Python Loops

What're Loops ?

- ✓ Loops mean repetition, looping back over the same block of code again and again.
- ✓ A loop statement allows us to execute a statement or group of statements multiple times.
- ✓ Loops can execute a block of code as long as a specified condition is reached.

Advantages of loops :

- ✓ It provides code re-usability.
- ✓ Using loops, we do not need to write the same code again and again.
- ✓ Using loops, we can traverse over the elements of data structures (array or linked lists)

Loops in Python

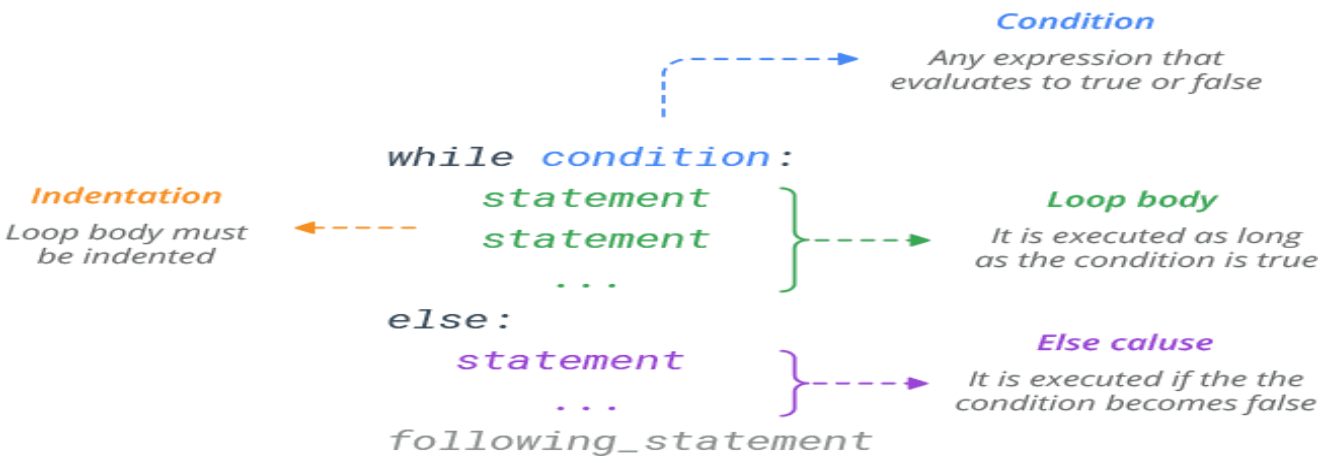
Python has two primitive loop commands**:**

- ✧ **While loop :** Loops through a block of code as long as a specified condition is True.
- ✧ **For loop :** When you know exactly how many times you want to loop through a block of code

Python While Loop

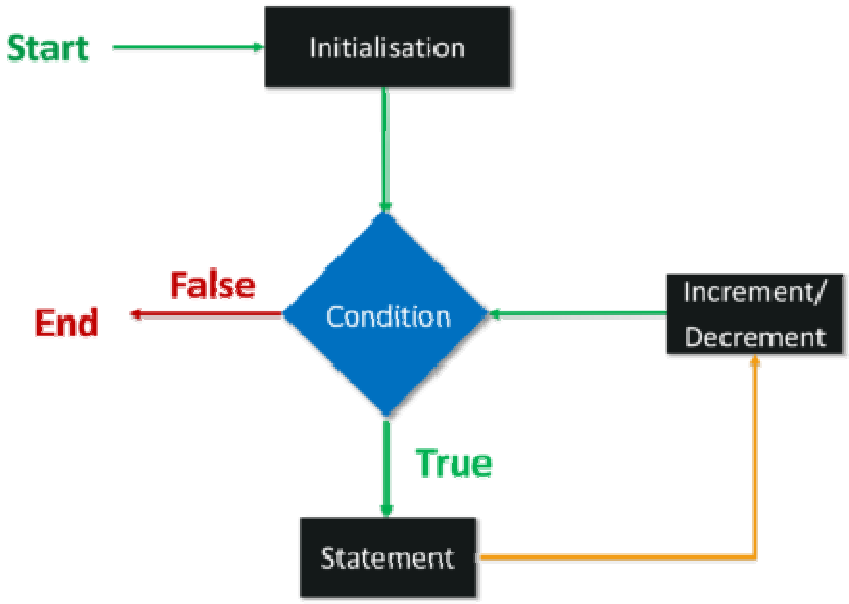
While loop is used to executing multiple times of statement or codes repeatedly until the given condition is true once it gets a false loop will stop . It's a **condition-controlled** loop

