Object-Oriented Programming (OOP) Concepts in Python

OOPS Concept - Python

Definition:

Object-Oriented Programming (OOP) is a programming paradigm that organizes code by bundling related data and functions together into objects, which represent real-world entities. The key principles of OOP are encapsulation, inheritance, polymorphism, and abstraction, commonly referred to as the four pillars of OOP.

Principles of OOP:

1. Abstraction

2. Encapsulation

3. Inheritance

4. Polymorphism

Topics:

Class:

A class is a blueprint or template for creating object. It defines a set of attributes and methods that created objects(instances) will have. In Python, you define a class using the class keywords

Example:

```
class Car:

    # class attribute

    wheels = 4

    # method

    def drive(self):

        print("The car is driving.")
```

Object:

A object is an instance of a class when a class is created it can be used for instances(create) multiple objects, each with its own unique data but based on the structure defined by class

Example:

```
# Creating an object of the Car class

my_car = Car()
```

__init__:

the init method is the special method in python known as constructor. It initializes an objects attributes when an object is created. This method is automatically called when an object is instantiated. You define it with the __init__ keyword, and it can take arguments to set up an object's initial state.

Example:

```
class Car:

    def __init__(self, color, model):

        self.color = color  # instance attribute

        self.model = model  # instance attribute


    def drive(self):

        print(f"The {self.color} {self.model} is driving.")
```

Usage :

```
my_car = Car("red", "Toyota")

print(my_car.color)  # Output: red

print(my_car.model)  # Output: Toyota

my_car.drive()  # Output: The red Toyota is driving.
```

Self Keyword in Python:

The self keyword represents the instance of the class and is used to access instance attributes and methods within the class. It must be the first parameter in instance methods (like __init__ and other methods) but does not need to be passed when calling the method

Example:

```
class Car:

    def __init__(self, color, model):

        self.color = color

        self.model = model


    def drive(self):

        print(f"The {self.color} {self.model} is driving.")
```

Methods:

A method is a function that is defined inside a class and is associated with the objects (or instances) of that class. Methods are used to define the behaviors or actions that an object of a class can perform. They allow objects to interact with or manipulate their own data, which is encapsulated within the object as attributes.

In other words, a method is similar to a function, but it is specifically designed to operate on instances of a class, with the ability to access and modify the instance's attributes and other methods.

Key Points About Methods:

1. Defined within a class: Methods are written inside a class block and are intended to work with instances of that class.

2. Called on an object: Methods are invoked using the syntax object.method(), where object is an instance of the class.

3. Self-parameter: Most methods in Python use self as their first parameter, which represents the instance of the class calling the method. This allows the method to access instance-specific data and other methods.

4. Encapsulation: Methods help encapsulate functionality that is specific to the class and work with the data related to instances of that class.

Types of Methods:

Instance Method:

An instance method is a method that operates on an instance of a class. It is the most common type of method, and it uses the self parameter to access instance-specific data, meaning it can read and modify instance attributes. Instance methods are tied to a particular instance of a class.

Defining: Defined with def and the self parameter.

Access: Can access instance attributes and other instance methods within the class.

Use Case: When you need to work with data specific to an instance.

Example :

```
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self):
        print(f"{self.name} is barking.")

dog = Dog("Buddy")
dog.bark()  # Output: Buddy is barking.
```

Class Method:

A class method is a method that operates on the class itself rather than on an instance. Class methods use the cls parameter to refer to the class, allowing them to modify or access class-

level data (attributes that are shared across all instances of the class). They are marked with the @classmethod decorator.

Defining: Defined with @classmethod and the cls parameter.

Access: Can access class attributes and other class methods but cannot modify instance-specific data.

Use Case: When you need to work with data that applies to all instances of the class or want to create alternative constructors.

```
class Dog:
    species = "Canis lupus familiaris"  # class attribute

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

    @classmethod
    def get_species(cls):
        return cls.species
print(Dog.get_species())  # Output: Canis lupus familiaris
```

Static Method:

A static method is a method that does not operate on an instance or class level. It behaves like a regular function but is included in the class for logical grouping. Static methods do not receive self or cls parameters and cannot modify class or instance state.

