TechOrigin

Intro to Python

Access this notebook on GITHUB or COLAB

Table of Contents

Click on the links to go directly to specific sections on the notebook.

Python - Let's get you writing some Python Code now!</h1>

Welcome! This notebook will teach you the basics of the Python programming language. Although the information presented here is quite basic, it is an important foundation that will help you read and write Python code. By the end of this notebook, you'll know the basics of Python, including how to write basic commands, understand some basic types, and how to perform simple operations on them.

Import Dependencies

In [ ]:

Assignment/Arithmentic

We can perform different arithmetic operations using pyhton notebook. Addition, Subtraction, Multiplication, etc.

In [ ]:
```
a = 2 + 2
a
```

In [ ]:
```
b = 2 - 1
b
```

In [ ]:
```
c = 5 * 6
c
```

In [ ]:
```
d = 8/7
d
```

In [ ]:
```
e = c // a
```

In [ ]:
```
f = (a (6 (b + e(90) + e)))
```

Logical/Comparison

Comparison Operators

Comparison operations compare some value or operand and, based on a condition, they produce a Boolean. When comparing two values you can use these operators:

- equal: **==**
- not equal: **!=**
- greater than: **>**
- less than: **<**
- greater than or equal to: **>=**
- less than or equal to: **<=**

Let's assign `a` a value of 5. Use the equality operator denoted with two equal **==** signs to determine if two values are equal. The case below compares the variable `a` with 6.

In [ ]:
```
# Condition Equal

a = 5
a == 6
```

The result is **False**, as 5 does not equal to 6.

Consider the following equality comparison operator `i > 5`. If the value of the left operand, in this case the variable **i**, is greater than the value of the right operand, in this case 5, then the

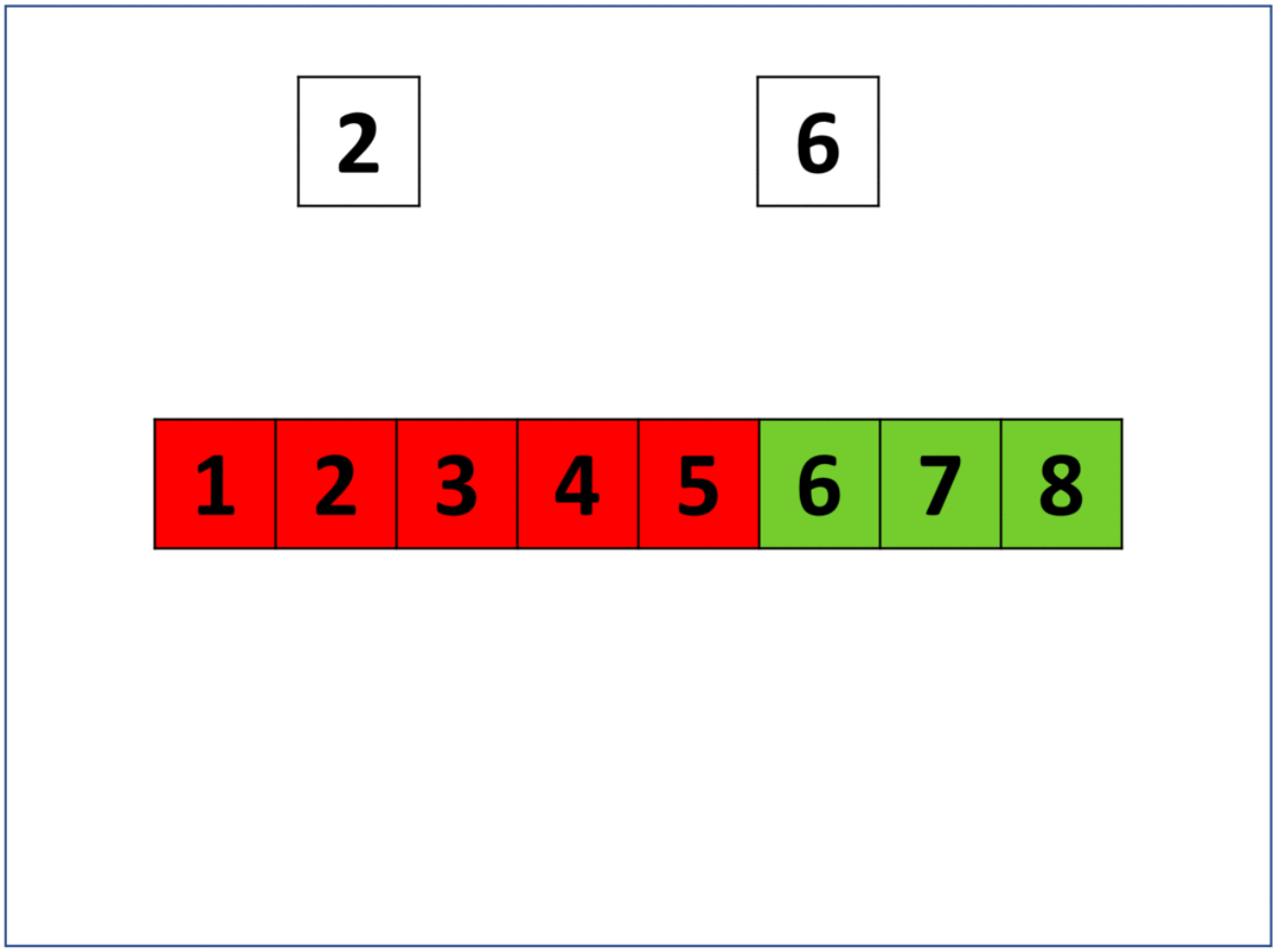
statement is **True**. Otherwise, the statement is **False**. If **i** is equal to 6, because 6 is larger than 5, the output is **True**.

