Python workshop experiments (2nd Part)

“Object-Oriented Practices with Python3”

Note: Object-Oriented Programming is very powerful in python, programmer has very less bounds & limitations, everything you can think, can happen in python, unlike other languages (C++/Java/C#), also it is very easy to use it, it follows pure object model, which we’ll discuss, without any restrictions, also we can easily check how classes & each stuff is being implemented at back-end of Python Interpreter, so be alert to enter the world of OOPs.

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> #Hi Everyone, this is for next level discussion of important programming paradigm of Python, earlier we have done discussion on structured & functional programming practices, from today onwards we will elaborate the importance & uses of Object-Oriented Practices(OOPs)
>>> #first i will explain the scoping rules of python global v/s non local
>>> #i have given scoping rules in a file named as 'global vs nonlocal.py'
>>> #actually in memory, scopes are associated with variable(names), not the object memory
>>> # as i said in python everything is an object
>>> #so firstly, we'll have a short explaination of what an object is
>>> #it is the memory having some members(data & methods) allocated in heap,  which is accessed by a name currently in stack
>>> a=5 #global a
>>> a
5
>>> def f():
	b=lambda x:x+x#twice function
	global a
	a=b

>>> a
5
>>> f()
>>> a
<function f.<locals>.<lambda> at 0x000000EB18D3E6A8>
>>> a(3)
6
>>> #so you just see that scope of a is global , but lambda function is created in local to f()
>>> b
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    b
NameError: name 'b' is not defined
>>> #see now 'b' is undefined, while lambda function still exist
>>> 
>>> #Where to apply OOP based thinking
>>> #complexity & module based management of software projects comes to an ease with OOP, 7 this is where they are highly useful,We thought about problem domain and figure out what features an object must have.
>>> #OOP is the natural way of thinking, how we relate to things around us i natural way, which makes us easy to manage & understand everything
>>> #Object-oriented programming revolves around defining and using new types.
>>> 
>>> #Object-oriented programming involves at least these phases:
```

```
>>> #1. Understanding the problem domain.
>>> #2. Figuring out what type(s) you might want.
>>> #3. Figuring out what features you want your type to have.
>>> #4. Writing a class that represents this type.
>>> #5. Testing your code.
>>> #lte us start with a function 'isinstance()'
>>> #this function returns a boolean values, that if given object/class is an
instance of class or not
>>> isinstance('abc',str)
True
>>> isinstance(2,str)
False

>>> #Python has a class called "object". Every other class is based on it.
>>> help(object)
Help on class object in module builtins:

class object
 |  The most base type

>>> isinstance(55.2, object)
True
>>> isinstance(SyntaxError, object)
True
>>> isinstance(list, object)
True
>>> isinstance(Exception, object)
True
>>> #every class in Python is derived from class object, and so every instance of
every class is an object
>>> dir(object)#it list attributes assoicted with object, see double underscores,
they are special to python
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> #these are the attributes associated with all objects
>>> #Accessing object methods
>>> #One way is to access the method through the class, and the other is to use
object-oriented syntax.
>>> str.capitalize('browning')
'Browning'
>>> 'browning'.capitalize()
'Browning'
>>> #memory model will be based on frames of variables & objects kept aside into
heap
>>> #remember that we have used math.sqrt(5), doesn't it is same as a Object-oriented
way, yes it is!
>>> import math
>>> type(math)
<class 'module'>
>>> math.sqrt(5)
2.23606797749979
>>> #you can define & import your own modules in python
>>> #discussing about main function, yes python have a main function named as
'__main__'
>>> __name__
'__main__'
>>> #shell here is running as __main__,so if you want some code to be written in
a module, which runs only when your module is running as main program, keep it into
```

```
>>> #if __name__=='__main__'
>>> #Methods belong to classes. Instance variables belong to objects. If we try to
access an instance variable as we do a method from a class, we get an error
>>> print(math)
<module 'math' (built-in)>
>>> # Classes and objects are two of programming’s power tools. They let good
programmers do a lot in very little time, but with them, bad programmers can create
a real mess. further i will introduce some underlying theory that will help you design
reliable, reusable object-oriented software.
>>> #defining class
>>> class Book:
    des='this is my class'#this is variable shared among all objects, even modified
    def __init__(self,title,authors,price):
        self.title=title
        self.authors=authors
        self.price=price
    def detail(self):
        print('Title is ',self.title,'Written by ',self.authors,'costs for
',self.price)


