Python

Python Keywords

Keyswords are reserved words in python. We can't use a keyword as a variable, function name or any other identifier. Keywords are case sensitive.

```
# Get all the keywords in python
import keyword

print(keyword.kwlist)
# Output -> ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

print("\nTotal number of keywords: ", len(keyword.kwlist))
# Output -> 35
```

Identifiers

It is the name given to the entities like class, functions, variables, etc. in Python.

Rules for writing Identifiers:-

1. Combination of letters in lowercase or uppercase or digits or an underscore
2. Cannot start with a digit.
3. Keywords cannot be used as an identifier.

```
variable_1 = 123
variable1 = 12
_variable = 1234
1variable = 1234 #invalid syntax
global = 1 #invalid syntax
```

Variables

A variable is a location in memory used to store some data (value).

Variable Assignments

```
a = 10
b = 5.5
c = "Machine Learning"
```

Multiple Assignments

```
a, b, c = 10, 5.5, "Machine Learning"
a = b = c = "ML"
```

Storage Locations

```
x = 3
print(id(x))    #print address of varibale x
```

Data Types

Every value in Python has a data type. Since everything is an object in python, data types are actually classes and variables are instance (object) of these classes.

Numbers

Integers, floating point numbers and complex numbers falls under numbers category. They are defined as int, float and complex class.

We can use the type() function to know which class a variable or a value belongs to and the instance() function to check if an object belongs to particular class.

```
a = 5
print(a, " is of type", type(a))    #5 is of type <class 'int'>
```

```
a = 1 + 2j
print(type(a))    #<class 'complex'>
print(isinstance(a, complex))   #true
```

Boolean

Boolean represents the truth value true and false

```
a = True
print(type(a))    #<class 'bool'>
```

Strings

String is a sequence of Unicode characters. We can use single or double quotes to represent strings. Multiline strings can be denoted using triple quotes, ''' or """.

The first character of the string has the index 0.

```
a = "We are learning Python"
print(type(a))    #<class 'str'>

s = '''This is multiline
      string'''

print(s[1])   #'h'
print(s[-1])    #'g'

#slicing
print(s[5:])  #'is multiline string'
```

List

List is an ordered sequence of items. All the item in a list do not need to of the same type.

Lists are mutable, meaning, value of the element of a list can be altered.

```
a = [10, 20, 30, "World"]
print(a[2])   #30

a[2] = "Hello"
print(a)    #[10, 20, "Hello", "World"]
```

Tuple

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

```
a = (1, 2, "ML")
print(a[1])   #2

a[1] = "DL"   #TypeError
```

Set

Set is an unordered collection of unique items. Items in a set are not ordered.

We can perform set operations like union, intersection on two sets. Sets have unique value

```
a = {10, 20, 30, 5, 23}
print(type(a))    #<class 'set'>

s = {1, 2, 2, 3, 4}
```

```
print(S)    #{1, 2, 3, 4}

print(s[1])   #TypeError, as sets are unordered
```

Dictionary

It is an unordered collection of key-value pairs.

```
d = {"a": "apple", "b": "bat"}
print(d["a"])   #'apple'
```

Conversion between Datatypes

Conversion to and from string must contain compatible values.

```
float(5)    #convert integer to float - 5.0

int(5.1)    #convert float to int - 5

str(20)   #convert integer to string - "20"
```

We can convert one sequence to other

```
a = [1, 2, 3]
print(type(a)) #<class 'list'>
s = set(a)  #convert list to set
print(type(s)) #<class 'set'>

#convert string to list
list("Hello")     #['H', 'e', 'l', 'l', 'o']
```

Python Input and Output

Output Formatting

```
a = 10; b = 20
print("The value of a is {} and b is {}".format(a, b))
# 'The value of a is 10 and b is 20'
```

```
a = 10; b = 20
print("The value of b is {1} and a is {0}".format(a, b))
# 'The value of b is 20 and a is 10'
```

```
print("Hello {name}, {greeting}!",format(name="Shashank", greeting="Whats Up"));
# 'Hell0 Shashank, Whats Up!'
```

```
print("Hello {0}, {1}!",format(name="Shashank", greeting="Whats Up"));
# 'Hell0 Shashank, Whats Up!'
```

Python Input

```
num = input("Enter a number: ") #gives a textbox
print(num)
```

Operators

Operators are special symbols in python that can carry out arithmetic and logical computation. The value that the operator operates on is called the operand.

