EXERCISE-6

OOPS

Object Oriented Programming (OOP)

OOP is a term used to describe programming in which data and functionalities are wrapped in objects. This is different from procedural programming (modular programming is sub type of it). OOP becomes necessary to develop programs when working on big and complex applications. Two essential concepts in OOP are Class and Object. Think of class and object as a car factory (car factory = class, car=object), a shoe factory (shoe factory = class, shoe = object) or library (library = class, book = object). Let's take the library as an example. Books can have (attributes) like name, code, authors, types. Books can be accessed, pulled out, scanned, sorted, etc. Some books can only be accessed by staff such as encyclopedia, but other books are on open shelves and can be accessed by public. This is exactly how objects can be described inside a class they belong to in a programming language such as Python. Objects have attributes and can be accessed through (methods). Objects can have variables which either accessed throughout the class (class variables) or variables accessed only within methods where they are defined (object variables).

Generally speaking, an object is defined by a class. A class is a formal description of how an object is designed, i.e. which attributes and methods it has. These objects are called instances as well.

Class: A framework that programmers define to which objects belong. It defines attributes to characterize objects. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Instance: refers to an object of a particular class. An object *obj* that belongs to a class Factory is an instance of the class Factory.

Method: A special kind of function that is defined in a class definition.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Creating classes in Python:

To create a new class, we define a class with a name. Inside the class, we define data, methods, attribute.

Syntax:

class className ([superclasses]):

'class documentation'

block of indented statements {here, you define initialization and user-defined methods and data memebers}

```
class Vehicle (Type):

‘ all classes in OOP are subclasses of the superclass Type, so we do not have to include the superclass in the definition’

        pass

class Car (Vehicle):

        pass
```

Creating instances:

We create an instance (object) by calling the class name and passing arguments that the class initialization method accepts.

# this line will create an instance (object) of class Car called car1

car1 = Car (arg_1*,…*arg_n*)*

Attributes:

Attributes are features or characteristics of instances or classes like name, age, brand, salary, etc. They are created inside the class definition after creating an instance. We can access attributes by separating the instance name and the attribute with a dot (.). We can also access instance attributes using special functions.

- **getattr(obj, name)** : accessing (retrieving) instance attributes and their values.
- **hasattr(obj,name)** : testing if an attribute exists or not for the instance obj.
- **setattr(obj,name,value)** : to setting a new attribute with a value. If attribute does not exist, it will be created.
- **delattr(obj, name)** : to delete an attribute.

Built-in class attributes can be accessed as follows:

- **_dict :** accessing class dictionary.
- **__doc :** accessing class documentation string.
- **__name :** accessing the class name.
- **__module :** accessing the module name in which the class is defined.

- **__bases :** accessing the base classes of the current class in the order of their occurrences.

```
#creating a class and superclass

class Vehicle:

	pass

class Car(Vehicle):

	' this subclass has a superclass'

	pass

#accessing built-in class attributes

print ("Car._doc_:", Car._doc_)

print ("Car.__name__:", Car.___name__)

print ("Car._module_:", Car._module_)
print ("Car._bases_:", Car._bases_)
```

```
#accessing attributes

class Car:

	' this subclass has a superclass'

	pass

x=Car()

# creating three attributes for instance x

x.make="Toyota"

x.color="white"

x.year="2001"


print (x.color)

print (x._dict_) # accessing the class dictionary
```

```
Car.model="Sienna"

print (x.model) # will print Sienna

x.model="Camry"

print (Car.model) #will print Sienna

print (getattr(x, 'model')) #another way to access attributes

setattr(x, 'fuel','diesel') #creating a new attribute associated with a new value

print (getattr(x, 'fuel'))

print (x._dict_)

delattr (x,'fuel') # deleting the attribute fuel from x and the class dictionary

print (x._dict_)

print (hasattr(x, 'fuel'))# checking if the attribute fuel exists in x
```

The __init___methods (double underscores before and after init):

The __init___is an important method in Python and is usually called immediately after the class definition. The name is fixed and can not be changed. This method is similar to constructors in Java and C#. It is called when an instance is created so its job is to initialize that instance with data we want.

