Some Fundamental R Programming Concepts

Lesson 02

Psychology 310

1 Creating Our Own Functions in R

When we program in R, often we find we can do things with a single command, especially when we are performing rudimentary statistical operations. This is because R has been designed from the ground up with statistics in mind.

Consider, for example, the computation of the sample mean. As we have already seen in class, we don't need to tell R how to compute the mean. For example, if a vector X has elements 1,2,3,6, we simply say

```
> X <- c(1,2,3,6)
> mean(X)
[1] 3
```

The mean of the numbers appears automatically. This works just fine, but how?

At some point, someone created an R *function* that automatically computes the mean of a set of numbers. In this lesson, we learn just a bit about how this is done. Learning to program functions in R is the secret to becoming a "power user" and getting the most out of R.

Let's examine a simple R function that we can create for ourselves. We have already learned in class that the *deviation score* equivalent of a set of numbers is generated by subtracting the mean from each score in the group. Specifically,

$$dx = X - \bar{X} \tag{1}$$

In R, we can create a function to do this automatically. Here is the code:

```
> deviation.score <- function(x) {
+ return(x - mean(x))
+ }
```

Let's see how it works

```
> X
[1] 1 2 3 6
> deviation.score(X)
[1] -2 -1  0  3
```

We also learned in lecture that the Z-scores corresponding to a list of numbers are obtained by dividing the deviation scores by the standard deviation.

$$Z_x = \frac{X - \bar{X}}{S_x} = \frac{dx}{S_x} \tag{2}$$

Since we already have a deviation score function available, we can use it to construct a Z-score function.

```
> z.score <- function(x){
+ return(deviation.score(x)/sd(x))
+ }
```

Now, since the deviation score function is so simple, we *could* have defined the Z score function without using it, i.e.,

```
> z.score.2 <- function(x){
+   return((x - mean(x))/sd(x))
+ }
```

However, in many situations the functions used to perform a more complex operation involve a substantial amount of code. In such situations, the more complex function is made much simpler and easier to read by cascading functions, i.e., by building complex functions out of simpler ones.

Let's try our Z score function

```
> X
[1] 1 2 3 6
> z.score(X)
[1] -0.9258201 -0.4629100  0.0000000  1.3887301
```

It works perfectly.

2 The Anatomy of a Simple Function

In order to construct a function in R, you need to use the proper syntax. You begin by thinking up a name for your function. Chapter 10 of the manual *An Introduction to R* gives the following simple example.

```
> twosam <- function(y1, y2) {
+          n1  <- length(y1); n2  <- length(y2)
+          yb1 <- mean(y1);   yb2 <- mean(y2)
+          s1  <- var(y1);    s2  <- var(y2)
+          s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
+          tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
+          tst
+ }
```

This is a function to compute the classic two-sample t-test for two *independent* samples of possibly different sizes n_1 and n_2. The first line of the function definition begins with `twosam <- function`, alerting the R system that `twosam` will from that point be used as the name of a function that will now be defined.

The *input* to the function is listed next in parentheses. In this case the input is the two vectors `y1` and `y2`.

The actual code defining what the function *does* with these two input vectors is then given between braces. (The braces define a block of code, and may be omitted if the function does its work in a single line.)

In the sample coding given above, we see that the first several lines each contain two commands separated by semi-colons, rather than having one command. This makes the function listing more compact, but in my opinion makes it more difficult to read.

2.1 Naming and Commenting

Several aspects of the sample function in the above example are open to question. To begin with, the function name is not very useful. The name `twosam` could stand for many things. One key to good function naming is that the function name should tell you (within reason) what the function does! If you have a flawless memory, you might remember what `twosam` does — for a while. But after several months, unless you use the function every

day, you will have to page through your code to find the proper function name, thereby wasting valuable time.

Expert programmers generally agree that function names should be both *long* and *fully descriptive*, using complete English words rather than abbreviations. The name `twosam` satisfies none of these prescriptions.

