Building an R package

Jumping Rivers

Do you need statistical or computational assistance in a grant application? We can help!

1 *My first package*

1.1 *RStudio projects*

RStudio projects make it straightforward to switch between analyses. You can create an RStudio project:

- in a brand new directory or
- in an existing directory where you already have R code and data or
- by cloning a version control[1] repository.

[1] Git or Subversion

Each project has its own working directory, workspace, history, and source documents.

When we create an RStudio package project, this will create a new directory that contains five files.

- `.RbuildIgnore`: this file contains a list of file names that the R build process should ignore when constructing your package. For example, the `.Rproj` file.
- `DESCRIPTION`: an overview of your package.[2]
- `*.Rproj`: an RStudio project file.
- `NAMESPACE`: a file that contains exported/exposed package functions[3].

[2] `http://goo.gl/AXSft`

[3] `http://goo.gl/I5yGf8`

Your package directory will also contain two directories.

- `man`: documentation directory: when you use the `help` function in R, the help pages returned live in this directory.
- `R`: R file directory: all R functions exported from your package belong in this directory.

Tasks

1. Create a "package" project, via

   ```
   File -> New Project -> New Directory -> R package
   ```

   Call the package `pkg` and select the directory you want to store the package in. Then click

   ```
   Create Project
   ```

   Congratulations - you have just created your first package called `pkg`
2. Click

```
Build -> Build and Reload
```

3. Now type

```r
library("pkg")
hello()
```

4. Look at the `NAMESPACE` file. Notice that the function `hello` has been exported.

5. The `hello` function has also been documented

```r
help("hello")
```

   The associated documentation file is in the `man` directory.

1.2 *roxygen2*

Keeping the `NAMESPACE` file and documentation up-to-date is a painful experience. To ameliorate the process, we use the `roxygen2` package to automatically generate the necessary entries. Above function definitions we add `roxygen2` tags. The tags are of the form:

```r
#' @export
#' @details
#' @aliases
```

Notice the tags are just R comments.

Tasks

1. Check that you have the necessary R packages installed

```r
library("devtools")
library("roxygen2")
```

   If you don't have them installed, then install them in the usual way

```r
install.packages(c("devtools", "roxygen2"))
```

2. Click on

```
Build -> Configure build tools
```

   then select

```
Generate documentation with Roxygen
```

   and click OK. Now when we build our package, RStudio will automatically run

```
library("roxygen2")
roxygenise(".")
```

3. In the file `R/hello.R` add

   ```
   #' @export
   ```

   just above the `hello` function, i.e.

   ```
   #' @export
   hello <- function() {
     print("Hello, world!")
   }
   ```

   The `export` tag above the `hello` function indicates that we want to export[4] this particular function.

4. Now delete the `man` directory and the `NAMESPACE` file[5]. Select[6]

   ```
   Build -> Build and reload
   ```

   You should now be able to load your package with

   ```
   library("pkg")
   ```

   and call the `hello` function

   ```
   hello()
   ```

5. Open the `NAMESPACE` file. You should see the entry

   ```
   # Generated by roxygen2: do not edit by hand
   export(hello)
   ```

[4] Export means the users loading this package, can access this function.

[5] The reason for deleting these is that we will automatically generate them using `roxygen2` - more details below.

[6] The keyboard shortcut for this is `Ctrl + B`

1.3 *Adding new functions*

All R functions that we create in our package are saved in the `R/` directory[7]. This directory can contain multiple files.

[7] The files can have a `.R` or `.r` file extension. Personally, I prefer `.R`

Tasks

1. Create a new file in the `R/` directory called `basic.R`. In this file add the following code

   ```
   #' @export
   add = function(x, y) {
     return(x + y)
   }
   ```

   Build and reload your package[8]. After reloading your package, the following code should run

[8] Remember the keyboard shortcut `Ctrl + B`

```
library("pkg")
add(1, 2)
```

2. Create a new function called `check_numeric`

   ```
   check_numeric = function(x) all(is.numeric(x))
   ```

   and save it in the `basic.R` file.
3. Now use `check_numeric` in the `add` function

   ```
   add = function(x, y) {
     if(!check_numeric(c(x, y))) stop("Not numeric")
     x + y
   }
   ```

   Rebuild your package and check that the `add` function still works.
4. Notice that we haven't exported the `check_numeric` function[9], so this will raise an error

   ```
   library("pkg")
   check_numeric(1)
   ```

   but we can access any non-exported function in a package using the `:::` operator

   ```
   pkg:::check_numeric(1)
   ```

   We can access[10] any exported function using `::`

   ```
   pkg::add(1, 1)
   ```

5. Now create a function `subtract` and export this function. Rebuild your package and check that this works OK.
6. Delete your package and re-add the functions: `add`, `check_numeric` and `subtract`[11].

[9] Remember to export a function, the function name should be in the `NAMESPACE` file.

[10] The benefit of doing this is that we haven't loaded the package.

[11] The purpose of this is to highlight how easy it is to create packages.