Here's the syntax of the while statement:



Concentrate on the following pieces that you need in order to create a for loop :

- ✓ Initial Value.
- ✓ The while keyword
- ✓ A condition that transates to either True or False
- ✓ A block of code that you want to execute repeatedly
- ✓ Increment and Decrement value



The above diagram displays the flow of the Python While Loop:

- ✓ When the program control reaches the while loop, the condition is checked.
- ✓ If the condition is true, the block of code under it is executed.
- ✓ After the increment/decrement , the condition is checked again.
- ✓ This continues until the condition becomes false. Then, the first statement, if any, after the loop is executed.

Example :

Code	Output:
a=3 while(a>0): print(a) a-=1	3 2 1

This loop prints numbers from 3 to 1.

*__Remember to indent all statements under the loop equally__

a. An Infinite Loop

Be careful while using a while loop. Because if you forget to increment the counter variable in python, or write flawed logic, the condition may never become false. In such a case, the loop will run infinitely, and the conditions after the loop will starve. To stop execution, press **Ctrl+C.**

b. The else statement for while loop

A while loop may have an else statement after it. When the condition becomes false, the block under the else statement is executed. However, it doesn't execute if you break out of the loop or if an exception is raised.

Code	Output:

a=3
while(a>0):
print(a)
a-=1
else:
print("Reached 0")
	3
	2
	1
	
	Reached 0

c. Single Statement while

Like an if statement, if we have only one statement in while's body, we can write it all in one line.

```
a=3
while a>0 : print(a) ; a-=1
```

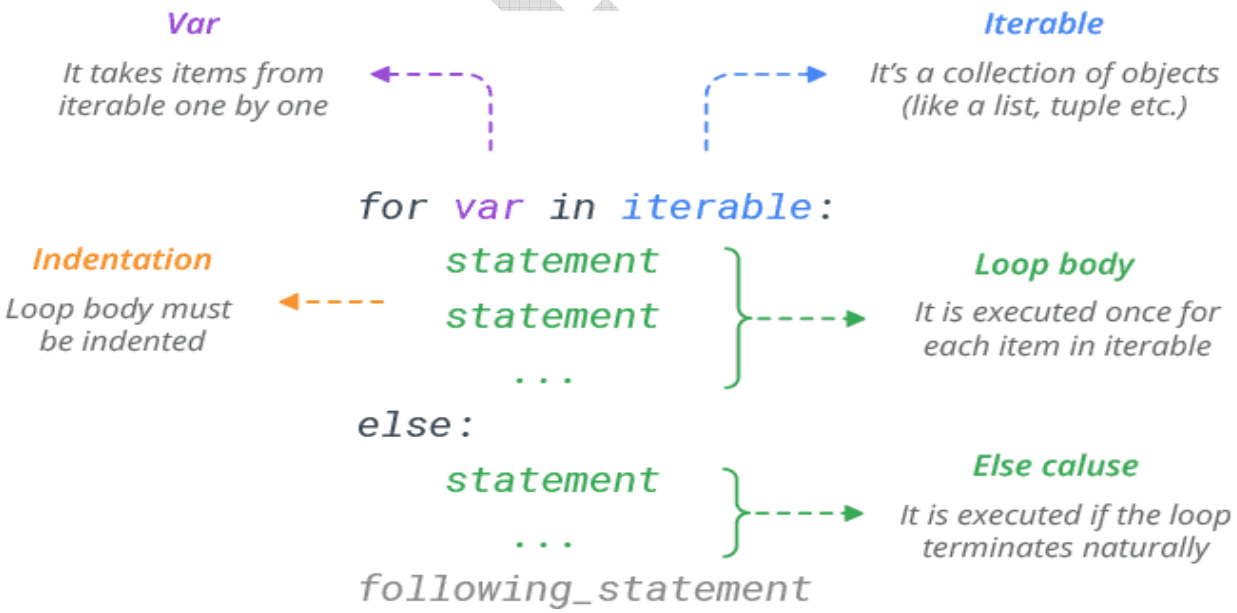
You can see that there were two statements in while's body, but we used semicolons to separate them. Without the second statement, it would form an infinite loop.

