Making R Packages

Stephanie Kirmer

Data Scientist, Uptake

www.stephaniekirmer.com

@data_stephanie

Prerequisites

1. I have written a function in R (or have a function script available).
2. I have used a package before.
3. I have RStudio installed on my computer.

That’s really it!

If you have any functions you have written previously, you can use these to follow along today. If not, you can clone mine from GitHub. https://github.com/skirmer/r_packages

Managing Scripts and Functions

We can do this…

Is that code in here?

No, maybe it's over there...

Darn. Maybe Jane knows where it is...

… or this!

```
git clone my_rad_package_repo

install.packages("~/my_rad_package_repo"
, repos = NULL, type = "source")

library(my_rad_package)
```

I know which option I prefer!

Package Building Steps

1. Load tools/resources you need
2. Initialize your package
3. Write documentation
4. Compile documentation
5. Install your package!

Other details we’ll discuss:

- Sharing your package
- Testing your package
- Writing more documentation
- Managing packages

Getting Started

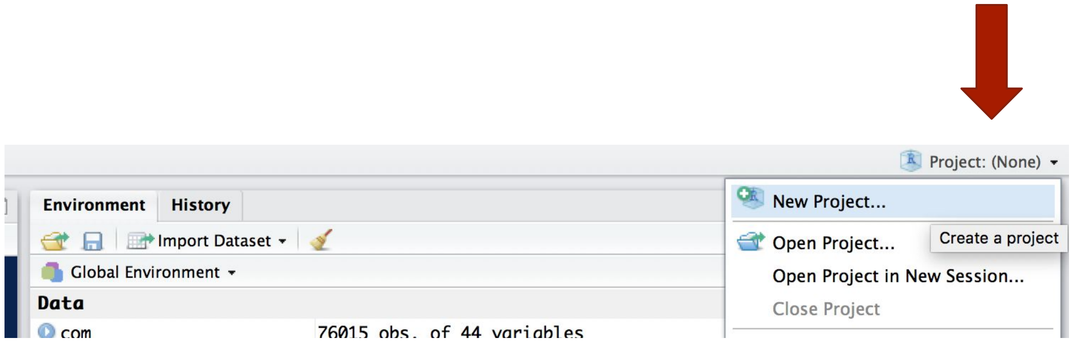
Install `devtools` and `roxygen2`

```
install.packages(“devtools”)
install.packages(“roxygen2”)

library(devtools)
library(roxygen2)
```

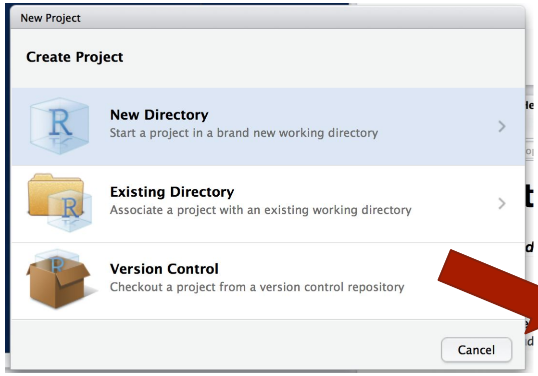
If you’re using my scripts, please also install `glue`, `feather`, and `dplyr` if you haven’t yet.

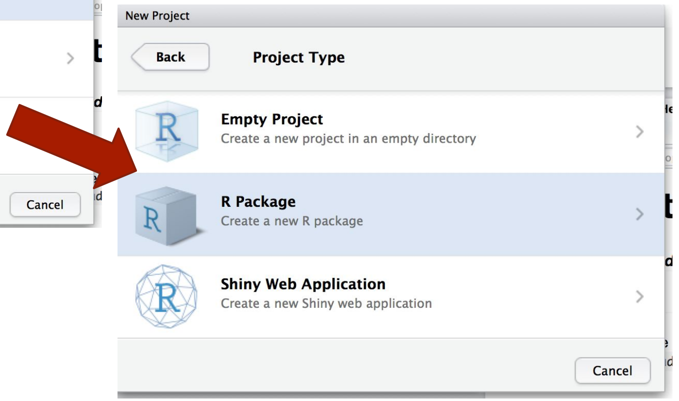
Create an R Project

Click “Project” and “New Project” in the top right corner of RStudio



Select “New Directory” > “R Package”