Operator Types

1. **Arithmetic**

   ```
   #addition (+)
   #subtraction (-)
   #multiplication (*)
   #division (/)
   #modulo division (%)
   #Floor Division (//)
   #Exponent (**)
   ```

2. **Comparison**

   > <, >, ==, !=, >=, <=

3. **Logical (Boolean)**

   > and, or and not

4. **Bitwise**

   > &, |, ~, ^, >>, <<

5. **Assignment**

```
#Add AND (+=)
#subtract AND (-=)
#Multiply AND (*=)
#Divide AND (/=)
#Modulus AND (%=)
#Floor division AND (//=)
#Exponent AND (*=)
```

6. **Special**
    - **Identity Operator** - **is and is not** are identity operators. They are used to check if two values are located on the same part of the memory.

    ```
    a = 5; b = 5
    print(a is b) #true

    L1 = [1, 2, 3]
    L2 = [1, 2, 3]
    print(L1 is L2) #false

    s1 = "ABCD"
    s2 = "ABCD"
    print(s1 is not s2) #false
    ```

    - **Membership Operators** - **in and in not** are membership operators. They are used to test weather a value or variable is found in a sequence (string, list, tuple, set or dictionary).

    ```
    l = [1, 2, 3, 4]
    print(1 in l) #true

    d = {1: "a", 2: "b"}
    print(2 in d) #true
    ```

Control Flow

if Statement

```
num = 10
if num > 0:
  print("Number is positive")
print("Number is Negative")

#'Number is positive'
```

if...else Statement

```
num = 10
if num > 0:
  print("Number is positive")
else:
  print("Number is Negative")

#'Number is positive'
```

if...elif...else Statement

```
num = 10
if num > 0:
  print("Number is positive")
elif num == 0:
  print("ZERO")
else:
  print("Number is Negative")

#'Number is positive'
```

while Loop

The while loop is used to iterate over a block of code as long as the condition is true.

```
lst = [10, 20, 30, 40, 50, 60]
product = 1
index = 0
while index < len(lst):
    product = product * lst[index]
    index = index + 1
print("The product of list is {}".format(product))

#'The product of list is 720000000'
```

while loop with else

```
numbers = [1, 2, 3]

index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1
else:
    print("No numbers left in the list.")
```

```
# 1
# 2
# 3
# No numbers left in the list.
```

```
# Program to find a prime number

num = int(input("Enter a number: "))
print(num)

isDivisible = False

i = 2;
while i < num:
    if(num % i == 0):
        isDivisible = True
        print("{} is divisible by {}".format(num, i))
    i += 1;

if isDivisible:
    print("{} is not a Prime number".format(num))
else:
    print("{} is a Prime number".format(num))
```

for Loop

```
itm = [10, 12, 13, 14, 15]

product = 1
for el in itm:
    product *= el

print("The product of all the items in the list is {}".format(product))

# The product of all the items in the list is 327600
```

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

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

# print range of numbers from 1 to 20 with step size of 2
for i in range(1, 20 , 2):
    print(i)
```

```
# Program to find the prime numbers within an interval

index1 = 1
index2 = 10

print("Prime numbers between {} and {} are:".format(index1, index2))

for num in range(index1, index2+1):
    if num > 1:
        isDivisible = False
        for i in range(2, num):
            if(num % i == 0):
                isDivisible = True
        if not isDivisible:
            print(num)
```

break Statement