In [ ]:
```
# Greater than Sign

i = 6
i > 5
```

Set `i = 2`. The statement is false as 2 is not greater than 5:

In [ ]:
```
# Greater than Sign

i = 2
i > 5
```

Let's display some values for `i` in the figure. Set the values greater than 5 in green and the rest in red. The green region represents where the condition is **True**, the red where the statement is **False**. If the value of `i` is 2, we get **False** as the 2 falls in the red region. Similarly, if the value for `i` is 6 we get a **True** as the condition falls in the green region.



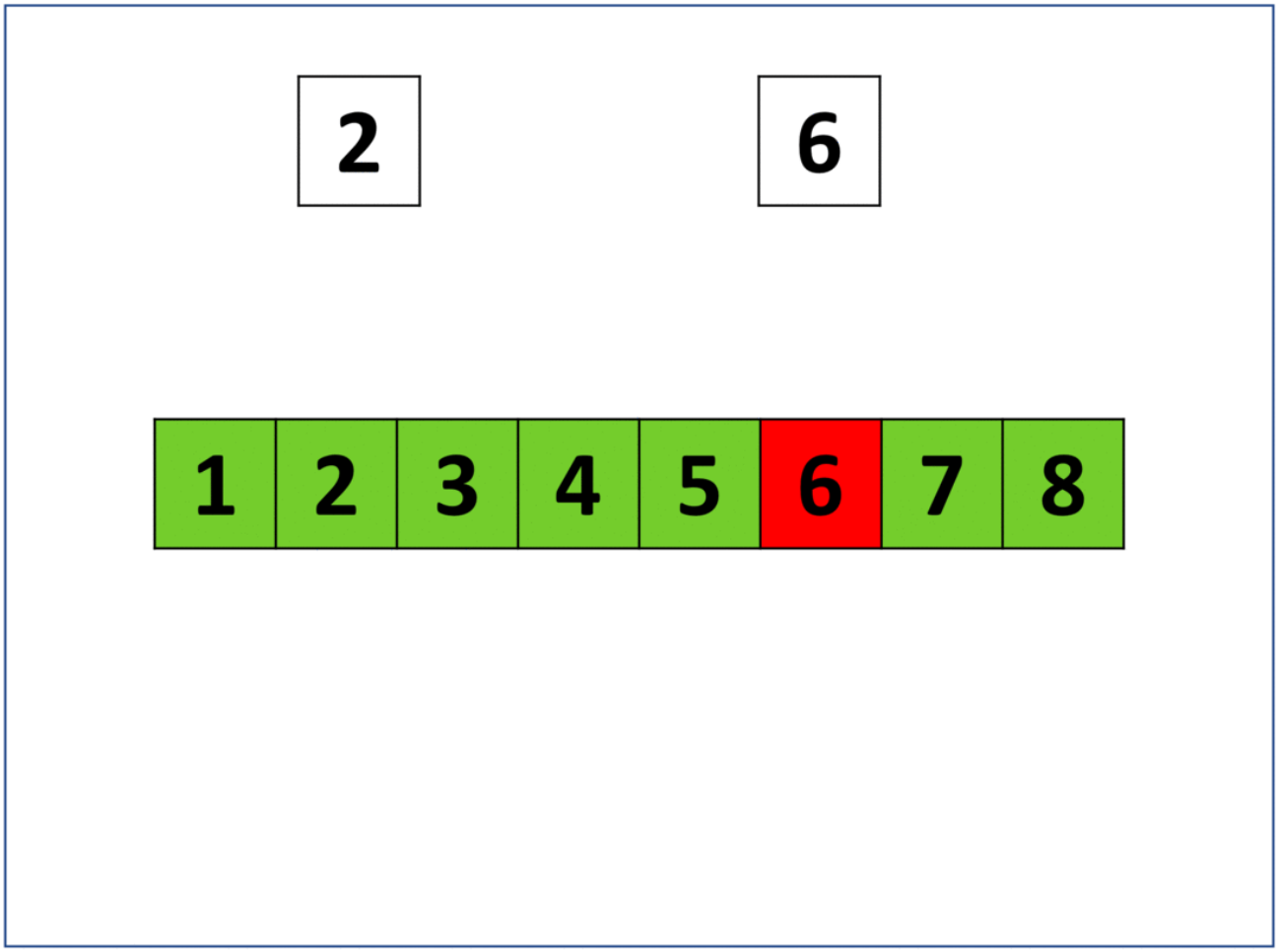
The inequality test uses an exclamation mark preceding the equal sign, if two operands are not equal then the condition becomes **True**. For example, the following condition will produce **True** as long as the value of `i` is not equal to 6:

In [ ]:
```
# Inequality Sign

i = 2
i != 6
```

When `i` equals 6 the inequality expression produces **False**.

