Over the last few missions, we've learned how to use loops, functions & conditional structures to analyze the goals scored for the red dragons & deep sea sailors soccer/football teams. Due to your great work with these teams, FIFA has decided to hire you to crunch their 2014 World Cup data. FIFA is the association that governs world football/soccer and organizes the immensely popular FIFA world cup.

FIFA would like to understand the number of goals scored for each team participating in the world cup. To analyze goals scored, we'll be using [FiveThirtyEight's 2014 World Cup Forecasts and Scores](https://www.kaggle.com/fivethirtyeight/world-cup) dataset. Here are the column descriptions of the dataset:

* match\_id: The ID of the match played.
* match\_date: The date the match was played.
* match\_time: The time the match was played.
* home\_country: The name of the home country.
* home\_code: The code corresponding to the home country.
* home\_goals: The number of goals the home team scored.
* away\_country: The name of the away country.
* away\_code: The code corresponding to the away country.
* away\_goals: The number of goals the away team scored.
* win\_country: The name of the country that won the match.
* win\_code: The code corresponding to the country that won the match.

Now, let's load our dataset!

instructions

* Load the csv file "scores.csv" and store this in df\_scores.

In this track, we've learned a variety of R programming concepts such as loops, control structures, functions etc. As you improve in your R programming skillset, you'll notice that there are multiple solutions to a problem. Many times, it won't matter which solution is better, as long as the program executes successfully. Other times, when you want to productionize your model, analyze a large dataset, or have someone else review your work, certain solutions work better than others.

When there are multiple ways of solving a problem, how do we pick? To choose the best solution, let's consider two factors:

**1. Ease of Understanding:** Programming is like writing an essay. If we want someone else to review & understand our code, with little friction, we'll want to make our code easily *readable*. For example, let's say we wanted to figure out the total number of goals scored by a team in a season:



goals <- c(4,3,2,3,3,4,2,1,0)

We *could* use a for loop to get the total number of goals:



total\_goals <- 0

for (g in goals){

total\_goals <- total\_goals + g

}

Printing total\_goals would display:



[1] 22

This for loop isn't incomprehensible. However, can we simplify this code? Yes we can!



goals <- c(4,3,2,3,3,4,2,1,0)

​

total\_goals <- sum(goals)

Printing total\_goals would also display:



[1] 22

Two different solutions that display the same answer. Since the second solution is easier to understand, we'll go with sum(goals).

**2. Efficient Runtime:** Runtime is the amount of time it takes for the R interpreter to execute your code. There will be times where writing efficient code will conflict with ease of understanding and vice versa. Deciding on whether you want to optimize your code towards understanding or runtime will be context dependent. As a general rule, writing code that runs efficiently will make both your lives & your engineer/database administrators lives a lot easier.

To see how fast your code runs, you can use the system.time({expression}) function. Let's compare the runtimes of our two current expressions:



system.time({

total\_goals <- 0

for (g in goals){

total\_goals <- total\_goals + g

}

})

The runtime for this code is:



user system elapsed

0.002 0.000 0.002



system.time({

goals <- c(4,3,2,3,3,4,2,1,0)

​

total\_goals <- sum(goals)

})

The runtime for this code is:



user system elapsed

0 0 0

Looks like sum(goals) is both easier to understand and much faster. For the context of this mission, don't worry too much about calculating the runtimes. The goal, is to be aware of runtime. Because we aren't dealing with production level code or huge datasets, there isn't a need to time everything. For this mission, we'll focus more on writing easy to understand code. We'll dive into more in-depth code runtime optimization strategies in later, more advanced courses.

instructions

* We've stored all the total goals scored in each game in a single total\_goals\_per\_game vector.
* Look at the current code that calculates the average number of goals per game. Re-write this code to make it easier to understand. Store this in average\_goals.
* Print average\_goals.

In the previous screen, notice how we replaced the entire for loop with one, built-in function. In that scenario, since we're looking to calculate the average, we can use the mean() function. Rather than calculate the average goals scored for each team, what if we wanted to calculate the average total goals scored?

