# Lab-Conditions_Loops

May 18, 2021

LOOPS and CONDITIONS

## 0.1 Table of Contents



Estimated Time Needed: 15 min

About the Dataset

Imagine you got many movie recomendations from your friends and compiled all of the recomendations in a table, with specific info about each movie.

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

You can see part of the dataset below

Lets first download the dataset that we will use in this notebook:

[ ]:
```
# code to download the dataset
download.file("https://ibm.box.com/shared/static/
 ↪n5ay5qadfe7e1nnsv5s01oe1x62mq51j.csv", destfile="movies-db.csv")
```

To begin, we can start by associating our data to a data frame. Let's call it `movies_Data`.

[ ]:
```
movies_data <- read.csv("movies-db.csv", header=TRUE, sep=",")
```

Control statements

**Control statements** are ways for a programmer to control what pieces of the program are to be executed at certain times. The syntax of control statements is very similar to regular english, and they are very similar to logical decisions that we make all the time.

**Conditional statements** and **Loops** are the control statements that are able to change the execution flow. The expected execution flow is that each line and command should be executed in the order they are written. Control statements are able to change this, allowing you to skip parts of the code or to repeat blocks of code.

Conditional Statements

We often want to check a conditional statement and then do something in response to that condition being true or false.

### 0.1.1 If Statements

**If** statements are composed of a conditional check and a block of code that is executed if the check results in **true**. For example, assume we want to check a movie's year, and print something if it greater than 2000:

[ ]:
```
movie_year = 2002

# If Movie_Year is greater than 2000...
if(movie_year > 2000){

    # ...we print a message saying that it is greater than 2000.
    print('Movie year is greater than 2000')

}
```

Notice that the code in the above block **{}** will only be executed if the check results in **true**.

You can also add an `else` block to `if` block – the code in `else` block will only be executed if the check results in **false**.

**Syntax:**

if (condition) {

`# do something`

} else {

`# do something else`

}

**Tip**: This syntax can be spread over multiple lines for ease of creation and legibility.

Let's create a variable called **`Movie_Year`** and attribute it the value 1997. Additionally, let's add an `if` statement to check if the value stored in **`Movie_Year`** is greater than 2000 or not – if it is, then we want to output a message saying that Movie_Year is greater than 2000, if not, then we output a message saying that it is not greater than 2000.

[ ]:
```
movie_year = 1997

# If Movie_Year is greater than 2000...
if(movie_year > 2000){

    # ...we print a message saying that it is greater than 2000.
    print('Movie year is greater than 2000')

}else{ # If the above conditions were not met (Movie_Year is not greater than␣
 ↪2000)...

    # ...then we print a message saying that it is not greater than 2000.
    print('Movie year is not greater than 2000')

}
```

Feel free to change **movie_year**'s value to other values – you'll see that the result changes based on it!

To create our conditional statements to be used with **if** and **else**, we have a few tools:

### 0.1.2 Comparison operators

When comparing two values you can use this operators

equal: `==`

not equal: `!=`

greater/less than: `> <`

greater/less than or equal: `>= <=`

### 0.1.3 Logical operators

Sometimes you want to check more than one condition at once. For example you might want to check if one condition **and** other condition are true. Logical operators allow you to combine or modify conditions.

and: `&`

or: `|`

not: `!`

Let's try using these operators:

[ ]:
```
movie_year = 1997

# If Movie_Year is BOTH less than 2000 AND greater than 1990 -- both conditions␣
 ↪have to be true! -- ...
if(movie_year < 2000 & movie_year > 1990 ) {
```

```
    # ...then we print this message.
    print('Movie year between 1990 and 2000')
}

# If Movie_Year is EITHER greater than 2010 OR less than 2000 -- any of the␣
 ↪conditions have to be true! -- ...
if(movie_year > 2010 | movie_year < 2000 ) {
    # ...then we print this message.
    print('Movie year is not between 2000 and 2010')
}
```

**Tip**: All the expressions will return the value in Boolean format – this format can only house two values: true or false!