Defining: Defined with @staticmethod and no self or cls parameter.

Access: Does not access or modify instance or class attributes.

Use Case: When a method performs a task that is related to the class but doesn't require access to the class or its instances.

```
class Dog:
    @staticmethod
    def sound():
        return "Bark"
print(Dog.sound())  # Output: Bark
```

Super:

super() is a built-in function used in inheritance to refer to the parent (or superclass) of a derived (or child) class. It allows you to call methods and access properties from the parent class within the child class, which is especially useful for extending or overriding methods in a subclass.

Key Uses of super():

1. Access Parent Class Methods: You can use super() to call a method in the parent class, enabling the child class to add its own behavior before or after the parent's method executes.

2. Initialize Parent Class: In the child class, you can use super().__init__() to ensure the parent's __init__ method is called, which is essential for initializing inherited attributes.

Example:

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        print(f"{self.make} {self.model} is starting.")

class Car(Vehicle):
    def __init__(self, make, model, color):
```

```
        # Use super() to call the parent class's __init__ method
        super().__init__(make, model)
        self.color = color

    def start(self):
        # Call the parent class's start method
        super().start()
        print(f"The {self.color} car is now in motion.")
```

usage :

```
my_car = Car("Toyota", "Camry", "blue")
my_car.start()
# Output:
# Toyota Camry is starting.
# The blue car is now in motion.
```

Benefits of Using super()

1. Avoids Code Duplication: It lets you reuse code from the parent class without rewriting it.

2. Ensures Proper Initialization: Ensures that the parent class's __init__ method runs, which is crucial for setting up inherited attributes.

3. Supports Multiple Inheritance: super() can be particularly useful in multiple inheritance scenarios, as it works with Python's method resolution order (MRO) to ensure methods are called in the correct order.

Setter and Getter:

In Python, setters and getters can be implemented using the @property decorator, which makes it easy to create managed attributes (attributes with controlled access). Here's how to add a setter and getter for the read_only attribute in your Phone class.

Example :

```
class Phone(Item):
    def __init__(self, name: str, price: float, quantity=0, broken_phones=0):
        # Call to super function to have access to all attributes / methods
        super().__init__(name, price, quantity)

        # Run validations to the received arguments
        assert broken_phones >= 0, f"Broken Phones {broken_phones} is not greater or equal to zero!"

        # Assign to self object
        self.broken_phones = broken_phones
        self._read_only = "AAA"  # private attribute for the read-only property

    @property
    def read_only(self):
        return self._read_only  # Getter for the read_only attribute

    @read_only.setter
    def read_only(self, value):
        # Optionally, add validation here if needed
        self._read_only = value  # Setter for the read_only attribute
```

Explanation:

Getter (@property):

@property makes the read_only method behave like an attribute. So phone1.read_only calls this getter method.

Setter (@read_only.setter):

The @read_only.setter decorator lets you define a method to set the read_only property. This allows phone1.read_only = "BBB" to set the _read_only attribute.

Usage:

```
phone1 = Phone("jscPhonev10", 500, 5, 1)

print(phone1.read_only)  # Output: AAA


phone1.read_only = "BBB"  # Sets the value of read_only

print(phone1.read_only)  # Output: BBB
```

Method Attribute Modes:

Public Attributes and Methods:

No underscore (my_attribute): Public attributes or methods can be accessed from anywhere, both inside and outside the class.

Example

```
class Item:

    def __init__(self, name, price):

        self.name = name  # Public attribute

        self.price = price  # Public attribute


    def display_info(self):  # Public method

        print(f"Name: {self.name}, Price: {self.price}")
```

Protected Attributes and Methods:

Single underscore (_my_attribute): By convention, attributes or methods prefixed with a single underscore are treated as protected. This signals that these members are intended for internal use within the class or its subclasses, though they can technically be accessed from outside the class.

Example:

```
class Item:

    def __init__(self, name, price):

        self._name = name  # Protected attribute

        self._price = price  # Protected attribute


    def _apply_discount(self):  # Protected method

        self._price *= 0.8  # Apply a 20% discount
```

Private Attributes and Methods:

Double underscore (__my_attribute): Attributes or methods prefixed with a double underscore are treated as private. Python performs name mangling to make these members harder to access directly from outside the class. This helps prevent accidental access or modification but can still be accessed using specific syntax (_ClassName__attribute).

Example :

```
class Item:

    def __init__(self, name, price):

        self.__name = name  # Private attribute

        self.__price = price  # Private attribute


    def __apply_secret_discount(self):  # Private method

        self.__price *= 0.7  # Apply a 30% secret discount
```

Demontrating the Modes:

```
item = Item("Phone", 500)

# Public access

print(item.name)  # Direct access

# Protected access (not recommended outside the class or subclasses)

print(item._name)  # Accessible but not recommended

# Private access

try:

print(item.__price)  # This will raise an AttributeError

except AttributeError as e:

print(e)

# Accessing a private attribute (name mangling)

print(item._Item__price)  # This works, but it's a workaround and not recommended
```

Encapsulation:

Encapsulation is a fundamental concept in object-oriented programming (OOP) that restricts direct access to certain components of an object. It involves bundling data (attributes) and methods (functions) that operate on that data within a single unit, or class, and controlling access to that data through public methods. This approach protects an object's internal state and hides its complexity from external code.

Real-Life Use Case of Encapsulation:

Imagine a bank account. When people deposit or withdraw money, they interact with an ATM or bank application interface, but they don't need to know the internal details of how transactions are processed. Encapsulation is used to control access to sensitive information like the account balance and to ensure that only valid operations (such as deposit or withdrawal) are performed.

Key Benefits of Encapsulation:

Control Access: Only specific parts of the program can modify or access the data.

Hide Complexity: Internal workings are hidden, simplifying the interface.

Reduce Errors: Restricts invalid data access or modification, ensuring data consistency.

Example:

```
class BankAccount:
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder  # Public attribute
        self.__balance = balance  # Private attribute

    # Getter method for balance
    def get_balance(self):
        return self.__balance

    # Setter method for deposit
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance is {self.__balance}.")
        else:
            print("Deposit amount must be positive.")

    # Setter method for withdraw
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. New balance is {self.__balance}.")
        else:
            print("Invalid withdrawal amount or insufficient funds.")
```

Usage:

```
# Creating an instance of BankAccount

account = BankAccount("John Doe", 100)

# Accessing balance through a public method

print(account.get_balance())  # Output: 100

# Depositing money

account.deposit(50)  # Output: Deposited 50. New balance is 150.

# Withdrawing money

account.withdraw(20)  # Output: Withdrew 20. New balance is 130.

# Trying to access the private attribute directly (not allowed)

try:

print(account.__balance)  # Raises AttributeError

except AttributeError as e:

print(e)
```

Real-Life Analogy

In a real bank account, customers do not directly access the internal systems managing their money. Instead, they use specific actions (depositing, withdrawing, checking balance) through a secure interface. By controlling these operations, the bank ensures that:

- Account balances remain accurate.
- Unauthorized access to the account balance is prevented.
- Invalid transactions (like overdrafts) are blocked.

Encapsulation in programming offers similar protections by managing access to sensitive data and providing an interface that controls how data is modified.

Abstraction:

Abstraction is an OOP concept that focuses on hiding complex implementation details from the user and only exposing the necessary functionality. By simplifying the interface, abstraction allows users to interact with objects at a high level without needing to know the intricate workings behind the scenes.

Real-Life Use Case of Abstraction:

Consider a car. To drive a car, you only need to know how to use the steering wheel, pedals, and other controls. You don't need to understand the details of how the engine, fuel system, and transmission work. Abstraction hides these complex components, exposing only the essential features to drive the car.

Key Benefits of Abstraction:

Simplifies Interaction: Provides a simplified view of complex systems.

Improves Code Reusability: Allows developers to use and reuse high-level interfaces without worrying about lower-level details.

Enhances Security: Hides the internal details, which prevents external code from interacting directly with complex systems.

Example Code for Abstraction:

Let's create an abstract Payment class that represents different ways to pay for items. Specific payment methods like CreditCardPayment and PayPalPayment can extend this abstract class.

In Python, abstraction is often implemented using abstract base classes, which can be created with the abc module.

