UNIT – 1-2

1. Understanding Python Variables – Multiple Variable Declarations

Python is not “statically typed”. We do not need to declare variables before using them or declare their type. **A variable is a name given to a memory location. It is the basic unit of storage in a program. Variables are containers for storing data values.** A variable is created the moment we first assign a value to it.

Rules for creating variables in Python:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (name, Name and NAME are three different variables).
- The reserved words (keywords) cannot be used naming the variable**.**

Declaring a variable:

An integer assignment

```
age = 45
```

A floating point

```
salary = 1456.8
```

A string

```
name = "John"
print(age)
print(salary)
print(name)
```

Output:

```
45
1456.8
John
```

Re-declare/re assign the Variable:

We can re-declare the python variable once we have declared the variable already.

```
# declaring the var
number = 100

# display
print("Before declare: ", number)

# re-declare the var
number = 120.3

print("After re-declare:", number)
```

Output:

```
Before declare: 100
After re-declare: 120.3
```

Casting

If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3) # z will be 3.0
```

Get the Type

You can get the data type of a variable with the type ( ) function.

```
x = 5
y = "cyber_security"
print(type(x))
print(type(y))
```

Output:

```
<class'int'>
<class 'str'>
```

Single or Double Quotes

String variables can be declared either by using single or double quotes:

```
x = "CSE"
print(x)
# is the same as
x = 'CSE'
print(x)
```

Output: when we print variable x

```
CSE
CSE
```

Case-Sensitive

Variable names are case-sensitive.

```
a = 4
A = "HELLO"
#A will not overwrite a

print(a)
print(A)
```

Output: when we print variable a and A

```
4
HELLO
```

Valid variables

```
myvar = "hello"
my_var = "hello"
_my_var = "hello"
myVar = "hello"
MYVAR = "hello"
myvar2 = "hello"
```

Invalid variables

```
2myvar = "hello"
my-var = "hello"
my var = "hello"
```

Multi Words Variable Names

Variable names with more than one word can be difficult to read. There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "lucky"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "lucky"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "lucky"
```

Assigning a single value to multiple variables:

Python allows assigning a single value to several variables simultaneously with “=” operators.

For example:

```
a = b = c = 10

print(a)
print(b)
print(c)
```

Output:

10

10

10

Assigning different values to multiple variables:

Python allows adding different values in a single line with “,”operators.

```
a, b, c = 1, 20.2, "hello"

print(a)
print(b)
print(c)
```

Output:
1
20.2
hello

Can we use the same name for different types?

If we use the same name, the variable starts referring to a new value and type.

```
a = 10
a = "hello"

print(a)
```

Output:
hello

How does + operator work with variables?

```
a = 10
b = 20
print(a+b)

a =  "hello"
b = "python"
print(a+b)
```

Output:
30
hellopython

Can we use + for different types also?

No using for different types would produce error.

```
a = 10
b = "hello"
```

```
print(a+b)
```

Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Output:

```
apple
banana
cherry
```

OPERATORS IN PYTHON

ARITHMATIC OPERATORS

Operators are special symbols in Python that carry out some computation. The value that the operator operates on is called the operand. **Python Operators** in general are used to perform operations on values and variables.

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

There are 7 arithmetic operators in Python:

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulus
6. Exponentiation
7. Floor division

Example program on Arithmetic Operators

```
a = 9
b = 4
add = a + b
sub = a - b
mul = a * b
div1 = a / b
div2 = a // b
mod = a % b
p = a ** b

# print results

print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
print(p)
```

Output

```
13
5
36
2.25
2
1
6561
```

NOTE: Ceiling division returns the closest integer greater than or equal to the current answer or quotient. In Python, we have an operator // for floor division, but no such operator existsfor the ceiling division

But it can be implemented in python using functions as follows:

```
from math import floor
from math import ceil

print(floor(10/6))
print(ceil(10/6))
```

Output

```
1
2
```

ASSIGNMENT OPERATORS

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

OPERATOR	EXAMPLE	EQUIVALENT TO
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Example program on Assignment Operators

```
a = 5
a+=5
print(a)
a-=5
print(a)
a*=5
print(a)
a/=5
print(a)
a//=5
print(a)
a%=5
print(a)
a**=5
print(a)
b=10
b&=5
print(b)
b|=5
print(b)
c=5
c^=5
print(c)
d=5
d<<=1
print(d)
d>>=1
print(d)
```

Output

```
10
5
25
5.0
1.0
1.0
1.0
0
5
0
10
5
```

RELATIONAL OPERATORS

Relational operators are used for comparing the values. It either returns True or False according to the condition. These operators are also known as **Comparison Operators**.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

```
x = 10
y = 12

# Output: x > y is False
print('x > y is',x>y)

# Output: x < y is True
print('x < y is',x<y)

# Output: x == y is False
print('x == y is',x==y)

# Output: x != y is True
print('x != y is',x!=y)

# Output: x >= y is False
print('x >= y is',x>=y)

# Output: x <= y is True
print('x <= y is',x<=y)
```

Output

```
x > y is False
x < y is True
```

```
x == y is False
x != y is True
x >= y is False
x <= y is True
```

LOGICAL OPERATORS

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

```
x = True
y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)
```

Output

```
x and y is False
x or y is True
not x is False
```

BITWISE OPERATORS

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

PROGRAM

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print ("The Value of c after AND is ", c)

c = a | b;        # 61 = 0011 1101
print ("The Value of c after OR is ", c)

c = a ^ b;        # 49 = 0011 0001
print ("The Value of c after XOR is ", c)

c = ~a;           # -61 = 1100 0011
print ("The Value of c after NEGATION is ", c)

c = a << 1;
print ("The Value of c after LEFT SHIFT is ", c)

c = a >> 1;
print ("The Value of c after RIGHT SHIFT is ", c)
```

OUTPUT:

```
The Value of c after AND is 12
The Value of c after OR is 61
The Value of c after XOR is  49
The Value of c after NEGATION is  -61
The Value of c after LEFT SHIFT is 120
The Value of c after RIGHT SHIFT is  30
```

BOOLEAN OPERATORS

Python boolean operators are similar to python bitwise operators in the sense that instead of bits here, we consider complete boolean expressions. In Python boolean operatorcalculations, we make use of the boolean expressions and decide the outcome of the expressions according to the operator.

- *AND boolean operator in Python*
- *NOT boolean operator in Python*
- *OR boolean operator in Python*

A boolean expression is an expression that yields just the two outcomes: ***true or false***. When we work with multiple boolean expressions or perform some action on them, we make use of the boolean operators. Since the boolean expression reveals true or false, the operations on these expressions also result in either ***"true"*** or ***"false".***

AND Boolean Operator in Python

The ***AND boolean operator*** is similar to the bitwise ***AND operator*** where the operator analyzes the expressions written on both sides and returns the output.

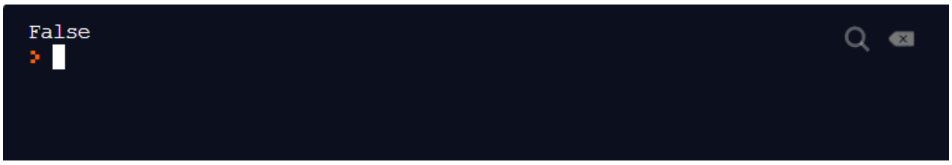
- *True and True = True*
- *True and False = False*
- *False and True = False*
- *False and False = False*

In python, you can directly use the word ***"and "*** instead of "&&" to denote the ***"and "*** boolean operator

Example Program

```
a = 30

b = 45

if(a > 30 and b == 45):

   print("True")

else:

   print("False")
```

Executing the above code, we get:



```
False
>
```

OR Boolean Operator in Python

The ***OR operator*** is similar to the ***OR bitwise operator***. In the bitwise OR, we were focussing on either of the bit being 1. Here, we take into account if either of the expression is true or not. If at least one expression is true, consequently, the result is true.

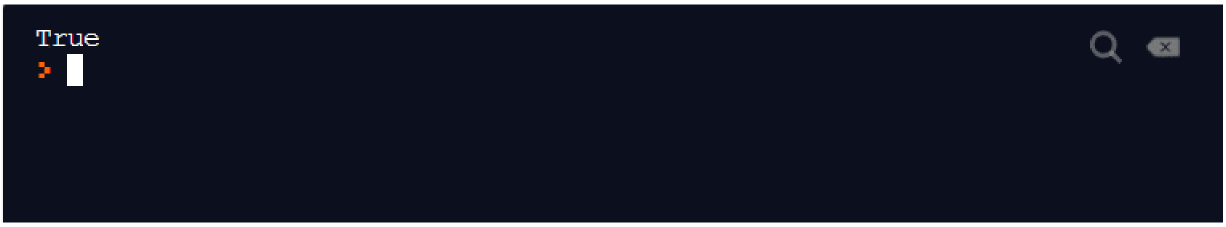
- *True or True = True*
- *True or False = True*
- *False or True = True*
- *False or False = False*

In python, you can use || as well as the word **"or "** directly into the code. Let's execute the following code to check the output:

Example Program