>>> a=Book('Theory of computation','John C. Martin',125)
>>> a.detail()
Title is  Theory of computation Written by  John C. Martin costs for  125
>>>
>>> b=Book('OS System concepts',['Silberchatz','Galvin','Greg'],200)
>>> b.detail()
Title is  OS System concepts Written by  ['Silberchatz', 'Galvin', 'Greg'] costs
for  200
>>> #Encapsulation: To encapsulate something means to enclose it in some kind of
container.
>>> #hiding the details of exactly how things work together
>>>
>>> #Polymorphism: Polymorphism means “having more than one form.” In programming,
it means that an expression involving a variable can do different things depending
on the type of the object to which the variable refers.
>>>
>>> #Inheritance: third fundamental feature of object-oriented programming called
inheritance, which allows you to recycle code in yet another way.
>>>
>>> #whenever you create a class, you are using inheritance: your new class
automatically inherits all of the attributes of class object, much like a child
inherits attributes from his or her parents
>>> class Member:
    """ A member of a university. """
    def __init__(self, name, address, email):
        """ (Member, str, str, str) -> NoneType
        Create a new member named name, with home address and email address.
        """
        self.name = name
        self.address = address
        self.email = email

>>> class Faculty(Member):
    """ A faculty member at a university. """
    def __init__(self, name, address, email, faculty_num):
        """ (Member, str, str, str, str) -> NoneType
        Create a new faculty named name, with home address, email address,
        faculty number faculty_num, and empty list of courses.
        """
        super().__init__(name, address, email)
        self.faculty_number = faculty_num
        self.courses_teaching = []

>>> class Student(Member):
    """ A student member at a university. """
    def __init__(self, name, address, email, student_num):
        """ (Member, str, str, str, str) -> NoneType
        Create a new student named name, with home address, email address,
        student number student_num, an empty list of courses taken, and an
        empty list of current courses.
        """
        super().__init__(name, address, email)
        self.student_number = student_num
        self.courses_taken = []
        self.courses_taking = []

>>>
>>> #class Faculty(Member): and class Student(Member) tell Python that Faculty and
Student are subclasses of class Member. That means that they inherit all of the
attributes of class Member
>>> paul = Faculty('Paul Gries', 'Ajax', 'pgries@cs.toronto.edu', '1234')
>>> paul.name
'Paul Gries'
>>> jen = Student('Jen Campbell', 'Toronto', 'campbell@cs.toronto.edu','4321')
>>> jen.name
'Jen Campbell'
>>> def myfun(self):
    """ (Member) -> str
    Return a string representation of this Member.
    >>> member = Member('Paul', 'Ajax', 'pgries@cs.toronto.edu')
    >>> member.__str__()
    'Paul\\nAjax\\npgries@cs.toronto.edu'
    """
    return '{}\n{}\n{}'.format(self.name, self.address, self.email)

>>> Member.__str__=myfun #as __str__ is a attribute alreday present in object,
reassigning it to new function, outside class
>>> str(paul)
'Paul Gries\nAjax\npgries@cs.toronto.edu'
>>> print(paul)
Paul Gries
Ajax
pgries@cs.toronto.edu
>>> print(jen)
Jen Campbell
Toronto
campbell@cs.toronto.edu
>>> #There must be a special first argument 'self' in all of method definitions which
gets bound to the calling instance
>>> #There is no “new” keyword as in Java. Just use the class name with ( ) notation
and assign the result to a variable
>>> #__init__serves as a constructor for the class.
>>> #self is similar to the keyword thisin Java or C++
>>> #But Python uses selfmore often than Java uses this
>>> class Student:
    '''A class representing a student '''
    def __init__(self,n,a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age

