Object-Oriented Programming (OOP) :

Object-Oriented Programming (OOP) with Python, an attribute is a characteristic or property of an object. Objects in Python are instances of classes, and classes define attributes to represent the data associated with those objects. Attributes can be thought of as variables that belong to a specific object.In object-oriented programming (OOP), an object is an instance of a class. A class is a blueprint or a template that defines the structure and behavior of objects. Objects are created from classes and can have attributes (data) and methods (functions) associated with them.

Attributes can be of various types, including:

Instance Attributes:

These attributes belong to a specific instance of a class. They are defined within the constructor method (__init__) and are unique to each object created from the class.

```
class Car:
    def __init__(self, make, model):
        self.make = make  # Instance attribute
        self.model = model  # Instance attribute

my_car = Car("Toyota", "Camry")
print(my_car.make)  # Accessing the instance attribute
```

Class Attributes:

These attributes are shared by all instances of a class. They are defined outside the constructor and are the same for every object created from the class.

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, make, model):
        self.make = make
        self.model = model

my_car = Car("Toyota", "Camry")
print(my_car.wheels)  # Accessing the class attribute
```

Property:

A special kind of attribute that is accessed like an attribute but is implemented using getter and setter methods. It allows you to perform actions when getting or setting the attribute.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Private attribute

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

my_circle = Circle(5)
print(my_circle.radius)  # Accessing the property
my_circle.radius = 7  # Setting the property
```

Attributes play a crucial role in encapsulating data within objects and defining the state of an object. They contribute to the overall structure and behavior of classes in Python's Object-Oriented Programming paradigm.

Here's a simple explanation and examples in Python:

Define a Class:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says Woof!")
```

In this example, we've defined a Dog class with a constructor (__init__ method) that initializes the name and age attributes. It also has a bark method.

Create Objects:

Now, we can create objects (instances) of the Dog class:

```
# Create instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)

# Access attributes
print(f"{dog1.name} is {dog1.age} years old.")
print(f"{dog2.name} is {dog2.age} years old.")

# Call methods
dog1.bark()
dog2.bark()
```

In this example, dog1 and dog2 are objects of the Dog class. We've set their attributes using the constructor (__init__) and called the bark method on each object.

Objects allow you to model real-world entities and their behaviors in your code. The class provides a blueprint, and each object is a specific instance of that blueprint with its own unique data.

Constructor :

In Python, a constructor is a special method called __init__ that is automatically called when an object is created. It is used to initialize the attributes of an object. Here's a simple explanation with examples:
Basic Constructor:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create an instance of the Person class
person1 = Person("John", 25)

# Access attributes
print(f"{person1.name} is {person1.age} years old.")
```

In this example, the __init__ method is used to initialize the name and age attributes of the Person class. When we create an instance of the class (person1), we pass values for name and age, and the constructor sets these values as attributes.

Default Values in Constructor:

```
class Book:
    def __init__(self, title, author, year_published=2020):
        self.title = title
        self.author = author
        self.year_published = year_published

# Create an instance of the Book class
book1 = Book("The Python Book", "John Smith")

# Access attributes
print(f"{book1.title} by {book1.author}, published in {book1.year_published}.")
```

In this example, the year_published parameter in the __init__ method has a default value of 2020. If no value is provided for year_published when creating an instance of the Book class, it defaults to 2020.

Constructors with Default Values and Additional Parameters:

```
class Rectangle:
    def __init__(self, length=1, width=1):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# Create instances of the Rectangle class
rectangle1 = Rectangle()
rectangle2 = Rectangle(4, 5)

# Access attributes and call methods
print(f"Rectangle 1 area: {rectangle1.area()}")
print(f"Rectangle 2 area: {rectangle2.area()}")
```

Here, the Rectangle class has a constructor with default values for length and width. When creating instances (rectangle1 and rectangle2), you can either use the default values or provide specific values.

Constructors in Python OOP are essential for setting up the initial state of objects, allowing you to define how objects should be instantiated with specific attributes.

Destructor :

In Python, a destructor is a special method called __del__ that is automatically invoked when an object is about to be destroyed or garbage collected. The primary purpose of a destructor is to perform cleanup operations before an object is removed from memory. Here's an example:

```
class MyClass:
    def __init__(self, name):
        self.name = name
        print(f"Object {self.name} created.")

    def __del__(self):
        print(f"Object {self.name} is being destroyed.")

# Create instances of the MyClass class
obj1 = MyClass("Instance1")
obj2 = MyClass("Instance2")

# Explicitly delete an object
del obj1

# The remaining object will be automatically deleted when the program ends
```

In this example:

The __init__ method is the constructor, and it is called when instances of the MyClass class are created.
The __del__ method is the destructor, and it is automatically called when an object is about to be destroyed.
When the program is run, you'll see output like:

```
Object Instance1 created.
Object Instance2 created.
Object Instance1 is being destroyed.
```

Here's a breakdown of what happens:

Two instances (obj1 and obj2) are created, invoking the __init__ method for each.
The del statement explicitly deletes obj1, causing its destructor (__del__) to be called.
When the program ends, the remaining instance (obj2) is automatically deleted, and its destructor is called.
It's important to note that relying on the __del__ method for cleanup is not always recommended, as the exact timing of when the destructor is called is not guaranteed. In practice, it's often better to use other mechanisms, such as context managers (with statement) or specific methods for cleanup tasks. The __del__ method is provided more for informational purposes and is not relied upon for critical cleanup in many scenarios.

```
class RecursiveFunction:
        def __init__(self, n):
                self.n = n
                print("Recursive function initialized with n =", n)

        def run(self, n=None):
                if n is None:
                        n = self.n
                if n <= 0:
                        return
                print("Running recursive function with n =", n)
                self.run(n-1)

        def __del__(self):
                print("Recursive function object destroyed")

# Create an object of the class
obj = RecursiveFunction(5)

# Call the recursive function
obj.run()

# Destroy the object
del obj
```

INHERITANCES:

One of the core concepts in object-oriented programming (OOP) languages is inheritance. It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are:

Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:

It represents real-world relationships well.
It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
Inheritance offers a simple, understandable model structure.
Less development and maintenance expenses result from an inheritance.