### 0.1.4 Subset

Sometimes, we don't want an entire dataset – maybe in a dataset of people we want only people with age 18 and over, or in the movies dataset, maybe we want only movies that were created after a certain year. This means we want a **subset** of the dataset. In R, we can do this by utilizing the **`subset`** function.

Suppose we want a subset of the **`movies_Data`** data frame composed of movies from a given year forward (e.g. year 2000) if a selected variable is **recent**, or from that given year back if we select **old**. We can quite simply do that in R by doing this:

[ ]:
```
decade = 'recent'

# If the decade given is recent...
if(decade == 'recent' ){
    # Subset the dataset to include only movies after year 2000.
    subset(movies_data, year >= 2000)
} else { # If not...
    # Subset the dataset to include only movies before 2000.
    subset(movies_data, year < 2000)
}
```

Loops

Sometimes, you might want to repeat a given function many times. Maybe you don't even know how many times you want it to execute, but have an idea like `once for every row in my dataset`. Repeated execution like this is supplemented by **loops**. In R, there are two main loop structures, **`for`** and **`while`**.

### 0.1.5 The `for` loop

The `for` loop structure enables you to execute a code block once for every element in a given structure. For example, it would be like saying **execute this once for every row in my dataset**, or "execute this once for every element in this column bigger than 10". **`for`** loops are a very useful structure that make the processing of a large amount of data very simple.

Let's try to use a **for** loop to print all the years present in the **year** column in the **movies_Data** data frame. We can do that like this:

[ ]:
```
# Get the data for the "year" column in the data frame.
years <- movies_data['year']

# For each value in the "years" variable...
# Note that "val" here is a variable -- it assumes the value of one of the data␣
 ↪points in "years"!
for (val in years) {
    # ...print the year stored in "val".
    print(val)
}
```

### 0.1.6 The `while` loop

As you can see, the `for` loop is useful for a controlled flow of repetition. However, what if we don't know when we want to stop the loop? What if we want to keep executing a code block until a certain threshold has been reached, or maybe when a logical expression finally results in an expected fashion?

The `while` loop exists as a tool for repeated execution based on a condition. The code block will keep being executed until the given logical condition returns a `False` boolean value.

Let's try using `while` to print the first five movie names of our dataset. It can be done like this:

[ ]:
```
# Creating a start point.
iteration = 1

# We want to repeat until we reach the sixth operation -- but not execute the␣
 ↪sixth time.
# While iteration is less or equal to five...
while (iteration <= 5) {

    print(c("This is iteration number:",as.character(iteration)))

    # ...print the "name" column of the iteration-th row.
    print(movies_data[iteration,]$name)

    # And then, we increase the "iteration" value -- so that we actually reach␣
 ↪our stopping condition
    # Be careful of infinite while loops!
    iteration = iteration + 1
}
```

### 0.1.7 Applying Functions to Vectors

One of the most common uses of loops is to **apply a given function to every element in a vector of elements**. Any of the loop structures can do that, however, R conveniently provides us

with a very simple way to do that: By inferring the operation.

R is a very smart language when it comes to element-wise operations. For example, you can perform an operation on a whole list by utilizing that function directly on it. Let's try that out:

[ ]:
```
# First, we create a vector...
my_list <- c(10,12,15,19,25,33)

# ...we can try adding two to all the values in that vector.
my_list + 2

# Or maybe even exponentiating them by two.
my_list ** 2

# We can also sum two vectors element-wise!
my_list + my_list
```

R makes it very simple to operate over vectors – anything you think should work will probably work. Try to mess around with vectors and see what you find out!

This is the end of the **Loops and Conditional Execution in R** notebook. Hopefully, now you know how to manipulate the flow of your code to your needs. Thank you for reading this notebook, and good luck on your studies.

**Scaling R with big data** As you learn more about R, if you are interested in exploring platforms that can help you run analyses at scale, you might want to sign up for a free account on IBM Watson Studio, which allows you to run analyses in R with two Spark executors for free.

### 0.1.8 About the Author:

Hi! It's Helly Patel and Walter Gomes, the author of this notebook. I hope you found R easy to learn! There's lots more to learn about R but you're well on your way. Feel free to connect with me if you have any questions.