Python OOPs Assignment

Couse:- Data Science with Generative AI (May 2025)

Q(1) What is Object-Oriented Programming (OOP)?

Ans:- Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic. An object is a self-contained unit that consists of data (attributes) and behavior (methods). OOP focuses on creating reusable, modular, and scalable code by modeling real-world entities.

Core concepts of OOPs

- Class, A blueprint or template for creating objects.
- Object (Instance), A specific instance of a class with actual data.
- Encapsulation, Bundling data (attributes) and methods that operate on the data within a single unit (class).
- Inheritance, Allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass).
- Polymorphism, The ability of different classes to be treated as instances of the same class through a common interface.
- Abstraction, Hiding complex implementation details and exposing only essential features.

Benefits of OOPs

- Modularity – Code is organized into reusable components.
- Reusability – Inheritance and composition reduce redundant code.
- Scalability – Easier to maintain and extend large applications.
- Security – Encapsulation protects data integrity.

Q(2) What is a class in OOP?

Ans:-

Class is a blueprint or template for creating objects.

It defines the structure (attributes and methods) that objects will have.

```
class Car:
    def init(self, brand, model):
        self.brand = brand # Attribute
            self.model = model # Attribute
    def drive(self): # Method
        print(f"{self.brand} {self.model} is driving.")
```

Q(3) What is an object in OOP?

Ans:-

Object (Instance) is a specific instance of a class with actual data.

```
    my_car = Car("Suzuki", "Alto") # Creating an object
    my_car.drive() # Output: "Suzuki Alto is driving."
```

Q(4) What is the difference between abstraction and encapsulation?

Ans:-

Abstraction

Abstraction focuses on hiding complex implementation details and exposing only the essential features of an object. It deals with the "what" (interface) rather than the "how" (implementation).

- It simplify complex systems by showing only relevant details.
- Reduce complexity for the user.
- Achieved via abstract classes, interfaces, and method overloading.

e.g. A car's dashboard (shows speed, fuel level) hides the engine's internal workings.

Encapsulation

Encapsulation is about bundling data (attributes) and methods (functions) that operate on the data into a single unit (class) and restricting direct access to some components. It emphasizes data protection.

- Prevents unintended modification of data. So data is secure.
- Control access via getters/setters or access modifiers (private, protected).
- Achieved through access control (e.g., private variables in Java/C++).

e.g. A bank account hides the balance but provides methods to deposit/withdraw.

Q(5) What are dunder methods in Python?

Ans:-

Dunder methods (short for "double underscore") are special predefined methods in Python that start and end with double underscores (__). They are also called magic methods because they enable powerful behaviors like operator overloading, object initialization, and more.

Some common dunder methods in Python

- **init**(self, ...) → Constructor (initializes an object).
- **str**(self) → Informal string representation (str(obj) or print(obj)).
- **repr**(self) → Formal string representation (used in debugging, repr(obj)).
- **add**(self, other) → addition
- **sub**(self, other) → subtration
- **eq**(self, other) → equal to
- **lt**(self, other) → less than
- **mul**(self, other) → multiplication
- **len**(self) → Called by len(obj).
- **getitem**(self, key) → Called by obj[key].
- **setitem**(self, key, value) → Called by obj[key] = value.
- **contains**(self, item) → Called by item in obj.
- **enter**(self) → Runs at the start of with.
- **exit**(self, exc_type, exc_val, exc_tb) → Runs at the end of with.
- **call** Makes an object behave like a function.

Q(6) Explain the concept of inheritance in OOP.

Ans:-

Inheritance is a fundamental OOP concept where a child class (subclass) inherits attributes and methods from a parent class (superclass). It promotes code reusability and establishes a hierarchical relationship between classes.

Types of Inheritance

- Single Inheritance, a subclass inherits from one parent class.
- Multiple Inheritance, a subclass inherits from multiple parent classes.
- Multilevel Inheritance, a subclass is used as a parent class to create another subclass. (A chain of inheritance as grandparent → parent → child)
- Hierarchical Inheritance, multiple subclasses inherit from a single parent.
- Hybrid Inheritance, a combination of multiple and multilevel inheritance.

