Writing R packages

Tools for Reproducible Research

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

biostat.wisc.edu/~kbroman
github.com/kbroman
@kwbroman
Course web: bit.ly/tools4rr

R packages and the Comprehensive R Archive Network (CRAN) are important features of R. R packages provide a simple way to distribute R code and documentation. And they really are rather simple to create.

Write an R package to keep track of the misc. R functions that you write and reuse. Write an R package to distribute the data and software that accompany a paper.

The most painful part of writing an R package is the construction of the documentation files, which are in a special `.Rd` format. But the Roxygen2 tool makes this rather easy: you write comments preceding each R function, in a specially structured way, and then use the Roxygen2 tool to create the `.Rd` files for you.

Why write an R package?

- To distribute R code and documentation
- To keep track of the misc. R functions you write and reuse
- To distribute data and software accompanying a paper.

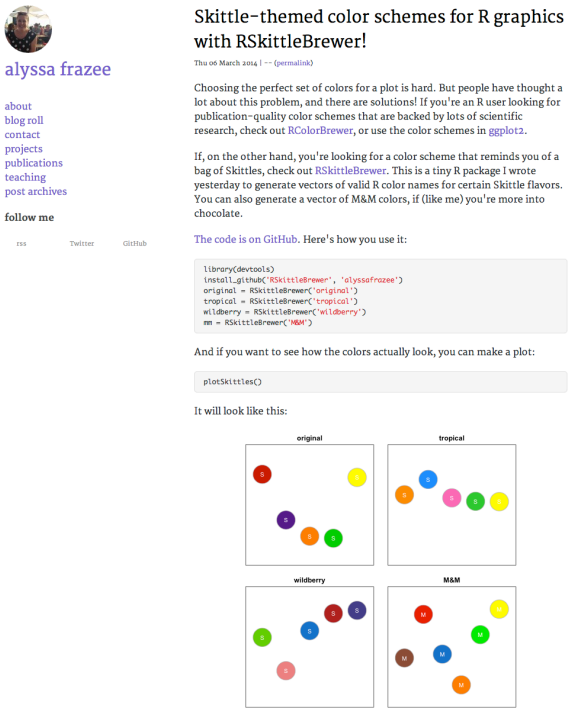
R packages can be big and important.

But that shouldn't scare you off. Assembling a few R functions within a package will make it vastly easier for you to use them regularly. You don't even need to distribute the package.

And really, the R package system is an incredibly important feature of R. Packages on CRAN are basically guaranteed to be installable, as they are regularly built, installed, and tested on multiple systems.

The Writing R Extensions manual is the key source for the specifications of R packages. It's rough going in parts, but if you want to get a package on CRAN, you should read it.

A simple example: RSkittleBrewer



alyssa frazee

about
blog roll
contact
projects
publications
teaching
post archives

follow me

rss Twitter GitHub

Skittle-themed color schemes for R graphics with RSkittleBrewer!

Thu 06 March 2014 | -- (permalink)

Choosing the perfect set of colors for a plot is hard. But people have thought a lot about this problem, and there are solutions! If you're an R user looking for publication-quality color schemes that are backed by lots of scientific research, check out RColorBrewer, or use the color schemes in ggplot2.

If, on the other hand, you're looking for a color scheme that reminds you of a bag of Skittles, check out RSkittleBrewer. This is a tiny R package I wrote yesterday to generate vectors of valid R color names for certain Skittle flavors. You can also generate a vector of M&M colors, if (like me) you're more into chocolate.

The code is on GitHub. Here's how you use it:

```
library(devtools)
install_github('RSkittleBrewer', 'alyssafrazee')
original = RSkittleBrewer('original')
tropical = RSkittleBrewer('tropical')
wildberry = RSkittleBrewer('wildberry')
mm = RSkittleBrewer('M&M')
```

And if you want to see how the colors actually look, you can make a plot:

```
plotSkittles()
```

It will look like this:

alyssafrazee.com/RSkittleBrewer.html

I was going to write a short example R package, but Alyssa Frazee saved me the effort. Her package is a perfect little example to illustrate how to write a package.

It's also a great example of a small but really useful package. One small function could be widely useful; you just need to package it and tell people about it.

R package contents

```
RSkittleBrewer/

    DESCRIPTION
    NAMESPACE

    R/RSkittleBrewer.R
    R/plotSkittles.R
    R/plotSmarties.R

    man/RSkittleBrewer.Rd
    man/plotSkittles.Rd
    man/plotSmarties.Rd
```


In the simplest form, an R package is a directory containing: a `DESCRIPTION` file (describing the package), a `NAMESPACE` file (indicating which functions are available to users), an `R/` directory containing R code in `.R` files, and a `man/` directory containing documentation in `.Rd` files.

DESCRIPTION file

```
Package: RSkittleBrewer
Version: 1.1
Author: Alyssa Frazee
Maintainer: Alyssa Frazee <afrazee@jhsph.edu>
License: MIT + file LICENSE
Title: fun with R colors
Description: for those times you want to make plots with...
URL: https://github.com/alyssafrazee/RSkittleBrewer
```


The `DESCRIPTION` file is pretty self-explanatory. It just contains basic information about the package and its author.