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

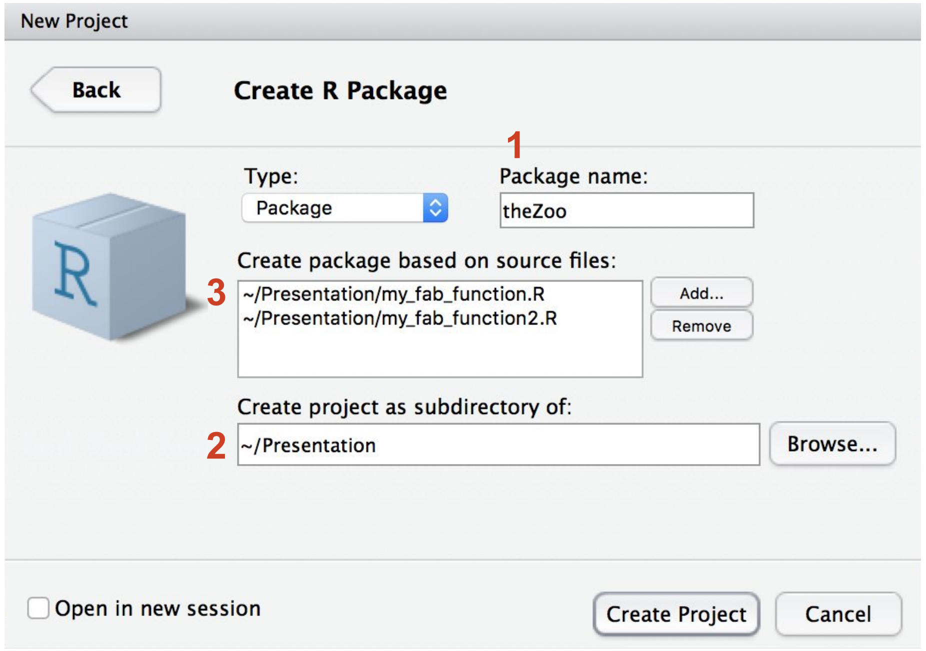


Set up your package

1. Give it a cool name! Camel case with initial lowercase is pretty common.

2. Make sure your package will be a subdirectory of the place where your scripts sit.

3. Put your files in there too. Include any data your package needs to run! (If you're using my scripts, include `zoo.feather`.)



Type: Package

Package name: theZoo

Create package based on source files:

~/Presentation/my_fab_function.R
~/Presentation/my_fab_function2.R

Create project as subdirectory of: ~/Presentation

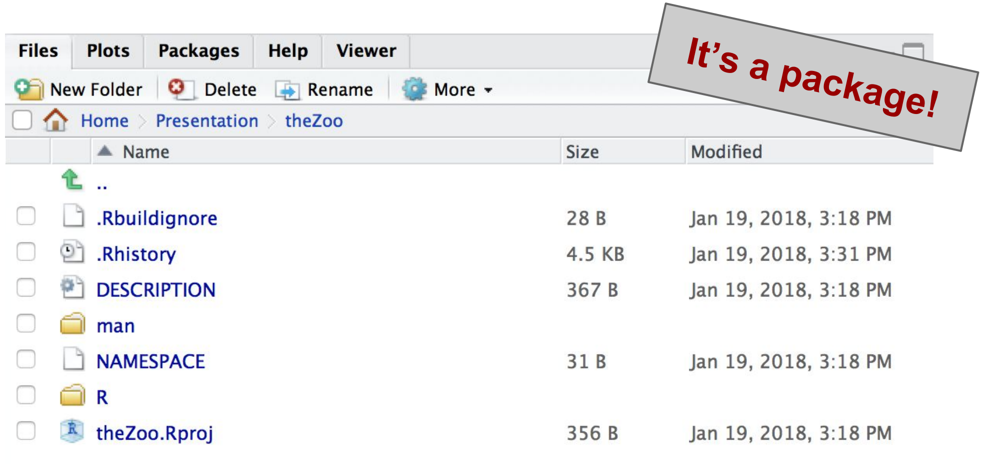
Create your package!

Now, your Files section will show you your functions, and a whole bunch more stuff! You have a package! But let's fix it up a tad.

