Content Delivered on class_22_23-July-2016

- Chapter 14: OOP in Python
  - Creating Classes
  - Instance Methods
  - Constructors & Destructors
  - Special Methods
  - Class Variables
  - Inheritance
  - Polymorphism
  - Custom Exception Classes
  - using class as struct in C language
  - creating iterator class
- Chapter 16: Advanced Python Topics
  - Decorators
  - Static and Class Methods

Assignments Given ¶

Assignment 1: Define a class which has atleast two methods:

```
getString  - to get a string from console input.
printString - to display the string, in upper case, on the console.
```

Assignment 2: create a class 'car' which has attributes for basic features of any car, like registration number, Engine chase number. Then, write two child classes 'bmw' and 'volvo', which inherit the basic car features from the class 'car'.

Interview Questions

Interview Question 1: what is the difference between function and method?

Interview Question 2: Is it mandatory to place 'self' in the methods?

Interview Question 3: what is the importance of __init__()?

Interview Question 4: what is the difference between **new**() and **init**() methods?

Interview Question 5: what is the difference between __del__ attribute and del operator?

Interview Question 6: what is Duck-Typing?

Interview Question 7: How Operator overloading can be achieved in Python ?

Interview Question 8: What are the default decorators in python?

Interview Question 9: What is the difference between function and static method?

Object Oriented Programming (OOP) in Python

Python supports OOP, but, in a way something different from that of other Object oriented Programming Languages.

Features of any OOP language:

- Data Abstraction:
  - Binding to the methods and variable
- Encapsulation
  - Dividing the code into a public interface, and a private implementation of that interface
- Polymorphism
  - Ability to overload standard operators so that they have appropriate behavior based on their context
- Inheritance
  - Ability to create subclasses that contain specializations of their parents

classes

Syntax:

```
class <className>:
    <statement 1>
    <statement 2>
    <statement 3>
    ....
```

In [1]:

```
class Calculator:
    pass
```

In [3]:

```
type(Calculator)
```

Out[3]:

```
classobj
```

In [4]:

```
c = Calculator()    # Process is called Instantiation
```

In [5]:

```
c, type(c)
```

Out[5]:

```
(<__main__.Calculator instance at 0x0330B080>, instance)
```

In [6]:

```
d = Calculator    # assigning the class object, rather than instantiation
```

In [7]:

```
d, type(d)
```

Out[7]:

```
(<class __main__.Calculator at 0x03261308>, classobj)
```

Every entity in python is treated as an Object.

```
Every Object has:
            1. type
            2. Address(es)
            3. value(s)
```

- Classes are used to create new kinds of objects.
- A class defines a set of attributes that are associated with, and shared by, a collection of objects known as instances.
- A class can constitute a collection of functions (known as methods), variable (which are known as class variables, or fields), and computed attributes (which are known as properties).
- Both fields and methods, together are called attributes of the class.
- class name starts with a Capital letter, based on PEP8 recommendations.

In [8]:

```
class newclass:
        courseName = 'Python'        # class variables or fields
        courseLevel = 'Advanced'
        def hello(self):             # class method
                print 'Hello World!'
```

In [9]:

```
newclass()
```

Out[9]:

```
<__main__.newclass instance at 0x0331AC60>
```

In [10]:

```
newclass
```

Out[10]:

```
<class __main__.newclass at 0x032F1A40>
```

In [11]:

```
c1 = newclass() # c1 will be the object of type 'newclass'
```

Python automatically maps instance method calls to class method functions as follows. Method calls made through an instance, like this:

```
instance.method(args...)   -->  class.method(instance, args...)
```

In [13]:

```
type(newclass()), type(newclass), type(c1)
```

Out[13]:

```
(instance, classobj, instance)
```

In [14]:

```
id(newclass()), id(newclass), id(c1)
```

Out[14]:

```
(53609336, 53418560, 53609016)
```

In [15]:

```
print c1.courseName
```

```
Python
```

In [16]:

```
print c1.courseLevel
```

```
Advanced
```

In [17]:

```
print c1.hello()
```

```
Hello World!
None
```

In [18]:

```
print c1.hello
```

```
<bound method newclass.hello of <__main__.newclass instance at 0x03320238>
>
```

In [19]:

```
dir(c1)
```

Out[19]:

```
['__doc__', '__module__', 'courseLevel', 'courseName', 'hello']
```

self and its usage

Interview Question 1: what is the difference between function and method?