The simplest way to create this sort of file is to copy and edit one from some other package.

The only part that might be unclear is the `License` field. You need to choose a license. We'll talk about this on the very last day of the course.

For now, I'd suggest choosing between the GPL-3 (the GNU Public License v3) and MIT licenses. GPL-3 has a "pass-through" provision: software that incorporates GPL-3 code must also be licensed as GPL-3. This is a good but restrictive thing. The MIT license is the most bare-bones license possible: it basically just says "Do what you want, but don't blame me."

An R package with the MIT license needs to also include a `LICENSE` file or R will complain; copy and edit the one from the RSkittleBrewer package.

NAMESPACE file

```
export(RSkittleBrewer)
export(plotSkittles)
export(plotSmarties)
```


The NAMESPACE file is a bit technical: it tells R what functions that will be accessible to users.

The point of this is to keep different packages from stepping on each others' toes.

An .Rd file

```
\name{RSkittleBrewer}
\alias{RSkittleBrewer}
\title{Candy-based color palettes}
\description{Vectors of colors corresponding to different
             candies.}
\usage{RSkittleBrewer(flavor = c("original", "tropical",
                      "wildberry", "M&M", "smarties"))
}
\arguments{
  \item{flavor}{Character string for candy-based color
  palette.}
}
\value{Vector of character strings representing the chosen
       set of colors.}
\examples{
plotSkittles()
plotSmarties()
}
\keyword{hplot}
\seealso{ \code{\link{plotSkittles}},
          \code{\link{plotSmarties}} }
```


The R documentation format is very LaTeX-like.

It describes what the function does, what its arguments are, and what output it produces.

You can further provide examples (which can also serve as tests) and links to related functions. The examples need to be fast ($\ll 5$ sec), because they're run frequently. (CRAN checks every package every day on multiple systems.)

Writing these help files is tedious! That's where Roxygen2 comes in.

Building, installing, and checking

```
R CMD build RSkittleBrewer
R CMD INSTALL RSkittleBrewer_1.1.tar.gz
R CMD check RSkittleBrewer_1.1.tar.gz

R CMD check --as-cran RSkittleBrewer_1.1.tar.gz

R CMD INSTALL --library=~/Rlibs RSkittleBrewer_1.1.tar.gz
# (~/.Renviron file contains R_LIBS=~/Rlibs)

# On windows:
R CMD INSTALL --build RSkittleBrewer_1.1.tar.gz
```

```
# also consider (within R):
library(devtools)
build("/path/to/RSkittleBrewer")
build("/path/to/RSkittleBrewer", binary=TRUE)
```


To install your package, you first need to build it. `R CMD build` creates the `.tar.gz` source file that you'd distribute.

You then use `R CMD INSTALL` to install the package. You may want to use `--library=/some/dir` to install to a different directory, in which case you need to set `R_LIBS` in your `~/.Renviron` file.

`R CMD check` runs extensive checks of the package, including running all of the examples in the help files. Pay attention to any errors, warnings, or notes, and revise the package to avoid them. This is particularly true if you want to submit the package to CRAN.

Before submitting to CRAN, be sure to run `R CMD check` with the lesser-known `--as-cran` flag, which checks further things.

On Windows, use `R CMD INSTALL --build` to create a `.zip` file of the "compiled" package. You can also use the `build()` function from the devtools package.

Roxygen2 comments

```
#  RSkittleBrewer
#' Candy-based color palettes
#'
#' Vectors of colors corresponding to different candies.
#'
#' @param flavor Character string for candy-based color palette.
#'
#' @export
#' @return Vector of character strings representing the chosen...
#'
#' @examples
#' plotSkittles()
#' plotSmarties()
#'
#' @seealso \code{\link{plotSkittles}},
#'   \code{\link{plotSmarties}}
#' @keywords hplot
RSkittleBrewer <-
...
```


The `.Rd` files are a pain to create and maintain. Plus, you'll end up writing documentation in two places: in the R code, and then in the separate `.Rd` files.

Roxygen2 is a system for writing structured comments in the R code that get converted to `.Rd` files. You just maintain the documentation in one place: in the R code. The Roxygen comments start with `#'`. The first line is the title; the second is the description.

The individual `\item`'s within `arguments` become `@param`. The `\value` field becomes `@return`.

You still need to know a bit about the `.Rd` format; for example, the `\code{\link{blah}}`.

Roxygen2 will across create the `NAMESPACE` file. That's what `@export` is for.

You may also be interested in the Rd2roxygen package, for converting a package with hand-written `.Rd` files to the Roxygen2 system.

Makefile

```
# build package documentation
doc:
  R -e 'library(devtools);document(roclets=c("namespace", "rd"))'
```


Here's the `Makefile` I use to build the `.Rd` files: I use the `document()` function within the `devtools` package.

