Coding best practices

September 9, 2020

Contents

The reality is that for any non-trivial task, most of your time will be spent de-bugging your code, not writing it. There are two over-arching principles that will, when practiced, greatly increase your efficiency: writing *modular* code and writing *clean* code. In class we'll discuss modularity first (it relates to the *project mindset* referred to in the context of `git`) and will then talk about code-writing best practices (styles guides).

Principle of Modularity

If your experience learning to code as been anything like mine then, either when you first started coding or when you wrote homework assignments for Stats class, the code you created for a given project probably consisted of one long giant file. If you learned to use `Markdown` or `Sweave`, that code probably had some number of sections within it, just like you might write a paper or thesis chapter (Intro, Methods, Results).

That'll work fine for small (tiny) projects or homework reports, but probably not for anything the size of a paper or thesis chapter that contains some combination or subset of data, analyses (whether statistical or non-statistical modeling), and data- or results visualization.

Thus, as alluded to in **??**, you should give your project a useful structure:

Your `data` folder should contain all the data you need for the project. Most of the time, this will include both the *original* unedited raw data files (regardless of their format) as well as *derived* data files (that you have used code to produce). No files should be duplicates or derived copies of each other (see Fig. 1); remember: versions will be tracked by `git`.

Your `code` folder contains your various script files. For anything but the smallest projects there should be multiple files that contain scripts for performing different tasks. For example, you might have:

`data_prep.R` Script that pulls in the original data, pre-processes them (rearranges, correct errors, standardizes names, splits up or merges different data sets), and spits them back out in a "clean" format needed by your subsequent analyses

`my_functions.R` Script containing the functions you have self-defined to perform your analyses

`analysis.R` Script that sucsk in your "clean" data, performs your analyses (using built-in, self-defined and package functions), makes some quick-and-dirty figures along the way, and exports the results to `output`

`final_figs.R` Script that sucks in the results of the analyses to produce the final figures (or tables) for your manuscript

In general, you don't want any file to become unwieldy. Thus you will likely have multiple scripts within each of the above categories (e.g., one for each of several different data set types, one for each of several different analyses or analysis steps, or one for each of several different self-defined functions) – each appropriately named (see Code style guides). If you end up with a whole lot of scripts to perform an analysis from start to finish, then you may want one additional `RunMe.R` script that sources each of the other scripts in the appropriate order.

The key utilities of separating out everything as specified above are (1) *readability* and (2) *unit-testing*. Readability means that it's easy for anyone (including you in 1 years time) to figure out where things are being pulled from and where they're going, and no individual script is overwhelming to look through. The idea behind unit-testing is to write independent tests for the smallest units of your code. For example, whenever you write a function, you also write an associated small piece of code that tests it (before the function is used in your primary analysis). That way, whenever you modify your code or function(s), it's easy to ensure that everything is still returning the correct output.

Working directory and directory access

In order to employ the principle of modularity you need to know how to navigate between your various folders and scripts. Within a project, for example, you have a choice between using the repository folder as your working directory or setting your working directory to be the directory in which you have saved your scripts (e.g., your RunMe script). Your scripts will therefore

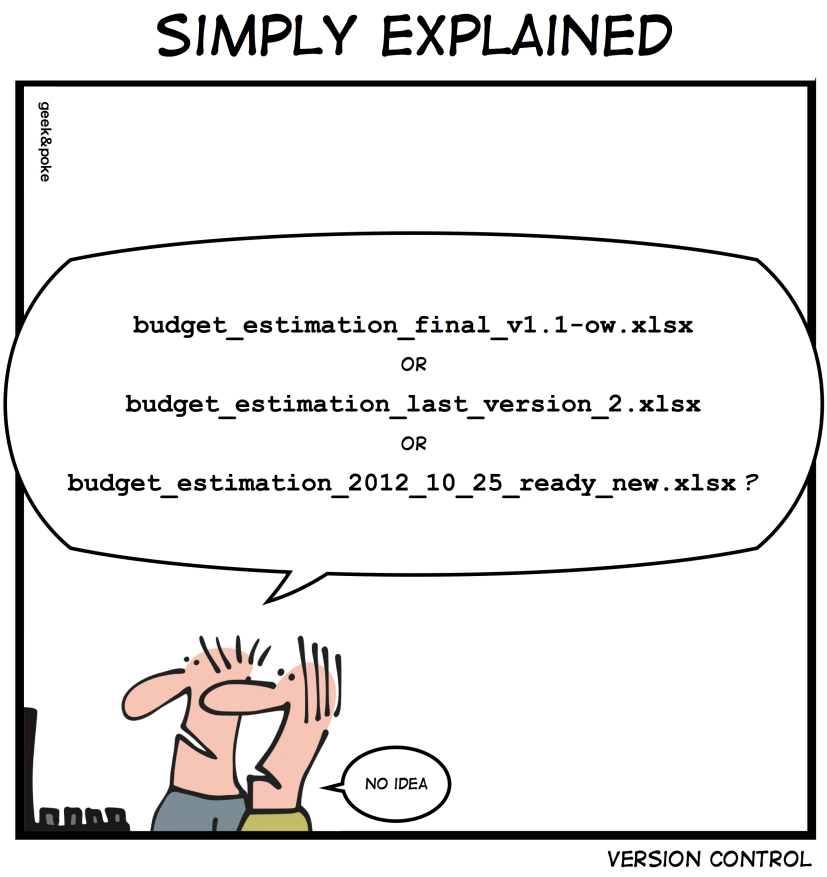

Figure 1: Never again should your data files look like this!

need to access data and source scripts that are located in other folders. You want to set these locations as generically as possible (so that you can move your entire project to a different location without destroying the workflow, for example). You never want to set the working directory more than once, or hard-write the locations of your data or other scripts within a given script.