```
a = 25
b = 30
if(a > 30 or b < 45):
   print("True")
else:
   print("False")
```

Consequently, run the above code to see the result:



```
True
>
```

NOT Boolean Operator in Python

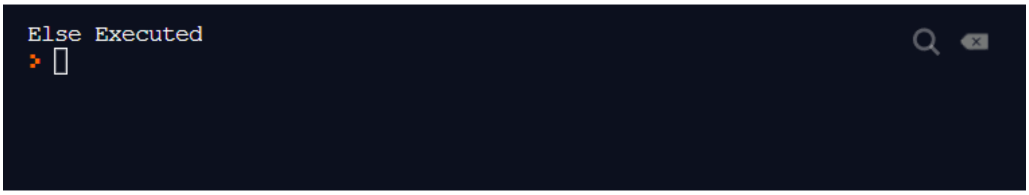
The ***NOT operator*** reverses the result of the boolean expression that follows the operator. It is important to note that the NOT operator will only reverse the final result of the expression that ***immediately follows.*** Moreover, the NOT operator is denoted by the keyword ***"not".***

- *not(True) = False*
- *not(False) = True*

Example Program

```
a = 2

b = 2

if(not(a == b)):
  print("If Executed")

else:
  print("Else Executed")
```

Press Run to see the output:



```
Else Executed
```

MEMBERSHIP OPERATOR

Membership operators are operators used to validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

- **in operator:** The ‘in’ operator is used to check if a value exists in a sequence or not. Evaluate to true if it finds a variable in the specified sequence and false otherwise.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

#MEMBER SHIP – sample code in python executed from jupyter notebook

x = ["apple", "banana"]

print("banana" in x)

# returns True because a sequence with the value "banana" is in the list

Output: True

```
x = ["apple", "banana"]

print("banana" not in x)

# returns False because a sequence with the value "banana" is in the list
```

Output: False

PYTHON IDENTITY OPERATORS

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

- **‘is’ operator –** Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

```
x = 5
if (type(x) is int):
    print("true")
else:
    print("false")
```

output: true

- **‘is not’ operator –** Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

```
x = 5.2
if (type(x) is not int):
    print("true")
else:
    print("false")
```

output: true

OPERATOR PRECEDENCE

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator. For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7. Operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator	Description	Associativity
()	**Parentheses**	**Left to right**
******	**Exponentiation**	**Right to left**
~, +,-	**Complement, unary plus (positive), and minus (negative)**	**Right to left**
***, /, %, //**	**Multiply, divide, modulo, and floor division**	**Left to right**
+, -	**Addition and subtraction**	**Left to right**
>>, <<	**Right and Left bitwise shift**	**Left to right**
&	**Bitwise ‘AND’**	**Left to right**
^, |	**Bitwise exclusive ‘OR’ and regular ‘OR’**	**Left to right**
<=, <, >, >=	**Comparison operators**	**Left to right**

<>, ==, !=	**Equality operators**	**Left to right**
=, %=, /=, //=, -=, +=, *=, **=	**Assignment operators**	**Right to left**
Is, is not	**Identity operators**	**Left to right**
In, not in	**Membership operators**	**Left to right**
Not, or, and	**Logical operators**	**Right to left, left to right, left to right**

```
a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ",  e

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print "Value of ((a + b) * c) / d is ",  e

e = (a + b) * (c / d);    # (30) * (15/5)
print "Value of (a + b) * (c / d) is ",  e

e = a + (b * c) / d;      #  20 + (150/5)
print "Value of a + (b * c) / d is ",  e
```

When you execute the above program, it produces the following result −

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

OUTPUT STATEMENTS

Python print() function prints the message to the screen or any other standard output device.

```
print('This sentence is output to the screen')
```

Output:

```
This sentence is output to the screen
```

```
a = 5
print('The value of a is', a)
```

Output:

```
The value of a is 5
```

The actual syntax of the print() function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, objects are the value(s) to be printed.

The **sep** separator is used between the values. It defaults into a space character.

After all values are printed, **end** is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen). Here is an example to illustrate this.

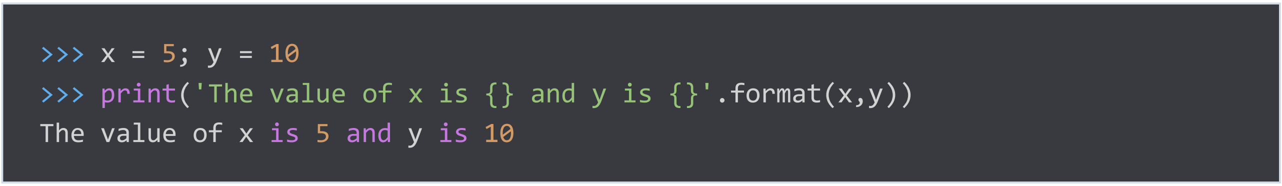
```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

Output:

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

1. Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

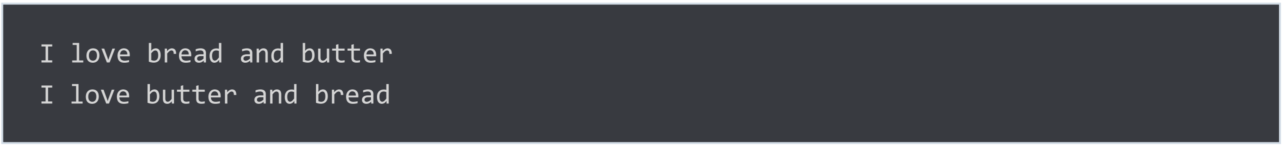


```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

2. Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

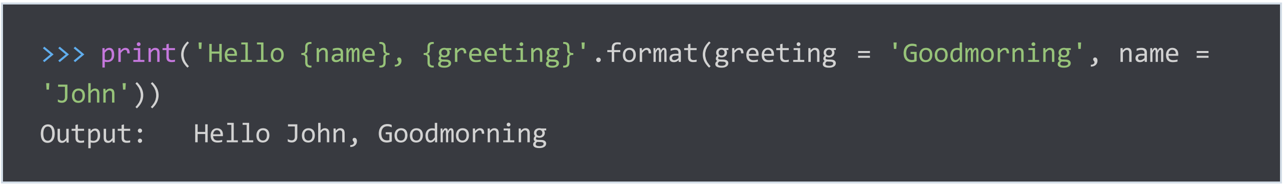
```
print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))
```

Output:



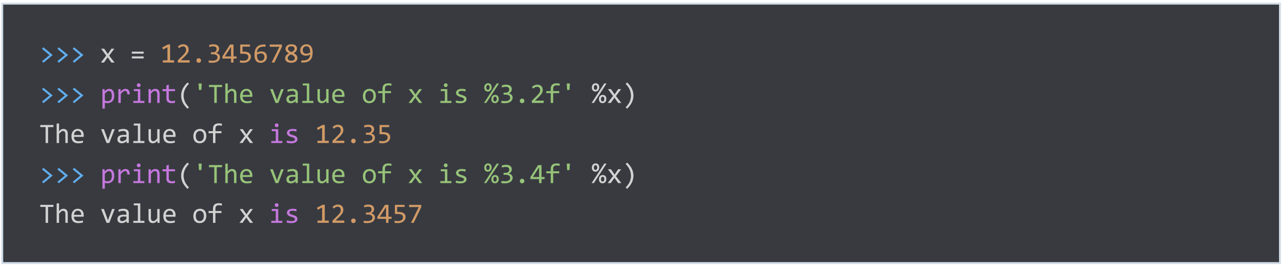
```
I love bread and butter
I love butter and bread
```

3. We can even use keyword arguments to format the string.



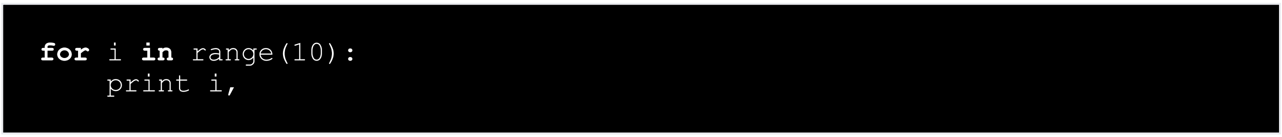
```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name =
'John'))
Output:   Hello John, Goodmorning
```

4. Float numbers printing



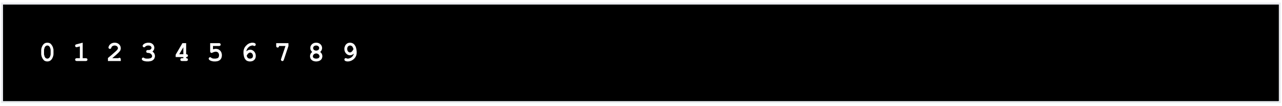
```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