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit the characteristics and behaviors of an existing class (superclass or base class). This promotes code reuse and the creation of a hierarchy of classes. Let's explore inheritance with a clear example in Python:
Base class (superclass)

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

# Derived class 1
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Derived class 2
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Derived class 3
class Bird(Animal):
    def make_sound(self):
        return "Tweet!"

# Create instances of the derived classes
dog_instance = Dog("Buddy")
cat_instance = Cat("Whiskers")
bird_instance = Bird("Tweetie")

# Access attributes and call methods
print(f"{dog_instance.name} says {dog_instance.make_sound()}")
print(f"{cat_instance.name} says {cat_instance.make_sound()}")
print(f"{bird_instance.name} says {bird_instance.make_sound()}")
```

In this example:

The Animal class is the base class, and it has an __init__ method for initializing the name attribute and a make_sound method (with a placeholder pass).

The Dog, Cat, and Bird classes are derived classes that inherit from the Animal class. They override the make_sound method to provide their own implementation.

Instances of Dog, Cat, and Bird are created: dog_instance, cat_instance, and bird_instance.

The make_sound method is called on each instance, and the appropriate sound is printed based on the overridden method in the derived classes.

Inheritance allows the derived classes to reuse the attributes and methods of the base class while extending or customizing their behavior. It promotes code organization and makes it easier to manage and modify classes in the future. In this example, all animals share common characteristics from the Animal base class, and each specific type of animal (dog, cat, bird) adds its own unique behavior.

```
Example 1: Vehicle Hierarchy
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        return f"The {self.brand} {self.model}'s engine is starting."

    def stop_engine(self):
        return f"The {self.brand} {self.model}'s engine is stopping."

class Car(Vehicle):
    def drive(self):
        return f"The {self.brand} {self.model} is now driving."

class Motorcycle(Vehicle):
    def wheelie(self):
        return f"The {self.brand} {self.model} is doing a wheelie."

# Create instances of the derived classes
car_instance = Car("Toyota", "Camry")
motorcycle_instance = Motorcycle("Harley-Davidson", "Sportster")

# Access attributes and call methods
print(car_instance.start_engine())
print(car_instance.drive())

print(motorcycle_instance.start_engine())
```

```
print(motorcycle_instance.wheelie())

Example 2: Shape Hierarchy
import math

class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2

# Create instances of the derived classes
circle_instance = Circle(5)
square_instance = Square(4)

# Access attributes and call methods
print(f"Circle area: {circle_instance.area()}")
print(f"Square area: {square_instance.area()}")

Example 3: Employee Hierarchy
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def get_info(self):
        return f"{self.name} earns ${self.salary} per year."

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department
```

```
    def get_info(self):
        return f"{self.name} is a manager in the {self.department} department."

class Programmer(Employee):
    def __init__(self, name, salary, programming_language):
        super().__init__(name, salary)
        self.programming_language = programming_language

    def get_info(self):
        return f"{self.name} is a programmer specializing in {self.programming_language}."

# Create instances of the derived classes
manager_instance = Manager("Alice", 80000, "IT")
programmer_instance = Programmer("Bob", 60000, "Python")

# Access attributes and call methods
print(manager_instance.get_info())
print(programmer_instance.get_info())
```

Inheritance in object-oriented programming (OOP) comes in various types, each serving different purposes. Here are some common types of inheritance and examples in Python:

1. Single Inheritance:
In single inheritance, a class inherits from only one base class.

Example:

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

class Dog(Animal):
    def bark(self):
        return "Dog barks."

# Create an instance of the derived class
dog_instance = Dog()

# Access methods
print(dog_instance.speak())  # Inherited from Animal class
print(dog_instance.bark())   # Specific to Dog class
```

2. Multiple Inheritance:

Multiple inheritance occurs when a class inherits from more than one base class.

Example:

```
class Bird:
    def chirp(self):
        return "Bird chirps."

class FlyingObject:
    def fly(self):
        return "Object is flying."

class FlyingBird(Bird, FlyingObject):
    pass

# Create an instance of the derived class
flying_bird_instance = FlyingBird()

# Access methods
print(flying_bird_instance.chirp())  # Inherited from Bird class
print(flying_bird_instance.fly())    # Inherited from FlyingObject class
```

3. Multilevel Inheritance:
In multilevel inheritance, a class inherits from another class, and another class inherits from the second class, forming a chain.

Example:

```
class Vehicle:
    def start_engine(self):
        return "Vehicle engine started."

class Car(Vehicle):
    def drive(self):
        return "Car is now driving."

class SportsCar(Car):
    def race(self):
        return "Sports car is racing."

# Create an instance of the derived class
sports_car_instance = SportsCar()

# Access methods
print(sports_car_instance.start_engine())  # Inherited from Vehicle class
print(sports_car_instance.drive())         # Inherited from Car class
```

```
print(sports_car_instance.race())         # Specific to SportsCar class
```

4. Hierarchical Inheritance:
In hierarchical inheritance, multiple classes inherit from a single base class.

Example:
```
class Shape:
    def area(self):
        return "Area calculation for a generic shape."

class Circle(Shape):
    def calculate_area(self, radius):
        return 3.14 * radius ** 2

class Square(Shape):
    def calculate_area(self, side_length):
        return side_length ** 2

# Create instances of the derived classes
circle_instance = Circle()
square_instance = Square()

# Access methods
print(circle_instance.area())                 # Inherited from Shape class
print(circle_instance.calculate_area(5))      # Specific to Circle class
print(square_instance.area())                 # Inherited from Shape class
print(square_instance.calculate_area(4))      # Specific to Square class
```

Hybrid inheritance refers to a combination of two or more types of inheritance within a single program. It often involves a mix of single inheritance, multiple inheritance, multilevel inheritance, or hierarchical inheritance. Hybrid inheritance can be a powerful way to model complex relationships in object-oriented programming. However, it's important to carefully design and manage the class hierarchy to avoid ambiguity and potential issues.

