www.metricsmania.com

Python Basic Programming

Authored by:

Raghavendra S

Section I

Python Features, Variables, Identifiers and Data Types

Python Language Features :

1) Easy to Learn and Use - It is developer-friendly and high level programming language

2) Expressive Language -Python language is more expressive means that it is more understandable and readable.

3) Interpreted Language - This makes debugging easy and thus suitable for beginners.

4) Cross-platform Language

5) Free and Open Source

6) Object-Oriented Language

7) Large Standard Library -Python has a large and broad library and provides

rich set of module and functions for rapid application development.

8) Supports GUI

9) Extensible --- Python Modules can be called in C Code and vice versa

Bonus Tip:

Check out this blog link to know more about Python language popularity

https://houseofbots.com/news-detail/3985-4-top-6-reasons-why-python-is-suddenly-super-popular-for-every-programmer

Python Keywords

a. Keywords are special words which are reserved and have a specific meaning

b. There are 33 keywords in Python

Task:

Execute this code to list out all the keywords, do not memorize, but get familiar with them.

Code:

```
import keyword

keyword.kwlist
```

Python Identifiers

1. Python identifiers are user defined names to represent a variable, function, class, module or any other object.

Example - productId, product_id, product_and_customer_id

2. Do not use digits to begin an identifier name, it will lead syntax error

2shape – wrong

shape2 - correct

3. '''[‘.’, ‘!’, ‘@’, ‘#’, ‘$’, ‘%’]''' forbidden symbols for constructing python identifiers.

Best Practices of Python Identifiers

1. Always start class names with Capital letter, other identifiers should begin with a lowercase

2. Declare private identifiers by using the (‘_’) underscore as their first letter. (User defined)

3. Don’t use ‘_’ as the leading and trailing character in an identifier.

As Python built-in types already use this notation.

4. Avoid using single character for identifiers like i = 1

5. Use _ to make between multiple words to make meaningful identifiers.

Ex: product_and_customer_id

Variable and Constants

Variables are containers of data, memory location which holds actual value

Always use camel case to write variables or use underscore between 2 words to make it meaningful and others to understand your code.

Example: prodductId, employeeName or employee_name

Python variables support multiple assignments as shown below:

var1 = var2=var3 = 800

Note: Python variables have id () and type ()

Id () – This function gives memory location of the variable (All variables are objects in Python! more on this in next session)

type () – returns data type of the variable.

Types of Variables (We shall discuss this in 4th Session in detail)

1. Local Variables

2. Global Variables

CONSTANTS

PI = 3.14

Python constants must be always in upper case. (It is best practice)

radius = 6

Task:

Try to print this as shown below:

Code:

print ("Area of Circle is :", PI * radius ** 2)

Note: ** is an exponential operator in python, (I discuss this in forthcoming sessions)

Python Data Types

Python has following data types :

1. Numbers
2. Strings
3. Booleans
4. List
5. Tuple
6. Dictionary
7. Sets

1. Number

Number has Int, Float and Complex data types. Note. (Double is deprecated in Python 3.x and above)

a. Int

It is any whole number
For example :
num = 100,

b. Float
Float holds all decimal numeric data. It is

Example:
num = 2.25

Type Coercion

Like other high level programming languages Python supports data type coercion. Change of data types is known as type conversion or coercion.

There are 2 types of coercion / conversions:

1. Implicit conversion or coercion

   Implicit conversion or coercion is when data type conversion takes place either during compilation or during run time and is handled directly by Python for you.

   Below example demonstrates the implicit coercion in python. Try to execute this code snippet and try to find out the type () at each line of code execution to get familiarity with data type.

   Code Snippet

   ```
   x_int  = 5
   type(x_int)
   y_float = 2.6
   type(y_float)
   coerced_dataType = x_int + y_float
   print(coerced_dataType)
   type(coerced_dataType)
   ```

   Note: When you add int with float, the resultant will always be type casted to float, hence in Python float takes precedence over int. It is because, float is higher order data type as compared to int.

2. Explicit conversion or coercion

   In case you want to convert float into int then you need to manual tell the Python to do so. It is explicit conversion or coercion.

Syntax:

```
(required_datatype)(Expression)
```

Try this example and check out type () at each step of code execution as shown below:

```
x_float

x_float = 25.25

type (x_float)

y_int = int (x_float)
type (y_int)
print (y_int)
```

Number Round up and down

The method ceil() in Python returns ceiling value of x i.e., the smallest integer not less than x , where x is a given value, to be passed as parameter to the ceil() function.

For example:

```
import math

x = math.ceil(3.5)

print(x)
```

Task

Execute the above code and write down your observations.

The method or function floor() in Python returns floor of x i.e., the largest integer not greater than x.

```
import math

x  = 9 / 4

y = math.floor(x)

print(y)
```

The above operations can be done with // floor division operator in python.

Try to execute the following and note your observations:

```
3 // 2
```

2. Strings

String is a word or a sentence or just a blank or empty white space between quotes (‘ ‘)

Strings are immutable objects in Python with very rich API to deal with string objects.

