Carpentries - R Programming - Best Practices
November 22, 2021

Much of what I will cover in this session comes under the heading of 'common sense'. So please excuse me if what I say is something you might already have thought of. I proceed following the idea that it's better to make things explicit rather than perhaps suffering later for not having spelt them out.

1. Begin an R script with some descriptor of what the code is supposed to do. Comments in R begin with the sharp sign, #.

This code defines a function to perform a rather routine mathematical operation which is essential in numerical operations. Can you quickly tell what it does?

```
mod<-function(a,n){
      return(a-floor(a/n)*n)
                              }
```

It isn't readily apparent but if we preface it with a comment, it is.

```
# The function mod finds the residue class of
# integer a modulo positive integer n.
# That is, it finds the nonnegative remainder left
# when a is divided by n.

mod<-function(a,n){
      return(a-floor(a/n)*n)
                       }
```

2. Many useful scripts you'll create will entail the use of packages which are not automatically loaded in a standard R session. It is useful to make explicit which such packages are used in the script by placing calls to them at the beginning of the script.

```
library(ggplot2)
library(reticulate)
library(rhandsontable)
```

Doing this allows subsequent users of your script to avoid unpleasant surprises by making certain in advance that all packages required to execute the script have been installed.

3. Somewhat along the same lines, it is useful to avoid calls to explicit data files inside of the script so that the script can be as flexible as possible. One way to do this is to use variable names when constructing the script in place of particular file names so that the script can be deployed more generally.

Thus we avoid something like

```
my_data<-read.csv('my_data.csv')
thingy<-somefunction(my_data)
write(thingy, 'my_fixed_data.csv')
```

in favor of

```
inputfile_name<-'my_data.csv'
outputfile_name<-my_fixed_data.csv'
my_data<-read.csv(file_name)
thingy<-somefunction(my_data)
write(thingy, outputfile_name)
```

4. The working directory can be a slippery thing. Ideally the R script you write will go on to be used repeated by you, perhaps by your collaborators. It might even find its way into an R package on CRAN. Therefore it is less than ideal for you to use the command

```
setwd()
```

within a script. If the script calls data files or R source files (other scripts), those things need to be in the current working directory when the script is run, otherwise it will produce an error. Any other user of your script, or you yourself in some subsequent iteration of you, might be using the script inside of an entirely different directory structure than that which applied when you wrote it.

If you need to use a call to setwd() in your script, as with libraries it's best to make that explicit by doing so at the outset.

5. Modularize your code within a script and between scripts. Recall the function 'mod' which I defined earlier. It can be used to construct the greatest common divisior (gcd) of two integers. That might look like this:

```
mod<-function(a,n){
       return(a-floor(a/n)*n)
                                  }
gcd<-function(a,b){
        if (a==0 | b==0)
              {return(max(abs(a), abs(b)))}
        else {
           a<-abs(a)
           b<-abs(b)
           while (b>0){
                r<-mod(a,b)
                a<-b
                b<-r}
               }
         return(a)}
```

However, a friendlier version of this would break the pieces of code up into pieces seperated by helpful comments:

```
# The function 'mod' finds the residue class of
# integer a modulo positive integer n.

mod<-function(a,n){
       return(a-floor(a/n)*n)
                                  }

# The function 'gcd' uses the 'mod' function together
# with the Euclidean Algorithm to find the greatest
# common divisor of integers a and b.

gcd<-function(a,b){
        if (a==0 | b==0)
              {return(max(abs(a), abs(b)))}
        else {
           a<-abs(a)
           b<-abs(b)
           while (b>0){
                r<-mod(a,b)
                a<-b
```

```
        b<-r}
      }
return(a)}
```

Along the same lines, scripts become unwieldy when they contain too many functions. Better to divide the code up into multiple scripts organized by some unifying theme for each. For instance, lately (the last few years anyhow) I've been writing a lot of Shiny applications which involve mathematics and graphics. I use graphics functions which I build using the basic graphics package and mathematical functions which I also build to form the 'guts' of the application. I put the graphics functions and the math functions in seperate source scripts.

7. It helps to put all relevant data files, scripts for a given project in a single directory, perhaps with subdirectories. This allows easier calls to components using relative file paths (assuming one is in the correct working directory) rather than lengthy absolute ones:

```
source(\graphics\'elliptic.r')
```

is preferable to

```
source('\Users\rcg6824\Docs\math\Shiny\ElCrvs\graphics\'elliptic.r')
```