The **open** method returns a file object. ***Syntax***: open(name[, mode]). where *mode* can be 'r' (read), 'w'(write) or 'a'(append). The default mode is 'r'.  
The **close** method closes an opened file object.

# Create a file called 'tmp.txt' for writing, and close the file after that.

# open a file

f = open('tmp.txt', 'w') # replace 'filename' and 'mode'

# write to file

f.write('hello')

# close a file

f.close()

# Create a function that appends the name and email to the end of a named file.

def addEmail(filename, name, email):

f = open(filename, 'a') # replace the mode

f.write('{}\n'.format(name))

f.write(email)

# Append name and email, each record should end with '\n'.

f.close()

# Write a function that returns 4 lists given a postive number.

def createLists(num):

x=list(range(1,num+1))

z=list(range(num,0,-1))

y=list(range(-1,-num-1,-1))

r=list(range(-num,0))

return (x,z,y,r)

# Write a function that returns the output as shown in the above examples.

def diff(a, b):

x=abs(a)-abs(b)

return abs(x)

# Write a function that returns a string of characters based on a list of ASCII codes.

def toString(alist):

numberlist=[]

for number in alist:

numberlist.append(chr(number))

return ''.join(numberlist)

# Write a function that capitalizes the first character of each word.

def capitalize(phrase):

x=phrase.title()

return x

# Write a function that returns minimum and maximum values of a list containing numbers in integer and string formats.

def mixedList(mlist):

newlist=[]

for element in mlist:

newlist.append(int(element))

x=min(newlist)

y=max(newlist)

return (x,y)

# Complete the code below so that the outputs are as shown in the examples above.

def lowercase(x):

if x.islower():

return True

else:

return False

def fn1(word):

return filter(lowercase, word)

def fn2(word):

return filter(lambda x:x[0] in '1234567890' , word)

# Write a factorial function using the 'reduce' function

def factorial(num):

start = 1

return reduce(lambda x,y:x\*y ,range( 1,num+1), start)

# Write down the values of variables b and c, with the output as shown in the above example.

b =('google')

c = ('e','n','g','i','n','e',)

# Write a function that returns the total size of the arguments.

# Note: \*args denotes a variable argument list, represented by a tuple.

def totSize(\*args):

total=0

for i in args:

total+=len(i)

return total

class Person:

def \_\_init\_\_(self, weight, height):

self.weight = weight

self.height = height

# Create a Person object with weight = 60, height = 1.7

p =Person(60,1.7)

It is possible to change the behaviour of built-in opertators with special methtods. For example, the '+' operator can be implemented with the \_\_add\_\_ method. Define a Point class that supports operator overloading for the '+' and '-' operators.

class Point:

"A class implementation of 2-Dimensional point."

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return '(%d, %d)' % (self.x, self.y)

def \_\_add\_\_(self, other):

X=self.x+other.x

Y=self.y+other.y

return (X,Y)

def \_\_sub\_\_(self, other):

X=self.x-other.x

Y=self.y-other.y

return (X,Y)

Inheritance allows the reuse of code, by implementing a parent-child relationship between classes. Create 2 derived classes Student and WorkingAdult from the base class Person.

class Person:

"""A base class"""

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

class Student(Person):

"""A derived class for Student"""

def \_\_init\_\_(self, name, age, school):

Person.\_\_init\_\_(self, name, age)

self.school=school

def introduce(self):

print("my name is {0}.I am {1} old.I am studying at {2} ".format(self.name,self.age,self.school))

class WorkingAdult(Person):

"""A derived class for WorkingAdult"""

def \_\_init\_\_(self, name, age, job):

Person.\_\_init\_\_(self,name,age)

self.job=job

def introduce(self):

print("my name is {0}.I am {1} old.I am {2} ".format(self.name,self.age,self.job))