BIMM-143: INTRODUCTION TO BIOINFORMATICS (Lecture 6)

R Functions

https://bioboot.github.io/bimm143_S18/lectures/#6

Dr. Barry Grant

Overview: In R programming, you use functions to incorporate sets of commands that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub program and called when needed. To begin with we can think of an R function as basically a named piece of code written to carry out a specified task or set of tasks that can be easily used again and again.

In R, there are many hundreds of built-in functions like **summary(), dist(), mean()** etc. but you can also write your own functions. You just use the same language you always use in R, in the same file as the rest of your code if you like (or in a separate file that you can **source()** when needed). You can write a quick, one-line function or long elaborate functions.

As you become more proficient in R you will start to write and use your own functions all the time to make your code cleaner and less repetitive. Personally, I believe the best way to learn more about the inner workings of R functions (and hence how R itself works), is to write your own ones and learn from others by looking at the code of their functions! So lets get started.

First we will take some working code snippets and improve them by encapsulating their main activities in functions. This will mirror what you will typically do in your own work when you find yourself writing repetitive code that does essentially the same thing 3 or more times. Subsequent sections provide further details and background on writing basic functions, tips and techniques for troubleshooting and debugging your functions and finally a set of generally good practices to follow when writing your own functions.

Section 1: Improving analysis code by writing functions

A. Improve this regular R code by abstracting the main activities in your own new function. Note, we will go through this example together in the formal lecture. The main steps should entail running through the code to see if it works, simplifying to a core working code snippet, reducing any calculation duplication, and finally transferring your new streamlined code into a more useful function for you.

```
# (A. Can you improve this analysis code?
df <- data.frame(a=1:10, b=seq(200,400,length=10),c=11:20,d=NA)

df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))

df$b <- (df$b - min(df$a)) / (max(df$b) - min(df$b))

df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))

df$d <- (df$d - min(df$d)) / (max(df$a) - min(df$d))
```

B. Next improve the below example code for the analysis of protein drug interactions by abstracting the main activities in your own new function. Then answer questions 1 to 6 below. It is recommended that you start a new *Project* in RStudio in a new directory and then install the bio3d package noted in the R code below (N.B. you can use the command `install.packages("bio3d")` or the RStudio interface to do this).

Then run through the code to see if it works, fix any copy/paste errors before simplifying to a core working code snippet, reducing any calculation duplication, and finally transferring it into a more useful function for you.

```
# Can you improve this analysis code?
library(bio3d)
s1 <- read.pdb("4AKE")  # kinase with drug
s2 <- read.pdb("1AKE")  # kinase no drug
s3 <- read.pdb("1E4Y")  # kinase with drug

s1.chainA <- trim.pdb(s1, chain="A", elety="CA")
s2.chainA <- trim.pdb(s2, chain="A", elety="CA")
s3.chainA <- trim.pdb(s1, chain="A", elety="CA")

s1.b <- s1.chainA$atom$b
s2.b <- s2.chainA$atom$b
s3.b <- s3.chainA$atom$b

plotb3(s1.b, sse=s1.chainA, typ="l", ylab="Bfactor")
plotb3(s2.b, sse=s2.chainA, typ="l", ylab="Bfactor")
plotb3(s3.b, sse=s3.chainA, typ="l", ylab="Bfactor")
```

Q1. What type of object is returned from the read.pdb() function?

Q2. What does the trim.pdb() function do?

Q3. What input parameter would turn off the marginal black and grey rectangles in the plots and what do they represent in this case?

Q4. What would be a better plot to compare across the different proteins?

Q5. Which proteins are more similar to each other in their B-factor trends. How could you quantify this? HINT: try the rbind(), dist() and hclust() functions together with a resulting dendrogram plot. Look up the documentation to see what each of these functions does.

```
hc <- hclust( dist( rbind(s1.b, s2.b, s3.b) ) )
plot(hc)
```

Homework with scoring rubric.

Q6. How would you generalize the original code above to work with any set of input protein structures?

Write your own function starting from the code above that analyzes protein drug interactions by reading in any protein PDB data and outputs a plot for the specified protein. (See class lecture 9 for further details).

Submit by email (to our IA, Alena, amartsul@ucsd.edu) your R markdown file (.Rmd) and either an HTML (.nb.html) or PDF (.pdf) file with the code and output saved.

scoring rubric:

Total 10 points assigned as follows:

Documentation:
1 pt - comments on what are the inputs to the function.
1 pt - what the function does and how to use it.
1 pt - what is the output of the function.

Code:

2 pt - function behaves as desired, producing the correct output and follows assignment specifications.

2 pt - the code is efficient meaning it uses best practices such as limiting calculation duplication.

2 pt - code is readable, meaning best practices are used including proper indentation and whitespace used, relevant variable names, and organized in a logical manner.

1 pt - function code and call executes and is working properly.

