WEEK 8: FUNCTIONS AND LOOPS

THOMAS ELLIOTT

1. Functions

Functions allow you to define a set of instructions and then call the code in a single line. In R, functions are defined much like any other object, assigning the function definition to a named object:

```
> my.add<-function(x,y) x+y
> my.add(3,5)
[1] 8
```

In the above code, I created a new function named `my.add`. The function takes two arguments, x and y, and returns their sum. This is a really basic function, containing only a single line of code. If you want to include more than one line of code in a function, you will need to enclose the code in braces:

```
> recode.missing<-function(x,recode=0) {
+   x[which(is.na(x))]<-recode
+   return(x)
+ }
> my.vector<-c(1,3,NA,6,5,NA)
> recode.missing(my.vector)
[1] 1 3 0 6 5 0
```

The above code introduced a few new things. First, you can assign default values to arguments. In the above code, the recode argument contains the value that missing data will be recoded to. By default, missing data is recoded to 0. However, you could supply an argument for recode to change what missing data is recoded to:

```
> recode.missing(my.vector,-99)
[1]   1   3 -99   6   5 -99
```

Second, the code is wrapped in braces, allowing multiple lines of code to be executed by the function. Finally, the return value of the function is returned with the `return()` function. The return function is not strictly required, we could have achieved the same output by simply printing x, but the programmer in me is horrified by that and so I use `return()` and strongly suggest you do the same. This also makes the return value much more explicit.

When you run a function, they have their own environment (work space) that gets deleted at the end of the function. This means you can create variables within a function, but they will not exist in your workspace after the function runs.

Date: March 1, 2016.

```
> myfunc<-function(x) {
+   y<-x+3
+   return(y)
+ }
> ls()
[1] "myfunc"
> myfunc(6)
[1] 9
> ls()
[1] "myfunc"
```

Notice that the function generates a new variable, y, when it runs, but y does not exist in the workspace after calling the function. Functions can access variables in the global environment, but relying on this can get confusing and makes the code difficult to read. This is because a function looks for variables FIRST in its own environment, THEN in the global environment. This means that if there is a variable in the function's local environment that is named the same as a variable in the global environment, it will use the local version. Within a function, if you try to assign a value to a variable with the same name as a variable in the global environment, it will instead create a local variable with the same name. You can see that the behavior here is not what we would intuitively expect, and so can easily lead to problems. As an illustration of all this, here is a quick example:

```
> myfunc<-function(x) {
+   z<-x+y
+   return(z)
+ }
> x<-4
> y<-3
> z<-78
> myfunc(6)
[1] 9
> z
[1] 78
> x
[1] 4
```

So we define a function that takes as its argument x, adds y to it and stores this sum in z, then returns z. We then define in the global environment and x, y, and z with their own values. Calling the function produces the sum of 6 (the number we pass) and 3 (the value of y in the global environment). In the function, this was assigned to z, but you can see that z in the global environment was unchanged. The moral of the story here is that you should pass as arguments any variables you might need from the global environment, do not access them directly from within the function.

1.1. **Passing an Indeterminate Number of Objects.**

Sometimes you may want to write a function that accepts an indeterminate number of objects. For example, the `c()` function will accept any number of objects to concatenate together. You could do this in a couple ways. One would be to accept a single argument that you expect to be a list of objects, but this requires combining the objects into a list prior to calling the function. The second option is to allow a user to supply as many arguments as they want in the function call and then grabbing those objects within the function. You can do this with the ... special character in the function argument definition.

```
> mysum<-function(...) {
+   thevalues<-list(...)
+   thevalues<-unlist(thevalues)
+   thesum<-sum(thevalues)
+   return(thesum)
+ }
> mysum(1,2,3,4,5,6,7,8)
[1] 36
```

The above function will accept any number of values as arguments. It stores the arguments in thevalues using the `list(...)` function, which returns the values of the arguments in a list format. We then convert this into a simple vector using `unlist()`. We then sum the values and return the sum. The ... special character can be used with named arguments:

```
> mysum<-function(...,na.rm=TRUE) {
+   thevalues<-list(...)
+   thevalues<-unlist(thevalues)
+   thesum<-sum(thevalues,na.rm=na.rm)
+   return(thesum)
+ }
> mysum(1,2,3,NA,5,6,7,8)
[1] 32
> mysum(1,2,3,NA,5,6,7,8,na.rm=FALSE)
[1] NA
```

So above we added a named argument to allow us to supply the na.rm flag for the sum function. By default, the value of this flag is TRUE. We call the mysum function, supplying a NA value in the list, and the sum is returned. We then supply the same list, but also supply the na.rm=FALSE argument. Now we get the sum NA, which is what happens when sum is applied to a vector that contains NAs and na.rm=FALSE. Notice that we have to explicitly name our argument (we have to supply `na.rm=FALSE`), otherwise the function will consider it another object to be stored in ....

2. Loops

Loops allow you to repeatedly run a series of code until some condition is met. This makes your code more efficient, and saves you time in writing code. The two types of loops available in R are `for` and `while`.

