


# Control Structures in R: Using If-Else Statements and Loops

 [dataquest.io/blog/control-structures-in-r-using-loops-and-if-else-statements](https://dataquest.io/blog/control-structures-in-r-using-loops-and-if-else-statements)

February 1, 2018

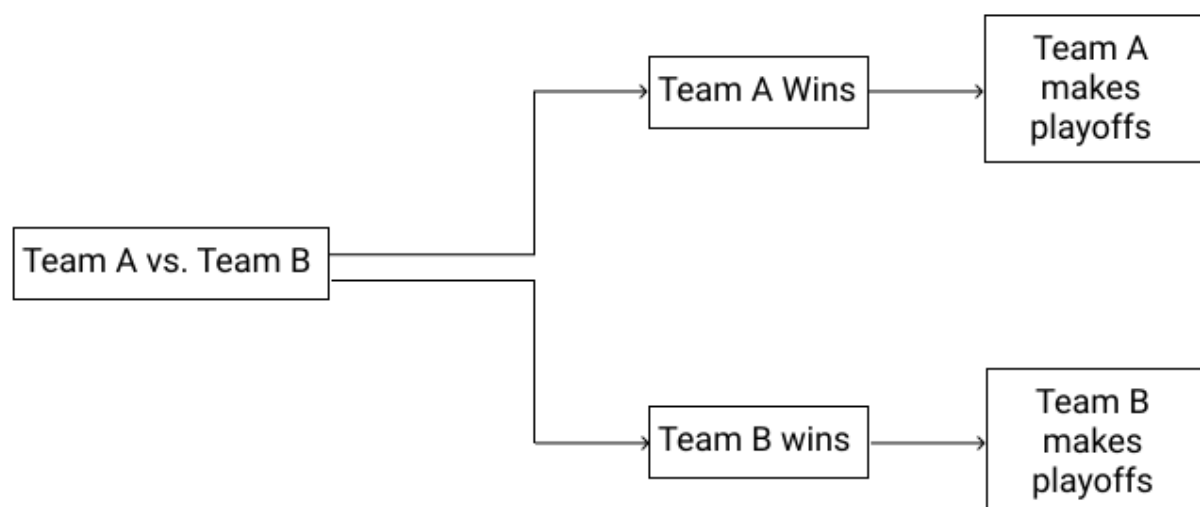
Control structures allow you to specify the execution of your code. They are extremely useful if you want to run a piece of code multiple times, or if you want to run a piece a code if a certain condition is met.

*This tutorial is based on part of our newly released [Intermediate R course](#). The course is a continuation of the [R Fundamentals](#) course and includes a certificate of completion.*

In this tutorial, we teach you how to use control structures by building a simple algorithm that tells you who won or lost a soccer match. We assume some familiarity with basic data structures, arithmetic operations, and comparison operators.