Strings can be written in any of the following way:

```
name = 'LKQ'

name = "LKQ"
```

name = """LKQ"""

string_var = """Python strings with triple quotes can be used to extend lengthy strings to multiple lines"""

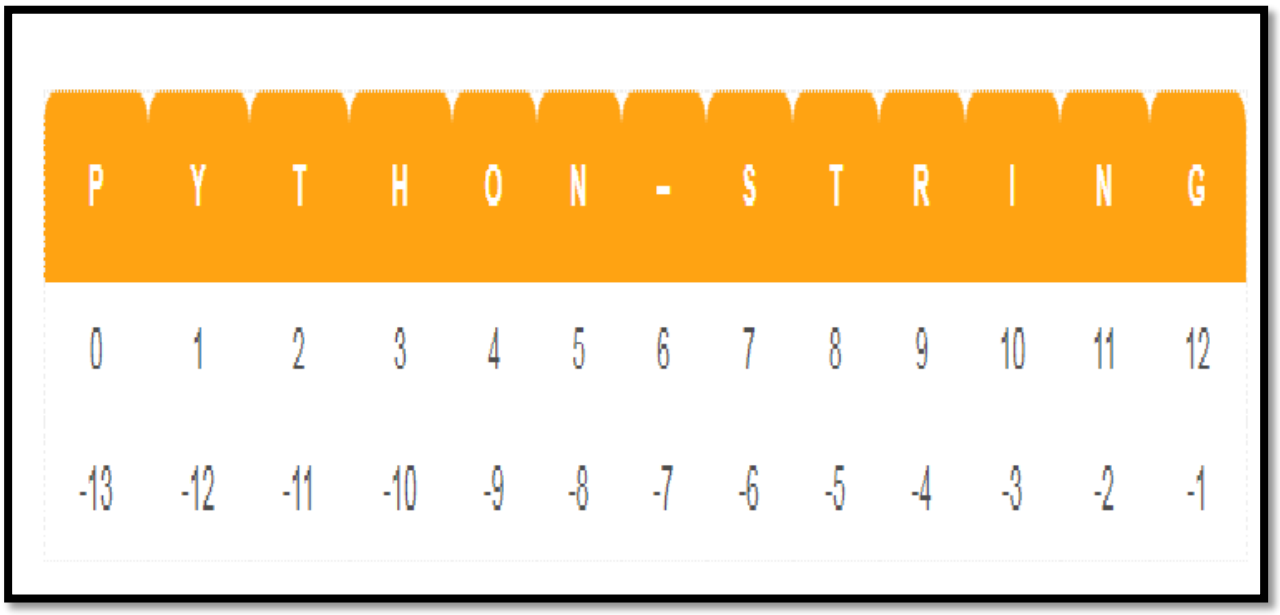
Triple quotes are used usually in multiline strings as shown above.

Note: Strings are immutable objects in Python, it means once created cannot be altered, but can be accessed through index.

Strings Slicing

string_extraction = 'Python-String'

Above string assignment can be logically viewed as below:



In above picture the string 'Python - String' is arranged with indices beginning from 0 to 13. Like all other programming languages, in python index begins with 0.

Python support forward indexing and backward indexing. Forward indexing begins with 0, whereas backward or reverse indexing begins with -1, as shown in the above picture.

Pass index to access the value in the given index.

For Ex:

string_extraction = 'Python-String'

string_extraction[0] returns P

string_extraction[-1] returns g

string_extraction[-2] returns n

We used slicing operator [:] to extract range of values in a string.

If you want to extract a range of elements from [2:8], it returns all the values from 2 to 8-1 elements.

For Ex:

To access in elements in string declaration:

string_extraction = 'Python-String'

string_extraction [2:8] returns ‘thon-S’.

Task:

Execute below code and make a note of your observations:

string_extraction [2:8]

string_extraction [7:]

string_extraction [:6]

string_extraction [7:-4]

Invalid String Usage and Errors

A. Index Error

```
string_extraction = 'Python-String'
string_extraction [13]
```

in the above code we are trying to extract an element which is beyond index. Hence, throws an Index Error

B. You can’t use 10.5 a factional value as index.

C. String are immutable can't modify, but can be deleted.

string_extraction = 'Python-String'

string_extraction [3] = 'I'

You get following error message: TypeError: 'str' object does not support item assignment.

