Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a particular programming language. Python provides a variety of operators described as follows.

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations between two operands. It includes +(addition), -(subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**).

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Syntax	Description
+ (Addition)	Operand1 + operand2	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	Operand1 - operand2	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a - b = 10
/ (divide)	Operand1 / operand2	It returns the quotient after dividing the first operand by the second operand. It always yields a floating – point result. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	Operand1 * operand2	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	Operand1 % operand2	It returns the reminder after dividing the first operand by the second operand. It is the result of remainder obtained by performing ' //' operation For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	Operand1 **	It is an exponent operator represented as it calculates the first

	operand2	operand power to second operand. For Example, if a =10 and b = 20 then a**b=10^{20}
// (Floor division)	Operand1 // operand2	It gives the floor value of the quotient produced by dividing the two operands. i.e Integer division rounded toward minus infinity. The Result is float if any one of the or two operands is/are floating-point type with single digit precision(i.e digit '0'). The Result is integer if both the operands are integer type (irrespective of whether they are exactly divisible or not).

```
Script:
a = 20
b = 10
print("a+b = ",a+b)
print("a-b = ",a-b)
print("a*b = ",a*b)
print("a/b = ",a/b)
print("a%b = ",a%b)
print("a//b = ",a//b)
print("a**b = ",a**b)

output:
a+b =  30
a-b =  10
a*b =  200
a/b =  2.0
a%b =  0
a//b =  2
a**b =  10240000000000
```

```
>>> 10.5/2
5.25
>>> 10.5//2
5.0
>>> 10.5%2
0.5
>>> 10/2.0
5.0
>>> 10//2.0
5.0
>>> 10%2.0
0.0
>>> 10.5/2.0
5.25
>>> 10.5//2.0
5.0
>>> 10.5%2.0
0.5
```

```
>>> 10/2
5.0
>>> 10//2
5
>>> 10%2
0
>>> 9/2
4.5
>>> 9//2
4
>>> 9%2
1
>>> 2**0
1
>>> 2.0**0
1.0
>>> 10**-2
0.01
```

```
>>> 10/0
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
>>> 10//0
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    10//0
ZeroDivisionError: integer division or modulo by
zero
>>> 1010%0
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    1010%0
ZeroDivisionError: integer division or modulo by
zero
```

Comparison (Relational) operator

Comparison operators are used to compare the values of the two operands and returns boolean true or false accordingly. Chaining of relational operators is allowed and result is false if atleast one of them is false otherwise the result is true. Boolean value arguments are internally interpreted as 1 (True) and 0 (False).
The comparison operators are described in the following table.

Operator	**Syntax**	**Description**
== (Equals operator)	Operand1 == operand2	If the value of two operands is equal, then the condition becomes true.
!= (Not Equals Operator)	Operand1 != operand2	If the value of two operands is not equal then the condition becomes true.

<= (Less than or equal operator)	Operand1 <= operand2	If the first operand is less than or equal to the second operand, then the condition becomes true.
>= (Greater than or equal operator)	Operand1 >= operand2	If the first operand is greater than or equal to the second operand, then the condition becomes true.
> (Greater than operator)	Operand1 > operand2	If the first operand is greater than the second operand, then the condition becomes true.
> (Less than operator)	Operand1 < operand2	If the first operand is less than the second operand, then the condition becomes true.

Script:
a = 20
b = 10
print("a>b = ",a>b)
print("a<b = ",a<b)
print("a>=b = ",a>=b)
print("a<=b = ",a<=b)
Output:
a>b = True
a<b = False
a>=b = True
a<=b = False
	Script:
	a = 'Engineering'
	b = 'science'
	print("a>b = ",a>b)
	print("a<b = ",a<b)
	print("a>=b = ",a>=b)
	print("a<=b = ",a<=b)
	Output:
	a>b = False
	a<b = True
	a>=b = False
	a<=b = True
		Script:
		a = True
		b = False
		print("a>b = ",a>b)
		print("a<b = ",a<b)
		print("a>=b = ",a>=b)
		print("a<=b = ",a<=b)
		Output:
		a>b = True
		a<b = False
		a>=b = True
		a<=b = False
			>>> 10<20<30
			True
			>>> 10>=20<30
			False
			>>> 10<20<30
			True
			>>> 10>=20<30
			False
			>>> 10=='Cse'
			False
			>>> 'cse' == 'cse'
			True
			>>> 10 ==20==30
			False
			>>> 'a' == 97
			False
			>>> 10 == 5+5
			True
				>>> (10+20j) == 'Ram'
				False
				>>> 10 == 10.0
				True
				>>> 10 == '10'
				False
				>>> 10 == 10.1
				False
				>>> 10.10 ==10.1
				True
				>>> 1 == True
				True
				>>> 10 == True
				False
				>>> 1.0 == True
				True

