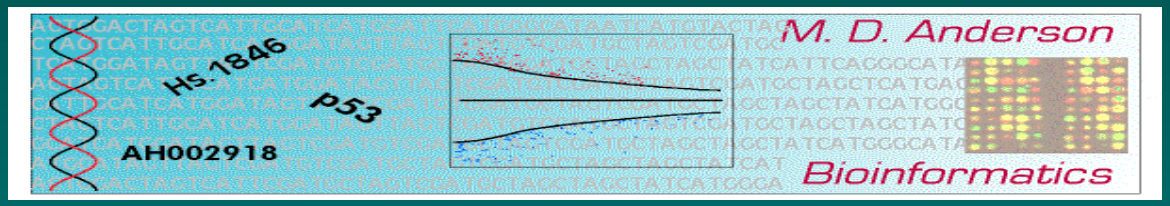
Reproducible Research with R and RStudio C12: R Packages

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

Moffitt, May 31, 2018



This is Not a Lecture

well, ok, it is a bit.

Since we're hoping to teach you to use a tool, a big part of this will involve live demo, and letting you follow along.

Thus, these notes are at least as much for
reference documentation
as for presentation.

We'll also point out the very nice
Cheatsheet from RStudio

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 was assembling packages hard?

They have specific structure and documentation requirements.

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

Getting 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

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

These lower the barriers to entry!

I'm assuming these have been loaded going forward.

In the Beginning...

there was creation.

One way we can start assembling a package involves shifting to the folder will work in, and invoking

```
> devtools::create("toyPackage")
```

This creates a new folder (toyPackage) populated with some of the files every package needs.

Equivalently (and more commonly what I do), we can start a New Project in RStudio, and tell it that the Project is to be an R Package.

What's There to Start With?

(using the latest versions of everything)

```
.Rbuildignore
DESCRIPTION
NAMESPACE
R/hello.R
man/hello.Rd
packagename.Rproj
```

Not a heck of a lot, and fortunately
we'll leave most of these alone.

What we will edit (first) is 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

This is outlined in the DESCRIPTION file Rstudio pre-assembled.

What needs changing

```
Package: packagename
Type: Package
Title: What the Package Does (one line,
  title case)
Version: 0.1.0
Author: Keith Baggerly
Maintainer: Keith Baggerly <kabagg@gmail.com>
Description: What the package does (one
  paragraph).
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

What needs changing (changed)

```
Package: packagename
Type: Package
Title: Project Setup Utilities
Version: 0.1.0
Author: Keith Baggerly
Maintainer: Keith Baggerly <kabagg@gmail.com>
Description: Encapsulates common infrastructure
  tasks, such as setting up a default directory
  structure, formatting Rmd files, etc.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
```

Only the “License” 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

Per the Rstudio cheatsheet,
CC0 (Creative Commons) is extremely broad; other commonly used ones are
MIT and
GPL (please see cheatsheet).

Karl Broman's Comments

Use CC licenses for your lecture notes, slides, and articles, but not for your software.

CC0 (public domain) seems appropriate for software: you're just saying that anyone can do anything with the code.

But in some states (e.g., Maryland, I think), software is treated as a "good" (like a car), and so if your code causes something terrible to happen, you could be sued for damages. Using a lenient license, like the MIT license, eliminates that potential problem through the "no warranty" clause.

So, use CC0 for your lecture notes, slides, and web sites, but use a lenient license, like the MIT license, for your software.

So, What Goes in LICENSE?

Just two lines:

YEAR: 2018
COPYRIGHT HOLDER: Your Name

Note: the file name should be LICENSE, not LICENSE.txt, or any other suffix.

That's it.

When we Change Things...

by, for example, editing an R script, we invoke

```
document() ## updates NAMESPACE, creates man/,
           ## edits DESCRIPTION
build()
install()
check()
.rs.restartR()
library(toyPackage)
```

Guess what, we're there.

It's useless, but we're there.

Mimicking External Commands

The 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.

Documenting a Package

The first thing we want to add is more text documenting what we want the package to do.

To do this, we're going to create a new R script in R/, "toyPackage.R", which will consist almost entirely of comments for roxygen2.

Almost every script needs to state

What it is (the TITLE)
What it's for (the DESCRIPTION)

Let's take a look at such a script...

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
```

Note the NULL; we're using this to anchor our text.

