Building R Packages

July 11, 2014

Devasena Inupakutika
The Software Sustainability Institute
University of Southampton

Why build an R package?

- Accessible
- Reliable
- Clarity
- Sharing data, functions, and an analysis online (Distribute R code and documentation, writing functions that are reusable)

If you want to get package on CRAN

Key source: Writing R Extensions

A simple example: RSkittleBrewer

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.

Clear code – Basic principles

- Code that works

  No bugs, efficiency is secondary.

- Readable

  Fixable; extendible

- Reusable

  Modular; reasonable general

- Reproducible

  Re-runnable

- Think before you code

  More thought => fewer bugs/ re-writes

- Learn from other codes → R itself; key R packages

What are all these packages?

- **Methods** – Tools for graphics, data exploration, complex numerical techniques, making it easier to work with big data sets etc.
- **Open research** – Researchers publish packages that implement new methods or release data, which supports reproducibility.
- **Data** – Sharing old, new, simulated or research data sets. Many best packages have both methods and data.

Ultimate goal

Build a package to fulfill a need

- **Considerations**
  - R community is wide.
  - Even if a method is already available, it doesn't mean it was written efficiently, accurately or reaches to all audiences
  - May be preferable to help improve existing package than building new one from scratch

You're going to build a package?

- Establish clear aims for software before starting and choose a clear point at which you'll publish your work
- Achieve the basics
- Good coding practices
- Document your work

Early planning

- Practical considerations
- Early code planning (Functions, utilities etc.)
- Planning function details (arguments, actions and relations)
- Complex function output is common (objects to be simple enough for users to interact with directly and diagnostics of these data objects)

When to start building?

Existing functions

- Package existing functions immediately to facilitate documentation and access
- May later remove depreciated functions or add new functions (same is true for data)

Upcoming projects

- Initialize package for project even if no code or data exist
- Save functions and add documentation

Evaluating and re-evaluating

- Build foundation of diverse examples
  - Use test cases to assess accuracy
  - Using Rprof, Sys.time or system.time, identify sections of code for efficiency improvements
- Sufficiently general
  - Does it work for original problem?
  - Is it easy to apply to similar scenarios and data?
  - Possible extensions?

Picking data sets

- Always include data in a package
- Which examples highlight the package?
  - Include examples, plotting functions, visualizations or simulated data (real data is preferred but better than no data)

Classes and methods

- Allow users to connect old, familiar functions with new objects
- Class → Set of objects that share specific attributes and a common label
- S3 classes in R
- Methods → Name of function or action that is applied to many type of objects
- Eg. print, summary, plot, predict
- When we create a new, complex data object from a new function, creating a new class with methods can drastically improve the usability of the function and results

How to create a new S3 class

- Apply class function:

```
> x<- list(beard=TRUE, legs=4, tails=1)
> class(x)
[1] "list"
```

- We can also use class to change an object's class:

```
> class(x)<= "goat"
> class(x)
[1] "goat"
```

Creating new S3 class

- Usually an object's class is changed before the end-user ever sees it
- To create a new class, simply assign a new class to an object before returning it from a function

  Example: the lm function
- The lm function outputs an object of class "lm", which is really just a list with its class changed
- The strategy: initialize the object to be returned, immediately change the initialized object's class, and then continue to add on attributes as needed

  Warning. It is possible to assign an existing class (e.g. "lm") to a new object, but this generally creates problems if the object doesn't match the structure of other objects in that class

Building a method (example)

Suppose we have a method, say print, that we would like to customize for the new "goat" class, then we build a new function called print.goat:

```
> print.goat <- function (x, ...){
    cat("Number of legs:", x$legs, "\n")
    cat("Number of tails:", x$tails, "\n")
    y<- ifelse(x$beard, "The goat has a beard", "")
    cat(y, "\n") }
> x Number of legs: 4 Number of tails: 1 This goat has a beard
>
```

Making more methods

Generalizing

- Consider a method called Method and a new class called "Class"
- Suppose we want to allow users to apply Method to an object of class "Class"
- We create a new function called Method.Class, which R will then invoke whenever Method is applied to an object of class "Class"
- Recall: to construct the print method for the "goat" class, we made a function called print.goat

Complex objects can and should work with a variety of familiar methods

- Specify a summary for a new, complex object of class "Class" by writing a new function called summary.Class

- Similarly, if appropriate, make a custom method of plot for an object of class "Class" by creating plot.Class