Python assignment operators

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Syntax	Description
= (Assignment	operand = expression	It assigns the the value of the right expression to the left operand.

Operator)		
+= (Addition assignment operator)	Operand1 += (operand2 / expression)	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-= (Subtraction assignment operator)	Operand1 -= (operand2 / expression)	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*= (Multiplication assignment operator)	Operand1 *= (operand2 / expression)	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%= (Modulo assignment operator)	Operand1 %= (operand2 / expression)	It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**= (Exponent assignment operator)	Operand1 **= (operand2 / expression)	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//= (Floor Division assignment operator)	Operand1 //= (operand2 / expression)	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Assignment statement forms

Operation	**Interpretation**
spam = 'Spam'	Basic form
spam, ham = 'yum', 'YUM'	Tuple assignment (positional)
[spam, ham] = ['yum', 'YUM']	List assignment (positional)
a, b, c, d = 'spam'	Sequence assignment, generalized
a, *b = 'spam'	Extended sequence unpacking (Python 3.X)
spam = ham = 'lunch'	Multiple-target assignment
spams += 42	Augmented assignment (equivalent to spams = spams + 42)

Script:

```
a = 10
a
```

Output:

```
10
```

Script:

```
a=b=2
print("a=",a)
print("b=",b)
```

Output:

```
a= 2
b= 2
```

Script:

```
a,b,c,d = 10,20,30,40
print("a=",a)
print("b=",b)
print("c=",c)
print("d=",d)
```

Output:

```
a= 10
b= 20
c= 30
d= 40
```

Script:

```
a,b,c,d = 10,20,30,40,50
print("a=",a)
print("b=",b)
print("c=",c)
print("d=",d)
```

Output:

```
ValueError
Traceback (most recent call
last)
<ipython-input-10-
305880621cce> in <module>
----> 1 a,b,c,d =
10,20,30,40,50
      2 print("a=",a)
      3 print("b=",b)
      4 print("c=",c)
      5 print("d=",d)

ValueError: too many values to
unpack (expected 4)
```

Note: Python doesn't have increment and decrement operators

```
>>> ++10
10
>>> --10
10
>>> ---10
-10
>>> 10++
Syntax Error: invalid syntax
```

Bitwise operator

The bitwise operators perform bit by bit operation on the values of the two operands. The arguments to bitwise operators can be integer type or Boolean type otherwise it gives syntax error. The result of bitwise operation is always an integer type.

For example,

```
if a = 7;
   b = 6;
then, binary (a) = 0111
    binary (b) = 0011

hence, a & b = 0011
a | b = 0111
             a ^ b = 0100
       ~ a = 1000
```

Operator	Syntax	Description
& (binary and)	Op1 & op2	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	Op1 | op2	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	Op1 ^ op2	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	~op1	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	Op1 << op2	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	Op1 >> op2	The left operand is moved right by the number of bits present in the right operand.

```
>>> ~4
-5
>>> ~10
-11
>>> ~32
-33
>>> ~-4
3
>>> 10<<2
40
>>> 10>>2
2
```

```
>>> -10>>2
-3
>>> True<<2
4
>>> 10 & 2
2
>>> 10 | 2
10
>>> 10 ^ 2
8
```

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Logical operators are useful to construct compound conditions. A compound condition is a combination of more than one simple condition. Simple conditions are evaluated to True or False first and then decision is taken to know whether the entire compound statement is True or False. These logical operators accept arguments of all types. It takes two arguments. Python supports the following logical operators.

