STRUCTURE PROGRAMMING IN R

In this R training you will learn about conditional statements, loops and functions to power your own R scripts. Next, you can make your R code more efficient and readable using the apply functions. Finally, the utilities chapter gets you up to speed with regular expressions in the R programming language, data structure manipulations and times and dates. This R tutorial will allow you to learn R and take the next step in advancing your overall knowledge and capabilities while programming in R.

1 Conditionals and Control Flow

To be TRUE or not be TRUE, that's the question. In this chapter you'll learn about relational operators to see how R objects compare and logical operators to combine logicals. Next, you'll use this knowledge to build conditional statements.

Relational Operators

Relational operators, or comparators, are operators which help us see how one R object relates to another. For example, you can check whether two objects are equal. You can do this by using a double equals sign. We can for example see if the logical value TRUE equals the logical value TRUE. Let's try it out in the console: we type TRUE equals TRUE. The result of this query is a logical, in this case TRUE, because TRUE equals TRUE. On the contrary, TRUE == FALSE will give us FALSE. Make sense, right?

```
Equality ==
> TRUE == TRUE
[1] TRUE

> TRUE == FALSE
[1] FALSE

> "hello" == "goodbye"
[1] FALSE

> 3 == 2
[1] FALSE
```

Apart from logical variables, we can also check the equality of other types. We can also compare strings and numbers. The opposite of the equality operator is the inequality. Operator, written as an exclamation mark followed by an equals sign. This sentence would read as: "hello" is not equal to "goodbye". Because this statement is correct, R will output TRUE. Naturally, the inequality operator can also be used for numerics, logicals, and of course other R objects as well. See how every time, the result of the equality operators is opposite for the inequality operator.

```
Inequality !=
> TRUE != TRUE
[1] FALSE

> TRUE != FALSE
[1] TRUE

> "hello" != "goodbye"
[1] TRUE

> 3 != 2
[1] TRUE
```

Of course, there are also cases where you need more than simply equality and inequality operators. What about checking if an R object is "less than" or "greater than" another R object? This will not come as a surprise: you can use the less-than and greater-than sign for this. In the case of numerical values, here is a straightforward example: 3 less than 5 will evaluate to TRUE, while 3 greater than 5 will evaluate to FALSE.

For numerics this makes sense, but how would this work for character strings and logical values? Is "Hello" greater than "Goodbye"? Let's find out! Apparently "Hello" greater than "Goodbye" evaluates to TRUE, but why so? It's because R uses the alphabet to sort character strings. Since "H" comes after "G" in the alphabet, "Hello" is considered greater than "Goodbye". How about

```
< and >
> 3 < 5
[1] TRUE

> 3 > 5
[1] FALSE

> "Hello" > "Goodbye" Alphabetical Order
[1] TRUE

> TRUE < FALSE
[1] FALSE
```

logical values? Is TRUE less than FALSE? The following query gives us the answer. It appears not; it evaluates to FALSE. That's because under the hood, TRUE corresponds to 1 and FALSE corresponds to 0. And of course, 1 is not less than 0, hence the FALSE result.

```
<= and >=
> 5 >= 3
[1] TRUE

> 3 >= 3
[1] TRUE
```

You can also check to see if one R object is greater than or equal to (or less than or equal to) another R object. To do this, you can use the less than sign, or the greater than sign, together with the equals sign. So 5 greater than or equal to 3 as well as 3 greater than or equal to 3 will evaluate to TRUE.