Inheritance is used for

- Code Reusability – Avoid rewriting the same code.
- Modularity – Organize code hierarchically.
- Extensibility – Easily add new features to existing classes.
- Polymorphism – Enables method overriding for flexible behavior.

Q(7) What is polymorphism in OOP?

Ans:-

Polymorphism ("poly" = many, "morph" = forms) is the ability of objects of different classes to respond to the same method call in different ways. It allows one interface to represent different underlying forms (data types).

It is of two types:

1. Compile-Time Polymorphism (Static Polymorphism)

- Achieved via method overloading (same method name, different parameters).
- Not natively supported in Python (but possible with default args or *args).

2. Runtime Polymorphism (Dynamic Polymorphism)

- Achieved via method overriding (subclass redefines a parent method).
- Supported in Python and most OOP languages.

Polymorphism is used for

- Flexibility: Same method name works across different classes.
- Extensibility: Easily add new classes without changing existing code.
- Cleaner Code: Reduces conditional checks (e.g., no if isinstance(obj, Dog):).

Q(8) How is encapsulation achieved in Python?

Ans:-

Encapsulation is the practice of bundling data (attributes) and methods (functions) that operate on the data into a single unit (class) while restricting direct access to some components. In Python, encapsulation is achieved using:

1. Naming Conventions (to indicate "private" members).
2. Property Decorators (@property, @attribute.setter) for controlled access.
3. Name Mangling (limited private variables with __).

** Encapsulation is used for**

- Data Protection: Prevent accidental modification of critical data.
- Validation: Ensure data integrity (e.g., negative balance check).
- Flexibility: Change internal implementation without breaking external code.
- Abstraction: Hide complex details from users.

Q(9) What is a constructor in Python?

Ans:-

A constructor is a special method used to initialize an object when it is created. The most commonly used constructor in Python is the **init**() method.

- It is called automatically when an object of a class is instantiated.
- It is used to set initial values for object attributes or perform setup
- It refers to the instance of the class (similar to "this" in Java) operations.

Types of Constructor

- Default Constructor, if **init**() is not defined, Python provides a default constructor that does nothing.
- Parameterized Constructor, Accepts arguments to initialize object attributes dynamically.

Q(10) What are class and static methods in Python?

Ans:-

n Python, *class methods* and *static methods* are special types of methods defined in a class, differing from regular *instance methods* in how they behave and their use cases.

Class Methods (@classmethod)

- Defined with the @classmethod decorator.
- Take cls as the first parameter (refers to the class, not the instance).
- Can modify class state (e.g., class variables) but not instance state.
- Often used as alternative constructors (e.g., from_string()).

Static Methods (@staticmethod)

- Defined with the @staticmethod decorator.
- No self or cls parameter (like a standalone function but belongs to the class).
- Cannot modify class or instance state (no access to cls or self).
- Used for utility functions logically grouped under the class.

Uses

- Instance Method is default for most cases (operates on instance data).
- Class Method is used when we need to work with the class itself (e.g., factory methods).
- Static Method is used for pure functions that don't need class/instance context.

Q(11) What is method overloading in Python?

Ans:-

In Python, method overloading means writing more than one method with the same method name but all differing in arguments and dynamic typing.

Example 1 - Using Flexible Arguments (Default Values, _args, *_kwargs)

```
class Calculator:
    def add(self, *args):
        return sum(args)
calc = Calculator()
calc.add(1, 2) # 3
calc.add(1, 2, 3, 4) # 10
```

Example 2 - Using Default Parameters

```
class Greeter:
    def greet(self, name, message="Hello"): # Default value
        return f"{message}, {name}!"
greeter = Greeter()
greeter.greet("Anoop") # "Hello, Anoop!"
greeter.greet("Verma", "Hi") # "Hi, Verma!"
```

Example 3 - Type Handling (Dynamic Typing)

class Printer:br>

def print_data(self, data):

```
if isinstance(data, str):<br>

        print(f"String: {data}")<br>
    elif isinstance(data, int):<br>
        print(f"Integer: {data}")
```

printer = Printer() printer.print_data("hello") # String: hello printer.print_data(42) # Integer: 42

Q(12) What is method overriding in OOP?

