OBJECT ORIENTED PROGRAMMING IN PYTHON

Python is a high level, and interpreted language known for its readability, simplicity, and versatility. It is like a master key capable of unlocking countless possibilities. It is a very powerful programming language that can be used for many different things, like developing websites using python frameworks, analyzing data, and integrating systems.

The main key feature of python is Object Oriented Programming that organizes code into objects, encapsulating data and behavior and also used in django models to inherit properties from one model to another model. Python OOPs promotes clean, modular, and reusable code through the creation of classes and objects.

This article explores the OOP, its principles, uses and conclusion.

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm which is intended to solve real world problems. OOP is used in many languages such as Python, Java, C++, . Python is a versatile language that supports OOP, which is built around several key features that allow developers to structure their code in a modular, reusable, and maintainable way.

Imagine a car manufacturing company where each car model is built using a blueprint (a class). The blueprint defines the properties and behaviors (methods) that all cars of that model will have. An actual car built from the blueprint is an object, which is an instance of the class. The cars share common properties and behaviors from the blueprint, but they can also have unique variations or custom features, which demonstrates the concepts of inheritance and polymorphism.

Principles of OOP:

1) Class and Object
2) Encapsulation
3) Inheritance
4) Polymorphism
5) Abstraction

1) Class and Object:

Classes and objects are fundamental concepts in object-oriented programming (OOP). A class is a blueprint or template that defines the structure and behavior of an object, while an object is an instance of a class with its own set of attributes(properties) and methods.

Classes in Python are defined using the class keyword followed by the class name. The body of the class contains properties and methods i.e., instance methods, static methods, or class methods. These class and static methods are declared using a decorator(@). In the class, __init__ method passes the self keyword as a parameter, which is a constructor used to initialize the objects. In django, model is a python class.

Objects in Python hold different values for their attributes but share the same structure defined by the class. When we create an object for the class, Python will automatically call the constructor.

Example:

```
#Creating Class
class Car:
    def __init__(self, name):
        self.name = name
    def display(self):
        return self.name
#Creating objects with different values
result1 = Car("Ford")

#Printing the result
print(result1.display())    #Ford
```

In this example, A Person class is defined using the class keyword. This class will serve as the blueprint for creating objects representing individual people.
The __init__ method is a special method called a constructor, which is automatically called when a new object is created. It initializes the instance variable name, using the self keyword to refer to the object itself. Here, self is a reference to the current instance of the class. The display method returns the value of the name attribute that is stored in the object.
The object result1, is created using the Car class, with the value "Ford". The attributes and methods of the object using dot notation. The display() method is called with the object reference then it will return the name. In the print statement, the display() method is called to print the name attribute.

Real World Example:
Imagine a class as a blueprint for a smartphone, containing specifications like the screen size, storage capacity, and camera resolution. An object is like an actual smartphone built from that blueprint, with its own unique color, serial number, and other attributes.

2) Encapsulation:

Encapsulation is an object-oriented concept that combines data (attributes) and functions (methods) within a class, giving controlled access to an important object. It helps achieve a clear separation of concerns, improving maintainability and reducing the chance of unintended code interference. Encapsulation is achieved using naming conventions that indicate the intended visibility of class members using access modifiers.

Access Modifiers: Access modifiers are keywords or conventions that control the visibility and access levels of class attributes and methods.Python has three levels of access modifiers: public, protected, and private.Attributes and Methods starting with a double underscore (__) are considered private and should not be accessed directly from outside the class, starting with a single underscore (_) are considered as protected and should not accessed from outside the class except subclass. Public is accessible to all instances of the class.

Features of Encapsulation:
Data Hiding: Encapsulation hides the internal implementation of an object to the users. This prevents direct access to the object's properties from outside the class. Example, Mobile internal implementation is not visible to the user.
Getters and Setters: To control access to attributes, you can use getter and setter methods. Getter methods are used to retrieve the value of an attribute, while setter methods are used to modify it.

Example :

```
#Creating class BankAccount
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def get_deposit(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds!")

# Creating an object of BankAccount
account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.get_deposit())  # Output: Balance: $1300

# Attempting to access private attribute
# print(account.__balance)  # This will raise an AttributeError
```

The BankAccount class represents a bank account with attribute (balance) and methods (get_deposit, deposit, withdraw). The balance attribute is prefixed with double underscores (balance), making it "private" to the class, restricting direct access. Instead, users interact with the balance through the provided methods (get_deposit, deposit, withdraw), ensuring a controlled and secure way to manage the account. When we call the get_deposit() method it returns the balance because we are accessing balance in the method but when we try to access balance directly with object reference, it will raise an error because balance is a private variable.