5. Printing output in same line without newline
   Objects can be printed on the same line without needing to be on the same line if one puts a comma at the end of a print statement:



```
for i in range(10):
    print i,
```

This will output the following:



```
0 1 2 3 4 5 6 7 8 9
```

6. To end the printed line with a newline, add a print statement without any objects.

```
for i in range(10):
    print i,
print
for i in range(10,20):
    print i,
```

This will output the following:

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
```

7. **end= " " statement**

   The end keyword is used to specify the content that is to be printed at the end of the execution of the print() function. By default, it is set to "\n", which leads to the change of line after the execution of print() statement.

   Example: Python print() without new line.

   #This line will automatically add a new line before the next print statement
   print ("hi")
   print("Welcome ")

   output: hi
   Welcome

   # This print() function ends with "**" as set in the end argument.
   print ("hi", end= "")**
   print("Welcome ")

   output: hi ** Welcome

8. **flush Argument**

   The I/Os in python are generally buffered, meaning they are used in chunks. This is where flush comes in as it helps users to decide if they need the written content to be buffered or not. By default, it is set to false. If it is set to true, the output will be written as a sequence of characters one after the other. This process is slow simply because it is easier to write in chunks rather than writing one character at a time.

9. **Separator**

   The print() function can accept any number of positional arguments. These arguments can be separated from each other using a "," separator. These are primarily used for formatting multiple statements in a single print() function. python print() function by default ends with a newline.

```
b = "python"
print("hi", b , "program")
```

output: hi python program

```
# using end argument

print("Python", end='@')
print("program")
```

output: python@program

10. # printing a element in same line

```
for i in range(4):
    print(a[i], end =" ")
```

output: `0 1 2 3`

11. **# Print without newline in Python 3.x without using for loop**

```
l=[1,2,3,4,5,6]
# using * symbol prints the list elements in a single line
print(*l)
```

12. **Python sep parameter in print()**

The separator between the arguments to print() function in Python is space by default (softspace feature) , which can be modified and can be made to any character, integer or string as per our choice. The ‘sep’ parameter is used to achieve the same, it is found only in python 3.x or later. It is also used for formatting the output strings.

```
#code for disabling the softspace feature
print('G','F','G', sep='')

#for formatting a date
print('09','12','2016', sep='-')

#another example
print('hi','PP', sep='@')
```

Output:

```
GFG

09-12-2016

hi@PP
```

PYTHON INPUT

The value of variables was defined or hard coded into the source code. To allow flexibility, we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is:

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

Python Import

When our program grows bigger, it is a good idea to break it into different modules.

A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

For example, we can import the math module by typing the following line:

```
import math
```

We can use the module in the following ways:

```
import math
print(math.pi)
```

Run Code

Output

```
3.141592653589793
```

Now all the definitions inside `math` module are available in our scope. We can also import some specific attributes and functions only, using the `from` keyword. For example:

```
>>> from math import pi
>>> pi
3.141592653589793
```

COMMAND LINE ARGUMENTS

The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:

- Using sys.argv
- Using getopt module
- Using argparse module

Using sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

One such variable is sys.argv which is a simple list structure. It’s main purpose are:

It is a list of command line arguments.

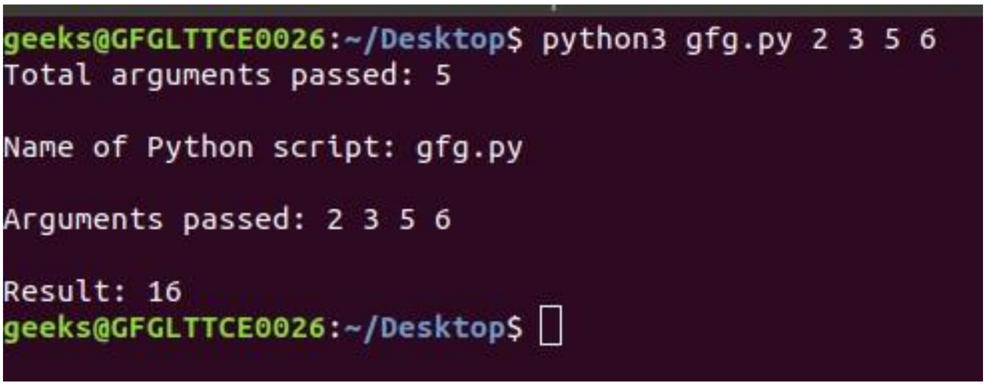
len(sys.argv) provides the number of command line arguments.

sys.argv[0] is the name of the current Python script.

Example: Let’s suppose there is a Python script for adding two numbers and the numbers are passed as command-line arguments.

```
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)
```

```
# Arguments passed
print("\nName of Python script:", sys.argv[0])

print("\nArguments passed:", end = " ")
for i in range(1, n):
    print(sys.argv[i], end = " ")

# Addition of numbers
Sum = 0
for i in range(1, n):
    Sum += int(sys.argv[i])
print("\n\nResult:", Sum)
```



```
geeks@GFGLTTCE0026:~/Desktop$ python3 gfg.py 2 3 5 6
Total arguments passed: 5

Name of Python script: gfg.py

Arguments passed: 2 3 5 6

Result: 16
geeks@GFGLTTCE0026:~/Desktop$ 
```

Using getopt module

Python **getopt module** is similar to the getopt() function of C. Unlike sys module getopt module extends the separation of the input string by parameter validation. It allows both short, and long options including a value assignment. However, this module requires the use of the sys module to process input data properly. To use getopt module, it is required to remove the first element from the list of command-line arguments.

Syntax: getopt.getopt(args, options, [long_options])

Parameters:

args: List of arguments to be passed.

options: String of option letters that the script want to recognize. Options that require an argument should be followed by a colon (:).

long_options: List of string with the name of long options. Options that require arguments should be followed by an equal sign (=).

Return Type: Returns value consisting of two elements: the first is a list of (option, value) pairs. The second is the list of program arguments left after the option list was stripped.

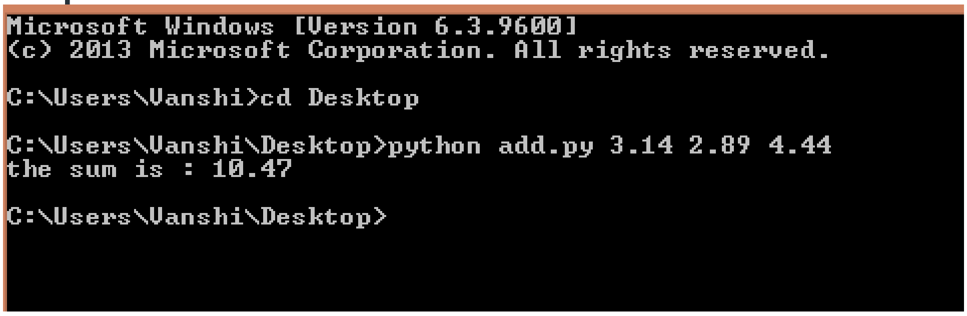
Using argparse module

Using argparse module is a better option than the above two options as it provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc.

Note: As a default optional argument, it includes -h, along with its long version –help.

```
# Python program to demonstrate using sys.argv

import sys
add = 0.0

# Getting the length of command
# line arguments
n = len(sys.argv)

for i in range(1, n):
    add += float(sys.argv[i])

print ("the sum is :", add)
```

Output:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Vanshi>cd Desktop

C:\Users\Vanshi\Desktop>python add.py 3.14 2.89 4.44
the sum is : 10.47

C:\Users\Vanshi\Desktop>
```

INDENTATION

In Python, the leading whitespace (spaces and tabs) before any statement is referred to as indentation. Other languages such as C++, C, and Java use indentation for readability purposes, but indentation is an essential and mandatory feature in Python that must befollowed while writing code; otherwise, the Python interpreter will throw an Indentation Error. Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code. A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation to highlight the blocks of code. Whitespace is used for indentation in Python.All statements with the same distance to the right belong to the same block of code. If a blockhas to be more deeply nested, it is simply indented further to the right.

Ex:

```
name = "python"
if name=="python":
   print("Welcome dear")
else:
   print("Its not python")
print("Enjoy!")
```

Output: Welcome dear
Enjoy!

In the first line, the variable name is assigned to python. This is the first statement. Now the control reaches statement 2, which is the if statement. This if statement returns true, so the control reaches the body of the if statement. The indentation of the body of the if statement is one step more than the if statement. After the execution occurs, the else part is skipped, and the control reaches the last statement, which is print("Enjoy!").