In [ ]:
```
# Inequality Sign

i = 6
i != 6
```

See the number line below. when the condition is **True** the corresponding numbers are marked in green and for where the condition is **False** the corresponding number is marked in red. If we set `i` equal to 2 the operator is true as 2 is in the green region. If we set `i` equal to 6, we get a **False** as the condition falls in the red region.



We can apply the same methods on strings. For example, use an equality operator on two different strings. As the strings are not equal, we get a **False**.

In [ ]:
```
# Use Equality sign to compare the strings

"ACDC" == "Michael Jackson"
```

If we use the inequality operator, the output is going to be **True** as the strings are not equal.

In [ ]:
```
# Use Inequality sign to compare the strings

"ACDC" != "Michael Jackson"
```

Inequality operation is also used to compare the letters/words/symbols according to the ASCII value of letters. The decimal value shown in the following table represents the order of the character:

For example, the ASCII code for **!** is 21, while the ASCII code for **+** is 43. Therefore **+** is larger than **!** as 43 is greater than 21.

Similarly, the value for **A** is 101, and the value for **B** is 102 therefore:

In [ ]:
```
# Compare characters

'B' > 'A'
```

When there are multiple letters, the first letter takes precedence in ordering:

In [ ]:
```
# Compare characters

'BA' > 'AB'
```

Note: Upper Case Letters have different ASCII code than Lower Case Letters, which means the comparison between the letters in python is case-sensitive.

Branching - If...Else Statement

Branching allows us to run different statements for different inputs. It is helpful to think of an **if statement** as a locked room, if the statement is **True** we can enter the room and your program will run some predefined tasks, but if the statement is **False** the program will ignore the task.

For example, consider the blue rectangle representing an ACDC concert. If the individual is older than 18, they can enter the ACDC concert. If they are 18 or younger than 18 they cannot enter the concert.

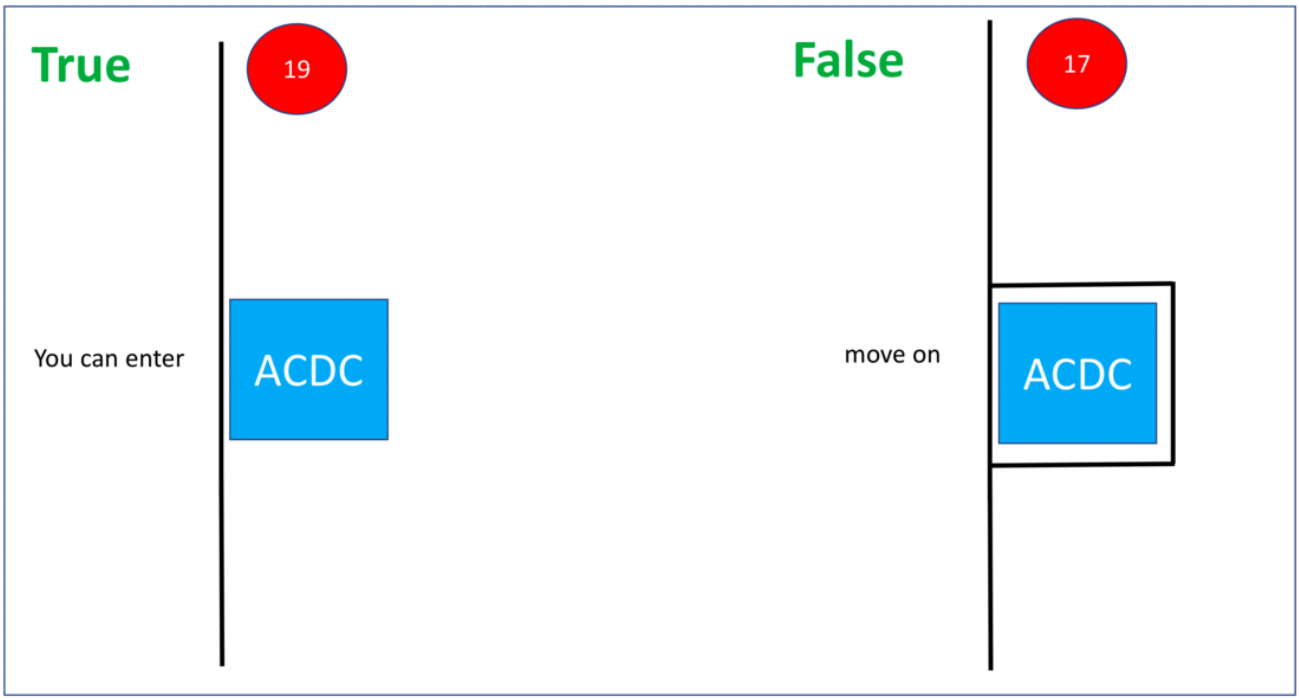
Use the condition statements learned before as the conditions need to be checked in the **if statement**. The syntax is as simple as `if` *`condition statement`* `:`, which contains a word `if`, any condition statement, and a colon at the end. Start your tasks which need to be executed under this condition in a new line with an indent. The lines of code after the colon and with an indent will only be executed when the **if statement** is **True**. The tasks will end when the line of code does not contain the indent.

