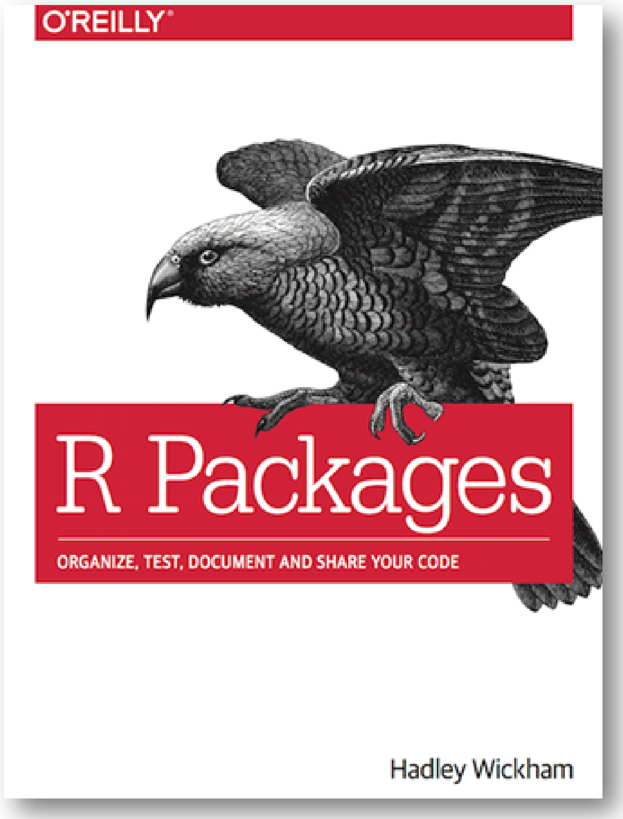
R Packages

RK

May 17, 2015

Abstract

This document contains a brief summary of the book titled, "`R` packages", by Hadley Wickham.



Context

This week, I took up the task of cleaning up some project files at work. In the past, I had written a bunch of `R` scripts to automate a few pre & post trade processing tasks. For other traders to access this functionality, I had created a `Shiny` app and deployed over local intra-net. I knew that some day I had to turn all the scripts in to an `R` package and host it on a common repository. Managed to do that activity this week. I have cleaned up the entire project directory, turned scripts in to functions and have hosted the entire functionality as a package on the intranet. During this exercise, I went over the book by Hadley Wickham. I guess it is quicker to read a tech book when the book meets your immediate requirement. After reading through the entire book, my biggest learning is,

> Write your code as though you are going to release it as a package

I think this kind of orientation will result in a code that is efficient and extensible. In this document I will list down some of my learnings from reading the book.

Introduction

A package bundles together code, data, documentation, and tests, and is easy to share with others. The packages needed to build your own package are `devtools, roxygen2, testthat, knitr`.

Package Structure

- A package can have five states, i.e. source, bundled, binary ,installed, in memory
- source : a source package is just a directory with components like *R/*, *DESCRIPTION/* etc..
- bundled : a bundled package is a package that's been compressed into a single file. By convention, package bundles in `R` use the extension *.tar.gz*. This means multiple files have been reduced in to a single file (*.tar*) and then compressed using gzip(*.gz*).
- binary : a binary package is useful if you want to distribute the package to an `R` user who doesn't have package development tools. The binary is different from a source package in the following ways
  - There are no *.R* files in *R/* directory. Instead there are three files that store parsed functions in an efficient file format.
  - A *Meta/* directory contains a number of *Rds* files
  - An *html/* directory contains filed needed for HTML help
  - Any code in *src/* directory will be present as compiled code in *libs/* directory
  - The contents of *inst/* are moved to the top-level directory
- The chapter provides a good visual (Figure 1) to bring out the differences between source, bundled and binary packages
- installed : an installed package is just a binary package that's been decompressed into a package library. The tool that powers all package installation is the command-line tool R CMD INSTALL. Devtools has a wrapper to the command line tool. The chapter provides a visual (Figure 2) to illustrates five ways to install a package.
- in memory : this is a state of the package where it can be used in other `R` functions. To use a package, you must load into memory. To use it without proving the package name, you need to attach it to the search path.

- The distinction between loading and attaching packages is not important when you are writing scripts, but its very important when you are writing packages.
- The chapter has a visual (Figure 3) to show three ways to load a package into memory
- A library is simply a directory containing installed packages. You can have multiple libraries on your computer. In fact, almost every one has at least two: one for packages you've installed, and one for the packages that come with every R installation (like base, stats, etc).
- I came across a nice analogy on stackoverflow that helps one to differentiate between the terms package and `library`.

  > Consider the library at your home. When you install a package, it is akin to buying a book and placing it in your book shelf. When you use `library` function, it is akin to taking out the book from the shelf and placing on your desk to read it.

