A Teaching Script for Learning Functions

Genevieve Nuttall

LEARNING BASIC FUNCTIONS IN R

References:

link 1

link 2

link 3

link 4

Functions:

In R, functions are extremely important. Every command that you run is probably a set of functions that someone else created for you to use. Every package is jammed full of tons of functions that we are able to call with a simple command.

For example, when you type mean(x), R will give you back a number. The only reason it can do this is because the mean() command has been programmed to execute a specific function. Take a look:

```
# Let's look at what makes up the function to find variance:

var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x00000000144637c8>
## <environment: namespace:stats>
```

```
# If you run this, you will see the basic syntax of the function that runs when you run the
# var() command, where the function is stored, and what it is named as ("var" in this case).
# Pretty cool!
```

Basically, using and creating functions comes down to two questions:

1. What do we want the computer to do?
   - Tells us which function to use
2. What information does the computer need to compute this?
   - Tells us which arguments we should use

Here are some examples of built-in functions you may have used before in R:

```
##### Calculating Functions

sum(6:8)
```

```
## [1] 21
```

```
mean(1:10)
```

```
## [1] 5.5
```

```
log(16)
```

```
## [1] 2.772589
```

```
sqrt(36)
```

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

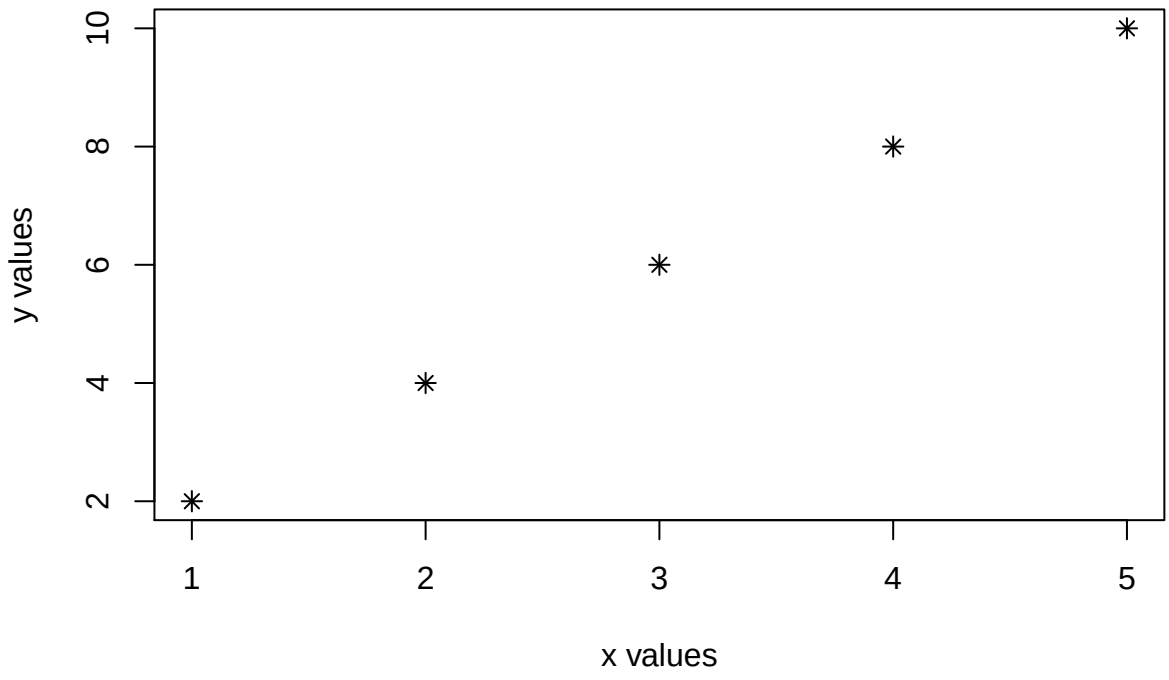
```
###### Plotting functions

## Plot info from a sample data set in R: trees
head(trees)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
## 5  10.7     81   18.8
## 6  10.8     83   19.7
```

```
hist(trees$Height,xlab="Tree Height")
```


```
## Plot your own graph
plot(x=c(1,2,3,4,5),y=c(2,4,6,8,10),xlab="x values",ylab="y values",pch=8)
```




```
###### Statistical functions

## t-test

z<-rnorm(15)
a<-rnorm(15)
t.test(z,a)
```

