Lecture 5 - Functional Programming

Will Calandra

Functional Programming

In this lecture, we will learn very useful procedures to make our code reusable, simpler, and easier to read and write! We will learn about functions, which are just as important as vectors in R. R was made popular by its unique function and vector capabilities, making it a convenient language for coding. We will also learn a bit about loops, which appear often in functions that you'll write in the future. Let's get started!

Functions

You already know several functions that you have used in R - think of functions like sample(), lm(), filter(), and mutate(). Even the simple ones like plot() and mean() are functions as well!

How have we used these functions in our scripts? We call them directly, and they usually get their own lines in our script. Often, we will use a function to add its outputs to a dataframe, like this:

```
dataframe$NewColumnName <- someFunction(someInput)
```

Other times, if we just want to see some results for our calculations, we will use functions and write them like this:

```
plot(dataframe$column)
summary(dataframe$column)
vif(someLinearModel)
```

This is totally valid coding - code like this works perfectly and is absolutely fine to include in your scripts! But when we are doing really long analyses or complex projects, wouldn't it be pretty annoying to copy this code over and over again to see your results? Remember from our linear modeling lecture - how many times did we copy and paste a code block like this?

```
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk)

Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)

summary(ProjRecYds)

vif(ProjRecYds)
```

It got pretty tedious after a few times, right? Well, the good news is that there is a better way to do this, and it is by using functions! A great use case for functions is when you find yourself repeating the same code but tweaking the inputs each time you want to run something. But what happens if you can't find a function in R that simplifies your work? The answer is that you can write your own custom functions in R to help you out! We'll see the usefulness of this at the end of this lecture when we understand custom (user-defined) functions a bit more.

Writing Our Own Functions

R has a neat way in which we can define a function in our script and “call” it (fancy way to say use a function) to do something for us.

Functions that are already built in R are ready to be called, so we use them like this:

```
functionName(parameter)

# or...

functionName(parameter1, parameter2, etc)
```

So how are we able to use a custom function? We have to define it first so we can call it! All functions in R are defined, but the functions already built are what we call “under the hood” - they are there, but we just can’t see their source code in our script. Notice how packages are basically just a bunch of functions with pre-built definitions, so when we load a package, we are downloading someone’s custom function definitions and using their functions on our computer. Pretty neat, right?

Defining Our Own functions

Source: https://www.tutorialspoint.com/r/r_functions.htm (https://www.tutorialspoint.com/r/r_functions.htm)

Every function you define will have a structure like this:

```
functionName <- function(parameter1, parameter2, ...) {
  functionBody
  returnValue
}
```

The components of a function are:

- Function name: Required. It is the name of the function (that you define, which is stored in R as an object by its name. Make sure you use the arrow to point to your function name.
- The word “function”: Required. You have to write this, or else R will not know what you are doing. “function” is the key word that lets R know you are defining a function here!
- Parameters: Optional. When you use a function, you pass a value(s) as its parameter(s). You can write a function without parameters (but you still have to include the parentheses before the brackets), but that typically isn’t best practice in a larger program.
- Function body: Required. Pardon the pun, but this is where the “functionality” of your function is built. You tell R what you want the function to do by writing some code in its body.
- Return value: Optional. You can have a function return a specific value that you define in your function body by using return(theValueYouWantToReturn).

After you do this, you can call your function in your script by doing this:

```
functionName(parameter1, parameter2, ...)

# or...

functionName()
```

Example

Let's start by writing a really simple function that does a math calculation for us.

```
# Simple math function
mathFunction <- function(a, b, c) {

result <- a * b + c
print(result)

}
```

What did we do here? We wrote a function called "mathFunction" with parameters "a, b, and c." In the function's body, we want the mathFunction to calculate a result for us, called "result," which we calculate by multiplying a and b and then adding c. We do not have a return value for this function (though we could have "result" as the return value and it would do the same thing) - we just ask the function to print the result using print(result) in the body.

So how do we use this function? Now that we defined it, we can call it!

```
## mathFunction calls
mathFunction(2, 3, 6)
```

```
## [1] 12
```

```
#mathFunction(2, 3)
mathFunction(2, 3, 0)
```

```
## [1] 6
```

```
mathFunction(2, 0, 4)
```

```
## [1] 4
```

```
mathFunction(a = 10, b = 5, c = 3)
```

```
## [1] 53
```

```
#mathFunction(a = 4, c = 7)
mathFunction(a = 2, c = 1, b = 6)
```

```
## [1] 13
```

Notice how we threw a bunch of different parameters in our function, and it works! Also notice that if we leave out a parameter, R returns an error. Since we did not define defaults, R does not know what value to assign that parameter, so we get an error. Coding tip: pick smart defaults for your parameters so you can easily detect errors in your code!

Also notice that order matters. That last line still works when we assign values to the parameters out of order, but otherwise, if you do not directly assign values to parameters, the order in which you enter parameters does matter.