> Functions defined within a class are called methods. Also, every method has self as the default input.
>
> - self acts as a placeholder for the object.

In [20]:

```
class newClass:
        def courseName(self, name):
                self.name = name
        def displayName(self):
                return self.name
        def printCourse(self):
                print "course name is %s"%self.name
```

In [21]:

```
course1 = newClass()
course2 = newClass()
```

In [22]:

```
course1.courseName('Core Python Programming')
course2.courseName('Python_Django')
```

In [23]:

```
course1.displayName
course2.displayName
```

Out[23]:

```
<bound method newClass.displayName of <__main__.newClass instance at 0x033
204E0>>
```

In [25]:

```
course1.displayName()
```

Out[25]:

```
'Core Python Programming'
```

In [26]:

```
course2.displayName()
```

Out[26]:

```
'Python_Django'
```

In [27]:

```
course1.printCourse()
```

```
course name is Core Python Programming
```

In [28]:

```
course2.printCourse()
```

```
course name is Python_Django
```

Interview Question 2: Is it mandatory to place 'self' in the methods?

- The methods require a placefolder for the object.
- The name 'self' is not mandatory.
- Any name can be used, but it should be consistently used throughout the program.
- PEP 8 recommends to use the name 'self' only.

Constructor Method __init__()

In [29]:

```
class new:
        def __init__(self):
                print 'I am constructor. I will be born, once the instance is created'
                self.variable = 'Hello'
```

In [30]:

```
myObject = new()
```

```
I am constructor. I will be born, once the instance is created
```

In [31]:

```
myObject.variable
```

Out[31]:

```
'Hello'
```

Interview Question 3: what is the importance of __init__()

- __init__() method is called the default constructor.
- It is method that get invoked, on instantiation of an object.

With this knowledge, let us try to write a small script.

In [103]:

```
#!/usr/bin/python

class shape:
        def __init__(self,x,y):
                self.x = x
                self.y = y
        def area(self):
                return self.x * self.y
        def perimeter(self):
                return 2*self.x + 2*self.y
        def describe(self, text):
                self.description = text
        def scalesize(self, scale):
                self.x = self.x * scale
                self.y = self.y * scale
```

In [104]:

```
rectangle = shape(100,45)
```

In [105]:

```
print "dir(rectangle) ", dir(rectangle)
```

```
dir(rectangle)  ['__doc__', '__init__', '__module__', 'area', 'describe',
 'perimeter', 'scalesize', 'x', 'y']
```

In [107]:

```
print rectangle.area()
```

```
4500
```

In [108]:

```
print rectangle.perimeter()
```

```
290
```

In [109]:

```
print rectangle.describe("This is a rectangle")  # Nothing gets printed, as there is no
 print in describe() method
```

```
None
```

In [111]:

```
rectangle.scalesize(0.5)
```

In [112]:

```
print rectangle.area()   # Observe the corresponding change
```

```
281.25
```

Interview Question 4: what is the difference between **new**() and **init**() methods?

- **new** is static class method, while **init** is instance method.
- **new** has to create the instance first, so **init** can initialize it.
- Note that **init** takes self as parameter. Until you create instance there is no self.

