Object Oriented Programming with Python Language

Dr. Ami Tusharkant Choksi
Associate Professor,
Computer Engineering Department,
C.K.Pithawala College of Engineering & Technology, Surat, Gujarat State, India.
email: ami.choksi@ckpcet.ac.in

April 25-May 01, 2020

Classes I

- Object Oriented Programming Characteristics:
- Class-A blueprint of something
- Object-instance of class
- Access specifier: public, private, protected
- Abstraction: hiding details at hardware level
- Encapsulation: hiding data from being accessed or accessed with specific mode only
- Inheritance: extending the properties of a class
- Polymorphism:Polymorphism is the ability to leverage the same interface for different underlying forms such as data types or classes. Polymorphism allows for flexibility and loose coupling so that code can be extended and easily maintained over time. This tutorial will go through applying polymorphism to classes in Python.
- Class is itself an object of type type

Classes II

- Class definition creates an object
- Class methods have type method_descriptor
- Operations for Class
- create instance of a class is Instantiation
- textttx = Calculator()
- dot(.) is used to access attributes and methods
- Constructor is defined as `def __init__(self, value):`
- Destructor is defined as `def __del__(self):`
- `self` is required in each method of the class, irrespective of constructor and destructor
- self represents the current object
- Class data without any _(underscore) is public

Classes III

- To make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.
- To make an instance variable private is to add a prefix __ (double underscore) to it.The members of a class which are declared private are accessible within the class only, private access modifier is the most secure access modifier.

Listing 1: Class Definition

```
#Class Definition}
class Calculator(object):
        def add(self, no1, no2):
                return (no1+no2)

#make object of a class
ob = Calculator()
print(ob.add(6,4))#10
```

Classes IV

Listing 2: Initializing class member in method

```
1  #Initializing class member in method
2  class A:
3          a=100
4          def first(self, a, b):
5                  #self is not the keyword in python
6                  self.a=a
7                  print(a)#10
8                  print(b)#20
9
10 r = A()
11 r.first(10,20)
```

Listing 3: Initializing class member in method:Box example

```
1  #box with two attributes width and height, depth calculate volume o
2  class Box:
3          width=0
4          height=0
5          depth=0
6          def volume(self, width, height, depth):
7                  self.width=width
8                  self.height=height
9                  self.depth=depth
10                 return(width*height*depth)
11
12 r = Box()
13 v=r.volume(10,20,30)
14 print(v)
15 #O/P:6000
```

Classes VI

- this.width in Java language here in Python as self.width, as in above code
- It allows ONLY one constructor
- Default arguments to mimic multiple constructors
- Called str(object) and built-in functions format() and print()
- Nicely printable string representaion of object
- `def __str__(self):` is used to define String Representaion of Object
- Call Only print(object)

Listing 4: String Representaion of Object

```
1  #String Representaion of Object
2  #box with two attributes width and height, depth calculate volume o
3  class Box:
4          width=0
5          height=0
6          depth=0
7          def volume(self, width, height, depth):
8                  self.width=width
9                  self.height=height
10                 self.depth=depth
11                 return(width*height*depth)
12         def __str__(self):
13                 return str(self.width)+":"+str(self.height)+":"+str(se
14
15 r = Box()
16 v=r.volume(10,20,30)
```

Classes VIII

```
17 print(v)
18 print(r)
19 #O/P:6000
20 #10:20:30
```

Special Methods I

- Special Methods __lt__ are used for <,>,<=,>=, !=

Listing 5: Special Methods

```
class Comparables:
        c =0
        def __eq__(self, other):
                return self.c==other
        def __ne__(self, other):
                return self.c!=other
        def __gt__(self, other):
                return self.c>other
        def __ge__(self, other):
                return self.c<other
        def __le__(self, other):
                return other.c<self

v = Comparables()
```

Special Methods II

```
v.c=3
j=Comparables()
j.c=4
print(v==j)#False
print(v!=j)#True
```

Listing 6: Special Methods

```
class A:
        def __init__(self, i = 0, j = 0):
                self.i = i
                self.j = j

def __str__(self):
        return "some string"

def __eq__(self, other):
        return self.i * self.j == other.i * other.j
```

Special Methods III

```
11
12 def main():
13         x = A(4, 3)
14         y = A(6, 2)
15         print(x == y)
16 if __name__== "__main__" :
17     main()#True
```