You can do this by specifying locations relationally using '`/.../file.R`', for example. Each repetition of '`/.../`' will move you up one level in the folder structure. For example, assuming your working directory is the `code` directory, rather than specifying the location of your data as

`read.csv(file='C://MyDocuments/Git/MyProject/data/data.csv')`

you should instead use

`read.csv(file='../data/data.csv')`.

The latter takes you up one level (out of `code` into your general `repository` folder) then into the `data` folder and to your data file. To pull in scripts, use

`source(file='my_functions.R')`

(assuming your working directory is the `code` directory).

Note: Macs (Unix) and Windows (Dos) use forwardslashes and backslashes differently. Macs use '`/.../.../file.R`' (forwardslashes) while Windows use '`\...\...\file.R`' (backslashes).

Code style guides

There are a lot of summarized sets of recommended best-practices for coding in general and for `R` as well. These includes aspects relating to object naming conventions (for filenames, function names, and variable names), syntax and grammar (spacing, indentation, etc.), and code structure. I won't repeat everything here, but we will go over and discuss the big ones in class. For our course, the **required reading** is:

https://google.github.io/styleguide/Rguide.xml

Other guides that are well worth reading for some additional suggestions and explanations are:

https://www.r-bloggers.com/r-code-best-practices/

and

http://adv-r.had.co.nz/Style.html

Within-script organization

In additional the best practices in the naming of objects and the syntax of your code, there's also an important aspect of within-code organization. Your code should consist of the following sections, each visually separated from the others:

1. Start each file with a preface that describes what it contains and how it fits into the project. You might also want to include who wrote it and when.
2. Load all required packages
3. Source required scripts (e.g., `my_functions.R`)
4. Load (or source) required data (or `data_prep.R` scripts)
5. Section(s) for major parts of your analyses
6. Export results section(s)

The last two may consist of just two sections or may have export parts immediately following an analysis. Wherever possible and appropriate, clear your workspace (`rm(list=ls())`). For example, you'd most definitely do this at the top of your `RunMe.R` script and perhaps to at the top of your `analysis.R` script (since your "cleaned" data was saved and can be reloaded, leaving all the temporary variables unnecessary), but you wouldn't put `rm(list=ls())` at the head of your `my_functions.R` script.

Thus a script file might look as follows:

```
1 ##########################################################
2 # simulateLV.R
3 # Simulates the dynamics of a predator and prey
     population according to the Lotka-Volterra model.
4 # The data produced will subsequently be used to test
     the performance of several population dynamic model-
     fitting routines.
5 ##########################################################
6 rm(list=ls()) # clear workspace
7
```

```
################################
# Load librairies
################################
library(deSolve)

################################
# Source files
################################
# None needed

################################
# Define model
################################
LVmod <- function(Time, State, Pars) {
   with(as.list(c(State, Pars)), {
      Ingestion    <- rIng  * Prey * Predator
      GrowthPrey   <- rGrow * Prey * (1 - Prey/K)
      MortPredator <- rMort * Predator

      dPrey        <- GrowthPrey - Ingestion
      dPredator    <- Ingestion * assEff - MortPredator

      return(list(c(dPrey, dPredator)))
      })
   }

################################
# Define parameters
################################
pars <- c( rIng   = 0.2,    # /day, rate of ingestion
           rGrow  = 1.0,    # /day, prey growth rate
           rMort  = 0.2 ,   # /day, predator mortality
           assEff = 0.5,    # assimilation efficiency
           K      = 10)     # mmol/m3, carrying capacity

################################
# Simulate model
################################
yini  <- c(Prey = 1, Predator = 2)
times <- seq(0, 200, by = 1)
```

```
48 out   <- ode(yini, times, LVmod, pars)
49 summary(out)
50
51 ##############################
52 # Plot and export data
53 ##############################
54 plot(out)
55 write.csv(out, file='.../output/LV_out.csv')
56
57 ############################################################
```

When defining a function, you should provide additional information that describes what the function does (in general terms), what inputs it takes, and what output it returns. After the function is defined it's worth adding a test case (commented out). [Note: some would separate out test cases, leaving only the function definition in its own script.] For example:

```
1  ############################################################
2  # Function to add stochastic noise to a time-series of
      population sizes.
3  ############################################################
4  # Input:
5  #  x -- a time series of population sizes (vector)
6  #  error_model -- gaussian (currently implemented model,
       default)
7  #  sd -- the standard deviation of gaussian errors (
      default=0)
8  # Returns:
9  #  Vector of length equal to the input vector of time
      series
10
11 add_error <- function(x, error_model='gaussian', sd=0){
12    error_model <- match.arg(error_model)
13    if(error_model=='gaussian'){
14       out <- x + rnorm(length(x), sd)
15    }else{
16       out <- x
17       warning(''Original time series returned'')
```

```
18      }
19      return(out)
20  }
21
22  # Test:
23  # source('simulateLV.R')
24  # new <- add.error(out, sd=1)
25  # par(mfrow=c(1,2))
26  #  plot(out)
27  #  plot(new)
```

When writing code, though shalt not...

- Copy-paste-edit (i.e. reuse and modify) the same code to repeatedly apply it to different subsets of data, etc.. Instead, you will convert your code to function(s) that can be applied to these data subsets
- Use `attach()` and `detach()` on your data. Instead, be explicit in naming and accessing data.frame columns.
- Repeatedly reset your working directory. Instead, use relational sourcing.
- Save your R workspace for later reuse. Instead, use `rm(list=ls())` wherever possible and appropriate.
- create large tables by hand. If you find yourself having to export large tables a lot, learn LaTeXand export them instead