`R` code

A practical advantage to using a package is its easy to reload your code. `devtools::load_all()` loads all the code.

- One way to automatically format your code is by using `formatR` package
- Variable and function names should be lowercase
- Put spaces around all infix operators and = in function calls
- Place a space before left parentheses, except in a function call. Do not place spaces around code in parentheses or square brackets
- When you load a script with `source,` every line of code is executed and the results are immediately made available. When you load a package with `library(),`the cached results are made available to you. This means that you should never run code at the top-level of a package
- package code should only create objects, mostly functions
- Never use `source()`to load code from a file
- Don't use `library()` or `require()`
- If you modify global `options,`save the old values and reset when you are done
- Avoid relying on the user's environment. Clearly state the dependencies in *DESCRIPTION* file

Package Metadata

- The job of the *DESCRIPTION* file is to store important metadata about your package.
- The main fields of metadata are
  - Package Name
  - Title
  - Description
  - Version
  - Date
  - Depends
  - Imports
  - Suggests
  - Enhances

  - Published
  - Author
  - Maintainer
  - BugReports
  - License
- The most important tags in the *DESCRIPTION* file are *Imports, Suggests, Depends*
- Packages listed in *Imports* must be present for your package to work
- Packages listed in *Suggests* are not automatically installed along with your package
- The easiest way to add *Imports* and *Suggests* to your package is to use `devtools::use_package()`
- You almost always want to specify a minimum version rather than the exact version of the package
- The difference between *Imports* and *Depends* is the former loads the library whereas the latter attaches a library
- LazyData makes it easier to access data in your package. The datasets will not be loaded until you use them
- If you do not know which license to use, just use *MIT* as it is simple and permissive.

Object Documentation

- Object documentation is a type of reference documentation. It works like a dictionary. If you know the name of the object, the documentation helps you know about it. If you want to find the object which does the job, you should use *vignette*
- `roxygen2`can be used to make documenting your code as easy as possible.
- The workflow for documentation is
  - Add roxygen comments to your `.R`files
  - Run `devtools::document()` to convert roxygen commands to *.Rd* files
  - Preview documentation with ?
  - Rinse and Repeat until the documentation looks the way you want
- To see documentation links between pages, use *Build and Reload* option
- For every function, one needs to add `@export`before the start the function code for `roxygen`to include it in *NAMESPACE* file.
- For documenting packages, create a `pkgname.R`file and add `NULL` at the end of `roxygen` documentation
- One can put multiple functions in one file and then use `@rdname`so that all the documentation for these functions will appear in a single page
- The common roxygen tags are `@param, @return, @examples, @export, @format, @source, @rname, @describein`
- For each dataset, one can also provide documentation using the `NULL` hack

Vignettes: Long-From Documentation

- `devtools::use_vignette("name")` will create a v*vignettes/* directory , adds the necessary dependencies to *DESCRIPTION* file, and puts in a sample *.Rmd* document in the vignetter directory
- Before R 3.0.0, the only way to create a vignette was with Sweave. This was challenging because Sweave only worked with LaTeX, and LaTeX is both hard to learn and slow to compile. Now, any package can

provide a vignette engine, a standard interface for turning input files into HTML or PDF vignettes.
- The book uses R markdown vignette engine provided by `knitr`
- There are three important components to an R Markdown vignette:
  - The initial metadata block : This is written in YAML
  - Markdown for formatting text
  - Knitr for intermingling text, code and results
- The `RMarkdown` also allows you to convert *.Rmd* files to beamer presentations

Testing

This chapter is extremely useful while developing package incrementally. It introduces `testthat` package and walks through various examples to communicate its usefulness. To setup a package to use `testthat`, one can use `devtools::use_testthat()` package. My mode of testing *.R* files is pretty naive. I will start using `testthat`from now on, to make the process systematic.

Namespace

- The *NAMESPACE* file helps you make your package self-contained. It won't interfere with outer packages, and other packages won't interfere with it.
- The basic steps of populating *NAMESPACE* file are
  - Export functions by placing `@export` in the roxygen comments
  - Import objects from other packages with `package::object`, `@import`, `importFrom`, `importClassesFrom`
- Irrespective of the order in which packages are loaded, you can use a function using `pkg::fun()` syntax
- There is a difference between *Loading* and *Attaching* a package. Loading an installed package will load code, data and any DLLs, register S3 and S4 methods, run the `.onLoad` function. After loading a package, it is available in memory but is not available on the search path. *Attaching a package* puts the package on the search path
- The difference between *Imports* and *Depends* is the former loads the library whereas the latter attaches a library
- Unless there is a good reason, you should always list packages in *Imports* as it does not mess with the search path
- Never manually write the directive of *NAMESPACE* file. Let `Roxygen2` generate the file
- If you are using other package functions selectively, you can probably write `exptpkg::fun1`
- If you are using other package functions in many places, you can probably write `importFrom extpkg fun1`

