Functions and Scoping in R

This week we will cover creation of custom functions and an important concept called "scoping".

Creating functions

The ability to create your own functions exemplifies the extreme flexibility of R. Custom functions are also central to simulation modeling and maximum likelihood inference in R. However, functions are also useful to simplify a common task. Breaking code into pieces contained in functions also makes code easier to create, reuse, and understand.

I. Function definition

As you hopefully recall from our introduction to predefined functions in R, a call to a function is done using the *function name* and a set of *arguments*. Definition of a custom function requires both of these items and the *body* of the function, which provides the syntax for completing the intended purpose of the function. Similar to Flow of Control structures, functions use braces to designate the function body. Usually, the body of the function ends with a call of the *return* function, which designates the values to be provided by the function.

```
> mean<-function(x){
+  mu=sum(x)/length(x)
+  return(mu)
+ }
> y=c(1,2,3,4,5)
> mean(y)
[1] 3
```

II. Returning values

The purpose of a function is usually to transform some input (*arguments*). To gain access to the results of that transformation, values must be returned by the function. To explicitly return an object, the `return()` function is used. Any single R object can be returned by a function. Many functions take a vector as an argument and return a vector of the same length. Other objects generate a large amount of diverse output, and often a list is used to return that complex output.

```
> mean<-function(x){
+  mu=sum(x)/length(x)
+  return(list(length=length(x),mu=mu))
+ }
> y=c(1,2,3,4,5)
> mean(y)
$length
[1] 5

$mu
[1] 3
```

An explicit return call is not always necessary. The last executed statement will be returned by a function by default. Therefore the functions below return similar values as

the function defined above. Note that if a for loop is used, a vector or matrix filled in that for loop cannot be returned without an explicit return call. This is because the last executed statement is the final return to the top of the for loop.

```
> mean<-function(x){
+  mu=sum(x)/length(x)
+  list(length=length(x),mu=mu)
+ }
> y=c(1,2,3,4,5)
> mean(y)
$length
[1] 5

$mu
[1] 3
```

If the last line executed in the function includes a variable assignment that variable will be returned, but it will not be printed at the console as above.

```
> mean<-function(x){
+  mu=sum(x)/length(x)
+  out=list(length=length(x),mu=mu)
+ }
> y=c(1,2,3,4,5)
> mean(y)
> x=mean(y)
> x
$length
[1] 5

$mu
[1] 3
```

Although an explicit return call is not necessary, it often helps in maintaining easy to understand code.

III. Default argument values

You've likely noticed that each argument does not have to be specified when calling a predefined R function. This is because default values for those parameters are specified. You can also specify default argument values for custom functions.

```
> mean<-function(x,na.rm=FALSE){
+   if(sum(is.na(x))>0 & na.rm==FALSE){
+     print("You have NAs")
+   }else{
+      x=x[!is.na(x)]
+     mu=sum(x)/length(x)
+     return(mu)
+   }
+ }
> y=c(1,2,3,NA,4,5)
> mean(y)
```

```
[1]  “You have NAs”
> mean(y,na.rm=TRUE)
[1] 3
```

IV. Scoping

When a function in R executes it creates a *local environment*. As a result, any variables defined within a function are not available after that function is run because that *local environment* ceases to exist. This is handy because it avoids accidental overwriting of a variable name that is shared in the main script body and a function body. However, it also means you need to specify that any values you want access to should be returned from a function. Conversely, any variable defined at the top scope or environment (called R_GLobalEnv or .GlobalEnv) is available inside of functions and local environments. If a function is defined in another function, these *scoping rules* apply in a hierarchical manner. Any variables defined in an environment or scope above the current local environment are available inside the function, but variables defined in local environments lower in the hierarchy will not be available in environments higher in the hierarchy.

```
> rm(list=ls())
> w=12
> f<-function(y){
+   d=8
+   h<-function(){
+     return(d*(w+y))
+   }
+   return(h())
+ }
> environment(f)
<environment: R_GlobalEnv>
> ls()
[1] “f” “w”
> ls.str()
f : function(y)
w: num 12
> f(2)
[1] 112
```

Modify the functions defined above to print the environment of h. What do you find?

What if we define the functions in a non-nested manner?

```
> rm(list=ls())
> w=12
> f<-function(y){
+   d=8
+   return(h())
+ }

> h<-function(){
+   return(d*(w+y))
+ }

> f(5)
```

```
Error in h() : object "d" not found
```

Why do we get this error?

V. Functions acting on functions

Because functions are objects, just like any thing else in R (e.g. vector, list, or dataframe) we can create functions that take other functions as arguments. This approach provides some very important functionalities, including minimization, which we will use fairly heavily in Module 2. Today we will work through an example of minimization to demonstrate how a function can take another function as an argument.

When estimating model parameters we often are minimizing some quantity. An example of minimization of a quantity is the minimization of the sum of squared errors in simple linear regression. Hopefully you all remember this from introductory statistics.

In this example we will create a function that takes vectors of observations from independent and dependent variables and returns the sum of squared errors (SSE) assuming a linear model (y=mx+b) can relate these two variables. We then use our function that calculates SSE along with a function that minimizes the value returned by another function, `optim()`, to find the best estimates of *m* and *b* given our observations. This is primarily intended as an example of a function that takes another function as an argument, but it also provides a glimpse ahead to where we will be going once we start talking about statistics.

```
> SSEcalc<-function(p,x,y){
+   b=p[1]
+   m=p[2]
+   yhat=m*x+b
+    errors=y-yhat
+   SSE=sum(errors*errors)
+   return(SSE)
+ }

> x=sample(1:25,10)
> y=rnorm(10,x*2+1,sd=5)
> plot(x,y)

> guess=c("b"=1,"m"=1)
> modelFit=optim(par=guess,fn=SSEcalc,x=x,y=y)
> modelFit
> abline(a=modelFit$par[1],b=modelFit$par[2])
```

What are the arguments taken by `optim()`? What is returned by `optim()`?