Rd files

Now, invoking

```
document()
```

generates an R documentation (.Rd) file in man/.

The syntax of Rd files is similar to that of LaTeX, but
we're not going to write Rd files by hand.

We'll let the packages do that so everything lines up correctly.

Cap'n, there be text here!

Once we cycle through build/install/check and rs.restartR/library, invoking

```
?toyPackage
```

will now render the Rd file to show the documentation (and a link to the package Index).

Similarly, we can add functions that do things...

A Trivial Function

```
#' A Trivial Function
#'
#' Just illustration; nothing really here
#'
#' @param x,y two values to be summed
#' @return the sum x+y
#' @examples
#' trivial(2,3)
#' @export
trivial <- function(x,y){ x + y }
```

Other Stuff to Add

What we've got is indeed a package.

That said, there are a few other basics you should (and know how to) include.

README.md
vignettes
data

Division of labor

```
use_readme_rmd()
use_vignettes()
use_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 write READMEs as .Rmd files, but GitHub will automatically render straight markdown README.md contents in html.

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

We've documented the package at a high level, but
people will not use a package they can't figure out.
We need to give details of how to apply the package to tasks they have.

These lengthier descriptions should be given as 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_vignettes()
build()
```

will build the vignette.

Adding Data

Every package should include the features discussed above.

Data is more tricky, but including some can be quite helpful, especially if the 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.

We can find the data with system.file, e.g.

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

Including Processed Data

Once you have object(s) in R,

```
use_data(myObject, pkg="toyPackage")
```

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

Each data 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
```

Once More, With Feeling...

Objects in `data/` are always effectively exported (they use a slightly different mechanism than `NAMESPACE`'s but the details are not important). This means that they must be documented. Documenting data is like documenting a function with a few minor differences. Instead of documenting the data directly, you document the name of the dataset and save it in `R/`. For example, the roxygen2 block used to document the diamonds data in ggplot2 is saved as `R/data.R` and looks something like this:

```
#' Prices of 50,000 round cut diamonds.
#'
#' A dataset containing the prices and other attributes of almost 54,000
#' diamonds.
#'
#' @format A data frame with 53940 rows and 10 variables:
#' \describe{
#'   \item{price}{price, in US dollars}
#'   \item{carat}{weight of the diamond, in carats}
#'   ...
#' }
#' @source \url{http://www.diamondse.info/}
"diamonds"
```

http://r-pkgs.had.co.nz/data.html Updated...

That Looked Familiar...

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

A Word of Caution

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

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

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

Ideally, examples should run in a few seconds.

Keep packages small and nimble.

Walking Through Another Package

There's a good basic package from Alyssa Frazee here

Alyssa Frazee's RSkittleBrewer

and there's an associated BLOG POST with more details

DESCRIPTION, NAMESPACE, functions, documentation...

does all of this look familiar?

Some Thoughts on Coding

The code should work - sanity checks

The code should be readable - style, naming, formatting

The code should be reusable - generalizing beyond the immediate case at hand

The code should be reproducible - will you get the same results 5 minutes from now?

Think before you code - hard when people want it NOW.

Learn from others' code (leveraging GitHub!)

Don't Assume

Please avoid referring to other objects in your workspace. Pass what's needed as an argument.

Name your arguments and functions. Don't be cute. "tmp1, tmp2, x1, x2, x3"? Having a few words can clarify things.

Text autocompletion can be your friend at times!

Name your constants / avoid magic numbers

```
foo(12) ## bad
nMonths <- 12
foo(nMonths) ## good
```

Legibility

INDENT YOUR CODE.

There are RStudio shortcuts for many common tasks; for indentation it's CTRL-I or CMD-I depending on your platform

Use white space. This makes it easier for human beings, not the computer.

Avoid long lines (wrap them).
The formatR package can help here

General Habits

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

Everything packages need 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.

Packages Can Automate Other Tasks

Is there a common report structure you like to use?

You can include template reports and the like using

`inst/rmarkdown/templates/my_template_folder`

and including, in each such folder

```
template.yaml
skeleton/skeleton.Rmd
```

Start with examples from other packages

Invoking Automation

Positioning Folders

New RMarkdown Document - from template!

Autocreate: make_all, make_clean

Can you automate numbering?