String Operations

A. **Concatenation " + "**

```
string_1 = "Python"
string_2 = "Programming"
print(string_1 + ' ' +  string_2)
```

B. Repetition "*"

```
string_1 = "Python"
```

```
print(string_1 * 100)
```

C. in (membership)

```
string_1 = "Python"
print('p' in string_1)
```

D. \ escape

```
print("Automotive industry  is lead by \"LKQ" in US")
```

E. String formatting

```
def string_formatt(str1 , age, exp):
  print("Employee Name: %s, \nEmployee Age:%d, \n Experience:%f" %(str1,age,exp))
    string_formatt('Rober',23, 3.5)
```

F. original_string = "Keystone Operation Pvt.Ltd.,"

```
replace_string =  original_string.replace("Keystone","LKQ")
print(replace_string)
```

G. CAPITALIZE / FIRST LETTER

```
var = 'PYTHON'
print (var.capitalize())
```

H. lower()

```
var = 'PYTHON'
print (var.lower())
```

I. upper()

```
var = 'python'
print (var.upper())
```

J . swapcase()

```
var = 'PyThOn'
print (var.swapcase())
```

K. islower() 'looks for all characters to be lower'

```
var='Python'
print(var.islower())
```

L. isupper() 'looks for all characters to be upper'

```
var='PYTHON'
print(var.isupper())
```

M .isalum()

```
var='Python4'
print(var.isalnum())
```

3. Booleans - True or False -- 1 / 0

```
condition = False
if condition == True:
   print("Condition is set True")
else:
   print("Condition is set False")
```

Boolean is either True or False, True stands for 1 false stands for 0.

4. List

A **list** is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ].

1. It is sequence and an ordered collection of objects
2. It holds any type of objects – Numbers, Strings, letters and lists itself.
3. Lists are mutable objects – so you can remove(), del(), add() elements to it.

4. All elements in list are referenced by 'INDEX'

5. List index starts with 0

List Creation

List can be created in 2 ways:

Method-1:

num = []

This creates an empty list .

num = [1,2,3,4,5,6,7]

Task:

Check out the type by executing following code snippet :

type(num)

Method-2:

You can use **list()** function to create as shown below :

Note : You need to used [ ] in list ( ) function to create List.

number = list ([10, 20, 30,40, 50])

Task:

Check out the type by executing following code snippet :

type(number)

List of List

As mentioned above you can insert list with in list

nums = [[1,2,3],[10,20,30],[1.2,2.2,3.2]]

Task:

Check out the type and length by executing following code snippet :

type(nums)

len (nums) : len() length of a data type passed as argument.

List Extension

An existing list can be extend as follows :

number = [1,2,3,4,5,6]

```
nums = [ [1,2,3] , [10,20,30] , [1.2,2.2,3.2] ]
```

Task:

Check out the type and length by executing following code snippet:

len (nums)

List Comprehension

List comprehension is a way of creating dynamic list, based on a condition:

Syntax:

```
var = [ itr for itr in list if condition]
```

List Comprehension

Example - 1

```
alphabets = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p', 'q','r','s','t','u','v','w','x','y','z']

extract_vowels = [vowels for vowels in alphabets if vowels in 'aeiou' ]

print(extract_vowels)
```

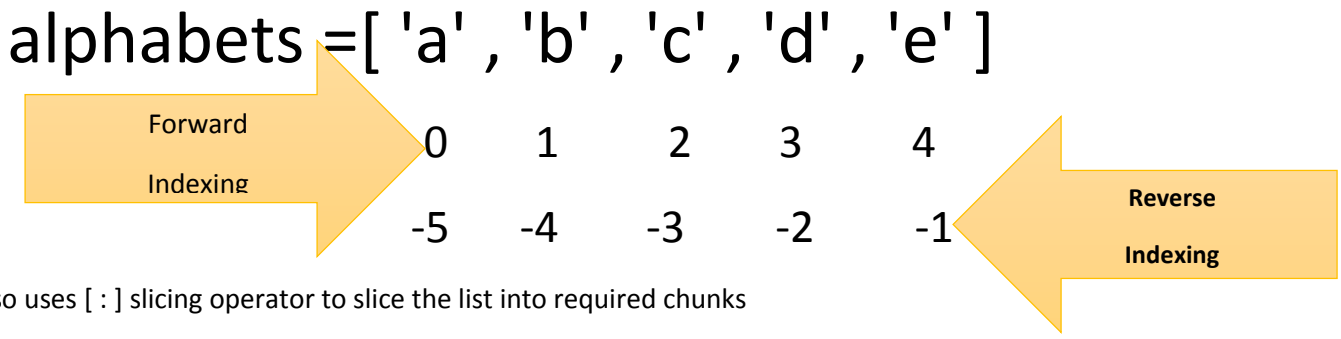
Example - 2

```
divisibleByThree = [i for i in range(300) if i % 3 == 0]

print (divisibleByThree)
```

List Slicing

List slicing is done as we did in String slicing (please refer String Slicing in Page No. 7).

Like string, list elements can be referred by Index, lists support both forward indexing and reverse indexing.