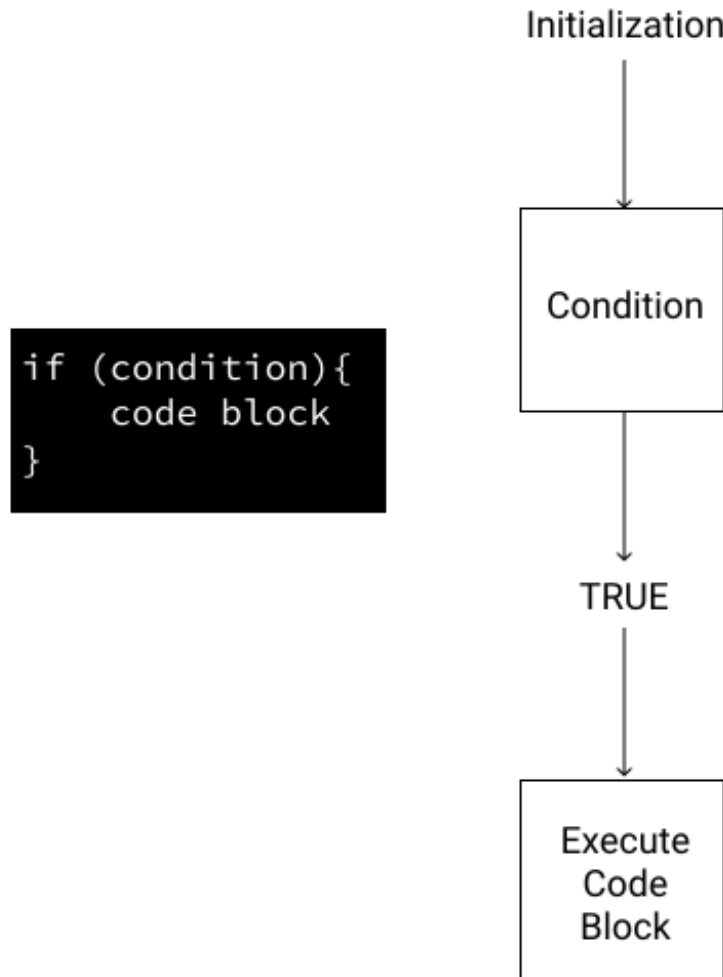
## Understanding if statements

Let's say we're watching a match that decides which team makes the playoffs. Note: we'll be using fictitious data in this post. We can visualize this scenario through a tree chart:



As we can see in the tree chart, there are only two possible outcomes. We can use an if statement to write a program that prints out the winning team. An if statement tells the interpreter to run a line of code if a condition returns TRUE. An if statement is a good choice here because it allows us to control the code execution depending on the conditional. The figure below shows the conditional flow chart of an if-statement.:

## If-Statement



`condition` should be an expression that evaluates to `TRUE` or `FALSE`. If the expression returns `TRUE`, then the program will execute all code between the brackets `{ }`. If `FALSE`, then no code will be executed. Knowing this, let's look at an example of an if statement that prints the name of the team that won.

```
team_A <- 3  
team_B <- 1  
  
if (team_A > team_B){  
  print ("Team A wins")  
}  
  
[1] "Team A wins"
```

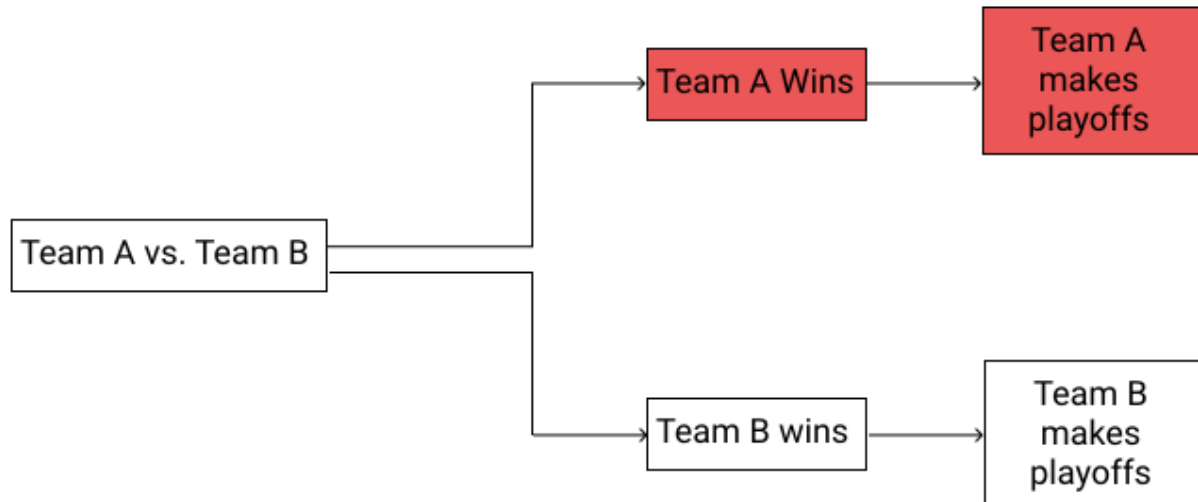
## Adding the else statement

In the previous exercise, we printed the name of the team that will make the playoffs based on our expression. Let's look at the new matchup of scores. What if, `team_A` had `1` goal and `team_B` had `3` goals. Our `team_A > team_B` conditional would evaluate to `FALSE`. As a result, if we ran our expression the R interpreter would not return anything:

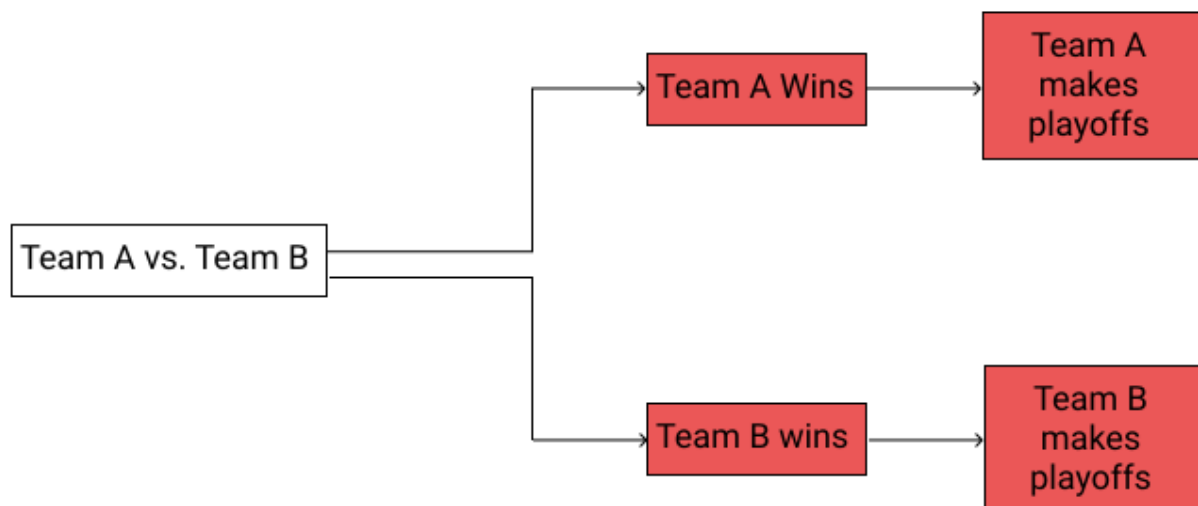
```
team_A <- 1
team_B <- 3

if (team_A > team_B){
  print ("Team A will make the playoffs")
}
```

We've only coded out one conditional branch of our flow chart:

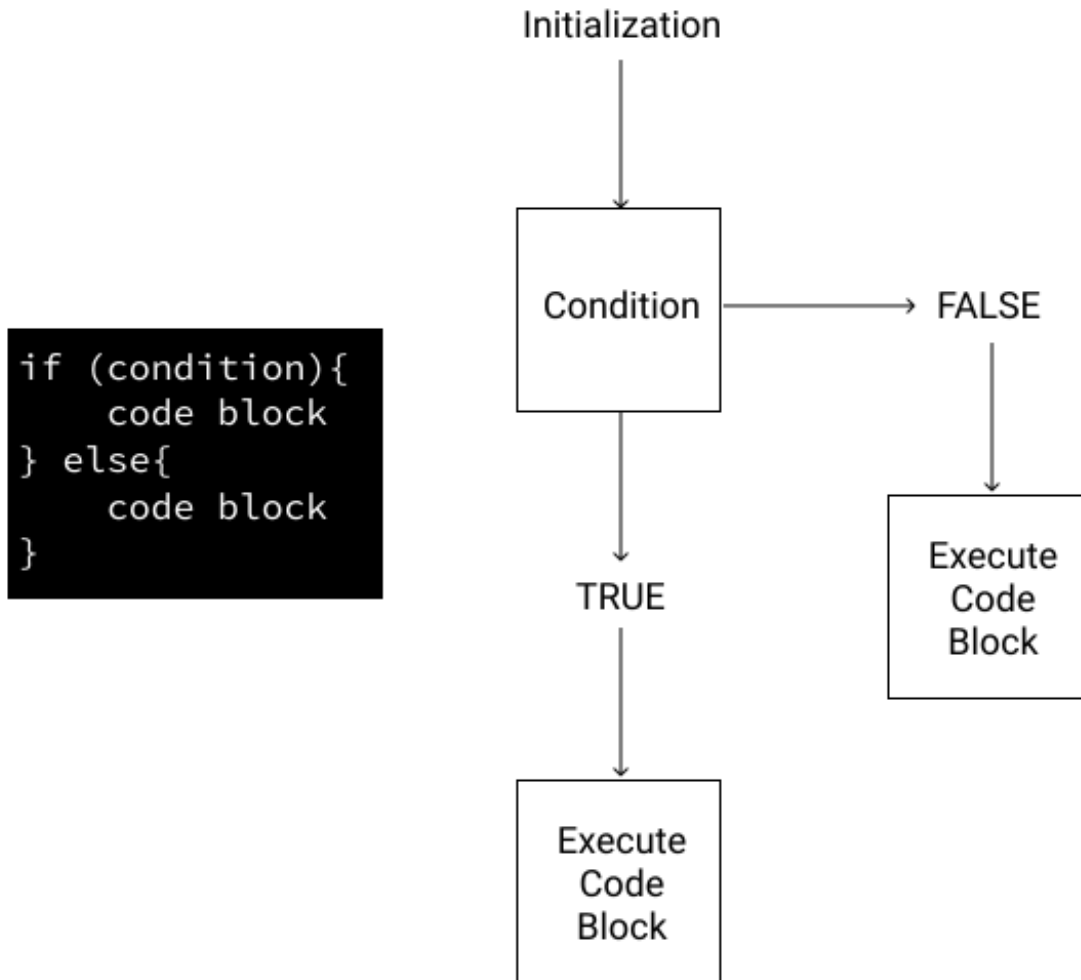


However, we'd like our program to print "Team B will make the playoffs" if the expression evaluates to `FALSE`. We'd like our program to be able to both conditional branches:



To do this, we'll add an `else` statement. An `else` statement tells the interpreter to run specific lines of code if our comparison operator evaluates to `FALSE`:

## If-Else Statement



If our conditional expression `team_A > team_B` condition returns `FALSE`, we'll need to add another an output for this branch. If our comparison operator evaluates to `FALSE`, let's print `"Team B will make the playoffs"`.

```
team_A <- 1
team_B <- 3

if (team_A > team_B){
  print ("Team A will make the playoffs")
} else {
  print ("Team B will make the playoffs")
}

[1] "Team B will make the playoffs"
```

## Using the for loop to run our code

Now that we've used an if-else statement to display the results of one match, what if we wanted to find the results of *multiple* matches? Let's say we have a list of vectors containing the results of our match: `matches <- list(c(2,1),c(5,2),c(6,3))`. Assuming that `team_A`'s goals fall in the first index of the vector and `team_A`'s opponent falls on the second, finding the results using if-else statements would look like this:

```

if (matches[[1]][1] > matches[[1]][2]){
    print ("Win")
} else { print ("Loss") }

if (matches[[2]][1] > matches[[2]][2]){
    print ("Win")
} else { print ("Loss") }

if (matches[[3]][1] > matches[[3]][2]){
    print ("Win")
} else { print ("Loss") }

