Session 1

R package deveopment

Matt Denwood

2023-05-17

Introduction

Material

Everything is on the public GitHub repo - to get the material:

1. Install GitHub Desktop from https://desktop.github.com (unless you already have it), open it, and log in with your GitHub credentials (unless you have already done so)
2. Choose “File” and “Clone Repository”
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added

Material

Everything is on the public GitHub repo - to get the material:

1. Install GitHub Desktop from https://desktop.github.com (unless you already have it), open it, and log in with your GitHub credentials (unless you have already done so)
2. Choose “File” and “Clone Repository”
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added

You only have write access, so you can’t push changes. It is best not to modify anything in the course files (notes etc) in case you get conflicts pulling changes later...

There are two versions of each presentation: PDF and html

I will present the PDF slides, but the R code is mangled by LaTeX

The html version is better for copying code from

Motivation

When should we create an R package?

IMMEDIATELY when starting a new project

When should we create an R package?

IMMEDIATELY when starting a new project

Even if:

- You will be the only person using the R package
- You are not expecting the code to be particularly complicated
- You have never created an R package before
- You already have an R package for a similar project

When should we create an R package?

IMMEDIATELY when starting a new project

Even if:

- You will be the only person using the R package
- You are not expecting the code to be particularly complicated
- You have never created an R package before
- You already have an R package for a similar project

Do not fall into these traps:

- "I will create an R package later, if I need to"
- "Creating an R package is not worth the time investment"
- "I will just put all of my code into one massive R package"

How should we create an R package?

I have a "standard template" that I suggest you follow, and an R package that exists only to create R packages (inspired by the usethis package). To install it run:

```
install.packages("neatpkg",
↪  repos=c("https://cran.rstudio.com/","https://ku-awdc.github.io/drat/"))
```

How should we create an R package?

I have a "standard template" that I suggest you follow, and an R package that exists only to create R packages (inspired by the usethis package). To install it run:

```
install.packages("neatpkg",
↪  repos=c("https://cran.rstudio.com/","https://ku-awdc.github.io/drat/"))
```

And then load the package:

```
library("neatpkg")
## Welcome to neatpkg!
## To get started see ?pkg_new - and remember fortunes::fortune(52) :
##
## > Can one be a good data analyst without being a half-good programmer?
## > The short answer to that is, 'No.'
## > The long answer to that is, 'No.'
## >   -- Frank Harrell
## >      1999 S-PLUS User Conference, New Orleans (October 1999)
```

Step 1: GitHub

1. Think of a name for your package

- ASCII only i.e. no øåæ etc
- Stick to normal text (a-z and A-Z) - no punctuation like - or .
- Shorter is better
- I prefer lowercase, but that is up to you

2. Go to github.com and:

- Log in if necessary
- Click the New button to create a new repository
- Select owner (ku-awdc recommended for KU employees)
- Enter the repository name as the package name
- Add a brief description
- Choose public or private (you can change this later, so start with private if you are unsure)
- Click the box next to 'Add a README file' (but NOT the other boxes)
- Click Create Repository

Step 2: GitHub Desktop

1. Open GitHub Desktop
2. Choose “File” and “Clone Repository”
3. Enter the name of your package then click “Clone”

Step 3: Create the R package

1. Launch RStudio
2. Change your working directory to the folder containing your git repositories
3. Run the following code (replacing "name" with your package name):

```
neatpkg::pkg_new("name")
```

Step 3: Create the R package

1. Launch RStudio
2. Change your working directory to the folder containing your git repositories
3. Run the following code (replacing "name" with your package name):

```
neatpkg::pkg_new("name")
```

That's it - you now have a functional R package!

The total time taken to do this should be around 1 minute, plus extra for deciding on the package name...

Exercise

Create your own R package!

Follow the instructions above to create a package that you will use to develop a simple disease model on this course. Once you have a package, start to have a look around.

Tour of an R package structure

Live tour...

Demonstrate adding roxygen comments...

Developing R packages

Writing functions

A function:

- Takes one or more user-supplied arguments (some may have default values)
- Checks that these arguments are sensible (using stopifnot, stop, or warning)
- Does something
- Returns either:
  - A simple object (data frame, numeric vector or similar)
  - A list of objects
  - A custom class (which is really just a list of objects with a class attribute)

Writing functions

A function:

- Takes one or more user-supplied arguments (some may have default values)
- Checks that these arguments are sensible (using stopifnot, stop, or warning)
- Does something
- Returns either:
  - A simple object (data frame, numeric vector or similar)
  - A list of objects
  - A custom class (which is really just a list of objects with a class attribute)

Functions must be exported, and the arguments should (eventually) be documented - use roxygen for this!

S3 classes

S3 classes are the simplest classes in R - they are particularly useful for associating methods (print, summary, plot, autoplot etc) with specific classes.

Using S3 classes is as simple as:

```
print.H2G2 <- function(x){
  cat("The answer to life, the universe and everything is ", x, "\n", sep="")
}

number <- 42L
class(number) <- "H2G2"
number
## The answer to life, the universe and everything is 42
```

S3 classes

S3 classes are the simplest classes in R - they are particularly useful for associating methods (print, summary, plot, autoplot etc) with specific classes.

Using S3 classes is as simple as:

```
print.H2G2 <- function(x){
  cat("The answer to life, the universe and everything is ", x, "\n", sep="")
}

number <- 42L
class(number) <- "H2G2"
number
## The answer to life, the universe and everything is 42
```

We will talk more about object-oriented programming later...

Variable names

Rule number 1: be consistent!

