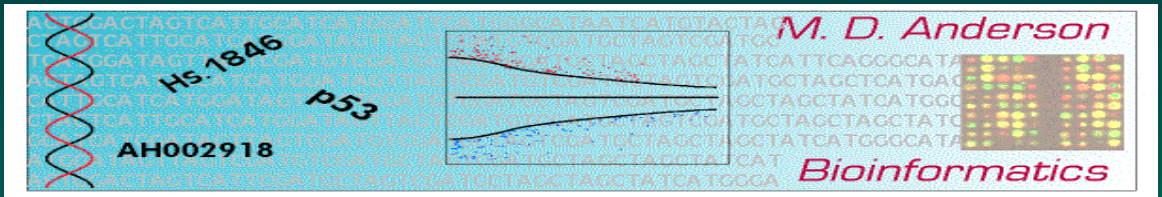
Writing R Packages

Keith A. Baggerly
Bioinformatics and Computational Biology
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org

SISBID, July 20, 2015



R Packages are *Cool*

They let us easily share code, data, and documentation

They can be shared at common sites such as

CRAN and
Bioconductor

We're going to show you how you can assemble a minimal R package in under 15 minutes.

Writing R Packages Can be Painful

Why is assembling packages hard?

They have fairly specific structure and *documentation requirements*.

Most of the documentation (.Rd) files use a syntax based on Latex, which can take practice in itself

Getting all the interconnections right can be annoying and time-consuming, not to mention counterintuitive

The final reference, the manual on "Writing R Extensions" is not ideally welcoming to newcomers

It's Less Painful Today (1)

Writing packages is less painful today primarily because of *other R packages* from Hadley Wickham and Yihui Xie.

These packages shift bookkeeping from frustrated mortals to laconic computers:

devtools
roxygen2
knitr
rmarkdown

It's Less Painful Today (2)

Writing R packages is also less painful today because Rstudio makes it easy to build *packages* as *projects*.

Let's test this out using the most basic package we can imagine: one with just one function.

We begin with

File/New Project/Package

Our Toy Package

Here's what we'll start with.

```
plotCircle <- function(r=1){
  myAngles <- seq(from=0, to=2*pi,
    length.out=200)
  x <- cos(myAngles)
  y <- sin(myAngles)
  plot(r*x, r*y, type="l",
       main=paste0("A Circle of Radius ",
         r))
}
```

What Rstudio automatically supplies

First, we set up a new package project in a brand new directory, and let Rstudio set it up.

Then we load the libraries mentioned above.

Rstudio (0.98.110) fills the package with 5 files and two folders, each containing one file each in turn (7 total).

While we'll look at all of them now, in most cases we'll

delete two,
leave two alone, and
edit three.

So, What Files Are There?

DESCRIPTION (edit)
NAMESPACE (don't touch directly)
Read-and-delete-me (delete) - not in 0.99
.Rbuildignore (maybe edit)
toyPackage.Rproj (leave alone)
R/toyPackage.R (empty, edit)
man/toyPackage-package.Rd (delete) - not in 0.99

To these, we add

R/plotCircle.R

Things We Need to Do

add comments for the function
add comments for the package
edit the DESCRIPTION
specify what to ignore in .Rbuildignore
remove some default files
document, build, install, check

Later, we'll

add a README file
add a vignette
add data

What do we need to say about a *Function*?

What it is (TITLE)
What it does (DESCRIPTION)
What arguments it takes
What outputs it returns
How it works (examples)
What other functions it relates to
Whether it's visible to the user

We do all of this using a special form of comments, which are then parsed by *roxygen2*.

Adding roxygen2 function comments (1)

All roxygen2 comments begin with #'

```
#' Plot a circle of radius r.
#'
#' Normally, we would now describe what
#' the function is supposed to do
#'
```

Every function must have

a TITLE (one line), and
a DESCRIPTION (one paragraph).

