Lab 3: Functional Programming

Table of Contents

You can download the .Rmd file [here](lab-3.Rmd). You can download the .doc file [here](lab-3.docx).

# Purpose

In today’s lab we will focus on understanding and building functions. R is an object oriented programming language, you can create objects in R and assign them specific names. Functions are a type of object which means that the useR has the power to create custom functions, which are saved to names. Functions are a fundamental part of data analysis in R. They help with carrying out coding operations in a swift and precise manner. They help to make workflows more dynamic and customized. Once you know how to build your own functions, coding in R becomes easier and you save a lot of time when analyzing data! It’s one of those skills which demands tall investments and needs time to grow. However once you reach a certain level of growth the yields are sizable and worth all the investment.

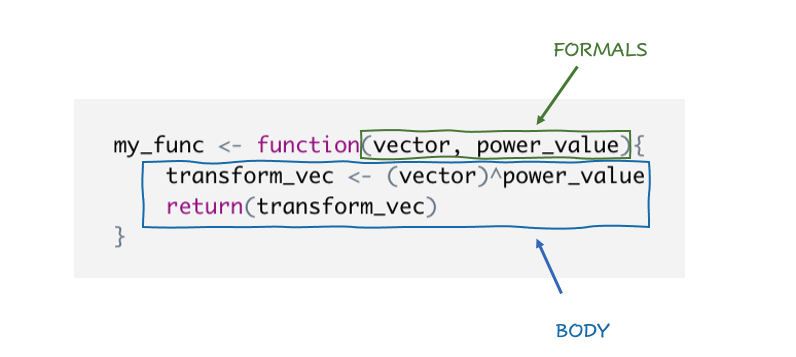
# Why use functions

Let’s talk a little more about why you would want to invest in this skill of function building?

1. Functions help in organizing your workflow. They reduce errors and help keep a track of complex tasks.
2. Functions allow for developing and maintaining algorithms specific to your project.
3. Functions support reproducibility. If you build a function and embed it in a R package, anyone who is able to download the package is likely to have access to the function.

# Components of a function

Functions in R have three components: - body() - the part inside the function - formals() - consist of arguments which direct how you use the function - environment() - location of the function

Let’s consider an example of a function. # 

The two arguments in this function are vector and power\_value. The function takes inputs for these arguments and spits out the output based on the directions specified in the function. These arguments consist the . Next we will need to specify how the inputs of the arguments will be manipulated. The part which specifies the tasks that the function does with the inputs is the of the function. The location wherein the function is mapped is the . We won’t go into the details of the environment in this lab.

Let’s run the following code to see how this function works.

my\_func <- function(vector, power\_value){  
 transform\_vec <- (vector)^power\_value  
 return(transform\_vec)  
}

Now let’s look at the components of the function.

#Peeking into the components of the function  
formals(my\_func)  
body(my\_func)  
environment(my\_func)

As you can see you can peek into the structure of the function in R. This allows for understanding the mechanics of how the data is being manipulated as we build the function.

#A vector x   
x <- c(5, 9, 8, 10)  
  
#Applying the function to the vector x  
my\_func(x, 2)

Let’s build a function together. You studied about *z-scores* scores last week in [lab-2](https://uopsych.github.io/psy611/labs/lab-2.html#z-scores). Recall that they are also called . The formula to compute *z-scores* is

Let’s build a function to compute z scores for a vector in R.

**Step** **1** - identify the input vector In the formula above, will be the input vector. Let’s build a random input vector x for the purpose of this example. Starting with a made-up input helps to picture and take decisions about the type of data your function will take. I highly recommend starting with this step, but your approach may differ and that’s totally fine.

x <- c(5, 9, 8, 10, 14, 18, 19, 24, 35)

**Step** **2** - compute values needed  
In the formula above you need to compute *mean* and *sd* in order to compute the *z-score*. Let’s compute the *mean* and *sd* of the vector *x*. I will save the values in objects *m* and *s*

m = mean(x)  
s = sd(x)

Now, you can compute the *z-scores*

z = ((x - m)/s)

At this point you can see that you have all the elements you need to build a function for computing z-scores.

**Step** **3** - Wrap the objects in a function At this stage you create the function object and name the function object.

z\_scores <- function(x){  
 m = mean(x)  
 s = sd(x)  
 z = ((x - m)/s)  
}  
  
z\_scores(x)

You can see R run the function but you cannot see the output. You need to tell R to return the values in z. This is how R knows which value(s) from the body to output.

z\_scores <- function(x){  
 m = mean(x)  
 s = sd(x)  
 z = ((x - m)/s)  
 return(z)  
}  
  
z\_scores(x)

You have a function ready!