- __add such methods are remaining

Constructor and Destructor I

- __init__ $self$:is used to define Constructor in Python
- __del__ $self$:is used to define Destructor in Python

Listing 7: Constructor:

```
1
2  #constructor
3  class Box:
4          def __init__(self):
5                  print("Inside Constructor")
6          width=0
7          _height=0
8          __depth=0
9
10         def volume(self, width, height, depth):
11                 self.width=width
12                 self.height=height
13                 self.depth=depth
```

Constructor and Destructor II

```
14                return(width*height*depth)
15 r = Box()
16 #O/P: Inside Constructor
```

Listing 8: Constructor & Destructor

```
1
2  #constructor & destructor
3  class Box:
4          #constructor
5          def __init__(self):
6                  print("in constructor")
7          #destructor
8          def __del__(self):
9                  print("In destructor")
10
11 r = Box()
12 #explicit calling of destructor
```

Constructor and Destructor III

```
13 del r
14 #O/P:
15 #In constructor
16 #In destructor
```

Class Example with Constructor, getter and setter methods I

Listing 9: Class Example with Constructor, getter and setter methods

```
1  #Class Example with Constructor, getter and setter methods
2  #Class Example with Constructor, getter and setter methods
3  import datetime
4  class Person(object):
5          #name='temp'
6          def __init__(self, name):
7                  self.name = name
8                  try:
9                          lastblank = name.rindex(' ')
10                         self.lastname = name[lastblank+1:]
11                 except:
12                         self.lastname = name
13                         self.birthdate = None
14
15         def getName(self):
```

Class Example with Constructor, getter and setter methods II

```
16              return self.name
17
18      def getLastName(self):
19              return self.lastname
20
21      def setBirthday(self, birthdate):
22              self.birthdate = birthdate
23
24      def getAge(self):
25
26              if self.birthdate ==None:
27                      raise ValueError('Birthday not set for '+self.n
28              return (datetime.date.today() -self.birthdate).days
29
30      def __lt__(self, other):
31              if self.lastname ==other.lastname:
```

Class Example with Constructor, getter and setter methods III

```
32                      return self.name < other.name
33                  return self.lastname < other.lastname
34          def __str__(self):
35                  return self.name
36
37  ob = Person("ami choksi")#choksi
38  ob.setBirthday(datetime.date(1960,9,22))
39  print("ob : ",ob)#ami choksi
40  print("name: ",ob.lastname)#choksi
41  print("birthday", ob.birthdate)#None
42
43  cat = Person('Gar field')
44  bat = Person('Bruce wayne')
45  cat.setBirthday(datetime.date(1978,6,19))
46  bat.setBirthday(datetime.date(1939,5,27))
47  print("Cat age : ",cat.getAge())#14645
```

Class Example with Constructor, getter and setter methods IV

```
48 print("Bat age : ",bat.getAge())#28913
49
50 #
51 #ob : ami choksi
52 #name: choksi
53 #birthday 1960-09-22
54 #Cat age : 15284
55 #Bat age : 29552
```

Inheritance I

- Inheritance is the capability of one class to derive or inherit the properties from some another class.
- *Object* of any class can be bound to a variable, list, inserted as a value in dictionary.
- BankPerson inherits the attributes of Person
- BankPerson(subclass) overrides some attributes from Person(superclass)
- subclass say Staff of Person can be written as
  `class Staff(Person):`
- Method of class Person and Staff can be called from main as
  `Person().printPerson()`
  `Staff().printPerson()`

Inheritance II

Listing 10: Inheritance:Person->Staff

```
1  class Person:
2          def getInfo(self):
3                  return "Person's getInfo is called"
4
5          def printPerson(self):
6                  print(self.getInfo())
7
8  class Staff(Person):
9          def getInfo(self):
10                 return "Staff's getInfo is called"
11
12 def main():
13                 Person().printPerson()
14                 Staff().printPerson()
15
16 main()
```

Inheritance III

```
17 #Output: Person's getInfo is called
18 #Staff's getInfo is called"
```

- Person->BankPerson->Customer
- *pass* is a keyword requires no definition just now, it may be implemented later.

