Packages I

Lecture 20

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Rough Plan

- Why packages?
- Basic package structure
- Package building
- Package documentation (roxygen2)
- Resources:
  - Lander, *R for Everyone*, 2017, Chapter 30
  - Wickham, *R Packages*, 2015
  - R Core, *Writing R Extensions*

Packages

Key Features

- Basic unit of code organisation for R
- Entire infrastructure for code
  - Building
  - Testing
  - Documenting
  - Distributing
  - Deploying
  - Versioning
- Even when you do not plan to submit to CRAN or alike
- Excellent model to organize code at work or in a lab

Key Resources

- Definite treatment is the Writing R Extensions manual
  - Included with every copy of R
  - Also on the web
  - Admittedly a little "dense" – but authorative
- Many on-line tutorials and posts plus short coverage
- We will look at at Lander (2017), *R for Everyone*, Chapter 30
- The book R Packages (Wickham, 2015) is also popular (website)

So What is a Package?

- Typically a collection of several R functions
- Usually with documentation
  - manual pages (also called help pages)
  - optional vignettes in html or pdf
- Help pages frequently contain examples
- Packages may contain compiled code (not our focus today)
- Packages may contain tests (more next lecture)

Essential Structure

- `DESCRIPTION` *required* file defining the package name, license, description and dependencies
- `NAMESPACE` to declare what is imported and exported
- `R/` very common directory with R functions
- `man/` very common directory with documentation
- `src/` *optional* directory with C or C++ or Fortran or ... source code which R will automatically compile and make loadable
- `data/` *optional* directory with data sets
- `inst/` *optional* directory for *other directories to be included*
- `tests/` *optional* tests starting point (more next lecture)

DESCRIPTION

- A simple collection of 'key: value' pairs
- The format follows the Debian Control File specs
  - So there are existing parsers
  - R itself has functions should you need them
- There are a number of important mandatory fields
  - See next few slides
- As well as additional optional fields

DESCRIPTION

- `Package`, `Title`, `Version`, `Type` supply important meta-data
- `Author` and `Maintainer` state authorship and copyright, as well as a person to contact for questions and issues
- `Description` is used for a brief synopsis including references
- `Depends`, `Imports`, `Suggests` delineate dependencies
  - `Depends` used to be standard, attaches the named packages
  - `Imports` now preferred in conjunction with the `NAMESPACE` file
  - `Suggests` lists *optional* packages that are used if present but are not essential for the package to be functional
- `License` should list a known open source license (see manual)

DESCRIPTION Example

```
Package: pkgKitten
Type: Package
Title: Create Simple Packages Which Do not Upset R Package Checks
Version: 0.1.4
Date: 2016-11-13
Author: Dirk Eddelbuettel
Maintainer: Dirk Eddelbuettel <edd@debian.org>
Description: Provides a function kitten() which creates cute little
 packages which pass R package checks. This sets it apart from
 package.skeleton() which it calls, and which leaves imperfect files
 behind. As this is not exactly helpful for beginners, kitten() offers
 an alternative.
License: GPL (>= 2)
Suggests: whoami (>= 1.1.0)
NeedsCompilation: no
Packaged: 2016-11-13 14:52:34.898805 UTC; edd
Repository: CRAN
Date/Publication: 2016-11-13 16:50:45
```

NAMESPACE

- Lists which packages are imported from
  - These must be listed in `DESCRIPTION` and its `Imports` field
  - Can import packages 'whole' or just particular identifiers
- Similarly states what is exported from a package
  - Default value is a regular expression for all visible identifiers
  - (Allowing invisible one starting with a dot)
  - You can also selected exports explicitly naming functions
- For packages with compiled code
  - Lists the dynamic library containing the code

Package Creation Helpers

- There are several
- The official one is `package.skeleton()` in base R
    - leaves suboptimal package that does not pass `R CMD check`
- So I once wrote `pkgKitten` and its `kitten()` to postprocess
- There is also RStudio's File -> New Project -> New Directory -> R Package (but sadly not with RStudio Cloud :-/)
- (And also `devtools` / `usethis` which I am less familiar with ...)