Adding roxygen2 function comments (2)

After the TITLE and DESCRIPTION, we define the function inputs (PARAM) and outputs (RETURN). These are prefixed with an . Rstudio can ”autocomplete” the names for you, since the vocabulary is fixed.

```
#' @param r The radius of the circle to
#'   be plotted
#'
#' @return None The function is called
#'   solely for its side effect
```

Adding roxygen2 function comments (3)

Finally, we add some use examples, directions for what to "see also", and whether to export the function to the package namespace. (note: "examples", not "example"!)

```
#' @examples
#' plotCircle(10)
#'
#' @seealso The general
#'   \code{\link[graphics]{plot}} function
#'
#' @export
plotCircle <- function(...)
```

Also note: indenting per above is required.

invoke document()

Once the function (with roxygen2 comments) is saved in R/, invoking

```
document()
```

will parse the information to

add man/plotCircle.Rd (properly formatted!)
edit the NAMESPACE

We've now documented the function. What about the package?

What do we need to say about a *Package?*

What it is (TITLE)
What it's for (DESCRIPTION)
Other verbiage, such as what functions it contains

We'll insert this information as roxygen2 comments in the toyPackage.R file Rstudio created.

But what are we commenting on?

The package.R file

```
#' A toy package
#'
#' This does nothing. I mean nothing.
#' What're you looking at?
#'
#' We can put more verbiage here if we want.
#'
#' @docType package
#' @name toyPackage
NULL
```

We're commenting on NULL, or rather using this to anchor our text.

document(), and clear debris

Now that we've created and saved toyPackage.R, we run

```
document()
```

again. This time, it produces

man/toyPackage.Rd

What about man/toyPackage-package.Rd? (in v0.98!) *Delete it.* It isn't formatted to compile correctly.

Now, we move on to the DESCRIPTION.

What do we need to DESCRIBE a package?

Who wrote it, and how to contact them
Who maintains it, and how to contact them
What other packages it builds on
What license it uses

Much of this is loosely outlined in the DESCRIPTION file Rstudio has pre-assembled.

What needs changing

```
Package: toyPackage
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2015-06-29
Author: Who wrote it
Maintainer: Who to complain to
  <yourfault@somewhere.net>
Description: More about what it does (maybe
  more than one line)
License: What license is it under?
```

What needs changing (changed)

```
Package: toyPackage
Type: Package
Title: Illustrates a 1 function package
Version: 1.0
Date: 2015-06-29
Author: Keith Baggerly
Maintainer: Keith Baggerly
  <kabagg@mdanderson.org>
Description: Nothing. Really, nothing.
  The 2nd line keeps code from complaining.
License: MIT + file LICENSE
```

Only the last line above really needs expansion

Licenses, Tigers and Bears

Your work is your property, by default.

Specifying the license tells others what they have your permission to do with your work. Fuller descriptions are here: R licenses

The “MIT” license says (essentially) “do what you like, but don’t blame me” (I think this is a Bromanism) and is the most open-ended. It requires us to create a separate LICENSE file with two lines:

```
YEAR: 2015
COPYRIGHT HOLDER: Keith Baggerly
```

Dependencies

Our toy example doesn't need it, but some functions presume other packages have already been loaded or they won't work. At this point, we can list which packages are "Imports" (required) or "Suggests", possibly with a version number.

```
use_package("myPackage",
            "Suggests")
```

will update the DESCRIPTION file properly.

.Rbuildignore

Occasionally, as you're working to develop a package, it may be convenient to have some intermediate files in your working copy that aren't distributed.

Latex, for example, starts with a .tex file and may produce .aux and .dvi files en route to a .pdf file; and we don't want to send around the .aux and .dvi files.

We do this by adding wildcard strings to .Rbuildignore, e.g.,

```
*.aux
*.dvi
```

What .Rbuildignore has by default

```
^.*\.Rproj$
^\.Rproj\.user$
```

You can add other stuff either by editing .Rbuildignore directly or by invoking

```
use_build_ignore("fileOrFolderName")
```

which will handle formatting for you.

It's almost time...

First, delete "Read-and-delete-me"
(v0.98, after reading, if you like)

Then, invoke

```
document()
build()
install()
check()
```

and (hopefully) we have a valid package!

Hopefully?

When you run through the above commands to produce a package, you will probably encounter some lurid red warning messages.

Most of these will suggest things that still need tiny bits of tweaking. Some you shouldn't worry about, e.g.

building toyPackage_1.0.tar.gz
Warning: invalid uid value replaced by that for user 'nobody'
Warning: invalid gid value replaced by that for user 'nobody'

Mimicking External Commands

The above commands within R are actually testing out commands generally used on the command line, specifically

R CMD BUILD myPackage
R CMD INSTALL myPackage
R CMD CHECK myPackage

This last is typically the most stringent, and applies many specific tests CRAN uses before accepting packages for distribution. If you've already built the package, you can CHECK myPackage_version.tar.gz.