Lists also uses [:] slicing operator to slice the list into required chunks

Task

Create a list as shown below and execute the code snippets and write down your observations:

```
countries =['India','Iran','Japan','Germany','Chili', 'Portugal','Kenya']
```

Traversing through list

fruits = ['apple', 'banana', 'orange', 'ooty_apple']

We can traverse through entire list with 'for' loop as shown below :

Code Snippet -1 : This loop prints out all the fruits (elemets) in the list

```
for fruit in fruits:
    print(fruit)
```

Code Snippet -2 : This loop prints out all the fruits (elemets) and their respective indices in the list.

```
for index, elements in enumerate(fruits):
    print(index, elements)
```

Note : To print out element and index we pass list as argument to enumerate() function as shown below

Code Snippet -3 : This loop preints only index, len()

```
for index in range(len(fruits)):
    print(index)
```

List Replacement --- Lists are mutable objects

Lists are popular because they allow you to modify by adding, removing elements during the run time.

Find the following code snippets, follow along and execute the code snippets, make a note of your observations.

```
num = [32,33,34,35,36,37,38,39]
num
num[1:4] = 'xyz' # Multiple replacement 1st Method
num
num[2:5]
num[2:5] = [ 'a',  'b' , 'c'] # Multiple replacement 2nd Method
num
num[2:5] = [ ]
num

num = []
num
```

List Built in Functions

A. append() : This function adds value at last

```
number = [1,2,3,4,5,6,7,8,9,10]
number.append(11)
number
```

B. extend() : This function extends an old list with new list passed as argument

```
number = [1,2,3,4,5,6,7,8,9,10]
number_1  = [11,12,13,14]
number.extend(number_1)
number
```

C. insert() : This function inserts a given value, after the index, passed as 2nd argument to the function.

```
number = [1,2,3,4,5,6,7,8,9,10]
number.insert(5,'a')
number
```

D. remove() : This function removes the list value passed s argument. Note : The argument passed is not index, but value existing on the index.

```
number = [1,2,3,4,5,6,7,8,9,10]# arg is value not index
number.remove(10)
number
```

E. pop() : This function removes and returns the removed element to I/O terminal or to calling object . Note : The argument passed to pop() is index of the list element to be removed.

```
vowels = ['a','e','i','o','u']
#print(vowels.pop(1))

removed = vowels.pop(1) # removed is an object holding the value removed by pop()
print(removed)
```

F. reverse() : Reverses all elements as shown in the example

```
alphabets = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p', \
            'q','r','s','t','u','v','w','x','y','z']
alphabets.reverse()
print (alphabets)
```

G. sum() : sums up all elements in the list

```
number = [1,2,3,4,5,6,7,8,9,10]
sum(number)
```

Tuples

1. It is sequence and an ordered collection of objects.
2. It holds any type of objects – Numbers, Strings, letters and lists.
3. Tuples are immutable objects.
4. All elements in Tuple are referenced by INDEX []

Tuple creation methods.

A. Tuple with creation with ‘ () ‘

```
empty_tuples = ()

print("Nothing Exist in Tuple", empty_tuples)
```

B. Tuple with creation without ‘ () ‘

```
no_parentesis_tuple = 0,1,2,3,4,5,8

print(no_parentesis_tuple)
```

C. Tuple creation with built in function “tuple()”

Tuples can hold any data type

A. Number tuple

```
number_tuple = (30,40,50,60,70)
number_tuple
```

B. Mixed number tuple

```
mixed_tuple = (10, 3.3, 3+3j)
mixed_tuple
```

C. Tuple with mixed data types

```
mixed_datatypes = (45, "Name", [1,2,3,4])
mixed_datatypes
len(mixed_datatypes)
```

D. Tuple of Tuple

```
tuple_of_tuple = (('A','B','C'),(1,2,3,4))
tuple_of_tuple[1]
```

Built-In Function “tuple()”

```
tuple_from_api = tuple([0,1,2,3,4,5,8])

print(tuple_from_api)
```

Single String and Tuple

Single String always creates a string object as shown below :

```
tuple_string_variation = ("MyName")
print(tuple_string_variation)
type(tuple_string_variation)
```

Acessing elements in tuples is same as list with [:]

Task :

Execute following code snippets and make a note of your observations

```
fruits_tuple = ('Apple','Orange','Grapes','Banana')
fruits_tuple[:]
fruits_tuple[-3:]
fruits_tuple[3:]
fruits_tuple[1:2]
```

Tuples are immutable :

```
simple_tuple = ( 10, 20, 30, [40, 50] )

simple_tuple[0] = 1500
```

The above code throws an error. " TypeError: 'tuple' object does not support item assignment"

All mutable elements in Tuple can be modified

```
simple_tuple = (10,20,30,[40,50])
simple_tuple [3][0] = 100
simple_tuple [3][0]
print(simple_tuple)
```

Task :

Why is it possible to assign 100 in the above index, even though tuple is immutable?
Note down your observation.

Membership check

```
fruits_tuple = ('Apple','Orange','Grapes','Banana')
print("Does Apple exist in fruits_tuple?",  'Apple' in fruits_tuple)
```

in is a membership operator, it is used to check an element does exist or not.