In the case below, the tasks executed `print("you can enter")` only occurs if the variable `age` is greater than 18 is a True case because this line of code has the indent. However, the execution of `print("move on")` will not be influenced by the if statement.

In [ ]:
```
# If statement example

age = 19
#age = 18

#expression that can be true or false
if age > 18:

    #within an indent, we have the expression that is run if the condition is true
    print("you can enter" )

#The statements after the if statement will run regardless if the condition is true or fa
print("move on")
```

Try uncommenting the age variable

It is helpful to use the following diagram to illustrate the process. On the left side, we see what happens when the condition is **True**. The person enters the ACDC concert representing the code in the indent being executed; they then move on. On the right side, we see what happens when the condition is **False**; the person is not granted access, and the person moves on. In this case, the segment of code in the indent does not run, but the rest of the statements are run.

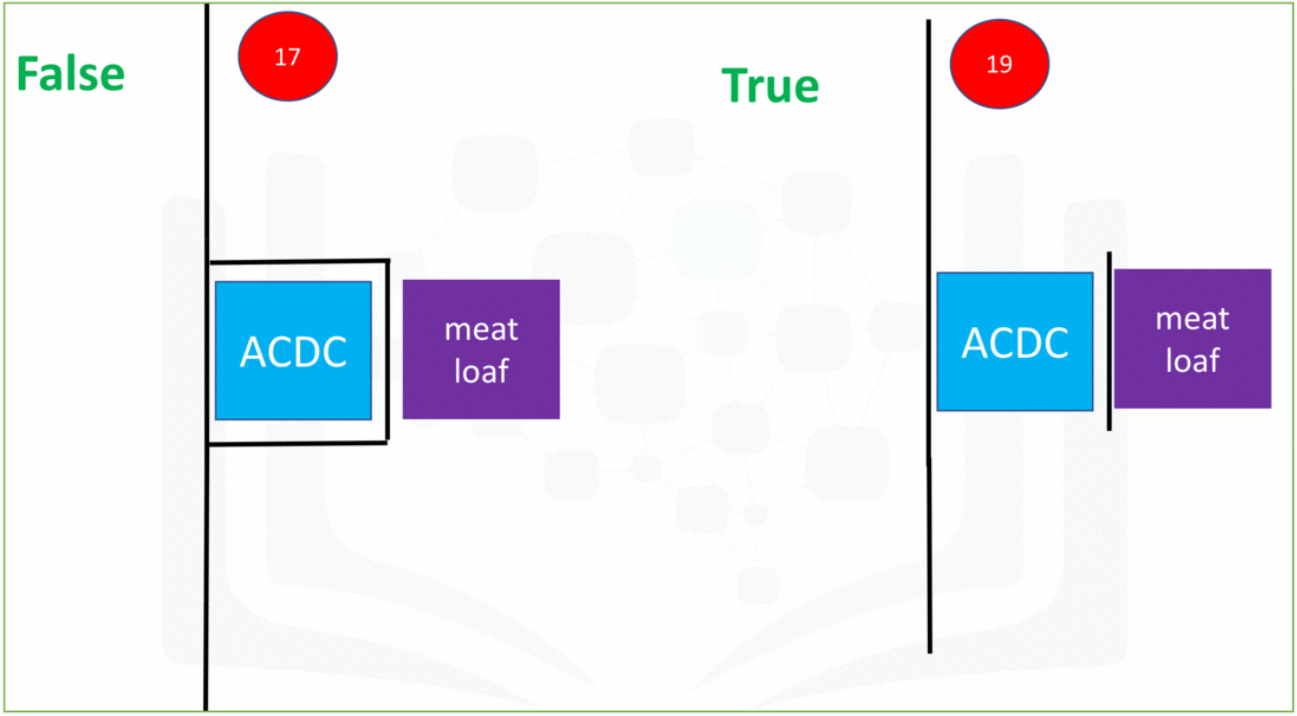


The `else` statement runs a block of code if none of the conditions are **True** before this `else` statement. Let's use the ACDC concert analogy again. If the user is 17 they cannot go to the ACDC concert, but they can go to the Meatloaf concert. The syntax of the `else` statement is similar as the syntax of the `if` statement, as `else :`. Notice that, there is no condition statement for `else`. Try changing the values of `age` to see what happens:

In [ ]:

```
# Else statement example

age = 18
# age = 19

if age > 18:
    print("you can enter" )
else:
    print("go see Meat Loaf" )

print("move on")
```

The process is demonstrated below, where each of the possibilities is illustrated on each side of the image. On the left is the case where the age is 17, we set the variable age to 17, and this corresponds to the individual attending the Meatloaf concert. The right portion shows what happens when the individual is over 18, in this case 19, and the individual is granted access to the concert.

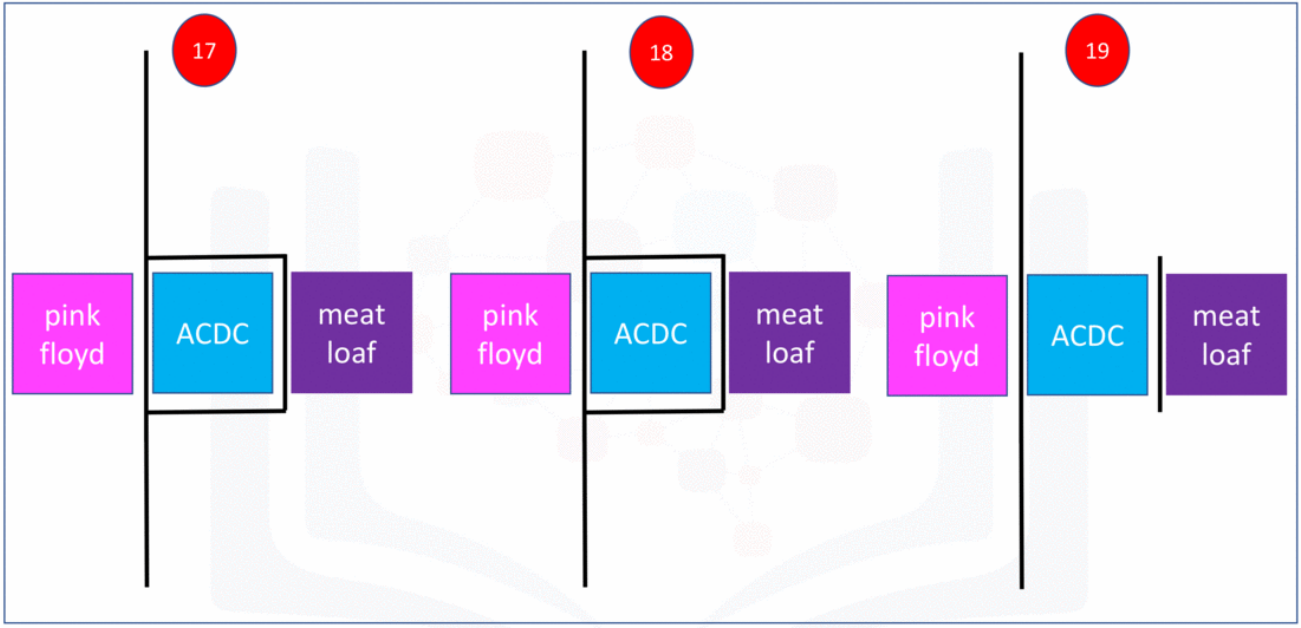


The `elif` statement, short for else if, allows us to check additional conditions if the condition statements before it are **False**. If the condition for the `elif` statement is **True**, the alternate expressions will be run. Consider the concert example, where if the individual is 18 they will go to the Pink Floyd concert instead of attending the ACDC or Meat-loaf concert. The person of 18 years of age enters the area, and as they are not older than 18 they can not see ACDC, but as they are 18 years of age, they attend Pink Floyd. After seeing Pink Floyd, they move on. The syntax of the `elif` statement is similar in that we merely change the `if` in `if` statement to `elif`.

In [ ]:

```
# Elif statment example

age = 18

if age > 18:
    print("you can enter" )
elif age == 18:
    print("go see Pink Floyd")
else:
    print("go see Meat Loaf" )

print("move on")
```

The three combinations are shown in the figure below. The left-most region shows what happens when the individual is less than 18 years of age. The central component shows when the individual is exactly 18. The rightmost shows when the individual is over 18.



Look at the following code:

In [ ]:
```
# Condition statement example

album_year = 1983
album_year = 1970

if album_year > 1980:
    print("Album year is greater than 1980")
n('do something..')
```

Feel free to change `album_year` value to other values -- you'll see that the result changes!

Notice that the code in the above **indented** block will only be executed if the results are **True**.

As before, we can add an `else` block to the `if` block. The code in the `else` block will only be executed if the result is **False**.