Other Stuff to Add

What we've got is indeed a package.

That said, there are a few other basics you should be in the habit of (and/or know how to) include.

README.md
vignettes
data

README.md

Adding a README file to any package is a good idea.

This should say what the package is designed to do, and what the motivation was in building it.

I mostly write these as .Rmd files, but *GitHub* will automatically display README contents in html if you use straight markdown (.md) files instead.

Fortunately, this usage has been anticipated...

README Templates

```
use_readme_rmd()
```

Creates a template README.Rmd file *and automatically adds it to .Rbuildignore*.

Once you've included the relevant content, simply invoking

```
knit("README.Rmd")
```

produces the README.md file desired.

Adding a vignette

While we've documented the function(s) and the package at a fairly high level, people typically *will not use* a package if they don't understand details of how to apply it to tasks they have.

These lengthier descriptions should be given as *package vignettes*, which we can write as .Rmd reports.

These should describe, in both words and code, how to use the package to do something real.

Vignette setup

```
use_vignette("vignetteName")
```

will automatically add a template vignetteName.Rmd file to vignettes/ (and create the latter if it doesn't already exist).

After adding your content, calling

```
document()
build()
```

will build the vignette.

A Word of Caution

In general, vignettes shouldn't be huge with respect to

processing requirements,
memory requirements (e.g., 20Mb figures), or
final page length.

A similar point applies to the use examples supplied for individual functions.

Ideally, examples should run in a few seconds.

Adding Data

Every package should include the features discussed above.

Data is a bit more tricky, but including at least some can often be quite helpful, especially if the standard files you're analyzing have quirks you want to highlight.

Data can be included in either raw (e.g., .csv, .bam) or processed (.RData) forms, but keep it small ($< 1Mb$) if this is for wide sharing.

Process Raw Data

Raw data, in whatever form, should be put in

/inst/extdata

and one of the first uses of a vignette should be to show how to map the raw data to its final processed form.

Opinion: Don't include raw data without including the processed version as well.

```
system.file("extdata", "myRawData.bam",
            package="toyPackage")
```

will point you to the data location.

Including Processed Data

Once you have object(s) in R,

```
use_data(myObject, myObject2, pkg="toyPackage",
         compress="gzip")
```

will create .rda file(s) and store it in /data

Each object should be *documented* (in R/) as (e.g.) myObject.R

Documenting Processed Data

```
#' A bunch of radii (TITLE)
#'
#' Really, do you need this explained?
#' Well, ok. (DESCRIPTION)
#'
#' @format description of the variables,
#'   with full names and explanations (esp
#'   for data frames). Dimensions help.
#'
#' @source where did the data come from?
#'   This could be a URL.
#' @name myObject
NULL
```

That Looked Familiar...

As with package documentation, we use NULL to anchor the text for data documentation.

Much of the description of what a dataset is and how to use it should also be repeated in an associated vignette.

General Habits

When you're writing functions, start including roxygen2 comments *right then*.

Everything they're looking for is pretty basic, and forcing yourself to write this out will help others (and you) a great deal.

For one thing, you'll be clearer about your initial motivations in writing it!

A similar heuristic applies to writing a README when you first write a package.

Encapsulating Analysis Projects as Packages

Even for those of us who work with big data, we still generate reports from some datasets which might be 20K in size.

In those cases,

include both the raw and processed data, and
include both your processing and analysis as vignettes.

Then, the entire analysis can be distributed as a package, and you can know
if the package builds, your analysis can be done.

Things We've Glossed Over/Ignored

S3, S4, RC classes
Source functions from other languages (e.g., Rcpp) and compiled code
Use tests (a good idea! see the testthat package)
NAMESPACE explanations
Getting it ready for CRAN
Version Numbering

Most packages just use functions, text and data.

Where to find other examples

Aside from just poking around GitHub, here are a few:

Hadley Wickham's book on R Packages is on github, but you can also buy it
The documentation vignettes for roxygen2
Hadley's GitHub Page
Alyssa Frazee's RSkittleBrewer
Karl Broman's Tools for RR course

Onwards!