```
lst1 = [2, 3, 4, 6, 7]

for el in lst1:
    if(el % 2 != 0):
        break
    print(el)
else:
    print("inside the for loop")
print("outside the for loop")

# 2
# outside the for loop
```

continue statement

```
lst1 = [2, 4, 6, 7, 8]

for el in lst1:
    if(el % 2 != 0):
        continue
    print(el)
else:
    print("inside the for loop")
print("outside the for loop")

# 2
# 4
# 6
# 8
# inside the for loop
# outside the for loop
```

List

List is one of the sequence data-structure. List are the collection of items (Strings, integers or even other lists). List are enclosed in []. Each item in the list have assigned index.

> Lists are mutable, which means they can be changed.

List Creation

```
emptyList = []
lst = ['one', 'two', 'three'] # list of strings
lst2 = [1, 2, 3, 4] # list of integers
lst3 = [[1, 2], [3, 4]] #list of lists
lst4 = [1, 'two', 3.45 ] #list of different data types
```

List Length

```
len(lst)
```

List Append

```
lst.append('four')
# will add 'four' at the end of the list
```

List Insert

```
lst.insert(2, "three")
# will add 'three' at index 2
```

List Remove

```
lst.remove('three')
# will remove the first occurance of 'three' from the list
```

List Append & Extend

```
lst = [1, 2, 3, 4]
lst2 = [5, 6]

lst.append(lst2) # [1, 2, 3, 4, [5, 6]]

lst.extend(lst2) # [1, 2, 3, 4, 5, 6]
# extend will join the lst2 with lst
```

List Delete

```
del lst[1] # removes the element at index 1

# or we can use pop() method
a = pop(1)
```

List related keywords in Python

```
# keyword 'in' is used to test if an element is in a list
lst = [1, 2, 3, 4]
if 2 in lst:
    print("True")

#keyword 'not' can be combined with 'in;
if 4 not in lst:
    print("False")
```

List Reverse

```
lst.reverse()
```

List Sorting

The easiest way to sort a List is to use sorted(list) function. That takes a list and returns a new list in sorted order. The original list is not changed.

> The sorted() optional argument, reverse=True, nakes it sort backwards

```
numbers = [3, 2, 5, 4, 1]

sorted_lst = sorted(numbers)
print(sorted_lst) # 1, 2, 3, 4, 5
```

```
reverse_lst = sorted(numbers, reverse=True)
print(reverse_lst) # 5, 4, 3, 2, 1
```

List having multiple reference

```
lst = [1, 2, 3, 4]
abc = lst
abc.append(5)

print(lst) # 1, 2, 3, 4, 5
```

String split to create a list

```
s = "Lets, learn, python"
slst = s.split(',')
print(slst) # ['Lets', 'learn', 'python']
```

> Default split is white-character: space or tab

List Indexing

Accessing elements in a list is called indexing.

```
lst = [1, 2, 3, 4]
lst[1] # 2
# print the last element using negative index
last[-1] # 4
```

List Slicing

> syntax - [start: end: step size]

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#print all the numbers
numbers[:]

#print from index 1 to 4
numbers[1:4] # [2, 3, 4], last index is not included

#print alternate numbers
numbers[1::2] # [2, 4, 6, 8, 10]
numbers[::2] # [1, 3, 5, 7, 9]
```

List extended using '+'

```
lst1 = [1, 2, 3]
lst2 = ['four', 'five']
lst = lst1 + lst2
print(lst) # [1, 2, 3, 'four', 'five']
```

List Count

```
#print frequency of element in the list
numbers = [1, 2, 3, 1, 4, 1]
print(numbers.count(1)) # 3
```

List Looping

```
lst = [1, 2, 3, 4]
for ele in lst:
    print(ele)
```

List Comprehensions

List comprehensions provide a concise way to create lists.

```
# without list comprehension
squares = []
for i in range(10):
    squares.append(i**2)
print(squares)

# with list comprehension
squares = [(i**2) for i in range(10)]
print(squares)
```