Avoiding indentation errors

1. If you make any mistake in indentation, then Python throws an "Indentation Error: expected an indented block" error.

```
if 1==2:
print("equal")
>>     print("equal")
   ^
IndentationError: expected an indented block.
```

2. Indented code should have the same number of whitespaces for each block of code. Otherwise, "IndentationError: unexpected indent" will be thrown by Python.

```
if( 1 == 2):
 print("Line 1")
     print("Line 2")
```

The correct indentation should be:

```
if( 1 == 2):
    print("Line 1")
    print("Line 2")
```

3. The indentation on the first line should be 0. Otherwise, "IndentationError: unexpected indent" will be thrown by Python.

Indentation rules in Python

- Python's default indentation spaces are four spaces. On the other hand, the amount of space is entirely up to the user. However, a minimum of one space is required to indent a statement.

- Indentation cannot be used on the first line of Python code.
- In order to define blocks of statements in Python, indentation is required.
- In a code block, the number of spaces must be equal.
- In Python, whitespaces are preferred over tabs for indentation. Also, the indentation should be done with either whitespace or tabs; combining tabs and whitespaces in indentation might result in incorrect indentation issues.

PYTHON CONDITIONAL STATEMENTS

What are Conditional Statements in Python?

Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

if statement:

Python if Statement is used for decision-making operations. It contains a body of codewhich runs only when the condition given in the **if statement is true**. An "if statement" is written by using the “ if “ keyword. ‘If’ statement in Python is an eminent conditional loop statement that can be described as an entry-level conditional loop, where the condition is defined initially before executing the portion of the code. “if” statement works basically on the Boolean conditions “True” & “False”. A given block of code passes when a given “if” condition is True and does not pass or is executed when a given condition is false.

“if” condition can also be used on simple mathematical conditions such as Equal (=), Not Equal (! =), Less than (<), Less than or equal to (<=), Greater than (>) Greater than or equalto (>=).

How If Statement Works?

The “if” statement is primarily used in controlling the direction of our program. It is used in skipping the execution of certain results that we don’t intend to execute. The basic structure of an “if” statement in python is typing the word “if” (lower case) followed by the condition with a colon at the end of the “if” statement and then a print statement regarding printing our desired output.

Python is case sensitive, too, so “if” should be in lower case.

Syntax:

```
if  <condition>:
   Print <statement>
```

Python is sensitive to indentation; after the “if” condition, the next line of code is spaced four spaces apart from the statement’s start. Any set of instructions or conditions that belongs to the same block of code should be indented. Indentation is unique to the python programming language. Python strictly adheres to indentation; it is developed that way to make the lines of code neat and easily readable.

Example #1

Code:

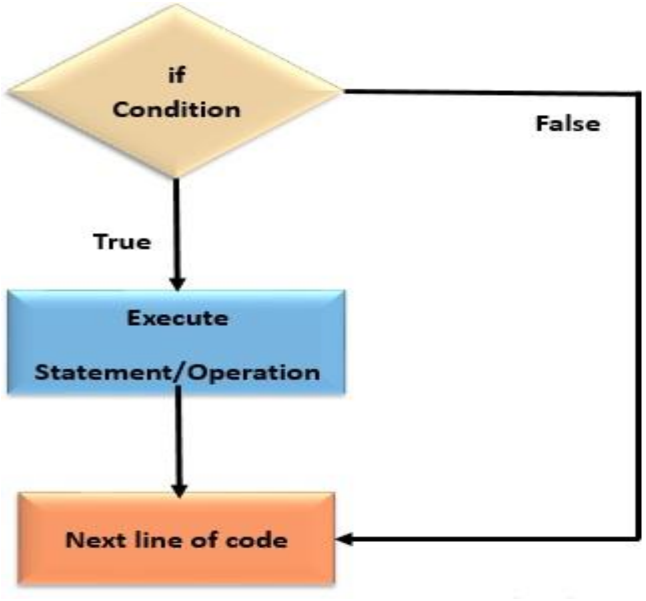
```
if 'cat' in ['dog', 'cat', 'horse', 'penguin']:
   print('Cat exists')
   print('Cat is my favorite pet')
```

Output:

```
Cat exists
Cat is my favorite pet
```

In example 1, the “if” condition is true since the cat is present inside the list; hence both the print statement is executed and printed. The whole of example 1 is a single block of code.

Flow chart:



* We can also use multiple "if" conditions inside the same block provided the statements follow indentation.

Example:

```
if 'horse' in ('dog', 'cat', 'horse', 'penguin'):
       print('horse exists')
        if 'cat' in ('dog', 'cat', 'sheep'):
               print('cat exist')
                if 'sheep' not in ('dog', 'cat', 'horse', 'penguin'):
                        print('sheep does not exist')
```

Output:

```
horse exists
cat exist
sheep does not exist
```

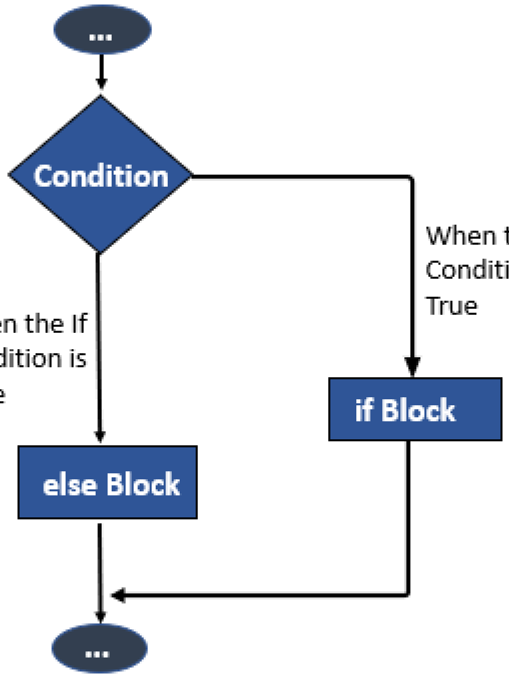
if … else statement:

This statement is similar to the If statement, but it adds another block of code that is executed when the conditions are not met. **Python If-Else is an extension of Python If statement** where we have an else block that executes when the condition is false. The syntax of Python if-else statement is given below. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

Syntax:

```
if (condition):

    # Executes this block if

    # condition is true

else:

    # Executes this block if

    # condition is false
```

Flow chart:



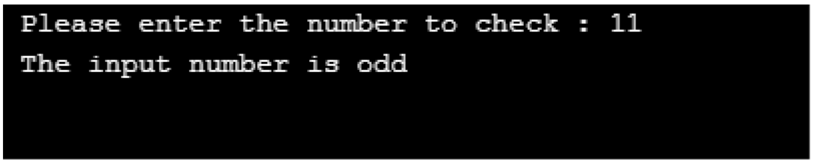
As you can see in the flowchart above, the condition in an if-else statement creates two paths for the program to go on. If the condition is not met, the code below it is not executed, and the program executes the Else block statement. On the other hand, when the "if" condition is met, only then if a block of code is executed and the program then jumps below, exiting the If else statement.

Example 1

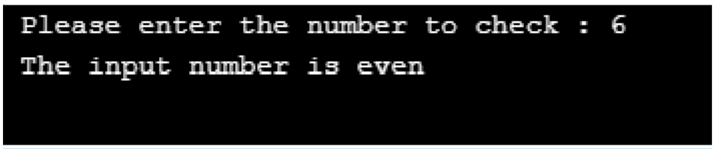
A Python Program to check if the input number is even or odd.

```
number = int(input(" Please enter the number to check : "))
if number %2 == 0:
   print(" The input number is even ")
else:
   print(" The input number is odd ")
```

Output:



```
Please enter the number to check : 11
The input number is odd
```



```
Please enter the number to check : 6
The input number is even
```

Example 2   python program to illustrate If else statement

```
i = 20
if (i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
print("i'm out of block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm out of block
```

if….elif…..else statement:

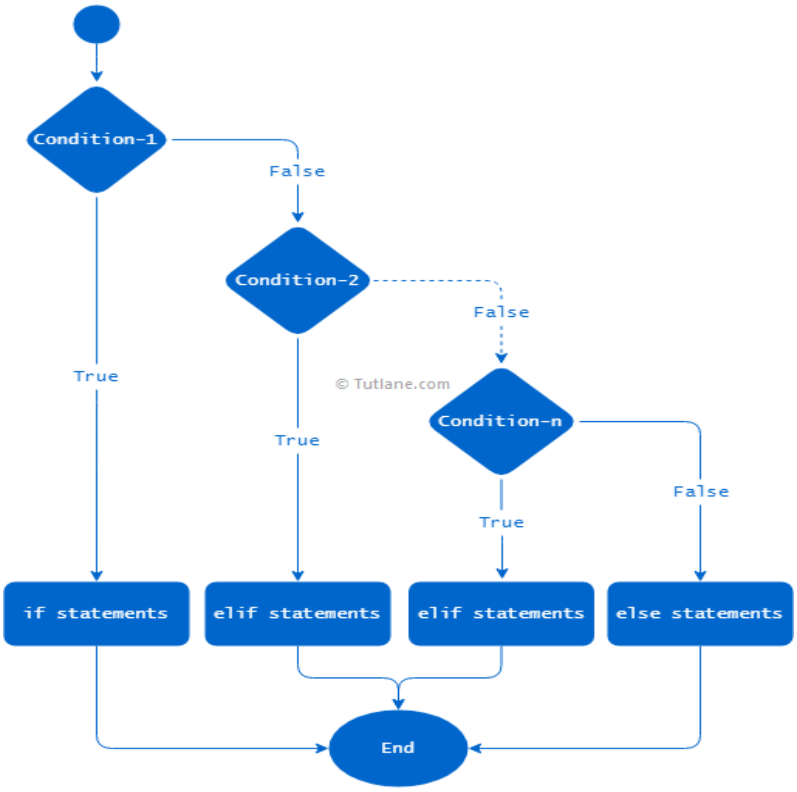
if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition):
      statement
elif (condition):
      statement
.
.
else:
      statement
```

Flow chart:



Examples:

```
price = 100
if price > 100:
    print("price is greater than 100")
```

```
elif price == 100:
    print("price is 100")
elif price < 100:
    print("price is less than 100")
else:
    print(“Am in else block”)
output:
  price is 100
```

In the above example, the elif conditions are applied after the if condition. Python will evalute the if condition and if it evaluates to False then it will evalute the elif blocks and execute the elif block whose expression evaluates to True. If multiple elif conditions become True, then the first elif block will be executed.

All the if, elif, and else conditions must start from the same indentation level, otherwise it will raise the IndentationError.

NESTED IF STATMENTS

A **nested if/else statement** is an if/else statement that is nested (meaning, inside) another if statement or if/else statement. With those statements we evaluate true/false conditions and make our program respond appropriately.

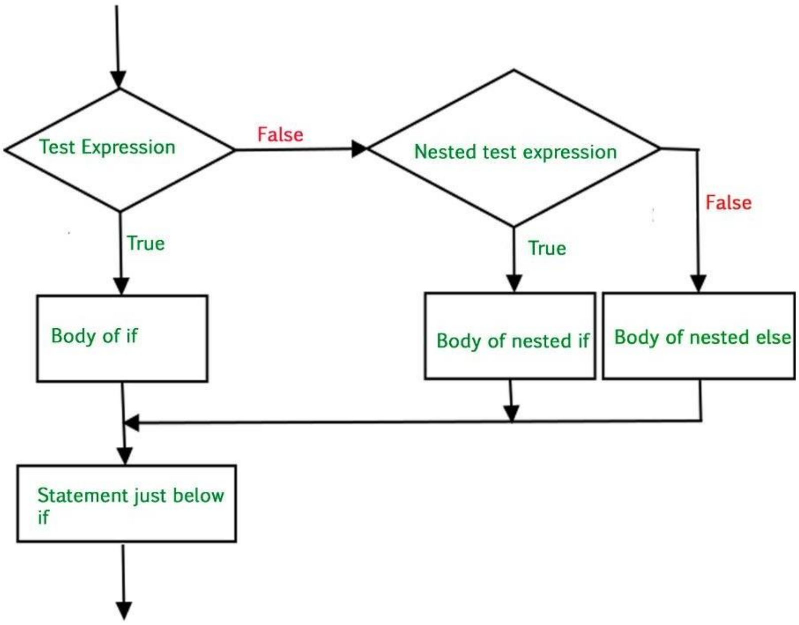
Nested-if Statement

We can have an if…elif…else statement inside another if…elif…else statement. This iscalled nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so it must be avoided if we can.

Syntax:

```
if (condition1):
      # Executes when condition1 is true
       if (condition2):
              # Executes when condition2 is true
       # if Block is end here
# if Block is end here
```

Flow chart: * else may present or not depends on our requirement



Example: compare age with a nested if statement

When we use a nested if statement, we often make additional comparisons following aregular if statement. The program below is an example of that. We first check if the person is an adult. Then we make additional comparisons to see if he or she graduated or gotten a drivers license:

```
age = 19
isGraduated = False
hasLicense = True

# Look if person is 18 years or older
if age >= 18:
    print("You're 18 or older. Welcome to adulthood!")
    if isGraduated:
        print('Congratulations with your graduation!')
    if hasLicense:
        print('Happy driving!')
```

Here’s the output that the program generates:

```
You're 18 or older. Welcome to adulthood!

Happy driving!
```

Python Nested if …elif….else

The if-elif statement is shortcut of if..else chain. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

Syntax

The syntax of the nested *if...elif...else* construct may be −

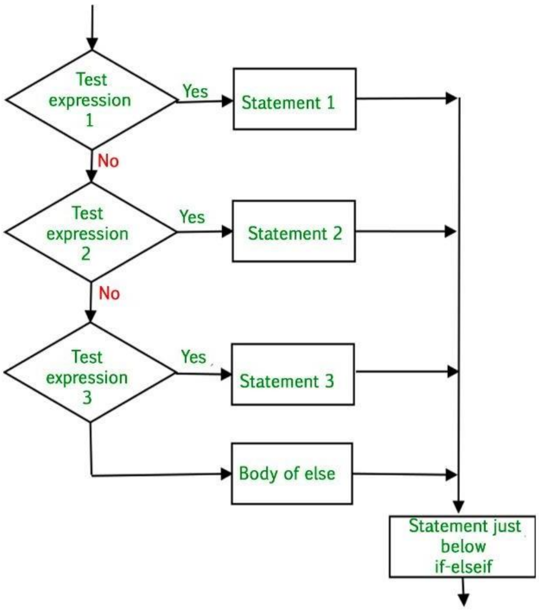
```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
     statement(s)
   elif expression4:
     statement(s)
   else:
      statement(s)
else:
   statement(s)
```

Example

```
#!/usr/bin/python

var = 100
if var < 200:
   print "Expression value is less than 200"
   if var == 150:
      print "Which is 150"
   elif var == 100:
      print "Which is 100"
   elif var == 50:
      print "Which is 50"
   elif var < 50:
      print "Expression value is less than 50"
else:
   print "Could not find true expression"

print "Good bye!"
```

When the above code is executed, it produces following result −

```
Expression value is less than 200
Which is 100
Good bye!
```



Python – Nested If Else

Nested If Else in Python is an extension for the idea of if-else block. Any statement in the if or else blocks could be another if or if-else block. This makes nesting possible, and write an if or if-else inside if or else blocks.

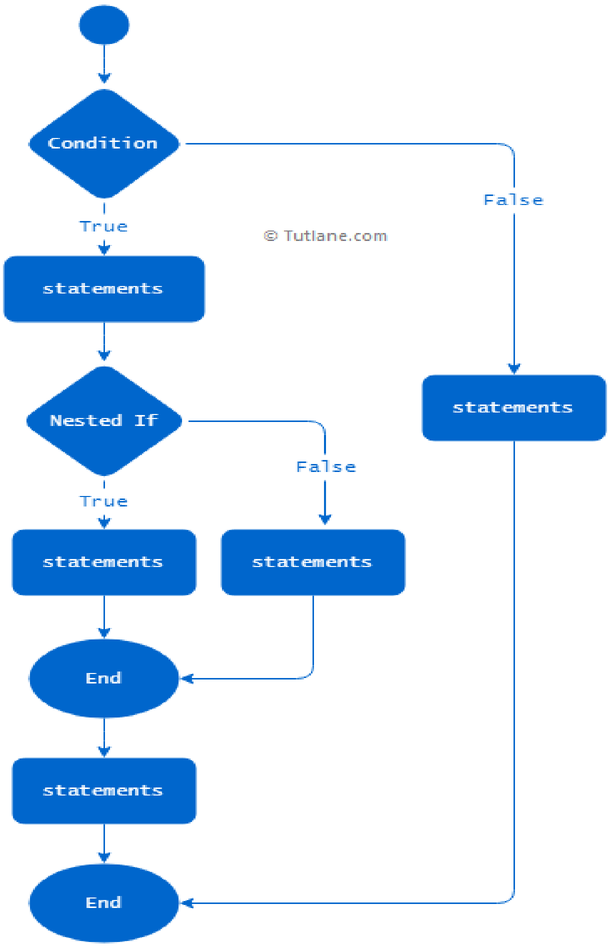
Syntax

The following is the syntax of two level nested if-else statement.