It is instructive to see how a system that involves thousands of functions does it. The computer algebra system *Mathematica* uses function names that are long and generally fully descriptive. This doesn't necessarily mean that you will recall the name of the function, but it might help.

For example, in *Mathematica* the equivalent function to `twosam` is called

```
MeanDifferenceTest
```

Notice that the emphasis is on what the function does, rather than what some statistician might choose to call it. However, one might just as well have called the function

```
MeanDifferenceTTestForTwoIndependentSamples
```

Notice that this latter name is more fully descriptive. It takes time to type out the name, but once it is typed, someone reading any block of code employing the function will know what that function is doing without having to look it up.

Compare

```
y1 <- rnorm(23)
y2 <- rnorm(32)
MeanDifferenceTTestForIndependentSamples(y1,y2)
```

with

```
y1 <- rnorm(23)
y2 <- rnorm(32)
twosam(y1, y2)
```

The former is, in an important sense, self-documenting. Someone reading the code would be likely to figure out what the code is doing without having to look up the function name. Since it is already clear that `y1` and `y2` are two samples, the function call `twosam(y1, y2)` is redundant, and probably not helpful.

Another aspect of the sample function `twosam` as shown above is that it is not properly commented. It takes time to include comments in your function code. However, these comments can be invaluable if, several months after writing the code, you have to return to it and (a) understand what it does, and (b) modify it or find an error in it.

Let's rewrite the sample function to make it more readable and easier to maintain.

```
> MeanDifferenceTTestForIndependentSamples <- function(y1, y2) {
+
+ ############### Function Description ############################
+ # Purpose: Computes 2-sample independent sample t-statistic     #
+ #          for comparing the means of two samples.              #
+ #                                                               #
+ # Input: y1 = data for sample 1                                 #
+ #        y2 = data for sample 2                                 #
+ # Output: The t-statistic                                       #
+ #################################################################
+ # Compute sample sizes
+    n1  <- length(y1)
+    n2  <- length(y2)
+ # Compute group means
+    ybar1 <- mean(y1)
+    ybar2 <- mean(y2)
+ # Compute group variances
+    var1  <- var(y1)
+    var2  <- var(y2)
+ # Compute pooled variance estimate
+    pooled.var <- ((n1-1)*var1 + (n2-1)*var2)/(n1+n2-2)
+ # Compute t-statistic and return
+    tStatistic <- (ybar1 - ybar2)/sqrt(pooled.var*(1/n1 + 1/n2))
+    return(tStatistic)
+ }
```

Compare my version of the function with the original. Notice that I inserted detailed comments. Also, notice how I changed several variable names to make them more descriptive. In particular, the author of `twosam` made what most coders would consider a serious error of judgment. Many statistics

students occasionally forget the distinction between the sample variance S^2 and the sample standard deviation S. So a test problem will give the variance and the student will insert it in a calculation that calls for the standard deviation without remembering to take the square root.

Mindful of how easy it is to commit this kind of error, it is important to eliminate the likelihood of such an error in one's coding. With that in mind, it was a mistake to use the notations `s1`, `s2`, and `s` to stand for variances in the `twosam` function code. I changed the names of the variables to `var1`, `var2`, and `pooled.var` respectively.

Note also how I changed `yb1` to `ybar1`. Perhaps `mean1` would have been even better.

The point is that, if you are going to go to all the trouble to create your own functions and thereby create your own, permanent, personal environment for statistical computation and document generation, you should also take a small amount of additional time to make sure that this effort will be leveraged. That is, you will not have to waste time in the future carefully re-reading your code to see how it works, and you will also be in a position to give the code to others who might profit from it, secure in the knowledge that they too will be able to figure out how the code works.

