BOSTON
TRAINING ACADEMY

Data Analysis With Python

© Confidentiality & Proprietary Information

This document contains information that is Proprietary and confidential ("Confidential Information") to Boston Training Academy and shall not be used or disclosed outside.

Further, the Confidential Information should not be transmitted, duplicated, or used in whole or in part for any purpose other than what it is intended for herein. Any use or disclosure in whole or in part of this Confidential Information without the express written permission of Boston Training Academy is strictly prohibited.

This is a confidential document prepared by Boston Training Academy.

The illustrative formats and examples have been created solely to simulate Learning and do not purport to represent/reflect on work practices of any particular party/parties.

Unauthorized possession of the material or disclosure of the Proprietary information may result in legal action.

©Boston Training Academy 2021

Relational Statement and Logical Conditions

Relation and Logic are the fundamental bricks of a program that defines its functionality. With these fundamentals, you decide what should be the flow of execution and what conditions should be kept to make sure the flow stays that way.

In every programming language including python, to manage the flow of any program, conditions are required, and to define those conditions, relational and logical operators are required.

Remember those days when your mathematics teacher in school used to ask you if 3 is greater than 2, say yes, otherwise no, that is pretty much what we do in programming world too.

You provide the compiler with some condition based on an expression, compiler computes the expression and executes the condition based on the output of the expression. In the case of relational and logical expressions, the answer will always be either True or False.

Operators are the conventional symbols, that bring one, two or more operands together to form an expression. Operators and operands are two key deciding factors of the output.

Now let's see some operators that are available in python language.

Relational Operator

Relational operators are used to establish some sort of relationship between the two operands. Some of the relevant examples could be **less than**, **greater than** or **equal to** operators. Python language is capable of understanding these types of operators and accordingly return the output, which can be either True or False.

Let's checkout a few relational expressions. Open your IDLE and try this:

```
>>> 5 < 9
```

```
True
```

Since 5 is less than 9, thus the output returned is True.

The list of operators available includes:

1. **Less than** → used with <
2. **Greater than** → used with >
3. **Equal to** → used with ==
4. **Not equal to** → used with !=
5. **Less than or equal to** → used with <=
6. **Greater than or equal to** → used with >=

You can try each of the operators to practice with some numbers (or even strings).

```
>>> "abc" > "aaa"

>>> "abc" == "bcd"
```

```
True

False
```

Python Logical Operators

Logical operators, as the name suggests are used in logical expressions where the **operands** are either True or False. The operands in a logical expression, can be expressions which returns True or False upon evaluation. There are three basic types of logical operators:

1. **Logical AND**: For AND operation the result is True if and only if both operands are True. The keyword used for this operator is and.
2. **Logical OR**: For OR operation the result is True if either of the operands is True. The keyword used for this operator is or.
3. **Logical NOT**: The result is True if the operand is False. The keyword used for this operator is not.

Let's see a few examples:

```
>>> True and False
```

```
False
```

```
>>> not True
```

```
False
```

Now, we also know that the relational expressions return a Boolean value as their output, therfore know we can combine relational and logical expressions to create something more meaningful. For example,

```
>>> (2 < 3) and (2 < 5)
```

```
True
```

```
>>> (2 < 3) and (2 < 1)
```

```
False
```

Taking a bit more realistic programming example, consider you have a variable x as input and you want to check if the user entered value is between some range, say 0 to 100, then:

```
>>> x = int(input())
```

```
25
```

```
>>> (x > 0) or (x < 100)
```

```
True
```

Conditional Statements

There will be various situations while writing a program when you will have to take care of different possible conditions that might arise while execution of the program. In such situations, if conditions can be used.

The if condition

Syntax for using the if keyword is as follows:

```
if [conditional expression]:

        [statement(s) to execute]
```

if keyword and the conditional expression is ended with a **colon**. In **[conditional expression]** some conditional expression is introduced that is supposed to return a boolean value, i.e., True or False. If the resulting value is True then the **[statement to execute]** is executed, which is mentioned below the if condition with a **tabspace**(This indentation is very important).

Taking a real-life example, let's say in a savings bank account a user A, has Rs.1000. The user A, visits the ATM to withdraw money from his savings account. Now the program which handles the ATM transactions should be able to perform in every situation, like the program should be aware of the user's bank account balance, and should not allow the user to withdraw money more than the available balance in his account. Also, the program must update the account balance once the user has withdrawn money from the account, to keep the records updated. Let's write a small piece of code to stop user from withdrawing money more than the available account balance.

```
>>> savingAmt = 1000

>>> withdrawAmt = int(input("Amount to Withdraw: "));

>>> if withdrawAmt > savingAmt:

          print ("Insufficient balance");
```

```
Amount to Withdraw: 2000

Insufficient balance
```

In the above program if the user enters any amount more than Rs. 1000, a message is displayed on the screen saying "Insufficient Funds".

