Objected Oriented Programming in Python

Overview of OOP Terminology

Class – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

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

Function overloading – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

Instance variable – A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance – The transfer of the characteristics of a class to other classes that are derived from it.

Instance – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation – The creation of an instance of a class.

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

Object – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading – The assignment of more than one function to a particular operator.

In [7]:
```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print ("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print ("Name : ", self.name,  ", Salary: ", self.salary)
```

The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.

The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

You declare other class methods like normal functions with the exception that the first argument to each method is **self**. Python adds the **self** argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

In [8]:
```
emp1 = Employee("Zara", 2000)
```

In [9]:
```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

In [10]:
```
emp1.displayEmployee()
emp2.displayEmployee()
```

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
```

You can add, remove, or modify attributes of classes and objects at any time:

In [13]:
```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

In [14]:
```
emp1.age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-0d569e33fad3> in <module>()
----> 1 emp1.age

AttributeError: 'Employee' object has no attribute 'age'
```

In [15]:
```
emp2.age = 12
```

In [16]:
```
print ("Total Employee %d" % Employee.empCount)
```

```
Total Employee 2
```

Instead of using the normal statements to access attributes, you can use the following functions:

The **getattr(obj, name[, default])** – to access the attribute of object.

The **hasattr(obj,name)** – to check if an attribute exists or not.

The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.

The **delattr(obj, name)** – to delete an attribute.

In [17]:
```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
```

Out[17]:
```
False
```

In [18]:
```
getattr(emp2, 'age')    # Returns value of 'age' attribute
```

Out[18]:
```
12
```

In [19]:
```
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
```

In [20]:
```
delattr(emp2, 'age')    # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

dict – Dictionary containing the class's namespace.

doc – Class documentation string or none, if undefined.

name – Class name.

module – Module name in which the class is defined. This attribute is "**main**" in interactive mode.

bases – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

In [21]:
```
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__)
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for
all employees', 'empCount': 2, '__init__': <function Employee.__init__ at 0x000
001E99D216AE8>, 'displayCount': <function Employee.displayCount at 0x000001E99D
216A60>, 'displayEmployee': <function Employee.displayEmployee at 0x000001E99D2
161E0>, '__dict__': <attribute '__dict__' of 'Employee' objects>, '__weakref_
_': <attribute '__weakref__' of 'Employee' objects>}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. But a class can implement the special method **del**(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance:

In [22]:
```
class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print (class_name, "destroyed")
```

In [23]:
```
pt1 = Point()
```

In [24]:
```
del pt1
```

```
Point destroyed
```

In [25]:
```
class Garage:
    def __init__(self):
        self.cars = []
    def __len__(self):
        return len(self.cars)
    def __str__(self):
        return f'Cars : {self.cars}'
    def __repr__(self):
        return f'My garage cars repr : {self.cars}'
    def __getitem__(self, i):
        return self.cars[i]
```

In [26]:
```
my_garage = Garage()
```

In [27]:
```
my_garage.cars.append("FOrd")
```

In [28]:
```
print (len(my_garage))
```

```
1
```

In [29]:
```
print(my_garage[0]) #Garage.__getitem__(my_garage, 0)
```

```
FOrd
```

In [30]:
```
for car in my_garage:
    print (car)
```

```
FOrd
```

Class Inheritance

In [31]:
```
class Student():
    def __init__(self, name, grades):
        self.name= name
        self.grades= grades
    def avg(self):
        return sum(self.grades)/len(self.grades)
```

In [32]:
```
class WorkingStudent(Student):
    def __init__(self,  name, school, salary):
        super().__init__( name, school)
        self.salary= salary

    # just get self and use self features to output smt
    @property
    def weekly_salary(self):
        return self.salary *7
```

In [33]:
```
wstu=WorkingStudent('hasan', 'kosaran', 2700)
```

In [34]:
```
wstu.weekly_salary()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-34d39df22496> in <module>()
----> 1 wstu.weekly_salary()

TypeError: 'int' object is not callable
```

In [35]:
```
wstu.weekly_salary
```

Out[35]:
```
18900
```

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

In [36]:
```
class Parent:        # define parent class
    def myMethod(self):
        print ('Calling parent method')

class Child(Parent): # define child class
    def myMethod(self):
        print ('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

```
Calling child method
```

Overloading Operators

In [37]:
```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)

    def __mul__(self, other):
        return Vector(self.a * other.a, self.b * other.b)
```

In [38]:
```
v1 = Vector(2, 10)
v2 = Vector(5, -2)
print (v1 + v2)
print (v1 * v2)
```

```
Vector (7, 8)
Vector (10, -20)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a **double underscore** prefix, and those attributes then are not be directly visible to outsiders.

In [39]:
```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print (self.__secretCount)
    def get_count(self):
        print ("Counter is %d" % self.__secretCount)


counter = JustCounter()
counter.count()
counter.count()
counter.get_count()
print (counter.__secretCount)
```

```
1
2
Counter is 2

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-39-2b1bc3a67d6e> in <module>()
     13 counter.count()
     14 counter.get_count()
---> 15 print (counter.__secretCount)

AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

You can access such attributes as object._className__attrName.

In [40]:
```
print (counter._JustCounter__secretCount)
```

```
2
```

@classmethod and @staticmethod

In [41]:
```
class A(object):
    def foo(self,x):
        print ("executing foo(%s,%s)"%(self,x))

    @classmethod
    def class_foo(cls,x):
        print ("executing class_foo(%s,%s)"%(cls,x))

    @staticmethod
    def static_foo(x):
        print ("executing static_foo(%s)"%x)

a=A()
```

In [42]:
```
a.foo(1)
```

```
executing foo(<__main__.A object at 0x000001E99D2C65C0>,1)
```

In [43]:
```
a.class_foo(1)
```

```
executing class_foo(<class '__main__.A'>,1)
```

In [44]:
```
A.class_foo(1)
```

```
executing class_foo(<class '__main__.A'>,1)
```

In [45]:
```
a.static_foo(1)
# executing static_foo(1)

A.static_foo('hi')
# executing static_foo(hi)
```

```
executing static_foo(1)
executing static_foo(hi)
```

In [46]:
```
class Date(object):

    def __init__(self, day=0, month=0, year=0):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999

date2 = Date.from_string('11-09-2012')
is_date = Date.is_date_valid('11-09-2012')
```

In [47]:
```
from datetime import date

# random Person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def fromBirthYear(cls, name, birthYear):
        return cls(name, date.today().year - birthYear)

    def display(self):
        print(self.name + "'s age is: " + str(self.age))

person = Person('Adam', 19)
person.display()

person1 = Person.fromBirthYear('John',  1985)
person1.display()
```

```
Adam's age is: 19
John's age is: 33
```