Why and how to build an R package

Chris Mainey - Healthcare Evaluation Data (HED)

University Hospitals Birmingham NHS FT

chris.mainey@uhb.nhs.uk

HED.nhs.uk

mainard.co.uk

github.com/chrismainey

twitter.com/chrismainey

Background image by Leone Venter: https://unsplash.com/@fempreneurstyledstock

Overview

- What is an R package and why might you want to build one
- Functions
- Package basic structures and concepts
- `Roxygen2` documentation and vignettes
- Dependencies
- Build and check
- Extras:
  - Source control (Git and GitHub)
  - Unit tests and code coverage
  - Releasing to CRAN

We'll look at this practically by building an R package

References:

- Hadley Wickham, Jenny Bryan's: Building R packages
- `R` official Documentation

R packages

- Use packages all the time (base, stats, utils). You probably know `tidyverse`, `dplyr`, `ggplot2`.
- `R` package repository CRAN has 16181 packages at time of writing.
- Other places like Bioconductor, source packages on GitHub etc.

- **What are they?**
  - Collections of functions related to certain workflows and tasks
  - Up to the author what goes in
  - Can be depend on other packages or be written from scratch

- **Why use them?**
  - Don't reinvent the wheel!
  - Do you know how to do *[Insert thing]* better than a topic expert?

- **Why build a package?**
  - You might be the topic expert!
  - Share your code with colleagues, or others, and build collaboratively.

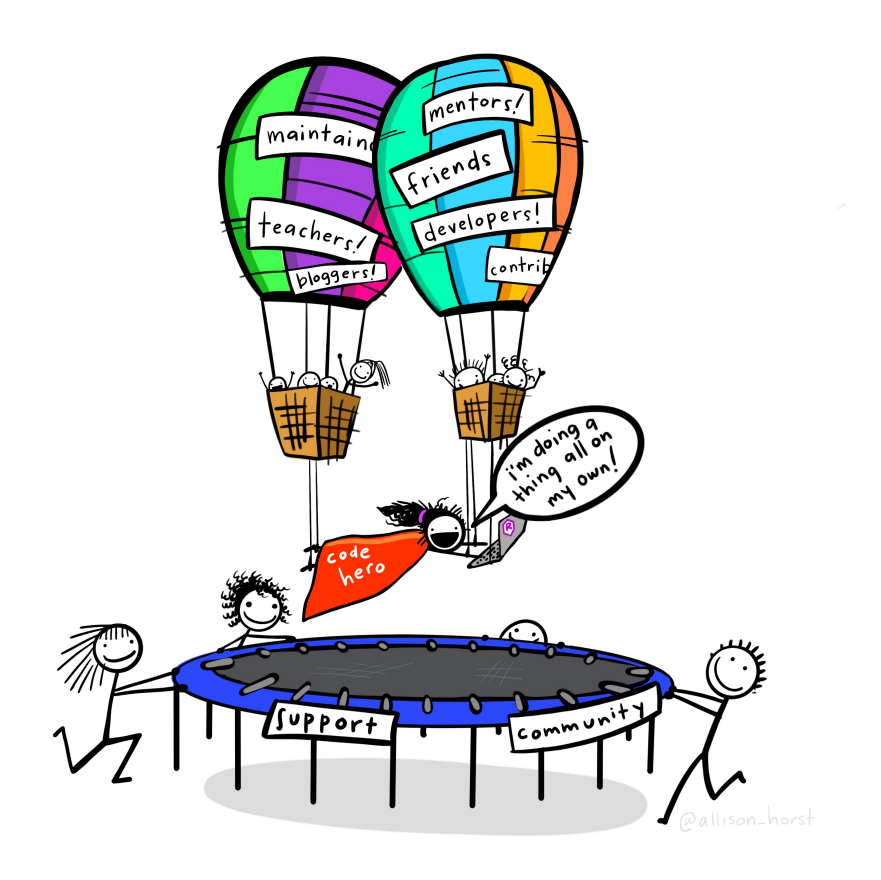
But I can't build one...

...yes you can! You don't need perfect code to share it! *As you'll see from mine...*

- You can get much of the background today and by looking online.
- You will get it wrong a lot. That's part of the process.

Our first step is to convert our code to a format that can be run by others

- We need to convert our code to a `function`.



Artwork by @allison_horst: https://github.com/allisonhorst/stats-illustrations

Functions

Functions have a structure that considers generic inputs and outputs, and performs something on them in between. *(bad explanation, but stay with me...)*

```
mydataframe <- data.frame(id=seq(3), old_col = c(97.5, 100, 147.5))
```

You have code like this:

```
mydataframe$new_col <-
  (mydataframe$old_col + 2.5) / 50

mydataframe
```

```
##   id old_col new_col
## 1  1    97.5    2.00
## 2  2   100.0    2.05
## 3  3   147.5    3.00
```

Turn it into a function:

```
my_function <- function(col){
  rtn <- (col + 2.5) / 50
  return(rtn)
}

mydataframe$new_col2 <- my_function(mydataframe$

mydataframe
```

```
##   id old_col new_col new_col2
## 1  1    97.5    2.00     2.00
## 2  2   100.0    2.05     2.05
```