```
from abc import ABC, abstractmethod


class Payment(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass  # Abstract method to be implemented by subclasses


class CreditCardPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")


class PayPalPayment(Payment):
```

```
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")


# Client code
def make_payment(payment_method, amount):
    payment_method.process_payment(amount)
# Using abstraction to handle different payment methods
credit_card = CreditCardPayment()
paypal = PayPalPayment()
make_payment(credit_card, 100)  # Output: Processing credit card payment of $100
make_payment(paypal, 150)       # Output: Processing PayPal payment of $150
```

Explanation of Abstraction in the Code

Abstract Base Class (Payment): Payment is an abstract class that defines a high-level interface for payment processing. It declares an abstract method process_payment, which subclasses must implement. Users only need to know they can call process_payment on any Payment instance without needing to know the specifics of each payment method.

Concrete Subclasses (CreditCardPayment and PayPalPayment): These classes implement the process_payment method in different ways for specific payment methods. The user does not need to know how each payment method works internally; they only need to call process_payment.

Client Code (make_payment function): The make_payment function abstracts away the type of payment method. It simply calls process_payment, knowing that the actual logic for processing will be handled by the specific subclass.

Real-Life Analogy

In a real shopping experience, people pay for items through various methods like credit cards, cash, or PayPal. They don't need to understand the underlying mechanisms of each payment type (like how a credit card is authorized or how PayPal handles transactions). They only interact with the interface (e.g., swiping a card, clicking a "Pay with PayPal" button), and the complex transaction details are abstracted away.

Differences Between Encapsulation and Abstraction

Aspect	Encapsulation	Abstraction
Purpose	To protect and restrict access to the internal data of a class.	To hide the complexity by showing only essential features.
Focus	Focuses on **how** data is accessed and modified.	Focuses on **what** functionality is provided to the user.
Implementation	Uses private and protected attributes and methods.	Uses abstract classes, interfaces, and high-level methods.
Accessibility	Limits access to the internal data (data hiding).	Exposes essential functionality, hiding internal implementation.
Example	Using private attributes like __balance in BankAccount.	Defining process_payment as an abstract method in Payment.
Role in OOP	Ensures data integrity by controlling access.	Simplifies interaction by hiding complexity.
Real-Life Analogy	Bank account balance is private and controlled by deposit/withdraw methods.	A car provides a simplified interface (steering, pedals) to drive, without exposing engine details.

Inheritance:

Inheritance is a core concept in object-oriented programming (OOP) that allows a class (known as a child or subclass) to inherit attributes and methods from another class (known as a parent or superclass). This enables code reuse and allows us to build complex systems in a structured, modular way by defining relationships between classes.

Real-Life Example of Inheritance:

Consider a university system where we have different types of members, such as students, professors, and administrative staff. All of them share common properties like name, ID, and contact information, and common actions like updating contact details. However, each type also has unique properties and behaviors, such as courses for students, research areas for professors, and departments for administrative staff. Using inheritance, we can define a general UniversityMember class and then have Student, Professor, and AdminStaff classes inherit from it, adding specific details for each member type.

Key Benefits of Inheritance:

- Code Reusability: Allows classes to reuse methods and attributes from parent classes.
- Logical Structure: Creates a hierarchical relationship, improving organization and clarity.
- Extendability: Makes it easy to add new functionality by extending existing classes.

Types of Inheritance in Python:

Python supports several types of inheritance, each serving different needs. Below, we'll explore each type with examples.

1. Single Inheritance:

In single inheritance, a subclass inherits from only one parent class. This is the simplest form of inheritance, establishing a one-to-one relationship between the parent and child class.

Example:

```
# Parent class

class Vehicle:

    def start(self):

        print("Starting the vehicle.")