Another example: Let us try to create a BankAccount class, with deposit, withdraw and status functionality

In [36]:

```
class BankAccount(object):     # name 'object' is optional. Has its usage in inheritanc
e
        '''
                These are class docstrings.
                This is BankAccount class
        '''
        num_accounts = 0
        def __init__(self,name,balance):
                '''
                This is __init__ method
                '''
                self.name = name
                self.balance = balance
                BankAccount.num_accounts += 1
        def __del__(self):
                '''
                This is __del__ method.
                It is also called default desctructor
                '''
                BankAccount.num_accounts -= 1
        def deposit(self,amt):
                '''
                This is deposit method.
                format: deposit(amount)
                usage: deposit(1000)
                '''
                self.balance = self.balance + amt
        def withdraw(self,amt):
                '''
                This is withdraw method.
                format: withdraw(amount)
                usage: withdraw(999)
                '''
                self.balance = self.balance - amt
        def status(self):
                '''
                This is status method.
                format: status()
                usage: status()
                It is used to get the balanace status
                '''
                return self.balance
```

In [37]:

```
BankAccount
```

Out[37]:

```
__main__.BankAccount
```

In [38]:

```
b = BankAccount()   # arguments given to instance, must match to that given in __init__
() method
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call last)
<ipython-input-38-6a47f35f18b5> in <module>()
----> 1 b = BankAccount()

TypeError: __init__() takes exactly 3 arguments (1 given)
```

In [39]:

```
b = BankAccount('Vijay Malya', 100000)
```

In [40]:

```
dir(b)
```

Out[40]:

```
['__class__',
 '__del__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'balance',
 'deposit',
 'inquiry',
 'name',
 'num_accounts',
 'withdraw']
```

In [41]:

```
b.__doc__
```

Out[41]:

```
'\n\t\tThese are class docstrings.\n\t\tThis is BankAccount class\n\t'
```

In [42]:

```
print b.__doc__
```

```
    These are class docstrings.
    This is BankAccount class
```

In [43]:

```
b.balance
```

Out[43]:

```
100000
```

In [44]:

```
b.name
```

Out[44]:

```
'Vijay Malya'
```

In [45]:

```
b.name?   # To get help in ipython environment. In python interpreter, use help()
```

In [46]:

```
b.num_accounts
```

Out[46]:

```
0
```

In [47]:

```
b.withdraw(100)
```

In [48]:

```
b.balance
```

Out[48]:

```
99900
```

In [50]:

```
b.withdraw(1000)
```

In [51]:

```
b.balance
```

Out[51]:

```
98900
```

In [52]:

```
b.deposit(1)
```

In [53]:

```
b.balance
```

Out[53]:

98901

Interview Question 5: what is the difference between __del__ attribute and del operator?

- del operator is used initiative.
- __del__ is the default destructor, which will do any further closures, after deleting the instance, like file/socket closures, reducing the reference counts to objects, etc.
- placing the __del__ is like an extra safety closure, before the garbage collector does the closure

In [54]:

```
del b     # deleting the instance
```

In [56]:

```
b.balance   # The instance 'b' is no more present
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call last)
<ipython-input-56-e1cc3ff27d1d> in <module>()
----> 1 b.balance

NameError: name 'b' is not defined
```

Another Example: Let us try to create a Cone.

In [113]:

```
class Cone():
        def __init__(self, d0, de, L):
                '''Create a cone'''
                self.a0 = d0/2
                self.ae = de/2
                self.L = L
        def __del__(self):    # default destructor
                pass              # Observe that no logic was written for any closures
        def radius(self,z):
                return self.ae + (self.a0-self.ae)*z/self.L
        def radiusp(self, z):
                '''derivative of the radius at z'''
                return (self.a0 - self.ae)/self.L
```

In [114]:

```
c = Cone(0.1, 0.2, 1.5)
```

In [115]:

```
c.radius(0.5)
```

Out[115]:

```
0.08333333333333334
```

In [116]:

```
c.radiusp(0.5)
```

Out[116]:

```
-0.03333333333333333
```

Assignment 1: Define a class which has atleast two methods:

```
getString  - to get a string from console input.
printString - to display the string, in upper case, on the console.
```

Inheritance

One of the benifits of object oriented programming is the reusage of code. Inheritance is one of the ways by which we can achive this.

