Programming: Functions, Loops, If/Else Statements

R Workshop - 8/14/2018

Functions

An *AWESOME* feature of R, and perhaps a main reason for using it, is that it allows you to program your own functions. In fact, all the commands that you use in R *are* functions.

We can look at the basic structure of a function in R by typing the function name, but excluding the parentheses. For example, lets take a look at the function for taking the row mean of an object, `rowMeans()`. Just type:

```
rowMeans # There is a lot going on, but you see the structure.
```

The basic structure of a function is:
`function.name <- function`(arguments or things you need separated by commas)`{`
actions or stuff you want the function to be doing
`return`(information you want from the function)
`}`

Basically, we are passing in an object or objects, doing something to it, and then returning an output.

For example, let's create a function that calculates the mean of a variable:

```
function.mean <- function(input){
  our.mean <- sum(input)/length(input)
  return(our.mean)
}
```

This function takes an object `input` and divides the sum of input by the length of input and returns this ratio as an object called `our.mean`. Note that the names we use for the objects are arbitrary.

The function reads as: create a function called `function.mean`, this function takes one argument called `input`, the function will create an object called `our.mean` by summing input and dividing it by the length of input, the function then returns the object `our.mean`.

Remember that we can see our function by just typing the name `function.mean`.

Now, let's create an object called `x` and calculate the mean using our function:

```
x <- c(1,2,3,4,5)
function.mean(x)
```

```
## [1] 3
```

```
x.mean <- function.mean(x)
x.mean
```

```
## [1] 3
```

Note here that `x` is the input in the function. Also, note that we can use the function we created to make an object. The command: `function.mean(x)` just returns the mean, but the command: `x.mean <- function.mean(x)` creates the object `x.mean` that is the mean of the object `x`.

Commenting inside the function is helpful for keeping track of what is going on, as well as for others who might want to use a function you have created. In R, any line with a `#` sign in front is commented out of the code in the function and it will pass over it. For example:

```
function.mean <- function(input){
  our.mean <- sum(input)/length(input) #the equation for the mean.
  return(our.mean) #the object 'our.mean' is what is created.
}
```

It is important to note that R will overwrite the names of existing functions. For example, the function `rowMeans()` can be overwritten by the command `rowMeans <- function(input){...}`. So be careful with your naming of functions. To be safe, just type the name of what you want to call your function to see if it exists in the loaded libraries.

More about the function of the `function()` Function

The `return()` command within the `function()` function is helpful when our function creates multiple objects, but we only want particular objects returned by the function. For example, let's create a function that takes the mean of two different inputs, then creates a new object that has both of these means as a single object:

```
function.two.means <- function(input1,input2){ #this function takes two arguments.
    mean.1 <- sum(input1)/length(input1) # take the mean of the first object.
    mean.2 <- sum(input2)/length(input2) # take the mean of the second object.
    both.means <- c(mean.1,mean.2) #combine the means into a object.
    return(both.means) #return the object that is both means.
    }
```

Let's give it a try:

```
x <- c(1,2,3,4,5)
y <- c(6,7,8,9,10)
function.mean(x)
```

```
## [1] 3
```

```
function.mean(y)
```

```
## [1] 8
```

```
function.two.means(x,y) #note that we have to put in two arguments here.
```

```
## [1] 3 8
```

Note that this function creates an object with two means. That is to say, a *vector* with dimensions *1 x 2*. That is different then a function which takes the mean of the means. For example:

```
function.mean.of.means <- function(input1,input2){ #this function takes two arguments.
    mean.1 <- sum(input1)/length(input1) # take the mean of the first object.
    mean.2 <- sum(input2)/length(input2) # take the mean of the second object.
    mean.of.means <- mean(c(mean.1,mean.2)) #take the mean of the means.
    return(mean.of.means) #return the object that is the mean of both means.
    }
```

Let's give it a try:

```
x <- c(1,2,3,4,5)
y <- c(6,7,8,9,10)
function.mean(x)
```

```
## [1] 3
```

```
function.mean(y)
```

```
## [1] 8
```

```
function.mean.of.means(x,y)
```

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

Now, compare that to:

```
function.two.means(x,y)
```

```
## [1] 3 8
```

Question: What is the difference between the statement `both.means <- c(mean.1,mean.2)` in the first function and the statement `mean.of.means <- mean(c(mean.1,mean.2))` in the second function?

Scripts and Source

Rather than writing line-by-line, it is easier to create functions in a text editor and then copy and paste them into R. Many text editors will auto-generate brackets, parentheses, etc., making the programming much easier.