```
if <condition_1>:
        statement(s)
        if <condition_2>:
               statement(s)
        else:
               statement(s)
else:
    statement(s)
```

Nesting is not limited to two levels. You can write another if-else inside the inner if or else block. Then it would be three level nested if-else statement.

Flow chart:



Example:

```
n = 15

if n%2 == 0:
    print(f'{n} is even.')
    if (n%10 == 0):
        print(f'{n} is divisible by 10.')
    else:
        print(f'{n} is not divisible by 10.')
else:
    print(f'{n} is odd.')
    if (n%5 == 0):
        print(f'{n} is divisible by 5.')
    else:
        print(f'{n} is not divisible by 5.')
```

Output:

```
15 is odd.
15 is divisible by 5.
```

One-Line if Statements – Short hand if:

It is customary to write `if <expr>` on one line and `<statement>` indented on the following line like this:

```
if <expr>:
    <statement>
```

But it is permissible to write an entire `if` statement on one line. The following is functionally equivalent to the example above:

```
if <expr>: <statement>
```

There can even be more than one `<statement>` on the same line, separated by semicolons:

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

If <expr> is true, execute all of <statement_1> ... <statement_n>. Otherwise, don’t execute any of them. The semicolon separating the <statements> has higher precedence than the colon following <expr>—in computer lingo, the semicolon is said to bind more tightly than the colon. Thus, the <statements> are treated as a suite, and either all of them are executed, ornone of them are:

```
>>> x = 2
>>> if x == 1: print('foo'); print('bar'); print('baz')
... elif x == 2: print('qux'); print('quux')
... else: print('corge'); print('grault')
```

Output:

```
qux
quux
```

Python supports one additional decision-making entity called a conditional expression. (It is also referred to as a conditional operator or ternary operator in various places in the Python documentation.)

In its simplest form, the syntax of the conditional expression IF ELSE is as follows:

```
<expr1> if <conditional_expr> else <expr2>
```

This is different from the if statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example, <conditional_expr> is evaluated first. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

Example:

```
>>> age = 12
>>> s = 'minor' if age < 21 else 'adult'
>>> s
```

Output: minor

WHILE LOOP

Loops are used in programming to repeat a specific block of code.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is −

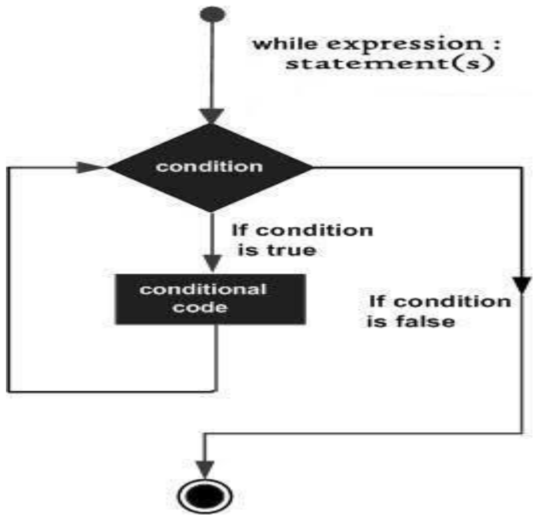
```
while expression:
        statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow chart



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop).

Finally, the result is displayed.

FOR LOOP

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal. For loops are used for sequentialtraversal. For example: traversing a list or string or array etc. In Python, there is no C stylefor loop, i.e., for (i=0; i<n; i++). There is “for in” loop which is similar to for each loop in other languages.

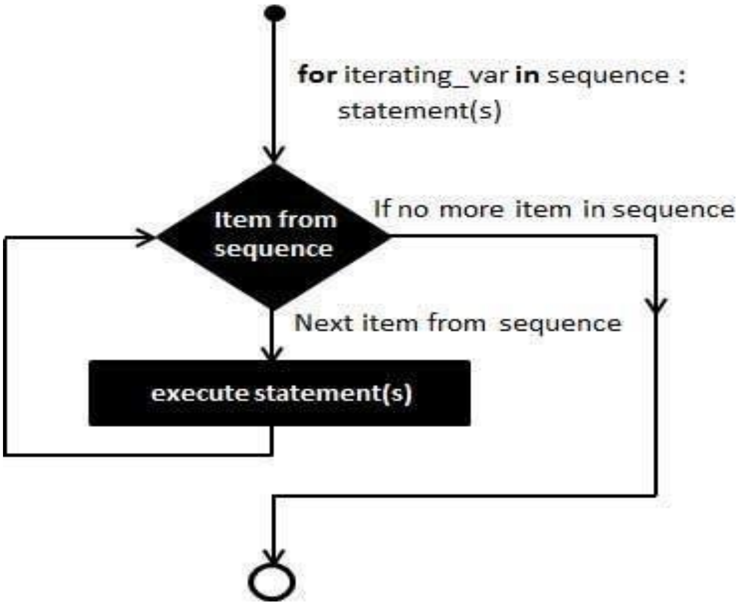
Syntax:

for iterating_var in sequence:

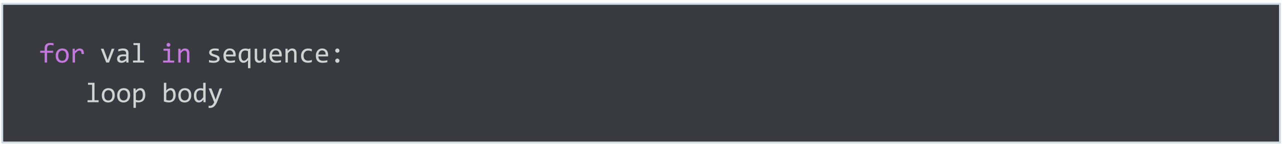
statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow chart:



Or we can also write as follows:



```
for val in sequence:
    loop body
```

Flowchart of for Loop

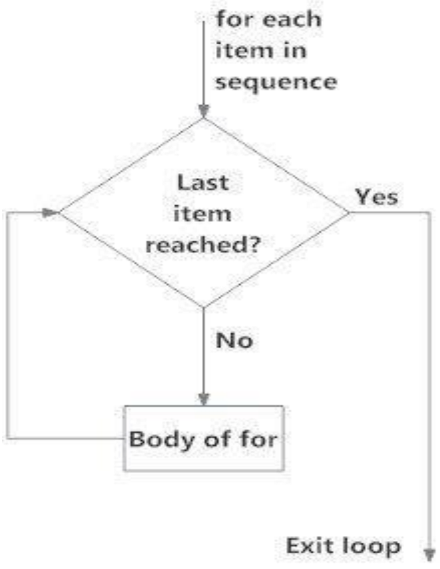


Fig: operation of for loop

Example:

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

When we run the program, the output will be:

```
The sum is 48
```

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and getitem_operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

Output

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

Output

```
I like pop
I like rock
I like jazz
```

Another example:

```
# program to display student's marks from record
student_name = 'Soyuj'

marks = {'James': 90, 'Jules': 55, 'Arthur': 77}

for student in marks:
    if student == student_name:
        print(marks[student])
        break
else:
    print('No entry with that name found.')
```

Output

```
No entry with that name found.
```

Example

```
for letter in 'Python':     # First Example
   print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit

print "Good bye!"
```

When the above code is executed, it produces the following result −

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Example:

```
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

When the above code is executed, it produces the following result −

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Example:

```
# Python program to illustrate
# Iterating by index
 list = ["geeks",  "for",  "geeks"]
  for index  in range(len(list)):
      print list[index]
```

Output:

```
geeks

for

geeks
```

INFINITE LOOP

An Infinite Loop in Python is **a continuous repetitive conditional loop that gets executed until an external factor interferes in the execution flow**, like insufficient CPU memory, a failed feature/ error code that stopped the execution or a new feature in the other legacy systems that needs code integration. A few types of Infinite Loop in Python include the Whilestatement, the If statement, the Continue statement, and the Break statement.

When are Infinite Loops Necessary?

An infinite loop may be useful in client/server programming where the server needs to run with continuity so that the client programs may communicate with the server programwhenever the necessity arises. It may also be helpful if a new connection needs to be created. There is the utility of a while loop in a gaming application or an application where we enter some sort of main event loop, which continues to run until the user selects an action to break that infinite loop. Also, if one has to play a game and wishes the game to reset after each session

Example

```
i=0

while i<10:

        print(“I will run for ever”)
```

As there is no code to increment the value of the integer, it will continue to print that until we terminate the program.

Output:

```
I will run for ever
```

```
I will run for ever
I will run for ever
I will run for ever
I will run for ever
```

Output continues to print like this

So, to avoid the unintentional loop, we add the following line to the code.

