Introduction to Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. These objects represent real-world entities and can contain data (attributes) and methods (functions).

Key Concepts of OOP

1. **Class**: A blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
2. **Object**: An instance of a class. It is created using the class blueprint and can have its own unique data.
3. **Attributes**: Variables that belong to an object. They describe the object's state.
4. **Methods**: Functions that belong to an object. They define the behavior of the object.
5. **Inheritance**: A way to form new classes using classes that have already been defined. It helps to reuse and enhance existing code.
6. **Encapsulation**: The bundling of data and methods that operate on the data within one unit, e.g., a class. It restricts direct access to some of the object's components.
7. **Polymorphism**: The ability to present the same interface for different data types. It allows methods to be used interchangeably.

OOP Use Case

- **Creating Reusable Components**: Design reusable classes for common functionalities like logging, data access, or UI elements.
- **Modeling Real-World Entities**: Represent real-world entities like customers, products, and orders in an e-commerce application with classes.
- **Encapsulation**: Encapsulate data and related methods within classes to hide implementation details and expose a clear interface.
- **Inheritance for Reusability and Extensibility**: Create a base class with common functionality and extend it in derived classes for specialized behavior.
- **Design Patterns Implementation**: Implement common design patterns (Singleton, Factory, Observer) to solve recurring design problems.
- **APIs and Web Services**: Develop APIs and web services with classes to manage requests, responses, and data handling.
- **Database Interaction**: Use classes to represent database tables and manage CRUD operations through Object-Relational Mapping (ORM) frameworks.
- **Data Structures**: Implement custom data structures (linked lists, trees, graphs) using classes and objects.
- **Testing and Mocking**: Use classes to create mock objects for unit testing, isolating the code being tested from its dependencies.

- **Security and Access Control**: Implement security features like authentication and authorization using classes to manage user roles and permissions.
- **File and Resource Management**: Manage file operations, network connections, and other resources with classes that ensure proper resource handling and cleanup.
- **Business Logic Implementation**: Encapsulate complex business rules and logic within classes to maintain clarity and separation from other parts of the application.

⬛ Creating Classes and Objects

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

```
class MyClass:
    x = 5
    y = 10

    def addTwo(self):
        z=10
        print(self.x+self.y+z)

p1 = MyClass()
p1.addTwo()
```

- `self` is essential for methods to access or modify the instance's attributes.
- It distinguishes between instance attributes and local variables within methods.
- Without `self`, the method would not know which instance's attributes to operate on.

⬛ Class Variables and Methods

Class Variables

Class variables are variables that are shared among all instances of a class. They are defined within the class but outside any instance methods.

Class Methods

Class methods are methods that operate on the class itself rather than on instances of the class. They are defined using the `@classmethod` decorator and take `cls` as their first parameter, which refers to the class itself.

```python
class MyClass:
#Class Variables
    x = 5
    y=10

#Class Methods

    def addTwo(self,extra):
        print(self.x+self.y+10)

    def addNew(self):
        self.addTwo(10)


p1 = MyClass()
p1.addTwo(100)
p1.addNew()
```

⬛ Constructors

- Constructors are special methods in Python that are automatically called when an object of a class is created.
- The primary purpose of a constructor is to initialize the instance variables of a class.
- In Python, the constructor method is always named `__init__`.

Without Parameter

```python
class MyClass:
    def __init__(self):
        print("I am constructor")


p1 = MyClass()
```

With Parameter

```python
class MyClass:
    def __init__(self,msg):
        print(msg)

p1 = MyClass("I am constructor")
```

Change Class Variable Using Constructor Params

```
class MyClass:
    y=0
    x=0
    def __init__(self):
        self.x = 10
        self.y = 20

p1 = MyClass()

print(p1.x)
print(p1.y)
```

⬛ Instance Variables and Methods

Instance Variables:

Attributes that are unique to each instance of a class. They are typically defined in the `__init__` method.

