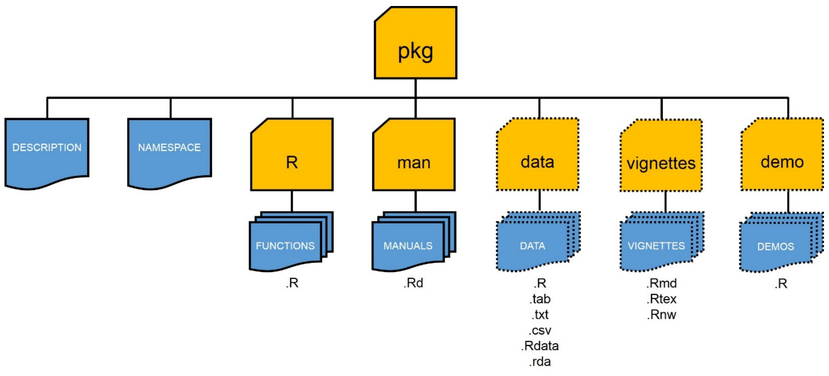

Abbildung 1: Package file structure

https://cran.r-project.org the official R Website or BioConductor a website for Biology Content (*http://bioconductor.org/*). Of course there are also countless packages on GitHub (*https://github.com/*).

1.4 Some important R-Commands to work with packages

There is one big problem: each repository (CRAN, BioConductor, etc) has its own way to install a package from them. To simplify this process you can use the package "*devtools*". But for this you might also need to install *Rtools*" for Windows, "*Xcode*" for Mac or if you using Linux $r - devel$" for Linux. But then you can install packages directly from the repository with the following code:

```
install_bioc() #from Bioconductor
install_cran() #from CRAN
install_github() #from GitHub
install_local() #from a local file
install_url() #from a URL
```

2 Making you own Package

2.1 What does a package consist of?

In this chapter we want to show how to create your own package. For this it is important to know how a package is built. If you create a new file to create an R-package, a folder structure is automatically created in the background. This folder structure is shown in figure 1. The DESCRIPTION-File, the NAMESPACE, the FUNCTIONS and the MANUALS are mandatory. The rest, which are dotted in the graphic, is optional.

While it is possible to write a package from scratch, there are a number of packages available which make the process much easier. By making use of functionality in `devtools` (Wickham, Hester & Chang, 2020), `usethis` (Wickham & Bryan, 2020) and `roxygen2` (Wickham, Danenberg, Csárdi & Eugster, 2020), almost all of the manual input is kept to a bare minimum. It is perfectly possible to have a functioning package (with no frills!) up and running in about 10 mins using the functions provided by these three packages. In terms of beginning a package project, there are several different ways to initialise a `.Rproj` which will require slightly altered ways which you need to use `roxygen2` in the initial project set up. For clarity, the work-flow described here is for those initialising a new .Rproj as an R package in a clear directory.

2.2 Initial setting up

Initialising a new R package project will give you the bare-bones file structure required to make a package, as illustrated in 1. The file structure was however not created with `roxygen2` and so there is some initial set up required to get round this. Firstly, buildtools needs to be configured to use `roxygen2` functionality, it's a good idea to also initialise a github repository for your package at this point if that is part of your intended work-flow. A feature of `roxygen2` is that it will not overwrite any files that it did not create, while this is beneficial if you are switching to `roxygen2` in a package that you have been previously building without it, it means that we need to delete some of the existing shell so that `roxygen2` can create these files itself. The shell files include the NAMESPACE file as well as the .Rd file in the man folder (leaving this file would not cause any issues to your package run but it is superfluous to the package structure requirements). The existing "Hello World".R file in the R folder (from where the .Rd file was created) can also be deleted, or just renamed and overwritten with your first function. Best practice for functions is to have one file per function, but they can happily stay in one file. All manual additions are made to files in the R folder, .Rd files in the man folder are read only and are generated by `roxygen2`. If you open one of these .Rd files you will see that the syntax is loosely based on LaTeXsyntax. Calling `devtools::document()` (which actually itself calls `roxygen2::roxygenise()`) at this point will rewrite NAMESPACE and the basic required structure will be once more intact.

3 Adding content & roxygensing your functions

Once you have a function that you want to include as part of your package, you need to tell `roxygen2` to include it. This is easily done by adding a roxygen block to your function:

```
#' Title
#'
#' @param
#' @return
#' @examples
#' @export
```

This roxygen skeleton can be inserted by putting your cursor anywhere in the function and then clicking code then *InsertRoxygenSkeleton*. running `devtools::document()` again will roxygenise your function and create a .Rd file in the man folder. The information you enter here will appear in the help files of your package. You may get a warning if you haven't started **`#' param #' return`** with a capital letter and ended with a full stop. A useful function which should be run often, and certainly after roxygenising functions, is `devtools::check()`, this installs and restarts your package and then performs a check on your code. It will then give you feedback on whether your package check has exited with a 0 or 1 status and give you either the error, warning or note that has been generated. You can simply work your way through each of these messages until your check passes clean.