Traversing in a Python Tuple

```
fruits_tuple = ('Apple','Orange','Grapes','Banana')
for fruits in fruits_tuple:
   print("Fruits:", fruits)
```

Dictionary

a. It is a scrambled collection of objects.
b. Dictionary is the only data type with key and values (rest are single value field)
c. The key is reference to the value

d. " : " seperates the key and value with { } braces

e. Key can't be duplicate, but values can repeat.

f. Values can be any valid python data types : string, number, or a tuple
Creating Dictionary

A.

```
empty_dict = {}
type(empty_dict)
```

B.

```
veg_dict = {1 : 'cabbage', 2 : 'banana', 3 : 'SweetCorn', 4:'greens'}
```

C.

```
mixed_dict = { 'Rank' : 'A Level', 1 : [1,2,3,4,5], 'Employee_Details': ('Emp_634', 30, 4485857) }
```

D. Dictionary can be created with built in function dict(), as shown below:

```
dict_created_func = dict({ 1: 'Veg', 2:'Non-Veg'})
type(dict_created_func)
```

Accessing Dictionary ([]) and get()

In both case we use key to access the values.

```
Employee_Details = {'Employee Name': 'Berry Jack', 'Roll No': 118, 'Jo Role': 'DataScientist'}

print (Employee_Details['Employee Name'])

print (Employee_Details['Roll No'])

print (Employee_Details.get('Employee Name'))
```

Dictionary is mutable objects :

pop () and popitem () are used to delete specific and an arbitrary item from dictionary.

```
monthNames = {1:'January', 2:'February', 3:'March', 4:'April', 5:'May', 6:'June'}

print(monthNames.pop(3)) # delete specific element
print(monthNames)

month = monthNames.popitem() # returns an arbitrary item
print("popitem() returns an arbitrary item and removes from dict:" ,month)

print(monthNames)

print(monthNames[4])

monthNames.clear() # clear all items
print(monthNames)
```

Iterate through a Dictionary

```
monthNames = {1:'January', 1:'February', 3:'March', 4:'April', 5:'May', 6:'June'}
Prints only keys
for key in monthNames:
   print(key)

Prints both key and value
for key, value in monthNames.items():
print(key,":",value)
```

Bonus Tip:
Check out this blog post
https://houseofbots.com/news-detail/4012-1-what-features-make-python-different-from-other-programming-languages

Section II

Python Operators, Precedence Control Statements and Loops

Following is an exhaustive list of operators:

Operators	Usage
{ }	Parentheses (grouping)
f(args...)	Function call
x[index:index]	Slicing
x[index]	Subscription
x.attribute	Attribute reference
**	Exponent
~x	Bitwise not
+x, -x	Positive, negative
*, /, %	Product, division, remainder
+, –	Addition, subtraction
<<, >>	Shifts left/right
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, membership, identity
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
lambda	Lambda expression

Above operators are listed according to their precedence

Arithmetic operator's precedence

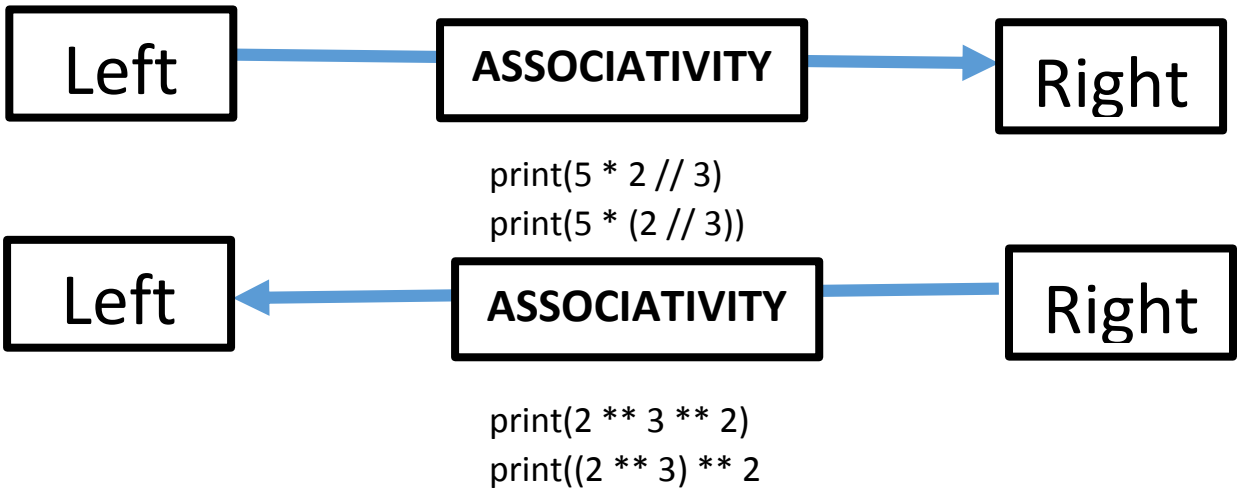
1. (), {}, []