Overview

Step1: Create the package files

- Load the new package's data objects and functions in **R** session
- To generate the basic package files, run

  package.skeleton("packageName")

Step2: Edit the package files

- Fill in the DESCRIPTION and help files (man> .Rd)
- Edit or add NAMESPACE file
- Function or data updates should be done within the package files

Cont…

Step3: Build, check and install the package

- Run a few Unix commands to build, check and install the package
- Usually errors arise when checking the package, so return to step2 as needed

Step 1: Create the package files

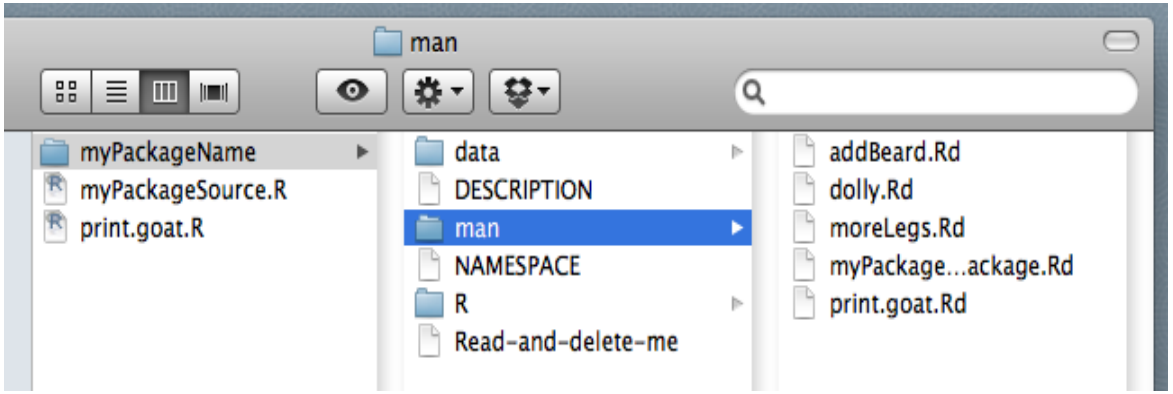
- Load new package's data objects and source functions into an R session
- To generate the basic package files, run

  package.skeleton("packageName")
- getwd function gives location of files
- Find the package files in this folder and move them, if needed, to where you want the package files to live in your computer

Cont...

```
> addBeard <- function(x){ x$beard <- TRUE;     return(x) }
> moreLegs <- function(x){ x$legs  <- x$legs+1; return(x) }
>
> dolly    <- data.frame(beard = FALSE, legs = 4, tails = 1)
> class(dolly) <- "goat"
>
> source("print.goat.R")
>
> package.skeleton("myPackageName")
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './myPackageName/Read-and-delete-me'.
>
```

The package source folder, which has the same name as that specified in `package.skeleton`, contains several files and folders that were automatically generated



`data` (folder)	Contains `.rda` files of each data object
`DESCRIPTION`	General package information
`man` (folder)	Help files
`NAMESPACE`	Manages function, method, and dependency info
`R` (folder)	Contains `.R` files for each function
`Read-and-delete-me`	File to be deleted

Step 2: Edit the package files

- Fill in the DESCRIPTION and help files (man > .Rd)
- Edit or add a NAMESPACE file
- Function or data updates should be done within package files

Edit the package files - DESCRIPTION

DESCRIPTION file instructions

- Update all information
- Choose your license (e.g. GPL-3 or GPL(>=2))
- If package is dependent on one or more packages, create new line that starts as Depends: and list the required packages, separated by commas
- If package depends on later version of R, then it can be specified as R (>= 3.0.1) or R (> 2.10.1) on Depends line

