Functions

## Learning Objectives

* Functions have two parts: arguments and body
* Functions have their own environment
* Convert code into functions to repeat operations

In the last lesson we wrote loops to performs some calculations. But what if we wanted to perform similar calculations on different columns? We would have to copy-paste the loops and change all the variable names. This strategy would be both tedious, error prone, and difficult to update if we want to make a change. To avoid these problems, we will review how to write our own functions.

### The parts of a function

We've already been using built-in R functions: read.delim, mean, apply, etc. These functions allow us to run the same routine with different inputs.

Let's explore read.delim further. All functions in R have two parts: the input arguments and the body. We can see the arguments of a function with the args.

args(read.delim)

function (file, header = TRUE, sep = "\t", quote = "\"", dec = ".",   
 fill = TRUE, comment.char = "", ...)   
NULL

So when we pass a character vector like "data/counts-raw.txt.gz", this gets assigned to the argument file. All the other arguments have defaults set, so we do not need to assign them a value.

After the arguments have been assigned values, then the body of the function is executed. We can view the body of a function with body.

body(read.delim)

read.table(file = file, header = header, sep = sep, quote = quote,   
 dec = dec, fill = fill, comment.char = comment.char, ...)

read.delim is very short. It just calls another function, read.table, using the input file and the default arguments as the arguments passed to read.table.

When we define our own functions, we use the syntax below. We list the arguments, separated by commas, within the parentheses. The body follows, contained within curly brackets {}.

function\_name <- function(args) {  
 body  
}

### The principle of encapsulation

An important feature of functions is the principle of encapsulation: the environment inside the function is distinct from the environment outside the function. In other words, variables defined inside a function are separate from variables defined outside the function.

Here's an small example to demonstrate this idea. The function ex\_fun takes two input arguments, x and y. It calculates z and returns its value.

ex\_fun <- function(x, y) {  
 z <- x - y  
 return(z)  
}

When we run ex\_fun, the only thing returned to the global environment is the value that was assigned to z. The variable z itself was only defined in the function environment, and does not exist in the global environment.

ex\_fun(3, 10)

[1] -7

#z

## Environments are complicated

The situation presented above is a simplified version of environments which will serve you well if you treat functions as truly encapsulated. In reality, things are more complicated. For example, if inside a function you have a variable that has not been defined in the function, it will actually search the global environment for this variable. To learn the advanced details, see the chapter [Environments](http://adv-r.had.co.nz/Environments.html) in Advanced R by Hadley Wickham.

## The return statement

R provides the shortcut of not needing to use return at the end of the function. Instead, the variable on the last line of the body of the function is returned. This is useful for writing very small functions, but in these lessons we will use return to be more explicit about what is happening.

### Examples

In the last lesson we wrote the following for loop to calculate the mean number of citations for each journal. Let's generalize this code to a function so that we can perform a similar calculation for any of the metrics across any of the categorical variables.

result <- numeric(length = length(levels(counts\_raw$journal)))  
names(result) <- levels(counts\_raw$journal)  
for (j in levels(counts\_raw$journal)) {  
 result[j] <- mean(counts\_raw$wosCountThru2011[counts\_raw$journal == j])  
}  
result

pbio pcbi pgen pmed pntd pone ppat   
28.705905 14.219258 22.928208 18.148110 7.348564 8.306972 20.892613

We'll name the function mean\_metric\_per\_var, and it will take two input arguments: metric and variable. The outline of our function looks like this.

mean\_metric\_per\_var <- function(metric, variable) {  
 # body goes here  
}

Now we can copy paste our loop code into the body of the function. We indent the code by two spaces as a convention to aid readability, it doesn't actually affect the ability of the code to run (to indent in RStudio you can highlight all the lines and press Ctrl-I).

mean\_metric\_per\_var <- function(metric, variable) {  
 result <- numeric(length = length(levels(counts\_raw$journal)))  
 names(result) <- levels(counts\_raw$journal)  
 for (j in levels(counts\_raw$journal)) {  
 result[j] <- mean(counts\_raw$wosCountThru2011[counts\_raw$journal == j])  
 }  
 result  
}

Now we need to replace the specific data we used, the journal and the 2011 citations, with the names of the function arguments. We'll also add the return.

mean\_metric\_per\_var <- function(metric, variable) {  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (j in levels(variable)) {  
 result[j] <- mean(metric[variable == j])  
 }  
 return(result)  
}

Lastly, instead of naming the looping variable j for "journal", let's change it to v for "variable"

mean\_metric\_per\_var <- function(metric, variable) {  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (v in levels(variable)) {  
 result[v] <- mean(metric[variable == v])  
 }  
 return(result)  
}

Now we can run the same analysis we did before:

mean\_metric\_per\_var(counts\_raw$wosCountThru2011, counts\_raw$journal)

pbio pcbi pgen pmed pntd pone ppat   
28.705905 14.219258 22.928208 18.148110 7.348564 8.306972 20.892613

Or a new analysis, like the mean number of tweets grouped by the type of article.

# Warning: Lots of ouput.  
# Not evaluated for printout  
mean\_metric\_per\_var(counts\_raw$backtweetsCount, counts\_raw$articleType)

The other loop we wrote used apply to calculate the mean of multiple metrics for each article, i.e. row, of the data frame.

counts\_sub <- counts\_raw[, c("wosCountThru2011", "backtweetsCount", "plosCommentCount")]  
sum\_stat <- apply(counts\_sub, 1, mean)  
summary(sum\_stat)

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 0.0000 0.6667 2.0000 4.7060 5.0000 245.7000

Let's generalize this to a function where we can choose which columns to include in the mean summary statistic. We'll call it calc\_sum\_stat, and it will take two input arguments: the data frame and a vector of the columns to select. Here's the outline of the function.

calc\_sum\_stat <- function(df, cols) {  
}

Now we copy-paste our previous code into the body of the function and indent.

calc\_sum\_stat <- function(df, cols) {  
 counts\_sub <- counts\_raw[, c("wosCountThru2011", "backtweetsCount", "plosCommentCount")]  
 sum\_stat <- apply(counts\_sub, 1, mean)  
 summary(sum\_stat)  
}

Also, replace the specific variable names with the argument names and add return.

calc\_sum\_stat <- function(df, cols) {  
 df\_sub <- df[, cols]  
 sum\_stat <- apply(df\_sub, 1, mean)  
 return(sum\_stat)  
}

Now we can perform the same analysis as before:

sum\_stat\_1 <- calc\_sum\_stat(counts\_raw, c("wosCountThru2011", "backtweetsCount", "plosCommentCount"))  
summary(sum\_stat\_1)

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 0.0000 0.6667 2.0000 4.7060 5.0000 245.7000

Or choose different metrics to summarize:

sum\_stat\_2 <- calc\_sum\_stat(counts\_raw, c("wosCountThru2010", "f1000Factor"))  
summary(sum\_stat\_2)

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 0.000 0.000 1.000 4.116 4.000 315.500

As we have seen, writing functions allows us to repeat operations without having to copy-paste code. In later lessons, we will learn how to debug functions when they are not working as expected.

### Challenges

## Write your own function

Write your own function to calculate the mean called my\_mean. It should take one input argument, x, which is a numeric vector. Compare your results with the results from R's function mean. Do you receive the same answer?

my\_mean <- function(x) {  
 result <- sum(x) / length(x)  
 return(result)  
}  
my\_mean(1:10)

[1] 5.5

mean(1:10)

[1] 5.5