Ans:-

Method overriding is an Object-Oriented Programming (OOP) concept where a subclass provides a specific implementation of a method that is already defined in its parent class. The overridden method in the subclass replaces the parent class's implementation when called on a subclass object.

- Only possible when a child class inherits from a parent class.
- The method name, parameters, and return type must match the parent class method.
- The correct method is determined at runtime (dynamic method dispatch).

Method overriding allows subclasses to redefine inherited methods. It is essential for runtime polymorphism in OOP.

Q(13) What is a property decorator in Python?

Ans:-

The @property decorator is used to define "getter", "setter", and "deleter" methods for class attributes, allowing controlled access to instance variables. It enables us to add logic around attribute access (e.g., validation, computed values) while maintaining a clean, attribute-like syntax.

It is used for:

- Encapsulation: Hide internal implementation details.
- Validation: Ensure values meet conditions before assignment.
- Computed Attributes: Dynamically generate values (e.g., full_name from first_name + last_name).
- Backward Compatibility: Modify attribute behavior without changing external code.

Q(14) Why is polymorphism important in OOP?

Ans:-

Polymorphism is a core principle of Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enhances flexibility, maintainability, and scalability in software design.

It means "Single Interface, Multiple Implementations" i.e. Different classes can define their own behavior for the same method.

It allows "Runtime Decision-Making" i.e. the correct method is chosen at runtime (dynamic binding).

Importance of Polymorphism

- Code Reusability
- Flexibility
- Extensibility
- Simplifies Complex coding
- Enables Dependency
- Mimics Polymorphic behaviour of real world

Q(15) What is an abstract class in Python?

Ans:

An abstract class is a class that cannot be instantiated and is designed to be inherited by subclasses. It is like a template for derived classes by defining abstract methods (methods that must be implemented by child classes).

It is defined using the abc module (ABC class and @abstractmethod decorator).

- ABC marks the class as abstract.
- @abstractmethod enforces that subclasses must implement the method.

It is used to:

- Ensure all subclasses implement critical methods (e.g., area(), save()).
- Share common logic in concrete methods (e.g., description() in Shape).
- Treat all subclasses uniformly via the abstract base class. (POlymorphism)

Q(16) What are the advantages of OOP?

Ans:-

Object-Oriented Programming (OOP) organizes software design around objects rather than functions and logic. It provides several key benefits that make code more modular, reusable, and maintainable.

Modularity & Organization

- Objects encapsulate data and behavior, making code easier to understand and manage.
- Each class is a self-contained module, reducing complexity in large projects.

Reusability

- Inheritance allows classes to reuse code from parent classes.
- Avoids duplication, saving development time and reducing errors.

Encapsulation (Data Hiding)

- Protects internal data by restricting direct access (using private attributes and getters/setters).
- Prevents unintended modifications, improving security and reliability.

Polymorphism (Flexibility)

- Same method behaves differently based on the object’s class.
- Simplifies code by allowing a single interface for different data types.

Easier Maintenance & Scalability

- Changes in one class don’t break others (low coupling).
- New features can be added without rewriting existing code.

Better Problem-Solving

- Models real-world entities (e.g., User, Product, Order).
- Mimics human thinking, making design more intuitive.

Support for Design Patterns

- OOP enables proven solutions to common problems

Collaboration & Teamwork

- Clear separation of concerns allows teams to work on different classes simultaneously.
- Easier debugging due to modular structure.

Q(17) What is the difference between a class variable and an instance variable?

Ans:-

Class Variables -Shared across all instances of the class.