Operator	syntax	Description
and	Exp1 and exp2	For both Boolean type of arguments, If both the expressions are true, then the condition will be true else the result is false. For Example, If a and b are the two expressions, a → true, b → true => a and b → true.

		For both non-boolean type of arguments, if exp1 evaluates to false, it returns exp1, otherwise it returns exp2.
or	Exp1 or Exp2	For both Boolean type of arguments, If either of the expressions are true, then the condition will be true else the result is false. For Example, If a and b are the two expressions, a → true, b → false => a or b → true. For both non-boolean type of arguments, if exp1 evaluates to false, it returns exp2, otherwise it returns exp1.
not	not Exp	If an expression **a** is true then not (a) will be false and vice versa.

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

>>> False or True
```

```
True
>>> True or True
True
>>> False or False
False
>>> not True
False
>>> not False
True
```

```
>>> 10 and 20
20
>>> 0 and 20
0
>>> 10+20 and 30+50
80
>>> 10 or 20
10
>>> 0 or 20
20
>>> 10+20 or 30+50
30
```

```
>>> not 10
False
>>> not 0
True
>>> not str()
True
>>> not ''
True

>>> not 'cse'
False
```

```
>>> 10 or 10/0
10
>>> 0 or 10/0
Traceback (most recent
call last):
  File "<pyshell#26>", line
1, in <module>
    0 or 10/0
ZeroDivisionError:
division by zero
```

Membership Operators

Python membership operators are used to check the membership of value inside a Python data structure or in sequences such as strings, tuples or dictionaries.. If the value is present in the data structure, then the resulting value is true otherwise it returns false.
The following are Membership operators:

Operator	Syntax	Description

in	**Op1 in op2** Here, op1 is an element to check for its presence in sequence represented by op2.	This operator returns True if an element is found in the specified sequence otherwise it returns False. i.e It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	**Op1 not in op2** Here, op1 is an element to check for its absence in sequence represented by op2.	This operator returns True if an element is not found in the specified sequence otherwise it returns False. It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Script:

```
x = 'cat under the table'
y = {3:'a',4:'b'}
print('under' in x)

print('the' not in x)

print('table' not in x)

print(3 in y)

print('b' in y)
```

output:

```
True
False
False
True
False
```

Script:

```
names = ["raj","antony","alice","bob"]
for name in names:
    print(name)
```

output:

```
raj
antony
alice
bob
```

Script:

```
# Python program to illustrate
# Finding common member in list
# using 'in' operator
list1=[1,2,3,4,5]
list2=[6,7,8,9,5,10]
for item in list1:
    if item in list2:
        print("overlapping--------",item)
    else:
        print("not overlapping ------- ",item)
```

output:

```
not overlapping--------------1
not overlapping--------------2
not overlapping--------------3
not overlapping--------------4
overlapping--------------5
```

Script:

```
# Python program to illustrate
# 'in and 'not in' operator
x = 24
y = 20
list = [10, 20, 30, 40, 50 ];

if ( x not in list ):
   print ("x is NOT present in
given list")
else:
   print ("x is present in given
list")

if ( y in list ):
   print ("y is present in given
list")
else:
   print ("y is NOT present in
given list")
```

output:

```
x is NOT present in given
list
y is present in given list
```

Identity Operators

These operators compare the memory locations of two objects. The memory location of an object can be seen using the **id ()** inbuilt function. This function returns an integer number, called the identity number that internally represents the memory location of an object.

>>> a = 10
>>> b = 10
>>> id(a)
1362311568
>>> id(b)
1362311568
	>>> b = 'Alice'
	>>> id(b)
	36037440
	>>> id(a)
	1362311568

The following are the Identity Operators:

Operator	Syntax	Description
is	Op1 is op2	It is evaluated to be true if both the references (op1 & op2) point to the same object. i.e if the identity numbers of two objects are same, then it will return True, otherwise it returns False.
is not	Op1 is not op2	It is evaluated to be true if both the references (op1 & op2) do not point to the same object. i.e if the identity numbers of two objects are different, then it will return True, otherwise it returns False.

Script:
a = 10
b = 20
print(a is b)
print(a is not b)
 	
Output:
`False`
`True`
	Script:
	a = "bob"
	b = "Bob"
	print("Address of a is:",id(a))
	print("Address of b is:",id(b))
	print("the result of 'a is b:'",a is b)
	print("the result of 'a is not b:'",a is not b)
	Output:
	`Address of a is: 57882400`
	`Address of b is: 57881920`
	`the result of 'a is b:' False`
	`the result of 'a is not b:' True`
		Script:
		l1 = [10,15.29,"bob",(10+20j),True]
		l2 = [10,15.29,"bob",(10+20j),True]
		print("Address of l1 is:",id(l1))
		print("Address of l2 is:",id(l2))
		print("the result of 'l1 is l2:'",l1 is l2)
		print("the result of 'l1 is not l2:'",l1 is not l2)
		print("the result of 'l1 == l2:'",l1 == l2)
		Output:
		`Address of l1 is: 57311064`
		`Address of l2 is: 57310864`
		`the result of 'l1 is l2:' False`
		`the result of 'l1 is not l2:' True`
		`the result of 'l1 == l2:' True`
			Script:
			a = "bob"
			b = "Bob"
			if(a is b):
			    print("a and b have same identity number")
			else:
			    print("a and b have different identity number")
			
			Output:
			`a and b  have different identity number`

Operator Precedence

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

Operator	Description
( )	Parenthesis
**	The exponent operator is given priority over all the others used in the expression.
~, +, -	The negation, unary plus and minus.
*, / ,%, //	The multiplication, divide, modules, reminder, and floor division.
+ , -	Binary plus and minus
>> ,<<	Left shift and right shift
&	Binary and.
^, |	Binary xor and or
<=, < ,>, >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<>, ==, !=	Equality operators.
= %,= /,= //,= -, = +,= *,= **,=	Assignment operators
isIs, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

```
>>> 7*3/7
3.0
>>> 2**3**2
512
>>> 3/2*4+3+(10/4)**3-2
22.625
>>> 8/4/2
1.0
```

Python Conditional Statements

If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if - else statement inside another if statement.
The ElIf Statement	If we want to execute some code where one of the several conditions is true,then use the else if statement.

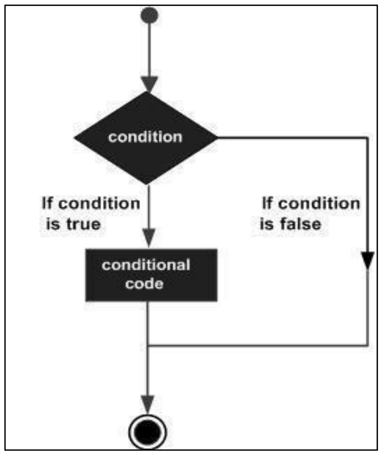
Indentation in Python

- For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. Python represents block structure and nested block structure with indentation. Indentation refers to spaces that are used in the beginning of a statement. The statements with same indentation belong to same group/block called a ***suite***.
- The empty block -- Use the **'*pass*'** no-op statement.
- Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.
- Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

- **Benefits of the use of indentation to indicate structure:**
  ● Reduces the need for a coding standard. Only need to specify that indentation is 4 spaces and no hard tabs.
  ● Reduces inconsistency. Code from different sources follows the same indentation style. It has to.
  ● Reduces work. Only need to get the indentation correct, not *both* indentation and brackets.
  ● Reduces clutter. Eliminates all the curly brackets.
  ● If it looks correct, it is correct. Indentation cannot fool the reader.

The if statement

The if statement is used to execute one or more statements depending on whether a condition is true or not. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
        if expression:
            block of statement(s) to be executed if condition is True
```

Example 1:Program to print if the given integer value is even

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Example 2 : Program to print the largest of the three numbers.

```
a = int(input("Enter value for a? "))
b = int(input("Enter value for b? "))
c = int(input("Enter value for c? "))
if a>b and a>c:
    print("a is largest")
if b>a and b>c:
    print("b is largest")
if c>a and c>b:
    print("c is largest")
```