Section 2: Writing and calling a function

A function needs to have a name, probably at least one argument (although it doesn't have to), and a body of code that does something. At the end it usually should (although doesn't have to) return an object out of the function. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function. But you can "return" the value of the object from the function, meaning pass the value of it into the global environment. I'll go over this in more detail.

Functions need to have curly braces around the statements, like so:

```
name.of.function <- function(argument1, argument2) {
    statements
    return(something)
}
```

The argument can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

As a very simple example, we can write a function that squares an incoming argument. The function below takes the argument x and multiplies it by itself. It saves this value into the object called square, and then it returns the value of the object square.

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

I can now call the function by passing in a scalar or a vector or matrix as its argument, because all of those objects will square nicely. But it won't work if I input a character as its argument because although it will pass "hi" into the function, R can't multiply "hi".

```
# square a number
square.it(5)

## [1] 25

# square a vector
square.it(c(1, 4, 2))

## [1]  1 16  4

# square a character (not going to happen)
square.it("hi")

## Error: non-numeric argument to binary operator
```

I can also pass in an object that I already have saved. For example, here I have a matrix called matrix1, so I pass that into the **square.it() function**. R takes this matrix1 into the function as x. That is, in the local function environment it is now called x, where it is squared, and returned.

```
matrix1 <- cbind(c(3, 10), c(4, 5))
square.it(matrix1)

##      [,1] [,2]
## [1,]    9   16
## [2,]  100   25
```

Local vs global environment

Now, it's not necessarily the case that you must use **return()** at the end of your function. The reason you return an object is if you've saved the value of your statements into an object inside the function – in this case, the objects in the function are in a local environment and won't appear in your global environment. See how it works in the following two examples:

```
fun1 <- function(x) {
    3 * x - 1
}
fun1(5)

## [1] 14

fun2 <- function(x) {
    y <- 3 * x - 1
}
fun2(5)
```

In the first function, I just evaluate the statement 3*x-1 without saving it anywhere inside the function. So when I run fun1(5), the result comes popping out of the function. However, when I call fun2(5), nothing happens. That's because the object y that I saved my result into doesn't exist outside the function and I haven't used **return(y)** to pass the value of y outside the function. When I try to print y, it doesn't exist because it was created in the local environment of the function.

```
print(y)

## Error: object 'y' not found
```

I can return the *value* of y using the **return(y)** at the end of the function fun2, but I can't return the object itself; it's stuck inside the function.

Section 3: Getting more complex

Obviously writing a whole function to square something when you could just use the ^ operator is silly. But you can do much more complicated things in functions, once you get the hang of them.

Calling other functions and passing multiple arguments

First, you can pass multiple arguments into a function and you can call other functions within your function. Here's an example. I'm passing in 3 arguments which I want to be a matrix, a vector, and a scalar. In the function body, I first call my previous function **square.it()** and use it to square the scalar. Then I multiply the matrix by the vector. Then I multiply those two results together and return the final object.

```
my.fun <- function(X.matrix, y.vec, z.scalar) {

    # use my previous function square.it() and save result
    sq.scalar <- square.it(z.scalar)

    # multiply the matrix by the vector using %*% operator
    mult <- X.matrix %*% y.vec

    # multiply the resulting objects together to get a final ans
    final <- mult * sq.scalar

    # return the result
    return(final)
}
```

When you have multiple arguments in a function that you call, R will just evaluate them in order of how you've written the function (the first argument will correspond to X.matrix, the second y.vec, and so on), but for clarity I would name the arguments in the function call. In this example below, I already have two saved objects, my.mat and my.vec that I pass through as the X.matrix and y.vec arguments, and then I just assign the z.scalar argument the number 9.

```
# save a matrix and a vector object
my.mat <- cbind(c(1, 3, 4), c(5, 4, 3))
my.vec <- c(4, 3)

# pass my.mat and my.vec into the my.fun function
my.fun(X.matrix = my.mat, y.vec = my.vec, z.scalar = 9)

##      [,1]
## [1,] 1539
## [2,] 1944
## [3,] 2025

# this is the same as
my.fun(my.mat, my.vec, 9)

##      [,1]
## [1,] 1539
## [2,] 1944
## [3,] 2025
```

Returning a list of objects

Also, if you need to return multiple objects from a function, you can use **list()** to list them together.

For example:

```
another.fun <- function(sq.matrix, vector) {

    # transpose matrix and square the vector
    step1 <- t(sq.matrix)
    step2 <- vector * vector

    # save both results in a list and return
    final <- list(step1, step2)
    return(final)
}

# call the function and save result in object called outcome
outcome <- another.fun(sq.matrix = cbind(c(1, 2), c(3, 4)),
vector = c(2, 3))

# print the outcome list
print(outcome)

## [[1]]
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
##
## [[2]]
## [1] 4 9
```

Now to separate those objects for use in your further code, you can extract them using the [[ ]] operator:

```
### extract first in list
outcome[[1]]

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4


## extract second in list
outcome[[2]]

## [1] 4 9
```