- Defined outside any method in the class.
- Typically used for constants, counters, or default values.

Instance Variables

- Unique to each object (instance).
- Defined inside **init** or methods using self.
- Used for object-specific attributes (e.g., name, age).

Q(18) What is multiple inheritance in Python?

Ans:-

Multiple inheritance allows a class to inherit attributes and methods from more than one parent class. This enables a subclass to combine functionalities from multiple sources.

class Parent1:

> def method1(self):
>
> > ```
> > print("Parent1's method")<br>
> > ```

class Parent2:

> def method2(self):
>
> > ```
> > print("Parent2's method")<br>
> > ```

class Child(Parent1, Parent2): # Inherits from both Parent1 and Parent2

obj = Child()
obj.method1() # Output: "Parent1's method"
obj.method2() # Output: "Parent2's method"

Q(19) Explain the purpose of ”**str**’ and ‘**repr**’ ‘ methods in Python.

Ans:-

Both **str** and **repr** are special methods used to define how objects are represented as strings, but they serve different purposes:

str -Provides a user-friendly, informal string representation of an object.

- Used by: print(obj) & str(obj)
- Best for: End-user output (e.g., logging, UI messages).

repr

- Provides a formal, unambiguous string representation (ideally, valid Python code to recreate the object).
- Used by: repr(obj)
- Best for: Debugging, logging, and developers.

Q(20) What is the significance of the 'super()' function in Python?

Ans:-

The super() function in Python is used to call methods from a parent class in inheritance hierarchies. It plays a crucial role in method overriding, cooperative multiple inheritance, and maintaining clean, reusable code.

In multiple inheritance, super() follows the Method Resolution Order to call methods from all parent classes without conflicts.

Q(21) What is the significance of the **del** method in Python?

Ans:-

The **del** method is a destructor in Python, called when an object is about to be destroyed (garbage-collected) or it is going out of scope. However, its behavior is unpredictable and rarely used in practice.

- Called automatically when an object's reference count drops to zero.
- Used for cleanup (e.g., closing files, releasing resources).

Q(22) What is the difference between @staticmethod and @classmethod in Python?

Ans:-

@classmethod

- Used when the method needs to access or modify the class state (e.g., class variables).
- cls (automatically passed, refers to the class) is the first parameter.
- Used as Alternative constructors (e.g., from_string()) or Modifying class-wide configurations.

@staticmethod

- Used when the method doesn't need class/instance state (like a standalone function but belongs to the class).
- No Implicit Parameters like cls or self
- Used for Helper/utility functions (e.g., mathematical operations) or Grouping related functions under a class namespace.

Q(23) How does polymorphism work in Python with inheritance?

Ans:-

Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexible and reusable code. In Python, polymorphism works primarily through method overriding and duck typing.

Method Overriding (Runtime Polymorphism) When a subclass provides its own implementation of a method already defined in its parent class, it overrides the parent's method.

Duck Typing (Flexible Polymorphism) Python doesn't require explicit inheritance for polymorphism. If an object behaves like the expected type (has the required method), it works!

Polymorphism with Built-in Functions Python's built-ins (e.g., len(), print()) use polymorphism to work with different types.

Polymorphism in Operator Overloading Operators (e.g., +, *) behave differently based on object types.

Q(24) What is method chaining in Python OOP?

Ans:-

Method chaining is a technique where multiple methods are called on an object in a single line, with each method returning the object itself (self). This creates a fluent, readable code.

- Each method returns self (or a new instance), enabling sequential calls.
- Avoids intermediate variables, making code concise.

Q(25) What is the purpose of the **call** method in Python?

Ans:-

The **call** method allows an instance of a class to be called like a function. When defined, we can use the object with () syntax, making it callable.