Instance Methods:

Functions that operate on instance variables. They take `self` as the first parameter to refer to the instance on which the method is called.

```
class Employee:

    def __init__(self,company_name,employee_count):
            self.company_name = company_name # Instance variable
            self.employee_count = employee_count  # Instance variable


    # Instance method
    def display_company_info(self):
        print(f"Company Name: {self.company_name}, Total Employees:
{self.employee_count}")


emp1 = Employee("Google",20000)

emp1.display_company_info()
```

⬛ Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit attributes and methods from another class.
- The class that is inherited from is called the **parent class** or **base class**, and the class that inherits is called the **child class** or **derived class**
- This mechanism promotes code reusability and can help in creating a more organized and manageable code structure.

PASS

- When you want to define a class or function that you plan to implement later, you can use `pass` to avoid syntax errors.
- When you have a loop or conditional statement and you do not want it to execute any code for a specific condition.
- When you are writing code and want to temporarily skip the implementation of a certain block while you focus on other parts.

Single Inheritance

Definition: A child class inherits from one parent class.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son1(Father):
    pass

son1 = Son1()
print(son1.addTwo(5, 3))  # Output: 8
```

Multiple Inheritance

Definition: A child class inherits from more than one parent class.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Mother:
    def subtractTwo(self, num1, num2):
        return num1 - num2

class Son1(Father, Mother):
```

```python
    pass

son1 = Son1()
print(son1.addTwo(5, 3))       # Output: 8
print(son1.subtractTwo(5, 3))  # Output: 2
```

Multilevel Inheritance

Definition: A child class inherits from a parent class, which in turn inherits from another parent class.

```python
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son1(Father):
    pass

class Grandson(Son1):
    pass

grandson = Grandson()
print(grandson.addTwo(5, 3))  # Output: 8
```

⬛ Constructor at inheritance situations

When Parent class has, but the child class has not

```python
class Father:
    x=10
    y=10
    def __init__(self):
        print(self.x+self.y)

class Son(Father):
    pass

obj = Son()
```

When child class has, but the parent class has not

```python
class Father:
    x=10
```

```
    y=10

class Son(Father):
    def __init__(self):
        print(self.x + self.y)

obj = Father()
```

When Parent and child both has contractor

```
class Father:
    a = 10
    b = 20

    def __init__(self):
        print(self.a + self.b)


class Son(Father):
    x = 100
    y = 200

    def __init__(self):
        super().__init__()
        print(self.x + self.y)

obj = Son()
```

Why is `super().__init__()` needed?

- **Inheritance of Initialization Logic**: The `Father` class might contain essential setup logic in its constructor that the `Son` class needs. Without calling `super().__init__()`, that logic will not be executed, and any initialization in `Father` would be skipped.
- **Avoid Code Duplication**: If the `Son` class should also perform the initialization defined in `Father`, `super().__init__()` ensures that the constructor of the base class runs without duplicating the logic in `Son`.
- **Object Creation Chain**: In Python's inheritance, calling `super().__init__()` is a best practice to ensure proper object creation, especially in complex cases where multiple classes are involved.

Accessing the Parent's Constructor

```
class Father:
    def __init__(self):
        print("Father's constructor")

class Son(Father):
    def __init__(self):
        # Access Father's constructor using super()
```

```python
        super().__init__()
        print("Son's constructor")

obj = Son()
```

⬛ Static Properties in inheritance situations

If Parent has static properties, child can access as it is like parent

```python
class Father:
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Father.a + Father.b)

class Son(Father):
    pass

Son.addtwo()
Father.addtwo()

print(Father.a)
print(Father.b)
```

If Child has static properties, Parent can't access as it is like child

```python

class Father:
    pass

class Son(Father):
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Son.a + Son.b)

Son.addtwo()
Father.addtwo()

print(Son.a)
print(Son.b)
```

How child can access parents static and non-static properties

```python
class Father:
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Father.a + Father.b)


    def addthree(self):
        print(self.a + self.b+10)


class Son(Father):

    def addnew(self):
        self.addthree()
        Father.addtwo()


obj = Son()
obj.addnew()
```

⬛ Method Overriding

- Method overriding occurs when a subclass provides a new implementation for a method that is already defined in its superclass.
- This allows the subclass to modify or extend the behavior of that method.