RStudio Use – and RStudio Cloud workaround

- On a regular RStudio instance on your computer (or lab), try
  - File -> New Project -> New Directory -> R Package
- Using RStudio Cloud, create a new package via
  - Packages menu, select 'Install' and write `pkgKitten`
  - Or at the R prompt do `install.packages("pkgKitten")`
  - Once installed do `library(pkgKitten)`
  - And, say, `kitten("simpleTestPackage")`
- Now we trick RStudio Cloud
  - select `project.Rproj` and 'Rename' to `simpleTestPackage/project.Rproj`
  - then navigate to the directory and open the project file

RStudio

- In the 'Build' menu (top right pane)
  - Click 'Install and Restart'
  - Click 'Check'
- These are the two main access points
- Under 'More' additional options
- Also on command-line
  - `R CMD build` to build
  - `R CMD check` to check

DESCRIPTION file

Auto-generated so some fields need filling in:

```
Package: simpleTestPackage
Type: Package
Title: What the Package Does Using Title Case
Version: 1.0
Date: 2018-12-28
Author: Your Name
Maintainer: Your Name <your@email.com>
Description: More details about what the package does. See
  <http://cran.r-project.org/doc/manuals/r-release/
  R-exts.html#The-DESCRIPTION-file> for
  details on how to write this part.
License: GPL (>= 2)
```

NAMESPACE file

Very simple for a simple package:

```
exportPattern("^[[:alpha:]]+")
```

R code

```
## a placeholder
hello <- function(txt = "world") {
    cat("Hello, ", txt, "\n")
}
```

Very simple `R/hello.R` with a placeholder function

Function has one argument with a default value

manual / help file

```
\name{hello}
\alias{hello}
\title{
  A simple function doing little
}
\description{
  This function shows a standard text on the
  console. In a time-honoured tradition, it
  defaults to displaying \emph{hello, world}.
}
\examples{
  hello()
  hello("and goodbye")
}
```

Also short and simple. Note that we reindent the `\description` block.

Rd files use a simple markup language.

It is documented in *Writing R Extensions*.

Roxygen2 help files: Start from empty stanza

```
#' Title
#'
#' @param txt
#'
#' @return
#' @export
#'
#' @examples
hello2 <- function(txt = "world") {
  cat("Hello, ", txt, "\n")
}
```

We copy `hello()` over to `hello2()` here.

We then use Code -> Insert Roxygen Skeleton and the file changes to what is shown on the left.

(Of course, the text lines can be added by hand too.)

Roxygen2 help files: Filled-in example

```
#' A simple function doing little
#'
#' @param txt A text argument
#'
#' @return Nothing, side effect of cat()
#' @export
#'
#' @examples
#' hello("world")
hello2 <- function(txt = "world") {
  cat("Hello, ", txt, "\n")
}
```

With `roxygen2` installed, one can edit Tools -> Project Options -> Build Tools and check the box for 'Generate documentatin with Roxygen'.

Generating Rd from the R file is the default, one can also add to `NAMESPACE` following the `@export` tag.

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/hello2.R
\name{hello2}
\alias{hello2}
\title{A simple function doing little}
\usage{
hello2(txt = "world")
}
\arguments{
\item{txt}{A text argument}
}
\value{
Nothing, side effect of cat()
}
\description{
A simple function doing little
}
\examples{
hello("world")
}
```

Here the file on the left gets generated.

Personally, I like `roxygen2` for Rd files. I am less convinced about `@export` tags but many find that useful too.

If you want to do it manually:

```
library(roxygen2)
roxygenize(".", roclets="rd")
```

Roxygen2

Following 'Help -> Roxygen Quick Reference' opens a viewer pane with a quick summary shown on the right (and a bit more)

Explains a few commands and contains some further references.

```
#' This is the title.
#'
#' This is the description.
#'
#' These are further details.
#'
#' @section A Custom Section:
#'
#' Text accompanying the custom section.
#'
#' @param x A description of the parameter 'x'. The
#'   description can span multiple lines.
#' @param y A description of the parameter 'y'.
#'
#' @export
#'
#' @examples
#' add_numbers(1, 2) ## returns 3
#' ## don't run this in calls to 'example(add_numbers)'
#' \dontrun{ add_numbers(2, 3) }
#' ## don't test this during 'R CMD check'
#' \donttest{ add_numbers(4, 5) }
add_numbers <- function(x, y) {
     x + y
}
```

Command-line

- As mentioned earlier:
  - `R CMD build .` builds in source directory
  - `R CMD build simpleTestPackage` builds from dir above
  - `R CMD check simpleTestPackage_1.0.tar.gz` checks it
  - `R CMD INSTALL simpleTestPackage_1.0.tar.gz` installs
- All these commands have RStudio IDE equivalents
- While `devtools` / `usethis` helper functions are available, knowing the basic functions is essential