Nested List Comprehensions

```
# transpose of a matrix without list comprehension
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

transpose = []
```

```
for i in range(3):
    lst = []
    for row in matrix:
        lst.append(row[i])
    transpose.append(lst)

print(transpose)

# with list comprehension
transpose_comp = [[row[i] for row in matrix] for i in range(3)]
print(transpose_comp)
```

Tuples

A tuple is similar to list. The difference between the two is that we can't change the elements of tuple once it is assigned whereas in the list, elements can be changed.

```
# empty tuple
t = ()

# tuple having int
t = (1, 2, 3)

#tuple having different data-types
t = (1, "two", 3.5)

# nested tuple
t = (1, (2, 3, 4), [5, 6, 7])
```

> A comma is required at the end for creating a tuple. Also, paranthesis is optional for creating a tuple

```
t = ("John")
type(t) # str

t = ("John",)
type(t) # tuple

t = "John",
type(t) # tuple
```

Accessing elements in tuple

```
t = ('one', 'two', 'three', 'four')

print(t[1]) # two

# use negative index to print the last element
```

```
print(t[-1]) # four

# slicing
print(t[1:3]) # 'two', 'three'
```

Changing a Tuple

Unlike lists, tuples are immutable. So elements cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

```
tpl = (1, 2, 3, 4, [5, 6, 7])
t[2] = 21 # TypeError: 'tuple' object does not support item assignment

tpl[4][2] = "Seven"
print(tpl) # (1, 2, 3, 4, [5, 6, 'Seven'])
```

Concatinating a tuple

```
tp1 = (1, 2) + (3, 4) # (1, 2, 3, 4)
```

Repeat elements in a tuple for number of times

```
t = (('John', ) * 4)
print(t) # ('John', 'John', 'John', 'John')
```

Tuple Deletion

We cannot delete individual element from a tuple as tuples are immutable. But we can delete the entire tuple

```
t = (1, 2, 3, 4)

del t
```

Tuple Count

Returns the frequency of a particular element in the tuple.

```
t = (1, 2, 1, 2, 3, 5, 1)
t.count(1) # 3
```

Tuple Index

Returns the index of the first matched element provided

```
t = (1, 2, 1, 2, 3, 5, 1)
t.index(2) # 1
```

Tuple Membership

Test if an item exist in a tuple or not, using the keyword in.

```
t = (1, 2, 1, 2, 3, 5, 1)
print(1 in t) # True
print(7 in t) # False
```

Tuple Length

```
t = (1, 2, 1, 2, 3, 5, 1)
len(t) # 7
```

Tuple Sort

Takes elements in the tuple and return a new sorted list.

```
t = (1, 2, 1, 2, 3, 5, 1)
sort_t = sorted(t)
print(sort_t) # [1, 1, 1, 2, 2, 3, 5]
```

Tuple max

Gets the largest element in the tuple.

```
t = (1, 2, 1, 2, 3, 5, 1)
print(max(t)) # 5
```

Tuple min

Gets the smallest element in the tuple

```
t = (1, 2, 1, 2, 3, 5, 1)
print(min(t)) # 1
```

Tuple Sum

Returns the sum of elements in the tuple.

```
t = (1, 2, 1, 2, 3, 5, 1)
print(sum(t)) # 15
```

Sets

A set is an unordered collection of items. Every element is unique (no duplicates). It is mutable, so we can add or remove items from set. Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Set Creation

```
s = {1, 2, 3}

# set doesn't allow duplicate items
s = {1, 2, 1, 3, 4}
print(s) # {1, 2, 3, 4}

# we can make set from a list
s = set([1, 2, 3, 1])
print(s) # {1, 2, 3}

# initialize an empty set
s = set()
```

Add element to a set

We can add single element using add() method.