You already knew that R is pretty good with vectors. How about R's comparators, can they also handle vectors? Suppose you have recorded the daily number of views your LinkedIn profile had the previous week and stored them in a vector, linkedin. If we want to find out on which days the number of views exceeded 10, we can directly use the greater than sign. For the first, third, sixth and seventh element in the vector, the number of views is greater than 10, so for these elements the result will be TRUE. You can also compare vectors to vectors; suppose you also recorded the number of views your Facebook profile had the previous week and saved them in another vector, facebook. When are the number of Facebook views less than or equal to the number of LinkedIn views? The following expression shows us how to calculate this. Does it make sense?

```
Relational Operators & Vectors
> linkedin <- c(16, 9, 13, 5, 2, 17, 14)

> linkedin
[1] 16 9 13 5 2 17 14

> linkedin > 10
[1] TRUE FALSE TRUE FALSE FALSE TRUE TRUE

> facebook <- c(17, 7, 5, 16, 8, 13, 14)

> facebook
[1] 17 7 5 16 8 13 14

> facebook <= linkedin
[1] FALSE TRUE TRUE FALSE FALSE TRUE TRUE
```

In this case, the comparison is done for every element of the vector, one by one. For example, in the third day, the number of Facebook views is 5 and the number of LinkedIn views is 13. The comparison evaluates to TRUE, as 5 is smaller than or equal to 13. Just as for vectors, R also knows how to compare other data structures, such as matrices and lists. Head over the interactive exercises and add Relational Operators to your ever growing R skillset!

Logical Operators

You know how to use relational operators in R, awesome! But what if you want to change or combine the results of comparisons? R does this using the AND, the OR, and the NOT operator. Let's have a closer look at each one of them and start with the AND operator.

```
Logical Operators
AND operator &
OR operator |
NOT operator !
```

The AND operator works just as you would expect. It typically takes two logical values and returns TRUE only if both the logical values are TRUE themselves. The means that TRUE and TRUE evaluates TRUE, but that FALSE and TRUE, TRUE and FALSE and FALSE and FALSE all evaluate to FALSE. Instead of using logical values, we can of course use the results of comparisons. Suppose we have a variable x, equal to 12. To check if this variable is greater than 5 but less than 15, we can use x greater than 5 but less than 15, we can use x greater than 5 and x less than 15. As you already learned, the first

```
AND operator &
> TRUE & TRUE   Only TRUE if both are TRUE
[1] TRUE

> FALSE & TRUE
[1] FALSE

> TRUE & FALSE  FALSE otherwise
[1] FALSE

> FALSE & FALSE
[1] FALSE
```

part will evaluate to TRUE. The second part, will also evaluate to TRUE. So, the result of this expression is TRUE. This makes sense, because 12 lies between 5 and 15. However, if x were equal to 17, the expression x greater than 5 & x less than 15 would simplify to TRUE and FALSE. Which results in this expression being FALSE.

```
AND operator &
> x <- 12

   TRUE    TRUE
> x > 5 & x < 15
[1] TRUE

> x <- 17

   TRUE    FALSE
> x > 5 & x < 15
[1] FALSE
```

The OR operator (|) works similarly, but the difference is that only at least one of the logical values it uses should be equal to TRUE for the OR operation to evaluate to TRUE. This means that, TRUE or TRUE equals TRUE, but that also TRUE or FALSE and FALSE or TRUE evaluate to TRUE. When both logicals are FALSE in an OR operation, so in the case of FALSE or FALSE, the result is FALSE. Remember that the OR operation is not an exclusive or operation, TRUE or TRUE equals TRUE as well. Just as for AND operators, we can use comparisons together with the OR operator.

```
OR operator |
> TRUE | TRUE
[1] TRUE

> TRUE | FALSE TRUE if at least one is TRUE
[1] FALSE

> FALSE | TRUE
[1] FALSE

> FALSE | FALSE Only FALSE If both FALSE
[1] FALSE
```

```
OR operator |
> y <- 4

   TRUE    FALSE
> y < 5 | y > 15
[1] TRUE

> y <- 14

  FALSE   FALSE
> y < 5 | y > 15
[1] FALSE
```

Suppose we have a variable y, equal to 4 this time. To see if this variable is less than 5 or greater than 15, we can use this expression. R will first carry out the comparisons, resulting in TRUE or FALSE, which in turn results in TRUE. Now, let's have y equal 14. The expression y less than 5 or y greater than 15 now evaluates to FALSE or FALSE. Neither one of the comparisons are TRUE, so the result is FALSE.