# Child class

class Car(Vehicle):

    def drive(self):

        print("The car is driving.")


# Usage

my_car = Car()

my_car.start()  # Output: Starting the vehicle.

my_car.drive()  # Output: The car is driving.
```

Explanation:

In this example, Car inherits from Vehicle, gaining access to its start method. Car can also define its own methods, like drive.

2. Multiple Inheritance:

In multiple inheritance, a subclass inherits from more than one parent class. This is useful when a class needs to combine functionality from multiple sources. Python supports multiple inheritance but requires careful handling to avoid complexity and ambiguity.

Example :

```
# Parent classes

class Engine:

    def engine_info(self):

        print("Engine type: V8")


class Body:

    def body_info(self):

        print("Body type: SUV")


# Child class

class Car(Engine, Body):

    def car_info(self):

        print("Car information.")

# Usage

my_car = Car()

my_car.engine_info()  # Output: Engine type: V8

my_car.body_info()    # Output: Body type: SUV

my_car.car_info()     # Output: Car information.
```

Explanation:

Note: Python uses the Method Resolution Order (MRO) to decide the order in which classes are searched for attributes and methods, which helps avoid ambiguity in multiple inheritance.

3. Multilevel Inheritance

In multilevel inheritance, a class inherits from another class, which in turn inherits from another class, forming a chain of inheritance. This creates a hierarchical relationship among classes.

Example :

```
# Base class
class Animal:
    def breathe(self):
        print("Animal is breathing.")

# Intermediate class
class Mammal(Animal):
    def feed_milk(self):
        print("Mammal is feeding milk.")

# Derived class
class Dog(Mammal):
    def bark(self):
        print("Dog is barking.")

# Usage
dog = Dog()
dog.breathe()     # Output: Animal is breathing.
dog.feed_milk()   # Output: Mammal is feeding milk.
dog.bark()        # Output: Dog is barking.
```

Explanation: Here, Dog inherits from Mammal, which in turn inherits from Animal. As a result, Dog has access to the breathe, feed_milk, and bark methods.

4. Hierarchical Inheritance:

In hierarchical inheritance, multiple classes inherit from the same parent class. This allows different subclasses to share the functionality of a single parent class.

Example :

```
# Parent class
class Animal:
    def eat(self):
        print("Animal is eating.")

# Child classes
class Dog(Animal):
    def bark(self):
        print("Dog is barking.")

class Cat(Animal):
    def meow(self):
        print("Cat is meowing.")

# Usage
dog = Dog()
dog.eat()    # Output: Animal is eating.
dog.bark()   # Output: Dog is barking.

cat = Cat()
cat.eat()    # Output: Animal is eating.
cat.meow()   # Output: Cat is meowing.
```

Explanation: Both Dog and Cat inherit from Animal, allowing them to use the eat method while also defining their own specific methods.

5. Hybrid Inheritance:

Hybrid inheritance combines two or more types of inheritance, typically creating a more complex hierarchy. Python supports hybrid inheritance, but it's crucial to use it carefully to avoid complexity and confusion.

Example :

```
# Base class
class Animal:
    def breathe(self):
        print("Animal is breathing.")

# Derived classes
class Mammal(Animal):
    def feed_milk(self):
        print("Mammal is feeding milk.")

class Bird(Animal):
    def lay_eggs(self):
        print("Bird is laying eggs.")

# Further derived class
class Platypus(Mammal, Bird):
    def unique_behavior(self):
        print("Platypus has unique behaviors.")