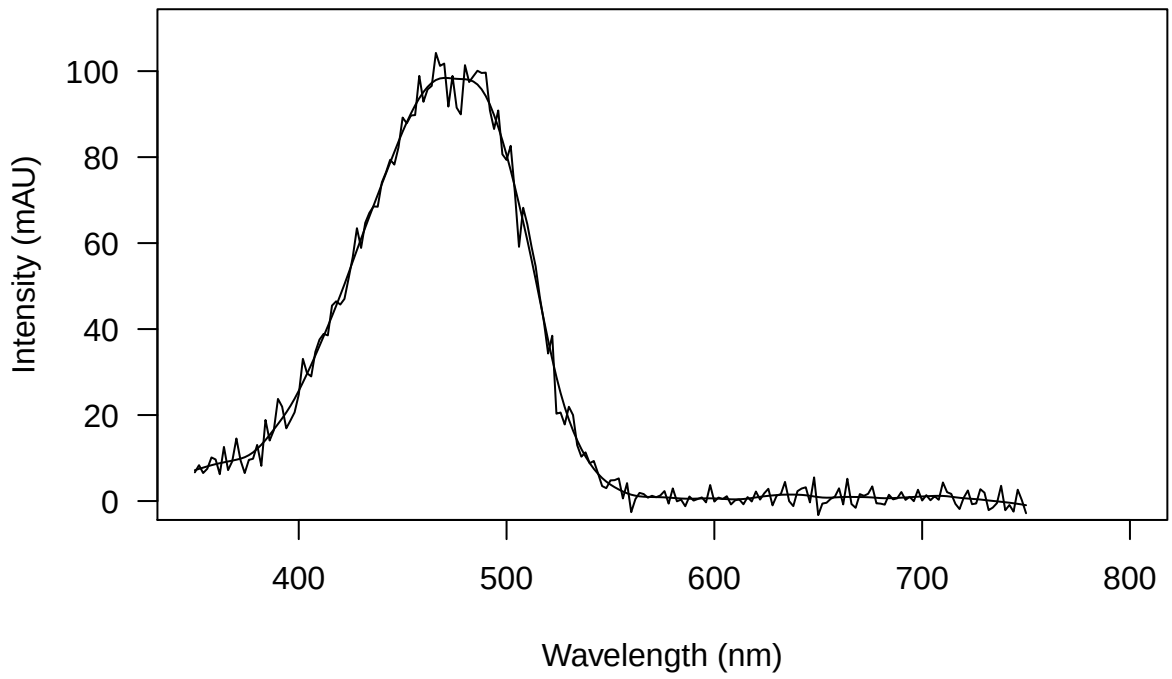
```
##
##  Welch Two Sample t-test
##
## data:  z and a
## t = -0.37545, df = 27.986, p-value = 0.7102
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.8544929  0.5897817
## sample estimates:
##  mean of x  mean of y
## -0.3727894 -0.2404338
```

```
# Look at the results - can you find the p value? Is there a significant difference?

## smooth.spline (fits a cubic smooth line to noisy curves) - one of my favorites!

peak1.standard<-read.csv("Peak1.csv")
# An example of VERY noisy data - unfortunately from my own data
plot(peak1.standard$Wavelength,peak1.standard$mAU,xlim=c(350,800), type="l", pch=1,cex=0.6,las=TRUE,
```

```
     ylab="Intensity (mAU)",ylim=c(0,110),xlab="Wavelength (nm)")
# Let's use the smooth.spline function to smooth this line out.
smooth.peak1.standard<-smooth.spline(peak1.standard$Wavelength, peak1.standard$mAU)
# Now plot the smooth line over the noisy line to see how well it fits
plot(peak1.standard$Wavelength,peak1.standard$mAU,xlim=c(350,800), type="l", pch=1,cex=0.6,las=TRUE,
     ylab="Intensity (mAU)",ylim=c(0,110),xlab="Wavelength (nm)")
lines(smooth.peak1.standard)
```




```
# Remember, these look like simple commands but they are actually functions
# that have been built into the software for us. Thanks, R!
```

Luckily, R isn't too selfish, and it lets you make functions too so let's give it a shot!

Writing functions!

This will be useful to you at some point when you need to make a specific function for data analysis.