>>> f = Student("Bob Smith", 23)
>>> f.full_name
'Bob Smith'
>>> f.get_age()
23
>>> print(f)
<__main__.Student object at 0x000000398234E5C0>
>>> f.__str__()
'<__main__.Student object at 0x000000398234E5C0>'
>>> getattr(f, “full_name”)
SyntaxError: invalid character in identifier
>>> getattr(f, "full_name")
'Bob Smith'
>>> getattr(f, "get_age")
<bound method Student.get_age of <__main__.Student object at 0x000000398234E5C0>>
>>> getattr(f, "get_age")()
23
>>> hasattr(f, "get_age")
True
>>> hasattr(f, "get_birthday")
False
>>> #Attributes are of two types, DATA & CLASS Attributes
>>> #data attributes are created using constructor, & calling them from reference
of class produces an error
>>> #class attricutes are already present in class, & can be shared among various
objects, & accessed by class name itself
>>> class a:
    class_var=5
    def __init__(self):
        return


>>> a
<class '__main__.a'>
>>> type(a)
<class 'type'>
>>> one=a()
>>> one.class_var
5
>>> two=a()
>>> one==two
False
>>> a.class_var=1
>>> one.class_var
1
>>> two.class_var
1
>>> two.class_var+=1
>>> two.class_var
2
>>> one.class_var
1
>>> a.class_var+=1
>>> two.class_var
2
>>> one.class_var
2
```

```
>>> class counter:
    overall_total = 0# class attribute
    def __init__(self):
        self.my_total = 0# data attribute
    def increment(self):
        counter.overall_total = counter.overall_total + 1
        self.my_total = self.my_total + 1


>>> a=counter()
>>> b=counter()
>>> a.increment()
>>> a.increment()
>>> b.increment()
>>> a.my_total
2
>>> b.my_total
1
>>> a.__class__.overall_total
3

>>> class A:
    a='old string'
    def __init__(self,name):
        self.b=name
        return


>>> one=A(1)
>>> two=(2)
>>> two=A(2)
>>> one.a
'old string'
>>> one.b
1
>>> two.a
'old string'
>>> two.b
2
>>> one is two
False
>>> one.a is two.a
True
>>> one.__class__.a='new string'
>>> one.a is two.a
True
>>> one.a
'new string'
>>> two.a
'new string'
>>> one.a='latest string'
>>> one.a is two.a
False
>>> one.a
'latest string'
>>> two.a
'new string'

>>> #Inheritance: mulitple inheritance is supported & no extends keyword
>>> class Parent:
    def __init__(self,name):
        self.name=name
    def get_detail(self):
        print('I am a parent '+self.name)


>>> class Child(Parent):
    def __init__(self,p_name,c_name):
        Parent.__init__(self,p_name)
        self.c_name=c_name
    def get_detail(self):
        print('I am a child '+self.c_name,' of ',self.name)


>>> a=Parent('India')
>>> a.get_detail()
I am a parent India
>>> b=Child('India','Pak')
>>> b.get_detail()
I am a child Pak  of  India

>>> #now we'll see how multiple inheritance is supported
>>> class A:
    def __init__(self,x):
        self.x=x


>>> class B:
    def __init__(self,x):
        self.y=x


>>> class C(A,B):
    def __init__(self,x,y,z):
        A.__init__(self,x)
        B.__init__(self,y)
        self.z=z


>>> a=A(5)
>>> a.x
5
>>> b=B(6)
>>> b.y
6
>>> c=C(5,6,7)
>>> c.x,c.y,c.z
(5,6,7)
>>> 
>>> #What if two parent classes have same attribute
>>> class A:
    def __init__(self,x):
        self.x=x


>>> class B:
    def __init__(self,x):
        self.x=x
```

```
>>> class C(A,B):
	def __init__(self,x,y,z):
		A.__init__(self,x)
		B.__init__(self,y)
		self.z=z


>>> c=C(5,6,7)
>>> c.x
6
>>> class C(A,B):
	def __init__(self,x,y,z):
		B.__init__(self,x)
		A.__init__(self,y)
		self.z=z