There are two more keywords that are optional, but can be accompanied with if statements, the are:

- else
- elif (also known as else if)

The else condition

Talking about else, let's first see how it is used alongside the if statement.

```
if[conditional expression]:

        [statement to execute]

else:

        [alternate statement to execute]
```

Continuing with the bank account-ATM example, if you noticed in the program above, we forgot to subtract the withdrawn amount from the savings account balance. Since now we have an additional else condition to use, we can print a warning with the message, "Insufficient balance" if the saving amount is less than the withdraw amount, or else we can subtract the withdrawn amount from the savings account amount to update the account balance. So the if condition written in the previous example, will get an additional else block.

```
if withdrawAmt > savingAmt:

        print ("Insufficient balance");

else:

        savingAmt = savingAmt - withdrawAmt

        print ("Account Balance:" + str(savingAmt));
```

And with the else block introduced, in case there is sufficient balance, the withdrawAmt will be subtracted from the savingAmt and the updated account balance will be displayed on screen.

With if and else, now we can diverge the flow of execution into two different directions. In case of a savings account, there will be only two cases, you have sufficient money or you don't, but what if we come across some situation where there are more than two possibilities? In such cases, we use yet another statement that is accompanied with the if and else statements, called elif (or else if in proper English).

The elif condition

elif statement is added between if and else blocks.

```
if[condition #1]:

            [statement #1]

elif[condition #2]:

            [statement #2]

elif[condition #3]:

            [statement #3]

else:

            [statement when if and elif(s) are False]
```

With this, you can add as many elif blocks as you want depending upon the possibilities that may arise.

Let's say you are given a time and you have to tell what phase of the day it is- (morning, noon, afternoon, evening or night). You will have to check the given time against multiple ranges of time within which each of the 5 phases lies. Therefore, the following conditions:

1. **Morning**: 0600 to 1159
2. **Noon**: 1200
3. **Afternoon**: 1201 to 1700
4. **Evening**: 1701 to 2000
5. **Night**: 2000 to 0559

Below we have a simple program, using the if, elif and else conditional statements:

```
if (time >= 600) and (time < 1200):

            print ("Morning");

elif (time == 1200):

            print ("Noon");

elif (time > 1200) and (time <= 1700):

            print ("Afternoon");
```

```
elif (time > 1700) and (time <= 2000):

        print ("Evening");

elif ((time > 2000) and (time < 2400)) or ((time >= 0) and (time < 600)):

        print ("Night");

else:

        print ("Invalid time!");
```

Notice the **logical operators** that have been used in each condition in the program, this example demonstrates how you will generally be using them with if-else statements.

Nesting if-else statements

Simply put, nesting if else means that you will be writing if-else statements inside another if-else statements. Syntactically,

```
if[condition #1]:

        if[condition #1.1]:

                [statement to exec if #1 and #1.1 are true]

        else:

                [statement to exec if #1 and #1.1 are false]

else:

        [alternate statement to execute]
```

Needless to say you can write the same if-else block inside an else block too. Or in fact you can add an elif condition, according to your needs. Although if you think about it, nested if-else is actually an alternative to elif. And just like we created a program which printed the phase of the day by checking the range of time, that can be done without using elif as well, by nesting if and else only.

```
if (time >= 600) and (time < 1200):

        print ("Morning");
```

```
else:

            if (time == 1200):

                        print ("Noon");

            else:

                        if (time > 1200) and (time <= 1700):

                                    print ("Afternoon");

                        else:

                                    if (time > 1700) and (time <= 2000):

                                                print ("Evening");

                                    else:

                                                if ((time > 2000) and (time < 2400)) or ((time >= 0) and
(time < 600)):

                                                            print ("Night");

                                                else:

                                                            print ("Invalid time!");
```

As you can see in the above program, all we did was, we started by adding a condition to the if block, and then used another if-else block in its else block. And we kept on doing the nesting until all the conditions were taken care of. This, however, is a little tedious than using elif. However, that's not all it is about. Nesting is very useful in various other situations, and in some cases, it's the only way forward.

Also, if you remember our first example while explaining the if statement, the savings bank account example. There is actually a quicker way to do it, a one-line way because, it's python. The following is the logic:

if (saving > withdraw) then saving = (saving - withdraw), else saving will remain as it is and there will be no transaction.

```
>>> savingAmt = savingAmt - withdrawAmt if (savingAmt > withdrawAmt) else savingAmt
```