Listing 11: Class definition without implementation:pass

```
1 class NormalCustomer(Customer):
2         pass
```

Listing 12: isinstance checking

```
1  #isinstance checking
2  class box:
3          def isBOX(self):
4                  return isinstance(self,box)
5
6  o = box();
7  #direct checking
8  print(isinstance(o,box))#True
9  print(isinstance(o,Person))#False
10
11 #checking through a method
12 print(o.isBOX())#True
```

Inheritance V

Listing 13: isinstance revisited

```
1  class box:
2          pass
3  class rupee:
4          pass
5
6  def isBOX(self):
7          if isinstance(self,rupee):
8                  return True
9          elif isinstance(self, box):
10                 return False
11
12 o = box()
13 ob = rupee()
14 print(isBOX(o))#False
15 print(isBOX(ob))#True
```

Switch case I

Listing 14: Switch case demo

```
def switch_demo(argument):
        switcher = {1: "January",
                2: "February",
                3: "March",
                4: "April",
                5: "May",
                6: "June",
                7: "July",
                8: "August",
                9: "September",
                10: "October",
                11: "November",
                12: "December"
}
        #Syntax : dict.get(key, default=None)
        print(switcher.get(argument,"Invalid Month"))
```

Switch case II

```
17
18 switch_demo(1)#January
19 switch_demo(9)#September
20 switch_demo(15)#Invalid Month
```

Public, Private, Protected I

- Class data without any _(underscore) is public
- Class member with double __(underscores) is private
- Class member with single _(underscore) is protected
- For private members, we need to have getter and setter methods

Listing 15: Inheritance Student->EnggStudent and public, private, protected variables

```
1
2 #Inheritance Student->EnggStudent and
3 #use of public, private, protected variables
4
5 class Student:
6       #constructor of student
7       def __init__(self, phoneno, name,book):
8             self._phoneNo = phoneno
9             self.name = name
```

Public, Private, Protected II

```
10                self.__book=book
11
12        #for private variable getter and setter methods are requ
13        def setBook(self,book):
14                self.__book = book
15
16        def getBook(self):
17                return(self.__book)
18
19        def show(self):
20                print("Phone:",self._phoneNo)
21                print("Name:",self.name)
22                print("Book:",self.__book)
23
24 class EnggStudent(Student):
25        pass #no implementation just now, we will think later
26
27 s = Student(9999999999,"ckp","Python")
```

Public, Private, Protected III

```
28 print(s._phoneNo)#9999999999
29 #print(s.__book)##private #Error: 'Student' object has no attri
30 print(s.getBook())
31 print(s.name)#ckp
32
33 es = EnggStudent(1111111111,"ac","Java")
34
35 print("Student show")
36 s.show()
37 print("Engg Student show")
38
39 #print(es.__book)#AttributeError: 'EnggStudent' object has no a
40 es.show()
41
42 #Output:
43 #9999999999
44 #Python
45 #ckp
```

Public, Private, Protected IV

```
#Student show
#Phone: 9999999999
#Name: ckp
#Book: Python
#Engg Student show
#Phone: 1111111111
#Name: ac
#Book: Java
```

Listing 16: Call protected method from subclass

```
#call protected method from subclass
class Student:
        #constructor
        def __init__(self, phoneno, name,book):
                self._phoneNo = phoneno
                self.name = name
                self.__book = book
```

Public, Private, Protected V

```
8
9           #protected method
10          def _protectmethod(self):
11                  print(self._phoneNo,":"+self.name,":",self.__book
12
13  #subclass EnggStudent
14  class EnggStudent(Student):
15          pass
16
17  #make Student object
18  s = Student(11111,"Ami","DREAMWORLD")
19  s._protectmethod();#11111 :Ami : DREAMWORLD
20
21  #make EnggStudent object
22  es = EnggStudent(2222,"Juhi","BEAUTYWORLD")
23  es._protectmethod()#2222 :Juhi : BEAUTYWORLD
```

Listing 17: Call private method from subclass via public method

```
1
2  #call private method from subclass via public method
3  class Student:
4          #constructor
5          def __init__(self, phoneno, name,book):
6                  self._phoneNo = phoneno
7                  self.name = name
8                  self.__book = book
9
10         #private method
11         def __privmethod(self):
12                 print(self._phoneNo,":"+self.name,":",self.__book)
13
14         #calling private method from public method
15         def callprivmethod(self):
16                 self.__privmethod()
```

