Introduction to Computational BioStatistics with R: functions

Erik Spence

SciNet HPC Consortium

22 September 2020

Today's slides

To find today's slides, go to the "Introduction to Computational BioStatistics with R" page, on the right, under Lectures, "Functions".

`https://support.scinet.utoronto.ca/education`

Today's class

Today we will visit the following topics:

- Functions.
- Arguments.
- Local versus global variables.
- Returning values.

Use functions!

What is a function?

- Functions are collections of commands which are bundled together into a single command.
- By bundling the collection of commands together, the commands can be re-used over and over again without re-typing them all out.
- You pass 'arguments' to the function, so that the collection of commands will do different things when different arguments are given to it.
- Functions usually 'return' a value, which is the result of the calculation or action the function performs.
- You've already used many of the built-in functions in R: print, is.vector, list, str, class...
- Of course, we are able to create our own custom functions to do specialized things for our research.

No really, use functions!

Do I really have to?

- Yes, yes, yes!
- DO NOT copy-and-paste code.
- If you find yourself experiencing the urge to copy-and-paste code, write a function that does whatever it is you are copying and pasting.
- In fact, if you ever do something more than once in your code, make a function to do it.
- If you find that your functions are used between different files of code, create libraries so that only ONE copy of a given function ever exists (don't copy code!). You can reference that file from as many other files as needed.
- Or, if you're particularly awesome, you can create your own R library of functions.

Create modular code

Why can't I copy-and-paste code?

- It's very difficult to find bugs (mistakes) in your code if you've got 50 copies of the same code block floating around in your script.
- By creating functions, you only need to fix one chunk of code if you make a mistake.
- Your code will be easier to read.
- Also, it's easier to test functions to make sure they work correctly (unit testing).
- By creating libraries of functions which you know work, you will save time because you'll be re-using code.
- "Modular code" means code which stands alone, only depends on other nearby functions, and can be tested on its own, outside of the whole program.

Use functions!

Defining functions

Functions are defined using the "function" function.

- The name of the function is declared by assigning the function to a variable name.
- Function names cannot start with a number.
- Just type the function name to see its definition.
- Indent your code blocks!
- Functions are run the same way that R's built-in functions are run.
- Assuming the function does not take any arguments, you just type the name of the function, with brackets.
- This is called "calling", or "invoking", the function.

```
>
> my.func <- function() {
+   cat("Hello adoring fans!\n")
+ }
>
> my.func
function() {
cat("Hello adoring fans!\n")
}
>
> my.func()
Hello adoring fans!
>
```

Code blocks

A few comments about code blocks.

- Code blocks are indicated using { and }.
- A code block is a chunk of code, used as the body of a function, the body of a loop, or the body of some other R functionality.
- It is customary to indent your code blocks, to make them easier to read, though R will work whether you indent your code or not.
- You will be penalized on your assignments if you do not indent your code blocks in your functions and scripts.
- Where you put your { and } is, to some extent, a matter of taste; there are many conventions out there.

Defining functions, with arguments

Functions usually take arguments.

- Arguments are simply listed in the definition of the function.
- You can list as many, or as few, arguments as you like. The brackets must be used.
- When called, that which is passed to the function is assigned to the variable declared in the function definition.
- That means, in this case, the value of "10" is given to the variable "a" within the definition of print.me.

```
>
> print.me <- function(a) {
+   cat(a, '\n')
+ }
>
> print.me(10)
10
>
> d <- 11
>
> print.me(d)
11
>
```

Defining functions, with arguments, continued

Functions are usually written with arguments in mind.

- If we don't use arguments, then the function can only be used for whatever parameters are "hard-coded" into the function.
- When you are writing a function with arguments, if you're not sure how to proceed, imagine that the argument already has a value, and write your code as if it does.
- When you write the function, of course, the arguments don't yet have values, they are just placeholders.
- The arguments only actually get values when the function is called.

```
>
> my.func
function() {
cat("Hello adoring fans!\n")
}
>
> my.fun.calc <- function(a) {
+   b <- 2 * a - 7
+   d <- a * b
+   cat(d, '\n')
+ }
>
```

Defining functions, multiple arguments

Functions can take multiple arguments.