In R, there is a basic format for writing a function:

function <- functionname(argument1,argument2){body of function}

There are three components to this command: 1. formal arguments (in parentheses) 2. body {in squiqqly brackets} 3. environment - where the function information is stored

When you run a function, R will always return the last statement of the command, which will be your {body}
Make sure you use the {} brackets when writing the body of a function

Try creating a basic function for practice:

```
fun_function<-function(x,y=5){x*y}

fun_function
```

```
## function(x,y=5){x*y}
```

```
# Let's look at the components (formal arguments, body, and environment) of this function:
formals(fun_function)
```

```
## $x
##
##
## $y
## [1] 5
```

```
body(fun_function)
```

```
## {
##     x * y
## }
```

```
environment(fun_function)
```

```
## <environment: R_GlobalEnv>
```

```
# Remember, every function is made up of these three basic components.

# Let's run this function now
fun_function(3)
```

```
## [1] 15
```

```
# This is telling us that our "x" value is 3, and we know that y=3 and the body is {x*y}
# So, the output should be 15 because {x*y} = {3*5}. Yay!
```

Now, let's try something more complicated.

```
absolutely<-function(x){if(x<0){-x}else{x}}

# This code tells us that if an x value is less than 1, then this negative value should
# be converted to positive
# AKA an absolute value function

# Try it!
absolutely(-7)
```

```
## [1] 7
```

```
absolutely(28)
```

```
## [1] 28
```

When will you need to write your own functions? *For loops* are good examples!!!

For loops are functions used to repeat a line of code. They are helpful if you need to cycle through a set of steps - instead of coding each repeated step, you can use a for loop to do it for you.

Incorporate the same idea of arguments and body that we learned before.

```
for (food in c("mac and cheese","watermelon","pizza","blueberries","rice krispie treat")){
print(paste("My favorite food is", food))}
```

```
## [1] "My favorite food is mac and cheese"
## [1] "My favorite food is watermelon"
## [1] "My favorite food is pizza"
## [1] "My favorite food is blueberries"
## [1] "My favorite food is rice krispie treat"
## Hopefully I covered one of your fav foods here, I'm really sorry if not.

## This is a silly example, but it shows you exactly what's going on with the loop.
## In the body, it's taking the text "My favorite food is" and linking it to the arguments
# specified as "food" arguments.
## Each line represents a new loop, and the function will continue until it runs out of arguments.
```

You want to start your for loop with for(), have your arguments be the list of the things that the loop should cycle through, and have your body be what you want done with these things.

Let's move on to a more scientific example:

```
# Let's make a vector:

vector1<-c(3,5,6,9,8,11,15,16,22,23)

# Next, set your inital condition to 0. Do you know why we should do this?

condition<-0

## Okay, now we're going to write a loop that counts the total number of odd values in our vector:

for(val in vector1){print(if (val %% 2) condition=condition+1)}
```

```
## [1] 1
## [1] 2
## NULL
## [1] 3
## NULL
## [1] 4
## [1] 5
## NULL
## NULL
## [1] 6
## Should spit back to you the odd numbers

# An alternative way of writing it where print(condition) tells you the total number of odd values

vector1<-c(3,5,6,9,8,11,15,16,22,23)
condition<-0

for(val in vector1){if (val %% 2) condition=condition+1}
print(condition)
```

```
## [1] 6
# In this loop, "val" corresponds to each values in our vector
# The body is telling R to exclude any values that are multiples of
# two in the vector (indicated by %%)
# The initial condition is 0, so each time the loop runs, it is cycling
```

```
# through the next value until all 8 values have been processed.
# The loop runs eight iterations (one of each value in our vector)
# For each value, it will return a TRUE or FALSE, and counts the number
# of TRUEs when we make the command print()
```

You can also create nested for loops which are loops inside of loops if you need multiple a function to run through multiple layers of cycles. An example:

```
# Let's make a 10 x 10 matrix and fill it using a nested for loop
matrix1 <- matrix(nrow=5, ncol=5)
matrix1 ## you can see it's empty right now
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   NA   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA   NA
## [5,]   NA   NA   NA   NA   NA
```

```
# This loop will assign values in each row/column based on the
# product of the i and j values that we specify here:
for(i in 1:dim(matrix1)[1]) {
  for(j in 1:dim(matrix1)[2]) {
    matrix1[i,j] = i*j}}

# Basically, this code tells R to put in a value in each
# cell that is the product of the corresponding product
# of row and column and this code is repeated until all cells are filled

# [1] is just an index that corresponds rows, [2] to columns

# It's "nested" because you have a for() command within another for() command

# Look at the matrix now:
matrix1
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
## [4,]    4    8   12   16   20
## [5,]    5   10   15   20   25
```

For loops are one type of loop. In R, there are three main types of loops:

- for loops
- while loops
- repeat loops

Depending on your needs, you might want to use a different type of loop. To learn more about how to write different loops, visit datacamp

In some cases, it might be easier to use an apply() function which can act as a shortcut for a loop.

> Remember, R is a world of functions. The ones we covered today are just a small percentage of what is out there on R.

To go over more commonly used fuctions, view this list of functions