Post One

Jessie Hua

October 30, 2017

Functions

How Do We Effectively Use Functions in R

Introduction

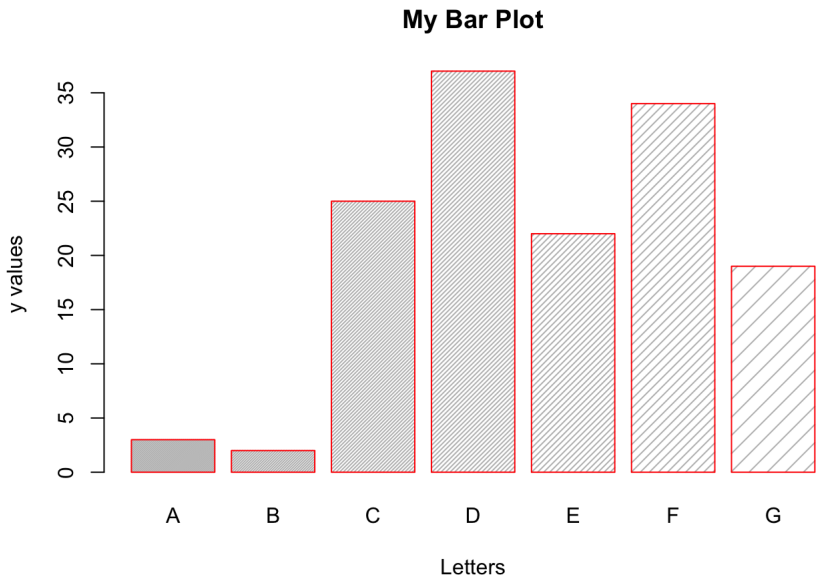
We've begun to explore the use of functions in RStudio using command function(). It is such a powerful tool, because it offers a variety of ways to help us analyze data. Even though RStudio provides many useful commands to organize and plot data, we want to have our own customized commands. This is where function() comes in place - it creates customized functions that will greatly help simplify our code. These functions not only does arithmetic procedures on lists, they can serve many purposes including plotting a graph. In this post, I will first explore the versatility of functions by using operations that plot graphs.

As a Computer Science major, I'm also curious to explore the differences in functions in R versus other programming languages, specifically Python. Therefore, I will also compare and contrast functions in R versus Python to learn more abuout the properties of R functions. In the end, I will describe more concepts in functions, including environments and scoping.

Plotting Graphs with a Function

In lecture and lab, we often associate functions with doing arithmetic operations like conversion between units on columns or rows of a data frame. However, function can actually store any type of code, even code that produces a graph! Below is an example of a function that graphs a bar chart, with y-value vector and density vectore as two arguments.

```
my_bar_plot <- function(x, y){
    barplot(x, main = "My Bar Plot", xlab = "Letters", ylab = "y values", names.arg = c("A","B","C","D","E","F","G
"), border="red", density=y)
}
B <- c(3, 2, 25, 37, 22, 34, 19)
D <- c(90, 70, 50, 40, 30, 20, 10)
my_bar_plot(B, D)
```



Functions in R vs. Python

There are many differences and similarities between functions defined in R and Python. I'm interested in comparing the two languages because I'm familiar with Python. However, I will carefully explain both languages, so no prior experience with Python is needed to understand the next few paragraphs. Through this comparison, you will learn more about the behaviors of functions in R. In addition, you might learn some Python too!

Syntax Differences

Since R and Python are two different programming languages, their syntaxes are obviously different. Here are two examples of defining the function "square" using R and Python respectively.

R

```r
square <- function(x) {
  return(x*x)
}
```

Python (From terminal)

```
>>> def square(x):
...     return x*x;
...
```

Both ways to declare the function “square” is very simplistic and nice. Python might use a little less characters. However, for R, even if we don’t include the “return” command and just put x*x inside the function, it will still return the value of x squared. However, note that the code will continue to run if return is not called. Python, on the other hand, requires the “return” keyword to return any value.

Default Argument Values

Arguments passed into the functions of R and Python can both have default values. This means that even if no argument provided matches with the parameters, the argument will still have a value. In addition, null values can be used as a default argument in both languages. Instead of “NULL” in R, Python more commonly uses “None” as a null value.

To check if a value is null in R, we can use the function is.null(). In Python, we can simply use “is” keyword to determine if x is null or not.

R

```r
square <- function(x = NULL) {
  if(is.null(x)) {
    print("No Input")
  }
  return(x*x)
}
```

Python

```
>>> def square(x=None):
...     if x is None:
...             print("No input")
...     else:
...             return x*x
...
```

Pass-By-Reference vs. Value

Pass-by-value means that the object that is passed in as argument is duplicated within the function. The function then potentially alters the local copy of the object, not the object outside the function. Pass-by-reference, on the other hand, means that the object altered within the function is the same as the object passed in from outside.

