Stat 2994: Statistical Computing

Introduction to R: 6

Writing R Functions

One of the most important features of R is the ability to write and modify functions to perform computations for which built-in R functions are not already available. The general form of an R function definition is:

function (*arguements*) *expression*

As a simple example of writing R functions, consider constructing a function to obtain the square of a number. The new function, which we will name sqr(), is defined as follows:

```
> sqr <- function(x) x*x
```

where x is the *argument* of the function and is assigned to the function name *sqr*. This function may be used the same way as a built-in R function is used:

```
[1] 4
> h <- c(15.1,11.3,7.0,9.0)
> sqr(h)
[1] 228.01 127.69  49.00  81.00
```

The function *sqr* is vectorized since it works on the individual elemnts of a vector. The reason is that the arithmetic operation"*" used in defining *sqr* is a vectorized operation. It is important to make sure that all operations in a function definition are vectorized if you require the function to be applied to each element separately. Often a function need more than a single argument for the required computation. Consider a function logbb() that will compute the logarithm to the base of a number x. This function requires both b and x as arguments and may be defined as:

```
> logbb <- function(x,b) log(x) / log(b)
> logbb(8,2)
[1] 3
> logbb(81,3)
[1] 4
```

The benefit of writing a function is that once it has been established in an R session, we can use it as often as we like, with whatever values we need as the argument.

Exercise 1: Write a function called *calc* that has two arguments: *a* and *b*, and calculates the following for any two numbers:

$$\left(\frac{a}{b}\right)^{a} - \left(\frac{b}{a}\right)^{b}$$

Now use the function for the following set of values: (1,2) (1,3) (2,4) (8, 3) (-5, -0.25) (π, log(2))

Many times when writing a function in R, you will use conditional statements (if else), looping statements (for, while, repeat) or both in order to achieve the final purpose. It is important to break it down into parts. Consider how you would write the conditional or loop statement for one specific value and then write the function replacing that specific value with the argument term. The example below is to calculate the factorial.

First, how might we use a for loop to calculate the factorial of the number 5? Let's define a result vector and have it equal 1. Now let the loop decrease from 5 to 1 while multiplying each time by the current number in the loop.

```
> result <- 1
> for (i in 5:1){
+   result <- result*i
+ }
> result
[1] 120
```

But this only works for the number 5, if we wanted to find the factorial for 3, 6, or any other number we would have to rewrite or resubmit the loop with the changes. Instead lets write a function and replace 5 with a placeholder that will be the *argument* for the function.

```
> fact <- function(n){
+   result <- 1
+   for (i in n:1){
+     result <- result*i
+   }
+   result
+ }
> fact(5)
[1] 120
> fact(3)
[1] 6
> fact(6)
[1] 720
```

This is a much better way to find the factorials if we want to use it for many different numbers.

There is another technique when writing R functions that can simplify the function above even more. Using *recursion*, we can use the function inside of itself to calculate our factorial. It is a more advanced technique, but if you walk through the process it makes sense.

```
> fac <- function(n) if(n <= 1) 1 else n * fac(n-1)

> fac(5)
[1] 120
> fac(3)
[1] 6
> fac(6)
[1] 720
```

Many R functions will have multiple expressions and different combinations of conditional and looping statements. The function below named larger1(), takes two vectors (assumed here to be the same length) as arguments, compares them element by element, and returns a vector that consists of elements that are larger in magnitude of the corresponding elements in the two vectors.

```
> larger1 <- function(x,y){
+   z <- x
+   for (i in 1:length(x)){
+     if (y[i] > x[i]) z[i]<-y[i] else z[i]<-x[i]
+   }
+   z
+ }
```

Remember that if you want a function to return some value, you should request that object as the last item before closing the final brace, which is why we have "z" by itself above.

```
> a<-1:10;a
 [1]  1  2  3  4  5  6  7  8  9 10
> b<-rep(5,10);b
 [1] 5 5 5 5 5 5 5 5 5 5
> larger1(a,b)
 [1]  5  5  5  5  5  6  7  8  9 10
```

But larger1 is not the most efficient way we can write a function to do this. Loops are relatively time consuming in R and if we can avoid them it is preferable. The function larger2() will perform the same task without a loop statement, and instead using a logical statement to index the vectors.

```
> larger2 <- function(x,y){
+   index <- y > x
+   x[index] <- y[index]
+   x
+ }
> larger2(a,b)
 [1]  5  5  5  5  5  6  7  8  9 10
```

And one more time we can simplify it by eliminating the object index and placing the logical statement directly inside the brackets

```
> larger3 <- function(x,y){
+   x[y>x] <- y[y>x]
+   x
+ }
> larger3(a,b)
 [1]  5  5  5  5  5  6  7  8  9 10
```

At this point in writing R code, you should be more concerned with finding a correct way to get R to perform what you want it to and less worried about its efficiency. However, when you consider that some data sets have millions of observations, it isn't hard to understand why efficiency matters. Every loop or extra object assignment may have to be performed for each of the observations and repeating it millions of times may take extra minutes or hours in extremely large data sets when it could have been simplified.

Exercise 2: Refer back to Exercise 2 in Lecture notes #4. Let's create a function that can compare the stats from two different football teams.

1. Write a function that can take any two scores (Home team first, Away team second) and results in a statement about the outcome, either "Home WIN!!" or "We Lost :(". Test your function with multiple different combinations of scores for the Home and Away team.

2. Write a function that can take the number of passing and rushing yards for both the Home and Away team and returns a prediction statement, "Sure Win" or "Not Sure" and also a predicted score using the criteria from part two of Exercise 2. Test the function out with different combinations of passing and rushing yards for the home and away teams.