External Data

- There are three main ways to include data in your package, depending on what you wan to do with it
  - If you want to store binary data and make it available to the user, put it in *data/*
  - If you want to store parsed data, but not make it available to the user, put it in *R/sysdata.rda* This is the best place to put data that your functions need.
  - If you want to store raw data, put it in *inst/extdata*
- Always use *LazyData : true*

- To document a dataset, create a *.R* with `NULL` and then write doxygen directive. One should not use `@export`. Instead one should use `@name`
- All the files in *insta/extdata* are copied in to the top directory while installation.

Installed Files

- All the contents in *inst/* directory are copied in to the top-level package direction post installation
- One should never use a subdirectory with the same name as an existing directory

The last part of the book talks about *best practices* that includes a chapter on using git and a chapter on the way to do automated testing during package development stage.

Takeaway :

"Write your code as though you are releasing it as a package" - This kind of thinking forces one to standardize directory structure, abandon adhoc scripts and instead code well thought out functions, and finally leverage the devtools functionality to write efficient, extensible and shareable code.

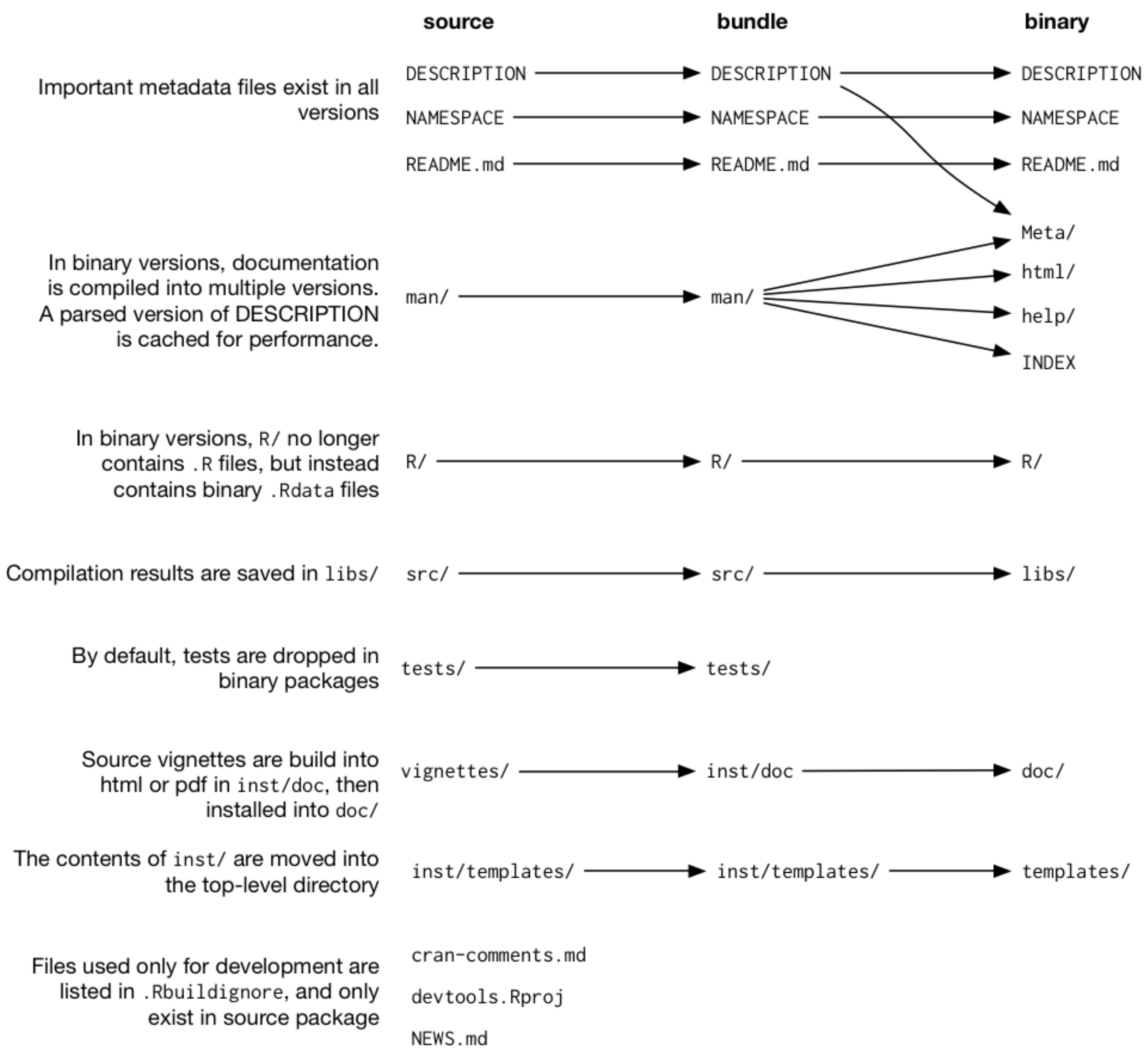

source
bundle
binary
Important metadata files exist in all versions
DESCRIPTION
NAMESPACE
README.md
Meta/
In binary versions, documentation is compiled into multiple versions. A parsed version of DESCRIPTION is cached for performance.
man/
html/
help/
INDEX
In binary versions, R/ no longer contains .R 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