Here's an example of hybrid inheritance in Python:
```
class Animal:
    def speak(self):
        return "Animal speaks."

class Mammal(Animal):
    def give_birth(self):
        return "Mammal gives birth to live young."
```

```
class Bird(Animal):
    def lay_eggs(self):
        return "Bird lays eggs."

class Bat(Mammal, Bird):
    def fly(self):
        return "Bat is flying."

# Create an instance of the derived class
bat_instance = Bat()

# Access methods
print(bat_instance.speak())        # Inherited from Animal class
print(bat_instance.give_birth())   # Inherited from Mammal class
print(bat_instance.lay_eggs())     # Inherited from Bird class
print(bat_instance.fly())          # Specific to Bat class
```

In this example:

The Animal class is a base class with a generic method speak.
The Mammal class inherits from Animal and has a specific method give_birth.
The Bird class also inherits from Animal and has a specific method lay_eggs.
The Bat class inherits from both Mammal and Bird, creating a hybrid inheritance. It has its own specific method fly.
While hybrid inheritance allows for flexibility in modeling complex relationships, it's important to be cautious, as it can lead to code complexity and potential conflicts. Proper design and careful consideration of class relationships are crucial to ensure that the code remains clear and maintainable. In practice, hybrid inheritance is not always recommended unless it is necessary for modeling the specific relationships in your application.

ENCAPSULATION:

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP), along with inheritance, polymorphism, and abstraction. It refers to the bundling of data

(attributes) and methods (functions) that operate on the data into a single unit, known as a class. Encapsulation helps in hiding the internal details of the object and exposing only what is necessary, promoting information hiding and a clear separation of concerns. In Python, encapsulation is achieved through the use of private and public access modifiers.

Example of Encapsulation in Python:

```
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute
        self.__model = model  # Private attribute
        self.__fuel_level = 100  # Private attribute

    def start_engine(self):
        print(f"{self.__make} {self.__model}'s engine started.")
        self.__fuel_level -= 10

    def drive(self, distance):
        print(f"{self.__make} {self.__model} is driving for {distance} miles.")
        self.__fuel_level -= distance

    def get_fuel_level(self):
        return self.__fuel_level  # Getter method

    def refuel(self, amount):
        print(f"Refueling {self.__make} {self.__model} with {amount} gallons.")
        self.__fuel_level += amount

# Create an instance of the Car class
my_car = Car("Toyota", "Camry")

# Access public methods
my_car.start_engine()
my_car.drive(20)

# Access private attribute indirectly using a getter method
print(f"Fuel level: {my_car.get_fuel_level()}")

# Attempting to access private attribute directly (will raise an error)
# print(my_car.__fuel_level)

# Access public method to modify private attribute
my_car.refuel(15)
print(f"Fuel level after refueling: {my_car.get_fuel_level()}")
```

In this example:

The Car class has private attributes (__make, __model, and __fuel_level) marked by using double underscores before their names. These attributes are not accessible directly from outside the class.

Public methods (start_engine, drive, get_fuel_level, refuel) provide controlled access to the private attributes. The get_fuel_level method acts as a getter method to retrieve the value of the private attribute __fuel_level.

Directly accessing private attributes from outside the class will result in an error.

Encapsulation allows the internal details of the Car class to be hidden, and access to the class is provided through well-defined public methods.

Encapsulation helps in achieving data hiding, reducing the impact of changes, and improving code maintainability by encapsulating the implementation details within the class.

Example 1

```
class BankAccount:
    def __init__(self, account_holder, balance=0):
        self.__account_holder = account_holder
        self.__balance = balance

    def deposit(self, amount):
        print(f"Depositing ${amount} into the account.")
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            print(f"Withdrawing ${amount} from the account.")
            self.__balance -= amount
        else:
            print("Insufficient funds!")

    def get_balance(self):
        return self.__balance

# Create an instance of the BankAccount class
account = BankAccount("John Doe")

# Access public methods to interact with the private attributes
account.deposit(1000)
```

```
account.withdraw(500)
print(f"Account balance: ${account.get_balance()}")
```

Example 2

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    def get_name(self):
        return self.__name

    def get_salary(self):
        return self.__salary

    def give_raise(self, raise_amount):
        print(f"Give raise of ${raise_amount} to {self.__name}.")
        self.__salary += raise_amount

# Create an instance of the Employee class
employee = Employee("Alice", 50000)

# Access public methods to retrieve and modify private attributes
print(f"{employee.get_name()}'s salary: ${employee.get_salary()}")
employee.give_raise(5000)
print(f"After raise, {employee.get_name()}'s salary: ${employee.get_salary()}")
```

Example 3

```
class SecuritySystem:
    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def authenticate(self, entered_username, entered_password):
        return (
            entered_username == self.__username
            and entered_password == self.__password
        )

    def change_password(self, new_password):
        print("Changing password.")
        self.__password = new_password
```

```
# Create an instance of the SecuritySystem class
security_system = SecuritySystem("admin", "secure_password")

# Access public methods to authenticate and change the password
if security_system.authenticate("admin", "secure_password"):
    print("Authentication successful.")
    security_system.change_password("new_secure_password")
else:
    print("Authentication failed.")
```

What is Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

POLYMORPHISM:

Polymorphism is one of the four fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance, and abstraction. It allows objects of different classes to be treated as objects of a common base class. Polymorphism enables a single interface to represent different types of objects, providing a way to use a single method or function with different types of inputs.

There are two types of polymorphism in Python: compile-time polymorphism (also known as method overloading) and runtime polymorphism (also known as method overriding). I'll provide examples for both.

1. Compile-Time Polymorphism (Method Overloading):
Method overloading allows a class to define multiple methods with the same name but different parameter lists. The appropriate method is selected at compile time based on the number and types of arguments provided.

```
class MathOperations:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

# Create an instance of the MathOperations class
math_obj = MathOperations()

# Call the overloaded methods
result1 = math_obj.add(2, 3)
```

```
result2 = math_obj.add(2, 3, 4)

print(f"Result 1: {result1}")
print(f"Result 2: {result2}")
```

In this example, the add method is overloaded with two different parameter lists. Depending on the number of arguments provided, the appropriate method is selected at compile time.

2. Runtime Polymorphism (Method Overriding):
Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The decision about which method to call is made at runtime.

```
class Animal:
    def make_sound(self):
        return "Generic animal sound."