Child can override parent method

```python
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son(Father):
    def addTwo(self, num1, num2):
        # Overriding the method
        print(f"Adding {num1} and {num2} in Son")
        return num1 + num2 + 1  # Modified behavior

# Creating instances of Father and Son
father = Father()
son = Son()

# Calling the addTwo method from Father and Son
```

```python
print(father.addTwo(5, 3))  # Output: 8
print(son.addTwo(5, 3))     # Output: Adding 5 and 3 in Son
                            #         9
```

⬛ Method overloading

- Method overloading is a concept where multiple methods have the same name but different parameters (number or type).
- Python doesn't support method overloading in the same way as some other languages like Java or C++.
- However, you can achieve similar behavior using default arguments or variable-length arguments.

Method Overloading Using Default Arguments

In this example, we will create a `Father` class with a method `addNumbers` that can take either two or three arguments.

```python
class Father:
    def addNumbers(self, num1, num2, num3=0):
        return num1 + num2 + num3

# Creating an instance of Father
father = Father()

# Calling the addNumbers method with two arguments
print(father.addNumbers(5, 3))  # Output: 8

# Calling the addNumbers method with three arguments
print(father.addNumbers(5, 3, 2))  # Output: 10
```

Method Overloading Using Variable-Length Arguments

In this example, we will use variable-length arguments to allow the method to accept any number of arguments.

```python
class Father:
    def addNumbers(self, *args):
        return sum(args)

# Creating an instance of Father
father = Father()

# Calling the addNumbers method with two arguments
print(father.addNumbers(5, 3))  # Output: 8
```

```python
# Calling the addNumbers method with three arguments
print(father.addNumbers(5, 3, 2))  # Output: 10

# Calling the addNumbers method with four arguments
print(father.addNumbers(1, 2, 3, 4))  # Output: 10
```

⬛ Abstract class in python

In Python, an **abstract class** is a class that cannot be instantiated directly and often serves as a blueprint for other classes. It is used when you want to define common behavior (methods) for a group of related classes, but you don't want to allow the creation of an object of the abstract class itself. Abstract classes are essential for designing interfaces in object-oriented programming.

Key Points of Abstract Classes:

1. **Purpose**: Define methods that must be implemented by subclasses.
2. **Cannot be instantiated**: You cannot create an object of an abstract class directly.
3. **Abstract methods**: Methods declared in the abstract class but without any implementation.
4. **Implementation in Subclasses**: Subclasses are required to provide implementations for the abstract methods.

To make the `Father` class abstract in Python, you need to:

1. Import the `ABC` (Abstract Base Class) module from the `abc` library.
2. Use the `ABC` class as a base class for `Father`.
3. Mark the method `addtwo` as abstract using the `@abstractmethod` decorator.

```python
from abc import ABC, abstractmethod

# Define the abstract class
class Father(ABC):
    x = 10
    y = 20

    # Define an abstract method
    @abstractmethod
    def addtwo(self):
        pass

# Subclass that implements the abstract method
class Son(Father):

    # Provide implementation for the abstract method
```

```python
    def addtwo(self):
        print(self.x + self.y)

# Instantiate Son class and call the method
obj = Son()
obj.addtwo()

# You cannot instantiate Father directly now:
# obj2 = Father()  # This will raise an error because Father is abstract
```

⬛ Access modifiers

In Python, **access modifiers** control the visibility and accessibility of class attributes and methods. While Python does not have strict access control like some other programming languages (e.g., Java or C++), it uses naming conventions to indicate access levels. The three main types of access modifiers in Python are:

Public (No leading underscore)

- Attributes and methods are **public** by default.
- They can be accessed and modified both inside and outside the class.
- There are no restrictions on public attributes.

```python
class Car:
    def __init__(self, brand):
        self.brand = brand  # Public attribute

    def display_brand(self):  # Public method
        return f"The brand is {self.brand}"

car = Car("Toyota")
print(car.brand)  # Accessing public attribute
print(car.display_brand())  # Accessing public method
```

Protected (Single leading underscore: `_`)