- It turns an object into a callable (like functions, but with state).
- Useful for objects that need to maintain state between calls.
- Often used in decorators, functor patterns, and stateful function-like objects.

```
# Q(1) Create a parent class Animal with a method speak() that prints a generic
# message. Create a child class Dog that overrides the speak() method to
# print "Bark!".

class Animal:
    def speak(self):
        print("This animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Bark!")

# Testing the classes
generic_animal = Animal()
generic_animal.speak()  # Output: "This animal makes a sound"

my_dog = Dog()
my_dog.speak()          # Output: "Bark!"
```

```
This animal makes a sound
Bark!
```

```
# Q(2)  Write a program to create an abstract class Shape with a method area().
# Derive classes Circle and Rectangle from it and implement the area() method
# in both.

from abc import ABC, abstractmethod
import math

class Shape(ABC):  # Abstract Base Class
    @abstractmethod
    def area(self):
        pass  # Abstract method (no implementation)

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

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

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

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

# Demonstration
if __name__ == "__main__":
    # Can't instantiate abstract class
    # shape = Shape()  # This would raise TypeError

    circle = Circle(5)
    rectangle = Rectangle(4, 6)

    print(f"Area of circle with radius 5: {circle.area():.2f}")
    print(f"Area of rectangle 4x6: {rectangle.area()}")
```

```
Area of circle with radius 5: 78.54
Area of rectangle 4x6: 24
```

```
# Q(3) Implement a multi-level inheritance scenario where a class Vehicle has
# an attribute type. Derive a class Car and further derive a class ElectricCar
# that adds a battery attribute.

class Vehicle:
    def __init__(self, type):
        self.type = type  # Common attribute for all vehicles

    def display(self):
        print(f"Vehicle type: {self.type}")

class Car(Vehicle):
    def __init__(self, type, brand):
        super().__init__(type)  # Initialize Vehicle attributes
        self.brand = brand     # Car-specific attribute

    def display(self):
        super().display()      # Call parent's display method
        print(f"Brand: {self.brand}")

class ElectricCar(Car):
    def __init__(self, type, brand, battery_capacity):
        super().__init__(type, brand)  # Initialize Car attributes
        self.battery_capacity = battery_capacity  # ElectricCar-specific attribute

    def display(self):
        super().display()      # Call Car's display method
        print(f"Battery capacity: {self.battery_capacity} kWh")

# Testing the classes
if __name__ == "__main__":
    print("Regular Vehicle:")
    vehicle = Vehicle("Motorcycle")
    vehicle.display()
    print("\nStandard Car:")
    car = Car("Sedan", "Toyota")
    car.display()

    print("\nElectric Car:")
    electric_car = ElectricCar("Tata", "Nexon", 200)
    electric_car.display()
```

```
Regular Vehicle:
Vehicle type: Motorcycle

Standard Car:
Vehicle type: Sedan
Brand: Toyota

Electric Car:
Vehicle type: Tata
Brand: Nexon
Battery capacity: 200 kWh
```

```
# Q(4) Demonstrate polymorphism by creating a base class Bird with a method
# fly(). Create two derived classes Sparrow and Penguin that override the
# fly() method.

class Bird:
    def fly(self):
        print("This bird can fly")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flies fast")

class Penguin(Bird):
    def fly(self):
        print("Penguin cannot fly.")

def demonstrate_flight(bird):
    print(f"\nDemonstrating flight for {bird.__class__.__name__}:")
    bird.fly()

# Creating instances
generic_bird = Bird()
house_sparrow = Sparrow()
emperor_penguin = Penguin()

# Demonstrating polymorphism
for bird in [generic_bird, house_sparrow, emperor_penguin]:
    demonstrate_flight(bird)
```

```

Demonstrating flight for Bird:
This bird can fly

Demonstrating flight for Sparrow:
Sparrow flies fast

Demonstrating flight for Penguin:
Penguin cannot fly.
```

```
