Writing Functions

Preamble

```r
rm(list=ls())
setwd("C:/Users/19107/Desktop/R Stuff")
```

Today's topic in R is writing functions, functions are just user-defined commands in R. Sometimes we might want to do something that R doesn't have a clear-cut command for or the command it has is inefficient or cumbersome. All functions have at least three components: a name, a set of arguments, and a process. The name serves an obvious purpose, we need a name to assign something and it helps if the name is informative though it certainly doesn't have to be. The arguments, as the R jargon doc will tell us, are the inputs and options that the function will use; some arguments are optional but for now we will only deal with functions who take REQUIRED arguments. Lastly, the process, this is what we want our function to do to our arguments. I'll give a few examples below:

Example 1: mean()

mean() is a base R function, so this is not a user-defined command but rather a built in one. Yet it still has the three components we defined above: name, arguments, and process.

The name is of course "mean". It takes 1 argument (for our purposes) which is defined as x (you can see this by hovering over mean() with your mouse, it will say mean(x,...) and a short description of what it does.) The process of mean() is to calculate the sum of all values in x, divide that sum by the total number of elements in x, and return that number i.e. the mean.

Let's use mean and then test to see if it actually does the process we described above

```r
# Create an object, vector, which is a vector/list of numeric values.
vector= c(1,2,3,1000,2,-19,.45, -854, sqrt(15))
```

```r
# calculate the mean using mean()
mean(vector)
```

```
[1] 15.48033
```

```r
# Create an object that imitates the process we outlined above
test=(sum(vector))/(length(vector))

# Test for equivalence between mean()'s output and our test object
mean(vector)==test
```

```
[1] TRUE
```

```r
# We've confirmed mean() does in fact follow the process above
```

Example 2: view vs View

In recitation I recounted to you my frustration with case-sensitive functions. Base R includes a command View() (note the uppercase V), this command is very simple, it takes 1 argument and allows the user to view it like you would view a spreadsheet, or Stata's data-viewer. This is a very useful command for visually inspecting your data and looking for issues. Unfortunately, the case sensitivity of the command means that it is easy make mistakes when working quickly. Well one day I decided no more, I would write a function of my own WITH A LOWERCASE V.

Writing a function yourself is a lot like simple assignment except with more steps. Like earlier we said functions have three parts: name, arguments, and process. so the structure should be like this

[some name] = function([argument(s)]){

[some process]

}

See below.

First, we need something to view,

```
data = data.frame(
  variable1 = seq(1,25),
  variable2 = seq(60,84)
)
 # Let's add a third variable for fun

data$variable3 = data$variable1/data$variable2

# Let's try and use view/View to demonstrate the problem
view(data)
```

```
Error in view(data): could not find function "view"
```

```
View(data)

 # No error! but I don't want to hit that shift key, it's about the
  ↪ principle

view  = function(x){
  # the name is view (lowercase), it takes 1 argument, "x",

  View(x)
  # and the process is simple, it takes whatever you put in, "x", and
   ↪ runs the View (capital v) command on it, effectively neutralizing
   ↪ the possibility for case-related error.

}

# Let's take it out for a spin.
view(data)
# No errors. We've done it! *side-note: the people who wrote the
 ↪ "tibble" package also had a similar gripe and their package does
 ↪ include a "view" function as well*
```

Example 3: Mode

Even tasks that seem simple at first glance can be quite complex, for example defining a mode function. I pulled the original version of this from StackOverflow. Seeking help from the internet is often essential, just make sure you can understand the solutions you get your hands on, otherwise they're useless to you.

```r
getmode <- function(v) {
# the name is "getmode", it takes one argument, here defined as "v".

  v=subset(v,is.na(v)==FALSE)
  # I added this line to remove NA values since they gum up the works
  # what line the above line does is it defines an object, v (though any
   ↪  name can be used), which is a subset of the initial input v. The
   ↪  way subset works is it uses a logical argument (here
   ↪  "is.na(v)==FALSE") to decide what elements of the initial input v
   ↪  make it into the created object v. Only those that satify the
   ↪  logical argument make it in.

  uniqv <- unique(v)
  # this line creates a list of the unique values of the object v

  uniqv[which.max(tabulate(match(v, uniqv)))]
  # this line is dense but we'll work our way from the innermost part to
   ↪  the outermost part.
  # First we use a match command to find all the matches between the
   ↪  object v and the object uniqv. Then we use tabulate to tally up
   ↪  all the matches from the match() command Next, which.max() returns
   ↪  the LOCATION of whichever row has the highest tally from tabulate.
   ↪  Finally, we put the whole thing in brackets next to uniqv which
   ↪  INDEXES uniqv using the row number generated by which.max
}
```

Test run

```r
x=c(1,2,3,4,5,6,7,7,8)
getmode(x)
```

```
[1] 7
```

```r
# success!
# To make sure we understand the intuition we'll go through each step
 ↪  and test that it does what we think it ought to. This is a very
 ↪  useful exercise when building complex functions.

# We'll create a real object to work with and test out the features, for
 ↪  ease it should match the intended name of the argument we're using
 ↪  for our function

v = c(1,2,3,4,5,5,5,5,7,7,NA,NA)

# does subset() remove NAs?
v=subset(v,is.na(v)==FALSE)
v
```

```
 [1] 1 2 3 4 5 5 5 5 7 7
```

```
# YES!

# Does unique() identify the unique values in v?
uniqv = unique(v)
uniqv
```

```
[1] 1 2 3 4 5 7
```

```
 # YES!

# Does match identify the location of matches between v and uniqv?
match(v, uniqv)
```

```
 [1] 1 2 3 4 5 5 5 5 6 6
```

```
 # YES!

# Does tabulate total the number of matches?
tabulate(match(v, uniqv))
```

```
[1] 1 1 1 1 4 2
```

```
 # YES!

# Does which.max tell us which row/element contains the highest number
 ↪  of matches
which.max(tabulate(match(v, uniqv)))
```

```
[1] 5
```

```
 # YES! it says the 5th unique value is the one with the most matches

 # Does indexing uniqv with either 5 or the set of commands that
  ↪  produces 5 give us the answer?
uniqv[which.max(tabulate(match(v, uniqv)))]
```

```
[1] 5
```

```
uniqv[5]
```

```
[1] 5
```

```
uniqv[which.max(tabulate(match(v, uniqv)))] == uniqv[5]
```

```
[1] TRUE
```

```
 # YES!
```

So we've actually gained a lot of understanding about not only how functions work & how to write our own, but also a whole set of previously unfamiliar commands. This is one of the best benefits of sharing code or finding code online, if your goal is learning you will be able to add a whole new set of tools to your coding toolbox, and before you know it you will fondly look back on your previous attempts and marvel at how your toolbox has grown.