Variable names

Rule number 1: be consistent!

Rule B: follow the group consensus!

- Use snake_case for functions and variables
  - Function arguments count as variables
  - R has first-class functions, so functions (and methods) are also variables...
  - Don't re-use the name of the package as a function name

Variable names

Rule number 1: be consistent!

Rule B: follow the group consensus!

- Use snake_case for functions and variables
  - Function arguments count as variables
  - R has first-class functions, so functions (and methods) are also variables...
  - Don't re-use the name of the package as a function name
- Use PascalCase for:
  - Class names (S3, R6, and Rcpp)
  - Columns within data frames (to avoid clashes with variables)
  - Probably also element names within lists (because data frames are lists, and we like consistency)

Do *NOT* under any circumstances use dot separated names for anything, e.g.:

```
print.my.thing <- function(x, ...) { }
```

Is this:

1. A function called print.my.thing??
2. A print method for a my.thing class??
3. A print.my method for a thing class??

Do *NOT* under any circumstances use dot separated names for anything, e.g.:

```
print.my.thing <- function(x, ...) { }
```

Is this:

1. A function called print.my.thing??
2. A print method for a my.thing class??
3. A print.my method for a thing class??

We can't tell (and this *has* created real-world problems, e.g. the mcmc and mcmc.list classes).

Do *NOT* under any circumstances use dot separated names for anything, e.g.:

```
print.my.thing <- function(x, ...) { }
```

Is this:

1. A function called print.my.thing??
2. A print method for a my.thing class??
3. A print.my method for a thing class??

We can't tell (and this *has* created real-world problems, e.g. the mcmc and mcmc.list classes).

[Technically this doesn't apply to non-function variables, but we like consistency]

Testing functions (and methods)

You will need some R code to run/test the functions as part of development.

Keep this under notebooks while you are developing that part of the code.

Testing functions (and methods)

You will need some R code to run/test the functions as part of development.

Keep this under notebooks while you are developing that part of the code.

Once you are happy, move the R code either to a new function (as part of higher-level code) or to a unit test, so that you can automatically check your function in future whenever R CMD check is run.

Unit tests

Why unit test? It saves you time in the long run!

Unit tests

Why unit test? It saves you time in the long run!

These are all run automatically when checking the package

See under tests/

Unit tests

Why unit test? It saves you time in the long run!

These are all run automatically when checking the package

See under tests/

[Demo]

Building/installing an R package during development

1. Use Load All:

- Under Build tab, More, Load All, or devtools::load_all(), or the shortcut command-shift-L
- Faster, but doesn't always work

2. Use Install

- Under Build tab, Install
- Slower, but safer

Building/installing an R package during development

1. Use Load All:

- Under Build tab, More, Load All, or devtools::load_all(), or the shortcut command-shift-L
- Faster, but doesn't always work

2. Use Install

- Under Build tab, Install
- Slower, but safer

If you have changed any roxygen comments (particularly imports or exports) then you need to Document first!

- Under Build tab, More, Document or devtools::document()

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer

- Always use curlies {} around if() and else if() etc
- Use stopifnot() and stop() *liberally*

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer

- Always use curlies {} around if() and else if() etc
- Use stopifnot() and stop() *liberally*

2. Assign to the global environment using «- for a temporary look at your objects within functions

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer
   - Always use curlies {} around if() and else if() etc
   - Use stopifnot() and stop() *liberally*
2. Assign to the global environment using «- for a temporary look at your objects within functions
3. Use browser() as a step-through debugger

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer
   - Always use curlies {} around if() and else if() etc
   - Use stopifnot() and stop() *liberally*
2. Assign to the global environment using «- for a temporary look at your objects within functions
3. Use browser() as a step-through debugger
4. Use options(error = utils::recover)

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer
   - Always use curlies {} around if() and else if() etc
   - Use stopifnot() and stop() *liberally*
2. Assign to the global environment using «- for a temporary look at your objects within functions
3. Use browser() as a step-through debugger
4. Use options(error = utils::recover)

[Demo]

Checking and building an R package

You should check the package regularly (Build tab, Check)

Checking and building an R package

You should check the package regularly (Build tab, Check)

To build a release:

- Build, More, Build Source Package
- Run R CMD check on the tarball
- Fix any issues and repeat
- Eventually put the final .tar.gz into releases
- Increment the version number in DESCRIPTION

For others to install your package:

- Send them the .tar.gz and tell them to install from local source file
- Ask them to use remotes::install_github() - although this is NOT the release version
- Ask me to put your release version on our drat repository

git

Basics of git

Basic git requires you to remember the following:

- Follow the instructions above to create a new R package
- Use GitHub Desktop to push/pull changes (unless you can't install it, in which case try RStudio: https://www.costmodds.org/rsc/teaching/Session_Preparation.html)
- Remember to pull changes before you start working
- Remember to commit regularly (ideally once you have finished "a thing") and give it a good summary
- Remember to push at the end of the day (or more regularly, if others are also working on the package)

- Stick to a single branch (main) until you are confident you know what you are doing
- Working with multiple branches and pull requests are useful but more advanced!

Exercise

Instructions

1. Add a function running a simple mechanistic, discrete-time, one-population, two-compartment (SI) disease model to your package. It can be deterministic or stochastic. The function should return a data frame with an additional class (name of your choice), and should take three arguments:

- start_infected
- total_individuals
- time_points

1. Add some argument checking to ensure that e.g. time_points is not negative, start_infected $<$ total_individuals, etc
2. Add a suitable print method
3. Optional: add a suitable autoplot method (see ?ggplot2::autoplot)