Let's quickly redefine this function with default parameters (and a return value):

```r
## With default parameters
mathFunctionNew <- function(a = 1, b = 4, c = 5) {
  result <- a * b + c
  return(result)
}

mathFunctionNew(2, 3, 6)
```

```
## [1] 12
```

```r
mathFunctionNew(2, 3)
```

```
## [1] 11
```

```r
mathFunctionNew(2)
```

```
## [1] 13
```

```r
mathFunctionNew(,3,)
```

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

```r
mathFunctionNew()
```

```
## [1] 9
```

```r
mathFunctionNew(a = 10, b = 5, c = 3)
```

```
## [1] 53
```

```r
mathFunctionNew(a = 4, c = 7)
```

```
## [1] 23
```

```r
mathFunctionNew(a = 2, c = 1, b = 6)
```

```
## [1] 13
```

Notice the new behavior of our function (no errors), but also notice the weird behavior with our function when we remove parameters. This is why it’s important to choose clever defaults! You can also have some error checks in your function body, but that’s some pretty advanced programming you can research on your own.

So How Can This Be Useful?

Functions are very useful if you are clever about it! Let’s run through a few examples of some useful applications of functions:

Source: https://github.com/hadley/adv-r/blob/master/FP-whole-game.Rmd (https://github.com/hadley/adv-r/blob/master/FP-whole-game.Rmd)

Data Cleaning

Let’s say you have a dataframe of numbers and want the -2’s to be NAs. Here is the dataframe:

```
# Generate dataframe
set.seed(101)
df <- data.frame(replicate(6, sample(c(1:10, -2), 6, rep = TRUE)))

# Rename headers, show dataframe
names(df) <- letters[1:6]
df
```

```
##    a b  c  d  e  f
## 1  9 6  2  6 -2  8
## 2  9 3  4  8  5  4
## 3  7 3  5 10 10  6
## 4  1 9 -2  5  5  8
## 5 10 3  1 10  4  7
## 6 -2 3  1 10  8 10
```

See how we got some -2’s in there? How would we normally get rid of them? Something tedious like this:

```
df$a[df$a == -2] <- NA
df$b[df$b == -2] <- NA
df$c[df$c == -2] <- NA
df$d[df$d == -2] <- NA
df$e[df$e == -2] <- NA
df$f[df$f == -2] <- NA
```

Well, what if we could write a function to help with this? See below:

```
# Replace -2 with NA
fix_missing <- function(x) {
  x[x == -2] <- NA
  return(x)
}

# Call fix_missing()
df <- fix_missing(df)
df
```

```
##    a b  c  d  e  f
## 1  9 6  2  6 NA  8
## 2  9 3  4  8  5  4
## 3  7 3  5 10 10  6
## 4  1 9 NA  5  5  8
## 5 10 3  1 10  4  7
## 6 NA 3  1 10  8 10
```

Pretty easy and compact, right? We give the fix_missing function a parameter (our dataframe), and then we remove all -2's in the dataframe and return the new dataframe. We then call the function and save it as our df. Done!

Exploratory Data Analyses

Here's another example. Using the same dataframe, say we want to figure out some number summaries for each column. How would we do this normally? Something like this:

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

For EVERY column. Not too fun, right? Well, we can write a function to help us solve this problem:

```
## Summary stats function
summaryFunc <- function(x) {
           c(mean(x, na.rm = TRUE),
           median(x, na.rm = TRUE),
           sd(x, na.rm = TRUE),
           mad(x, na.rm = TRUE),
           IQR(x, na.rm = TRUE))
}

# Apply the function to our dataframe
lapply(df, summaryFunc)
```

```
## $a
## [1] 7.20000 9.00000 3.63318 1.48260 2.00000
##
## $b
## [1] 4.50000 3.00000 2.50998 0.00000 2.25000
##
## $c
## [1] 2.60000 2.00000 1.81659 1.48260 3.00000
##
## $d
## [1] 8.166667 9.000000 2.228602 1.482600 3.500000
##
## $e
## [1] 6.40000 5.00000 2.50998 1.48260 3.00000
##
## $f
## [1] 7.166667 7.500000 2.041241 1.482600 1.750000
```

Dataframes are lists, so we can use the lapply() function to apply the summaryFunc to this dataframe. Notice how quickly we got that output!

Model Building

Remember in linear modeling when we had blocks of code like this?

```
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk)

Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)

summary(ProjRecYds)

vif(ProjRecYds)
```

We could get this into a function, right? It could look something like this:

```
LmFunc <- function(data, projectedVar, dataFrameVar, predictVar, explanatory1, explanatory2) {

  projectedVar <- lm(data = data, predictVar ~ explanatory1+explanatory2)

  data$dataFrameVar <- predict(projectedVar)

  # Return multiple results in a list
  results <- list(
    ggplot(data, aes(x=dataFrameVar,y=predictVar))+geom_point()+geom_smooth(method=lm,se=FALSE),
    summary(projectedVar),
    vif(projectedVar)
  )

  return(results)

}

# Call function
run <- LmFunc(Yearly_Data, ProjRecYds, ProjRecYdsWk, Yearly_Data$RecYdsWk, Yearly_Data$TgtWk, Ye
arly_Data$RecTDsWk)
run
```

This really simplifies the process, right? All you have to do is define the function once and your workflow becomes much easier, concise (limiting potential for mistakes), and faster!