- All non-optional arguments must be specified when a function is called.
- Notice that the argument values are assigned in the order they appear in the function declaration.

```
>
> print.me.2 <- function(a, b) {
+   cat("a is", a, "\n")
+   cat("b is", b, "\n")
+ }
>
> print.me.2(3, 4)
a is 3
b is 4
>
> print.me.2(6)
a is 6
Error in cat("b is", b, "\n") : argument
"b" is missing, with no default
>
```

Defining functions, default values

You can set default values for arguments.

- Arguments which are given default values are optional.
- Values of arguments can be specified explicitly when the function in invoked.
- In the example here, the default value is a string. If the default value was a number, there would be no quotes.
- Note that you must use the "=" to declare optional arguments, NOT "<-".
- The default values can be of any type (numeric, string, list).
- You can have as many optional arguments as you want.

```
> print.me.3 <- function(a,
+              b = "pants") {
+   cat("a is", a, "\n")
+   cat("b is", b, "\n")
+ }
>
> print.me.3(5, 6)
a is 5
b is 6
>
> print.me.3(7, b = 8)
a is 7
b is 8
>
> print.me.3(3)
a is 3
b is pants
```

Local variables

Where variables are 'declared' matters.

- 'Declared' means "defined for the first time".
- Variables which are declared within functions are only accessible from within the function.
- Such variables are "local" to the function, including those in the argument list.
- These variables do not exist outside the function.
- This is called "local scope".

```
>
> g <- function() {
+   b <- 10
+ }
>
> g()
>
> b
Error: object 'b' not found
>
```

Local variables, continued

Be careful which variable you are referencing.

- If you declare variables outside of functions, and they are named the same as variables within functions, they are still NOT the same variable.

```
>
> b <- "pants"
>
> g <- function() {
+   b <- 10
+ }
>
> g()
>
> b
[1] "pants"
>
```

Local variables, continued more

Even if variables have the same name, that does not mean that they are the same variable!

- All variables which are declared in a function are local to the function.
- This means they only exist within the function.
- This means that you can have multiple distinct variables with the same name, inside and outside of functions.
- This includes variables which are declared within the function argument list.

```
> a <- 10
>
> print.a <- function(a) {
+   a <- a + 3
+   cat(a, '\n')
+ }
>
> print.a(a)
13
>
> a
[1] 10
>
```

Do not reassign function arguments

The value an argument gets when a function is called should be respected.

- Do not do what is done in the code to the right.
- Do NOT reassign the value of a function argument, within the function.
- Doing this can lead to problems under certain circumstances.
- It is also considered bad form.

```
> a <- 10
>
> # Bad!
> print.a <- function(a) {
+   a <- a + 3
+   cat(a, '\n')
+ }
>
> # Good!
> print.b <- function(a) {
+   b <- a + 3
+   cat(b, '\n')
+ }
>
> print.a(a)
13
> print.b(a)
13
```

Local variables, continued even more

Here's a more-complicated example.

- Functions can be declared within each other.
- These functions then become local to the function in which they are declared.
- Why do that? Because you can then have a local "global" variable, which is not accessible outside the larger function.

```
> outer.func <- function() {
+   a <- 20
+   inner.func <- function() {
+     a <- 30
+     cat(a, '\n')
+   }
+   inner.func()
+   cat(a, '\n')
+ }
```

```
> a <- 10
```

```
> outer.func()
30
20
```

```
> a
[1] 10
```

```
> inner.func()
Error: could not find function "inner.func"
```

```
>
```

Global versus local variables

Where variables are declared matters.

- Variables which are declared within a function are called "local". They may only be accessed within the function.
- Variables which are declared outside of functions are called "global".
  - Global variables can be directly accessed from within functions, but this is a VERY BAD idea.
  - It's better to pass all information you need in your function as an argument.
  - Global variables can be modified within functions, but this is VERY VERY bad form. Just don't.
- If you start using global variables within functions, without passing the variables through the argument list, you will break the modularity of your code. Your code will become less portable and much harder to debug.

Global variables

As mentioned earlier, do not use global variables, directly, within functions.