>>> c=C(5,6,7)
>>> c.x
6
>>> #that variable x is over-written & modified later by passed parameter 'y' into
constructor of C
>>> c.__repr__()
'<__main__.C object at 0x00000039825B0C18>'
>>> str(c)
'<__main__.C object at 0x00000039825B0C18>'
>>> a=A(5)
>>> b=A(5)
>>> a==b
False
>>> a is b
False
>>> a.__doc__
>>> a.__class__.__module__
'__main__'
>>> a.__dict__
{'x': 5}
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'x']
>>> 
>>> a.__weakref__
>>> #In python, as i said nesting is allowed at any level
>>> #1. class inside a class
>>> #2. fuction inside a class(method technically)
>>> #3. class inside a function
>>> #4. function inside a function
>>> #classses are maintained by dictionaries(__dict__) containing namespace
>>> #so, you could fetch attribute, check if attribute is present
>>> #modify attrubute outside of a class
>>> #add a new attribute to object, or class, outside its definition
>>> #delete a new attribute to object or class, outside its definition
>>> #let me work upon last two of these
>>> 
>>> class A:#most basic class
	pass
```

```
>>> a=A(5)
Traceback (most recent call last):
  File "<pyshell#201>", line 1, in <module>
    a=A(5)
TypeError: object() takes no parameters
>>> #as i had not yet created constructor
>>> def f(self,n):
    self.n=n


>>> setattr(A,'__init__',f)
>>> a=A(5)
>>> a.n
5
>>> #init is already defined in class, now let us add a new method 'A'
>>> setattr(A,'get_detail',lambda self:print(self.n))
>>> a.get_detail()
5
>>> b=A(B)
>>> #now let us add a method to 'a' only not 'b'#Exclusively
>>> setattr(a,'hello',lambda self:print('you are inside object'))
>>>a.hello
<function <lambda> at 0x00000039825AFD90>
>>> a.hello()
Traceback (most recent call last):
  File "<pyshell#224>", line 1, in <module>
    a.hello()
TypeError: hello() missing 1 required positional argument: 'self'
>>> a.hello(a)
you are inside object
>>> #ok let me not use lambda functions they are awful
>>> def hello(self):
    print('you r inside object')


>>> setattr(a,'hello',hello)
>>> a.hello(a)
you r inside object
>>> b.hello(a)
Traceback (most recent call last):
  File "<pyshell#226>", line 1, in <module>
    b.hello(a)
AttributeError: 'A' object has no attribute 'hello'
>>> #hello method added to 'a' only, not 'b'
>>> #similarly we can delete attributes
>>>
>>> #getattr(obj, name[, default]): to access the attribute of object.
>>> #hasattr(obj,name): to check if an attribute exists or not.
>>> #setattr(obj,name,value): to set an attribute. If attribute does not exist, then
it would be created.
>>> #delattr(obj, name): to delete an attribute.
>>> delattr(a,'hello')
>>> a.hello(a)
Traceback (most recent call last):
  File "<pyshell#237>", line 1, in <module>
    a.hello(a)
AttributeError: 'A' object has no attribute 'hello'
>>> delattr(a,'hello')
Traceback (most recent call last):
  File "<pyshell#238>", line 1, in <module>
```

```
    delattr(a,'hello')
AttributeError: hello
>>> 
>>> #Achieving Encapsulation
>>> #variables with double underscore in front of their name are private, nothing like protected in Python, so members have access to all private variables of class
>>> class A:
    def __init__(self,n):
        self.__x=n
    def get_detail(self):
        print(self.__x)


>>> class B(A):
    def __init__(self,n,m):
        A.__init__(self,n)
        self.m=m
    def get_detail(self):
        print(self.__x,self.m)


>>> a=A(5)
>>> a.get_detail()
5
>>> b=B(4,5)
>>> b.get_detail()
Traceback (most recent call last):
  File "<pyshell#263>", line 1, in <module>
    b.get_detail()
  File "<pyshell#261>", line 6, in get_detail
    print(self.__x,self.m)
AttributeError: 'B' object has no attribute '_B__x'
>>> class B(A):
    def __init__(self,n,m):
        A.__init__(self,n)
        self.m=m
    def get_detail(self):
        A.get_detail(self)
        print(self.m)


>>> b=B(4,5)
>>> b.get_detail()
4
5
>>> b.__dict__
{'m': 5, '_A__x': 4}
>>> b._A__x
4
>>> 
```