Public, Private, Protected VII

```
17
18 class EnggStudent(Student):
19         pass
20
21 #make Student object
22 s = Student(11111,"Ami","DREAMWORLD")
23 #s.__privmethod();
24 #AttributeError: 'Student' object has no attribute '__privmetho
25 #call private method from public method
26 s.callprivmethod()#11111 :Ami : DREAMWORLD
27
28 #make EnggStudent object
29 es = EnggStudent(2222,"Juhi","BEAUTYWORLD")
30 #es.__privmethod()
31 #AttributeError: 'EnggStudent' object has no attribute '__privm
32 es.callprivmethod()#2222 :Juhi : BEAUTYWORLD
```

Listing 18: User Defined Module

```
#User defined module one.py, imported in two.py or here.
#one.py should be available in the same directory as this file.
#Contents of the file shoould be

#Moduleone.py as follow, remove comments in
##########################################################
#def getModuleName():
# print("module :",__name__)#module name

##########################################################
#useModule.py
from Moduleone import getModuleName
def main():
        getModuleName()

if __name__=='__main__':
```

Public, Private, Protected IX

```
17          main()
18
19  #running useModule.py
20  #O/P: Module : Moduleone
```

Polymorphism I

- In literal sense, Polymorphism means the ability to take various forms.
- In Python, Polymorphism is through method overriding.
- Method overriding allows us to define methods in the child class with the same name as defined in its parent class.

Listing 19: Polymorphism-Method overriding

```
1  #run() method is available in both parent and child class
2
3  class Animal:
4          def run(self):
5                  print("ANIMAL IS RUNNING")
6
7  class Cat(Animal):
8          def run(self):
9                  print("CAT IS RUNNING")
10
```

Polymorphism II

```
11
12 #create object of Animal
13 a=Animal()
14 #call run method
15 a.run()#ANIMAL IS RUNNING
16
17 #create object of Cat
18 c=Cat()
19 #call run method
20 c.run()#CAT IS RUNNING
```

Listing 20: Call super class method using super()

```
#call super class method using super()
#run() method is available in both parent and child class
class Animal:
        def run(self):
                print("ANIMAL IS RUNNING")

class Cat(Animal):
        def run(self):
                super().run()#calling super class run()
                print("CAT IS RUNNING")


#create object of Cat
c=Cat()
#call run method
c.run()
```

Polymorphism IV

```
17 #Output:
18 #ANIMAL IS RUNNING
19 #CAT IS RUNNING
```

Method Overloading I

- Method overloading means two methods with the same name and different parameters.i.e.type and number of parameters are different.
- Two methods cannot have the same name in Python. i.e. it will consider latest method as the ONLY method of that class

Listing 21: Method Overloading

```
1  #Method overloading: method with same name and different signat
2  class Calculator:
3        def add(self, no1, no2):
4              return(no1+no2)
5        def add(self, no1, no2, no3):#latest method
6              return(no1+no2+no3)
7
8  #create object
9  c=Calculator()
10
11 #print(c.add(4,5))# add() missing 1 required positional argumen
```

Method Overloading II

```
#because it considers latest definition of add method
#if we change the sequence of methods, two arguments method wil
#so this way of method overloading is not possible in Python
```

- One way to have method overloading in Python is through *. i.e. variable no of arguments

Listing 22: Method overloading in Python

```
#Method overloading
class Man:
        def FavouriteDish(self,*arg):
                for i in arg :
                        print(i)

m=Man()
m.FavouriteDish('Gujarati','SouthIndian')#method call with 2 ar
m.FavouriteDish('Punjabi','Chienese','Dhokala')#method call wit
```

Method Overloading III

```
10 #Output:
11 #Gujarati
12 #SouthIndian
13 #Punjabi
14 #Chienese
15 #Dhokala
```

References I

1. Switch statement, https://jaxenter.com/implement-switch-case-statement-python-138315.html
2. Public, private, protected visibility modifiers, https://www.geeksforgeeks.org/access-modifiers-in-python-public-private-and-protected/
3. Public, private, protected visibility modifiers, https://www.tutorialsteacher.com/python/private-and-protected-access-modifiers-in-python
4. Public, private, protected methods, https://www.geeksforgeeks.org/__name__-special-variable-python/
5. Polymorphism, https://overiq.com/python-101/inheritance-and-polymorphism-in-python/
6. Method overloading, https://www.edureka.co/blog/python-method-overloading/