Yet Another 'Writing R packages' tutorial

George G. Vega Yon

January 24, 2017

Today

- Have a look at some R-package-dev practices based mostly on my own experience (so the product is sold AS-IS)
- We won't discuss either Rcpp or coding R itself.
- We won't discuss github in depth. I'll assume that the user already knows the basics of git(hub): Creating a repo online, and start using it locally.
- We won't talk about software engineering/design, rather we'll take a look at some tools and dev cycle.

What we'll be using today

a. **devtools**: "Package development tools for R" (more).

b. **roxygen2**: "A 'Doxygen'-like in-source documentation system for Rd, collation, and 'NAMESPACE' files" (more).

c. **covr**: (An R package for) "Test Coverage for Packages" (more).

d. **RStudio**: "RStudio is an integrated development environment (IDE) for R" (more).

e. **valgrind**: (if you use C/C++ code) "a memory error detector" (more).

Why 'spending' time writing an R package?

To name a few:

1. Easiest way to share code: Just type `install.packages` and voilà!
2. Already standarized: So you don't need to think about how to structure it.
3. CRAN checks everything for you: Force yourself to code things right.
4. ...

What's an R package, anyway?

According to Hadley Wickham's "R packages"

> Packages are **the fundamental units of reproducible R code**. They include **reusable R functions**, the **documentation** that describes how to use them, and **sample data**.

Dev cycle

Writing R packages is an iterative process

1. **Set up the structure**: Create folders and files `R/`, `data/`, `src/`, `tests/`, `man/`, `vignettes/`, `DESCRIPTION`, `NAMESPACE`(?)
2. **Code!** For `f` in `F` do
   2.1. Write `f`
   2.2. Document `f`: what it does, what the inputs are, what it returns, examples.
   2.3. Add some tests: is it doing what is supposed to do?
   2.4. `R CMD Check`: Will CRAN, and the rest of the functions, 'like' it?
   2.5. Commit your changes!: Update `ChangeLog+news.md`, and commit the changes!
   2.6. next `f`

Step 1: Set up the structure

You have several methods: package.skeleton(), Rstudio's "New Project", etc. I personally do it from scratch (including the code):

1. Create the package dir, say `funnypkg`
2. Create the `R` dir
3. Create the `DESCRIPTION` file (more):

```
Package: funnypkg
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: MIT + file
Encoding: UTF-8
LazyData: true
```

4. From RStudio, create a new project pointing to the package folder.

And, using the `devtools` package, we can add some extras

```r
# Creating a README for the project
devtools::use_readme_rmd()

# Infrastructure for testing
devtools::use_testthat()

# Infrastructure for Code Coverage
devtools::use_coverage(type = "codecov") # This creates the .travis.yml
devtools::use_appveyor()

# A LICENSE file (required by CRAN)
```

```
# The line "license: MIT + file" in the DESCRIPTION file
devtools::use_mit_license(copyright_holder = "George G. Vega Yon")
devtools::use_news_md()
```

What did just happen?

1. `use_readme_rmd()`: Creates a readme file that will be in the main folder of the project. Think about it as the home page of your project.
2. `use_testthat()`: Will create the basic infrastructure for package testing.
3. `use_coverage(type = "codecov")`: Creates the .travis.yml file for Unix CI, and sets it up for code coverage using codecov.io.
4. `devtools::use_appveyor()`: Creates the appveyor.yml for windows CI.
5. `devtools::use_mit_license(copyright_holder = "George G. Vega Yon")`: Creates the `LICENSE` file and puts it under 'George G. Vega Yon'.
6. `devtools::use_news_md()`: Creates the `news.md` file which us used for tracking changes, and communicating them to the users (e.g. netdiffuseR)

Step 1: Write a function roxygen2

```
#' The title of -foo-
#'
#' @param a Numeric scalar. A brief description.
#' @param b Numeric scalar. A brief description.
#'
#' @details Computes the sum of \code{x} and \code{y}.
#' @return A list of class \code{funnypkg_foo}:
#' \item{a}{Numeric scalar.}
#' \item{b}{Numeric scalar.}
#' \item{ab}{Numeric scalar. the sum of \code{a} and \code{b}}
#' @examples
#' foo(1, 2)
#'
#' @export
foo <- function(a, b) {
  ans <- a + b
  structure(list(a = a, b = b, ab = ans)
  , class = "funnypkg_foo")
}
```