Syntax:

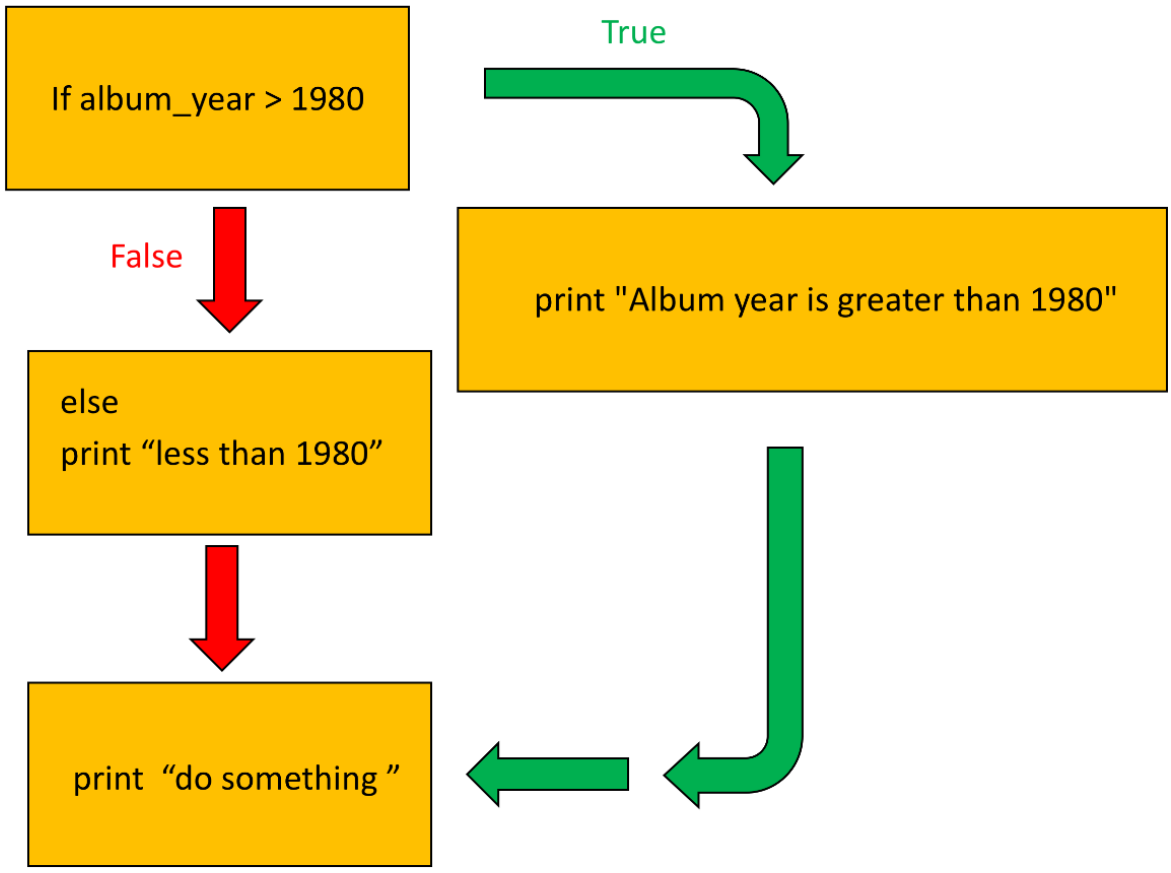
if (condition):

```
    # do something
```

else:

```
    # do something else
```

If the condition in the `if` statement is **False**, the statement after the `else` block will execute. This is demonstrated in the figure:



In [ ]:

```
# Condition statement example

album_year = 1983
#album_year = 1970

if album_year > 1980:
    print("Album year is greater than 1980")
else:
    print("less than 1980")

print('do something..')
```

Feel free to change the `album_year` value to other values -- you'll see that the result changes based on it!

Logical operators

Sometimes you want to check more than one condition at once. For example, you might want to check if one condition and another condition is **True**. Logical operators allow you to combine or modify conditions.

- `and`
- `or`
- `not`

These operators are summarized for two variables using the following truth tables:

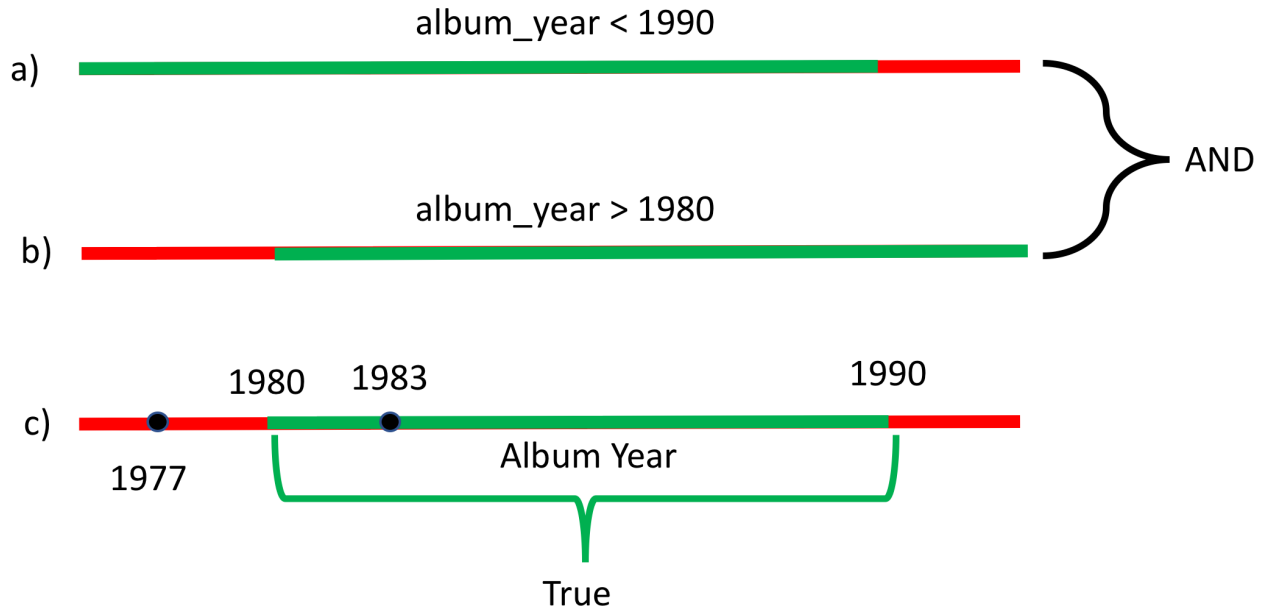
A	B	A & B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	Ture
True	True	True

A	A!
False	True
True	False

The `and` statement is only **True** when both conditions are true. The `or` statement is true if one condition is **True**. The `not` statement outputs the opposite truth value.

Let's see how to determine if an album was released after 1979 (1979 is not included) and before 1990 (1990 is not included). The time periods between 1980 and 1989 satisfy this condition. This is demonstrated in the figure below. The green on lines **a** and **b** represents periods where the statement is **True**. The green on line **c** represents where both conditions are **True**, this corresponds to where the green regions overlap.

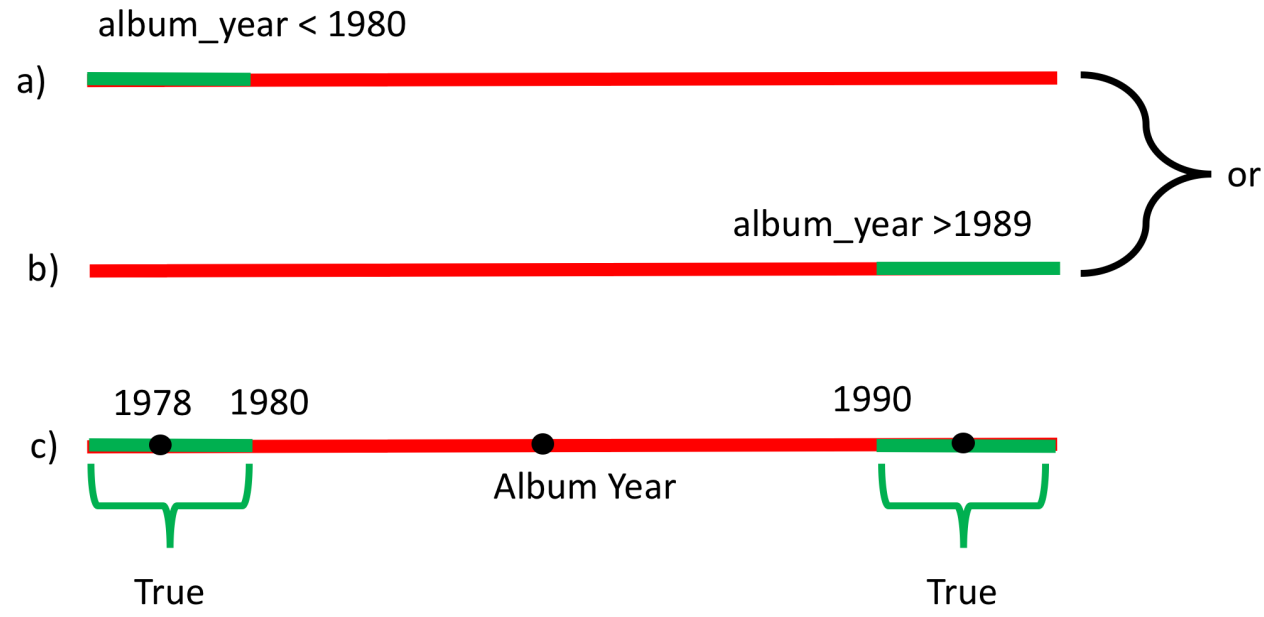


The block of code to perform this check is given by:

In [ ]:

```
# Condition statement example

album_year = 1980

if(album_year > 1979) and (album_year < 1990):
    print ("Album year was in between 1980 and 1989")

print("")
print("Do Stuff..")
```

To determine if an album was released before 1980 (~ - 1979) or after 1989 (1990 - ~), an **or** statement can be used. Periods before 1980 (~ - 1979) or after 1989 (1990 - ~) satisfy this condition. This is demonstrated in the following figure, the color green in **a** and **b** represents periods where the statement is true. The color green in **c** represents where at least one of the conditions are true.