# Q(5) Write a program to demonstrate encapsulation by creating a class
# BankAccount with private attributes balance and methods to deposit, withdraw,
# and check balance.

class BankAccount:
    def __init__(self, initial_balance=0):
        self.__balance = initial_balance  # Private attribute

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ₹{amount}. Remaining balance: ₹{self.__balance}")
        else:
            print("Invalid withdrawal amount")

    def get_balance(self):
        return self.__balance

    # Alternative using property decorator
    @property
    def balance(self):
        return self.__balance
```

```
# Demonstration
if __name__ == "__main__":
    account = BankAccount(1000)

    print("Initial balance:", account.get_balance())  # Access via method
    account.deposit(500)
    account.withdraw(200)
    account.withdraw(2000)  # Should fail

    # Trying to access private attribute directly
    try:
        print(account.__balance)  # This will fail
    except AttributeError as e:
        print(f"Error: {e}")

    # Proper way to check balance
    print("Current balance (via property): ₹", account.balance)
```

```
Initial balance: 1000
Deposited ₹500. New balance: ₹1500
Withdrew ₹200. Remaining balance: ₹1300
Invalid withdrawal amount
Error: 'BankAccount' object has no attribute '__balance'
Current balance (via property): ₹ 1300
```

```
# Q(6) Demonstrate runtime polymorphism using a method play() in a base class
# Instrument. Derive classes Guitar and Piano that implement their own version
# of play().

class Instrument:
    def play(self):
        print("An instrument is playing...")

class Guitar(Instrument):
    def play(self):
        print("Strumming guitar strings!")

class Piano(Instrument):
    def play(self):
        print("Pressing piano keys!")

def perform(instrument: Instrument):
    print("\nPerformance starting...")
    instrument.play()
    print("Performance ended!\n")

# Create instances
generic_instrument = Instrument()
electric_guitar = Guitar()
grand_piano = Piano()

# Demonstrate runtime polymorphism
for instrument in [generic_instrument, electric_guitar, grand_piano]:
    perform(instrument)

# Alternatively, we can call play() directly
print("Direct calls:")
electric_guitar.play()
grand_piano.play()
```

```

Performance starting...
An instrument is playing...
Performance ended!


Performance starting...
Strumming guitar strings!
Performance ended!


Performance starting...
Pressing piano keys!
Performance ended!

Direct calls:
Strumming guitar strings!
Pressing piano keys!
```

```
# Q(7) Create a class MathOperations with a class method add_numbers() to add
# two numbers and a static method subtract_numbers() to subtract two numbers.

class MathOperations:

    @classmethod
    def add_numbers(cls, num1, num2):
        """Class method to add two numbers"""
        return num1 + num2
    @staticmethod
    def subtract_numbers(num1, num2):
        """Static method to subtract two numbers"""
        return num1 - num2

# Using the methods
sum_result = MathOperations.add_numbers(10, 5)
difference = MathOperations.subtract_numbers(10, 5)

print(f"Addition result: {sum_result}")       # Output: Addition result: 15
print(f"Subtraction result: {difference}")    # Output: Subtraction result: 5
```

```
Addition result: 15
Subtraction result: 5
```

```
# Q(8) Implement a class Person with a class method to count the total number
# of persons created.

class Person:
    # Class variable to keep track of count
    _total_persons = 0

    def __init__(self, name):
        self.name = name
        Person._total_persons += 1  # Increment count when new instance is created

    @classmethod
    def get_total_persons(cls):
        """Class method to return the total number of Person instances created"""
        return cls._total_persons

    def __del__(self):
        """Destructor to decrement count when instance is deleted"""
        Person._total_persons -= 1

# Demonstration
if __name__ == "__main__":
    print("Initial count:", Person.get_total_persons())  # Output: 0

    p1 = Person("Alice")
    p2 = Person("Bob")
    p3 = Person("Charlie")

    print("After creating 3 persons:", Person.get_total_persons())  # Output: 3

    del p2  # Explicitly delete one instance
