Object-Oriented Programming in Python

Created by Sabrina Aliyeva

Class and Objects

There are two ways to assign values to properties of a class:

1. Assigning values when defining the class.

```
class Employee:
    # defining the properties
    ID = 1655
    department = "Engineer"
# creating an object of the Employee class
John = Employee()
# printing properties of
print("ID =", John.ID)
print("Department:", John.department)
```

2. Assigning values in the main code.

```
class Employee:
    # defining the properties
    ID = None
    department = None
# creating an object of the Employee class
John = Employee()
# assigning values to properties of John
John.ID = 1655
John.department = " Engineer"
# creating a new attribute for John
John.title = "Manager"
# Printing properties of John
print("ID =", John.ID)
print("Department:", John.department)
print("Title:", John.title)
```

♦ Initializing Objects

The initialization method is similar to other methods but has a pre-defined name, __init__. Python interpreter will treat the double underscores as a special case. The initializer is a special method because it does not have a return type. The first parameter of __init__ is self, which is a way to refer to the object being initialized.

```
class Employee:
    # defining the properties
    def __init__(self, ID, department):
        self.ID = ID
        self.department = department
# creating an object of the Employee class
Anna = Employee(2565, "Scientist")
```

A default initializer can also have all properties as optional. In this case, all the new objects will be created using the properties initialized in the initializer definition.

```
class Employee:
    # defining the properties
    def __init__(self, ID=None, salary=0):
        self.ID = ID
        self.salary = salary
# creating an object of the Employee class with default parameters
Anna = Employee()
John = Employee("1655", 2500)
# Printing properties of Anna and John
print("Anna")
print("ID :", Anna.ID)
print("Salary :", Anna.salary)
print("John")
print("ID :", John.ID)
print("Salary :", John.salary)
```

♦ Class and Instance Variables

1. The class variables are shared by all instances or objects of the classes. A change in the class variable will change the value of that property in all the objects of the class.
2. The instance variables are unique to each instance or object of the class. A change in the instance variable will change the value of the property in that specific object.

```
class Employee:
    department = "Engineer" # class variables
    def __init__(self, name):
            # creating instance variables
        self.name = name
e1 = Employee ('Anna')
e2 = Employee ('John')
print("Name:", e1.name) # Name: Anna
print("Department Name:", e1. department) # Department Name: Engineer
print("Name:", e2.name) # Name: John
print("Department Name:", e2. department)
```

♦ Implementing Methods in a Class

- Instance methods

```
class Employee:
    # defining the properties
    def __init__(self, ID=None, salary=None):
        self.ID = ID
        self.salary = salary
    # Method to return Salary per month
    def salaryPerMonth(self):
        return (self.salary / 12)
Anna = Employee(3789, 140000)
print(Anna.salaryPerMonth())
```

- Class methods

To declare a method as a class method, the decorator @classmethod and cls is used to refer to the class just like self is used to refer to the object of the class. You can use any other name instead of cls, but cls is used as per convention

```
class Employee:
    department = "Engineer" # class variables
    def __init__(self, name):
            # creating instance variables
        self.name = name
    @classmethod
    def getDeptName(cls):
        return cls.department
print(Employee.getDeptName())
```

- Static methods

These methods are usually limited to class only and not their objects. They have no direct relation to the class variables or instance variables. They are used as utility functions inside the class or when we do not want the inherited classes to modify a method definition.

```
class Employee:
    department = "Engineer" # class variables
    def __init__(self, name):
            # creating instance variables
        self.name = name
    @staticmethod
    def test():
        print("Printing a static method")
John = Employee ('John')
John.test() # Printing a static method.
Employee.test() # Printing a static method
```

♦ Method Overloading

Overloading refers to making a method perform different operations based on the nature of its arguments.

```
class Employee:
    def __init__(self, ID=None, salary=None):
        self.ID = ID
        self.salary = salary
    def tax(self):
        return (self.salary * 0.2)
    def salaryPerMonth(self):
        return (self.salary / 12)
    # method overloading
    def test(self, a, b, c = 10, d=None):
        print("a =", a)
        print("b =", b)
        print("c =", c)
        print("d =", d)
Anna = Employee()
print("Test 1")
Anna.test(1, 2, 3)
print("Test 2")
Anna.test(1, 2, 3, 4)
```

♦ Method Overriding

Method overriding is the process of redefining a parent class's method in a subclass.