> It is implemended by establishing a relation between the classes. parent-child or super-sub classes relation

In [57]:

```
class Parentclass:
        def courseName(self, name):
                self.name = name
                return self.name

class Childclass(Parentclass):
        pass
```

In [58]:

```
parObj1 = Parentclass()
parObj1.courseName('Python Programming')
```

Out[58]:

```
'Python Programming'
```

In [59]:

```
childObj = Childclass()
```

In [60]:

```
childObj.courseName('Django') # using 'Parentclass' class method in for object created
 using 'Childclass' class
```

Out[60]:

```
'Django'
```

In [61]:

```
#!/usr/bin/python

class SchoolMember:
        '''
        Represents any school member
        '''
        def __init__(self, name, age):
                self.name = name
                self.age = age
                print 'Initialization of School members: %s' %self.name
        def tell(self):
                '''
                Tell my details
                '''
                print 'Name: %s Age: %s'%(self.name, self.age)

class Teacher(SchoolMember):
        '''
        Represents a teacher
        '''
        def __init__(self, name, age, salary):
                SchoolMember.__init__(self, name, age)# __init__ undergoes operator Over
loading
                self.salary = salary
                print 'Initialization of Teacher: %s' %self.name
        def tell(self):
                SchoolMember.tell(self)
                print 'salary: %d' %self.salary

class Student(SchoolMember):
        '''
        Represents a student.
        '''
        def __init__(self, name, age, marks):
                SchoolMember.__init__(self, name, age)
                self.marks = marks
                print 'Initialization of Student: %s' %self.name

        def tell(self):
                SchoolMember.tell(self)
                print 'Marks: %d' %self.marks
```

In [63]:

```
t = Teacher('Dr. Radha Krisha', 40, 70000)
```

```
Initialization of School members: Dr. Radha Krisha
Initialization of Teacher: Dr. Radha Krisha
```

In [64]:

```
s= Student('Narayana', 17, 98)
```

```
Initialization of School members: Narayana
Initialization of Student: Narayana
```

In [65]:

```
members = [t,s]
print "\n\tSchoolMember details\n".upper()

for member in members:
        member.tell()
```

```
        SCHOOLMEMBER DETAILS

Name: Dr. Radha Krisha Age: 40
salary: 70000
Name: Narayana Age: 17
Marks: 98
```

Assignment 2: create a class 'car' which has attributes for basic features of any car, like registration number, Engine chase number. Then, write two child classes 'bmw' and 'volvo', which inherit the basic car features from the class 'car'.

1.5 OverWrite Variables on a subclass

In [67]:

```
class mySuperclass:
        course1 = 'core Python'
        course2 = 'Django'

class mySubclass(mySuperclass):
        course2 = 'Matlab'
```

In [68]:

```
supObj = mySuperclass()
supObj.course1
```

Out[68]:

```
'core Python'
```

In [69]:

```
supObj.course2
```

Out[69]:

```
'Django'
```

In [70]:

```
subObj = mySubclass()
```

In [71]:

```
subObj.course1
```

Out[71]:

```
'core Python'
```

In [72]:

```
subObj.course2 # child class fields and methods are prioritized, compared to the parent
 class
```

Out[72]:

```
'Matlab'
```

Multiple Inheritance

Let us try a case where a single child class inherits from two parent classes.