Figure 1: Package states : source, bundled and binary

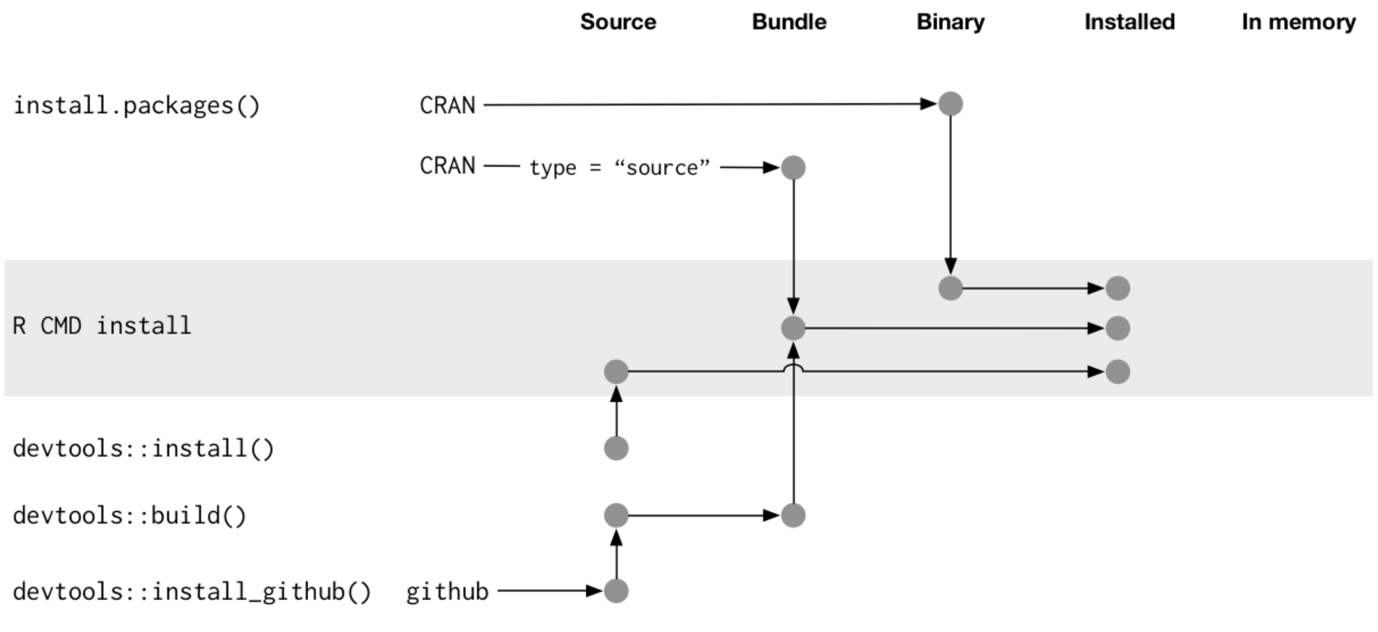


Figure 2: Five ways to install a package

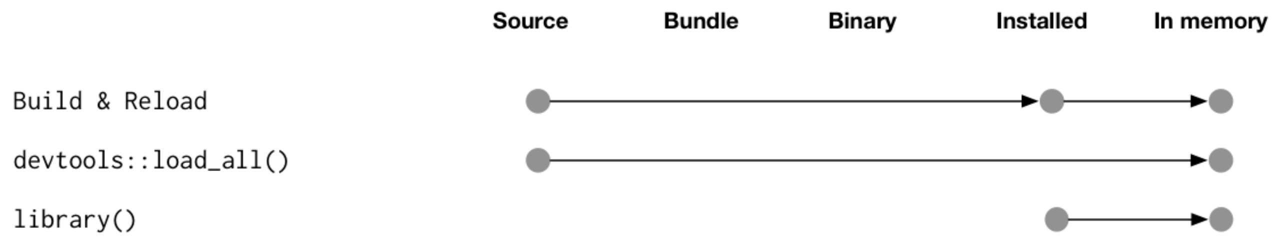


Figure 3: Three ways to load a package into memory