- Attributes and methods prefixed with a single underscore `_` are **protected**.
- This is a **convention** indicating that these members should not be accessed or modified outside the class (or its subclasses), but they are still accessible.
- This is a "soft" access control, meaning it is more of a warning to developers.

```python
class Car:
    def __init__(self, brand):
        self._brand = brand  # Protected attribute

    def _display_brand(self):  # Protected method
        return f"The brand is {self._brand}"

car = Car("Toyota")
print(car._brand)  # Technically accessible, but not recommended
print(car._display_brand())  # Accessible, but should be used with caution
```

Private (Double leading underscore: `__`)

- Attributes and methods prefixed with **double underscores (`__`)** are considered **private**.
- These members are not meant to be accessed or modified outside the class. Python applies **name mangling** to private members to make them harder to access from outside the class.
- Private attributes and methods are intended for internal use only within the class.

```python
class Car:
    def __init__(self, brand):
        self.__brand = brand  # Private attribute

    def __display_brand(self):  # Private method
        return f"The brand is {self.__brand}"

    def get_brand(self):  # Public method to access private attribute
        return self.__brand

car = Car("Toyota")
# print(car.__brand)  # Raises AttributeError, can't access directly
print(car.get_brand())  # Access through a public method
```

⬛ Getter Setter

Python has a way to define **getters** and **setters**, typically using the `property()` function or the `@property` decorator. Here's how they work:

- **Getters**: Used to access (get) the value of a private or protected attribute.
- **Setters**: Used to set (update) the value of a private or protected attribute.

```python
class CAR:
    __BRAND = "Toyota"
```

```python
    @property
    def BRAND(self):
        return self.__BRAND

    @BRAND.setter
    def BRAND(self,value):
        self.__BRAND=value


OBJ = CAR()
OBJ.BRAND="Suzuki"
print(OBJ.BRAND
```

⬛ Encapsulation

Encapsulation in Python refers to the concept of restricting access to certain details of an object and exposing only the necessary parts. Here are 3 key points:

1. **Data Protection**: Encapsulation hides internal state by using private attributes (`__attribute`), protecting the data from unauthorized modification.
2. **Controlled Access**: Provides controlled access to data through public methods (getters/setters), ensuring consistency and validation.
3. **Modularity**: Enhances modularity by keeping the internal workings of a class hidden, making the system easier to maintain and modify.

```python
class BankAccount:
    __balance = 0

    # Deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}")
        else:
            print("Invalid deposit amount!")

    # Withdraw money
    def withdraw(self, amount):
        if 0 < amount and amount<= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. New balance: {self.__balance}")
        else:
            print("Invalid or insufficient funds for withdrawal!")

    # Check balance
    def check_balance(self):
```

```python
        return self.__balance


# Create an account
account = BankAccount()

# Deposit money
account.deposit(500)

# Withdraw money
account.withdraw(200)

# Check Balance
print(account.check_balance())
```

⬛ Polymorphism

Polymorphism is one of the core concepts of object-oriented programming (OOP), along with inheritance, encapsulation, and abstraction. Polymorphism allows objects of different classes to be treated as objects of a common superclass. It provides a way to use a single interface to represent different underlying forms (data types).

Key Points of Polymorphism

1. **Definition**: Polymorphism means "many shapes" and allows methods to do different things based on the object it is acting upon.
2. **Types of Polymorphism**:
    - **Compile-time Polymorphism (Method Overloading)**: Achieved by function overloading. Not directly supported in Python, but can be simulated using default arguments or variable-length arguments.
    - **Runtime Polymorphism (Method Overriding)**: Achieved when a method in a subclass has the same name, return type, and parameters as in its superclass.

```python
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son(Father):
    def addTwo(self, num1, num2):
        # Overriding the method
        print(f"Adding {num1} and {num2} in Son")
        return num1 + num2 + 1  # Modified behavior

# Function to demonstrate polymorphism
def demonstrate_polymorphism(obj, num1, num2):
```

```python
    return obj.addTwo(num1, num2)

# Creating instances of Father and Son
father = Father()
son = Son()

# Demonstrating polymorphism
print(demonstrate_polymorphism(father, 5, 3))  # Output: 8
print(demonstrate_polymorphism(son, 5, 3))     # Output: Adding 5 and 3 in Son
                                               #         9
```