All arguments in Python is pass-by-reference. For example, if we create a list outside of a function and try to change the values of its elements through a function, the list will remain altered after the function finishes executing. In the following example, we created “mylist” outside of the function “add_one”; however, after it is passed into “add_one” and the function exits, “mylist” still has 1 appended to its end.

```
>>> def add_one(mylist):
...     mylist.append(1)
...
>>> mylist = [10, 20, 30, 40]
>>> add_one(mylist)
>>> mylist
[10, 20, 30, 40, 1]
```

The situation in R is quite tricky. Pass-by-reference is possible using class “environment”; however, we haven’t learned about this class yet so we will disregard that method for now. Other than that, functions in R are pass-by-value.

Using a similar example as above, we create a vector that holds a sequence of integers 1, 2, 3. After passing myvector into add_one and the function exists, myvector still holds the value 1, 2, 3.

```r
add_one <- function(myvector) {
  mylist <- c(myvector, 1)
  return(myvector)
}
myvector <- c(1,2,3)
add_one(myvector) #Only in the function, mylist has an appended 1
```

```
## [1] 1 2 3
```

```r
myvector #Still the same value: 1, 2, 3
```

```
## [1] 1 2 3
```

Scoping in Functions

Scoping is a new concept in properties of functions that I am introducing. Scoping helps us build better tools from functions and avoid unusual errors. Scoping is centered on the idea of environment.

Environment in R

Environment is a set of objects, including functions and variables. The most common and important environment, Global environment (R_GlobalEnv), is created when we first start typing code in the R interpreter. In RStudio, under the Environment tab on the upper right corner, variables and functions in the Global Environment is displayed. Each function defined creates a new environment that is local to the particular function.

We can use the ls() function to check what variables and functions are defined in the current environment. We can also use the environment() function to get the current environment.

The code below demonstrates a cascading of environments. You will see the change in environment as we step into functions.

```
f <- function(x){
   g <- function(y){
       print("Inside g")
       print(environment())
   }
   print("Inside f")
   print(environment())
   g(5)
}
print("In global")
```

```
## [1] "In global"
```

```
print(environment())
```

```
## <environment: R_GlobalEnv>
```

```
f(3)
```

```
## [1] "Inside f"
## <environment: 0x7f8ac3c94548>
## [1] "Inside g"
## <environment: 0x7f8ac3c99270>
```

Even though we defined function f and function g inside of f, we did not step into these functions yet. The global environment is thus printed first. When we call the f function, we step into this function and the environment of funciton f(0x10a69a580) is printed. Consequently, function g is called in f. Therefore, we step into the environment in g function (0x10a69a3c0).

Globel vs. Local Variables

Since we now understand the difference between global and local environments, we can move on to global and local variables. Global variables are not necessarily created in the global environment. Global variables are variables that can be accessed and changed from any part of the program. Local variables are those that exist in certain functions.

This all depends on the perspective of a function. For example, in the following nested function, a and b are global variables in the perspective from inner_func, while c is a local variable. However, from the perspective of outer_func, only a is a global variable, while b is a local variable. Outer_func cannot access variable c at all.

```
a <- 1
outer_func <- function(){
   b <- 2
   inner_func <- function(){
       c <- 3
   }
}
```

Remember the idea of pass-by-value in the first section? We always create a local duplicate of the argument inside a function; therefore, we won't be able to change any global variables. **However**, there is still a way to access global variables!

Accessing Global Variable

Introducing a new operator called superassignment operator: "<<-"! When using this operator within a function, it searches for the variable in the parent environment frames until it reaches environment. If the variable is still not found, it is created in the global environment. In the following example, the superassignment operator searches for a in the outer_func environment. It does not find a, so it goes to find it in the global environment. It still does not find a; therefore, a is created in the global environment and assigned.

```
outer_func <- function(){
   inner_func <- function(){
       a <<- 30
   }
   inner_func()
   print(a)
}
```

Conclusion

There are many more properties of functions that we can continue to explore outside of the lab and lecture materials. Functions are very powerful - they can not manipulate data but also plot graphs. Properly understanding the environment of functions and variables can help us correctly access data that we wish to access. Furthermore, we learned how to access any global variable if we wish using the superassignment operator, since R functions are pass-by-value. Finally, I hope that you realized the versatility and complexity of R function and its connection with the environment around it.

Sources:

1. https://www.tutorialspoint.com/python/python_functions.htm

2. http://adv-r.had.co.nz/Functions.html
3. http://effbot.org/zone/default-values.htm
4. https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial#what_are
5. http://adv-r.had.co.nz/Functions.html#function-components
6. http://adv-r.had.co.nz/Functions.html#lexical-scoping
7. https://www.programiz.com/r-programming/return-function