Real World Example:
Imagine a car's internal combustion engine. It consists of various components (attributes) and offers some functionality (methods) like starting and stopping. The car engine's design (encapsulation) protects critical components from being directly accessed or modified, ensuring proper operation and safety.

3) Inheritance :

Inheritance is a one of the fundamental concept in object-oriented programming, where one class (child class) inherits or takes on the properties and methods of another class (parent class). Inheritance allows for code reusability and modularity, making it easier to create and maintain complex applications.
In Python inheritance, the super() function allows you to call a method from a parent class, usually within methods of a child class.

- **Base Class:** Base class is called Parent class or Super class. Base class is the class whose properties and methods are inherited by another class.
- **Derived Class:** Derived Class is called Child Class or Sub Class. Derived Class is the class that inherits attributes and methods from the base class. It can add new attributes and methods or override existing ones.

Syntax:

```
class BaseClass:
    # Base class attributes and methods
    pass

class DerivedClass(BaseClass):
    # Derived class attributes and methods
    pass
```

Types of Inheritance:

1) Single Inheritance: A derived class inherits from a single base class.

Syntax:

```
class BaseClass:
    # Base class attributes and methods
    pass

class DerivedClass(BaseClass):
    # Derived class attributes and methods
    pass
```

2) Multiple Inheritance: A derived class can inherit from multiple base classes. In Python, Multiple Inheritance is allowed but in Java, multiple inheritance is not allowed but it is allowed by using Interface.

Syntax:

```
class BaseClass1:
    # Base class1 attributes and methods
    pass

class BaseClass2:
    # Base class2 attributes and methods
    pass
class DerivedClass(BaseClass1, BaseClass2):
    # Derived class attributes and methods
    pass
```

3) MultiLevel Inheritance: A class can inherit from a derived class, forming a multi-tiered hierarchy.
Syntax:

```
class BaseClass:
    # Base class attributes and methods
    pass
class DerivedClass1(BaseClass):
    # Derived class1 attributes and methods
    pass
class DerivedClass2(DerivedClass1):
    # Derived class2 attributes and methods
    pass
```

4) Hierarchical Inheritance: Multiple derived classes inherit from the same base class.
Syntax:

```
class BaseClass:
    # Base class attributes and methods
    pass

class DerivedClass1(BaseClass):
    # Derived class1 attributes and methods
    pass
class DerivedClass2(BaseClass):
    # Derived class2 attributes and methods
    pass
```

Inheritance Example:

```
#Define a base class called Animal
class Animal:
    def __init_(self, name):
        self.name name

    def speak(self):
        return "I am an anima."

# Define a subclass called Dog that inherits from Animal
class Dog (Animal):
    def speak(self):
        return "Woof! Woof!"

# Create an instance of Dog
my_dog = Dog("Buddy")
print(my_dog.name)          # Output: Buddy
print(my_dog.speak())       # Output: Woof! Woof!
```

In this example, we first define a base class called Animal with a speak() method. We then create a subclass called Dog that inherits from Animal. In the Dog subclass, we override the speak() method to return a different message. When we create an instance of the Dog class and call its speak() method, it returns the overridden message.

Real World Example:
Think of inheritance like a family tree. A child inherits certain traits and characteristics from their parents, such as eye color or height. Similarly, in programming, a subclass inherits properties and methods from a base class, which can be further extended or overridden as needed.

4) Polymorphism:

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types, making it easier to write more flexible and extensible code.

Types of Polymorphism:

1) Method Overriding (Runtime Polymorphism): Method overriding occurs when a derived class provides a specific implementation of a method that is already defined in its base class. The version of the method that is executed is determined at runtime.

Example:

```
class Animal:
   def speak(self):
      return "Animal speaks"

class Dog(Animal):
   def speak(self):
       return "Woof!"

 # Creating objects of Dog and Cat
 animal = Animal()
 dog = Dog()
 cat = Cat()
 print(animal.speak())  # Output: Animal speaks
 print(dog.speak())     # Output: Woof!
```

2) Method Overloading (Compile-time Polymorphism): Python does not support method overloading in the traditional sense like some other languages (e.g., Java or C++), you can achieve similar behavior by using default arguments or variable-length arguments.