```
i=0

while i<10:

        print(“I will run for ever”)

        i+=1
```

And then, the definite number of lines gets printed as below in the output.

Output:

```
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
I will run for ever
```

Example

```
While True:

        print(“Am in infinite loop”)
```

Output:

```
Am in infinite loop
Am in infinite loop
Am in infinite loop
Am in infinite loop
…..
```

The above output continues for infinite times as the condition in while is always True. Therefore using break statement we can control the infinite loop or we can press CNTL + C to stop looping.

Kinds of Infinite Loops

There are three kinds of infinite loops:

1. The fake infinite loops
2. The intended infinite loops
3. The unintended infinite loops

The fake infinite loops: Loop with condition at the top

This is a normal while loop without break statements. The condition of the while loop is at the top and the loop terminates when this condition is False otherwise it keeps on looping since condition is always True.

The intended infinite loops: Loop with condition in the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

The unintended infinite loops: Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the do...while loop in C.

NESTED LOOPS

In Python, **a loop inside a loop is known as a nested loop**. The inner or outer loop can be any type, such as a while loop or for loop. For example, the outer for loop can contain a while loop and vice versa. The outer loop can contain more than one inner loop. There is no limitation on the chaining of loops. In the nested loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop. In each iteration of the outer loop inner loop execute all its iteration. **For each iteration of an outer loop the inner loop re-start and completes its execution** before the outer loop can continue to its next iteration.

Python Nested for Loop

In Python, the for loop is used to iterate over a sequence such as a list, string, tuple, other iterable objects such as range.

Syntax of using a nested for loop in Python

```
# outer for loop
for element in sequence
   # inner for loop
   for element in sequence:
       body of inner for loop
   body of outer for loop
```

Example:

Write a nested for loop program to print multiplication table in Python

```
# outer loop
for i in range(1, 11):
    # nested loop
    # to iterate from 1 to 10
    for j in range(1, 11):
        # print multiplication
        print(i * j, end=' ')
    print()
```

Output:

```
1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50

6 12 18 24 30 36 42 48 54 60

7 14 21 28 35 42 49 56 63 70

8 16 24 32 40 48 56 64 72 80

9 18 27 36 45 54 63 72 81 90

10 20 30 40 50 60 70 80 90 100
```

- In this program, the outer for loop is iterate numbers from 1 to 10. The range() return 10 numbers. So total number of iteration of the outer loop is 10.
- In the first iteration of the nested loop, the number is 1. In the next, its 2 and so on till 10.
- Next, For each iteration of the outer loop, the inner loop will execute ten times. The inner loop will also execute ten times because we are printing multiplication table up to ten.
- In each iteration of an inner loop, we calculated the multiplication of two numbers.

Example: Nested Loop to Print Pattern

Pattern:

```
*

* *
* * *

* * * *

* * * * *
```

Program:

```
rows = 5
# outer loop
for i in range(1, rows + 1):
    # inner loop
    for j in range(1, i + 1):
        print("*", end=" ")
    print('')
```

- In this program, the outer loop is the number of rows print.
- The number of rows is five, so the outer loop will execute five times
- Next, the inner loop is the total number of columns in each row.
- For each iteration of the outer loop, the columns count gets incremented by 1
- In the first iteration of the outer loop, the column count is 1, in the next it 2 and so on.
- The inner loop iteration is equal to the count of columns.
- In each iteration of an inner loop, we print star

While loop inside a for loop

It is very common and helpful to use one type of loop inside another. we can put a while loop inside the `for` loop. Assume we wanted to repeat each name from a list five times.

- Here we will iterate the list using an outer for loop
- In each iteration of outer for loop, the inner for loop execute five times to print the current name five times

```
names = ['Kelly', 'Jessa', 'Emma']
# outer loop
for name in names:
    # inner while loop
    count = 0
    while count < 5:
        print(name, end=' ')
        # increment counter
        count = count + 1
    print()
```

Output:

```
Kelly Kelly Kelly Kelly Kelly

Jessa Jessa Jessa Jessa Jessa
```

```
Emma Emma Emma Emma Emma
```

Break Nested loop

The break statement is used inside the loop to exit out of the loop. If the break statement is used inside a nested loop (loop inside another loop), it will terminate the innermost loop. In the following example, we have two loops. The outer `for` loop iterates the first four numbers using the `range()` function, and the inner `for` loop also iterates the first four numbers. Ifthe **outer number and a current number of the inner loop** are the same, then break the inner (nested) loop.

Example:

```
for i in range(4):
    for j in range(4):
        if j == i:
            break
        print(i, j)
```

Output:

```
1 0

2 0

2 1

3 0

3 1

3 2
```

Continue Nested loop

The continue statement skip the current iteration and move to the next iteration. In Python, when the `continue` statement is encountered inside the loop, it skips all the statements below it and immediately jumps to the next iteration. In the following example, we have two loops. The outer for loop iterates the first list, and the inner loop also iterates thesecond list of numbers. If the outer number and the inner loop's current number are the same, then move to the next iteration of an inner loop.

Example:

```
first = [2, 4, 6]
second = [2, 4, 6]
for i in first:
    for j in second:
        if i == j:
            continue
        print(i, '*', j, '= ', i * j)
```

Output:

```
2 * 4 =  8

2 * 6 =  12

4 * 2 =  8

4 * 6 =  24

6 * 2 =  12

6 * 4 =  24
```

Nested while Loop in Python

In Python, The while loop statement repeatedly executes a code block while a particular condition is true. We use a while loop when number iteration is not fixed. we will see how to use a while loop inside another while loop.

The syntax to write a **nested while loop** statement in Python is as follows:

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

Example:

In this example, we will print the first 10 numbers on each line 5 times.

```
i = 1
while i <= 5:
    j = 1
    while j <= 10:
        print(j, end='')
        j = j + 1
    i = i + 1
    print()
```

Output:

```
12345678910

12345678910

12345678910

12345678910

12345678910
```

for loop inside while loop

Sometimes it is helpful to use one type of loop inside another. we can put a `for` loop inside the `while` loop.

Assume we wanted to **print all perfect numbers from 1 to 100**

- Here we will iterate the first 100 numbers using a while loop
- In each iteration of the outer while loop, the inner for loop execute from 1 up to the current outer number to check if the current number is a perfect number.

```
print('Show Perfect number fom 1 to 100')
n = 2
# outer while loop
while n <= 100:
    x_sum = 0
    # inner for loop
    for i in range(1, n):
        if n % i == 0:
            x_sum += i
    if x_sum == n:
        print('Perfect number:', n)
    n += 1
```

When To Use a Nested Loop in Python?

- Nested loops are handy when you have nested arrays or lists that need to be looped through the same function.
- When you want to print different star and number patterns using rows can columns

Loops Inside Loops

A nested loop is a loop inside a loop.
The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

PYTHON ELSE SUITE

In Python, it is possible to use 'else' statement along with for loop or while loop in the form shown in Table:

for with else	while with else

for(var in sequence):
statements
else:
statements
	while(condition):
	statements
	else:
	statements

The else suite will be always executed irrespective of the statements in the loop are executed or not.

```
for i in range(5):
  print("Yes")
else:
print("No")
```

Output:

```
Yes
Yes
Yes
Yes
Yes
No
```

It means, the for loop statement is executed and also the else suite is executed.

A Python program to search for an element in the list of elements.

```
group1 = [1,2,3,4,5]
search = int(input('Enter element to search:'))
for element in group1:
    if search == element:
        print('Element found in group')
        break #come out of for loop
else:
    print('Element not found in group1') #this is else suite
```

Output:

```
Enter element to search:7
Element not found in group1
Enter element to search:4
Element found in group
```

while-loop-else-clause

The else-block is only executed if the while-loop is exhausted.

Ex1

```
n = 10
while n > 0:
    n = n - 1
    if n == 2:
        break
    print(n)
else:
    print("Loop is finished")
```

Output:

```
9
8
7
```

```
6
5
4
3
```

Ex2

```
n = 5
while n > 0:
    n = n - 1
    if n == 2:
        continue
    print(n)
else:
    print("Loop is finished")
```

Output:

```
4
3
1
0
Loop is finished
```

PYTHON BREAK STATEMENT

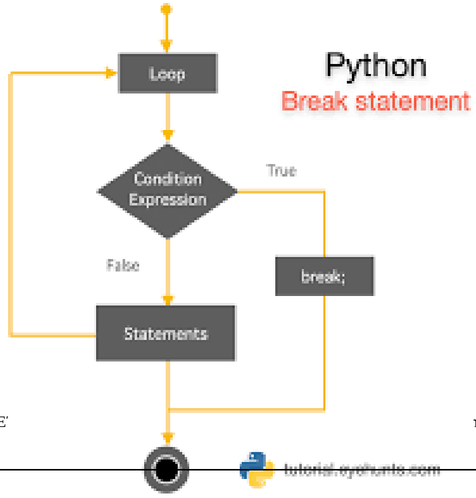
The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

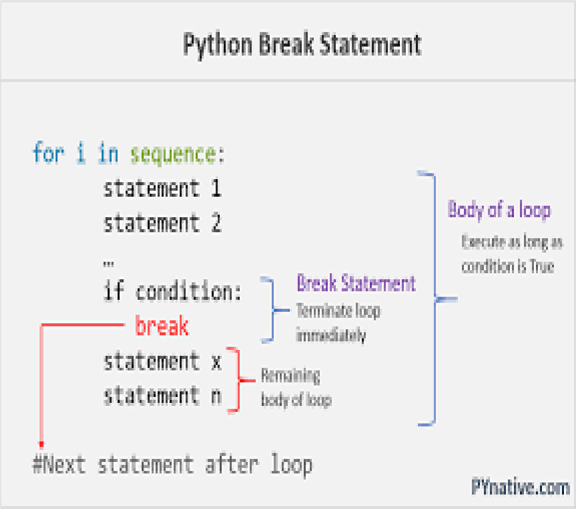
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

Syntax of break

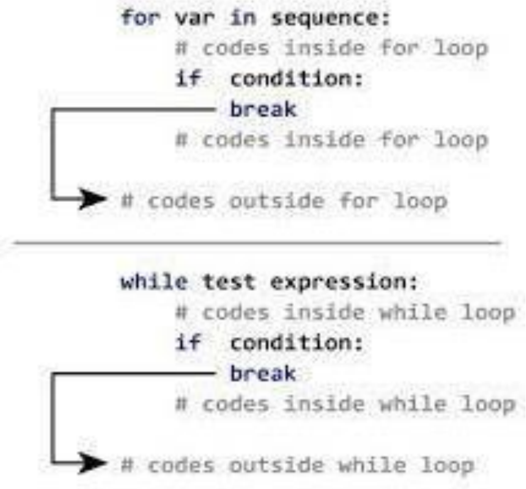
```
break
```

Flowchart of break





```
for i in sequence:
      statement 1
      statement 2
      ...
      if condition:
         break
      statement x
      statement n
#Next statement after loop
```



```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter
```

Output

```
Current Letter : P
Current Letter : y
Current Letter : t
```

Example

```
var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

Output

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

PYTHON CONTINUE STATEMENT

The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

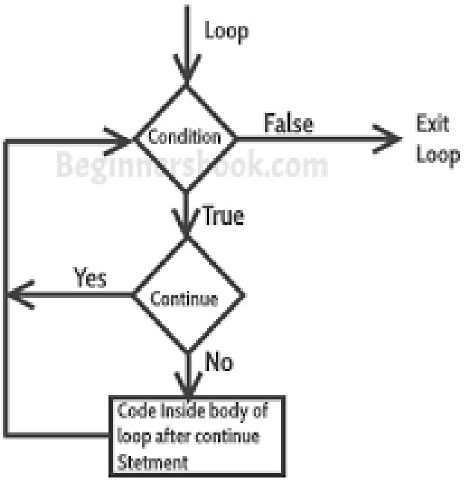
The continue statement can be used in both *while* and *for* loops.

The `continue` statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

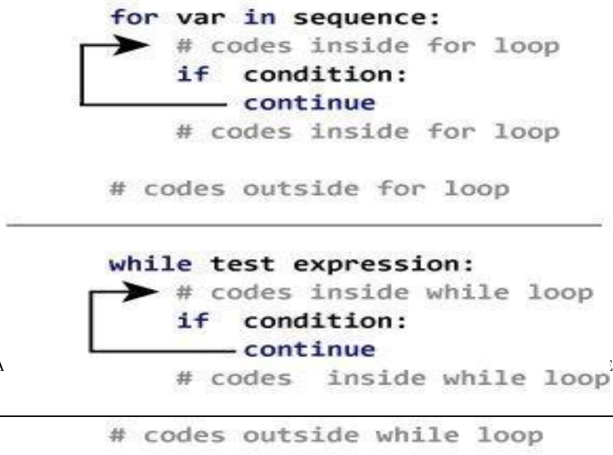
Syntax of Continue

```
continue
```

Flowchart of continue



The working of the `continue` statement in for and while loop is shown below.



```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop

while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

Example 1: Python continue

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

Output

```

s
t
r
n
g
The end
```

Example 2

```
for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

Example 3

```
var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

Output:

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Good bye!
```

PYTHON PASS STATEMENT

In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when the pass is executed. It results in no operation (NOP).

When we don't want to execute code, the pass can be used to execute empty. It is the same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used. It is beneficial when a statement is required syntactically, but we want we don't want to execute or execute it later.

Syntax of pass

```
pass
```

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the `pass`ment to construct a body that does nothing.

Example 1: pass Statement

```
'''pass is just a placeholder for
functionality to be added later.'''
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Example 2

```
for i in [1,2,3,4,5]:
    if(i==4):
        pass
        print("This is pass block",i)
    print(i)
```

Output:

```
1
2
3
This is pass block 4
4
5
```

PYTHON ASSERT STATEMENT

In Python, the assert statement is used to continue the execute if the given condition evaluates to True. If the assert condition evaluates to False, then it raises the AssertionError exception with the specified error message. The assert keyword is used when debugging code.

Syntax

```
assert condition
assert condition [, Error Message]
```

Example 1

```
x = 10
assert x > 0
print('x is a positive number.')
```

Output:

```
x is a positive number.
```

In the above example, the assert condition, x > 0 evalutes to be True, so it will continue to execute the next statement without any error.

The assert statement can optionally include an error message string, which gets displayed along with the AssertionError.

Consider the following assert statement with the error message.

Example 2

```
x = 0
assert x > 0, 'Only positive numbers are allowed'
print('x is a positive number.')
```

Output:

```
Traceback (most recent call last):
    assert x > 0, 'Only positive numbers are allowed'
AssertionError: Only positive numbers are allowed
```

Above, x=0, so the assert condition x > 0 becomes False, and so it will raisethe AssertionError with the specified message 'Only positive numbers are allowed'. It doesnot execute print('x is a positive number.') statement.

Example 3

```
def square(x):
    assert x>=0, 'Only positive numbers are allowed'
    return x*x

n = square(2) # returns 4
n = square(-2) # raise an AssertionError
```

Output:

```
Traceback (most recent call last):
    assert x > 0, 'Only positive numbers are allowed'
AssertionError: Only positive numbers are allowed
```

Above, square(2) will return 4, whereas square(-2) will raise an AssertionError because we passed -2. (The last value passed is -2 to the function)

PYTHON RETURN STATEMENT

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned.

Note: Return statement can not be used outside the function.

Functions are created to perform some task and this entire process is divided into three parts.

- accepting argument
- processing the argument
- producing result

Return function is used at the third step that is producting or returning result. This result is also called ‘value’ or return.

Function return None by default, in other words, if the programmer do not specify return value then None is returned.

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

Example 1:

```
# Python program to
# demonstrate return statement

def add(a, b):
     return a + b
res = add(2, 3)
print("Result of add function is {}".format(res))
```

Output:

Result of add function is 5

Example 2:

```
def addEven(n, m):
      if n%2 == 0 and m%2 == 0:
            return n+m
      else:
            return n*m
print(addEven(2,2))
print(addEven(2,3))
```

Output:

4

6

LIST COMPREHENSION

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list. It is an elegant way of creating list with other python constructs. One main benefit of using a list comprehension in Python is that **it's a single tool that you can use in many different situations**. In addition to standard list creation, list comprehensions can also be used for mapping and filtering. List comprehensions **provide a concise way to create lists**. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

Syntax:

1. **[expression for var in iterable]**
2. **[expression for var in iterable if cndn]**
3. **[expression if cndn else stmt for var in iterable]**

<u>Example: 1</u>

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Output:

```
[‘apple’, ‘banana’, ‘mango’]
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

Output:

```
[‘apple’, ‘banana’, ‘mango’]
```

Example: 2

#with out using list comprehension

```
list2=[ ]
def squarelist(list1):
    for x in list1:
        list2.append(x*x)
    return list2
squarelist([1,2,3,4])
```

Output:

```
[1, 4, 9, 16]
```

#with using list comprehension

```
listt=[1,2,3,4]
list1=[x*x for x in listt]
print(list1)
```

Output:

```
[1, 4, 9, 16]
```

Example: 3

```
list2=[x*x for x in range(1,5)]
print(list2)
```

Output:

```
[1, 4, 9, 16]
```

Example: 4

```
listt=[1,2,3,4]
list3=[x*x for x in listt if x%2==0]
print(list3)
```

Output:

```
[4,16]
```

Example: 5

```
list4=[i if i>2 else i+1 for i in range(1,11)]

print(list4)
```

Output:

```
[2, 3, 3, 4, 5, 6, 7, 8, 9, 10]
```

ALL THE BEST