1. Head back up to that top right corner, and hit the button again, but this time select "Project Options" > "Build Tools"
2. Select "Package" from the dropdown.

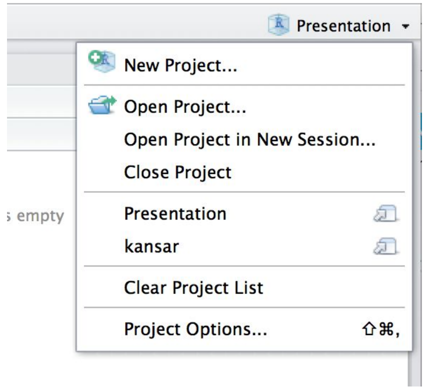
You're already where you want to be, folder wise, so you can accept the defaults. Just click the "Generate Documentation with Roxygen" box and hit "Vignettes" in the next window.

(This will become important later.)

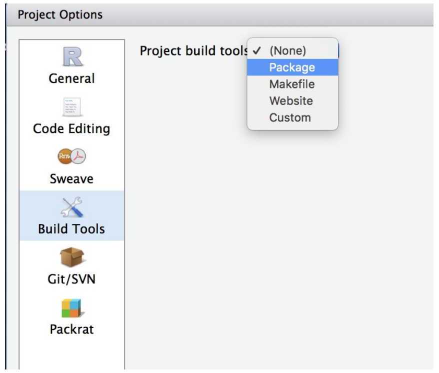


Name	Size	Modified
..		
.Rbuildignore	28 B	Jan 19, 2018, 3:18 PM
.Rhistory	4.5 KB	Jan 19, 2018, 3:31 PM
DESCRIPTION	367 B	Jan 19, 2018, 3:18 PM
man		
NAMESPACE	31 B	Jan 19, 2018, 3:18 PM
R		
theZoo.Rproj	356 B	Jan 19, 2018, 3:18 PM

1

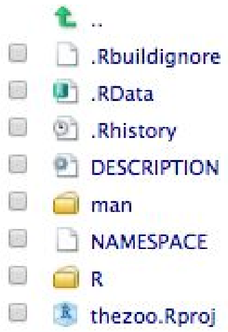


2



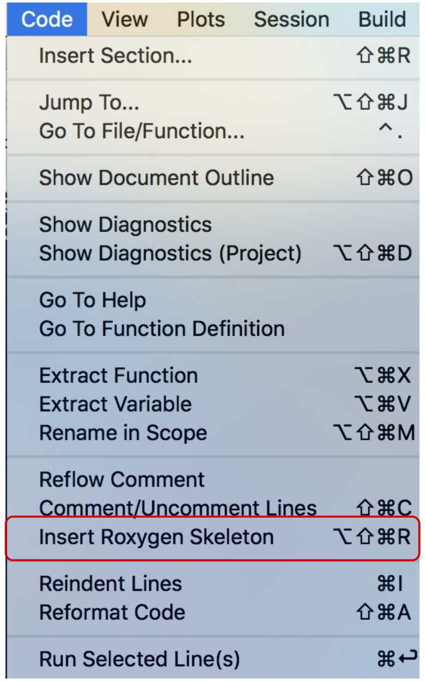
Document your Functions

Open one of your function scripts, inside the package subfolder “R”.



Place your cursor inside the function’s outer { } (curly braces).

Select “Code” from the menu and hit “Insert Roxygen Skeleton”.