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

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Create instances of the derived classes
dog_instance = Dog()
cat_instance = Cat()

# Call the overridden method
print(dog_instance.make_sound())  # Outputs "Woof!"
print(cat_instance.make_sound())  # Outputs "Meow!"
```

In this example, the Animal class has a method make_sound, and both the Dog and Cat classes override this method with their own specific implementations. The method called is determined at runtime based on the actual type of the object.

Polymorphism allows for more flexible and dynamic code, making it easier to extend and maintain as new classes can be added without modifying existing code. The ability to treat objects of different classes uniformly enhances code reusability and readability.

1. Operator Overloading:

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

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type.")

# Create instances of the Point class
point1 = Point(1, 2)
point2 = Point(3, 4)

# Use the overloaded '+' operator
result_point = point1 + point2

print(f"Result Point: ({result_point.x}, {result_point.y})")
```

In this example, the Point class overloads the + operator using the __add__ method, allowing instances of the class to be added together.

2. Polymorphic Function:

```
def print_sound(animal):
    print(animal.make_sound())

class Animal:
    def make_sound(self):
        return "Generic animal sound."

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

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Create instances of the derived classes
dog_instance = Dog()
cat_instance = Cat()
```

```
# Call the polymorphic function with different objects
print_sound(dog_instance)
print_sound(cat_instance)
```

In this example, the print_sound function can accept any object that has a make_sound method, demonstrating polymorphism.

3. Polymorphic List:

```
class Shape:
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2

# Create instances of the derived classes
circle_instance = Circle(5)
square_instance = Square(4)

# Create a list of shapes
shapes = [circle_instance, square_instance]

# Calculate and print the area of each shape in the list
for shape in shapes:
    print(f"Area: {shape.area()}")
```

In this example, both the Circle and Square classes inherit from the common base class Shape, and instances of these classes are stored in a list. The area method is called polymorphically on each object in the list.

These examples demonstrate different aspects of polymorphism, including operator overloading, polymorphic functions, and polymorphic lists. Each showcases the flexibility and versatility that polymorphism provides in Python OOP.

Abstraction :

Abstraction is one of the four fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance, and polymorphism. Abstraction involves simplifying complex systems by modeling classes based on essential characteristics and ignoring irrelevant details. It allows you to focus on what an object does rather than how it achieves its functionality. Abstract classes and abstract methods are key components of abstraction in Python.

Example of Abstraction:

```python
from abc import ABC, abstractmethod

# Abstract class with an abstract method
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Concrete class implementing the abstract class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Concrete class implementing the abstract class
class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2

# Create instances of the derived classes
circle_instance = Circle(5)
square_instance = Square(4)

# Call the area method polymorphically
```

```
print(f"Circle area: {circle_instance.area()}")
print(f"Square area: {square_instance.area()}")
```

In this example:

- The `Shape` class is an abstract class containing an abstract method `area`. Abstract classes cannot be instantiated, and abstract methods must be implemented by concrete (non-abstract) subclasses.

- The `Circle` and `Square` classes are concrete classes that inherit from the abstract `Shape` class. They provide specific implementations for the `area` method.

- By defining the abstract class `Shape` with an abstract method `area`, we abstract away the details of how each shape calculates its area. Users can work with shapes in a generic way, focusing on the common characteristics of shapes.

- The `abstractmethod` decorator from the `abc` module is used to indicate that the `area` method must be implemented by any concrete subclass.

Abstraction in this example allows users to work with shapes in a high-level way, using the common interface provided by the abstract class `Shape`. Concrete shapes (such as circles and squares) implement the details of their specific behavior, but users don't need to concern themselves with those details when working with shapes in a more abstract sense.

Certainly! Here are three simple examples demonstrating abstraction in Python OOP:

Example 1: Abstract Class and Method

```python
from abc import ABC, abstractmethod

# Abstract class with an abstract method
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

# Concrete classes implementing the abstract class
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
```

```
    def make_sound(self):
        return "Meow!"

# Create instances of the derived classes
dog_instance = Dog()
cat_instance = Cat()

# Call the abstract method polymorphically
print(dog_instance.make_sound())
print(cat_instance.make_sound())
```

In this example, the `Animal` class is an abstract class with an abstract method `make_sound`. Concrete classes `Dog` and `Cat` provide specific implementations for the `make_sound` method.

Example 2: Abstract Base Class (ABC)

```python
from abc import ABC, abstractmethod

# Abstract base class for shapes
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Concrete class implementing the abstract base class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Create an instance of the derived class
circle_instance = Circle(5)

# Call the abstract method polymorphically
print(f"Circle area: {circle_instance.area()}")
```

In this example, the `Shape` class is an abstract base class (ABC) with an abstract method `area`. The concrete class `Circle` inherits from the abstract base class and provides a specific implementation for the `area` method.

Example 3: Abstract Base Class for Data Validation

```python
from abc import ABC, abstractmethod

# Abstract base class for data validation
class Validator(ABC):
    @abstractmethod
    def validate(self, data):
        pass

# Concrete class implementing the abstract base class
class EmailValidator(Validator):
    def validate(self, data):
        if "@" in data and "." in data:
            return True
        else:
            return False

# Create an instance of the derived class
email_validator = EmailValidator()

# Call the abstract method polymorphically
email_result = email_validator.validate("user@example.com")
print(f"Email validation result: {email_result}")
```

In this example, the `Validator` class is an abstract base class (ABC) for data validation with an abstract method `validate`. The concrete class `EmailValidator` inherits from the abstract base class and provides a specific implementation for email validation. Users can create different validators by subclassing `Validator` and implementing the `validate` method.

These examples illustrate how abstraction allows you to define common interfaces and behaviors in abstract classes, leaving the specific implementations to concrete subclasses. Users can work with objects at a higher level, using the common interface provided by the abstract class, without needing to be concerned with the details of each concrete implementation.

STATIC VARIABLE:

In Python, a static variable is a variable that is shared among all instances of a class, rather than being unique to each instance. It is also sometimes referred to as a class variable, because it belongs to the class itself rather than any particular instance of the class.

Static variables are defined inside the class definition, but outside of any method definitions. They are typically initialized with a value, just like an instance variable, but they can be accessed and modified through the class itself, rather than through an instance.

