**Q1**

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of “objects,” which can contain data and code to manipulate that data. This approach helps in organizing software design by modeling real-world entities as objects.

The four main pillars of OOP are:

**1.Abstraction**:

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. It simplifies the interface for the user and reduces complexity by allowing the programmer to focus on interactions at a higher level without worrying about the underlying details.

**2.Encapsulation**:

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object’s components. This principle helps in protecting the internal state of an object from unintended interference and misuse, promoting modularity and maintainability.

**3.Inheritance**:

Inheritance allows a new class (child class) to inherit properties and behaviors (methods) from an existing class (parent class). It promotes code reusability and establishes a natural hierarchy, making it easier to create and maintain an application by building on existing, tested code.

**4.Polymorphism**:

Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. The most common use of polymorphism is when a parent class reference is used to refer to a child class object. It enables a single interface to represent different underlying forms (data types), enhancing flexibility and integration of code.

These pillars collectively contribute to better software design by;

Promoting code reusability

Scalability

maintainability, making it easier to manage and extend complex systems.

**Q2**

In Python, a constructor is a special method used to initialize the attributes of an object when it is created. The constructor method in Python is called \_\_init\_\_. This method is automatically invoked when a new instance of a class is created, allowing you to set initial values for the object’s attributes.

Purpose of the \_\_init\_\_ Method

The primary purpose of the \_\_init\_\_ method is to ensure that the object is properly initialized with the necessary attributes. It helps in setting up the initial state of the object, making it ready for use.

How the \_\_init\_\_ Method Works

The \_\_init\_\_ method typically takes at least one parameter, self, which refers to the instance being created. You can also pass additional parameters to initialize the object’s attributes.

Example

Here’s a simple example to illustrate how the \_\_init\_\_ method is used:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def display\_info(self):

print(f"Name: {self.name}, Age: {self.age}"

person1 = Person("Jordan", 30)

person1.display\_info()

In this example:

* The Person class has an \_\_init\_\_ method that takes name and age as parameters.
* When person1 is created, the \_\_init\_\_ method is called with “Alice” and 30 as arguments, initializing the name and age attributes of the person1 object.
* The display\_info method is then used to print the initialized attributes.

This approach ensures that every instance of the Person class is created with a specific name and age, providing a consistent and predictable way to initialize objects.

**Q3**

Class Variables and Instance Variables

Classes have two types of variables:

Class Variables

- Shared among all instances of a class

- Defined inside the class definition, but outside any method

- Belong to the class itself, not to any instance

- Changed in one place, affects all instances

Instance Variables

- Unique to each object (instance) of a class

- Defined inside methods (e.g., \_\_init\_\_)

- Belong to a specific instance, not to the class

- Changes do not affect other instances

Example

class Car:

# Class variable

wheels = 4

def \_\_init\_\_(self, color, model):

# Instance variables

self.color = color

self.model = model

# Create two instances of Car

car1 = Car("Red", "Toyota")

car2 = Car("Blue", "Honda")

# Access class variable

print(Car.wheels)

print(car1.wheels)

print(car2.wheels)

# Change class variable

Car.wheels = 6

print(Car.wheels) # Output: 6

print(car1.wheels) # Output: 6

print(car2.wheels) # Output: 6

# Access instance variables

print(car1.color) # Output: Red

print(car1.model) # Output: Toyota

print(car2.color) # Output: Blue

print(car2.model) # Output: Honda

# Change instance variable

car1.color = "Green"

print(car1.color) # Output: Green

print(car2.color) # Output:

Blue (unchanged)

- Class variables are shared among all instances.

- Instance variables are unique to each object.

- Changing a class variable affects all instances.

- Changing an instance variable only affects that specific object.

**Q4**

**Instance Methods**

Instance methods are the most common type of methods in Python classes. They are called on an instance of the class and can access and modify the instance’s attributes.

**Example:**

class MyClass:

def \_\_init\_\_(self, value):

self.value = value

def instance\_method(self):

return f'Instance method called, value is {self.value}'

obj = MyClass(10)

print(obj.instance\_method())

**Class Methods**

Class methods are methods that are bound to the class and not the instance of the class. They can modify class state that applies across all instances of the class. They are defined using the @classmethod decorator and take cls as the first parameter.

**Example:**

class MyClass:

class\_variable = 0

@classmethod

def class\_method(cls, x):

cls.class\_variable += x

return cls.class\_variable

print(MyClass.class\_method(3)) # Output: 3

print(MyClass.class\_method(7)) # Output: 10

**Static Methods**

Static methods are methods that do not modify class or instance state. They are defined using the @staticmethod decorator and do not take self or cls as the first parameter. They behave like regular functions but belong to the class’s namespace.

**Example:**

**Python**

class MathOperations:

@staticmethod

def add(x, y):

return x + y

@staticmethod

def subtract(x, y):

return x - y

print(MathOperations.add(5, 3)) # Output: 8

print(MathOperations.subtract(10, 4)) # Output: 6

**Differences**

* **Instance Methods**: Operate on an instance of the class and can access and modify instance attributes.
* **Class Methods**: Operate on the class itself and can modify class attributes. They are called on the class, not on instances.
* **Static Methods**: Do not modify class or instance state. They are utility functions that belong to the class’s namespace.

**Q5**

**Scenario**

In banking system where each customer has a bank account. The bank wants to keep track of the total number of accounts created and the interest rate applied to all accounts. Here, using class variables is more suitable.

**Reasons**

1. **Tracking Total Accounts**: The total number of accounts is a property that belongs to the class as a whole, not to any individual account. This count should be shared across all instances of the class.
2. **Interest Rate**: The interest rate is a policy set by the bank and applies uniformly to all accounts. It should be consistent across all instances and can be changed centrally.

**Example**

class BankAccount:

total\_accounts = 0 # Class variable to track total accounts

interest\_rate = 0.05 # Class variable for interest rate

def \_\_init\_\_(self, owner, balance):

self.owner = owner

self.balance = balance

BankAccount.total\_accounts += 1 # Increment total accounts when a new account is created

def apply\_interest(self):

self.balance += self.balance \* BankAccount.interest\_rate

# Creating bank accounts

account1 = BankAccount("Joan", 10000)

account2 = BankAccount("Frank", 15000)

# Accessing class variables

print(BankAccount.total\_accounts) # Output: 2

print(BankAccount.interest\_rate) # Output: 0.05

# Applying interest to accounts

account1.apply\_interest()

account2.apply\_interest()

print(account1.balance) # Output: 1050.0

print(account2.balance) # Output: 1575.0

**Explanation**

* **Class Variable**total\_accounts: This variable keeps track of the total number of BankAccount instances created. It is incremented in the \_\_init\_\_ method, ensuring that every time a new account is created, the count is updated.
* **Class Variable**interest\_rate: This variable holds the interest rate applicable to all accounts. It can be accessed and modified through the class, ensuring consistency across all instances.

Using class variables in this scenario ensures that the total number of accounts and the interest rate are managed centrally and consistently, reflecting changes across all instances of the class. This approach avoids redundancy and ensures that these shared properties are maintained accurately.