Python For Loop :

for loop is a programming concept that, when it's implemented, executes a piece of code over and over again "for" a certain number of times, based on a sequence.

In contrast to the while loop, there isn't any condition actively involved - you just execute a piece of code repeatedly for a number of times. In other words, while the while loop keeps on executing the block of code contained within it only till the condition is True, the for loop executes the code contained within it only for a specific number of times. This "number of times" is determined by a sequence or an ordered list of things.It **iterate over a sequence of items.**

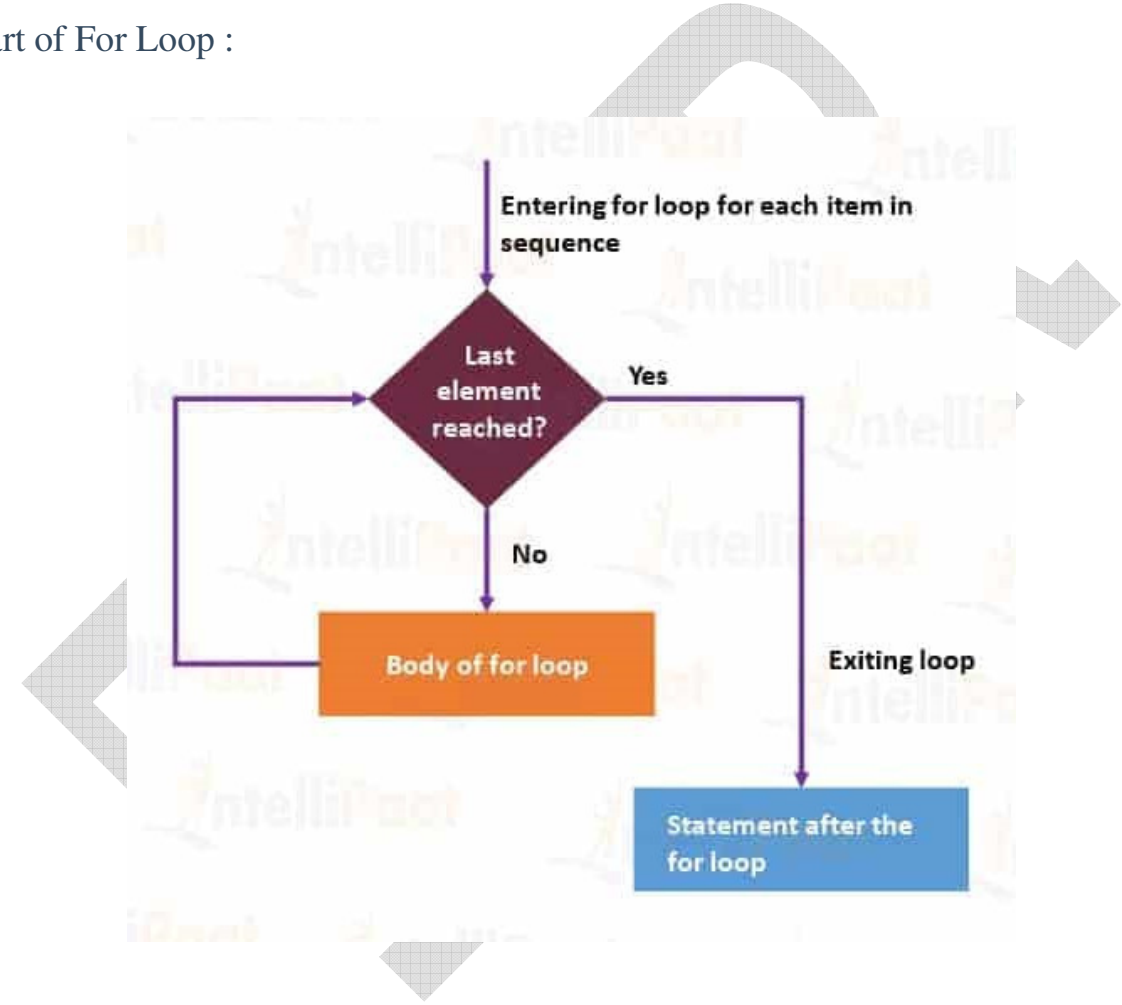
Here's the syntax of the for statement:



Concentrate on the following pieces that you need in order to create a for loop :

- ✓ The **for** keyword
- ✓ A variable
- ✓ The **in** keyword
- ✓ The range() function, which is an built-in function in the Python library to create a sequence of numbers
- ✓ The code that you want to execute repeatedly

Flowchart of For Loop :



The above diagram displays the flow of the Python For Loop:

- ✓ We grab the first item, and execute an expression on it.
- ✓ We return to the sequence and check if another item exists.
- ✓ If there is another item, we execute another expression.
- ✓ Otherwise, we end the loop

Lets see a Python for loop Example

```
for a in range(3):
    print(a)
```

0

1

2

If we wanted to print 1 to 3, we could write the following code.

```
for a in range(3):
        print(a+1)
```

1

2

3

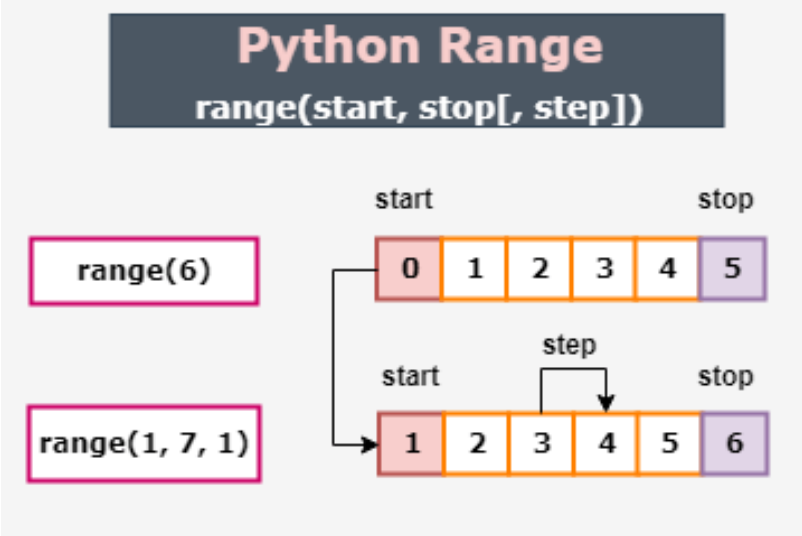
a.Iteration, Iterables, Iterators :

Term	Meaning
Iteration	The process of looping through the objects or items in a collection • **Definite** iteration, in which the number of repetitions is specified explicitly in advance (for loop) • **Indefinite** iteration, in which the code block executes until some condition is met (while loop)
Iterable	An object (or the adjective used to describe an object) that can be iterated over
Iterator	The object that produces successive items or values from its associated iterable

b. The range() function

The **range()** function in Python is an inbuilt method which is used to generate a list of numbers which we can iterate on using loops. The range() function is a renamed version in Python(3.x) of a function named xrange() in Python(2.x).