    print("After deleting one person:", Person.get_total_persons())  # Output: 2
```

```
Initial count: 0
After creating 3 persons: 3
After deleting one person: 2
```

```
# Q(9) Write a class Fraction with attributes numerator and denominator.
# Override the str method to display the fraction as "numerator/denominator".

class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

# Example usage
half = Fraction(1, 2)
third = Fraction(1, 3)

print(half)    # Output: 1/2
print(third)   # Output: 1/3
```

```
1/2
1/3
```

```
# Q(10) Demonstrate operator overloading by creating a class Vector and
# overriding the add method to add two vectors.

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

    def __add__(self, other):
        """Overload the + operator to add two vectors"""
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        raise TypeError("Operands must be of type Vector")
    def __str__(self):
        """String representation of the vector"""
        return f"Vector({self.x}, {self.y})"

# Create two vectors
v1 = Vector(2, 3)
v2 = Vector(1, 4)

# Add them using the overloaded + operator
result = v1 + v2

print("Vector 1 = ", v1)     # Output: Vector(2, 3)
print("Vectotr 2 = ", v2)     # Output: Vector(1, 4)
print("Added = ", result) # Output: Vector(3, 7)
```

```
Vector 1 =  Vector(2, 3)
Vectotr 2 =  Vector(1, 4)
Added =  Vector(3, 7)
```

```
# Q(11) Create a class Person with attributes name and age. Add a method greet()
# that prints "Hello, my name is {name} and I am {age} years old."

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Example usage
person1 = Person("Anoop", 49)
person2 = Person("Bablu", 25)

person1.greet()  # Output: Hello, my name is Alice and I am 30 years old.
person2.greet()  # Output: Hello, my name is Bob and I am 25 years old.
```

```
Hello, my name is Anoop and I am 49 years old.
Hello, my name is Bablu and I am 25 years old.
```

```
# Q(12) Implement a class Student with attributes name and grades. Create a
# method average_grade() to compute the average of the grades.

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

    def average_grade(self):
        """Calculate and return the average grade"""
        if not self.grades:  # Handle empty list case
            return 0.0
        return sum(self.grades) / len(self.grades)

# Example usage
student1 = Student("Anoop", [85, 90, 78, 92])
student2 = Student("Bablu", [])  # Student with no grades

print(f"{student1.name}'s average grade: {student1.average_grade():.2f}")
# Output: Alice's average grade: 86.25

print(f"{student2.name}'s average grade: {student2.average_grade():.2f}")
# Output: Bob's average grade: 0.00
```

```
Anoop's average grade: 86.25
Bablu's average grade: 0.00
```

```
# Q(13) Create a class Rectangle with methods set_dimensions() to set the
# dimensions and area() to calculate the area.

class Rectangle:
    def __init__(self, length=0, width=0):
        self.length = length
        self.width = width

    def set_dimensions(self, length, width):
        """Set the dimensions of the rectangle"""
        if length <= 0 or width <= 0:
            raise ValueError("Dimensions must be positive numbers")
        self.length = length
        self.width = width

    def area(self):
        """Calculate and return the area of the rectangle"""
        return self.length * self.width
    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width}, area={self.area()})"

# Example usage
rect = Rectangle()  # Creates rectangle with default dimensions 0, 0
print(rect)  # Output: Rectangle(length=0, width=0, area=0)

rect.set_dimensions(5, 3)
print(rect)  # Output: Rectangle(length=5, width=3, area=15)

# Trying to set invalid dimensions
try:
    rect.set_dimensions(-2, 4)  # Will raise ValueError
except ValueError as e:
    print(f"Error: {e}")  # Output: Error: Dimensions must be positive numbers
```

```
Rectangle(length=0, width=0, area=0)
Rectangle(length=5, width=3, area=15)
Error: Dimensions must be positive numbers
```

```
# Q(14) Create a class Employee with a method calculate_salary() that computes
# the salary based on hours worked and hourly rate. Create a derived class
# Manager that adds a bonus to the salary.

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