Features of Static Variables
Static variables are allocated memory once when the object for the class is created for the first time.
Static variables are created outside of methods but inside a class
Static variables can be accessed through a class but not directly with an instance.
Static variables behavior doesn't change for every object.

In Python, static and class methods are special types of methods that are associated with a class rather than an instance of the class. They provide alternative ways to interact with a class without creating an instance of the class. Both types of methods are defined using the `@staticmethod` and `@classmethod` decorators, respectively.

Static Method:

A static method is a method that belongs to a class rather than an instance of the class. It does not have access to the instance or its attributes. Static methods are defined using the `@staticmethod` decorator.

Example:

```python
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

# Calling static methods without creating an instance
sum_result = MathOperations.add(2, 3)
product_result = MathOperations.multiply(4, 5)

print(f"Sum: {sum_result}")
print(f"Product: {product_result}")
```

```

In this example, `add` and `multiply` are static methods of the `MathOperations` class. They can be called using the class name without creating an instance of the class.

## Class Method:

A class method is a method that is bound to the class and not the instance of the class. It takes the class itself as its first parameter, usually named `cls`. Class methods are defined using the `@classmethod` decorator.

**Example:**

```python
class MyClass:
    class_variable = 0

    def __init__(self, value):
        self.instance_variable = value
        MyClass.class_variable += 1

    @classmethod
    def get_class_variable(cls):
        return cls.class_variable

    def get_instance_variable(self):
        return self.instance_variable

# Creating instances of the class
obj1 = MyClass(10)
obj2 = MyClass(20)

# Calling class and instance methods
print(f"Class variable: {MyClass.get_class_variable()}")
print(f"Instance variable (obj1): {obj1.get_instance_variable()}")
print(f"Instance variable (obj2): {obj2.get_instance_variable()}")
```

In this example, `get_class_variable` is a class method that accesses and returns the class variable `class_variable`. Class methods can be called on the class itself and are often used for operations that involve the class as a whole.

Key points:
```

- Static methods are independent of class and instance variables; they don't have access to `self` or `cls`.
- Class methods take the class itself (`cls`) as the first parameter and can access class variables.
- Both static and class methods are called on the class rather than an instance, using `ClassName.method()`.

Use static methods when the method doesn't depend on instance-specific or class-specific data. Use class methods when the method needs access to class-level data or operations.

Composition and **Aggregation** are two principles in object-oriented programming (OOP) that describe how objects can be related to each other. They define the relationships between classes and how they interact. Let's explore each concept with examples in Python.

Composition:

Composition is a strong form of association where one class contains an object of another class. It implies that the contained object has a part-of relationship with the container, and it cannot exist independently. When the container object is destroyed, the contained object is also destroyed.

Example:

```python
class Engine:
    def start(self):
        return "Engine started."

    def stop(self):
        return "Engine stopped."

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return f"Car started. {self.engine.start()}"

    def stop(self):
        return f"Car stopped. {self.engine.stop()}"

# Create an instance of the Car class
my_car = Car()
```

```
# Use composition to start and stop the car
print(my_car.start())
print(my_car.stop())
```

In this example, the `Car` class contains an instance of the `Engine` class. The `Car` object is composed of an `Engine`. The `start` and `stop` methods of the `Car` class delegate to the corresponding methods of the `Engine` class.

Aggregation:

Aggregation is a weaker form of association where one class contains another class, but the contained object can exist independently. It implies a part-of relationship, but the contained object has a longer lifecycle than the container. If the container is destroyed, the contained object can still exist.

Example:

```python
class Author:
    def __init__(self, name):
        self.name = name

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

# Create instances of the Author and Book classes
author = Author("John Doe")
book = Book("Python Programming", author)

# Use aggregation to associate an author with a book
print(f"Book title: {book.title}")
print(f"Author: {book.author.name}")
```

In this example, the `Book` class has an `Author` object as a member, but the `Author` object can exist independently of the `Book`. If the book is destroyed, the author object still exists.

Key points:
- Composition implies a strong relationship where one class is part of another.
- Aggregation implies a weaker relationship where one class contains another, but the contained object can exist independently.

- Both composition and aggregation help in creating modular and reusable code by building relationships between classes.

Choose between composition and aggregation based on the nature of the relationship between the classes. Use composition when the objects have a strong relationship, and use aggregation when the objects have a weaker relationship or can exist independently.

Association and **Dependency** are terms used in object-oriented programming (OOP) to describe relationships between classes.

Association:

Association represents a relationship between two or more classes, where objects of one class are related to objects of another class. It can be a simple or complex relationship. Association can be one-to-one, one-to-many, or many-to-many.

Example:

```python
class Student:
    def __init__(self, name):
        self.name = name

class Course:
    def __init__(self, title):
        self.title = title

class Enrollment:
    def __init__(self, student, course):
        self.student = student
        self.course = course

# Create instances of Student, Course, and Enrollment
student1 = Student("Alice")
course1 = Course("Python Programming")
enrollment1 = Enrollment(student1, course1)

# Representing association
print(f"{enrollment1.student.name} is enrolled in {enrollment1.course.title}.")
```

In this example, there is an association between the `Student` and `Course` classes through the `Enrollment` class. An `Enrollment` object associates a student with a course.

Dependency:

Dependency is a relationship where a change in one class (e.g., its method signature or behavior) can affect another class that depends on it. It's a one-way relationship, and a class is said to be dependent on another class if it uses the other class, but the reverse is not necessarily true.

Example:

```python
class Logger:
    def log(self, message):
        print(f"Logging: {message}")

class UserManager:
    def __init__(self, logger):
        self.logger = logger

    def create_user(self, username):
        # Business logic for creating a user
        self.logger.log(f"User created: {username}")

# Create instances of Logger and UserManager
logger = Logger()
user_manager = UserManager(logger)

# Representing dependency
user_manager.create_user("JohnDoe")
```

In this example, the `UserManager` class depends on the `Logger` class for logging purposes. If the `Logger` class changes its log method, it can affect the `UserManager` class.