Press Ctrl + Shift + D (RStudio will: create the manual, and the NAMESPACE). Make sure you activate this option in RStudio (not the default)

Step 2: Extend it

```
#' @rdname foo
#' @export
#' @param x An object of class \code{funnypkg_foo}.
```

```
#' @param y Ignored.
#' @param ... Further arguments passed to
#' \code{\link[graphics:plot.window]{plot.window}}.
plot.funnypkg_foo <- function(x, y = NULL, ...) {
  graphics::plot.new()
  graphics::plot.window(xlim = range(unlist(c(0,x))), ylim = c(-.5,1))
  graphics::axis(1)
  with(x, graphics::segments(0, 1, ab, col = "blue", lwd=3))
  with(x, graphics::segments(0, 0, a, col = "green", lwd=3))
  with(x, graphics::segments(a, .5, a + b, col = "red", lwd=3))
  graphics::legend("bottom", col = c("blue", "green", "red"),
         legend = c("a+b", "a", "b"), bty = "n",
         ncol = 3, lty = 1, lwd=3)
}
```

You'll need to update the `man/*Rd` everytime that you add new roxygen content. Just press Ctrl + Shift + D and RStudio will do it for you

---

- Notice that we are using the functions `plot.new`, `plot.window`, `axis`, and `legend` from the `graphics` package, which `R CMD check` will notice. You'll need to add it in the NAMESPACE via roxygen2, for which I recommend using a file called `funnypkg-pkg.r`

  ```
  #' @importFrom graphics plot.new plot.window axis legend
  NULL

  #' funnypkg
  #'
  #' A (not so) funny collection of functions
  #'
  #' @description We add stuff up... You can access to the project
  #' website at \url{http://github.com/USCBiostats/funnypkg}
  #'
  #' @docType package
  #' @name funnypkg
  #'
  #' @author George G. Vega Yon
  NULL
  ```

  Notice that the description of the file is also here.

- And in the `DESCRIPTION` file:

  ```
  Package: funnypkg
  Type: Package
  Title: What the Package Does (Title Case)
  Version: 0.1.0
  Author: Who wrote it
  Maintainer: The package maintainer <yourself@somewhere.net>
  Description: More about what it does (maybe more than one line)
      Use four spaces when indenting paragraphs within the Description.
  License: What license is it under?
  Encoding: UTF-8
  LazyData: true
  ```

```
Imports: graphics
```

Step 3: Add some tests

In the `tests/testthat/` dir, add/edit a source file with tests, e.g. `test-basic.r`

```
context("Basic set of tests")
test_that("foo(a, b) = a+b", {
  # Preparing the test
  a <- 1
  b <- -2

  # Calling the function
  ans0 <- a+b
  ans1 <- foo(a, b)

  # Are these equal?
  expect_equal(ans0, ans1$ab)
})

test_that("Plot returns -funnypkg_foo-", {
  expect_s3_class(plot(foo(1,2)), "funnypkg_foo")
})
```

You can run the tests using Ctrl + Shift + t

Step 4: Checking the package

1. If you don't have C/C++ code Just press Ctrl + Shift + E
2. If you have C/C++ code, use `R CMD Check` with valgrind (check for segfaults)

   ```
   $ R CMD build funnypkg
   $ R CMD check --as-cran --use-valgrind funnypkg*.tar.gz
   ```

   You can ask RStudio to use valgrind too.

Step 5: Commit your Changes

- If after adding/changing the code nothing breaks, then you are good to go with your changes!

  ```
  $ git commit -a -m "Adding function abc"
  $ git pull
  $ git push
  ```

- Ideally, you'll want to track changes using `ChangeLog` and `NEWS.md` files.

References

Mostly from experience, and

- Hadley's R-pkgs
- Writing R Extensions