Output:

```
Enter value for a? 10
Enter value for b? 20
Enter value for c? 30
c is largest
```

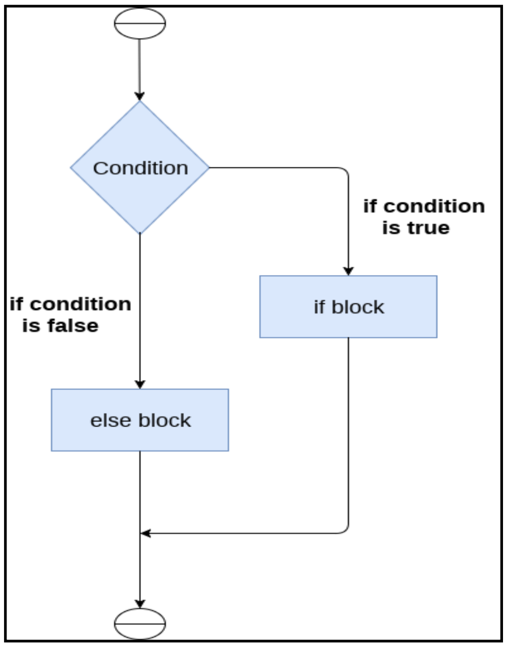
The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

The syntax of the if-else statement is given below.

```
if condition:
        block of statements to be executed if condition is True
else:
        block of statements to be executed if condition is False
```



Example 1 : Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
   print("You are eligible to vote !!")
else:
   print("Sorry! you have to wait !!")
```

Output:

```
Enter your age? 18
You are eligible to vote !!

Enter your age? 52
You are eligible to vote !!
```

Example 2: Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num%2 == 0:
   print("Number is even...")
else:
   print("Number is odd...")
```

Output:

```
enter the number?25
Number is odd...
enter the number?22
Number is even...
```

Example 3: The following example will output "Have a nice weekend!" if the current day is Friday, Otherwise, it will output "Have a nice day!":.

```
d = "Friday"
if d == " Friday ":
    print("Have a nice weekend!")
else:
    print("Have a nice day")
```

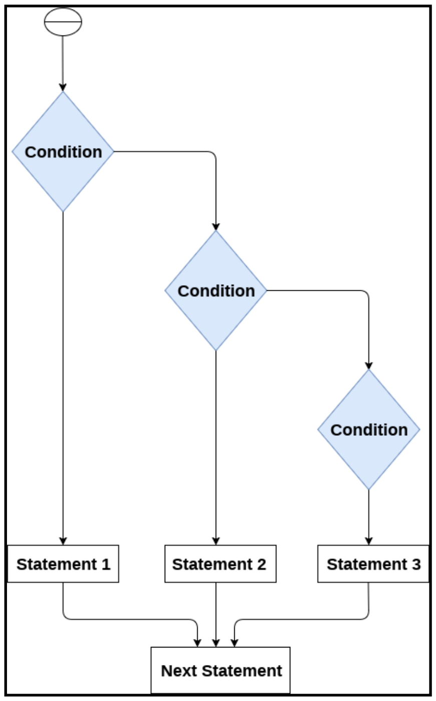
Output:

```
Have a nice day
```

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.



The syntax of the elif statement is given below.

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Example 1

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50")
elif number==100:
    print("number is equal to 100")
else:
    print("number is not equal to 10, 50 or 100")
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

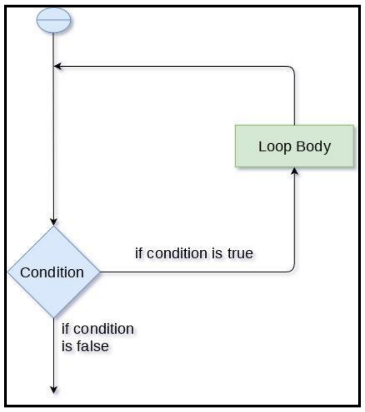
Example 2

```
marks = int(input("Enter the marks? "))
if marks > 85 and marks <= 100:
   print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
   print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
   print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
   print("You scored grade C ...")
else:
   print("Sorry you are fail ?")
```