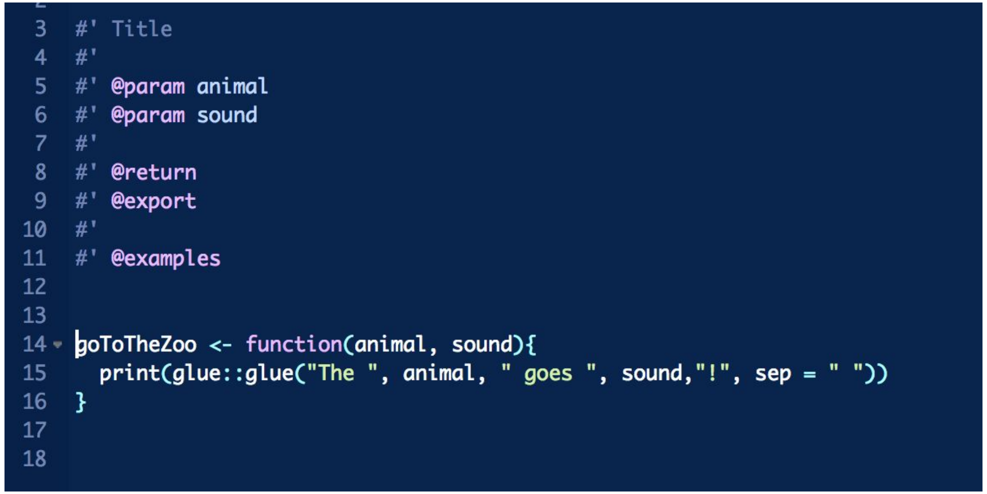
Document your Functions

Now, you need to fill out your function’s documentation!

- give it a title
- describe the meaning of the inputs it takes
- write an example of a correct usage
- what does it return to the user?

There are lots of other things you can add.

Look at the docs for other people’s functions that you use a lot to get ideas!



```r
#' Title
#'
#' @param animal
#' @param sound
#'
#' @return
#' @export
#'
#' @examples


goToTheZoo <- function(animal, sound){
  print(glue::glue("The ", animal, " goes ", sound,"!", sep = " "))
}


```

Document your Functions

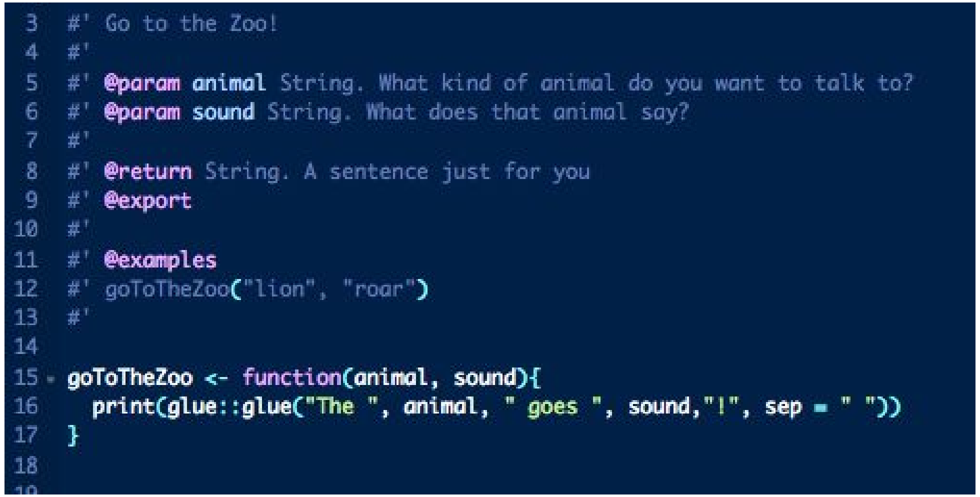
Now, you need to fill out your function’s documentation!

- give it a title
- describe the meaning of the inputs it takes
- write an example of a correct usage
- what does it return to the user?

There are lots of other things you can add.

Look at the docs for other people’s functions that you use a lot to get ideas!

Replace the defaults, and write in the details

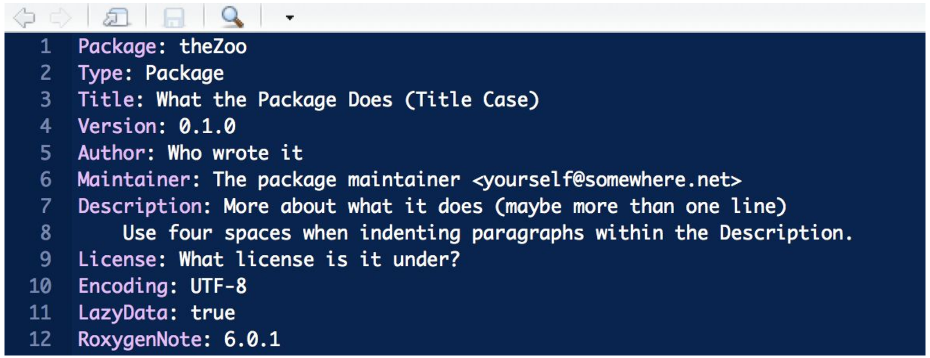


```
#' Go to the Zoo!
#'
#' @param animal String. What kind of animal do you want to talk to?
#' @param sound String. What does that animal say?
#'
#' @return String. A sentence just for you
#' @export
#'
#' @examples
#' goToTheZoo("lion", "roar")
#'

goToTheZoo <- function(animal, sound){
  print(glue::glue("The ", animal, " goes ", sound,"!", sep = " "))
}

```

Description

After your documentation is all nice and neat, save the R script right where you found it.