- Variables defined outside of functions have "global scope".
- It's better to pass all information into your functions as arguments.
- R will first look for locally defined variables before looking for global variables.
- If your function depends upon global variables the modularity of your code may become broken.
- It's possible to modify global variables from within functions, but I'm not going to show you how.

```
> a <- 10
>
> # Wrong!
> print.b <- function() {
+   b <- a + 3
+   cat(b, '\n')
+ }
> print.b()
13
>
> # do this instead!
> print.b2 <- function(a) {
+   b <- a + 3
+   cat(b, '\n')
+ }
> print.b2(a)
13
>
```

Global variables, continued

We see this mistake all the time.

- DO NOT access global variables from within functions: pass them through the argument list!
- This includes your data!
- Pass your data into your functions, otherwise the modularity of your code will be broken.
- Again, this means that, if I move the function somewhere else, and "my.data" is not defined in that new location, the function will fail.

```
>
> my.data <- read.csv("mydata.csv")
>
> # Wrong!
> my.bad.analysis <- function() {
+   a <- my.data$somecolumn
+   b <- some.other.analysis(a)
+ }
>
> # Correct!
> my.good.analysis <- function(input.data) {
+   a <- input.data$somecolumn
+   b <- some.other.analysis(a)
+ }
>
> my.good.analysis(my.data)
>
```

R return statement

So far we've only used our function to print things. What if we need to return a value?

- Use the "return" statement to return ("give back") values from your function.
- The return statement must be the last command in the function.
- Recall that, if a returned value is not assigned to a variable, it is automatically printed to the screen.
- Do not put return values in functions that do not return something.

```
>
> double.vector <- function(x) {
+   z <- x * 2
+   return(z)
+ }
>
> y <- double.vector(10)
>
> y
[1] 20
>
> double.vector(y)
[1] 40
>
```

R return statement, continued

It is possible to return values without the return statement.

- If the function ends with a value, the value will be returned by the function.
- Notice how double.vector.2 has the same behaviour as double.vector.
- For clarity, use the return statement if your function is returning a value.
- You will lose marks in your assignments if you don't use the return statement in your functions, if your function is returning something.

```
>
> double.vector <- function(x) {
+   z <- x * 2
+   return(z)
+ }
>
> double.vector.2 <- function(x) {
+   x * 2
+ }
>
> double.vector(10)
[1] 20
>
> double.vector.2(10)
[1] 20
>
```

Do not use 'print' as a return statement

Do not use the 'print' statement to return values.

- Bizarrely, the 'print' statement returns what it's printing.
- Don't use the 'print' statement in place of a proper 'return' statement, even if it works. It's bad form.
- In fact, DO NOT use print statements in your assignments, period, unless "cat" does not work.
- You will lose marks on your assignments if you do this.
- Use 'cat' to print things, and 'return' to return things.

```
>
> print.double.x1 <- function(x) {
+   y <- x * 2
+   # Do not use 'print' in place of
+   # a 'return' statement!
+   print(y)
+ }
>
> z <- print.double.x1(1:10)
[1] 2 4 6 8 10 12 14 16 18 20
>
> z
[1] 2 4 6 8 10 12 14 16 18 20
>
```

Data frames, accessing columns within functions

If you use the $ to access a column within a function, you must "hard code" the name of the column into the function. This is not ideal.

Instead, it is better to use the column's name, as a string, to access the column within the function.

This gives the flexibility of changing the name of the column that is accessed by the function.

```
> # Not the best way.
> some.analysis.1 <- function(input.data) {
+   # This function can only work on "somecolumn".
+   a <- input.data$somecolumn
+   b <- some.other.analysis(a)
+ }
>
> # A better way.
> some.analysis.2 -> function(input.data,
+                     working.col = "somecolumn") {
+   # This function can work on any column.
+   a <- input.data[, working.col]
+   b <- some.other.analysis(a)
+ }
>
```

Enough to get started

There's obviously a lot more to learn about functions. Nonetheless, this is enough functionality to get you started, and to complete the second homework assignment.

Note that the second homework assignment is assigned today, not Thursday.