RAMAIAH
Institute of Technology

Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science & Engineering

Data Visualization with Python Lab(CSL48)

USN: **Week #: 03**

Semester: **Section:** **Date:**

Instructions:

- **Implement the following programs using python language.**

Topic: Object-Oriented Programming (OOP), covering **Classes, Objects, Inheritance, Polymorphism, Encapsulation, and Abstraction**.

Programs:

1. Write a Python class named `Car` with attributes `brand`, `model`, and `year`. Create an object and display its attributes.

2. Write a Python class `Student` with attributes `name` and `marks`, and a method to display the student details.

3. Write a Python program where `Dog` and `Cat` classes inherit from a base class `Animal`, and implement a method `speak()`.

4. Create a class `Vehicle` with attributes `brand` and `speed`. Derive a class `Car` that adds the attribute `fuel_type`.

5. Write a Python program where different classes (`Car`, `Bike`, `Truck`) have the same method `start_engine()`, demonstrating polymorphism.

6. Write a Python program that uses operator overloading to add two `Vector` objects using the `+` operator.

7. Write a Python program demonstrating encapsulation using a class `BankAccount` with private attributes `__balance` and methods to deposit and withdraw money.

8. Create a class `Laptop` with private attributes for `price` and `model`, and public methods to access and modify them safely.

9. Create an abstract class `Shape` with an abstract method `calculate_area()`, then implement `Circle` and `Square` classes.

10. Write a Python program that demonstrates abstraction using an interface (abstract class) for different payment methods like `CreditCard`, `PayPal`, and `Bitcoin`.

1. Classes and Objects

Concept:

- **Class**: Blueprint for creating objects.
- **Object**: Instance of a class.

🧪 **Program: Car Class**

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

# Creating an object
my_car = Car("Toyota", "Camry", 2022)

# Display attributes
print("Brand:", my_car.brand)
print("Model:", my_car.model)
print("Year:", my_car.year)
```

🔍 **Explanation**:

- `__init__` is the constructor that initializes object attributes.
- `my_car` is an instance of the `Car` class.

---

2. Class with Method

Concept:

- Methods define behavior for the object.

🧪 **Program: Student Class**

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

    def display(self):
        print(f"Name: {self.name}, Marks: {self.marks}")
```

```
# Object creation
s1 = Student("Alice", 89)
s1.display()
```

🔍 **Explanation**:

- `display()` is a method to print student details.

---

3. Inheritance

Concept:

- Child class inherits properties/methods from parent class.

  🧪 **Program: Animal → Dog/Cat**

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

  class Dog(Animal):
      def speak(self):
          print("Dog barks")

  class Cat(Animal):
      def speak(self):
          print("Cat meows")

  # Object creation
  d = Dog()
  c = Cat()
  d.speak()
  c.speak()
  ```

🔍 **Explanation**:

- `Dog` and `Cat` override the `speak()` method.

---

4. Inheritance with Additional Attributes

🧪 **Program: Vehicle → Car**

```
class Vehicle:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

class Car(Vehicle):
    def __init__(self, brand, speed, fuel_type):
        super().__init__(brand, speed)
        self.fuel_type = fuel_type

# Object creation
c = Car("Honda", 120, "Petrol")
print(c.brand, c.speed, c.fuel_type)
```

🔍 **Explanation**:

- `super().__init__()` is used to call the parent class constructor.

---

5. Polymorphism

Concept:

- Same method name behaves differently depending on the class.

🧪 **Program: Vehicle Polymorphism**

```
class Car:
    def start_engine(self):
        print("Car engine started")

class Bike:
    def start_engine(self):
        print("Bike engine started")

class Truck:
    def start_engine(self):
        print("Truck engine started")
```

```
# Polymorphic behavior
for vehicle in (Car(), Bike(), Truck()):
    vehicle.start_engine()
```

🔍 **Explanation**:

- `start_engine()` is implemented differently in each class.

---

6. Operator Overloading

Concept:

- Customize behavior of operators for user-defined classes.

  🧪 **Program: Vector Addition**

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

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

      def __str__(self):
          return f"({self.x}, {self.y})"

  v1 = Vector(2, 3)
  v2 = Vector(4, 5)
  print(v1 + v2)
  ```

🔍 **Explanation**:

- `__add__` allows use of `+` with Vector objects.

---

7. Encapsulation

Concept:

- Hiding internal state using private attributes (`__var`).

  🧪 **Program: BankAccount**

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

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance

acc = BankAccount()
acc.deposit(1000)
acc.withdraw(500)
print("Balance:", acc.get_balance())
```

🔍 **Explanation**:

- Direct access to `__balance` is not allowed; only controlled via methods.

---

8. Encapsulation with Getter and Setter

🧪 **Program: Laptop**

```
class Laptop:
    def __init__(self, model, price):
        self.__model = model
        self.__price = price

    def get_model(self):
        return self.__model

    def set_model(self, model):
        self.__model = model

    def get_price(self):
        return self.__price

    def set_price(self, price):
        if price > 0:
            self.__price = price

l = Laptop("Dell", 50000)
print("Model:", l.get_model())
l.set_price(55000)
print("Updated Price:", l.get_price())
```

🔍 **Explanation**:

- Getter/setter methods control access and modification.

---

9. Abstraction

Concept:

- Abstract class has unimplemented methods. Forces child class to implement them.

  🧪 **Program: Shape Abstraction**

```
from abc import ABC, abstractmethod

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

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

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

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

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

c = Circle(5)
s = Square(4)
print("Circle area:", c.calculate_area())
print("Square area:", s.calculate_area())
```

🔍 **Explanation**:

- `Shape` is an abstract class; it cannot be instantiated.
- Subclasses must implement `calculate_area()`.

---

10. Abstraction: Payment Interface

🧪 Program: Payment Methods

```
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class CreditCard(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card")

class PayPal(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using PayPal")

class Bitcoin(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Bitcoin")

# Usage
for method in [CreditCard(), PayPal(), Bitcoin()]:
    method.pay(1000)
```

🔍 **Explanation**:

- Each payment method implements the `pay()` method differently using abstraction.

📌 Summary of Concepts

Concept	Purpose
Class	Template for creating objects
Object	Instance of a class
Inheritance	Reuse code from a base class
Polymorphism	One interface, many forms
Encapsulation	Data hiding using private attributes
Abstraction	Define abstract methods for flexibility