Then go back out to the package top level, and open “DESCRIPTION”. This needs your input also. Fill out the obvious stuff, <u>don’t stress about it</u>.



```
Package: theZoo
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
RoxygenNote: 6.0.1
```

You might want to add a couple more things:

- What packages does it depend on?
- If you write vignettes, what package does that need?
- What version of R does this work with?

```
Depends: R (>= 3.4.0)
Imports:
    data.table,
    dplyr,
    feather
Suggests: testthat
VignetteBuilder: knitr
```

Put it all together

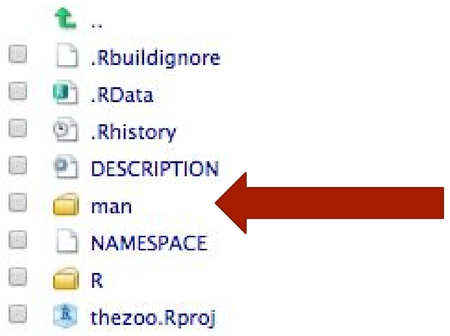
One more tiny thing - if you have hard dependencies it should import, open up NAMESPACE and put them in there too.

```
exportPattern("^[[:alpha:]]+")
import(data.table)
```

Time to make it real! I like the simple console command approach for this bit. Make sure your working directory is the package folder, and then…

```
devtools::document()
```

Fix any bugs that pop up, make sure it ran successfully (sometimes it takes a couple of tries, we all miss things sometimes!), and then swing over to the “man” (for manual) folder to see your docs!



Install your package!

Make sure you're still in the same folder, and just run this…

```
devtools::install()
library(theZoo)
```

Now, navigate to your list of packages, and yours is there! Open it up and read your pretty docs.

```
?theZoo::goToTheZoo
```

Then try using your functions! If you are using my scripts, you have two functions you can try.

Alternate installation choice:

```
install.packages("~/theZoo", repos =
NULL, type = "source")
```

Share the package with others!

The simplest way is to just load your package to a Git repo, and then anyone can download, and install it just the same way you just did.

If you collaborate with others on the package, most of the hard work is in git, and all you need to do is manage your branches wisely and remember to update your local versions.

Help with Github:

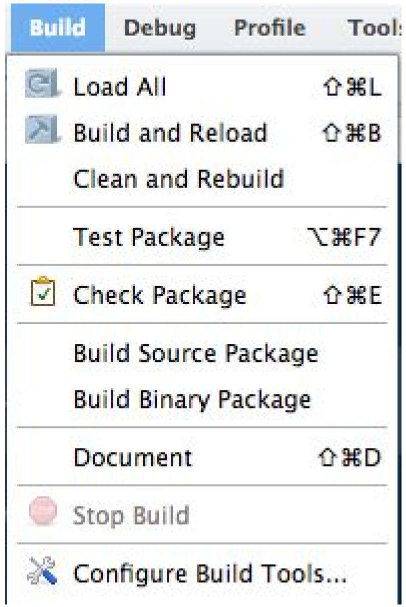
- https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners
- https://guides.github.com/
- https://try.github.io/levels/1/challenges/1

Extra Fun Stuff

Continuing to add to the package will be easy, since you know exactly where to put stuff and how to make it work.

But, as you continue to develop your package, these tips might be useful. This is possible because of that “Project Options” stuff we did earlier.

- **Checking**: Run through some best practice tests from RStudio
- **Rebuilding fast**: document and rebuild with a couple of clicks!
- **Unit testing**: Make sure your functions behave as expected
- **Make a documentation webpage**: Use pkgdown, http://pkgdown.r-lib.org/



Best Practices for Package Management

Possible organization ideas:

- Thematic similarity (all about the zoo)
- Sequential tools (things you use together a lot)
- User similarity (stuff that the same people use a lot)

Don't just throw everything you have into one package if they have no relationship. Make another package, if you need one!