    def calculate_salary(self, hours_worked):
        """Calculate base salary from hours worked and hourly rate"""
        if hours_worked < 0:
            raise ValueError("Hours worked cannot be negative")
        if self.hourly_rate < 0:
            raise ValueError("Hourly rate cannot be negative")

        return hours_worked * self.hourly_rate

class Manager(Employee):
    def __init__(self, name, hourly_rate, bonus):
        super().__init__(name, hourly_rate)
        self.bonus = bonus

    def calculate_salary(self, hours_worked):
        """Calculate salary with bonus included"""
        base_salary = super().calculate_salary(hours_worked)
        return base_salary + self.bonus

# Example usage
regular_employee = Employee("Anoop Verma", 20.0)
manager = Manager("Bablu Singh", 30.0, 1000.0)  # ₹1000 bonus

print(f"{regular_employee.name}'s salary: ₹{regular_employee.calculate_salary(40):.2f}")
# Output: Anoop Verma's salary: ₹800.00

print(f"{manager.name}'s salary: ₹{manager.calculate_salary(40):.2f}")
# Output: Bablu Singh's salary: ₹2200.00
```

```
Anoop Verma's salary: ₹800.00
Bablu Singh's salary: ₹2200.00
```

```
# Q(15) Create a class Product with attributes name, price, and quantity.
# Implement a method total_price() that calculates the total price of the
# product.

class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def total_price(self):
        """Calculate the total price (price × quantity)"""
        return self.price * self.quantity

    def __str__(self):
        return (f"Product: {self.name}\n"
                f"Price per unit: ₹{self.price:.2f}\n"
                f"Quantity: {self.quantity}\n"
                f"Total price: ₹{self.total_price():.2f}")

# Example usage
laptop = Product("Laptop", 999.99, 3)
print(laptop)

# Output:
# Product: Laptop
# Price per unit: ₹999.99
# Quantity: 3
# Total price: ₹2999.97
```

```
Product: Laptop
Price per unit: ₹999.99
Quantity: 3
Total price: ₹2999.97
```

```
# Q(16) Create a class Animal with an abstract method sound(). Create two
# derived classes Cow and Sheep that implement the sound() method.

from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract base class
    @abstractmethod
    def sound(self):
        pass

class Cow(Animal):
    def sound(self):
        return "Cow Sound - Moooooon!"

class Sheep(Animal):
    def sound(self):
        return "Sheep Sound - Baaaan!"

# Demonstration
if __name__ == "__main__":
    # animal = Animal()  # This would raise TypeError (can't instantiate abstract class)

    cow_obj = Cow()
    sheep_obj = Sheep()

    print(f"Cow says: {cow_obj.sound()}")    # Output: Cow says: Moo!
    print(f"Sheep says: {sheep_obj.sound()}")   # Output: Sheep says: Baa!
```

```
Cow says: Cow Sound - Moooooon!
Sheep says: Sheep Sound - Baaaan!
```

```
# Q(17) Create a class Book with attributes title, author, and year_published.
# Add a method get_book_info() that returns a formatted string with the
# book's details.

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

    def get_book_info(self):
        """Return a formatted string with the book's details"""
        return f"'{self.title}' by {self.author} ({self.year_published})"

# Example usage
book = Book("The C++ Programming Language", "Bjarne Stroustrupe", 1991)
print(book.get_book_info())
```

```
'The C++ Programming Language' by Bjarne Stroustrupe (1991)
```

```python
# Q(18) Create a class House with attributes address and price. Create a derived
# class Mansion that adds an attribute number_of_rooms.

class House:
    def __init__(self, address, price):
        self.address = address
        self.price = price

    def display_info(self):
        return f"Address: {self.address}, Price: ₹{self.price:,}"

class Mansion(House):
    def __init__(self, address, price, number_of_rooms):
        super().__init__(address, price)  # Initialize House attributes
        self.number_of_rooms = number_of_rooms

    def display_info(self):
        base_info = super().display_info()  # Get House's display info
        return f"{base_info}, Rooms: {self.number_of_rooms}"

# Example usage
```