OOP with Python

Prepared by Fahad Siddiqui on 28 Feb 2020

Outline

- Object-Oriented Programming (OOP)
- Overview of OOP Terminology
- Classes and Objects
- Built-In Class Attributes
- Inheritance
  - Base Overloading Methods
- Encapsulation
- Polymorphism
- Abstraction
- Destroying Objects (Garbage Collection)

What Is Object-Oriented Programming (OOP) ?

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

- Python is a multi-paradigm programming language. Meaning, it supports different programming approach.
- One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - attributes
  - behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

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.

Instance variable:

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

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.

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.

Operator overloading:

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

Inheritance:

The transfer of the characteristics of a class to other classes that are derived from it. A process of using details from a new class without modifying existing class.

Encapsulation

Hiding the private details of a class from other objects.

Polymorphism

A concept of using common operation in different ways for different data input.

Classes and Objects

In [9]:

```
'''
"__init__" is a reseved method in python classes. It is known as a constructor in object or
This method called when an object is created from the class and it allow the class to initi

'''


class Patient():
    first_name = ""
    def __init__(self, first_name):
        self.first_name = first_name

patient1 = Patient("Ahmed")

patient1.first_name
```

Out[9]:

```
'Ahmed'
```

In [10]:

```
class Patient():
    first_name = ""
    last_name = ""
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def showFullName(self):
        print(self.first_name, " ", self.last_name)


patient1 = Patient("Ahmed","Shaikh")

patient1.showFullName()
```

```
Ahmed   ALi
```

In [24]:

```
# freestanding function

class Patient():
    first_name = ""
    last_name = ""
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def showFullName(self):
        print(self.first_name, " ", self.last_name)

    def changeLastName(self,last_name):
        self.last_name = last_name
        return True

patient1 = Patient("Ahmed","Shaih")

patient1.showFullName()

response = patient1.changeLastName("Shaikh")

if response:
    print("Last name has changed" , end="\n\n")

patient1.showFullName()
```

```
Ahmed   Shaih
Last name has changed

Ahmed   Shaikh
```

In [25]:

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

```
Blu sings 'Happy'
Blu is now dancing
```

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 [42]:

```
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)

print("Employee.__doc__:", Employee.__doc__ , end="\n\n")

print("Employee.__name__:", Employee.__name__ , end="\n\n")

print("Employee.__module__:", Employee.__module__ , end="\n\n")

print("Employee.__bases__:", Employee.__bases__ , end="\n\n")

print("Employee.__dict__:", Employee.__dict__ , end="\n\n")
```

```
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': 0, '__init__': <function Employee.__init__ a
t 0x000001C85A373AF8>, 'displayCount': <function Employee.displayCount at 0x
000001C85A3735E8>, 'displayEmployee': <function Employee.displayEmployee at
0x000001C85A3739D8>, '__dict__': <attribute '__dict__' of 'Employee' objects
>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>}
```

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

In [65]:

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()
```

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In [67]:

```
# Checking instance!
# isinstance(obj, Class)

print("Is Panguin object ? ", isinstance(peggy,Penguin) , end = "\n\n" )

print("Is Panguin object ? ", isinstance(peggy,Bird) , end = "\n\n" )


# Checking superclass
# issubclass(sub, sup)

print("Is Panguin is super class of Bird ? ", issubclass(Penguin, Bird) , end = "\n\n" )

print("Is Bird is super class of Panguin ? ", issubclass(Bird, Penguin) , end = "\n\n" )
```

```
Is Panguin object ?  True

Is Panguin object ?  True

Is Panguin is super class of Bird ?  True

Is Bird is super class of Panguin ?  False
```

Base Overloading Methods

__init__ (self [,args...])

```
    Constructor (with any optional arguments)
    Sample Call : obj = className(args)
```

__del__(self)

```
    Destructor, deletes an object
    Sample Call : del obj
```

__repr__(self)

```
    Evaluable string representation
    Sample Call : repr(obj)
```

__str__(self)

```
    Printable string representation
    Sample Call : str(obj)
```

__cmp__ (self, x)

```
    Object comparison
    Sample Call : cmp(obj, x)
```

In [76]:

```
# Example

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)

v1 = Vector(2,10)
v2 = Vector(5,-2)

'''
Suppose you have created a Vector class to represent two-dimensional vectors,
what happens when you use the plus operator to add them? Most likely Python will yell at yo

You could, however, define the __add__ method in your class to perform vector addition and
the plus operator would behave as per expectation

'''

# str will be called
print(v1)

# add will be called
print(v1 + v2)
```

```
Vector (2, 10)
Vector (7, 8)
```

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.

In [78]:

```
class Computer:

    def __init__(self):
        # __maxprice is a private variable
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    # Setter Function
    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

In [35]:

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")


class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

```
Parrot can fly
Penguin can't fly
```

Abstraction

Abstraction in Python is achieved by using abstract classes and interfaces.

An abstract class is a class that generally provides incomplete functionality and contains one or more abstract methods.

Abstract methods are the methods that generally don’t have any implementation, it is left to the sub classes to provide implementation for the abstract methods.

In [80]:

```
from abc import ABC, abstractmethod

class Payment(ABC):
    def print_slip(self, amount):
        print('Purchase of amount- ', amount)

    # With pass, we indicate a "null" block.
    @abstractmethod
    def payment(self, amount):
        pass


class CreditCardPayment(Payment):
    def payment(self, amount):
        print('Credit card payment of- ', amount)


class MobileWalletPayment(Payment):
    def payment(self, amount):
        print('Mobile wallet payment of- ', amount)


obj = CreditCardPayment()
obj.payment(100)
obj.print_slip(100)
print(isinstance(obj, Payment))

print()

obj = MobileWalletPayment()
obj.payment(200)
obj.print_slip(200)
print(isinstance(obj, Payment))
```

```
Credit card payment of-  100
Purchase of amount-  100
True

Mobile wallet payment of-  200
Purchase of amount-  200
True
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space.Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with del.

In [50]:

```
'''

a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>



A class can implement the special method __del__(), called a destructor, that is invoked wh
to be destroyed. This method might be used to clean up any non memory resources used by an

'''


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" )

pt1 = Point()
pt2 = pt1
pt3 = pt1

# prints the ids of the obejcts

print("Point 1", id(pt1) , end = "\n\n")

print("Point 2", id(pt2) , end = "\n\n")

print("Point 3", id(pt3) , end = "\n\n")


del pt1
del pt2
del pt3
```

```
Point 1 1960018544968

Point 2 1960018544968

Point 3 1960018544968

Point destroyed
```

The End!