Maggie's Golden Rules of (R) Programming

Margaret Swift

9/24/2019

INTRO

Thought I'd put together a list of my top golden rules for programming. These are things that I have learned over the years that have made me much less stressed about coding:

I. Structure is Key
II. Pick a naming convention and stick with it
III. Assume the user is an idiot
IV. Work from the inside out
V. Simple is Best
VI. DRY – Don't Repeat Yourself

I. Structure is Key

I cannot stress this one enough. Please don't just dump your files onto your desktop. Folders and subfolders dedicated to a single project will be easier to manage and keep you less stressed. Here's an example of how I'd structure my folders for this class:

```
# - Fall2019
#   - ENV710
#     - Lab1
#         mississippi.csv
#         CountryCO2.csv
#         Lab1.rmd
#         Lab1.pdf
#     + Lab2
#     + Lab3
```

This goes for the structure of your .R file itself—keep things organized into sections, indent within curly braces `{}`, separate subsections with whitespace, and add lots of helpful `#comments`. For example, here's a simple script I wrote for my research that creates a table of species and indicates whether each species is included in each year's survey:

```
# EAS_SpeciesPresence.R
# Created 09/22/19
# Author: Margaret Swift
# margaret.swift@duke.edu

# Script to create a table of number of observations per
# species, per year.

#---------------------------------------------
# Clean ws, set wd, get utility files. I keep all of my
# functions and important variables in utility files so that I
# can use them in different places.

rm(list=ls())
dir <- dirname(rstudioapi::getSourceEditorContext()$path) # this gets the current filepath
setwd(dir)
```

```r
utilfile <- paste(gsub('/Kruger.*', '', dir), 'Kruger/KNP_Utilities.R', sep="/")
source(utilfile)
source('EAS_Utilities.R')

#-----------------------------------------
# Pull species data file name, read it into a data
# frame, and get a list of species and years represented.
file <- list.files(path.r)[which(grepl('EAS_ungulates', list.files(path.r)))]
all.df <- read.table(paste(path.r, file, sep="/"), header=TRUE)
sp <- unique(all.df$Sp)
years = unique(all.df$Year)

# Create a data frame to hold species and years.
years.df <- data.frame(matrix(0, nrow=length(years), ncol=length(sp)))
row.names(years.df) <- years
names(years.df) <- sp

# Fill data frame with species presence/absence data
# (I know I talk later about being careful with loops, but I know my
# dataset isn't that large, and I only needed this script once. :) )
for (i in 1:length(years)) {
  # Loop over each year and grab the rows for that year.
  year <- as.character(years[i])
  df <- all.df[all.df$Year==year, ]
  for (j in 1:length(sp)) {
    # Loop over each species in our species list and find
    # the number of observations per species for this year.
    species <- as.character(sp[j])
    inx <- which(df$Sp==species)
    years.df[i,j] <- length(inx)
  }
}

# Write data to CSV to save it for later.
filename <- paste(path.r, "yearly_species_observations.csv", sep="/")
write.csv(years.df, filename)
```

II. Pick a naming convention and stick with it

Using `camelCaseNames` for your variables? `scores_of_underscores`? `lots.of.dots`? Doesn't matter, just pick something and stick with it so you don't confuse yourself later. I like to include the type of variable in the name, with suffixes like `.df` for data frames, and `.ls` for lists. So if I'm using a dataset called `epa`, I might have a data frame `epa.df` or a plot `epa.plot`.

While we're here, make sure your names are descriptive and consistent, for files and variables both. No more 'asdf.doc' or `whatever <- length(x)`. Finally, don't use things like `for` or `TRUE` for your variable names. These are reserved by R for special cases.

III. Assume the user is an idiot

When writing functions, be sure to account for user error. Don't assume that the user (who may well be your future self!) remembered to sort the data before passing it to the function—sort it for them upon arrival. Don't assume that vector 1 and vector 2 will be the same length—compare the two and print an error with `{r}warning()` if they don't match:

```
plot.vecs.func <- function( x, y ) {
  if ( length (x) != length(y) ) warning('WARNING: Samples have different length!')
  else plot(x, y)
}
plot.vecs.func(c(1, 2, 3), c(5, 6))
```

```
## Warning in plot.vecs.func(c(1, 2, 3), c(5, 6)): WARNING: Samples have
## different length!
```

IV. Work from the inside out

When trying to subset a data frame by all values that are larger than the mean, don't try to write it all in one go. Start from the inside and work outward:

```
mean( data$mycol )
data$mycol > mean( data$mycol )
which( data$mycol > mean( data$mycol ) )
data[ , which( data$mycol > mean( data$mycol ) ) ]
```

Better yet, assign some of these values to variables to make your life less complicated:

```
mu <- mean( data$mycol )
inx <- which( data$mycol > mu )
data[ , inx ]
```

This is really important for functions and loops. Start with the simple case and work your way out to the wrapper, when writing and troubleshooting. Instead of just running a loop and having it break, set `i=1` on your own and run through the loop line-by-line to see where the error is.

V. Simple is best

A lot of times, we're tempted to use loops because they're easy or they're what we're used to. But if something can be done without a loop, it should be. This is because loops take up a lot of time and memory, especially when running over a huge dataset. So, for example, if I wanted to create a vector `p` filled with percentiles, I could use `for` loops, or I could use vector manipulation (ie `1:100`). Run the following code to see for yourself:

```
n <- 500000

# for loop way:
p1 <- vector(length=n)
end1 <- system.time(
  for ( i in 1:n ) {
    p1[i] <- ( i - 0.5 ) / n
  }
)['elapsed']

# vector manipulation way:
end2 <- system.time(
  p2 <- (1:n - 0.5) / n
)['elapsed']

cat('for loop: ', end1, 'sec.\n', 'vector: ', end2, 'sec.')
```

```
## for loop:  0.046 sec.
##  vector:  0.003 sec.
```

As you can see, the `for` loop option is *much* slower than using ranges, despite running the same calculation. Nested loops compound this. `for` loops are useful, yes, but be careful that you know their weaknesses.

VI. DRY — Don't Repeat Yourself

All this said, one of the most important rules I've learned is — Don't Repeat Yourself. If you're copying and pasting code from one place to another, you're not being very efficient. Whenever possible, use vector manipulation, loops, or functions like `sapply` to accomplish a task multiple times without repeating yourself. Something that my dad always liked to tell me about skiing seems appropriate here–the goal of the thing is to *do no work*. Think of programming as trying to be as lazy as possible–don't copy and paste when a better method is availab le.

This also goes for creating utility files for larger projects. If you find yourself writing and re-writing a function to find a certain file, or to plot a certain graph, in every file of your project — it may be well worth it to put all of your functions into one big 'Utilities.R' file that is accessed by every other script, using `source('Utilities.R')`. I do this for all of my projects.