Python Loops

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of loops

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Redundancy of code is less.
   Suppose if you take any example, to print 1 to 100 numbers we need to use 100 printf() statements. But if you use loop then one printf() statement is enough for this example.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

Loop Statement	**Description**
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

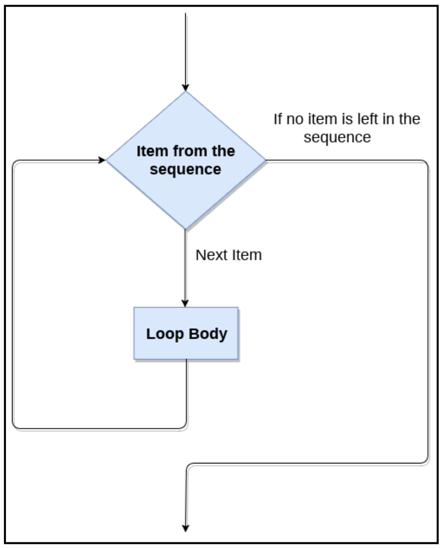
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

Python for loop

- The for **loop in Python** is used to iterate the statements or a part of the program several times.
- It is frequently used to traverse the data structures like list, tuple, or dictionary. i.e it is useful to iterate over the elements of a sequence or to execute a group of statements repeatedly depending upon the number of elements in the sequence.

The syntax of for loop in python is given below.

```
for iterating_var in sequence:
        statement(s)
```



Nested for loop in python

Python allows to write one loops inside another loop. The inner loop is executed n number of times for each iteration of the outer loop. We can write a 'for loop' inside a 'while loop' or vice-versa or a 'for loop' inside another 'for loop' or 'while loop' inside another 'while loop'. Such loops are called as ***'nested loops'***.

The syntax of the nested for loop in python is given below.

```
for iterating_var1 in sequence:
    #block of statement(s) inside 'outer for loop'
    for iterating_var2 in sequence:
        #block of statement(s) inside 'inner for loop'
#Other statements
```

Example 1: A program to illustrate the concept of 'Nested loop'

```
for i in range(3):
   for j in range(4):
      print('i=%d \t j=%d' % (i,j))
   print('----------------')
```

Output:

```
i=0         j=0
i=0         j=1
i=0         j=2
i=0         j=3
----------------
i=1         j=0
i=1         j=1
i=1         j=2
i=1         j=3
----------------
i=2         j=0
i=2         j=1
i=2         j=2
i=2         j=3
----------------
```

Example 2: A program to display the following pattern for given number of rows?

```
*
**
***
****
```

```
n = int(input("Enter the number of rows you want to print?"))
i,j=0,0
for i in range(0,n):
    print()
    for j in range(0,i+1):
        print("*",end="")
```

Output:
```
Enter the number of rows you want to print?5
*
**
***
****
*****
```

Using '*else suite*' with for loop

- Unlike other languages like C, C++, or Java, python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.
- This else suite is useful when we write programs where searching for an element is done is in a sequence. When the element is not found, we can indicate that in the else suite easily.

The syntax of the 'else suite' with for loop in python is given below.

```
for(var in sequence):
    #Statement(s)
else:
    #Statement(s)
```

Example 1: # A Program to illustrate 'else suite' with for loop
```
for i in range(0,5):
    print(i)
else:
    print("for loop completely exhausted, since there is no break.")
```

In the above example, for loop is executed completely since there is no break statement in the loop. The control comes out of the loop and hence the else block is executed.

Output:
```
0
1
2
3
4
for loop completely exhausted, since there is no break.
```

Example 2
```
for i in range(0,5):
    print(i)
    break
else:
    print("for loop is exhausted")
print("The loop is broken due to break statement...came out of loop")
```

In the above example, the loop is broken due to break statement therefore the else statement will not be executed. The statement present immediate next to else block will be executed.