Example:

```
class MathOperations:
     def add(self, a, b, c=0):  # c is a default parameter
           return a + b + c
 math = MathOperations()
 print(math.add(2, 3))      # Output: 5
 print(math.add(2, 3, 4))   # Output: 9
```

3) Operator Overloading: Operator overloading is a programming concept that allows us to define custom behavior for operators (such as +, -, *, /, etc.) when they are used with user-defined data types, like classes. By overloading operators, we can use them in more intuitive and expressive ways, making the code easier to read and understand.

4) Duck typing: Python's dynamic typing allows us to use any object that provides the required behavior, without explicitly checking its class.

Example:

```
class Car:
    def start(self):
        return "The car starts."
class Bicycle:
    def start(self):
        return "The bicycle starts."
 class Bus:
    def start(self):
         return "The bus starts.”
 #A function that demonstrates polymorphism
 def start transport (transport):
     print(transport.start())
 #Create instances of Car, Bicycle, and Bus
 my_car = Car()
 my_bicycle = Bicycle()
 my_bus = Bus()

 #Use the start_transport function with different types of objects
 start_transport(my_car)         # Output: The car starts.
 start_transport(my_bicycle)    # Output: The bicycle starts.
 start_transport (my_bus)        # Output: The bus starts.
```

In this example, we define three different classes, Car, Bicycle, and Bus, each with their own start() method. We then create a function called start_transport() that accepts any object with a start() method, demonstrating polymorphism. When we pass instances of Car, Bicycle, and Bus to this function, it calls their respective start() methods, regardless of their specific class.

Real world Example:
Consider different modes of transportation, such as cars, bicycles, and buses. They all have a common function: to transport people from one place to another. Despite their differences, we can treat them as vehicles and use their common functionality, like starting and stopping.

5) Abstraction:

Data abstraction in Python is a programming concept that hides complex implementation details while exposing only essential information and functionalities to users. This enhances the modularity and simplicity of code, making it easier to work with complex systems by focusing on high-level functionality rather than low-level details. In Python, we can achieve data abstraction by using abstract classes and abstract classes can be created using abc (abstract base class) module and abstract method of abc module.

Abstraction class: Abstract class is a class in which one or more abstract methods are defined. When a method is declared inside the class without its implementation is known as abstract method

Abstract Method: In Python, abstract method feature is not a default feature. To create abstract method and abstract classes we have to import the “ABC” and “abstractmethod” classes from abc (Abstract Base Class) library.

Abstract method of base class force its child class to write the implementation of all abstract methods defined in base class. If we do not implement the abstract methods of base class in the child class then our code will give error. This abstract method created using @abstractmethod decorator.

Example:

```
from abc import ABC, abstractmethod
class Animal (ABC):
    @abstractmethod
     def speak(self):
         pass
class Dog(Animal):
    def speak(self):
         return "Woof!"
class Cat(Animal):
    def speak(self):
          return "Meow!"
#animal Animal() - # This will raise an error, as Animal is an abstract class
dog = Dog()
print(dog.speak())     # Output: Woof!
cat = Cat()
print(cat.speak())      # Output: Meow!
```

In this example, we first import the ABC (Abstract Base Class) and abstractmethod from the abc module. We then define an abstract class Animal by inheriting from ABC. Inside the Animal class, we declare an abstract method speak() using the @abstractmethod decorator. This method has no implementation in the Animal class.
We then create two concrete subclasses, Dog and Cat, which inherit from the Animal class. Both subclasses provide an implementation for the speak() method. Finally, we instantiate Dog and Cat objects and call their speak() methods, which return "Woof!" and "Meow!" respectively.

Real World Example:
Imagine an abstract class as a blueprint for a building. You cannot live in the blueprint itself, but you can use it as a base to construct different types of buildings (subclasses) with specific features and designs. Similarly, abstract methods are like the essential rooms that must be built in the final structure, but the details and functionalities of each room depend on the specific building being constructed.

Uses Of OOP in Python:

1) Used in python frameworks like Django, Flask which are used to develop web applications.
2) Libraries like Pandas and Scikit-learn use OOP for data manipulation and analysis
3) OOP makes it easier to implement design patterns (e.g., Singleton) which are widely used to solve common software design problems.

Conclusion:

Object-Oriented Programming (OOP) in Python is a powerful paradigm that allows developers to solve real-world problems more naturally and efficiently. With the core principles of **classes and objects**, **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**, Python's OOP approach promotes cleaner, modular, and reusable code. It simplifies complex systems by breaking them down into smaller, manageable components while enhancing maintainability and scalability.