R programming style and functions

Everyone has a unique way of thinking about programmatic problems. While this can lead to many great solutions it becomes problematic if a unique way of thinking becomes a unique way of program annotation and a nightmare for someone else to look at, reproduce or debug.

In order to have as much consistency across the board it is recommended that all programmers follow a language specific style. This helps with legibility and orienting yourself within your own code after not having used it for a few months.

A good guide is Google's recommended style for R, based off of the tidyverse style guide. Their guide also covers a wide variety of other languages and I highly suggest you at least skim it before starting to program. It will make your life easier in so many different ways.

Below are some recommendations. Style is of course entirely up to you but I highly recommend that you decide on one early and then stick to it for consistency.

```
# Assignments
x <- 5  # good
x = 5  # bad

# Variables
variable_name <- c()  # preferred
variable.name <- c()   # ok
variableName <- c()  # bad

# Functions
function_name <- function(){} # good
FunctionName <- function(){} # bad
functionName <- function(){}  # bad

# Spacing
tab.prior <- table(df[df$days.from.opt < 0, "campaign.id"])  # good
tab.prior <- table(df[df$days.from.opt<0,"campaign.id"])  # bad

# if - else setup
#################
# good
if (condition) {
  #one or more lines
} else {
  #one or more lines
}

#################
# bad
if (condition) {
  #one or more lines
} else
    {
```

```
  #one or more lines
}

# Long lines of code
do_something_very_complicated(
  something <- "that",
  requires <- many,
  arguments <- "some of which may be long"
)

# Bad
do_something_very_complicated("that", "requires", "many", "arguments",
                              "some of which may be long"
                              )
```

Lastly, make sure you always comment your code. This is emphasized in every course but ignored by many students until the day it comes around to bite you. Set your habits from the beginning and stick to them. It will take an extra 20s at the benefit of avoiding hours of debugging. So help you, help YOU.

Example from (roxygen)[https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html], a documentation helper package for R.

```
# Add together two numbers.
#
# @param x A number.
# @param y A number.
# @return The sum of x and y.
# @examples
# add(1, 1)
# add(10, 1)

add <- function(x, y) {
  return(x + y)
}

add(1, 1)
```

R Functions

Functions are used to link several operations which can be repeated by typing just one command instead of having to re-type the whole thing. If you find yourself doing something over and over again it should probably go into a function...

Functions help you clean up your code and make things more organized. Better organized means both less mistakes and it“s easier to catch mistakes.

The best thing about functions it that once you create and adequately test your function, you can continue to re-use it from analysis to analysis!

Structure:

```
# function documentation
my_function <- function(my_input_argument) {
  # some operation (s)
  return(output)
}
```

Yesterday we copied code over and over to generate plots, this is a sign that it should be wrapped in a function.

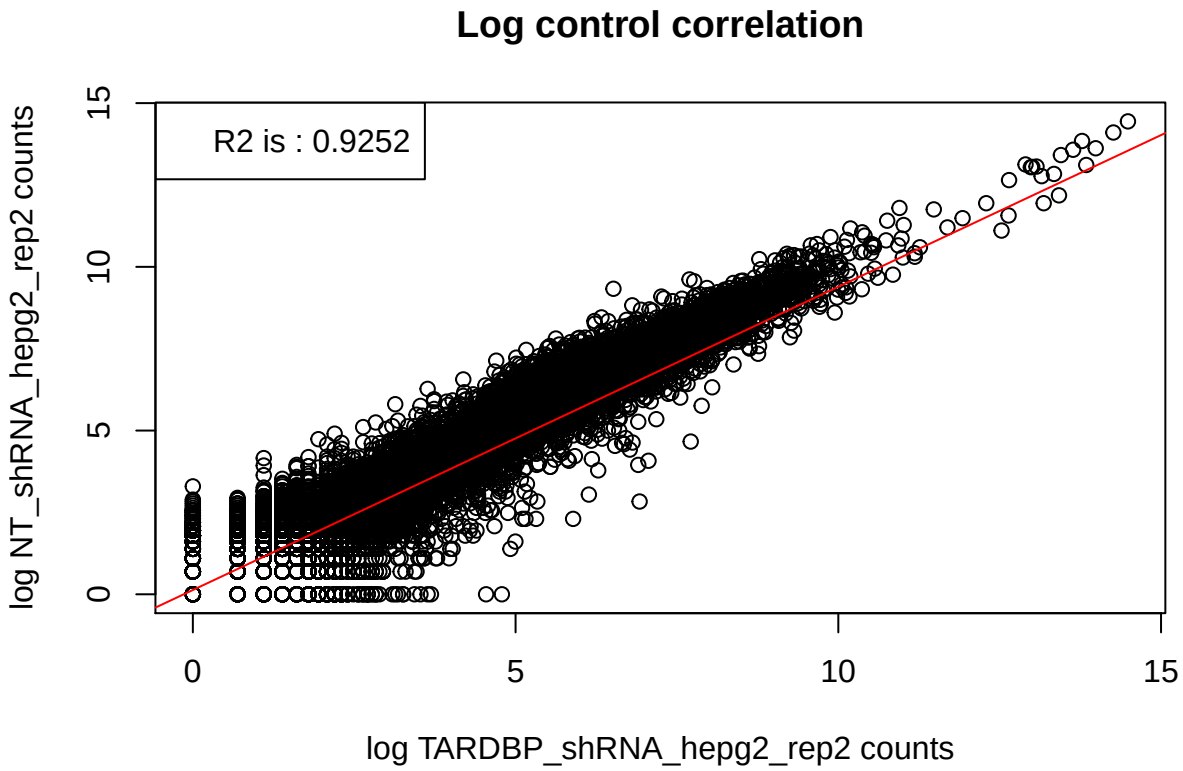
```
# Let's create a function to plot a correlation between two samples, performs a linear regeression betw

# reading in data
counts <- read.csv("~/work/cmm262-2020/Module_1/Data/tardbp_counts_only.csv",
                   header = TRUE,
                   row.names = 1,
                   stringsAsFactors = FALSE
                   )
```

