Object Oriented Programming

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

For instance, an object could represent a person with **properties(variables)** like a name, age, and address and **behaviors(methods)** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

There are 3 important concepts of OOP:

1. Classes
2. Objects
3. Object variable (or) Reference variables

Class:

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

A class is a blueprint for how something should be defined. It doesn't actually contain any data. The `Dog` class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

Defining a Class:

```
class Demo:
    pass
```

1. There can be multiple no of classes in a single python file.
2. Each class may contain variables, methods and constructors.
3. A single class can have any no of variables and methods.
4. Each class name should be distinct.

Object:

An object or **instance** is an object that is built from a class and contains real data. An instance of the `Dog` class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

1. An object is a reference memory for a class.
2. Whenever we create an object, memory get allocated for the class.
3. Objects are created for classes.
4. A class can have any no of objects with distinct names.
5. Each object has its own execution.
6. Objects can be created and destroyed when needed.
7. An object can refer a class member like variable or method for any no of times in another class or outside the class.

Creating an Object:

```
class Demo:
    pass

obj = Demo()
```

Reference Variable (or) Object Variable:

By using a 'reference variable' we can perform required operations on a given class.

1. Any variable which is used to refer object is called 'reference variable'.
2. Per one single object of a class, any no of reference variables can be maintained.
3. Using a reference variable, we can access methods and variables of a class.
4. An object can have any no of 'reference variables'.

```
Obj1 = Demo()
Obj2 = Demo()
Obj3 = Demo()

3 reference variables for class Demo()
```

Example 1:

```
# defining class Employee
class Employee:
   'class variable'
   empCount = 100

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

# object of class Employee
obj = Employee()

# accessing members of class Employee using its object
obj.displayCount()
print("Total emp count is: %d" %(obj.empCount))
```

Constructors:

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created.

Example 2:

```
# Constructor:
#   Is like a method.
#   Constr's are only for classes
#   Constr's are used to initialize the class var's
#   Constr's name should always be like '__init__(self)'
#   Constr's are called at the time of creating objects for the class.
#   Constr's can have parameters

class Demo:

    # defining a parameterized constructor
    def __init__(self,x,y):
        print("Iam the constr....")
        self.i = x
        self.j = y

    def display(self):
        print("From method display: ", self.i + self.j)
        pass

# calling a constr
obj = Demo(1000,20)
obj.display()

obj1 = Demo(2000,10)
obj1.display()

a,b = input("Enter two numbers: ").split()
a,b = int(a),int(b)
obj2 = Demo(a,b)
obj2.display()
```

Example 3:

```
# defining class Employee with constructor
class Employee:
   'class variable'
   empCount = 0

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

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

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

# object of class Employee
# create first object of Employee class
emp1 = Employee("Venkat", 2000)
```

```python
# create second object of Employee class
emp2 = Employee("Satish", 5000)

# accessing members of class Employee using its object
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Note:

- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
- Python will provide a default constructor if no constructor is defined.

Types of Constructors

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor

Constructor With Default Values

Example 4:

```python
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=7):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)

# creating object of the Student class
emma = Student('Emma')
emma.show()

kelly = Student('Kelly', 13)
kelly.show()
```

Example 5:

```
# Chainaing the constructor

class Vehicle:
    # Constructor of Vehicle
    def __init__(self, engine):
        print('Inside Vehicle Constructor')
        self.engine = engine

class Car(Vehicle):
    # Constructor of Car
    def __init__(self, engine, max_speed):
        super().__init__(engine)
        print('Inside Car Constructor')
        self.max_speed = max_speed

class Electric_Car(Car):
    # Constructor of Electric Car
    def __init__(self, engine, max_speed, km_range):
        super().__init__(engine, max_speed)
        print('Inside Electric Car Constructor')
        self.km_range = km_range

# Object of electric car
ev = Electric_Car('1500cc', 240, 750)
print(f'Engine={ev.engine}, Max Speed={ev.max_speed}, Km
range={ev.km_range}')
```

Destructors to Destroy the Object

Destructor is a special method that is called when an object gets destroyed. On the other hand, a constructor is used to create and initialize an object of a class.

Destructor is used to perform the clean-up activity before destroying the object, such as closing database connections or file handle.

Python has a garbage collector that handles memory management automatically. For example, it cleans up the memory when an object goes out of scope.

But it's not just memory that has to be freed when an object is destroyed. We **must release or close the other resources object were using**, such as open files, database connections, cleaning up the buffer or cache. **To perform all those cleanup tasks, we use destructor** in Python.

In Python, destructor is not called manually but completely automatic. **destructor gets called in the following two cases**

- When an object goes out of scope or
- The reference counter of the object reaches 0.

In Python, The special method `__del__()` is used to define a destructor. For example, when we execute `del object_name` destructor gets called automatically and the object gets garbage collected.

Example 1:

```python
class Student:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)
    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma')
s1.show()

# delete object
del s1
```

Access Modifiers in Python(Data Abstraction)

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from outside oclass.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes

Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

Example 1:

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Private Member

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

Example 2:

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

We can access private members from outside of a class using the following two approaches

- Create public method to access private members
- Use name mangling

Let’s see each one by one

Public method to access private members

Example: Access Private member outside of a class using an instance method

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show()
```

Name Mangling to access private members

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this `_classname__dataMember`, where `classname` is the current class, and data member is the private variable name.

```python
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```

Protected Member

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore `_`.

Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

Example: Protected member in inheritance.

```python
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

- When we want to avoid direct access to private variables

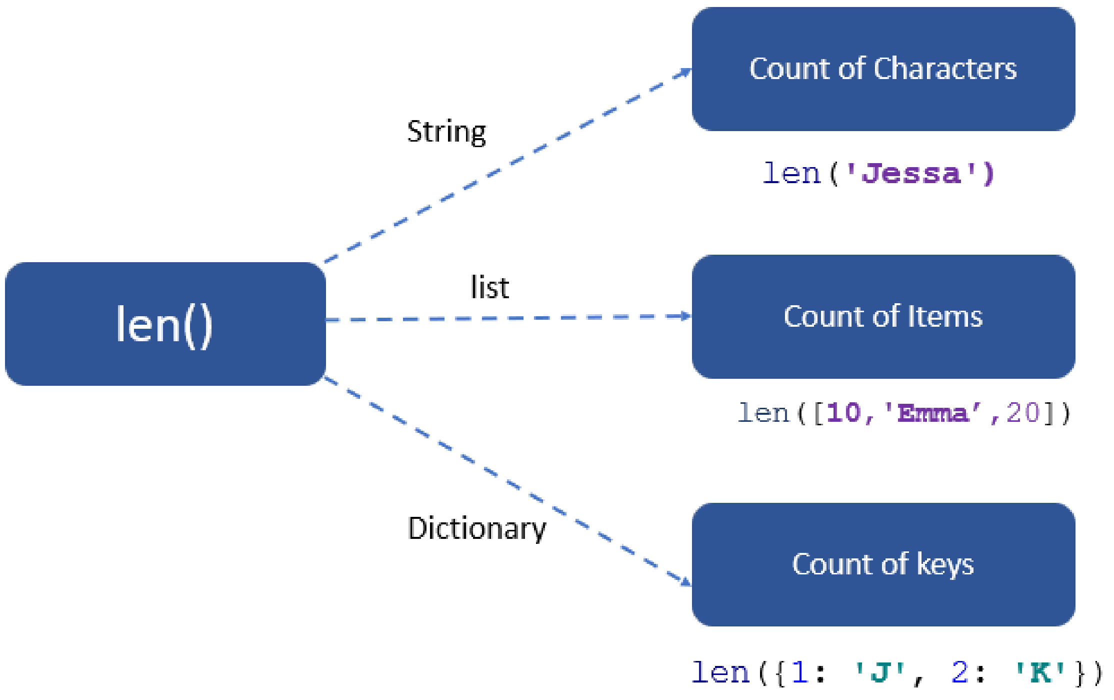
- To add validation logic for setting a value

Example

```
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

Polymorphism in Python

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.



Polymorphic len() function

The **process of re-implementing the inherited method in the child class** is known as Method Overriding. We can implement Polymorphism using 'Method Overriding'.

```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def change_gear(self):
        print('Vehicle change 6 gear')
```

```python
# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

NOTE: As you can see, due to polymorphism, the Python interpreter recognizes that the max_speed() and change_gear() methods are overridden for the car object. So, it uses the one defined in the child class (Car)

On the other hand, the show() method isn't overridden in the Car class, so it is used from the Vehicle class.

Overrride Built-in Functions

In this example, we will redefine the function `len()`

```python
class Shopping:
    def __init__(self, basket, buyer):
        self.basket = list(basket)
        self.buyer = buyer

    def __len__(self):
        print('Redefine length')
        count = len(self.basket)
        # count total items in a different way
        # pair of shoes and shir+pant
        return count * 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')
print(len(shopping))
```

Polymorphism in Class methods

In the below example, `fuel_type()` and `max_speed()` are the instance methods created in both classes.

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```

Polymorphism with Function and Objects

```python
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```