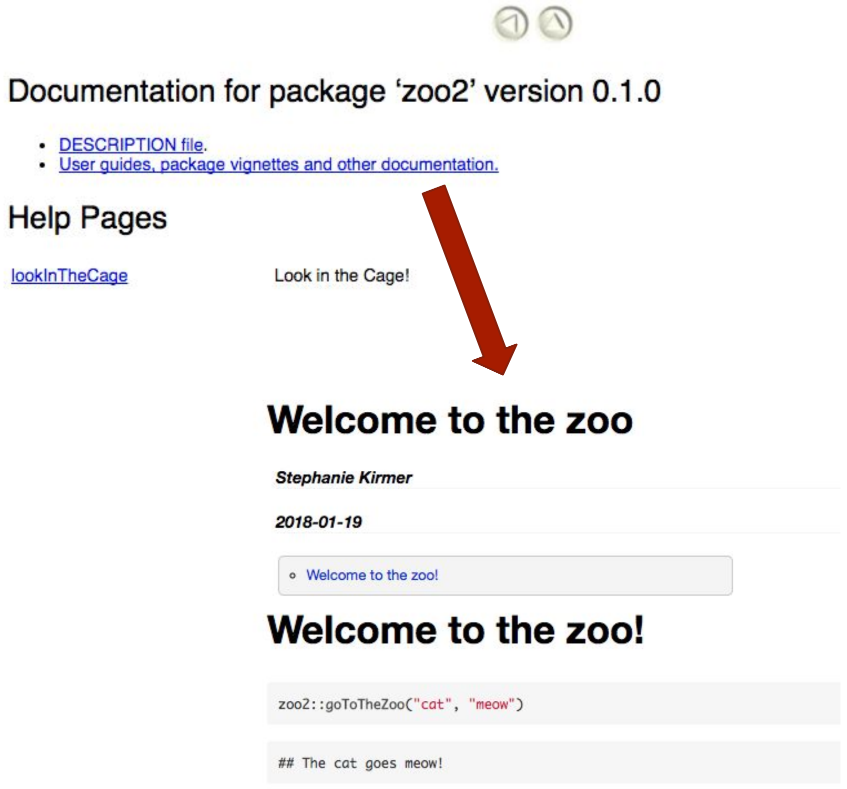
Writing a good function:

- Don't overcomplicate. If you need to make two functions, that's ok! Keep each short.
- Give your objects and arguments sensible names. "a" is never the right name for an object in R.
- Write for readability. Think about how well someone else would understand it, if they read your code. Be clear and neat. Write for collaboration, and future you!

Vignettes

If you want to go above and beyond, you can write a vignette to help people understand your package.

1. Run `devtools::use_vignette("my-vignette")`, using whatever the file should be called.
2. Write your vignette, give all the information someone might need.
3. Run `devtools::build_vignettes()`
4. Document, build the package, and check it out!



Documentation for package 'zoo2' version 0.1.0

- DESCRIPTION file.
- User guides, package vignettes and other documentation.

Help Pages

lookInTheCage	Look in the Cage!

Welcome to the zoo

Stephanie Kirmer

2018-01-19

- Welcome to the zoo!

Welcome to the zoo!

```
zoo2::goToTheZoo("cat", "meow")
```

```
## The cat goes meow!
```

Reference

Licensing:

http://r-pkgs.had.co.nz/description.html#license

General FAQ:
https://support.rstudio.com/hc/en-us/articles/200486488-Developing-Packages-with-RStudio

Tutorials (I regularly use these!):

- http://tinyheero.github.io/jekyll/update/2015/07/26/making-your-first-R-package.html
- http://www.masalmon.eu/2017/12/11/goodrpackages/
- https://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/

Unit testing:
https://www.r-bloggers.com/unit-testing-with-r/

Writing good functions:

https://nicercode.github.io/guides/functions/

Vignettes:

http://r-pkgs.had.co.nz/vignettes.html

Descriptions:

http://r-pkgs.had.co.nz/description.html