We can add multiple element using update() method.

```
s = {1, 3}

print(s[1]) # TypeError : 'set' object doesn't support indexing

# add single element
s.add(2) # {1, 2, 3}

# add multiple element
s.update([4, 5, 6]) # {1, 2, 3, 4, 5, 6}
```

Remove element from set

A particular item can be removed from sets using discard() and remove() method.

```
s = {1, 2, 3, 4, 5}

s.discard(4) # {1, 2, 3, 4}
s.remove(2) # {1, 3, 4}

# difference between remove and discard
s.remove(7) # KeyError, 7 does not exist in the set
s.discard(7) # {1, 2, 3, 4, 5}, no error

# remove random elements from set using pop
s.pop()

# remove all the items from the set
s.clear()
```

Python Set Operations

```
s1 = {1, 2, 3, 4, 5}
s2 = {2, 4, 6, 7, 8}

# union of two sets using | operator
print(s1 | s2) # {1, 2, 3, 4, 5, 6, 7, 8}

# another way of union
print(s1.union(s2))

# intersection of two sets using & operator
print(s1 & s2) # {2, 4}

# another way of intersection
print(s1.intersection(s2))
```

> **Set Difference** - Set of elements that are only in s1 but not in s1

```
print(s1 - s2) # {1, 3, 5}

# another way for set difference
print(s1.difference(s2))
```

> **Symmetric Difference** - Set of elements in both s1 and s2, except those that are common in both.

```
print(s1 ^ s2) # {1, 3, 5, 6, 7, 8}

# another way for symmetric difference
print(s1.symmetric_difference(s2))

# check x is subset of y
x = {1, 2}
y = {1, 2, 3, 4}

print(x.issubset(y)) # True
print(y.issubset(x)) # False
```

Frozen Sets

Frozen sets are immutable sets. Theys are created using the function frozenset().

> Sets being mutable are unhashable, so they can't be used as a dictionary keys. Frozen sets are hashable, so they can be used as keys to a dictionary.

This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union().

```
fs = frozenset([1, 2, 3, 4, 5])

fs.add(6) # AttributeError
print(fs[1]) # TypeError
```

Dictionary

Dictionary is an unordered collection of items. A dictionary has a key: value pair.

Dictionary Creation

```
# empty dictionary
my_dict = {}

#dict with integer keys
my_dict = {1: 'abc', 2: 'xyz'}

# dict with mixed keys
my_dict = {'name': 'John', 1: ['abc', 'xyz']}

# creating dict using dict()
my_dict = dict()

# create dict with list of tuples
my_dict = dict([(1, 'abc'), (2, 'xyz')])
```

Dictionary Access

```python
my_dict = {'name': 'John', 'age': 27, 'address': 'USA'}

# get name
print(my_dict['name']) # John

print(my_dict['degree']) # KeyError

# another way of accessing key
print(my_dict.get('address')) # USA

# if key is not present, it will give None using get()
print(my_dict.get('degree')) # None
```

Add or Modify Elements in Dictionary

```python
my_dict = {'name': 'John', 'age': 27, 'address': 'USA'}

# update name
my_dict['name'] = 'Raju'

# add a new key
my_dict['degree'] = 'M.Tech'
```

Delete or Remove Element from Dictionary

```python
my_dict = {'name': 'John', 'age': 27, 'address': 'USA'}

# remove a particular element
print(my_dict.pop('age')) # 27

# remove an arbitrary key
my_dict.popitem()

# delete a particular key
del my_dict['name']

# remove all items
my_dict.clear()

# delete dictionary
del my_dict
```

Dictionary Methods

1. **copy()** - Returns a copy of the dictionary.

```
copy_dict = my_dict.copy()
```

2. **fromkeys()** - Returns a dictionary with specified keys and value.

```
subjects = {}.fromkeys(['Math', 'English', 'Science'], 0)
print(subjects) # {'Math': 0, 'English': 0, 'Science': 0}
```

3. **items()** - Returns a list containing a tuple for each key value pair.

```
subjects = {'Math': 0, 'English': 0, 'Science': 0}
print(subjects.items())
# dict_items([('Math', 0), ('English', 0), ('Science', 0)])
```

