Lab 6: The Complete Package

Gregor Thomas

Monday, February 15, 2016

Set-up

Set your working directory (if necessary) and load helper packages.

```
library(knitr)
# I want to *not* run any code when compiling this document
opts_chunk$set(eval = FALSE)
# Setting working directory isn't necessary if
# you use RStudio "projects" - they take care of it
setwd("C:/Dropbox/statr/statr")
library(devtools)
```

Create the package framework

```
create(path = "demo2")
```

Now you should have a directory, `demo2`, with `R` subdirectory, NAMESPACE and DESCRIPTION files.

```
setwd("./demo2")
# move working directory to the package
# (or just use rproject of the package)
```

Add a function

Let's add the `samp` function we used before. Put all of the code in an R script save it in `demo2/R` as `samp.R`.

```
#' Quick random sample of a data frame
#'
#' \code{samp} can be used to quickly check on a data frame.
#' @param x data frame to sample from
#' @param n positive integer number of rows to return in the sample
#' @return Returns a data frame
#' @examples
#' samp(mtcars)
#' @export
samp <- function(x, n = 10) {
    if (n >= nrow(x)) return(x)
    return(x[sample(1:nrow(x), size = n), ])
}
```

Create package-level documentation

Package-level documentation is the help file if you do `?package_name`. It's not a strict requirement for packages in general, but it is nice to include. **It is a requirement for your final project.**

To have Roxygen make the package-level docs, we need a `.R` file in the `R` folder with the appropriate Roxygen comment blocks. Either you can manually create a file, or you can use `devtools::use_package_doc("demo2")` to create a file `demo2-package.r` and just make edits.

```
use_package_doc()
```

Regardless of how you create the file, edit it so that it's contents are:

```
#' demo2.
#'
#' This is a tiny little package to illustrate the
#' package building process.
#'
#' @name demo2
#' @docType package
NULL
```

A valid line of code **must** follow a Roxygen comment. For code-less things like data and package-level documentation, the recommendation used to be to put `NULL` for the line of code.

More modern documents recommend strings, `"_PACKAGE"` for package-level documentation, the name of the data set, e.g., `"mtcars"` for data documentation.

Adding data

To add data to a package, you use a similar function `use_data()`, where you specify the object(s) you want to save. They must be present in your working environment.

```
use_data(mtcars)
```

Or you can create a `data/` folder and save files with `save` and either `.rda` or `.RData` extensions.

Data documentation

Data documentation is similar to package documentation: it needs to go in a `.R` file in the `R/` folder. You can tell Roxygen that it is data documentation by including the line `#' @docType data` with the `NULL` after the comment block.

Or, if you have the string name of the data set after the block, Roxygen should figure out that it's data documentation.

Add a package vignette

Once again, `devtools` to the rescue!

```
use_vignette(name = "my_vignette", pkg = "demo2")
```

This is a convenience function that creates a `vignettes/` directory and givees the start of an Rmarkdwon vignette. I recommend using the function for this one as the template is quite nice. It also edits the DESCRIPTION file to suggest `knitr` and specify `knitr` and the vignette-building engine.

Edit the title of the vignette appropriately.

Using your package in your vignette

There's a bit of a cart-before-the-horse problem with working on the vignette in an early package draft. Your vignette is almost identical to a normal `knitr` document. This means to use your packge in your vignette you still need to `library(your_package)`!

First you need an installed version of your package when you use the vignette.

Though, even this `devtools` will take care for you if you `build()` or `check()` your package.

DESCRIPTION

Speaking of the description file, let's add that.

I'd recommend consulting the DESCRIPTION section of Writing R Extensions to see the allowed values for things like the LICENSE field. For now, edit the DESCRIPTION so that it looks something like this:

```
Package: demo2
Title: Complete package demp
Version: 0.1
Authors@R: "Gregor Thomas <gregorp@uw.edu> [aut, cre]"
Description: This package demos building a package with vignettte.
Depends: R (>= 3.1.2)
License: GPL-3
LazyData: true
Suggests: knitr
VignetteBuilder: knitr
```

Adding dependencies

If your package uses functions from any other (non-base) packages, they need to be included in the description file to make sure they are installed, and to specify *exactly* what happens when your package is loaded.

There are 3 main choices:

- **Depends** this is the strongest. If `demo2` Depends on, say `dplyr`, then it will be impossible for a user to load `demo2` without also loading `dplyr`. You impose this decision on the end-user, so it **should be used sparingly**.
- **Imports** is the standard choice. If you use this option, installation of `demo2` will make sure that any `Imports` packages are also installed, so they are present in the system, however is *does not load them*, leaving the user's search path uncluttered. With this option, you will need to call functions explicitly using `::` for your internal functions. E.g., if you use `summarize`, you will need to write `dplyr::summarize`.

- **Suggests** is a weak choice. It will not require that the other package is present, so it cannot be relied on. The typical use of `Suggests` is if you have examples that can only be run if the suggested package is present. Users who don't need the examples, then, won't be forced to install the extra packages.

Let's add an Imports dependency for `dplyr`:

```
use_package(package = "dplyr", type = "Imports", pkg = "demo2")
```

This should add a line to the description file.

Checking

Make sure all the files you have edited are saved, then

```
check("demo2") # by default, this will call roxygenize first
```

No errors is necessary, no warnings is nice, and notes are okay. I have 2 notes, one is that I'm immporting `dplyr` but not using it at all, and the other that there's a "non-standard file", which is a .Rproj file created by default.

A common warning is having undocumented objects (if you saved data but didn't document it, or have exported but undocumented functions). The solution is to add documentation.

Another common warning is having an "object with no visible binding". That means your code refers to an object, but the definition of the object can't be found anywhere. All variables shuold be either part of the included data, passed in as a function argument, or created from the above.

Building and installing

To test installing your package use

```
devtools::install("demo2", build_vignettes = T)
```

You can check everything.

```
samp(mtcars)
?samp
?demo2
vignette(package = "demo2")
vignette("my_vignette")
```

`load_all("demo2")` approximates this, but it won't, e.g., build the vignette. If you're *just* focusing on the vignette, you can use `build_vignette()` to update it without doing everything else (in your local copy, not the installed copy). When things are completed to your satisfaction,

```
devtools::build("demo2")
```

will create a compressed `.tar.gz` file (with built vignette) that can be distributed and installed.

At this point, if you package passes `check()`, you could submit it to CRAN. Devtools can even help you do this, with `release()`, which, after asking you a few confirmatory questions, will upload the package to CRAN and draft you an email to notify them of the incoming package. Along with the first submitted version of a package, you must also explicitly agree to CRAN's terms of use.