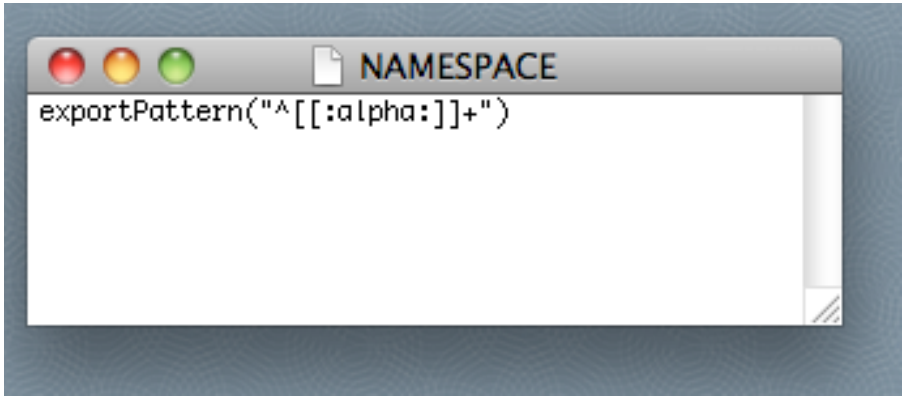
DESCRIPTION

```
Package: myfirsttestpackage
Type: Package
Title: Prints goats characteristics
Version: 1.0
Date: 2014-07-08
Author: Devasena Inupakutika
Maintainer: Devasena Inupakutika <di1c13@ecs.soton.ac.uk>
Description: Test package which prints number of legs and tails of goat and print function
is overridden function.
License: GPL-3
Depends: stats, utils, graphics, grDevices, R(>= 3.0.1)
```

Edit the package files - NAMESPACE

Very basic `NAMESPACE` file

- Earlier versions of R don't automatically generate a `NAMESPACE` file, so add one if needed with no extension (eliminate the `txt` extension after the file is created, if it was added by the text editor)
- If there are no special functions you want hidden, no methods in the package, and no package dependencies, then just leave the file as-is
- If you had to make your own `NAMESPACE` (probably because you are using R version $< 2.14.0$), put in the `exportPattern` command listed below



Edit the package files - NAMESPACE

Editing the NAMESPACE file

- To have hidden functions Replace the exportPattern command with an export command, where export's arguments are comma-separated function names that should be accessible to users
- **Methods**. To specify S3 method called Method for class "Class", create a line in the NAMESPACE file as S3method(Method, Class)
- **Dependencies.** Some users may prefer Imports: instead of Depends: in the DESCRIPTION file, and they must then also provide a command import in the NAMESPACE file whose arguments are the names of packages that the new package imports

NAMESPACE

```
export(addBeard,moreLegs)
S3method(print,goat)

# e.g.
# import(pkgToImport1, pkgToImport2)
```

Edit the package files – man files

Basic rules of help (`.Rd`) files

- Notation is similar to LaTeX where commands start with a backslash
- Use `\code{ }` to write in `Courier`
- Note: the `\example` section already uses `Courier`
- To create a link to a help file for a data object or function, say `addBeard`, use `\link{addBeard}`
- If the data object or function is from another package, its package name must also be in the link: `\link[otherPkg]{otherFcn}`
- Usually place `\link` command inside of `\code` but never vice-versa

Equations with LaTeX notation require two new commands

- In-line equations use the `\eqn{ }` command instead of dollar signs
- Stand-alone one-line equations use `\deqn{ }`
- The LaTeX-formatted equations will only show up in the package manual and otherwise appear plain

Edit the package files – man files

Delete help (`.Rd`) files for functions that are both not exported and not S3 methods