- The method in the parent class is called overridden method.
- The method in the child class is called overriding methods.

```
class Shape:
    # initializing sides of all shapes to 0
    def __init__(self):
        self.sides = 0
    def getArea(self):
        pass
# derived form Shape class
class Rectangle(Shape):
    def __init__(self, width=0, height=0):
        self.width = width
        self.height = height
        self.sides = 4
    # method to calculate Area
    def getArea(self):
        return (self.width * self.height)
# derived form Shape class
class Circle(Shape):
    def __init__(self, radius=0):
        self.radius = radius
    # method to calculate Area
    def getArea(self):
        return (self.radius * self.radius * 3.142)
print("Area of rectangle is:",
str(Rectangle(6, 10).getArea()))
print("Area of circle is:", str(Circle(7).getArea()))
```

♦ Operator Overloading

The second argument can be named anything, but as per convention, we will be using the word other to reference the *other* object.

```
class ComplexNum:
    def __init__(self, real=0, img=0):
        self.real = real
        self.img = img
    # overloading the `+` operator
    def __add__(self, other):
        temp = ComplexNum (self.real + other.real, self.img +
other.img)
        return temp
obj1 = ComplexNum (3, 7)
obj2 = ComplexNum (2, 5)
addition = obj1 + obj2
print("real of addition:", addition.real)
print("imaginary of addition:", addition.img)
```

♦ Access Modifiers

- Public attributes are those that be can be accessed inside the class and outside the class.

```
class Employee:
    def __init__(self, ID, salary):
        # all properties are public
        self.ID = ID
        self.salary = salary
    def displayID(self):
        print("ID:", self.ID)
John = Employee(3789, 2500)
John.displayID()
print(John.salary)
```

- Private attributes cannot be accessed directly from outside the class but can be accessed from inside the class.

```
class Employee:
    def __init__(self, ID, salary):
        self.ID = ID
        self.__salary = salary # salary is a private property
John = Employee(3789, 2500)
print("ID:", John.ID)
print("Salary:", John.__salary) # this will cause an error
```

- Accessing Private attributes in the Main Code

```
class Employee:
    def __init__(self, ID, salary):
        self.ID = ID
        self.__salary = salary  # salary is a private property
John = Employee(3789, 2500)
print(John._Employee__salary)  # accessing a private property
```

- Private methods: methods are usually public since they provide an interface for the class properties and the main code to interact with each other.

```
class Employee:
    def __init__(self, ID, salary):
        self.ID = ID
        self.__salary = salary  # salary is a private property
    def displaySalary(self):  # displaySalary is a public method
        print("Salary:", self.__salary)
    def __displayID(self):  # displayID is a private method
        print("ID:", self.ID)
John = Employee(3789, 2500)
John.displaySalary()
John.__displayID()  # this will generate an error
```

Data hiding

Encapsulation

Binds the data and the methods to manipulate that data together in a single class.

Advantages of Encapsulation

- Classes make the code easy to change and maintain.
- Properties to be hidden can be specified easily.
- We decide which outside classes or functions can access the class properties.

```
class Employee:
    def __init__(self, name=None):  # defining initializer
        self.__name = name
    def setUsername(self, n):
        self.__name = n
    def getUsername(self):
        return (self.__name)
Anna = Employee('Anna B.')
print('Before setting:', Anna.getUsername())
Anna.setUsername('Anna K.')
print('After setting:', Anna.getUsername())
```

Abstraction

Set of methods and properties that a class must implement in order to be considered a duck-type instance of that class. (See Polymorphism Using Duck Typing for more details)

```
from abc import ABC, abstractmethod
# Shape is a child class of ABC, it will
# prevent users from making a Shape class
# object, because Shape object cannot stand on # its own.
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
class Square(Shape):
    def __init__(self, length):
        self.length = length
    def area(self):
        return (self.length * self.length)
    def perimeter(self):
        return (4 * self.length)
shape = Shape()
# The code will not compile since Shape has abstract methods
without method definitions in it
```

Inheritance

Inheritance provides a way to create a new class from an existing class and is the <u>IS A</u> relation between classes

The new class is a specialized version of the existing class such that it inherits all the non-private fields (variables) and methods of the existing class. The existing class is used as a starting point or as a base to create the new class.