```

And this would print:

```

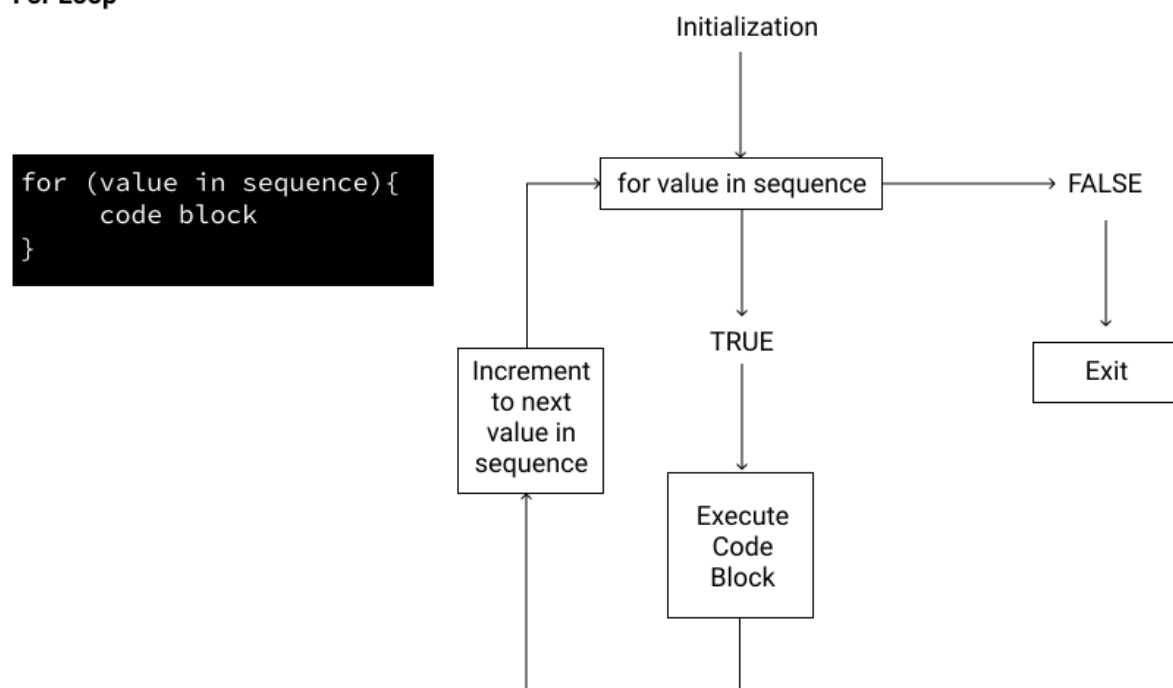
[1] "Win"
[1] "Win"
[1] "Win"

```

Keep in mind—we're using `[[ ]]` when indexing, since we want to return the single value, not the value with the list object. Indexing with `[ ]` will return a *list object*, not the value. Writing the results using `if-else` statements can work, but if our `matches` list contains 100 matches, it would be extremely cumbersome to write out each statement.

Instead, we can perform the same action using a **for loop**. A for loop repeats a chunk of code, multiple times for each *element* within an object:

### For Loop



In this diagram, for each value in the sequence, the loop will execute the code block. However, if there are no more values left in the sequence, this will return `FALSE` and exit the loop.

Let's break down what's going on here.

- **sequence:** This is a set of objects. For example, this could be a vector of numbers `c(1,2,3,4,5)`.
- **value:** This is an iterator variable you use to refer to each value in the `sequence`. See variables naming conventions in the first course for valid variable names.
- **code block:** This is the expression that's evaluated.

Let's look at a concrete example. If we were to write a loop for the following code:

```
teams <- c("team_A", "team_B")
```

```
for (value in teams){
  print(value)
}
```

```
[1] "team_A"
```

```
[1] "team_B"
```

Since `team` has two values, the loop will run twice. Let's look at the loop:

```
teams <- c("team A", "team B")

for (value in teams) {
  print(value)
}
```

```
print("team A")
print("team B")
```

```
[1] "team A"
[1] "team B"
```

Once the loop displays the result from the first iteration, the loop will look at the next value in the position. As a result, it'll go through another iteration. Since there aren't any more values in the sequence, the loop will exit after `"deep sea sailors"`. In aggregate, the final result will look like this:

```
[1] "team A"
```

```
[1] "team B"
```

## Adding the results of a loop to an object

---

Now that we've written out our loop, we'll want to store each result of each iteration in our loop. To store these values, we could store it in a data structure. In this post, we'll store our values in a vector, since we're dealing one data type.