Output:

```
0
The loop is broken due to break statement...came out of loop
```

Example 3

```
The loop is broken due to break statement...came out of loop
list = [1,2,3,4,5]
search_element = int(input("enter element to search in list?:"))
for i in list:
    if i == search_element:
        print('element = %d found in the list' % (search_element))
        break
else:
    print('element not found')
```

Output:

```
enter element to search in list?:6
element not found

enter element to search in list?:3
element = 3 found in the list
```

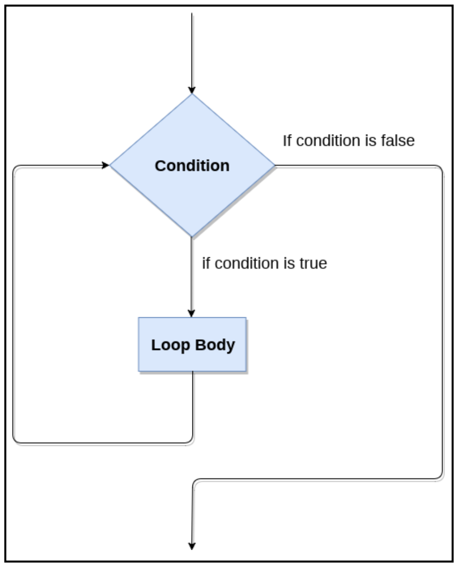
Python while loop

- The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.
- It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax is given below.

```
while expression:
    Block of statement(s) to execute repeatedly until the expression evaluates to False.
```

Here, the statements can be a single statement or the group of statements. The expression should be any valid python expression resulting into true or false. The true is any non-zero value.



Example 1: A Program to display numbers from 1 to 10

```
i=1;
while i<=10:
   print(i)
   i=i+1
Print(“End of … ….”)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
End of … ….
```

Example 2: A Program to display Multiplication table of given number

```
i=1
number = int(input("Enter the number:"))
while i<=10:
   print("%d X %d = %d \n" % (number,i,number*i));
   i = i+1;
```

Output:

```
Enter the number: 10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30
```

```
10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100
```

Example 3: A Program to display even numbers between 100 and 200

```python
number = 100
while number>=100 and number<=200:
    print(number)
    number+=2
```

Output:

```
100

102

104

106

108

110

112

114

116

118

120

122

124

126

128

130

132

134

136

138

140

142

144

146
```

```
148
150
152
154
156
158
160
162
164
166
168
170
172
174
176
178
180
182
184
186
188
190
192
194
196
198
200
```

Infinite while loop

- If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.
- Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.
- Infinite loops are drawbacks in a program because when the user is caught in an infinite loop, he/she cannot understand how to come out of the loop. So, it is always recommended to avoid infinite loops in any program.

Example 1

```
while (1):
    print("Hi! we are inside the infinite while loop")
```

Output:

```
Hi! we are inside the infinite while loop
(infinite times)
```

Example 2

```
var = 1
while var != 2:
    i = int(input("Enter the number?"))
    print ("Entered value is %d"%(i))
```

Output:

```
Enter the number?102
Entered value is 102
Enter the number?102
Entered value is 102
Enter the number?103
Entered value is 103
Enter the number?103
(infinite loop)
```

Using ‘else suite’ with Python while loop

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

The syntax of the ‘else suite’ with while loop in python is given below.

```
While(condition):
    #Statement(s)
else:
    #Statement(s)
```

Consider the following example.

Example 1

```
i=1;
while i<=5:
    print(i)
    i=i+1
else:
    print("The while loop exhausted");
```

Output:

```
1
2
3
4
```

```
5
The while loop exhausted
```

Example 2

```
i=1;
while i<=5:
   print(i)
   i=i+1;
   if(i==3):
      break
else:
   print("The while loop exhausted");
```