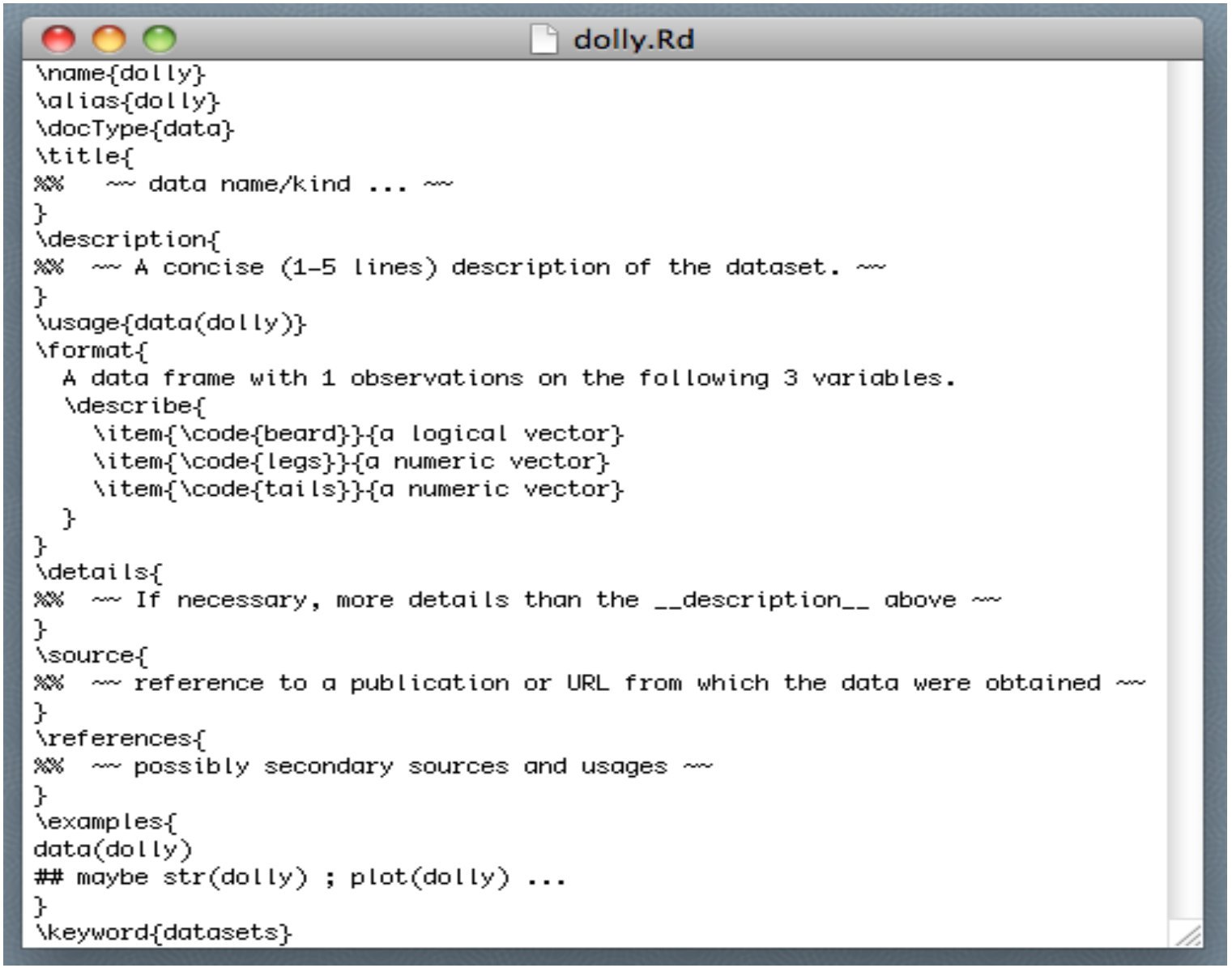
Follow the template instructions in each help file

- Must provide a title for every help file
- Delete sections that are not needed, perhaps `\details{ }` or `\references{ }`
- The package help file may have a few lines outside of any commands (starting with ~), which should be deleted

Merging help files for two or more functions

- Choose one help file that will be the help file for the functions
- Copy the `\alias{ }` and possibly also any `\usage{ }` commands from the other help files into this main help file
- Add in additional argument descriptions, as needed, and any supplemental descriptions to the merged help file
- Finally, delete the other help files

Edit the package files – man files (data)