4. **keys()** - Returns a list containing the dictionary's keys.

```
subjects = {'Math': 0, 'English': 0, 'Science': 0}
print(subjects.keys())
# dict_keys(['Math', 'English', 'Science'])
```

5. **values()** - Returns the list of all the values in the dictionary.

```
subjects = {'Math': 0, 'English': 0, 'Science': 0}
print(subjects.values())
# dict_values([0, 0, 0])
```

Dictionary Comprehension

```
d = {1: 'a', 2: 'b', 3: 'c'}
for pair in d.items():
    print(pair)
# (1, 'a')
# (2, 'b')
# (3, 'c')

# creating a dict with only pairs where the value is larger than 2
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
new_d = {k:v for k, v in dic.items() if v > 2}
print(new_d)
# 'c': 3, 'd': 4}
# we can also perform operations on key value pairs
d = {k + 'c': v ** 2 for k, v in dic.items() if v > 2}
print(d)
# {'cc': 9, 'dc': 16}
```

Strings

A string is a sequence of characters.

Strings can be created using single or double quotes. Triple quotes are used in python to create a multiline strings and docstrings.

```
mystring = 'Hello'

mystring = "Hello"

mystring = '''Hello'''
```

Access character in a string

We can access individual characters using indexing and a range of characters using slicing.

```
print(mystring[0]) # H

print(mystring[-1]) # o

print(mystring[2:4]) # ll

print(mystring[12]) # IndexError

print(mystring[1.5]) # TypeError
```

Change or Delete a string

Strings are immutable. We can simply reassign different strings to the same name.

```
mystring[4] = 'D' # TypeError

# delete the string
del mystring
```

String Operations

1. **Concatenation** - The '+' can be used to concatinate string. The '*' can be used to repeat the strings for a given number of times.

   ```
   s1 = "Hello"
   s2 = "World"

   print(s1 + s2) # HelloWorld

   print(s1 * 3) # HelloHelloHello
   ```

2. **Iterating through string**

   ```
   string = "Hello World"
   count = 0
   for l in string:
       if l == 'o':
           count += 1
   print(count, 'o letters found') # 2 o letters found
   ```

3. **Membership Test**

   ```
   print('l' in string) # True
   ```

String Methods

1. **lower()**

   ```
   print(string.lower()) # hello world
   ```

2. **upper()**

   ```
   print(string.upper()) # HELLO WORLD
   ```

3. **join()**

   ```
   print(string.split(' ')) # ['Hello', 'World']
   ```

4. **split()**

```
print('-'.join(string)) # H-e-l-l-o- -W-o-r-l-d
```

5. **find()**

```
print(string.find('lo')) # 3
```

6. **replace()**

```
print(string.replace('World', 'Earth')) # Hello Earth
```

Python Functions

Function is a group of related statements that perform a specific task. It avoids repetition and makes code reusable.

Syntax

```
# defining a function
def print_name(name):
    print("Hello" + str(name))

# calling a function
print_name("John") # This will print 'John'
```

Return Statement

Return statement is used to exit a function and go back to the place where it was called.

1. return Statement can contain an expression which gets evaluated and the value is returned.
2. if there is no expression in the statement or the return statement itself is not present inside a function, then the function will return None object.

```
def get_sum(lst):
    _sum = 0
    for num in lst:
        _sum += num
    return _sum

s = get_sum([1, 2, 3, 4])
print(s) # 10
```

Scope and Life Time of Variables

- Scope of a variable is the portion of a program where the variable is recognized.
- Variable defined inside a function is not visible from outside. Hence, they have a local scope.
- Lifetime of a variable is the period throughout which the varibale exists in the memory.
- The lifetime of a variable inside a function is as long as the function executes.
- Variables are destroyed once we return from the function.

```
global_var = "This is global var"
def test_scope():
    local_var = "This is local variable"
    print(local_var)
    print(global_var)

test_scope()
# This is local variable
# This is global var

print(global_var) # This is global var

print(local_var) # NameError: name 'local_var' is not defined
```