Self: class methods in Python have special arguments added as a first arg in the argument list. We do not pass a value to it as Python provides it. The (self) is there to refer to the object itself. This is important so we know to which object the initialization was triggered (there could be multiple objects are initialized using the same __init__). It is recommended to keep the name as (self), just like Java which uses (this).

```
class Library:

        def_init_(self, name):

                self.name = name

        def say_hi(self):

                print('Hello, you are in’, self.name, ‘library’)

p = Person(‘Fairview’)

p.say_hi()
```

```
# working with __init___to initialize an instance.

class Library:

        def init (self, name): # this is a special method, self is a reference to the instance p of class
        #Library which has a name Fairview.

                self.name = name

        def greeting(self): # this is a normal method which is defined to print out a greeting.

                #if self.name:

                        print('Hello, you are in', self.name, 'library')

                #else:

                        #print ('forgot to provide the library name')

p = Library('Fairview') # creating an instance p and initialize it with name Fairview by calling  __init___
#method.

p.greeting() # calling the user-defined method, greeting.

q = Library('')

q.greeting()
```

Other normal methods can also be defined and called inside a class.

Class variables and instance (object) variables:

Variables in a class are bound to the namespaces of that class and their instances which means these variables are only valid (visible) in the context of the class and the instances. There are *class variables* and *instance variables*. The former is defined outside methods but inside a class and can be accessed by all instances. A change that happens to a class variable can be seen by other instances in that class. The later are possessed by an individual instance inside a method of a class so it is not seen outside that method so it is OK to have two instance variables with the same name in two different methods.

Class variables are accessed as follows: *classname.classVariable*

Instance variables are accessed as follows: *instanceName.instanceVariable*

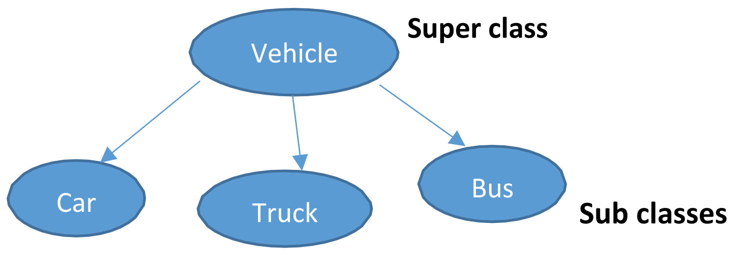
```
#looking into class variable and instance variables

class Student:

        'Common base class for all students'

        stuCounter = 0 # this is a class variable
```

```
    def_init_(self, name, mark):
        self.name = name
        self.mark = mark
        Student.stuCounter += 1



    def dispStudent(self):
        print ("Student : ", self.name, ", Mark: ", self.mark)



#creating a first object of Student class
student1 = Student("John", 91)
#creating a second object of Student class
student2 = Student("Diana", 94)

student1.dispStudent()
student2.dispStudent()
print ("Total number of students %d" % Student.stuCounter)
```

Inheritance:

Inheritance allows us to use existing class (super classes) to define new classes (sub classes) where we can re-use predefined methods in the existing class without the need to re-writing them inside sub classes.

The sub class inherits the attributes of its super class, and we can use them as if they were defined in the sub class. A sub class can also override methods from the super class.



```
class ShoeFactory:


   def init (self, kind, code):
      self.shoetype = kind
      self.shoecode = code


   def description(self):
      return (self.shoetype + " : " + self.shoecode)


class Shoe(ShoeFactory):


   def init (self, kind, code, color):
      ShoeFactory. init (self, kind, code)
      self.shoecolor = color


   def getShoe(self):
      return (self.description() + " : " +  self.shoecolor)


x = ShoeFactory("boots", "111111")
y = Shoe("sport", "555555", "brown")
print(x.description())
print(y.getShoe())
```