In the first mission, you used a for loop to loop through the following vector to find the total number of goals scored in each match:



matches <- list(c(5,4),c(2,1),c(4,1),c(7,5),c(3,5),c(3,3),c(2,3),c(4,2))

​

for (match in matches){

total\_goals <- c(total\_goals,sum(match))

}

Now that we've learned the concept of making our code easier to understand, is there an easier way of finding the total? Yes there is! We can iterate over each value in this list by using the apply family. apply will take a data structure with a number of values, and *apply* a function on every item within the structure. The output will be a data structure containing the new values.

Returning to our previous example:



matches <- list(c(5,4),c(2,1),c(4,1),c(7,5),c(3,5),c(3,3),c(2,3),c(4,2))

​

for (match in matches){

total\_goals <- c(total\_goals,sum(match))

}

In this scenario, because matches is a list, we'll be using lapply. lapply is an apply function within the apply family, that takes in a list or vector, applies the function to the list, and returns a *list*. The syntax of lapply is as follows:



lapply(x, function)

**x**: The list or vector we want to apply our function to.

**function**: The function we're using.



total\_goals <- lapply(matches,sum)

Printing total\_goals would display:



1. 9

2. 3

3. 5

4. 12

5. 8

6. 6

7. 5

8. 6

Now, let's use lapply to find the total goals scored by Brazil in the 2014 world cup!

instructions

* We've provided a list containing the goals scored for the brazilian soccer/football team in the 2014 world cup. Use lapply to find the total number of goals scored in each game:

brazil <- list(c(3,1),c(0,0),c(1,4),c(1,1),c(2,1),c(1,7),c(0,3))

* Store the results in total\_goals.

Now that we've used lapply with a built-in function, let's take it a step further by writing our own function. We have a list of match scores for the 2014 Brazilian soccer team. Assuming that the first number in each vector is the number of goals Brazil scored, let's write a function that tells us whether Brazil won, lost or tied the game.

In the previous screen, we used lapply to find the total number of goals scored in each game brazil played in. To find the total number of goals scored, we used the custom, built-in function sum(). In the previous mission, we learned how to write our own functions as well. Now, let's take it a step further. Using the list of scores for each game, what if we wanted to know if Brazil won, tied or lost the game? Using what you've learned from the previous mission, let's write a function that tells us whether Brazil won, tied or lost the match using lapply.

As a review, here's the syntax for writing your own function:



function\_name <- function(input){

expression

return(result of expression)

}

Let's write our function!

instructions

* Write a function called get\_result that tells us whether Brazil won, tied or lost the game.
  + Use "Win", "Tie","Loss" as the resulting values.
* Do not use any loops. Use lapply to apply the function on the list.
* Store the results in match\_results.

Now that we've learned how to use lapply, there might be scenarios where we want our results in vector form rather than a list. When we want to apply a function to a list or vector and return a *vector*, we'll use sapply(). sapply() is a wrapper of lapply. Wrappers are functions that use another function when executing its code. In this case, sapply() uses lapply(). The only difference is that sapply returns a vector instead of a list.

The syntax for sapply is the same as lapply:



sapply(x, function)

**x**: The list or vector we want to apply our function to.

**function**: The function we're using.

Let's use the same matches list and use sapply to find the total number of goals scored.



matches <- list(c(5,4),c(2,1),c(4,1),c(7,5),c(3,5),c(3,3),c(2,3),c(4,2))

​

total\_goals <- sapply(matches,sum)

The interepeter will return a vector containing the results:



[1] 9 3 5 12 8 6 5 6

Compared to the returned output using lapply:



1. 9

2. 3

3. 5

4. 12

5. 8

6. 6

7. 5

8. 6

The values are the same, but they're structured differently. Now, let's use sapply on the scores for Englands matches to find the total number of goals!

instructions

* Here are the match results for England in the 2014 world cup:

england\_scores <- list(c(1,2),c(1,2),c(0,0))

* Use sapply to return the total number of goals scored in England's matches. Store this in england\_totals\_s.
* Use lapply to return the total number of goals scored in England's matches. Store this in england\_totals\_l.
* Print both outputs and note the difference.