We would like to create a function which takes a data frame of counts, and two column names then output a scatter plot and the result of a linear regression.

How would we fill in this function?

```
# Plot correlation between two samples.
#
# @param data A data frame.
# @param sample_one A column name of data as character.
# @param sample_two A column name of data as character.
# @return Summary of lm() between sample_one and sample_two.
# @examples
# plot_count_correlation(counts,
#                        "TARDBP_shRNA_hepg2_rep2",
#                        "NT_shRNA_hepg2_rep2")


plot_count_correlation <- function(data,
                                   sample_one,
                                   sample_two) {

  # fit a linear model to log transformed counts data
  fit <- lm(log(data[[sample_one]] + 1) ~ log(data[[sample_two]] + 1))

  # generate a scatterplot of counts of sample 1 vs sample 2
  plot(x = log(data[[sample_one]] + 1),
       y = log(data[[sample_two]] + 1 ),
       main = 'Log control correlation',
       xlab = paste('log', sample_one, 'counts'),
       ylab = paste('log', sample_two, 'counts')
       )

  # plot the line from the linear regression on same plot
  abline(fit, col = 'red')

  # display the adjusted r-squared value on the plot
  legend('topleft',
       legend = paste('R2 is :', format(summary(fit)$adj.r.squared,digits = 4))
       )

  # save summary statistics from linear model to a variable
  s <- summary(fit)

  return(s)
```

```
}

linear_reg_stats <- plot_count_correlation(counts,
                                          "TARDBP_shRNA_hepg2_rep2",
                                          "NT_shRNA_hepg2_rep2")
```



If we were curious about the implementation of a function we can always view details by simply entering it in the interface without braces

```
plot_count_correlation
```

```
## function(data,
##                                     sample_one,
##                                     sample_two) {
##
##   # fit a linear model to log transformed counts data
##   fit <- lm(log(data[[sample_one]] + 1) ~ log(data[[sample_two]] + 1))
##
##   # generate a scatterplot of counts of sample 1 vs sample 2
##   plot(x = log(data[[sample_one]] + 1),
##        y = log(data[[sample_two]] + 1 ),
##        main = 'Log control correlation',
##        xlab = paste('log', sample_one, 'counts'),
##        ylab = paste('log', sample_two, 'counts')
##        )
##
##   # plot the line from the linear regression on same plot
##   abline(fit, col = 'red')
##
##   # display the adjusted r-squared value on the plot
```

```
##   legend('topleft',
##        legend = paste('R2 is :', format(summary(fit)$adj.r.squared,digits = 4))
##        )
##
##   # save summary statistics from linear model to a variable
##   s <- summary(fit)
##
##   return(s)
## }
## <bytecode: 0x7f9745881be0>
```