In our free [R Fundamentals course](#), we learned how to combine vectors using the `c()` function. We'll use the same method to store the results of our for loop. Let's take the following for loop:

```
for (match in matches){  
  print(match)  
}
```

And let's say we wanted to get the total goals scored in a game and store them in the vector:

```
matches <- list(c(2,1),c(5,2),c(6,3))  
  
for (match in matches){  
  sum(match)  
}
```

Now, if we want to save the total goals for each match, we'll can initialize a new vector and then append each additional calculation onto that vector.

```
matches <- list(c(2,1),c(5,2),c(6,3))  
  
total_goals <- c()  
for (match in matches){  
  total_goals <- c(total_goals, sum(match))  
}
```

## Using if-else statements within for loops

---

Now that we've learned if-else statements and for loops, we can use if-else statements within our for loops to give us the results of *multiple* matches. To combine two control structures, we'll place one control structure in between the brackets `{ }`. Let's take the following match results of `team_A`:

```
matches <- list(c(2,1),c(5,2),c(6,3))
```

And then, let's loop through it:

```
for (match in matches){  
}
```

This time, rather than print our results, let's add an if-else statement into the expression. In our scenario, we want our program to print whether `team_A` won or lost the game. Assuming `team_A`'s goals is the first index of each pair of values and the opponents is the second index, we'll need to use a comparison operator to compare the values. After we

make the comparison, if `team_A` 's score is higher, we'll print `"Win"` . If not, we'll print `"Lose"` . When indexing into the iterable variable `match` , you can use either `[]` or `[[ ]]` since the iterable is a vector, not a list.

```
matches <- list(c(2,1),c(5,2),c(6,3))
```

```
for (match in matches){  
  if (match[1] > match[2]){  
    print("Win")  
  } else {  
    print ("Lose")  
  }  
}
```

```
[1] "Win"  
[1] "Win"  
[1] "Win"
```

## Breaking the for loop

---

Now that we've added an if-else statement, let's look at how to stop a for loop based on a certain condition. In our case, we can use a `break` statement to stop the loop as soon as `team_A` won a game. In our current for loop, we insert the `break` statement inside our if-else statement.

```
matches <- list(c(2,1),c(5,2),c(6,3))
```

```
for (match in matches){  
  if (match[1] > match[2]){  
    print("Win")  
    break  
  } else {  
    print("Lose")  
  }  
}
```

```
[1] "Win"
```

## Using a while loop

---

In the previous exercise, we used a for loop to repeat a chunk of code that gave us the result of the match. Now that we've returned the results of each match, what if we wanted to count the number of wins to determine if they make the playoffs? One method of returning the results for the first four games, is to use a **while** loop.

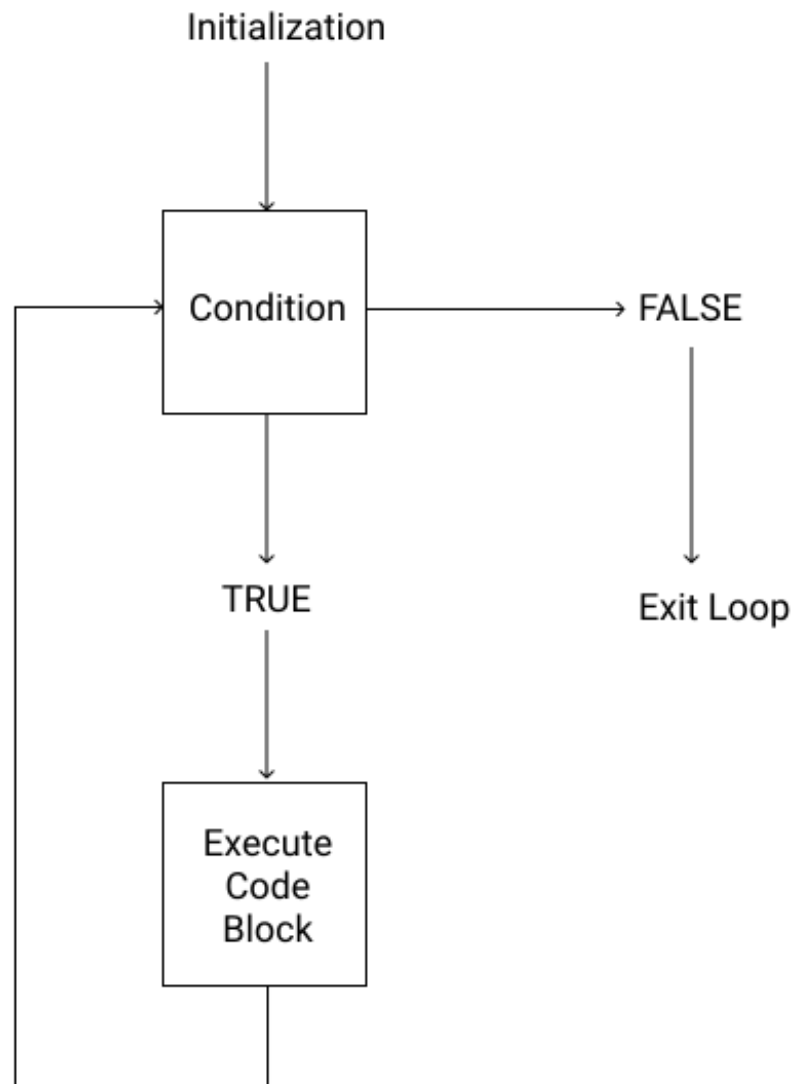
A while loop is a close cousin of the for loop. However, a while loop will check a logical condition, and keep running the loop as long as the condition is true. Here's what the syntax of a while loop looks like:

```
while(condition){  
  expression  
}
```