Let's try out our revised function to see how it works

```
> y1 <- rnorm(23)
> y2 <- rnorm(32)
> twosam(y1,y2)
[1] -1.200395
> MeanDifferenceTTestForIndependentSamples(y1,y2)
[1] -1.200395
```

Here is a pointer. You might be asking yourself, "Suppose I have a situation in which I am going to use the `MeanDifferenceTTestForIndependentSamples` function several times. Isn't awfully inconvenient to type that name all those times?" Yes, it is. However, you always have the option of assigning the longer function name to a shorter temporary name within your code. That way, anyone else reading the code in the future (including you) will still have faster access to what the code means. Consider the following:

```
> T2 <- MeanDifferenceTTestForIndependentSamples
> T2(y1,y2)
[1] -1.200395
```

2.2 Parameter Lists and Default Values

At the beginning of a function definition is the *input parameter list*, enclosed in parentheses. The names that appear on this list are the *named arguments*. In some cases, the named arguments will have *default values*. Here is a simple example of such a function, with some sample calculations.

```
> NormalIntervalProbability <- function(a, b, mu=0, sigma=1){
+ ############ Function Description ########################
+ # Purpose: Computes the probability that a normal       #
+ #          random variable with mean mu and standard    #
+ #          will have a value between a and b            #
+ # Note:  a need not be less than b                      #
+ # Output: The probability                               #
+ ########################################################
+
+ if(a==b)return(0) else
+   return(abs(pnorm(b,mu,sigma)-pnorm(a,mu,sigma)))
+ }
> # Some Example Calculations
> # Example 1
> NormalIntervalProbability(-1, 1)
[1] 0.6826895
> # Example 2
> NormalIntervalProbability(1, -1)
[1] 0.6826895
> # Example 3
> NormalIntervalProbability(100, 115, 100, 15)
[1] 0.3413447
> # Example 4
> NormalIntervalProbability(100, 115, sigma=15, mu=100)
[1] 0.3413447
```

The above sample calculations demonstrate some important properties of the way R handles parameter values input to a function:

1. *If values are input without being assigned to a parameter name, they are assigned to the parameters in the order they are listed in the function definition.* Such parameter values are said to be given in "unnamed positional" form. In the first two examples, the input values were simply numbers. Consequently, they were assigned to the parameters `a` and `b` in the order given in the function definition. For example, in the first call, -1 was assigned to `a` and 1 was assigned to `b`.

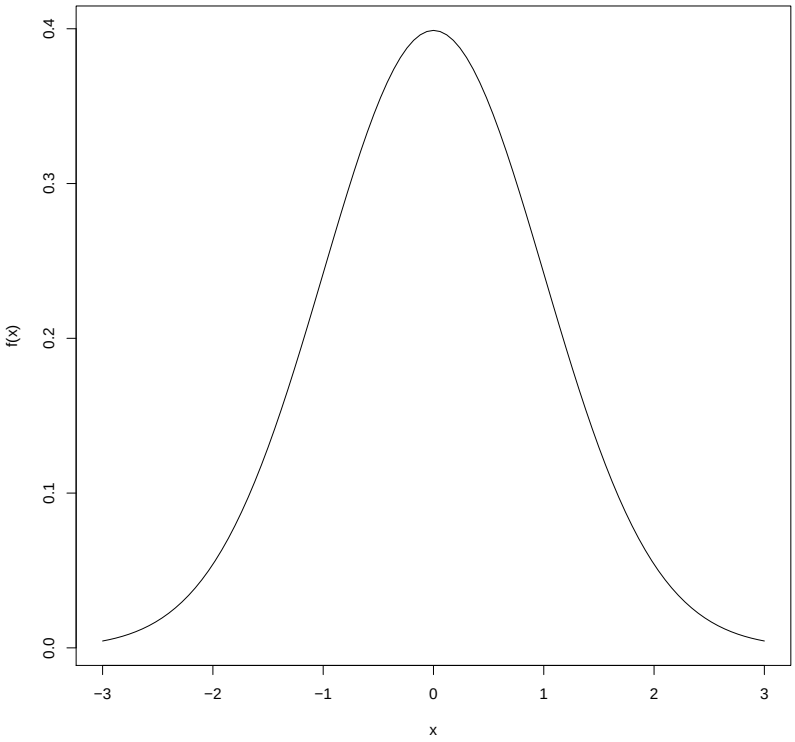
2. *If a parameter has a default value, and no input value is provided, then the default value is assigned to the parameter.* So in the first two examples, 0 and 1 are assigned to the parameters `mu` and `sigma` respectively. On the other hand, in the third example, `mu` is given the value 100 and `sigma` is given a value of 15.