In the previous screen, we used a built-in function sum() with sapply. Now, let's write our own function and use it within sapply.

instructions

* Write a function that returns the *difference* in goals scored between the winning & losing team for England's matches. Use the abs() within your function to ensure each value is positive:

england\_scores <- list(c(1,2),c(1,2),c(0,0))

* Store the resulting vector in goal\_diff.

Now that we've learned how to use sapply, in relation to the first screen, where we want to focus on writing clean & efficient code, let's introduce another apply function that can improve the quality of our code: vapply.

vapply performs the same function as an sapply, however, with vapply, you'll explicitly state the data type that is returned. The reason why vapply may be better than sapply is that having control of the returned data type can prevent future errors. In many instances, you may have a function that can return multiple data types. However, only one data type would be valid for our code. Using vapply would control for this.

Here's the syntax of the vapply function:



vapply(x, function, function\_value)

**function\_value**: The specified format you want your results to be in. For example, if we want to return a single numeric value, we'd make function\_value as.numeric(1).

For example, let's compare & contrast sapply and vapply. Here's our original sapply function:



matches <- list(c(5,4),c(2,1),c(4,1),c(7,5),c(3,5),c(3,3),c(2,3),c(4,2))

​

total\_goals <- sapply(matches,sum)

Now using vapply, let's return a single numeric value:



matches <- list(c(5,4),c(2,1),c(4,1),c(7,5),c(3,5),c(3,3),c(2,3),c(4,2))

​

total\_goals <- vapply(matches,sum, as.numeric(1))

Let's use vapply!

instructions

* For England's scores, use the vapply function to get the total number of goals. Output it as a single numeric value.
* Store this in total.

In this mission, we've used two different types of apply statements on a vector of values. Now, let's expand our data structure to a dataframe. Using our dataframe of 2014 world cup soccer scores, what if we wanted to find the average number of home goals & average number of away goals for each country?

Whenever we want to apply a function, in this case, mean(), to *subsets* of a data structure, we'll use the tapply statement.

To use tapply, we'll need to understand a few key components of tapply:



tapply(x, index, function ...)

**x**: This is the object we want to apply. In our case, this object will be a dataframe.

**index**: This is a list of *factors*.

**function**: This is the function we're applying over the subset.

To better understand what's going on, let's look at an example. Looking at our df\_scores dataframe, let's find the *total* number of goals scored for each home team:

tapply(df\_scores$home\_goals, list(df\_scores$home\_country),sum)

The first five results of the display would be:



Algeria 1

Argentina 5

Australia 2

Belgium 5

Let's dig into what's happening. We've chosen the subset df\_scores$home\_goals as the subset of the dataframe we want to apply a function to. We're applying the sum function to this subset. However, simply running the following will return an error:

tapply(df\_scores$home\_goals,sum)

The error message displays:



Error in unique.default(x, nmax = nmax): unique() applies only to vectors

We'll need to add an *index*. Think of the index as an organization tool. If we're calculating the total goals, the index will organize the total goals *by factor*. In our case, the factor is the home\_country. As a result, the expression will calculate *the total number of goals scored for each home team* as you'll see by the results.

Now, let's find the *average* number of goals scored by the home team and the *average* number of goals scored by the away team.

instructions

* Using the df\_scores dataset, find the average number of home goals by country. Store this in home\_average.
* Using the df\_scores dataset, find the average number of away goals by country. Store this in away\_average.

In this mission, we learned a more effective way of iterating on multiple values with the apply family. Different functions within the apply family are better used for different contexts. It'll depend on the problem you're looking to solve.

In the next mission, we'll round out your R toolkit by showing you how to manipulate strings & dates. We'll show you how to concatenate, split, replace characters within a string. We'll also show you how to transform strings into dates so you can extract valuable information out dates.