Key points:
- **Association** represents a broader relationship between classes, often involving instances of those classes interacting with each other.
- **Dependency** is a more specific relationship where one class depends on another, and changes in the dependent class may affect the dependent class.
- Both association and dependency are essential for creating modular and maintainable code in OOP.

Understanding these concepts helps in designing classes that are loosely coupled and have clear and manageable relationships.

Aggregation and **Composition** are concepts in object-oriented programming that describe how classes can be related to each other in terms of ownership and lifecycle.

Aggregation:

Aggregation is a type of association that represents a "has-a" relationship between classes. It is a weaker form of relationship where one class contains another, but the contained object can exist independently. Aggregation implies a part-of relationship, but the contained object has a longer lifecycle than the container. If the container is destroyed, the contained object can still exist.

Example:

```python
class Department:
    def __init__(self, name):
        self.name = name

class University:
    def __init__(self, name):
        self.name = name
        self.departments = []  # Aggregation - University has-a list of departments

    def add_department(self, department):
        self.departments.append(department)

# Create instances of Department and University
math_department = Department("Mathematics")
physics_department = Department("Physics")

university = University("Example University")

# Use aggregation to associate departments with the university
university.add_department(math_department)
university.add_department(physics_department)

# Accessing departments from the university
for department in university.departments:
    print(f"Department in {university.name}: {department.name}")
```

In this example, the `University` class aggregates a list of `Department` objects. The departments can exist independently of the university, and the university can have multiple departments.

Composition:

Composition is a stronger form of association, often described as a "whole-part" relationship. In composition, one class is composed of another class, and the contained object cannot exist independently of the container. If the container is destroyed, the contained object is also destroyed.

Example:

```python
class Engine:
    def start(self):
        return "Engine started."

    def stop(self):
        return "Engine stopped."

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition - Car has-an Engine

    def start(self):
        return f"Car started. {self.engine.start()}"

    def stop(self):
        return f"Car stopped. {self.engine.stop()}"

# Create an instance of the Car class
my_car = Car()

# Use composition to start and stop the car
print(my_car.start())
print(my_car.stop())
```

In this example, the `Car` class is composed of an `Engine` class. The `Car` cannot exist without its `Engine`, and the `Engine` is tightly bound to the lifecycle of the `Car`.

Key points:

- Aggregation and composition both represent relationships between classes.
- Aggregation is a weaker relationship where the contained object can exist independently.

- Composition is a stronger relationship where the contained object cannot exist independently and is part of the container.
- Both concepts are important for designing flexible and modular object-oriented systems based on the desired level of ownership and lifecycle management.

Encapsulation and abstraction are both principles of object-oriented programming (OOP), but they address different aspects of designing and structuring code.

1. **Encapsulation:**
   - **Definition:** Encapsulation is the bundling of data and methods that operate on the data into a single unit, known as a class. It involves restricting access to the internal details of an object and exposing only what is necessary.
   - **Implementation:** This is achieved through the use of access specifiers like private and protected. Private attributes and methods are only accessible within the class, while protected attributes and methods are accessible within the class and its subclasses.
   - **Purpose:** Encapsulation helps in hiding the implementation details of an object, promoting data integrity and preventing direct access to the internal state. It also allows for controlled access to the object's attributes and methods through well-defined interfaces.

2. **Abstraction:**
   - **Definition:** Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors they exhibit, while hiding unnecessary details. It allows you to focus on the relevant aspects of an object or system without worrying about implementation details.
   - **Implementation:** Abstraction is often implemented using abstract classes and abstract methods. Abstract classes cannot be instantiated and may declare abstract methods that must be implemented by concrete subclasses.
   - **Purpose:** Abstraction helps in modeling real-world entities in a simplified manner, emphasizing what an object does rather than how it achieves it. It allows for the creation of well-defined interfaces and promotes code reusability by allowing different implementations to conform to a common abstraction.

In summary:
- **Encapsulation** is about bundling data and methods together, controlling access to them, and hiding the internal details of an object.
- **Abstraction** is about simplifying complex systems by modeling classes based on essential properties and behaviors, providing a clear separation between what an object does and how it achieves it.

While these principles are related, they serve different purposes in the design and implementation of object-oriented systems. Encapsulation provides a way to manage access to the internal state of an object, while abstraction allows for the creation of simplified and reusable models of complex systems.

In Python, the `@staticmethod` decorator is used to define a static method within a class. Static methods are methods that belong to a class rather than an instance of the class. Unlike regular methods, static methods don't have access to the instance or its attributes, and they don't modify or interact with the instance's state.

Here's a simple example to illustrate the concept of static methods in Python:

```python
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Using static methods without creating an instance of MathOperations
sum_result = MathOperations.add(5, 3)
product_result = MathOperations.multiply(4, 6)

print("Sum:", sum_result)       # Output: 8
print("Product:", product_result)  # Output: 24
```

In this example:

- The `add` and `multiply` methods are marked with the `@staticmethod` decorator.
- These methods can be called on the class itself (`MathOperations.add(...)`) without creating an instance of the class.

Key points about static methods:

1. **No access to instance attributes:** Static methods don't have access to the instance or its attributes. They work with the arguments passed to them and don't modify the instance's state.

2. **Bound to class, not instance:** Static methods are bound to the class rather than an instance. They can be called on the class itself or on an instance, but they don't receive the instance as the first parameter (`self`).

3. **Can't modify class or instance state:** Since static methods don't have access to the instance or class state, they can't modify attributes or call other instance methods.

Static methods are useful for utility functions that don't depend on the instance's state and don't need to be tied to a specific instance. They promote code organization and can be called more easily without the need to create an instance of the class.

Certainly! Here are a few more examples to illustrate the use of static methods in Python:

Example 1: Date Utility

```python
from datetime import date

class DateUtility:
    @staticmethod
    def is_leap_year(year):
        return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

# Using the static method without creating an instance
leap_year_check = DateUtility.is_leap_year(2024)
print("Is 2024 a leap year?", leap_year_check)  # Output: True
```

In this example, the static method `is_leap_year` checks if a given year is a leap year. It doesn't depend on any instance-specific state.