But the real takeaway from these examples is that functions can be used in ANY part of the data workflow, from cleaning, to exploring, to analyzing, to interpreting results!

Looping

Now that we know a bit about functions, I'll briefly introduce looping. Looping is useful and often deployed in many programming languages - in this respect R is no different! Just like functions, looping is a special function that allows a computer's instructions to repeat. Loops often appear within functions, so that's why we will learn about them in this lecture!

There are two types of loops we will learn about for R:

- For loop
  - This loop will iterate over itself for a predetermined, specified number of times as stated by a condition. When you know how many times you need to iterate something, use a "for loop."
- While loop
  - This loop will check a condition first, and then as long as the condition is true, the loop will iterate over itself. When you want R to just iterate until some condition changes, then use a "while loop."

This may be a bit confusing to understand now, so let's do some examples (and you can see how they appear/work within functions)!

For Loop

Let's write a function where I want to print a sequence of squares. I can do this in a function like this:

```
## Function for squares
sqFunction <- function(x) {
    print(x^2)
}
sqFunction(1)
```

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

```
sqFunction(2)
```

```
## [1] 4
```

```
sqFunction(3)
```

```
## [1] 9
```

```
sqFunction(4)
```

```
## [1] 16
```

Notice how this function does work, but we have to call it so many times! This is where a loop would be useful - we want to call the function many times where we don't have to manually script for each call. We use a for loop for this one:

```
## For Loop fix
sqFunction <- function(x) {

  for(i in 1:x) {
    print(i^2)
  }

}
sqFunction(4)
```

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

Notice how we used the for loop within the sqFunction - we used "x" as our parameter to control how many times sqFunction was called, or really to define how many times the for loop runs.

In programming, we often use "i" to represent the "index" of a loop, which in this case is 1, 2, 3, and 4. The loop runs for each index of i, and prints (because the print is in the loop. Notice what happens if you put the print outside of the for loop function)!

For more info on for loops: https://www.w3schools.com/r/r_for_loop.asp (https://www.w3schools.com/r/r_for_loop.asp)

While Loop

The while loop is a little easier to understand grammatically; the while loop will iterate “while” a condition is true. Let’s write a while loop that will “print integers while they are less than 6”:

```
## While loop example
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Notice first that we don’t have to put a loop in a function, we just leave it by itself here. Also notice that we initialize i outside of the while loop, so i can start at 1. We then check the condition (“is i less than 6?”), and if true, run what is inside the while loop. Notice that i increments up by 1 each time (by the i <- i + 1 line). This is how we get i to reach 6, which is when the loop stops. If you do not increment your i, your loop will continue forever (which is never good)!

For more info on while loops (there are some cool things you can do within loops) : https://www.w3schools.com/r/r_while_loop.asp (https://www.w3schools.com/r/r_while_loop.asp)

Loops Use Cases

In any case where you want to recursively define a column in your dataframe, looping is the way to do this! Say you are taking a moving average of something; looping can achieve that goal by iterating over each cell as you go! You can also nest loops in your if… else statements, in your functions, and in each other (oh my!). It is a difficult and seemingly scary concept as you are new to programming, but as you see them in practice and start using them, you will start to recognize when they are useful! We cover this very briefly, so checking out those links/finding other online resources will be helpful for you on this topic.

Some Additional Notes on Scope and Practices for Functions

This marks the end of the functional programming lecture! Some additional notes are here that will be helpful as you start to write and work with functions:

- Variables have what is called a “scope,” meaning R understands what variables (and saved values) mean ONLY within their scope.
    - In the past, we have only been using what are called “global” variables - global variables can be read anywhere in your code by R (after you define them, of course).
    - Functions in R have what are called “local” variables - local variables are only readable within their function but not outside of it. This is the case when you define a new variable inside of a function.
        - Example: the “result” variable in our mathFunc can be read by mathFunc, but not anywhere else!

- We have to be careful about scope! So, why don’t we just define all of our variables globally? Well, this can lead to difficult debugging if our functions share the same variables. Therefore, it is best practice to have as few global variables as possible, so faulty operations stay within their functions and can be tested easily.
- You can also write functions inside of functions. If in defining your function you find replication (like in our number summary function, the na.rm), then this technique may be useful.
- Remember, R reads top to bottom. Therefore, it is good practice to list all of your function definitions at the top of your source code, so you don’t have to worry about defining your functions in the middle of a script.
- Try to learn functions in R that take functions as parameters/arguments (like the lapply() function). It can help you see more outputs in your console and help you save complex outputs as objects in your code!
- I hope this helps you out, and I hope you can see the usefulness of functions in R. It really is a nice thing when you put it into practice!