2. **
3. *, /, //, %
4. + , -

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

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

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

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

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

Associativity

All operators have left to right associativity, except ** , has right to left associativity .




```
print(5 * 2 // 3)
print(5 * (2 // 3))
```



```
print(2 ** 3 ** 2)
print((2 ** 3) ** 2
```

Non Associative Operators

```
x = 3
y = 4
```

```
z = 5

(x < y < z)
```
The following line is evaluated as follows:
```
(x  <  y) ( y <  z)
```

1. Arithmetic Operators (+ , - , *, /, //, % , **)

```
a = 4
b = 2
x = 7
y = 2

print('sum :' ,a + b)
print('Subtract: ', a - b)
print('Product :', a * b)
print('Division :', a / b)
print('Floor Division:' , x // y)
print('Modules :' ,a % b)
print('Exponent:', a ** 2)
print(2 ** -2) # gives 0.25
```

2.Comparison Operator (> , < , == , !=, >=, <=)

Comparison operators have outcome either True /False

Follow along the example and make a note of your observation:

```
x = 8
y = 4

print ( x > y )
print( x < y )
print( x == y)
print( x != y )
print( x >= y )
print(x <= y)
```

3. Logical Operators (and, or, not)

True if both operands are true
or - True if either of the operand is true
not - real boolean

```
x = True
y = True
print(y and x)

print(y or x)
print(not x) # complements
```

4. Assignment Operators (= , +=, -=,*=,/=,//= , %=,=)**

```
x = 1
x += 5 # (x = x + 5)
print(x)

x=6
x -= 5 #( x = x- 5)
print(x)

x = 2
x *= 5  #(x = x * 5 )
print(x)

x = 10
x *= 5 #(x = x * 5 )
print(x)


x = 10
x  /= 5  #(x = x / 5 )
print(x)


x = 7
x  //= 5  #(x = x // 5 )
print(x)

x = 7
x  %= 5  #(x = x % 5 )
print(x)

x = 2
x **= 5 #( x = x ** 5)
print(x)
```

5. Identity Operators (is , is not)

a. Compare the memory locations of two Python objects/variables
b. Find out specific class or type

Task:

Execute following code snippets and make a note of your observations:

```
a = 7
b = 7
id(a)
id(b)
if ( a is b):
   print('True')
```

```
else:
   print('False')

if ( a is not b):
   print('True')
else:
   print('False')

if (type(a) is int):
 print("true")
else:
 print("false")
```

 # Using 'is not' identity operator

```
b = 7.5
if (type(b) is not int):
 print("true")
else:
 print("false")
```

‘is’ with list objects

```
list_1 = [1,2,3]
list_2 = [1,2,3]

id(list_1)
id(list_2)

if (list_1 is list_2):
   print("True")
else:
   print("False")
```

6. Membership Operators (in, not in)

```
a = "LKQ India Operations"
print('Q' in a )
print('LKQ' not in a)

b = {1:'APPLE',
    2:'Orange'}
print(1 in b)
```

Python Conditional Statements

a. if..... if/else

==========

Example – 1:

```
days = 366
if days == 366:
   print("You Answered Correct")
   print("Congratulation")
else:
   print("You Answered Wrong")
   print("Better Luck Next Time")
```

Example – 2:

```
while True:
   days = int(input("How Many days are there in a leap year"))
   if days == 366:
      print("You Answered Correct")
print("Congratulation")
```

Example – 3:

```
while True:
   days = int(input("How Many days are there in a leap year"))
   if days == 366:
      print("You Answered Correct")
      print("Congratulation")
      break
   else:
      print("You Answered Wrong")
      print("Better Luck Next Time")
```

if ' with' not operator

==========

```
x = 10
y = 20

if not y > x:
   print('X is smaller than y')

else:
   print('Y is greater than X ')
```

if – elif – elif –else

==========

Example -1

```
while True:
   answer = input("Name an ordered python datastructe in python..").lower()
```

```
   ans = answer.strip()
if ans == "list" :
      print("You are right")
      break

elif ans == "tuple" :
      print("You are right")
      break
else:
      print("You are wrong")
```

Example -2

```
while True:
   answer = input("Enter Vowels in English..").lower()
   ans = ''.join(answer.split()) #to remove white space before and after

   if ans == 'a':
      print("You are right")

   elif ans == "e":
      print("You are right")

   elif ans == "i":
      print("You are right")

   elif ans == "o":
      print("You are right")

   elif ans == "u":
      print("You are right")

   else:
      print("You are wrong")
```

Nested ifs

```
while True:
   response = int(input("How many days are there in a leap year? "))
   print("You entered:", response)

   if response == 366 :
      print("You have cleared the first level.")
      response = input("What month has an extra day in leap year?? ").lower()

       if response == "February" :
       print("You have cleared the test.")
          break
```

```
    else :
        print("You have failed the test.")
        break
  else :
    print("Your input is wrong, please try again.")
```

for loops

'for' loop is a versatile control statement, can be used on complex iterative objects, which have elements existing in sequence or in a range.