Python range function has basically three arguments.
Start: Integer representing the start of the range object.
Stop: Integer representing the end of the range object.
Step(optional): Integer representing the increment after each value



✓ When called with range(stop) – One argument, say n, it creates a sequence of numbers from 0 to n-1.

```
list(range(10))
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

*Note that range() returns an object of class range, not a list or tuple of the values. Because a range object is an iterable . We use the list function to convert the range object into a list object.

- ✓ Calling it with range(start, stop) – Two arguments , creates a sequence of numbers from the first to the second.

```
list(range(2,7))
```

[2, 3, 4, 5, 6]

- ✓ Calling it with range(start, stop, step) – Three arguments creates a sequence of numbers from the first to the second with intervals

```
list(range(2,12,2))
```

[2, 4, 6, 8, 10]

Remember, the interval can also be negative.

```
list(range(12,2,-2))
```

[12, 10, 8, 6, 4]

However, the following codes will return an empty list.

```
list(range(12,2))
list(range(2,12,-2))
list(range(12,2,2))
```

*__start__ should be smaller than __stop__ otherwise we will get an empty list

Please note the following points while using range() :

- ✓ The default value of 'step' is 1. It comes into play when the step argument is missing.
- ✓ A zero value for 'step' results in a ValueError.
- ✓ A non-integer value causes a TypeError.
- ✓ A non-zero integer step (>= stop) value would at least return a range with one element.
- ✓ Please note that range function only accepts integer arguments.

c. Iterating on lists or similar constructs

You aren't bound to use the range() function, though. You can use the loop to iterate on a list or a similar construct.

```
for a in [1,2,3]:
    print(a)
```

1
2
3

```
for i in {2,3,3,4}:
    print(i)
```

2
3
4

You can also iterate on a string.

```
for i in 'wisdom':
    print(i)
```

```
w
i
s
d
o
m
```

d. Iterating on indices of a list or a similar construct

The len() function returns the length of the list. When you apply the range() function on that, it returns the indices of the list on a range object. You can iterate on that.

```
list=['Romanian','Spanish','Gujarati']
for i in range(len(list)):
    print(list[i])
```

```
Romanian
Spanish
Gujarati
```

e. The else statement for for-loop

Like a while loop, a for-loop may also have an else statement after it. When the loop is exhausted, the block under the else statement executes.

```
for i in range(10):
    print(i)
else:
    print("Reached else")
```

0

.

.

9

Reached else

Like in the while loop, it doesn’t execute if you break out of the loop or if an exception is raised.

```
for i in range(10):
        print(i)
        if(i==7): break
else:
    print("Reached else")
```

0

1

.

.

7

Nested Loops

Nested looping is the process of looping one loop within the boundaries of others. So when the control flows from the outer loop to the inner loop it returns back to the outer-loop only when the inner loops are completed. Indentation is used to determine the body of the nested loops. Indentation starts the loop and the line from which it starts to be unindented represents the end of the mentioned loop . You can put a for loop inside a while, or a while inside a for, or a for inside a for, or a while inside a while. Or you can put a loop inside a loop inside a loop. You can go as far as you want.

Nested for loop : An example to use a for loop inside another for loop:

Code	Output:

`for i in range(1 , 6):`
`  for j in range(i):`
`  print( i , end="")`
`  print()`
	1
	2 2
	3 3 3
	4 4 4 4

Nested 'while' loop : An example to use a While loop inside another While loop:

Code	Output:

`i=6`
`while(i>0):`
`  j=6`
`  while(j>i):`
`  print("*",end=' ')`
`  j-=1`
`  i-=1`
`  print()`
	*
	* *
	* * *
	* * * *
	* * * * *

Nested while and for loop

An example to show a for loop inside a while loop