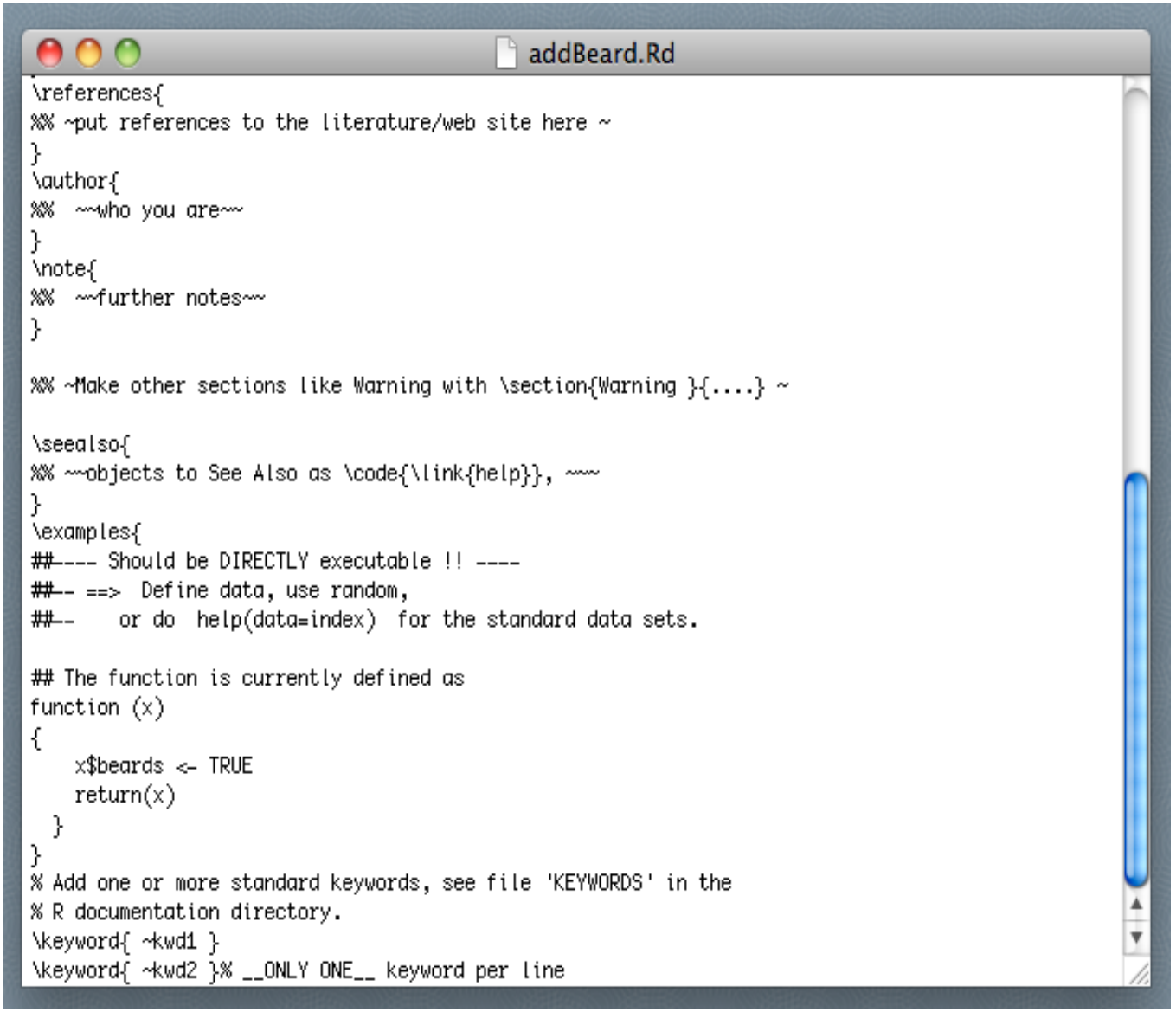


Edit the package files – man files (function)

addBeard.Rd

```
\name{addBeard}
\alias{addBeard}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{
%%  ~~function to do ... ~~
}
\description{
%%  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
addBeard(x)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{x}{
%%     ~~Describe \code{x} here~~
}
}
\details{
%%  ~~ If necessary, more details than the description above ~~
}
\value{
%%  ~Describe the value returned
%%  If it is a LIST, use
%%  \item{comp1 }{Description of 'comp1'}
%%  \item{comp2 }{Description of 'comp2'}
%% ...
}
\references{
%% ~put references to the literature/web site here ~
}
```

Cont…


addBeard.Rd
\references{
%% ~put references to the literature/web site here ~
}
\author{
%% ~~who you are~~
}
\note{
%% ~~further notes~~
}
%% ~Make other sections like Warning with \section{Warning }{....} ~
\seealso{
%% ~~objects to See Also as \code{\link{help}}, ~~~
}
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##--   or do help(data=index) for the standard data sets.
The function is currently defined as
function (x)
{
    x$beards <- TRUE
    return(x)
  }
}
% Add one or more standard keywords, see file 'KEYWORDS' in the
% R documentation directory.
\keyword{ ~kwd1 }
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line

Edit the package files – adding data

To add a new data object, say obj

- Load the data into R and save the data object to a file: save(obj, file="obj.rda")
- Create a help file: prompt(obj)
- See where the 2 files got saved: getwd()
- Move the files into the data and man folders, respectively
- If the package didn't already have a data folder, then add one

Edit the package files – adding functions

Add a new function, say fcn

- Save the function declaration/definition to a .R file (a text file with .R extension)
- Load the function into R and generate a help file: prompt(fcn)
- See where the help file for saved: getwd()
- Move the .R file to package's R folder and move the help file to man folder

Step3: Build, check and install

Inside the terminal

- Build a .tar.gz file for sharing the package

  R CMD build myPackageName

- To check the package, perhaps before submitting to CRAN

  R CMD check myPackageName.tar.gz

- Install the package from its folder

  R CMD INSTALL myPackageName

Any Questions?