In [76]:

```
class CorePython:
        chp1 = "Introduction"
        chp2 = "Help"
        chp6 = "CP Advanced"

class PythonAdvanced:
        chp4 = 'numpy'
        chp5 = 'scipy'
        chp6 = 'PA Advanced'

class ChildPython(CorePython, PythonAdvanced):
        chp3 = "dictionaries"


courseObject = ChildPython()
```

In [77]:

```
print "courseObject.chp1 ", courseObject.chp1
print "courseObject.chp2 ", courseObject.chp2
print "courseObject.chp3 ", courseObject.chp3
print "courseObject.chp4 ", courseObject.chp4
print "courseObject.chp5 ", courseObject.chp5
print "courseObject.chp6 ", courseObject.chp6   # parentclass attributes will be prioti
zed during inheritance, based on their inheritance placement
```

```
courseObject.chp1  Introduction
courseObject.chp2  Help
courseObject.chp3  dictionaries
courseObject.chp4  numpy
courseObject.chp5  scipy
courseObject.chp6  CP Advanced
```

Overloading

Overloading includes function, method or operator overloading. Overloading feature is NOT supported by python. But, that can be achived using duck-Typing.

> **Interview Question 6:** what is Duck-Typing?
>
> Making a work-around to achieve something which is not supported by the programming language.

Method Overloading:

In [78]:

```
#!/usr/bin/env python

class Human:
    def sayHello(self, name=None):
        if name is not None:
            print 'Hello ' + name
        else:
            print 'Hello '

# Create instance
obj = Human()
```

In [79]:

```
# Call the method
obj.sayHello()
```

```
Hello
```

In [80]:

```
# Call the method with a parameter
obj.sayHello('Guido')
```

```
Hello Guido
```

Function Overloading:

In [81]:

```
import types

def testtype(input):
        if type(input) ==  types.FloatType:
                print "I got a float"
        elif type(input) == types.ComplexType:
                print "A complex number"
        else:
                print "Something else"
```

In [82]:

```
testtype(2.3)
```

```
I got a float
```

In [83]:

```
testtype(23)
```

```
Something else
```

In [84]:

```
testtype(2.3j)
```

```
A complex number
```

In [85]:

```
testtype('23')
```

```
Something else
```

Interview Question 7: How Operator overloading can be achieved in Python ?

Using the '+' operator, we can perform both arithmetic addition of two numbers, and String concatenation.

1+2 - addition

'1'+'2' - concatenation

Encapsulation

- Encapsulation is the mechanism for restricting the access to some of an object's components. Accessing internal fields(variables) of a class, from outside, is achieved though special methods.

```
__Name    Notation   Behaviour__
-----------------------------------------
name      Public     Can be accessed from inside and outside
_name     Protected  Like a public member, but they shouldn't be directly access
ed from outside.
__name    Private    Can't be seen and accessed from outside
```

In [86]:

```
class myClass(object):
     def __init__(self):
         self.var = 'pokemon go1'      # can be accessed directly
         self._var = 'pokemon go2'     # private variable
         self.__var = 'pokemon go3'    # private vatible, name mangled
```

In [87]:

```
myInstance = myClass()
```

In [88]:

```
myInstance.var
```

Out[88]:

```
'pokemon go1'
```

In [89]:

```
myInstance._var  # accesing protected
```

Out[89]:

```
'pokemon go2'
```

In [90]:

```
myInstance.__var  # private attributes can't be accessed directly
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call last)
<ipython-input-90-70d4c4e73570> in <module>()
----> 1 myInstance.__var

AttributeError: 'myClass' object has no attribute '__var'
```

In [91]:

```
dir(myInstance)
```

Out[91]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_myClass__var',
 '_var',
 'var']
```

In [92]:

```
myInstance._myClass__var  # accessing the private attribute
```

Out[92]:

```
'pokemon go3'
```

This way of naming is called **Name Mangling**.

> Name Mangling is used to ensure that subclasses don't accidentally override the private methods and attributes of their superclasses. It's not designed to prevent deliberate access from outside.

In [93]:

```
class Foo(object):
    def __init__(self):
        self.__baz = 42
    def foo(self):
        print self.__baz


class Bar(Foo):
    def __init__(self):
        super(Bar, self).__init__()
        self.__baz = 21
    def bar(self):
        print self.__baz
```

In [94]:

```
x=Bar()
```

In [95]:

```
x.foo()
```

```
42
```

In [96]:

```
x.bar()
```

```
21
```

In [97]:

```
print x.__dict__
```

```
{'_Bar__baz': 21, '_Foo__baz': 42}
```

Using classes as C struct

In [119]:

```
class Person:
        pass
```

In [120]:

```
prakash = Person # creats an empty employee record
```

In [121]:

```
prakash.name = 'Prakash Phani'
```

In [122]:

```
prakash.age = '28'
```

In [123]:

```
prakash.weight = 68
```

At this point you may get a double, as what is the different between class and dictionary, in this case?

> Dictionary stored elements in key-value pairs, and need to be retrieved using dict['key'].
> Whereas classes store elements as instance attributes.

Exceptions

User-defined exceptions are identified by classes as well.

There are two ways of raising custom exception classes::

```
raise Class, instance
raise instance

(or)

raise instance.__class__, instance
```

In [124]:

```
#!/usr/bin/python

class B:
        pass
class C(B):
    pass
class D(C):
    pass

for i in [B, C, D]:
    try:
        raise i()  # It works like B()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

```
B
C
D
```

Creating a Iterator class object

In [125]:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

In [126]:

```
rev = Reverse('Python Programming')
```

In [127]:

```
type(rev)  # object is of instance type.
```

Out[127]:

```
instance
```

In [128]:

```
rev.next()
```

Out[128]:

```
'g'
```

In [129]:

```
rev.next()
```

Out[129]:

```
'n'
```

In [130]:

```
for char in rev:    # Observe that the first two character weren't printed
    print char,
```

```
i m m a r g o r P   n o h t y P
```

Decorator

- takes function as an input, and returns the same function back, or a function with a similar signature.

```
    @mydecorator
    def myfunc():
        pass

    def mydecorator():
        pass
```

is equivalent to

```
    def myfunc():
        pass
    myfunc = mydecorator(myfunc)
```

Let us start with a identity decorator, which just return back the input

In [99]:

```
def identity(ob):
    return ob

@identity
def myfunc():
    print "my function"

myfunc()
print myfunc
```

```
my function
<function myfunc at 0x03330BF0>
```

@wraps - prints some text before and after calling the decorated function.

In [131]:

```
from functools import wraps

def mydecorator(f):
    @wraps(f)
    def wrapped(*args, **kwargs):
        print "Before decorated function"
        r = f(*args, **kwargs)
        print "After decorated function"
        return r
    return wrapped

@mydecorator
def myfunc(myarg):
    print "my function", myarg
    return "return value"

r = myfunc('asdf')
print r
```

```
Before decorated function
my function asdf
After decorated function
return value
```

Interview Question 8: What are the default decorators in python?

> @staticmethod and @classmethod are the default decorators in python. All the instance methods have 'self' as default input. classmethod has 'cls' as input. Static method has no default input at all.

Interview Question 9: What is the difference between function and static method?

> Both dont have any default input; but, static method need to be accesses either using the class() or its object.

In [134]:

```
#!/usr/bin/python
# Default Decorators: @staticmethod, @classmethod

class myClass(object):
    def display(self,x):
        print "executing instance method display(%s,%s)"%(self,x)

    @classmethod
    def cmDisplay(cls,x):
        print "executing class method display(%s,%s)"%(cls,x)

    @staticmethod
    def smDisplay(x):
        print "executing static method display(%s)"%x

a = myClass()
a.display('Django')             # accessing instance method

myClass.cmDisplay('Django')     # accessing class method

a.smDisplay('Django')           # accessing static method

myClass.smDisplay('Django')     # accessing static method
```

```
executing instance method display(<__main__.myClass object at 0x0333FDB0>,
Django)
executing class method display(<class '__main__.myClass'>,Django)
executing static method display(Django)
executing static method display(Django)
```

Recommended Tutorial for decorators:

1. [python decorators in 12 steps](http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/)