```
travelling = input("yes or no")
while travelling == "yes" :
    num = int(input("number of people"))
     for num in range( 1 , num+1):
         name = input("name")
         age = input("age")
         gender = input("gender")
         print(name)
         print(age)
         print(gender)
      travelling = input("oops missed someone ? ")
```

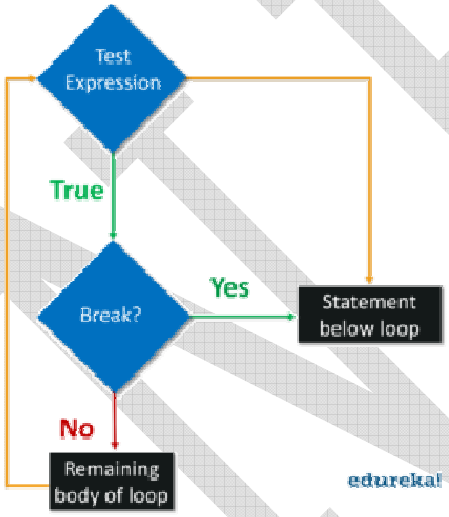
In this program we have used a **while** loop and inside the body of the **while** loop, we have incorporated a **for** loop.

Loop Control Statements

To control the flow of the loop or to alter the execution based on a few specified conditions we use the loop control statements discussed below. The control statements are used to alter the execution based on the conditions.In Python we have three control statements:

Break

Break statement is used to terminate the execution of the loop containing it. As soon as the loop comes across a break statement, the loop terminates and the execution transfers to the next statement following the loop.



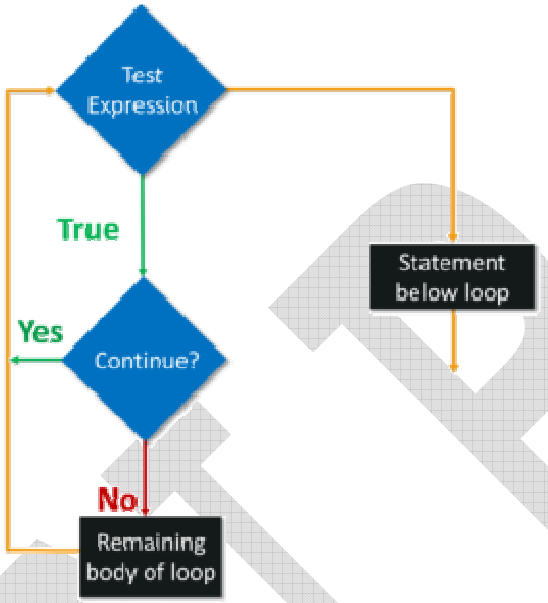
As you can see the execution moves to the statement below the loop, when the **break** returns true.Let us understand this better with an example:

Code :	Output :
for val **in** "string" : **if** val == "i": **break** print(val) print("the end")	s t r The end

Here the execution will stop as soon as the string “i” is encountered in the string. And then the execution will jump to the next statement.

Continue

The **continue** statement is used to skip the rest of the code in the loop for the current iteration. It does not terminate the loop like the **break** statement and continues with the remaining iterations.



When continue is encountered it only skips the remaining loop for that iteration only.

For example:

Code :	Output :
for val in "string" : if val == "i": Continue print(val) print("the end")	s t r n g The end

It will skip the string “i” in the output and the rest of the iterations will still execute. All letters in the string except “i” will be printed.

Pass

The **pass** statement is a null operation. It basically means that the statement is required syntactically but you do not wish to execute any command or code.

Take a look at the code below:

Code :	Output :
for val **in** "sing" :	s
if val == "i":	Pass block
pass	i
print("pass block")	n
print(val)	g
print("the end")	The end

The execution will not be affected and it will just print the **pass** block once it encounters the "i" string. And the execution will start at "i" and execute the rest of the remaining iterations.

'while' Loop Using The Break Statement

Lets understand how we use a **while** loop using a **break** statement with an example below:

```
i = 1
while i < 6 :
    print(i)
    if i == 3 :
      break
      i += 1
```

Output: it will print 1 2

The execution will be terminated when the iteration comes to 3 and the next statement will be executed.

Using Continue Statement

Let's take an example for continue statement in a **while** loop:

```
i = 1
while i < 6:
  print(i)
  if i == 3 :
    continue
    i += 1
```

Output: 1 2 4 5

Here the execution will be skipped, and the rest of the iterations will be executed.