Example 2: File Management

```python
import os

class FileManager:
    @staticmethod
    def file_exists(file_path):
        return os.path.exists(file_path)

# Using the static method without creating an instance
file_check = FileManager.file_exists("example.txt")
print("File exists:", file_check)  # Output: False (assuming "example.txt" doesn't exist)
```

Here, the static method `file_exists` checks whether a file exists at the specified path. It doesn't need access to any instance-specific data.

Example 3: Currency Converter

```python
class CurrencyConverter:
    exchange_rate = 1.18  # Example exchange rate (EUR to USD)

    @staticmethod
    def convert_eur_to_usd(amount):
        return amount * CurrencyConverter.exchange_rate

# Using the static method without creating an instance
usd_amount = CurrencyConverter.convert_eur_to_usd(100)
print(f"Equivalent amount in USD: ${usd_amount:.2f}")  # Output: $118.00
```

In this example, the static method `convert_eur_to_usd` converts an amount from Euros to US Dollars. The exchange rate is stored as a class variable.

These examples demonstrate how static methods can be useful for encapsulating utility functions that are related to a class but don't depend on instance-specific data. They provide a convenient way to organize and reuse code within a class.

Cls :
In Python, `cls` is a commonly used convention as a parameter name within a class method. It stands for "class" and is used to refer to the class itself. Unlike the `self` parameter, which refers to the instance of the class, `cls` is used when you need to access or modify class-level attributes or methods.

Here's a breakdown of how `cls` is typically used in a class method:

1. **Class Method Definition:**
   ```python
   @classmethod
   def some_class_method(cls, arg1, arg2, ...):
       # Method implementation
   ```

2. **Using `cls` Inside the Method:**
   - `cls` is used to refer to the class itself.
   - You can access class-level attributes and methods using `cls.attribute` or `cls.method()`.
   - You can also create or modify class-level attributes using `cls.attribute = value`.

Here's a simple example to illustrate the use of `cls`:

```python
class MyClass:
```

```
    class_variable = 10  # Class variable

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

    @classmethod
    def update_class_variable(cls, new_value):
        cls.class_variable = new_value

    def display_info(self):
        print(f"Instance Variable: {self.instance_variable}")
        print(f"Class Variable: {MyClass.class_variable}")

# Create an instance of MyClass
obj = MyClass(instance_variable=5)

# Access instance method and class method
obj.display_info()

# Call the class method to update the class variable
MyClass.update_class_variable(new_value=20)

# Display information again
obj.display_info()
```

In this example:

- `cls` is used in the `update_class_variable` class method to update the class variable.
- The `display_info` instance method accesses both the instance variable and the class variable using `self` and `MyClass`.

The use of `cls` provides a way to work with class-level attributes and methods within class methods, offering flexibility and consistency in the object-oriented design of Python classes. Composition and aggregation are two concepts in object-oriented programming that describe relationships between classes.

Composition:

Definition:
- Composition is a strong association between two classes, where one class is composed of another class. It implies a whole-part relationship.
- In composition, the lifetime of the contained object is dependent on the lifetime of the container object. If the container is destroyed, the contained object is also destroyed.

Example:
```python
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start_car(self):
        print("Car starting...")
        self.engine.start()

# Example usage of composition
my_car = Car()
my_car.start_car()
```

In this example, the `Car` class has a composition relationship with the `Engine` class. The `Car` class has an instance variable `engine`, which is an instance of the `Engine` class. The `Car` class is composed of an `Engine`. When a car is created, it contains an engine, and the car's `start_car` method also starts the engine.

Aggregation:

Definition:
- Aggregation is a weaker association between two classes, where one class is part of another class but can exist independently. It implies a relationship where the contained object can exist outside the scope of the container object.
- In aggregation, the lifetime of the contained object is not dependent on the lifetime of the container object. If the container is destroyed, the contained object can still exist.

Example:
```python
class Author:
    def __init__(self, name):
        self.name = name

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

```
# Example usage of aggregation
author = Author("John Doe")
book = Book("Introduction to Python", author)
```

In this example, the `Book` class has an aggregation relationship with the `Author` class. The `Book` class has an instance variable `author`, which is an instance of the `Author` class. The `Author` object can exist independently of the `Book` object, and it can be associated with multiple books.

In summary, composition implies a strong relationship where one class is part of another, and the lifetime of the contained object is dependent on the container. Aggregation implies a weaker relationship where one class is part of another, but the contained object can exist independently. Both concepts help in designing flexible and maintainable object-oriented systems.
The `__str__` method in Python is a special method that is called by the built-in `str()` function and the `print()` function to obtain a string representation of an object. When you define the `__str__` method in a class, you are specifying how instances of that class should be represented as strings.

Here's a brief explanation of the `__str__` method:

- **Purpose:**
  - The primary purpose of the `__str__` method is to provide a human-readable and concise string representation of an object.

- **Usage:**
  - The `__str__` method is called automatically when you use the `str()` function on an object or when you print an object using the `print()` statement.

- **Syntax:**
  - The `__str__` method has the following syntax:
    ```python
    def __str__(self):
        # Return a string representation of the object
    ```

- **Example:**
  ```python
  class MyClass:
      def __init__(self, value):
          self.value = value

      def __str__(self):
```

```
        return f"MyClass instance with value: {self.value}"

# Example usage
obj = MyClass(42)
print(obj)  # This will call obj.__str__() automatically
```

- **Default Implementation:**
  - If you don't define a custom `__str__` method in your class, the default implementation inherited from the `object` class will be used. The default implementation returns a string containing the object's memory address.

- **Override vs. Inheritance:**
  - You can override the `__str__` method in your class to provide a custom string representation. This is often done to make the output more informative and user-friendly.

- **Human-Readable Representation:**
  - The `__str__` method is intended for a human-readable representation, so the string it returns should be clear and concise.

By defining a `__str__` method in your classes, you enhance the readability and usability of your code, especially when debugging or interacting with instances of your classes.

Association:

Association is a relationship between two or more classes or objects. It describes how objects interact with each other. In association, one class is connected to another class, and it can be one-to-one, one-to-many, or many-to-many.

There are different types of association:

1. **One-to-One (1:1):** A single instance of one class is associated with a single instance of another class.

   ```python
   class Person:
       def __init__(self, name):
           self.name = name

   class Passport:
       def __init__(self, number):
           self.number = number

   person = Person("John")
```

```
passport = Passport("ABC123")

# One-to-One association
person.passport = passport
```

2. **One-to-Many (1:N):** A single instance of one class is associated with multiple instances of another class.