You can add more layers to this function, like the error message I have added here. I am stopping the function from doing any further operations if the input vector is non-numeric.

z\_scores <- function(x){  
   
 if(!is.numeric(x)) {stop("x is not numeric")} else   
 return(x)  
 m = mean(x)  
 s = sd(x)  
 z = ((x - m)/s)  
 return(z)  
 return(message)  
}  
  
#With numeric vector  
z\_scores(x = c(5, 9, 8, 10, 14, 18, 19, 24, 35))  
  
#With character (non-numeric) vector  
z\_scores(x= c("apple", "banana"))

scale function in the {base} package can carry out the same task.

scale(x = c(5, 9, 8, 10, 14, 18, 19, 24, 35))  
  
scale(x= c("apple", "banana"))

You can see that the output from the scale function is a matrix. There are times when obtaining a data frame or a tibble format for your output may be more helpful (for e.g., when you are using {tidyverse}. You will learn more about {tidyverse} in the future). The z\_scores function, we just built, helps with this issue by returning the output as a numeric vector. When you write your functions you could also specify the formats of your output. It really does make life easier!

# Suggestions for writing functions

In this section we will talk about some of the practical aspects of writing functions. First, when you are analyzing data, you may come across instances which are ideal for building functions. It is helpful to identify them and recognize how functions can help your workflow. Secondly, writing functions is a skill, it takes time to understand and write good functions. It is also important to remember that a function that works better in one project may need modifications to be as effective in another. There is no one write answer to what a good function looks like.  
Here are some suggestions to get you started on how to think about writing functions. Please note that this section does not cover all the information there is on this topic, you are encouraged to explore beyond this lab.

## When to use functions?

1. When you find yourself repeating operations - for e.g., you need to compute mean, sd and range for a multiple columns in a data frame.
2. You want to reuse a set of operations at another point in your workflow - for e.g., you need to use a function you built to create a plot in four different places in your workflow.
3. When you need some flexibility (sometimes you can build arguments which can take more than one value) - for e.g., you may want to specify an argument na.rm as TRUE in certain cases and FALSE in others.
4. When you want to carry out iterations or simulations - for e.g., you may want to find the mean of a sub sample (subset of the larger sample) and ask R to do this 100 times.

## What is a good function?

1. One that does what it is exactly supposed to do - make sure you try the function on smaller vectors/values to test whether it is giving the desirable output.
2. One that has object names which are appropriate and easy to identify what they stand for - for e.g., if you are building a function to compute *Cohen’s* *d* you might want to consider naming the function Cohen\_d, it is within 10 characters and leaves no ambiguity in terms of what it does. There can be instances where function names exceed 10 characters which aid clarity (see bootnet::bootnet\_piecewiseIsing). The same also stands for object names in the body of the function.
3. One that is annotated - it is good practice to add comments and annotate your code especially if you want people to use your function.

# Minihacks

## Minihack 1: Components of Function

This is a function that computes confidence intervals for the mean of a variable. You don’t need to know what a confidence interval is at this point (although your function may come in handy later in the term!).

mean\_ci <- function(x, alpha) {  
 se = sd(x) / sqrt(length(x))  
 mean(x) + se \* qnorm(c(alpha / 2, 1 - alpha / 2))  
}

* What are the formals in this function?
* What is the body of this function?
* What is the environment of this function?

#Your code here

## Minihack 2: Write a function (guided)

Build a function to estimate the of a variable in R.

* First generate a vector random\_var which takes values from the 1 to 10. *Hint*: You can use cbind to do this or look up the seq function.
* Build an object vector col\_sum which contains the sum of all the values in random\_var.
* Build an object col\_length which contains the total number of values in vector random\_var. *Hint*: Look up length function.
* Build the vector m which will contain the mean of all values in vector random\_var. Hint: use the objects col\_sum and col\_length to do this.
* Now wrap col\_sum, col\_length and m in a function with two formals random\_var and na.rm. Don’t forget to set a default for na.rm (remember it takes logical inputs).
* Name this function object as my\_mean.
* Compute the mean of random\_var using the my\_mean function.
* Check if it matches the output from the mean function in base R.

#Your code here

## Minihack 3: Debugging

Use the z\_scores() function that we built above for this mini hack. Compute z-scores of the following vector.

x <- c(5, 9, 8, 10, 14, 18, NA, 19, 24, 35, NA)  
z\_scores(x)

* Does the function work? Why?
* Try fixing the bug in this function. Modify the functions so that it ignores NAs, and computes z-scores for all other values in the vector. *Hint* - Look up ?mean and ?sd, use the na.rm argument.

#Your code here

## Minihack 4: Write a function (hands off)

Sara mentioned in the [class](https://uopsych.github.io/psy611/lectures/04-describing_data.html#3) last week. Write a function to compute . Build a random vector and test the function.

#Your code here

## Minihack 5: Function and data types

* Build a function to compute [coefficient of variation](https://en.wikipedia.org/wiki/Coefficient_of_variation) for a vector.
* Use this function to compute the coefficient of variation for the two following vectors  
  x <- c(1:10)  
  y <- c(1:10, NA)

*Hint*: You can use the base R functions to compute and . You also want to explore the na.rm argument.

#Your code here