Functions

- **Function has:**
  - A name (`my_function`)
  - An input (`cols`)
  - A return value (by default, returns the last line, but can be explicit with `return()`)

```
my_function <- function(col){
  new_col <- col + 2.5 / 50
  return(new_col)
}
```

Functions

- **Function has:**
  - A name (`my_function`)
  - An input (`cols`)
  - A return value (by default, returns the last line, but can be explicit with `return()`)

```
my_function <- function(col){
  new_col <- col + 2.5 / 50
  return(new_col)
}
```

It's these functions that we use, and often chain together to build packages. We need to:

- Document them, including help files for users.
- Tell `R` whether they depend on any other files/packages.
- Better packages will consider error handling and how best to return values.

Structure of a package

Can contain one or many functions. The parts it requires are:

- **DESCRIPTION** - A file with meta data about package, authors etc. Includes a list dependencies
- **NAMESPACE** - A short-hand file so `R` can understand what functions and dependencies to import
- **Function(s)** - Files with `R` functions, saved in a directory called `R`
- **Help files** - Text describing functions. These are written in syntax similar to latex, but it's common to generate them using `roxygen2` - a package documentation tool.

Optional:

- **Vignette** - Worked examples of using your functions. I think all good packages should have at least one. Again, written in same format as help files, or Rmarkdown.
- **Unit tests** - Automated tests help to detect errors in your code when working on a package.

Dependencies

If you package requires other packages, you need to convey that so they are installed.

- Description file can specify:
  - `Imports` - list packages that are ***required*** to use your code (added to NAMESPACE).
  - `Suggests` - A courtesy to your users, packages that are not required but help (e.g. used in vignette). Don't need to use this with a local file

You no longer need `Depends` since NAMESPACE files are used, and `LinkingTo` can be used to link to C++ or other files. `Enhanced` can be used to indicate your functions enhance another package.

Refer to Jim Hester's post: https://www.tidyverse.org/blog/2019/05/itdepends/

Beware of tidyverse dependencies! Hadley's Wickham's advice:

> *Because the tidyverse is a set of packages designed for interactive data analysis, this is, in short, a bad idea. The tidyverse package includes a substantial number of direct and indirect dependencies (79 packages, as of this writing), many of which are likely unnecessary for the purposes of your package. Furthermore, the CRAN maintainers frown upon depending on it, which can cause hassle for you down the line.*

Extras (1)

Source control:

- Source control is important in software development, allowing you to track changes and 'rollback'.
- Git is common, but not only option. With Git you have a local repository, and can sync to a 'remote' like GitHub.
- You or other users can make 'branches' to build functions before merging them together.
- Happy Git with `R` https://happygitwithr.com/
- GitHub automatically renders README.md when you land on it.

Unit tests:

- Use `testthat` package, use a control script called 'testthat', in a folder called 'tests'.
- Sub-folder called 'test_that' containing test scripts.
- See `testthat` vignettes for more details.
- Can be combined with 'Code coverage' e.g. `covr` package.

Extras (2)

Continuous integration (CI)

- When building larger packages collaboratively, CI builds packages after each push.
- Recommend GitHub actions, but Travis-CI is also common.

`pkgdown`

- A extension builds a package website from your helpfiles, README, help files, and metadata.
- Can then be hosted anywhere, but GitHub pages is common.
- Can be built from CI systems

Trust the `usethis` package!

Lets build a package!

- Use Rstudio to set up a new project with an 'R package template'
- Write our functions.
- Insert a 'Roxygen Skeleton' ('Code' menu), and fill in
- Build our package, use CRAN checks
- Write a brief unit test

- Commit this all to GitHub.

- We will use Rstudio, `devtools`, `usethis` and `roxygen2` in many places, as they set up the elements for you.
- You need Rtools on Windows!

Here's one I prepared earlier:

https://github.com/chrismainey/brilliant2

Releasing to CRAN

- CRAN is the most popular repository for packages. It has clear rules, and moderated by wonderful volunteers.
- Rules around release, and package must pass CRAN checks and build on Windows, macOS and various Linux distros.
- CI can help with builds, but also use Win-builder and R-Hub, through `devtools`.
- Should include a 'NEWS' file and 'cran_comments' describing package, changes, and any errors in builds.
- Follow all the steps in `devtools::release()`

Look at an existing package:

https://github.com/chrismainey/FunnelPlotR

Summary

- Packages are ways to make functions portable
- Help you share code, collaborate and make code available to non-experts
- Rstudio gives you package templates, and integrated with build, check and other tools
- Metadata is saved in DESCRIPTION file
- NAMESPACE contains import/dependency details, generated in build in `RStudio`/`devtools`/`roxygen2`
- `R` functions saved in R folder, document using `roxygen2`
- Unit test check code is working properly (`testthat` package)
- Build (using RTools in Windows). If releasing to CRAN, use checks with `--as-cran` option.
- Can build to binary and install from zip