Within R, you can create script files (which have the .R extension) than can be called using the `source()` function. Rather than copy and pasting all of the text in a script, you can use the `source()` function to run an entire script.

For example, execute the following script to plot a Christmas tree:

```
source("https://www.jacobtnyoung.com/uploads/2/3/4/5/23459640/tree_source.r")
```

Now, download the file by going to https://www.jacobtnyoung.com/uploads/2/3/4/5/23459640/tree_source.r and looking through the script.

Loops

Often we need to repeat an action several times and would like a procedure that does this for us. In R, the `for()` function accomplishes this task and takes the following form:

```
for(i in 1:n){
  #do some stuff here
  }
```

In this syntax, `for` indicates that we are going to loop from a start index to an end index, `i` is the index we are looping over, `1` is our start index, `n` is the end index, `{` opens the loop and `}` closes the loop. With loops, actions that take place in the loop involve objects, and these objects must be created before we initiate the loop.

Let's create a look that takes each value of an object and squares it. For example:

```
  x <- seq(1,5,1)            #create an object called x that is 1,2,3,4,5.
  x.squared <- NULL        #create an object that is empty which we will use in the loop.
    for(i in 1:5){          #for each element for 1 through 5.
      x.squared[i] <- x[i]^2 #each element of x.squared will be x[i] squared.
    }
x.squared # print the object to see the value.
```

```
## [1]  1  4  9 16 25
```

R also has some very helpful loops already programmed in the `apply()` function family. The `apply()` function allows us apply to a function to a two-dimensional object. The `apply()` function has three arguments: the object, whether to apply across rows or columns, the functions to use. Let's check out the help:

```
?apply
```

For example, say we have a data set with 5 individuals and 3 variables and we want the mean for each of the variables:

Person	Var1	Var2	Var3
John	1	2	3
Essa	4	5	5
Stan	4	3	2
Cora	1	1	2
Dezy	3	4	5
Mean	?	?	?

If we read these data in as a matrix, then the mean for each variable is just the column means:

```
#Create the data matrix using the matrix function.
data <- matrix(
  c(1,2,3,4,5,5,4,3,2,1,1,2,3,4,5), # a vector of the values.
  nrow=5, # the number of rows in the matrix.
  ncol=3, # the number of columns in the matrix.
  byrow=TRUE #instructions to read by row (vs. by column).
  )
```

Note, we could write:

```
mean(data[,1])
mean(data[,2])
mean(data[,3])
```

Or, we could just use the `apply()` function:

```
apply(data,2,mean)
```

```
## [1] 2.6 3.0 3.4
```

If/Else Statements

R has a useful function called `which()` that returns T/F statements about objects. The function returns the position of the values meeting some condition. For example:

```
x <- c(1,2,3,4,5,5,4,3,2,1,1,2,3,4,5)
which( x > 3 ) #which elements of x are greater than 3?
```

```
## [1]  4  5  6  7 14 15
```

```
#shows that elements in the 4th, 5th, 6th, 7th, 14th, and 15th positions are > 3.
```

Using the same idea, the `if()` function can be used if we want to perform a certain task depending on some condition. The basic structure is:
`if`(statement that gives TRUE or FALSE) `{`

action to take
}

For example, say we want to recode some value if it meets some condition. Specifically, lets recode all values in `x` to be 0 of they are greater than 3. To do this, we need to put our if statement in a `for()` loop.

```
x <- c(1,2,3,4,5,5,4,3,2,1,1,2,3,4,5)

for(i in 1:15){
    if(x[i] > 3){ #define the condition.
      x[i] <- 0 #assign the value if condition is met.
    }
  }
x
```

```
##  [1] 1 2 3 0 0 0 0 3 2 1 1 2 3 0 0
```

There is also the `else` command that lets us perform actions if some condition is not met. For example, recode all values to be 1 if they are not greater than 3 (i.e. going to be changed to 0).

```
x <- c(1,2,3,4,5,5,4,3,2,1,1,2,3,4,5)
for(i in 1:15){
  if(x[i] > 3){
    x[i] <- 0
    }
  else x[i] <- 1
  }
x
```

```
##  [1] 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0
```

Finally, there is also the `else if()` function that lets you combine several possible actions. For example:

```
x <- c(1,2,3,4,5,5,4,3,2,1,1,2,3,4,5)

for(i in 1:15){
  if(x[i] > 3){ #recode 4 and 5 to 0.
    x[i] <- 0
    }
  else if(x[i] == 3){ #recode 3 to 2.
    x[i] <- 2
    }
  else x[i] <- 1 # recode 1 and 2 to 1.
  }
x
```

```
##  [1] 1 1 2 0 0 0 0 2 1 1 1 1 2 0 0
```