There's one last operator I want to talk about here, the NOT operator. The NOT operator, represented by an exclamation mark, simply negates the logical value it's used on. So exclamation mark TRUE evaluates to FALSE, while exclamation mark FALSE evaluates to TRUE. Just as the OR and AND operators, you can use the NOT operator in combination with logical operators. This is not always necessary, however, because this line of code is exactly the same as this one. However, there are cases in R where the NOT operator is really handy. For example, the built-in R function, is.numeric() checks if an R object is a numeric.

```
NOT operator !
> !TRUE
[1] FALSE

> !FALSE
[1] TRUE

> !(x < 5)
> x >= 5
```

As an illustration, take is.numeric(5), which evaluates to TRUE, as 5 is a numeric. If we negate this result using the NOT operator, we get false. If however, we type is.numeric("hello") we get FALSE. Negating this results in TRUE.

```
NOT operator !
> is.numeric(5)
[1] TRUE

> !is.numeric(5)
[1] FALSE

> is.numeric("hello")
[1] FALSE

> !is.numeric("hello")
[1] TRUE
```

Now, how do logical operators work with vectors and matrices? Well, just as relational operators, they perform the operations element-wise. The and operation on these two vectors, results in a vector with the elements TRUE, FALSE and FALSE. The first elements in both vectors are TRUE, so the first element of the resulting vector contains TRUE. Similarly, for the second

elements where TRUE and FALSE result in FALSE, and the third elements, where FALSE and FALSE give FALSE.

A similar thing happens with the OR operator: TRUE or TRUE gives TRUE, TRUE or FALSE also gives TRUE, and FALSE or FALSE gives FALSE.

```
Logical Operators & Vectors
> c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
[1] TRUE FALSE FALSE

> c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
[1] TRUE TRUE FALSE

> !c(TRUE, TRUE, FALSE)
[1] FALSE FALSE TRUE
```

The NOT operator also works on every element of the vector: TRUEs are converted to FALSEs, and FALSEs are converted to TRUEs.

```
& vs &&, | vs ||
> c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
[1] TRUE FALSE FALSE

> c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)
[1] TRUE

> c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
[1] TRUE TRUE FALSE

> c(TRUE, TRUE, FALSE) || c(TRUE, FALSE, FALSE)
[1] TRUE
```

Now, there's one last thing I want to warn you about. It's about the difference between a single and a double ampersand or vertical bar. In R you can use both the single sign version or the double sign version, but the result of the logical operation you're carrying out can be different. The biggest difference occurs when you use the two types of operations on vectors. As we've seen before, this expression, evaluates to a vector containing TRUE, FALSE and FALSE. However, if we use a double ampersand, we simply get TRUE. That's because the double ampersand operation only examines the first element of each vector. In this case the first elements are TRUE and TRUE, so the expression returns TRUE.

You can see similar things happening with the OR operator. The single sign version returns an entire vector. The double sign version returns only the result of the OR operator on the first element of each vector.

Another difference between a single and a double ampersand or a vertical bar that is less obvious has something to do with control structures, but that's more advanced material. For now, just remember that you have to pay attention when doing logical operations on vectors. You will very likely want to use the single sign versions.

Conditional Statements

```
if statement
if(condition) {
  expr
}
```

In this chapter, you already learned about relational operators, which tell us how R objects relate, and logical operators, which allow us to combine logical values. Now R also provides a way to use the results of these operators to change the behavior of your own R scripts. Sure enough, I'm talking about the if and else statements here.

Have a look at the recipe for the is statement: The if statement takes a condition; if the condition evaluates to TRUE, the R code associated with the if statement is executed. The condition to check appears inside parentheses, while the R code that has to be executed if the condition is TRUE, follows in curly brackets. Let's have a look at an example.