- Parent Class (Super Class or Base Class): This class allows the *re-use* of its public properties in another class.
- Child Class (Sub Class or Derived Class): This class is the one that *inherits* or *extends* the superclass.

```
class Vehicle: # defining the parent class
    def __init__(self, make, model):
        self.make = make
        self.model = model
    # print method in the parent class
    def printDetails(self):
        print("Manufacturer:", self.make)
        print("Model:", self.model)
class Car(Vehicle): # defining the child class
    def __init__(self, make, model, doors):
        Vehicle.__init__(self, make, model)
        self.doors = doors
    # print method in the child class
    def printCarDetails(self):
        self.printDetails()
        print("Name:", self.doors)
# creating object of the Car class
car = Car("Toyota", "2019", 4)
car.printCarDetails()
```

Using Super() Function

- Using Initializers

```
class Vehicle: # defining the parent class
    def __init__(self, make, model):
        self.make = make
        self.model = model
    # print method in the parent class
    def printDetails(self):
        print("Manufacturer:", self.make)
        print("Model:", self.model)
class Car(Vehicle): # defining the child class
    def __init__(self, make, model, doors):
        super().__init__(make, model)
        self.doors = doors
    # print method in the child class
    def printCarDetails(self):
        self.printDetails()
        print("Name:", self.doors)
# creating object of the Car class
car = Car("Toyota", "2019", 4)
car.printCarDetails()
```

- Calling parent class

```
class Vehicle:  # defining the parent class
    # print method in the parent class
    def printOut(self):
```

```python
        print("I am from the Vehicle Class")
class Car(Vehicle):  # defining the child class
    # print method in the child class
    def printOut(self):
        super().printOut ()
        print("I am from the Car Class")
# creating object of the Car class
Honda = Car()
Honda.printOut ()  # calling the Car class method printOut()
```

♦ Types of Inheritance:

- **Single**

```python
class Vehicle:  # parent class
    # defining the set
    def setSpeed(self, speed):
        self.Speed = speed
        print("Speed is set to", self.Speed)

class Car(Vehicle):  # child class
    def startEngine(self):
        print("Engine is now running.")
# creating object of the Car class
Honda = Car()
# accessing method from the parent class
Honda.setSpeed(70)
# accessing method from its own class
Honda.startEngine()
```

- **Multi-level**

```python
class Vehicle:  # parent class
    # defining the set
    def setSpeed(self, speed):
        self.Speed = speed
        print("Speed is set to", self.Speed)
class Car(Vehicle):  # child class
    def startEngine(self):
        print("Engine is now running.")
class Hybrid(Car):  # child class of Car
    def turnOnHybrid(self):
        print("Hybrid mode is now on.")
# creating an object of the Hybrid class
ToyotaPrius = Hybrid()
# accessing methods from the parent class
ToyotaPrius.setSpeed(90)
# accessing method from the parent class
ToyotaPrius.startEngine ()
# accessing method from the parent class
ToyotaPrius.turnOnHybrid()
```

- **Hierarchical**

```python
class Vehicle:  # parent class
    # defining the set
    def setSpeed(self, speed):
        self.Speed = speed
        print("Speed is set to", self.Speed)

class Car(Vehicle):  # child class of Vehicle
    pass

class Motorcycle(Vehicle):  # child class of Vehicle
    pass
# creating object of the Car class
Crysler = Car()
# accessing method from the parent class
Crysler.setSpeed(100)
# creating an object of the Motorcycle class
BMW = Motorcycle()
# accessing method from the parent class
BMW.setSpeed(40)
```

- **Multiple**

```python
class Engine():
    def setEngine(self, engine):
        self.engine = engine
class ElectricEngine():
    def setBattery(self, battery):
        self.battery = battery
# Child class inherited from Engine and ElectricEngine
class HybridEngine(Engine, ElectricEngine):
    def printDetails(self):
        print("Engine Power:", self.engine)
        print("Battery Capacity:", self.battery)
car = HybridEngine()
car.setBattery ("250 W")
car.setEngine("2000 CC")
car.printDetails()
```

- **Hybrid**

```python
class Engine:  # Parent class
    def setEngine(self, engine):
        self.engine = engine
# Child class inherited from Engine
class TankCapacity(Engine):
    def setTank(self, tank):
        self.tank = tank
# Child class inherited from Engine
class ElectricEngine(Engine):
    def setBattery(self, battery):
        self.battery = battery
# Child class inherited from TankCapacity and ElectricEngine
class HybridEngine(Tank, ElectricEngine):
    def printDetails(self):
        print("Engine:", self.engine)
        print("Tank:", self.tank)
        print("Battery:", self.battery)
car = HybridEngine()
car.setEngine("2000 CC")
car.setBattery ("250 W")
car.setTank ("20 Liters")
car.printDetails()
```

Polymorphism

In programming, polymorphism refers to the same object exhibiting different forms and behaviors.