Syntax :

for index **in** sequence:

In a for loop sequence can be a list, tuple, dictionary, set, a range() etc., which are essentially a sequence of elements existing in a data type.

Python for loop is enhanced to use 'else' clause with it, this is first of its kind existing in python, not in any of its predecessor.

Example – 1 :

```
number_list = [10,20,30,40,8,9]
for num in number_list:
      print(num)
```

Example – 2 :

```
vowels = ('A','E','I','O','U')
for vow in vowels:
   if vow not in ('A','U'):
      print(vow)
```

Example – 3 :

```
int_list = [1,2,3,4,5,6]
sum = 0
for iter in int_list:
   sum += iter
print("Sum = ", sum)
print("Avg = ", sum /len(int_list))
```

Example – 4 :

```
for iter in range(1 , 10):
   print(iter)
```

Example – 5: 'for' loop with 'else'

```
fruits_list = ['apple','orange','muskmelon','grapes']

        for iter in fruits_list:
           if iter == 'kiwi':
```

```
            print(iter)
            break
    else:
       print('Did not find required fruit')
```

Example – 6: for loop with continue clause

```
for num in range (8, 16):
   if num % 2 ==0:
       print('Found an Even Number', num)
          continue
       print('Found an odd number', num)
```

Example – 7 : for loop and in operator usage

```
number_list = [10,20,30,40,8,9]

for num in number_list:
   if not num in (8,9):
      print("Allowed Item:", num)
```

while loop

Execute the following example and make a note of your observation:

```
def while_demo():
    count = 0
    while count < 5:

    num = int(input("Enter number between 1 - 10"))

    if (num < 1) or (num > 10):

       print('You have entered Beyond Range', num)

    else:
       print("You have entered ", num)
       count += 1

while_demo()
```

Python Functions

Python functions are very powerful, yet simple to implement and write:

1. Python functions have to begin with keyword "**def**"
2. Python functions cannot be body less, at least it must have pass keyword with in it.

Example- 1 : Simple Function with print statement

```
def greet_hello():
   print("Hello to all")
greet_hello()
```

Example- 2 : Bodyless Function with 'pass' keyword

```
def greet_empty_hello():
   pass
greet_empty_hello()
```

Example- 3 : Bodyless Function with 'pass' keyword

```
def number_check(num):
        if num % 2 == 0:
          def message():
                print('It is an Even Number')
       else:
          def message():
                print('It is an Odd Number')
      message()
number_check(11) --------- > Execute first
number_check(100)------- > Execute second
```

Example- 4 : Call By Reference

```
def change_list(itemlist):
        itemlist.append([10,20,30,40])
        print("Values Inside Function", itemlist)
        print(id(item_list))
        item_list = [1,2,3,4]
        change_list(item_list)
        print('Values outside Function', item_list)
        id(item_list)
```

Example- 5 : Call By Value

```
def change_list(item_list):
        print("Values Inside Function- 1", item_list)
```

```
        item_list= [10,20,30,40]

        print("Values Inside Function - 2 ", item_list)
        print(id(item_list))

item_list = [1,2,3,4]
change_list(item_list)
print('Values outside Function', item_list)
id(item_list)
```

Example- 6 : Keyword Argument

```
def display_name(str):
  print("My Company Name is", str)

str = 'LKQ'
display_name(str = 'LKQ')
```

Example- 7 : Function keywords with no order in passing arguments

```
def display_LKQ_Details( company_size, company_name):
        print("My Name Company Name is", company_name)
        print("My Name Company Size is", company_size)
display_LKQ_Details(company_name = 'LKQ', company_size= 10000)
```

Example- 8 : Function with default arguments

```
def display_LKQ_Details(company_name, company_size = 10000):
    print("My Name Company Name is", company_name)
    print("My Name Company Size is", company_size)
display_LKQ_Details(company_name = 'LKQ', company_size=20000)
display_LKQ_Details(company_name = 'LKQ')
```

Variable Length Arguments and Keyword Arguments

Variable length arguments are two or more than two variables packed in a variable as arguments, but the variable is prefixed with *. For ex: *var is a variable length argument which can be assigned with any number of data type as arguments, which in turn can be passed to a function.

*var = (“Apple”, 1 , 5.26)

Variable length arguments are useful in a situation, where we do not know number of parameters to be passed to a function, it may vary from time to time. In such situations variable length arguments are very handy to handle such situations.

Follow along the examples:

Example – 1:

```
def test_args_args(*args):

   for var in args:

      print(var)

test_args_args('Workingat', 'lkq' ,'is fun')
```

Example – 2:

```
def test_args_args(*args):

   for var in args:

      print(var)

args = ("two", 3, 5)
test_args_args(args)
```

Along with variable length arguments, we can pass formal arguments to as shown in the following example:

```
def test_args_args(extra, *args):

   print("extra argument:", extra)
   for var in args:
      print("Printing from args:" , var)
test_args_args("We all say", "Working at", "lkq" ,"is fun")
```

Keyword Arguments are another type of variable where as you can pass a dictionary type data structure, essentially key and value pair of arguments to functions. It is one of the most versatile way processing a huge data structure like dictionary.