Section 4: Tricks for troubleshooting and debugging

When you execute multiple statements in a function, sometimes things go wrong. What's nice about functions is that R evalutes every statement until it reaches an error. So in the last function, the dimensions of the objects really matter. You can't multiply matrices of incomptabile dimensions. Like this:

```
my.fun(X.matrix = my.mat, y.vec = c(2, 3, 6, 4, 1), z.scalar =
9)

## Error: non-conformable arguments
```

Using the Debug() function

When you have an error, one thing you can do is use R's built-in debugger **debug()** to find at what point the error occurs. You indicate which function you want to debug, then run your statement calling the function, and R shows you at what point the function stops because of errors:

```
debug(my.fun)
my.fun(X.matrix = my.mat, y.vec = c(2, 3, 6, 4, 1), z.scalar =
9)

## debugging in: my.fun(X.matrix = my.mat, y.vec = c(2, 3, 6, 4,
1), z.scalar = 9)
## debug at #1: {
##     sq.scalar <- square.it(z.scalar)
##     mult <- X.matrix %*% y.vec
##     final <- mult * sq.scalar
##     return(final)
## }
## debug at #4: sq.scalar <- square.it(z.scalar)
## debug at #7: mult <- X.matrix %*% y.vec

## Error: non-conformable arguments
```

We see that the first line calling the **square.it()** function was fine, but then an error occurred in the line defining mult. This debugging is useful especially if you had many more statements in your function that multiplied matrices and you weren't sure which one was causing the issues. So now we know the problem is that X.matrix and y.vec won't multiply. But we still need to know why they won't multiply.

Printing out what's happening (sanity checks)

At this point, a good way to troubleshoot this is to print out the dimensions or lengths of the objects or even the objects themselves that are going into the statement causing errors. The great part about functions is that they evaluate all the way until there's an error. So you can see what is happening inside your function before the error.

If the object is too long, you can **print(head(object))**. This helps to see if you're doing what you think you're doing. Note that you have to use the function **print()** to actually print out anything from within a function.

```
my.fun <- function(X.matrix, y.vec, z.scalar) {
    print("xmatrix")
    print(X.matrix)

    print("yvec")
    print(y.vec)

    print("Dimensions")
    print(dim(X.matrix))
    print(length(y.vec))

    # use previous function square.it() and save result
    sq.scalar <- square.it(z.scalar)
    print(paste("sq.scalar=", sq.scalar))

    # multiply the matrix by the vector using %*% operator
    mult <- X.matrix %*% y.vec

    # multiply the two resulting objects
    final <- mult * sq.scalar

    # return the result
    return(final)
}

my.fun(X.matrix = my.mat, y.vec = c(2, 3, 6, 4, 1), z.scalar =
9)

## [1] "xmatrix"
##      [,1] [,2]
## [1,]    1    5
## [2,]    3    4
## [3,]    4    3
## [1] "yvec"
## [1] 2 3 6 4 1
## [1] "Dimensions"
## [1] 3 2
## [1] 5
## [1] "sq.scalar= 81"

## Error: non-conformable arguments
```

Now we can see the actual dimensions of the objects and fix them accordingly. This example is really simple, but you can imagine that if you've written a long function that uses many arguments, you could easily lose track of them and not be sure where the issue in your function was. You can also throw in these statements along the way as sanity checks to make sure that things are proceeding as you think they should, even if there isn't any error.

Using the stop() and stopifnot() functions to write your own error messages

One other trick you can use is writing your own error messages using the **stop()** and **stopifnot()** functions. In this example, if I know I need dimensions to be the right size, I can check them and print out a message that says they are incorrect. That way I know what the issue is immediately. Here's an example:

```
my.second.fun <- function(matrix, vector) {

    if (dim(matrix)[2] != length(vector)) {
        stop("Can't multiply matrix%*%vector because the
dimensions are wrong")
    }

    product <- matrix %*% vector

    return(product)

}

# function works when dimensions are right
my.second.fun(my.mat, c(6, 5))

##      [,1]
## [1,]   31
## [2,]   38
## [3,]   39

# function call triggered error
my.second.fun(my.mat, c(6, 5, 7))

## Error: Can't multiply matrix%**%vector because the dimensions
are wrong
```

You can do these kinds of error messages for yourself as checks so you know exactly what triggered the error. You can think about putting in a check for if the value of an object is 0 if you are dividing by it as another example.

Section 5: Good function writing practices

Based on my experience, there are a few good practices that I would recommend keeping in mind when writing function.

1. Keep your functions short. Remember you can use them to call other functions!
    - If things start to get very long, you can probably split up your function into more manageable chunks that call other functions. This makes your code cleaner and easily testable.
    - It also makes your code easy to update. You only have to change one function and every other function that uses that function will also be automatically updated.
2. Put in comments on what are the inputs to the function, what the function does, and what is the output.
3. Check for errors along the way.
    - Try out your function with simple examples to make sure it's working properly
    - Use debugging and error messages, as well as sanity checks as you build your function.