The block of code to perform this check is given by:

In [ ]:
```
# Condition statement example

album_year = 1990

if(album_year < 1980) or (album_year > 1989):
    print ("Album was not made in the 1980's")
else:
    print("The Album was made in the 1980's ")
```

The `not` statement checks if the statement is false:

In [ ]:
```
# Condition statement example

album_year = 1983

if not (album_year == '1984'):
    print ("Album year is not 1984")
```

For Loop

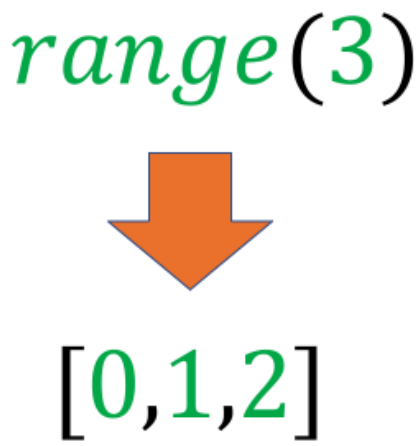
Range

Sometimes, you might want to repeat a given operation many times. Repeated executions like this are performed by **loops**. We will look at two types of loops, `for` loops and `while` loops.

Before we discuss loops lets discuss the `range` object. It is helpful to think of the range object as an ordered list. For now, let's look at the simplest case. If we would like to generate a sequence that contains three elements ordered from 0 to 2 we simply use the following command:

In [ ]:
```
# Use the range

range(3)
```

In [ ]:
```
range?
```



What is `for` loop?

The `for` loop enables you to execute a code block multiple times. For example, you would use this if you would like to print out every element in a list.
Let's try to use a `for` loop to print all the years presented in the list `dates`:

This can be done as follows:

In [ ]:
```
# For loop example

dates = [1980, 1990, 2000]
N = len(dates)

for i in range(N):
    print(dates[i])
```

The code in the indent is executed `N` times, each time the value of `i` is increased by 1 for every execution. The statement executed is to `print` out the value in the list at index `i` as shown here:

In this example we can print out a sequence of numbers from 0 to 7:

In [ ]:
```
# Example of for loop

for i in range(0, 8):
    print(i)
```

In Python we can directly access the elements in the list as follows:

In [ ]:
```
# Exmaple of for loop, loop through list

for year in dates:
    print(year)
```

For each iteration, the value of the variable `years` behaves like the value of `dates[i]` in the first example:

```
for year in dates:

    print(year)

      Dates=[1982,1980,1973]
```

We can change the elements in a list:

In [ ]:
```
# Use for loop to change the elements in list

squares = ['red', 'yellow', 'green', 'purple', 'blue']

for i in range(0, 5):
    print("Before square ", i, 'is',  squares[i])
    squares[i] = 'weight'
    print("After square ", i, 'is',  squares[i])
```

We can access the index and the elements of a list as follows:

In [ ]:
```
# Loop through the list and iterate on both index and element value

squares=['red', 'yellow', 'green', 'purple', 'blue']

for i, square in enumerate(squares):
    print(i, square)
```

While Loop

What is `while` loop?

As you can see, the `for` loop is used 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 condition is met? 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 say we would like to iterate through list `dates` and stop at the year 1973, then print out the number of iterations. This can be done with the following block of code:

In [ ]:
```
# While Loop Example

dates = [1982, 1980, 1973, 2000]

i = 0
year = 0

while(year != 1973):
    year = dates[i]
    i = i + 1
    print(year)

print("It took ", i ,"repetitions to get out of loop.")
```

In [ ]:
```
names = ["chisom", 'age', 'light']

i = 0
name =0

while(name =="age"):
    name = names[i]
    i += i
    print(name)

print("the are ", i , "repetitions to get ")
```

A while loop iterates merely until the condition in the argument is not met, as shown in the following figure:

Assignment Link

Now we would try out some practical examples with what we have learnt so far ! Let us try out this notebook

After Thoughts ??

What do you think so far ?

About this Instructor:

ChisomLoius is very passionate about Data Analysis and Machine Learning and does lot of free lance teaching and learning. Holding a B.Eng. in Petroleum Engineering, my focused is leveraging the knowledge of Data Science and Machine Learning to help build solutions in Education and High Tech Security. I currently work as a Petrochemist.

#
M
Ir

Visit our website, or further enquire more informatio via our email.