Take up the following example, to know the power of keyword arguments:

Example – 1:

```
def myCompanyNames(**kwargs):
   for key, value in kwargs.items():
      print ("%s == %s" %(key, value))

myCompanyNames(first ='LKQ', mid ='ECP', last='Rihag')
```

Along with **Keyword Arguments,** we can pass formal arguments as shown below ;

Example – 2:

```
def myCompanyNames(arg1, **kwargs):

   print(arg1)
   for key, value in kwargs.items():
      print ("%s == %s" %(key, value))
 myCompanyNames("Hi", first ='LKQ', mid ='ECP', last='LKQ India')
```

Python functions take formal, variable length and keyword arguments simultaneously as shown below:

function (formal_arguments, variable_length_arguments, keyword_arguments)

Example - 3:

```
def function_with_kwargs(arg_1, *argv, **kwargs):

   print("Normal Argument : ", arg_1)

   for var in argv:
      print("Argument Variables:", var)

   for kvar in kwargs:

      print("Keyword Argument Variables:", kvar)

args = 20,30,40,50,60
kwargs = {'Bangalore' : 1 , 'Karnataka':2,"Kannada": 3}
function_with_kwargs(10,*args,**kwargs)
```

Anonymous or Lamda Functions

Lamda functions nameless functions, used to get a function object, lambda functions can have any number of arguments, but it should have one expression.

Syntax:

lambda arguments: expression

Example – 1 :

```
square_root = lambda x: x ** 2
print(square_root(5))
```

Lambda functions come as handy to pass as arguments to higher order function like **filter()** and **map()**

Example – 2 :

Lambda with filter() function:

```
num_list = [1,2,3,4,5,6,9,8]
new_numlist = list(filter(lambda x: (x % 2 == 0), num_list))
print(new_numlist)
```

Example – 3 :

Lambda with map() function:

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Python - Object Oriented Programming

it's a whole programming paradigm based around objects (data structures) that contain data fields and methods. Object oriented programming is technique of solving complex problems by identifying classes and creating objects

out of those classes to interact with other objects to make a software as an integrated system to solve the give problem.

Important definitions of Object Oriented Programming.

A **class** is an arrangement of variables and functions into a single logical entity. Class is a template and just a concept.

An **object** is an instance of a class. An object implements all are required attributes (features) and functions (behaviors). Also, object is a working instance of a class created at runtime.

Minimal Requirements of creating a Python class:

1. The 'class' keyword

2. The instance attributes

3. The class attributes

4. The 'self' keyword

5. The '__init__' method

1. The 'class' keyword

```
class Employee:
   pass
```

2. The instance attributes

instance attributes are visible at objet level

```
class Employee:

   def __init__(self, emp_name, emp_id):
      self.empolyee_name = emp_name
      self.empolyee_identity = emp_id
```

3. The class attributes

Class attributes are visible at class level, hence available for all objects

```
class Employee:
```

```
   empCount = 0

   def __init__(self, emp_name, emp_id):
      self.empolyee_name = emp_name
      self.empolyee_identity = emp_id
```

4. self

1. It is used to represent the instance of a class.
2. It is handle for accessing the class members such as attributes from the class methods
3. It is implicitly the first argument to the __init__() in every Python class

5 __init()__

1. It is a unique method associated with every Python class
2. It is to initialize the class attributes with user supplied values
3. It is generally known as Constructor in OOP languages

Putting it all together we can build a class and instantiate objects as follows:

Example – 1 :

```
class Employee:
   empCount = 0

   def __init__(self, emp_name, emp_id):
      self.empolyee_name = emp_name
      self.empolyee_identity = emp_id
      Employee.empCount +=1

   def showEmployeeDetails(self):

      print('Employee Name is :' , self.empolyee_name)
      print('Employee ID  is :' , self.empolyee_identity)

Employee_1 = Employee('Maria',634)
Employee_2 = Employee('Raja',9355)

Employee_1.showEmployeeDetails()
Employee_2.showEmployeeDetails()
print('Total Employees:', Employee.empCount)
```

Example – 2:

```
class Animal:
```

```
    # class attributes
    species_1 = 'Mammals'
    species_2 = 'Birds'

    def __init__(self,name,sound):

        #instance attributes
        self.Animal_name = name
        self.Animal_sound = sound

        #instance methods
          def sound(self, name, animal_sound):

            return "{} sound is {}".format(self.Animal_sound)

        def eats(self, food):

            return "{} eats {}".format(self.food, food)

Animal_1 = Animal('Tiger', 'roars')
print ("{} makes {}".format(Animal_1.Animal_name, Animal_1.Animal_sound))
Animal_1.species_1

Bird_1 = Animal("Nighingale", "Chirping")
print("{} makes {} ".format(Bird_1.Animal_name,Bird_1.Animal_sound))
Bird_1.species_2
```

****