```python
class Department:
    def __init__(self, name):
        self.name = name

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

department = Department("HR")
employee1 = Employee("Alice")
employee2 = Employee("Bob")

# One-to-Many association
department.employees = [employee1, employee2]
```

3. **Many-to-Many (N:M):** Multiple instances of one class are associated with multiple instances of another class.

```python
class Student:
    def __init__(self, name):
        self.name = name

class Course:
    def __init__(self, title):
        self.title = title

student1 = Student("John")
student2 = Student("Alice")
course1 = Course("Math")
course2 = Course("English")

# Many-to-Many association
```

```
class Student:
    def __init__(self, student_id, name):
        self.student_id = student_id
        self.name = name
        self.courses = []  # List to store enrolled courses

    def enroll_in_course(self, course):
        self.courses.append(course)
        course.add_student(self)  # Notify the course that the student is enrolled

    def __str__(self):
        return f"Student ID: {self.student_id}, Name: {self.name}"

class Course:
    def __init__(self, course_code, title):
        self.course_code = course_code
        self.title = title
        self.students = []  # List to store enrolled students

    def add_student(self, student):
        self.students.append(student)

    def __str__(self):
        return f"Course Code: {self.course_code}, Title: {self.title}"

# Example usage:
student1 = Student(student_id=1, name="John Doe")
student2 = Student(student_id=2, name="Jane Smith")

course1 = Course(course_code="CS101", title="Introduction to Computer Science")
course2 = Course(course_code="MATH202", title="Advanced Mathematics")

# Enroll students in courses
student1.enroll_in_course(course1)
student2.enroll_in_course(course1)
student2.enroll_in_course(course2)

# Display enrolled students for each course
for course in [course1, course2]:
    print(f"\nStudents enrolled in {course.title}:")
    for student in course.students:
        print(student)
```

Dependency:

Dependency represents a relationship where one class depends on another class. It means that a change in one class may affect the other class.

```python
class Car:
    def start_engine(self):
        pass

class Driver:
    def drive_car(self, car):
        car.start_engine()

# Dependency example
driver = Driver()
car = Car()
driver.drive_car(car)
```

In this example, the `Driver` class depends on the `Car` class. The `drive_car` method of the `Driver` class depends on the `start_engine` method of the `Car` class. If the `start_engine` method changes, it may affect the behavior of the `Driver` class.

In summary, association describes how classes are related, and dependency describes how one class depends on another. Associations can be long-term or short-term, while dependencies are usually more immediate and are often related to method calls or parameter passing.

In Python Object-Oriented Programming (OOP), the application of design principles is crucial for creating maintainable, scalable, and readable code. Let's explore how some design principles are applied in Python OOP.

1. **Single Responsibility Principle (SRP):**

- **Example:**
  ```python
  class FileManager:
      def read_file(self, filename):
          # Read file logic here
  ```

```
        pass

    def write_file(self, filename, data):
        # Write file logic here
        pass

class DataProcessor:
    def process_data(self, data):
        # Data processing logic here
        pass
```
Here, `FileManager` and `DataProcessor` each have a single responsibility.

2. **Open/Closed Principle (OCP):**

- **Example:**
```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side
```
Here, you can add new shapes (extension) without modifying the existing `Shape` class.

3. **Liskov Substitution Principle (LSP):**

- **Example:**
```python
```

```
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        # Sparrow-specific flying logic
        pass
```

The `Sparrow` class is a subclass of `Bird` and can be substituted for a `Bird` without affecting the program's correctness.

4. **Interface Segregation Principle (ISP):**

- **Example:**
```python
class Worker(ABC):
    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def eat(self):
        pass

class Engineer(Worker):
    def work(self):
        # Engineer-specific work logic
        pass

    def eat(self):
        # Engineer-specific eating logic
        pass
```

Here, `Engineer` implements only the methods relevant to it.

5. **Dependency Inversion Principle (DIP):**

- **Example:**
```python
class LightBulb:
    def turn_on(self):
        pass
```

```
    def turn_off(self):
        pass

class Switch:
    def __init__(self, device):
        self.device = device

    def operate(self):
        # Dependency inversion
        self.device.turn_on()

bulb = LightBulb()
switch = Switch(bulb)
switch.operate()
```

Here, `Switch` depends on the abstraction (`turn_on` method of `LightBulb`) rather than the concrete implementation.

6. **DRY (Don't Repeat Yourself):**

- **Example:**
```python
class MathUtility:
    @staticmethod
    def square(number):
        return number ** 2

result = MathUtility.square(5)
```

The `square` method provides a reusable function without duplicating code.

7. **KISS (Keep It Simple, Stupid):**

- **Example:**
```python
class Calculator:
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y
```

The `Calculator` class provides simple and easy-to-understand methods.

8. **YAGNI (You Aren't Gonna Need It):**

- **Example:**
  ```python
  class ReportGenerator:
      def generate_report(self, data):
          # Only implement what is needed now
          pass
  ```
  Avoid implementing features that are not currently required.

9. **Separation of Concerns (SoC):**

- **Example:**
  ```python
  class DatabaseConnector:
      def connect(self):
          pass

  class DataProcessor:
      def process_data(self, data):
          pass

  class ReportGenerator:
      def generate_report(self, data):
          pass
  ```
  Each class has a specific concern, such as connecting to a database, processing data, or generating reports.

10. **Law of Demeter:**

- **Example:**
  ```python
  class Person:
      def __init__(self, name):
          self.name = name

      def get_name(self):
          return self.name

  class Team:
      def __init__(self):
          self.members = []
```

```
    def add_member(self, person):
        self.members.append(person)

    def get_member_names(self):
        return [member.get_name() for member in self.members]
```

The `Team` class does not directly access the internals of `Person` but relies on the `get_name` method.

Applying these design principles in Python OOP helps create code that is more modular, flexible, and maintainable. It also promotes good coding practices and makes the codebase easier to understand and extend.