In the previous mission, you learned how to use control structures to direct the execution of your code. In addition to looping and if-else statements, what if you wrote a line of code and wanted to re-use it again *without* constant re-looping. What if you wanted to use certain lines of code for one specific case? Perhaps you've written a piece of code that fixes a nasty bug and you happened to come across that bug again. Or maybe, you automated a part of your system and you wanted to use the same code to automate another part.

To do this, we'll learn how to write **functions**. In this path, you've already used a variety of functions like mean(), class() etc. What we don't see is that when we write these functions out, the interpreter is pulling up code *already written* and then executing it. A function takes in some **input**, executes code using that input, and then returns an **output**.

Now, let's look at a function we've used earlier called the length() function. To review, using the length() function on a vector will return the number of elements inside the vector. Let's look at a vector called goals:



goals <- c(3,5,2,1,0,3)

Now, when calling the length() function, you're actually executing specific, hidden lines of R code:



length <- function(x){

count <- 0

for (val in x){

count <- count + 1

}

print(count)}

If we call length on goals, this would return:



length(goals)

Keep in mind, this is only one, narrow method of writing length(). There are many ways to write this implementation.

The **input**, goals, is also known as the parameter. Functions can have multiple inputs & multiple outputs depending on the code block. All functions follow the same pattern. Functions like mean() and length() are called **built-in** functions. These are functions the R interpreter has already defined for us.

In this mission, we'll write a function that predicts the scores of a soccer/football match & predicts the winner. The data we'll be using is fictitious. To kick it off, let's review the built-in functions we've learned!

instructions

* For the first game in the playoffs, we have two teams: thunderbirds, flamethrowers. Here are the number of goals for each team stored in a vector:

thunderbirds <- c(4,3,5,1,0,2,4,3,2,2,1,4)

flamethrowers <- c(2,4,6,0,3,4,2,3,3,2,1,0)

* Use the built-in mean() function to find the averages for each. Store the averages in thunderbirds\_mean and flamethrowers\_mean.

Now that we've calculated the average goals scored using mean(), let's dig under the hood to understand how to *write* our own mean() function. To write our own function, we'll need to understand the syntax behind writing a function:



name <- function(x){

expression

}

**name**: This is the name of the function. Whenever we want to call our function, we'll use this name.

**function**: This is the keyword for the R interpreter to interpret this line of code as a function.

**x**: This is the input parameter for the function. You can name the input parameter anything you want. If you have more than one, make sure you name it something different i.e (x,y).

**expression**: This is the code the function executes.

The result of the expression will automatically be returned. Let's take two teams with two different goal amounts: 4 and 2. Let's write a function that return the *difference* between these two goals:



difference <- function(x,y){

x - y

}

Let's call our function using 4 and 2 as the input values:



difference(4,2)

And this would return:



2

Now, let's write our own sum function!

instructions

* Define a function called get\_sum() that has two input parameters. Use the following two goals scored to get total goals:

thunderbirds <- 3

flamethrowers <- 1

* Store in total\_goals.

Congratulations! You've written your first function! Notice that our input parameters are x and y:



difference <- function(x,y){

x - y

}

Can we access the x & y variables *outside* of the function? No, we can't. When we use variables used inside of a function, we can only use variables inside the function. We *cannot* use these variables outside of our function. For example, let's run the following code:



difference <- function(x,y){

x - y

}

​

print(x)

The interpreter will display the following error:



Error in print(x): object 'x' not found

Traceback:

​

1. print(x)

Any variable or object we create in our function can only be used inside the function. We can use the object to return an output of our function. However, we cannot *call* the object outside of the function. Let's try to call a variable outside of our function!

instructions

* Using our get\_sum() function, print the y variable outside the fuctnion. Notice the error.

Now that we've written a sum & difference function, let's take it one step further by re-creating the mean() function. The mean is defined as adding up all your values and then dividing by the total number of values. To be able to create the mean() function, we'll need to understand another attribute of a function, *calling functions within a function*. Let's look at an example of using our difference function. Let's call the absolute value function on the difference between goals to make sure we return a non-negative value (abs()):



difference <- function(x,y){

abs(x - y)

}

Now, let's subtract the goals of two teams:



thunderbirds <- 3

flamethrowers <- 1

​

difference(flamethrowers,thunderbirds)

This would display:



2

Now, let's build the mean function!

instructions

* Write a new function called get\_mean().
* Call the sum() and length() functions within get\_mean().
* Calculate the mean for the following team goals:

thunderbirds <- c(4,3,5,1,0,2,4,3,2,2,1,4)

flamethrowers <- c(2,4,6,0,3,4,2,3,3,2,1,0)

* Store these means in thunderbirds\_mean and flamethrowers\_mean.

Now that we've built our own mean() function and have understood the basics of functions, let's start building our soccer match prediction function. To review, let's look at the syntax of an R function:



name <- function(x){

expression

}

In addition to our current function, we can also specify the value we want our function to return by adding return:



name <- function(x){

output <- expression

return(output)

}

Using return is useful when you have more complex functions with multiple expressions. Using return, let's predict the scores of the soccer match!

instructions

* We've provided the following vectors:

thunderbirds <- c(4,3,5,1,0,2,4,3,2,2,1,4)

flamethrowers <- c(2,4,6,0,3,4,2,3,3,2,1,0)

* Write a function called get\_predictions().
* Within the function, perform the following:
  + Find the mean of both input vectors. Store both in separate variables.
  + Store both these variables in a vector called predictions.
  + Return the vector as the output
* Run your function on the two vectors and store the results in predicted\_score.

Congratulations! You've written a function that predicts the scores of two teams playing against each other. Now, let's add an additional feature that will print the *name* of the team that wins. To do this, we'll need to add an if-else statement to our function. If you'd like to review control structures, feel free to turn back to the previous mission.

To predict the name of the winner, we'll need our function to have two possible values, either team A or team B. To have our function return two values, we'll need to use return twice. Let's look to our difference function:



difference <- function(x,y){

x - y

}

Now, let's add an if-else statement to difference. If the difference is positive, we'll have our function print "positive". If the difference is negative, we'll have the function print negative:



difference <- function(x,y){

diff <- x - y

if (diff >0){

return("positive")

} else{

return ("negative")

}

}

And then running the following:



difference(4,2)

This will return:



[1] 'positive

Now, let's add an if-else statement to our prediction function!

instructions

* Add a condition in the function that prints the name of the team with the higher score.
* Add two additional arguments into the function, denoting the teams of both names. Call these x\_name,y\_name. The input for these will be the team names "thunderbirds" and "flamethrowers". We're doing this so the function has a name to return.
* Call this function on thunderbirds and flamethrowers vectors. Store the results of this in winner.

In review, we'ved learned a few key concepts regarding functions:

* We dove deeper into built-in functions.
* We learned how to write our own functions.
* We learned how to incorporate control structures into our functions.

In the next session, we'll introduce you to the concept of writing clear, understandeable code. We'll also introduce you to the alternative to control structures: the apply family.