.Rbuildignore

```
Makefile
```


You'll want to include a `.Rbuildignore` file in your package directory, containing the single line "`Makefile`". This tells R about files to not include in the `.tar.gz` file it builds. You'd get complaints from `R CMD check` otherwise.

Include a `README` or `README.md` file

````
fun with R Colors
=================

If you want high-quality, scientifically-researched color
schemes for your R plots, check out
[RColorBrewer](http://cran.r-project.org/web/packages/RColorBrewer).
If you want your plots to be colored the same way as packs of
Skittles (or M&Ms), then this package (RSkittleBrewer) is the
way to go.

### install
with `devtools`:

```S
devtools::install_github('RSkittleBrewer', 'alyssafrazee')
```

### use
There are only three functions in this package.

Call `RSkittleBrewer` on a flavor to get a vector of R color
names that correspond to that Skittle flavor.
...
````


Include a `README` or `README.md` file; R will ignore it, but it will show up on GitHub.

You can use `ReadMe` or `ReadMe.md`, but then you need to include it in the `.Rbuildignore` file or R will complain.

That's it!

That's all you need to make a package.

There's more, but you should start with just that stuff.

Package vignettes

- Include *vignettes* to demonstrate the use of your package.
- It's simplest to use R Markdown.
  - Create a `vignettes/` subdirectory.
  - Place a `.Rmd` file there.
  - The name of the file becomes the name of the vignette.
- Include the following at the top of the `.Rmd` file

```
%\VignetteEngine{knitr::knitr}
%\VignetteIndexEntry{Intro to RSkittleBrewer}
```

- Load the package in an initial chunk

```
library(RSkittleBrewer)
```


In my experience, people tend not to read detailed documentation, but they like tutorials. This is what vignettes are for.

It's easiest to write vignettes in R Markdown (allowed in R 3.0 onwards). You create a subdirectory `vignettes/` and put the usual sort of `.Rmd` files there. You just need to add a couple of special lines to the top of the file: to tell R to use knitr to build the vignette, and to give a title to the vignette.

In your `.Rmd` file, you'll need to load the package before you start using it.

Package vignettes

- Add the following lines to your `DESCRIPTION` file.

```
VignetteBuilder: knitr
Suggests: knitr
```

- The following lists the vignettes for a package and then opens a selected vignette.

```
library(RSkittleBrewer)
vignette(package="RSkittleBrewer")
vignette("RSkittleBrewer", "RSkittleBrewer")
```


There's one other little detail: you need to mention knitr as the vignette builder in your `DESCRIPTION` file.

If you submit the package to CRAN, links to the vignettes will be listed on web page for the package. You might also refer to them in your `README` file.

Within R, you can use the `vignette` function to view the available vignettes for a package and to open a particular vignette. (With R Markdown vignettes, the compiled html file will be opened in your web browser.)

Optional stuff

- `NEWS` file describing changes in each version of the package.
- `inst/CITATION` file describing how to cite your package.
- `inst/doc/` directory any sort of misc. documentation (e.g., pre-compiled computationally heavy vignettes)
- `data/` directory containing data
- `src/` directory containing C/C++/Fortran code
- `demo/` directory with demonstrations (like vignettes, but to be executed in real-time).
- `tests/` and/or `inst/tests/` containing tests.

We could go on and on about packages. Here are some of the key additional things to consider adding to your package.

The `inst/` subdirectory is ignored by R but its contents get moved back to the root directory for the package when installed. If you want your `README` file to be named `ReadMe`, you'd put it here.

It's useful to include example data sets. Your package could contain only data! You should document them. See
`http://stackoverflow.com/questions/2310409`

devtools

Get to know the devtools package.

- `dev_mode()`
- `load_all()`
- `install_github()`, `install_bitbucket`, ...
- `document()`
- `build()`
- `check()`
- `check_doc()`
- `run_examples()`
- `test()` (next time)

I've tried to characterize package development as super simple, but all coding involves considerable testing, debugging, and exploration, and re-building and re-installing R packages can be tedious.

The devtools package reduces a lot of the hassle. Try it out!

`dev_mode()` puts you in "development mode" so that package installs won't affect your regular R package directory. `load_all()` does the equivalent of installing and reloading a package. (Otherwise, you might exit R, re-build the package, re-install, invoke R, and re-load the package.)

You can use `install_github()` to install a package directly from GitHub.

On Mac OSX Mavericks (10.9), gnutar is gone, and devtools gives an error. The simplest solution is to make a symbolic link from tar to gnutar:

```
sudo ln -s /usr/bin/tar /usr/bin/gnutar
```

Summary

- R packages really aren't that hard.
- R packages are really useful.
  - Distributing software and data
  - Organizing code for a paper
  - Organizing your misc. R functions
- Look at others' packages, and learn from them.
- Adopt the tools in the devtools package.

It's surprising to me just how many plain R scripts are sitting on people's web pages. Getting them into a package is really not that hard and would make the material much more useful.