In flow-chart form:



## While Loop



If the condition in the while loop is *a/ways* true, the while loop will be an infinite loop. When writing a while loop, we want to ensure, that at some point, the condition will be false, so the loop can stop running. Let's take a team that's starting the season with zero wins. They'll need to win 10 matches to make the playoffs. We can write a while loop to tell us whether the team makes the playoffs:

```
wins <- 0
```

```
while (wins < 10){  
  print ("Does not make playoffs")  
  wins <- wins + 1  
}
```

```
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
[1] "Does not make playoffs"
```

Our loop will stop running when wins hits 10. Notice, that we continuously add 1 to the win total, so eventually, the `win < 10` condition will return `FALSE` . As a result, the loop exits. Let's write our first while loop, counting `red dragon` wins!

## Using an if-else statement within a while loop

---

Now that we've printed the status of the team when they don't have enough wins, we'll add a feature that indicates when they do make the playoffs. To do this, we'll need to add an if-else statement into our while loop. Adding an if-else statement into our while loop is the same as adding it to our for loop. In our previous example, where 15 wins allowed us to make the playoffs, let's add an if-else conditional. The if-else conditional will go between the brackets of the while loop.

```
wins <- 0
while (wins <= 15){
  if (wins < 15){
    print("does not make playoffs")
  } else {
    print ("makes playoffs")
  }
  wins <- wins + 1
}
```

```
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "makes playoffs"
```

## Breaking the while loop

---

Let's say the maximum number of wins a team can have in a season is 15. To make the

playoffs, we'll still need 10 wins. We want to write a program that tells us if we made the playoffs or not. To do this, we can use a while loop and then insert a break statement when wins hits 10.

```
wins <- 0
playoffs <- c()

while (wins <= 15){
  if (wins < 10){
    print("does not make playoffs")
    playoffs <- c(playoffs, "does not make playoffs")
  } else {
    print ("makes playoffs")
    playoffs <- c(playoffs, "makes playoffs")
    break
  }
  wins <- wins + 1
}

[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "does not make playoffs"
[1] "makes playoffs"
```

## Next Steps

---

If you'd like to learn more, this tutorial is based on our [R Intermediate](#) course, which is part of our Data Analyst in R track. Building upon the concepts in this tutorial, you'll learn:

- How to build your own soccer match prediction function:
  - Writing your own functions
  - Using built-in functions
  - Writing functions with control structures
- An simple method of iterating over multiple values
  - Applying functions over lists
  - Applying functions over vectors
  - Applying functions over dataframes
- How to manipulate strings and dates by writing your own headline generator
  - Concatenate, extract characters in a string
  - Extracting years, months, days from a date object
  - Using string/date manipulation to write functions
- How to use and install RStudio
  - The installation process of RStudio
  - Creating markdown and R scripts in RStudio
  - Using the R console in RStudio

[Start the R Intermediate course](#)