2 Documentation

Using roxygen2 simplifies documentation[12]. The premise of roxygen2 is simple: describe your functions in comments next to their definitions and roxygen2 will process your source code and comments to produce Rd files in the man/ directory. In theory, you should never directly edit the Rd files.

[12] The first package I wrote didn't use roxygen2 and it was a very painful experience.

2.1 Tasks

1. Copy the following roxygen2 descriptions to your add function

```r
#' @title A function for adding
#'
#' @description A really good adding function.
#' Perhaps the best function ever!
#'
#' A work of pure genius.
#' @param x a number
#' @param y another number
#' @return a number
#' @export
#' @examples
#' add(5, 10)
#' ## Can also use negative numbers
#' add(-5, 10)
add = function(x, y) {
  if(!check_numeric(c(x, y))) stop("Not numeric")
  x + y
}
```

2. Rebuild your package and look at the help file for the add function, i.e. ?add. Run the examples via

Tag name	Description
@title	Short title for documentation page.
@description	Longer description page. Skip a line for a new paragraph.
@param	Function parameter description.
@inheritParams	Use the parameter definition from another function.
@export	Add the function to the NAMESPACE file.
@return	What does the function return, e.g. a data frame.
@examples	Function examples (will be run when building).
@rdname	Point multiple functions to the same help file, e.g. ?substr.
@seealso	Pointers to other documentation pages.
@importFrom	Import functions from other packages.

Table 1: Useful roxygen2 tags for documenting functions.

```
example(add)
```

3. Add a help page for the `subtract` function.[13]

[13] Build & reload.

4. Create a function called `multiply` and add an associated help page.[14]

[14] Build & reload.

5. Create a function called `times`

```
times = function(x, y) multiply(x, y)
```

   Export the `times` function.[15]

[15] Build & reload.

6. Use the `@rdname` tag above the `times` function to point to the `multiply` help page, i.e.

```
#' @rdname multiply
```

   Build and reload. Look at `?multiply` and `?times`. Now add `@examples` to the `times` function. Look at the new `times` help page.

2.2 *Importing functions*

We often want to use functions from other R packages. When we do this we need to be explicit, i.e. state what we want and from where. The great thing about R packages, is that when we install a package, the dependencies are also automatically installed.

Tasks

1. Install the package `nclRpackage`. First, we need the `drat` package

```
install.packages("drat")
```

   Then we add the `rcourses` repo[16]

[16] Run the `.libPaths()` function to see the repository location.

```
drat:::add("rcourses")
```

   Then install as usual

```
install.packages("nclRpackage", type = "source")
```

2. The `nclRpackage` contains a very useful function called `div` that we want to use

```
library("nclRpackage")
div(10, 2)

## [1] 5
```

   To use the `div` function within our package, we have to import it first

```
#' @importFrom nclRpackage div
```

and add an entry to the `DESCRIPTION` file

```
Imports: nclRpackage
```

Create a function `divide` that uses the `div` function.

2.3 *The `DESCRIPTION` file*

The `DESCRIPTION` file contains high level information about your package. For example, the package name, a brief description, the licence, and your email address.

Open the `DESCRIPTION` file and update fields with relevant information. An example is given below.

```
Package: pkg
Type: Package
Title: My First package
Version: 0.1
Date: 2016-11-01
Authors@R: person(given="Colin", family="Gillespie",
    email="colin@jumpingrivers.com", role = c("aut", "cre"))
Maintainer: Colin Gillespie <colin@jumpingrivers.com>
Description: This is my very first package. It contains
  exceedingly useful functions, such as add and subtract.
  Make sure you add a couple of spaces to indent the
  Description otherwise you will waste hours of your life
  trying to find the bug.
License: GPL-2 | GPL-3
LazyData: TRUE
```

2.4 *Package checks*

One of great things about R packages, is that there are a number of package checks that are available. These include

- Checking the syntax of the `DESCRIPTION` and `NAMESPACE` file.
- Checking your examples run.
- Checking all exported functions are documented.

Tasks

1. Run the standard package checks on your package, via

   ```
   Build -> Check Package
   ```

   Check that you package passes all tests. [17] Fix any errors, warnings or notes.

2. Add the following example to the `add` function

[17] CTRL + E

```
#' add("A", "B")
```

Build the package. Does the package still build? Check the package. Does the package pass all tests?

3 Data and demos

3.1 Data in packages

Packages can also contain example data sets[18]. Data files[19] can be one of three types as indicated by their extension.

- plain R code (`.R` or `.r`)
- tables (`.tab`, `.txt`, or `.csv`)
- `save()` images (`.RData` or `.rda`).

Data files live in the `data/` directory.

Each data file should also have an associated help page. The easiest way to generate a help page is to use `roxygen2` and a dummy R function. Typically, I have a file called `data_help_files.R`, which has entries for the each data set. For example,

[18] Full details are given at `http://goo.gl/Y4Srx4`.

[19] Note that R code should be "self-sufficient" and not make use of extra functionality provided by the package, so that the data file can also be used without having to load the package.