# Usage
platypus = Platypus()
platypus.breathe()        # Output: Animal is breathing.
```

platypus.feed_milk() # Output: Mammal is feeding milk.

platypus.lay_eggs() # Output: Bird is laying eggs.

platypus.unique_behavior() # Output: Platypus has unique behaviors.

Explanation: Platypus inherits from both Mammal and Bird, creating a hybrid inheritance structure. Platypus has access to methods from Mammal, Bird, and the base class Animal.

Summary of Inheritance Types

Type	Description	Example
Single Inheritance	Inherits from one parent class.	Car(Vehicle)
Multiple Inheritance	Inherits from multiple parent classes.	Car(Engine, Body)
Multilevel Inheritance	Chain of inheritance across multiple levels.	Dog(Mammal(Animal))
Hierarchical Inheritance	Multiple subclasses inherit from the same parent class.	Dog(Animal), Cat(Animal)
Hybrid Inheritance	Combines two or more types of inheritance, creating a mixed structure.	Platypus(Mammal, Bird)

Polymorphism:

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables the same method or operation to behave differently based on the object it's acting upon. The term "polymorphism" means "many forms" in Greek, and it enables flexibility and extensibility in programming by allowing objects to implement shared behaviors in unique ways.

Types of Polymorphism in Python:

- Method Overriding (Runtime Polymorphism): Achieved by defining a method in the child class with the same name as a method in the parent class. Python decides which method to call at runtime based on the object type.

- Method Overloading: While Python doesn't support method overloading (i.e., multiple methods with the same name but different arguments) directly, similar functionality can be achieved using default arguments or by checking the types of inputs.
- Operator Overloading: Enables custom behavior for operators (like +, -, etc.) when used with instances of a class by overriding special methods.

Real-Life Example of Polymorphism:

Imagine a remote control that can operate different devices like a TV, sound system, and air conditioner. Although each device responds to the "power" button, they behave differently when it's pressed. The remote control provides a single interface (the power button), while each device implements its own response to that button.

Key Benefits of Polymorphism:

- Flexibility: Allows functions to use objects of different types, promoting code flexibility.
- Simplified Code: Provides a unified interface, which reduces the need for complex conditional checks.
- Extensibility: Enables easy expansion of functionality by adding new classes that implement the same interface.

Example of Polymorphism in Python:

Let's look at a practical example to illustrate polymorphism. Here, we have a Shape class with subclasses Circle and Square. Each subclass implements its own version of the area method.

```
# Base class

class Shape:

    def area(self):

        pass  # Abstract method, not implemented here



# Subclass for Circle

class Circle(Shape):

    def __init__(self, radius):

        self.radius = radius


    def area(self):

        return 3.14 * (self.radius ** 2)
```

```
# Subclass for Square
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

# Function that uses polymorphism
def print_area(shape):
    print(f"The area is: {shape.area()}")

# Usage
circle = Circle(5)
square = Square(4)

print_area(circle)  # Output: The area is: 78.5
print_area(square)  # Output: The area is: 16
```

Explanation

Polymorphic Behavior:

- The Shape class defines the area method, but without an implementation.
- Both Circle and Square classes provide their own implementation of the area method.

Unified Interface:

- The print_area function calls the area method without needing to know whether the shape is a Circle or a Square. This is possible because both subclasses implement area, making them polymorphic.

Runtime Method Resolution:

- At runtime, Python determines the correct area method to call based on the object type passed to print_area. This is runtime polymorphism or method overriding.

Operator Overloading (Another Form of Polymorphism)

Operator overloading allows us to define custom behavior for operators when applied to instances of a class. For example, we can override the + operator to add two objects in a specific way.

Example :

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"
# Usage
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2

print(v3)  # Output: Vector(6, 8)
```

Explanation:

By overriding the __add__ method, we allow the + operator to combine two Vector instances. This is a form of polymorphism where the + operator behaves differently depending on the type of objects it's adding.

Summary of Polymorphism Types

Type	Description	Example
Method Overriding	Subclasses implement a method with the same name as a method in the parent class.	area method in Circle and Square classes.
Method Overloading	Not directly supported, but can be emulated with default arguments.	Function with default arguments.

Type	Description	Example
Operator Overloading	Custom behavior for operators on objects.	Overloading + with __add__ in Vector class.

Key Points:

- Polymorphism allows for writing generic code that can work with objects of various types, as long as they implement the required methods.
- By using polymorphism, developers can introduce new types easily, as long as they follow the expected interface, making code more extensible and maintainable.
- In short, polymorphism enables flexibility and reusability in code by providing a consistent interface that can be implemented differently depending on the object's type.