Lexical scoping in R: a few things to be aware of

```
# what will the output below return? An error or something else
x <- 2
g <- function(){
    y <- 1
    c(x, y)
}
output <- g()
output
```

```
## [1] 2 1
```

```
# try predicting the output below
x <- 2
g <- function(){
    x <- 5
    x
}
x <- g()
x
```

```
## [1] 5
```

There is an additional set of functions from the "apply" family which are commonly used. As the name suggests they apply a function over and over again and are use as and alternative to for loops.

```
# using the "apply" function
# Construct a 5x6 matrix

x <- matrix(1:30, nrow = 5, ncol = 6)
head(x)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    6   11   16   21   26
## [2,]    2    7   12   17   22   27
## [3,]    3    8   13   18   23   28
## [4,]    4    9   14   19   24   29
## [5,]    5   10   15   20   25   30
```

```
# Sum the values of each column with a for loop

col_sum <- c()
for(i in 1:ncol(x)){
  col_sum <- c(col_sum, sum(x[,i]))
}
col_sum
```

```
## [1]  15  40  65  90 115 140
```

```
# Sum the values of each column with `apply()`
# apply(matrix, 1 for rows or 2 for columns,  function)

col_sum_two <- apply(x, 2, sum)
col_sum_two
```

```
## [1]  15  40  65  90 115 140
```

```
# lapply goes over lists instead:
my_list <- list(first = c(1:10),
                second = c(10:20),
                third = c(100:1000))

# the output is again in list format
lapply(my_list, sum)
```

```
## $first
## [1] 55
##
## $second
## [1] 165
##
## $third
## [1] 495550
```

```
# use sapply to have the ouput in vector format
sapply(my_list, sum)
```

```
##  first second  third
##     55    165 495550
```

R debugging

The majority of the time spent on a programming assignment will be in the realm of debugging. Most commonly this is due to miscommunication between code and user. One way to minimize the number of error made is by writing clear code, having meaningful variable and function names, and testing each step separately. It is generally considred bad practice to write an entire program or pipeline of potentially a few hundered lines without testing the individual components. Usually it won't work the first time around. If you incrementally test different sections (compartmentalize) you'll be able to move a lot faster.

The main debugging functions in R are: browser(), debug(), traceback(), recover(). Here we will only take a look at the first two but I recommend you take the time to read up on the other two as well so you know in what situation which debugging tool will prove most useful.

Note: debugging in notebooks does not really work so use an interactive R session instead

browser(): This stops wherever you set the browser() option and allows you to poke around. This is useful if you already have an idea where the error might be sitting.

```
browser_function_example <- function(x){
    browser()
    a  <- 2
    y <- x + a
    return(y)
}
```

```
browser_function_example(4)
```

```
## Called from: browser_function_example(4)
## debug at <text>#4: a <- 2
## debug at <text>#5: y <- x + a
## debug at <text>#6: return(y)
```

```
## [1] 6
```

See additional debugging support from RStudio.

Useful R tricks

Loading R libraries: Every time you load a library you're adding a new environment to R. The library specific functions become available because R searches all environments for your function call. Sometimes you override an existing function by loading a new library. You can reference the package-specific function via the package specifier.

```
# example for loading a library and calling a library-specific function.

library(ggplot2)
ggplot2::label_value()
```

Installing libraries:

```
# General guide for installing packages in R
# 1. Install standard packages within R:
install.packages("whatever")
install.packages("MESS", dependencies=TRUE, repos="http://cran.rstudio.com/")

# 2. Install Bioconductor packages within R
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("DESeq2")

# 3. If all else fails; use sledgehammer approach and install from source:
#  download tarball (.tar) or gzipped tarball (.tar.gz)
#  install.packages("edgeR_3.18.1.tar.gz",repos=NULL,type="source")
```

Directory switching:

```
# current location

getwd()

#change directory; the directory change is the same format as with unix but has to be given as string
setwd("../../")
```

Adding your own scripts: To add a file containing functions or a workflow use the source() function. The script you"re sourcing can contain functions, variable assignments, loading packages etc.

```
source("my_script.r")
```

Timing functions: When dealing with large amounts of data certain functions can become a bottleneck. To get an idea what running time to expect and whether you might have to re-write a function you can time how long it takes to run on a subset, or a few instances.

```r
ptm <- proc.time() # record the current time
ptm
```

```
##    user  system elapsed
##   2.262   0.353   3.100
```

```r
h <- 5 + 1  #function to be performed/timed
proc.time() - ptm # obtain the time difference
```

```
##    user  system elapsed
##   0.004   0.001   0.004
```

```r
# Add additional function timing resources
```