Suppose we have a variable x equal to -3. If this x is smaller than zero, we want R to print out "x is a negative number!" How can we do this using if statement? We first assign the variable,

x and then write the if test. If we run this bit of code, we indeed see that the string "x is a negative number" gets printed out. However, if we change x to 5, and re-run the code, the condition will be FALSE, the code is not executed, and the printout will not occur.

```
if statement
> x <- -3
> if(x < 0) {
    print("x is a negative number")
  }
[1] "x is a negative number"
```

```
if statement
> x <- 5
    FALSE
> if(x < 0) {
    print("x is a negative number") Not executed
  }
  No printout
```

This brings us to the else statement: this conditional statement does not need an explicit condition; instead, it has to be used together with an if statement. The code associated with an else statement gets executed whenever the condition of the if test is not satisfied. We can extend our recipe by including an else statement as follows. Returning to our example, suppose we want to print out "x is positive or zero", whenever the condition is not met. We can simply add the else statement. If we run the code with x equal to -3, we still get the printout "x is a negative number", because the if condition is TRUE.

```
else statement
if(condition) {
  expr1
} else {
  expr2
}
```

```
else statement
x <- -3
if(x < 0) {
    print("x is a negative number")
  } else {
    print("x is a either a positive number or zero")
}
"x is a negative number"
```

However, if we now change x to 5, the next "x is either a positive number or zero" is printed out; the x smaller than zero condition was not satisfied, so R turned to the expression in the else statement.

```
else statement
x <- 5
if(x < 0) {
    print("x is a negative number")
  } else {
    print("x is a either a positive number or zero")
}
"x is a either a positive number or zero"
```

There are also cases in which you want to customize your programs even further. Maybe we want yet another printout if x equals exactly 0. How to do this? Well, R also provides the else if statement. Let's first extend the recipe. The else if statement comes in between the if and else statement. To see how R deals with these different conditions and corresponding code blocks, let's first extend our example. We want R to print out "x is zero" if x equal 0 ant to print out "x is a positive number" otherwise. We add the else if, together with a new print statement, and adapt the message we print on the else statement.

```
else if statement
if(condition1) {
  expr1
} else if (condition2){
  expr2
} else {
  Expr3
}
```

```
else if statement
x <- -3
if(x < 0) {
    print("x is a negative number")
  } else if(x == 0){
    print("x is zero")
  } else
    print("x is a positive number")
}
"x is a negative number"
```

How does R process this control structure? Let's first go through what happens when x equals -3. In this case, the condition for the if statement evaluates to TRUE, so "x is a negative number" gets printed out, and R ignores the rest of the statements. If x equals 0, R will first check the if condition, sees that it is FALSE, and will then head over to the else if condition. This condition, x == 0, evaluates to TRUE, so "x is zero" gets print to the console, and R ignores the else statement entirely. Finally, what happens when x equals 5? Well, the if condition evaluates to FALSE, so does the else if condition, so R executes the else statement, printing "x is a positive number".

```
else if statement
x <- 0
if(x < 0) {
    print("x is a negative number")
  } else if(x == 0){
    print("x is zero")
  } else
    print("x is a positive number")
}
"x is a zero"
```

```
else if statement
x <- 5
if(x < 0) {
    print("x is a negative number")
  } else if(x == 0){
    print("x is zero")
  } else
    print("x is a positive number")
}
"x is a positive number"
```

Remember that as soon as R stumbles upon a condition that evaluates to TRUE, R executes the corresponding code and then ignores the rest of the control structure. This becomes important if the conditions you list are not mutually exclusive. Have a look at this example, that sees if number is a number is divisible by 2 or by 3. When x equals 6, the first condition evaluates to TRUE, so R prints out "divisible by 2". Now R exists the control structure and will not look at the rest of statements. So although the second condition, for the else if part, would evaluate to TRUE, nothing gets printed out.

```
if, else if, else
x <- 6
if(x %% 2 == 0) {
    print("divisible by 2")
  } else if(x %% 3 == 0){
    print("divisible by 3")
  } else
    print("not divisible by 2 nor by 3...")
}
"divisible by 2"
```