3.1 Further development

Ultimately the amount of time invested in writing supplementary files that are not essential to the basic functioning of your package depends on the final destination you envisage for your package. If you plan to keep your package available only in your local files, there is little to no point in providing extensive vignettes for each function for example. Similarly, if you'd be content to live with any number of warnings when you call your package, then you may not feel it necessary to spend time cleaning up your file structure to pass `devtools::check()` without warnings. If you plan to release your package into the wild however, your package structure needs to adhere to much stricter package checks, certainly if you plan to submit to CRAN, there will be a greater number of supplementary checks and files that need to be included in your package. On this note, there seems to be very mixed reactions to CRAN's submission process. Although CRAN developers are actively encouraging submissions, a cursory search of the #CRAN handle on twitter suggests that even very well established members of the R community, including staff on the Rstudio team, have had very negative experiences of pushing their packages to CRAN.

3.1.1 Checking package names

If you do plan to release your package, you need to check first of all that your planned package name is available on (ideally) all platforms. Rather than having to go to each site individually, there is a handy package named `available` (Ganz, Csárdi, Hester, Lewis & Tatman, 2019) which has a few useful functions. The most useful of these is `available::available()`. This function does two things, it first checks github, Bioconductor and CRAN and tells you if your name is available or not. It also checks a number of websites including urbandictionary.com to see if your name has any offensive meanings you may not be aware of. It also optionally opens these pages in your web browser (you can turn this option off with `browse = FALSE`). you can optionally do a target search e.g. `available::available_on_github()`. Once you have done these checks, your package is theoretically ready to go.

3.2 Making your package more attractive to users

If you want people to actually use the package you have created, rather than someone else's package which may have similar functionality, it's a good idea to make your package as accessible as possible to your intended users. This means initializing an informative README and creating vignettes to walk the reader through the main functions in your package. READMEs can easily be generated with the `usethis::use_readme_rmd()`. This will create both a .rmd and a .md file in the main directory. You can edit the .rmd file and it will automatically update the .md file. There is a similar function from `usethis::use_vignette()` which automatically generates a vignette for you to complete. All this information will then be available to users once they have installed your package through the `help()` function, but to call attention to your package in the first place is also important. The `pkgdown` package (Wickham & Hesselberth, 2020) can help tremendously here. `pkgdown` takes the README and any vignettes you have initialised along with your description file and turns them into a static HTML site. This site can then be hosted free of charge on github pages. This is simply executed by calling `pkgdown::build_site()`. You may also want to think about creating a hex logo for your package, while it is absolutely not necessary, a good hex logo could help your package stand out in a crowded market. There is a `hexSticker` package (Yu, 2020) which will help you here but hex stickers could be simply created as .png files in photoshop.

If you have initialised a pkgdown site you may need to add some of the files it creates to build ignore (unless you don't mind getting a note/ warning!).

3.3 Once your package is done

Once you have a package which contains useful, well-documented functionality, and which is hosted in a way that allows people to make use of it, there are a number of options available to track and store usage metrics. If your package is available through github you can track downloads through packages such as `badger` which allow you to embed badges into your README file. Badges are also capable of tracking downloads from CRAN and Bioconductor and these metrics can be broken down by month or year. To track package usage straight from CRAN or Bioconductor, there are packages such as `cranlogs` (Csárdi, 2019) and `dlstats`, which have the advantage of providing data in a format which can be stored and visualised in R or further processed into (for example) a Shiny app and tracked that way.

Literatur

Csárdi, G. (2019). cranlogs: Download logs from the 'rstudio' 'cran' mirror [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=cranlogs` (R package version 2.1.1)

Eddelbuettel, D. & Balamuta, J. J. (2017, aug). Extending extitR with extitC++: A Brief Introduction to extitRcpp. *PeerJ Preprints*, *5*, e3188v1. Zugriff auf `https://doi.org/10.7287/peerj.preprints.3188v1` doi: 10.7287/peerj.preprints.3188v1

Ganz, C., Csárdi, G., Hester, J., Lewis, M. & Tatman, R. (2019). available: Check if the title of a package is available, appropriate and interesting [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=available` (R package version 1.0.4)

Wickham, H. & Bryan, J. (2020). usethis: Automate package and project setup [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=usethis` (R package version 1.6.1)

Wickham, H., Danenberg, P., Csárdi, G. & Eugster, M. (2020). roxygen2: In-line documentation for r [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=roxygen2` (R package version 7.1.0)

Wickham, H. & Hesselberth, J. (2020). pkgdown: Make static html documentation for a package [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=pkgdown` (R package version 1.5.1)

Wickham, H., Hester, J. & Chang, W. (2020). devtools: Tools to make developing r packages easier [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=devtools` (R package version 2.3.0)

Yu, G. (2020). hexsticker: Create hexagon sticker in r [Software-Handbuch]. Zugriff auf `https://CRAN.R-project.org/package=hexSticker` (R package version 0.4.7)