Types of function in Python

1. Built-in Function
2. user-defined function

Built-in Functions

- **abs()** - Find the absolute value

  ```
  num = -100
  print(abs(num)) # 100
  ```

- **all()** - This will return 'True' if all the elements in the iterable are true, else will return 'False' if any of the element in the iterable is false.

  ```
  lst = [1, 2, 3, 4]
  lst2 = [0, 1, 2]
  lst = []

  print(all(lst)) # True
  print(all(lst2)) # False
  print(all(lst3)) # True, empty list are always true
  ```

- **dir()** - Returns the list of valid attributes of the object.

```
numbers = [1, 2]
print(dir(numbers))
# ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- **divmod()** - Takes two numbers and returns a pair of numbers(tuple) consisting of their quotient and remainder.

```
print(divmod(9, 2)) # (4, 1)
```

- **enumerate()** - Adds counter to an iterable and returns it.

```
numbers = [1, 2, 3, 4]
for index, num in enumerate(numbers):
    print("index {0} has value {1}".format(index, num))

# index 0 has value 1
# index 1 has value 2
# index 2 has value 3
# index 3 has value 4
```

- **filter()** - Constructs an iterator from elements of an iterable for which a function return true.

```
def print_positive_numbers(num):
if(num > 0):
    return num

numbers_list = range(-10, 10)

positive_numbers = list(filter(print_positive_numbers, numbers_list))
print(positive_numbers)
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **isinstance()** - Checks if the object (first argument) is an instance or subclass of classinfo class (second argument).

```
lst = [1, 2, 3, 4]
print(isinstance(lst, list)) # True
```

```
lst = (1, 2, 3)
print(isinstance(lst, list)) # False, as lst is a tuple
```

- **map()** - Applies a function to all the element in a list.

  ```
  numbers = [1, 2, 3, 4]

  def num_square(num):
      return num**2

  squares = list(map(num_square, numbers))
  print(squares) # [1, 4, 9, 16]
  ```

- **reduce()** - Performs some computation on a list and returns the result.

  ```
  from functools import reduce

  numbers = [1, 2, 3, 4]
  def multiply(x, y):
      return x*y

  product = reduce(multiply, numbers)
  print(product) # 24
  ```

User Defined Functions

Functions that we define ourselves to do a specific task are referred as user-defined functions. If we use functions written by others in the form of library, it can be termed as library functions.

Advantages

1. It helps to decompose a large program into small segments which make program easy to understand, maintain and debug.
2. If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

Function Arguments

Variables that we pass into a function.

```
def greet(name, message):
    print("Hello {0}, {1}".format(name, message))

# name and message are arguments of function greet
```

Different forms of Arguments

1. **Default Arguments** - We can provide a default value to an argument by using the assignment operator (=)

```
def greet(name, message="Good Morning"):
    print("Hello {0}, {1}".format(name, message))

greet("John"); # Hello John, Good Morning
greet("John", "Good Night") # Hello John, Good Night
```

> Once we have a default argument, all the arguments to its right must also have default values

```
def greet(message="Good Morning", name):
    print("Hello {0}, {1}".format(name, message))

greet("John") # SyntaxError
```

2. **Keyword Arguments** - kwargs allow you to pass keyworded variable length of arguments to a function.

```
def greet(**kwargs):
    if kwargs:
      print("Hello {0}, {1}".format(name, message))

greet(name="John", msg="Good Day") # Hello John, Good Day
```

3. **Arbitrary Arguments** - Sometimes, we don't know in advance the number of arguments that will be passed into a function. This kind of situation can be handled using arbitrary arguments.

```
def greet(*names):
    for name in names:
        print("Hello {0}".format(name))

greet("John", "Jane", "Joe")
# Hello John
# Hello Jane
# Hello Joe
```