Try the following inheritance program ?

```
class SchoolMember:
    'Represents any school member.'
    def init ( , ,  ):
        self.name =
        self.age =
        print('(Initialized SchoolMember: {})'.format(self.name))
    def tell( ...  ):
        'Tell my details.'
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")
class Teacher(       ):
    'Represents a teacher.'
    def init (self, name, age, salary):
        SchoolMember. init (self, name, age)
        self.salary = ........
        print('(Initialized Teacher: {})'.format(self.name))
    def tetell(self):
        self.tell()
        #SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))


class Student(        ):
    'Represents a student.'
    def init (, , , ):
```

```
                    ...................
                    ...................
                    ...........................................
          def sttell( ..... ):
                    ................
                    .................

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)
# prints a blank line
print()
t.tetell()
s.sttell()
```

Overriding:

We can override the super class methods. The reason for overriding is because you may want to implement a special or different functionality in your subclasses.

```
class SchoolMember:
          'Represents any school member.'
          def_init_(self, name, age):
                    self.name = name
                    self.age = age
                    print('(Initialized SchoolMember: {})'.format(self.name))
          def tell(self):
                    'Tell my details.'
                    print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")
class Teacher(SchoolMember):
          'Represents a teacher.'
```

```
        def init (self, name, age, salary):

                SchoolMember. init (self, name, age) # overriding SchoolMember init
                method

                self.salary = salary

                print('(Initialized Teacher: {})'.format(self.name))

        def tell(self): # overriding SchoolMember tell method

                SchoolMember.tell(self)

                print('Salary: "{:d}"'.format(self.salary))
```

Data Hiding:

Programmers may need to hide attributes so they are invisible by outsiders (can not be accessed directly as class variables). To do so, they define these attributes with two underscore prefix. Data encapsulation and Data Hiding terms are often used interchangeably. However, the former refers to the fact that data are bound to methods. Data abstraction is a broader term which includes both.

```
# data hiding

class MyCounter:

        __hidenCount=0

        def init (self,name="hello"):

                self.name=name

                print("initializing the object", self.name)

        def count(self):

                self._hidenCount += 1

                print (self._hidenCount)


counter = MyCounter()

counter.count()

counter.count()

#print (counter._hidenCount) # this will generate an error, hidenCounter can not be seen
```

```
print (counter._MyCounter_hidenCount)
```

Deleting objects (destructors):

When no references to an object, Python deletes that object to free memory spaces. Object reference decreases when the number of pointers to that object decreases. Assigning a references to another object also decreases the object reference count. A special method can be used to delete an object *(__del__)* which is called destructor which is invoked when an object is going to be deleted (using *del* function or implicitly when there is no more reference to the object)

```
# deleting (destroying) an object
class AnObject:
  def_init( self, x=0, y=0):
    self.x = x
    self.y = y
  def_del_(self):
        print ("object is deleted")
ob1 = AnObject()
ob2 = ob1
ob3 = ob1
print (id(ob1), id(ob2), id(ob3)) # prints the ids of the objects
del ob1
del ob2
del ob3
```

```
#Author: Swaroop C H, A Byte of Python
class Person:
        'Represents a person.'
        population = 0
        def_init_(self, name):
```

```
        'Initializes the person\'s data.'
        self.name = name
        print ('(Initializing %s)' % self.name)
        # When this person is created, he/she
        # adds to the population
        Person.population += 1
    def_del_(self):
        'I am dying.'
        print ('%s says bye.' % self.name)
        Person.population -= 1
        if Person.population == 0:
            print ('I am the last one.')
        else:
            print ('There are still %d people left.' % Person.population)
    def sayHi(self):
        'Greeting by the person.Really, that\'s all it does.'
        print ('Hi, my name is %s.' % self.name)
    def howMany(self):
        'Prints the current population.'
        if Person.population == 1:
            print ('I am the only person here.')
        else:
            print ('We have %d persons here.' % Person.population)
swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()
print ('\n')
kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

print('\n')

swaroop.sayHi()

swaroop.howMany()

del swaroop

del kalam
```