I tend to use `RData files`.

This is entry is taken from the `poweRlaw` package.

```
## Entry in data_help_files.R
## Name is name of the data set.
#' @name moby
#' @aliases moby_sample
#' @title Moby Dick word count
#' @description The frequency of occurrence of unique words
#' in the novel Moby Dick by Herman Melville.
#'
#' The data set moby_sample is 2000 values sampled from the
#' moby data set.
#' @docType data
#' @format A vector
#' @source M. E. J. Newman, "Power laws, Pareto distributions
#' and Zipf's law." Contemporary Physics 46, 323 (2005).
NULL
```

3.2 Tasks

1. Create a `data/` directory.
2. Create the following data frame

   ```
   example_data = data.frame(x = runif(10), y = runif(10))
   ```

   Now save[20] the data frame `example_data` in the `data/` directory

   ```
   save(example_data, file="data/example_data.RData")
   ```

3. Create a file called `data_help_files.R` in the `R/` directory and document your new data set.
4. Build and reload your package. Can you access the help page and the data set?
5. Check that your package still passes all tests[21].

[20] Use the `save` function.

[21] CTRL + E

3.3 *Demos*

A demo is similar to function examples, but is typically longer and shows how to use multiple functions together. Demos are plain `.R` files that live in the `demo/` package directory. The demos are accessed with the `demo()` function.

In `demo/` directory, there should also be an `00Index` file, that lists the demos[22]. For example,[23]

```
demo1      My very first demo
demo2      My very second demo
```

[22] There is a planned `demoTitle` tag for `roxygen2`, but currently this hasn't been implemented.

[23] Note the white space separation in the `00Index`. Use at least four spaces to avoid annoying error messages.

3.4 *Tasks*

1. Create a `demo/` directory.
2. Create a file called `first.R` and save it in the `demo/` directory. In this file show how you can use some of your newly created function.
3. Add a `00Index` file to the `demo/` directory.
4. Build and check your package.

4 Vignettes

If you want to include more extensive examples or even just further documentation, then you should consider creating a vignette:

> a **vignette** is a small illustration placed at the beginning or end of a book or chapter.
>
> *http://dictionary.reference.com/browse/vignette*

We can view vignettes from other packages using the `vignette` function

```
vignette(package="knitr")
```

or to view in your web browser

```
browseVignettes(package="knitr")
```

4.1 Example: markdown vignettes

Vignettes are stored in the `vignettes/` directory. The simplest vignette uses R-markdown and is formatted by the `knitr` package. To create a package vignette, we simply place the file in the `vignettes/` directory. For example, suppose we have a file `intro.Rmd` that contains the following text:

The output style is `html_vignette`. This is more lightweight (in terms of file size) than the standard `html_document`
The `UTF-8` line specifies the file encoding.

~~~
---
title: "My very first vignette"
author: "Colin Gillespie and Robin Lovelace"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{My very first vignette}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
---

## My first package

This is my **first** package vignette.
I can include mathematics, such as $x^2$.
R code is also nicely formatted and displayed.

```{r}
x = runif(10)
x
~~~

and plots
```{r}
plot(x)
```
```
```

1. The first few lines register `knitr` as the vignette engine and provide an entry for the list of vignettes.
2. We have used markdown to add simple styling[24]. For example `**first**` becomes **first** and `$x^2$` becomes x^2.
3. R code is executed in the "' regions.

[24] See `http://goo.gl/y9F05` and the RStudio markdown reference sheet for further commands.

4.2 *Tasks*

1. Create a `vignettes/` directory in your package.
2. Create an R markdown file

   ```
   File -> New File -> R markdown
   ```

   and save it in the `vignettes/` directory.
3. Copy the markdown example above into your file and *knit* that file.
4. Add

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

   to the `DESCRIPTION` file[25]. This will tell your package to build vignettes using `knitr`.
5. Vignettes won't actually be built unless you are creating a source bundle[26]. To install a package with vignettes included, we first create the source package

   ```
   Build -> Build Source Package
   ```

   and then install the package from source

   ```
   install.packages("pkg_1.0.tar.gz", repos=NULL,
                    type="source")
   ```

   Build and install your package. Check that you can access the vignette.
6. Check and ensure that your package passes all tests[27].

[25] Notice the `knitr` is only a **suggested** package, since it's not essential.

[26] If you submit your package to CRAN, then the vignettes will be built ready for distribution.

[27] CTRL + E

4.3 *Package level documentation*

You package typically also has documentation associated with the package name. For example,

```
library("pkg")
?pkg
```

should bring up an overview of your package. Again we use `roxygen2` to generate the man page. Go to

`http://goo.gl/W2tJrF`

to view the entry for the `poweRlaw` package.

4.4 *Tasks*

1. Create a file called `pkg-package.R` in the `R/` directory.
2. Using the `poweRlaw` package as an example, create a man page for your package[28].
3. Do one final check and ensure that your package passes all tests.

[28] What do you think the `@aliases` tag does?