Output:

```
1
2
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

Control Statement	Description
break	Terminates the execution of loop statement and continues with execution of the statement immediately following the loop.
continue	Causes to skip the execution of the remaining statements after 'continue' statement in the loop body and continues with testing of the condition of the loop.
pass	The pass statement in Python is used when a control statement is required syntactically but you do not want any command or code to execute as a part of loop.

break statement

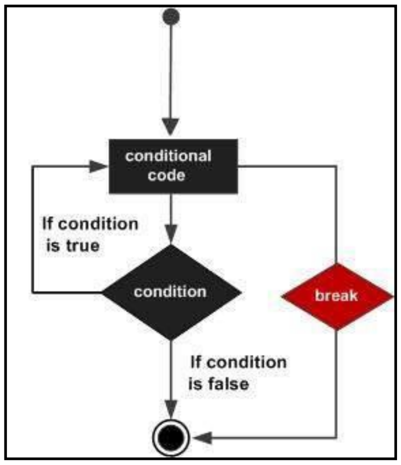
break statement can be used within for as well as while loops. Python Interpreter terminates the current loop when it executes break statement and resumes execution with the statement following the loop body, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop.

In case of nested loops, the successful execution of 'break statement' causes Python interpreter to terminate the execution of the loop having 'break statement' and continues with the execution of statements of outer loop/enclosing loop.

The syntax of the break is given below.

```
#loop statements
break
```



Example 1

```
str = "python"
for i in str:
    if i == 'o':
        break
    print(i)
```

Output:

```
p
y
t
h
```

Example 2:

```
i = 0;
while 1:
    print(i," ",end="")
    i=i+1
    if i == 10:
        break
print("came out of while loop")
```

Output:

```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

Example 3:

```
# A program to print multiplication table interactively illustrating the usage of 'break statement'

while 1:
    i=1
    choice = input("Do you want to continue printing the table, press 'yes' or 'no'?: ")
    if choice == 'yes':
        n = int(input("enter the table to print: "))
        while i<=10:
            print("%d X %d = %d\n"%(n,i,n*i))
            i = i+1
    else:
        break
```

Output:

```
Do you want to continue printing the table, press 'yes' or 'no'?: yes
enter the table to print: 2
2 X 1 = 2
```

```
2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 'yes' or 'no'?: yes
enter the table to print: 5
5 X 1 = 5

5 X 2 = 10

5 X 3 = 15

5 X 4 = 20

5 X 5 = 25

5 X 6 = 30

5 X 7 = 35

5 X 8 = 40

5 X 9 = 45

5 X 10 = 50

Do you want to continue printing the table, press 'yes' or 'no'?: no
```

Python continue Statement

The continue statement in python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and continues with the next iteration.

The syntax of Python continue statement is given below.

```
#loop statements
continue
#the code to be skipped
```

Example 1

```
i = 0;
while i!=10:
    print("%d" % i)
    continue
    i=i+1
```

Output:

```
infinite loop
```

Example 2

```
i=1; #initializing a local variable
#starting a loop from 1 to 10
for i in range(1,11):
    if i==5:
        continue
    print("%d"%i)
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

Pass is also used where the code will be written somewhere but not yet written in the program file.

The syntax of the pass statement is given below.

```
# statements
pass
# statements
```

Example

```
list = [1,2,3,4,5]
flag = 0
for i in list:
    print("Current element:",i,end=" ")
    if i==3:
        pass
        print("\nWe are inside pass block\n")
        flag = 1
    if flag==1:
        print("\nCame out of pass\n")
        flag=0
```

Output:

```
Current element: 1 Current element: 2 Current element: 3
We are inside pass block

Came out of pass
```

Current element: 4 Current element: 5

Example:

```
# A Program to illustrate the usage of 'pass statement'
# A program to retrieve only -ve numbers from the list

num = [1,2,3,-4,-5,-6,-7,8,9,10]
for i in num:
    if (i>0):
        pass
    else:
        print(i)
```

Output

```
-4
-5
-6
-7
```

Note:

Iterable is an object, which one can iterate over. It generates an Iterator when passed to iter() method. **Iterator** is an object, which is used to iterate over an iterable object using __next_() method. Iterators have__next__() method, which returns the next item of the object.

Note that every iterator is also an iterable, but not every iterable is an iterator. For example, a list is iterable but a list is not an iterator. An iterator can be created from an iterable by using the function iter(). To make this possible, the class of an object needs either a method__iter__, which returns an iterator, or a__getitem__method with sequential indexes starting with 0.