2.1. **for loops.**

For loops are for situations in which you need to repeat code a set number of times. The for loop in R is defined in the following way:

```
for( i in c(start:stop) ) {
the code
}
\begin{verbatim}

In the above code, i will contain the value for the current iteration of the loop. The v

\begin{verbatim}
> for(i in c(1:5) ) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

In the for loop above, we iterate over a vector that contains the values 1 through 5, and prints the value. The `c(1:5)` above is called the sequence, and defines what values i iterates over. This sequence can be any sequence of values:

```
> for(i in c(5:10) ) {
+   print(i)
+ }
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
>
> for(i in c(5,2,8,3,4)) {
+   print(i)
+ }
[1] 5
[1] 2
[1] 8
[1] 3
[1] 4
```

Let's see some more practical examples. First, one way to generate a series of dummy variables for a categorical variable would be to loop over the different categories and create a dummy variable for each:

```
for( i in unique(mtcars$cyl) ) {
  mtcars[[paste0("cyl_",i)]]<-(mtcars$cyl==i)*1
}
```

The sequence over which we are looping are the unique values of the `cyl` variable in mtcars. Then we have a single command within the loop that accomplishes a lot, so let's unpack it a little. The right-hand side of the assignment creates a variable that is one when the cyl variable is equal to i, and 0 otherwise. So this creates the dummy variable. On the left hand side, we are creating a new variable named cyl_(i) where (i) is the number of cylinders the dummy variable is indicating. We accomplish this by first creating the variable name as a character value with the `paste0()` command. `paste0` takes two or more arguments and concatenates them into a single character with no space in between. You can define a custom separator with `paste` - see the help file for more information about how paste works. So if $i = 4$, then the new variable name will be equal to `cyl_4`. We use the double brackets to create the new variable and assign it the dummy values. In the context of data frames, the double brackets function much like $ does, except it allows us to algorithmically access or assign variables. We couldn't combine the `paste()` command with $ to access variables in the data frame. There are more ways in which [[and $ differ, but these differences are much more technical and I haven't come across instances in they made a difference in my programming.

The above is how we can algorithmically create new variables within a data frame, but how do we create new objects within the environment? The following example illustrates this use case:

```
for( i in unique(mtcars$cyl) ) {
  df<-mtcars[mtcars$cyl==i,]
  df.name<-paste0("mtcars.cyl",i)
  assign(df.name,df)
}
```

In the above example, we again are iterating over the unique values of the `cyl` variable. We extract a new data frame that contains only those cars with the number of cylinders equal to i. We then create a character object that contains the name of the object we want to create in the environment (in this case, we want to save the extracted data frame with a unique name), then we use the `assign()` function to assign the df to an object named df.name. The `assign()` function works much like $< -$ does, but you can create object names algorithmically, and you can also assign variables to environments other than the current working environment (this could be useful if you are writing a function in which you need to modify objects in the global environment). The result after the for loop runs is that in the environment there are now three new data frames that contain only those cars with matching cylinder numbers.

2.2. **while loops.** While loops will iterate while an expression remains true. For example:

```
> x<-0
> while( x<5 ) {
+   print(x)
+   x<-x+1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
```

```
[1] 4
```

Here we define a variable x and assign it the value of 0. Then while x is less than five, we first print x to the console and then add 1 to x. Notice the while loop stops executing when x equals 5.

I tend not to have many cases in which I use a while loop. To show a more substantial (though not more practical) example of a while loop, below is an algorithm for performing a bubble sort on a vector of numbers. Bubble sorts are a simple sorting algorithm that are not especially efficient at sorting, but often used in introductory computer science classes when introducing the concept of sorting algorithms. Bubble sorts work by iterating over a list of numbers, comparing adjacent numbers and swapping them if the two numbers are out of order from each other. The algorithm keeps iterating over the list until no swaps are required.

```
> x<-sample(10)
> x
 [1]  9  4 10  3  7  8  1  5  6  2
> sorted<-FALSE
> while( !sorted ) {
+   anyFlips<-FALSE
+   for(i in c(2:length(x)) ) {
+     if ( x[i-1] > x[i] ) {
+       anyFlips<-TRUE
+       a<-x[i-1]
+       b<-x[i]
+       x[i-1]<-b
+       x[i]<-a
+     }
+   }
+   if( !anyFlips ) sorted<-TRUE
+ }
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

In the above code, the while loop keeps repeating until the list is sorted, and the list is sorted when the for loop didn't have to swap any numbers.

2.3. **apply functions.** In addition to the for and while loops, R also includes some functions that make iterating over vectors, arrays, and lists fairly easy. These functions are more limited in what they can do, but are substantially faster than a loop.

The `apply()` function iterates over margins of an array, repeatedly evaluating a function on the dimensions of the array defined by the margins. In other words, `apply()` allows you to evaluation a function over rows, or over columns, of a matrix or array.

```
> x<-matrix(sample(16),nrow=4)
> x
     [,1] [,2] [,3] [,4]
```

```
[1,]    3    4   13    6
[2,]   10    9    7    8
[3,]   16    2   15   12
[4,]    5    1   11   14
> apply(x,1,sum) # calculate row sums
[1] 26 34 45 31
> apply(x,2,sum) # calculate column sums
[1] 34 16 46 40
```

The function that is evaluated can be any function that accepts a vector of values and returns a single value. This can also include user-written functions, making the apply functions very powerful.

The `lapply()` function iterates over a list, runs the function on each element in the list, and returns a list with the results of the function. In the following example, a list of vectors of random variables is iterated over and the mean of the vectors are returned in a list.

```
> x<-list()
> for(i in 1:5) {
+   x[[i]]<-runif(10,min=10,max=100)
+ }
> lapply(x,mean)
[[1]]
[1] 53.90789

[[2]]
[1] 70.35918

[[3]]
[1] 41.68023

[[4]]
[1] 61.29006

[[5]]
[1] 55.44191
```

The benefit of the lapply function is that since the return object is a list, the function that gets evaluated can return any object.

The `sapply()` function iterates over a list (or an object that can be coerced into a list) and returns a vector, matrix, or array as appropriate. In other words, it will attempt to simplify the list to as simple an object as it can. Using sapply instead of lapply in the above example produces:

```
> sapply(x,mean)
[1] 53.90789 70.35918 41.68023 61.29006 55.44191
```

There are a few other apply functions available, but the above three are probably the most commonly used.