Polymorphism Using Methods

```
class Rectangle():
    # initializing sides of all rectangles to 4
    def __init__(self, width=0, height=0):
        self.width = width
        self.height = height
        self.sides = 4
    # method to calculate area of rectangle
    def getArea(self):
        return (self.width * self.height)
class Circle():
    # initializing sides of all circles to 0
    def __init__(self, radius=0):
        self.radius = radius
        self.sides = 0
    # method to calculate area of circle
    def getArea(self):
        return (self.radius * self.radius * 3.142)
print("Sides of a rectangle are", str(Rectangle(6, 10).sides))
print("Area of rectangle is:",
str(Rectangle(6, 10).getArea()))
print("Sides of a circle are", str(Circle(7).sides))
print("Area of circle is:", str(Circle(7).getArea()))
```

Polymorphism Using Inheritance

```
class Shape:
    def __init__(self):  # initializing sides of all shapes to 0
        self.sides = 0
    def getArea(self):
        pass
# derived form Shape class
class Rectangle(Shape):
    # initializing sides of all rectangles to 4
    def __init__(self, width=0, height=0):
        self.width = width
        self.height = height
        self.sides = 4
    # method to calculate area of rectangle
    def getArea(self):
        return (self.width * self.height)
# derived form Shape class
class Circle(Shape):
    # initializing radius
    def __init__(self, radius=0):
        self.radius = radius
    # method to calculate area of circle
    def getArea(self):
        return (self.radius * self.radius * 3.142)
print("Area of rectangle is:",
str(Rectangle(6, 10).getArea()))
print("Area of circle is:", str(Circle(7).getArea()))
```

Polymorphism Using Duck Typing

Duck typing is one of the most useful concepts in Object-Oriented Programming in Python. Using duck typing, one can implement polymorphism without using inheritance. The object is a duck that if an object *quacks* like a duck, *swims* like a duck, *eats* like a duck or in short, *acts* like a duck.

```
class Cat:
    # method to print sound of cat
    def Speak(self):
        print("Meow meow")
class Dog:
    # method to print sound of dog
    def Speak(self):
        print("Woof woof")
class AnimalSound:
    def Sound(self, animal):
        animal.Speak()
animal = AnimalSound()
animal.Sound(Cat())
animal.Sound(Dog())
```

Aggregation

Aggregation follows the <u>Has-A</u> model. This creates a parent-child relationship between two classes, with one class owning the object of another. Class A and class B have a <u>Has-A</u> relationship if one or both need the other's object to perform an operation, but both class objects can exist independently of each other.
<u>In aggregation, the lifetime of the owned object does not depend on the lifetime of the owner.</u>

```
class Country: # Parent class
    def __init__(self, name=None, population=0):
        self.name = name
        self.population = population
    def printDetails(self):
        print("Country Name:", self.name)
        print("Country Population", self.population)
class Person: # Child class
    def __init__(self, name, country):
        self.name = name
        self.country = country
    def printDetails(self):
        print("Person Name:", self.name)
        self.country.printDetails()
c = Country("Narumu", 1500)
p = Person("John", c) # Person (p) object has County object ( c)
p.printDetails()
# delete the object p
del p
print("")
c.printDetails() # Country class can exist w/o Person class
```

Composition

Composition is the practice of accessing other class objects in a class. In such a scenario, the class which creates the object of the other class is known as the *owner* and is responsible for the lifetime of that object.
Composition relationships are <u>Part-of</u> relationships where the *part* must constitute a segment of the whole object. One can achieve composition by adding smaller parts of other classes to make a complex unit.
<u>In composition, the lifetime of the owned object depends on the lifetime of the owner.</u>

```
class Engine: # Parent class
```

```
    def __init__(self, capacity=0):
        self.capacity = capacity
    def printDetails(self):
        print("Engine power:", self.capacity)
class Model: # Parent class
    def __init__(self, model=None):
        self. model = model
    def printDetails(self):
        print("Car model:", self.model)
class Car: # Child class that contains parts of Engine and Model
class
    def __init__(self, eng, model, color):
        self.eObj = Engine(eng)
        self.mObj = Model(model)
        self.color = color
    def printDetails(self):
        self.eObj.printDetails()
        self.mObj.printDetails()
        print("Car color:", self.color)
# creating an object of the Car class
car = Car(1600, 4, 2, "Grey")
car.printDetails()
```