3. if arguments to called functions are given in the "`name=object`" form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments. So in the 4th example, `a` is assigned the value 100, `b` is assigned the value 115. These values are said to be input in "unnamed, positional form." The final two parameters are correctly assigned, even though they are given out of order, because they are provided in `name=object` form.

It is also important to realize that not only can the inputs in a call to a function be arbitrary expressions, it is also the case that default values for a parameter are not restricted to be constants! The default values can be arbitrary expressions, including functions of other parameters input to the function.

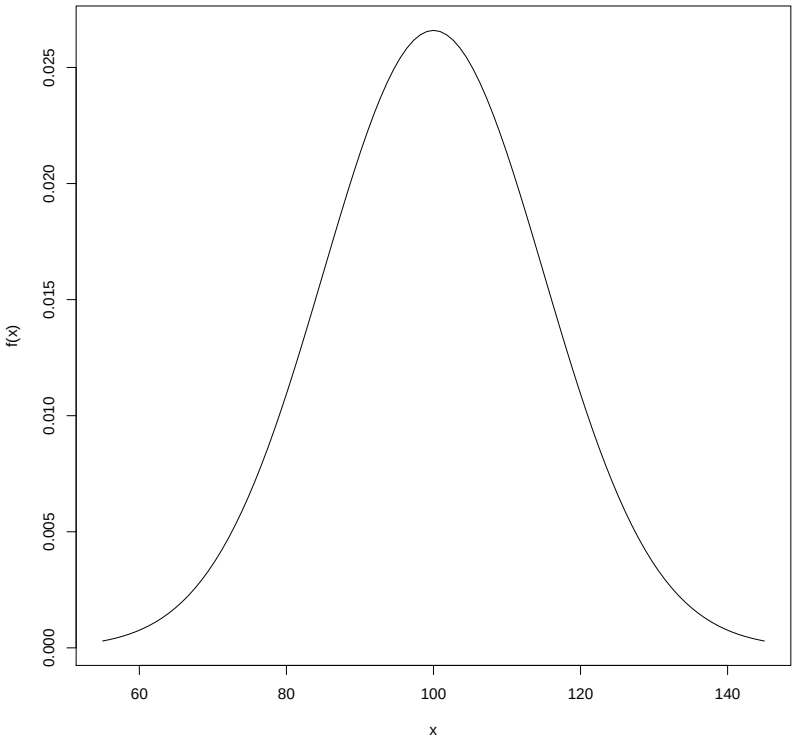
Here is an example.

```
> PlotNormalCurve <- function(mu = 0, sigma = 1,
+                             left.limit = mu - 3*sigma,
+                             right.limit = mu + 3*sigma) {
+ ############ Function Description ########################
+ # Purpose: Plots a normal density function for a        #
+ #          N(mu,sigma) distribution between x values    #
+ #          of left.limit and right.limit                #
+ #        left.limit, right limit = plot range           #
+ # Output: The probability density plot                  #
+ ##########################################################
+
+   curve(dnorm(x,mu,sigma),left.limit,right.limit,ylab = "f(x)")
+ }
> PlotNormalCurve()
```



In the above example, the mean and standard deviation of the normal curve are 0 and 1 by default, and the plot is drawn over the default computed range from -3 to 3. In the next example, we change the mean and standard deviation to 100 and 15, and since we do not input the lower and upper range of the graph, it is computed for us as 55 to 145.

```
> PlotNormalCurve(100,15)
```


f(x)
0.000
0.005
0.010
0.015
0.020
0.025
60
80
100
120
140
x