QUESTION 1

Object-oriented Programming is a programming paradigm that organizes code using objects and classes. It helps in organizing large and complex programs in more modular and reusable way.

The four main pillars of Object-oriented Programming are as followed

**Encapsulation**:

It refers to bundling data (attributes) and methods (functions) that operate on the data into a single unit called a class. It also allows restricting direct access to some of an object's components, which is known as data hiding.

Contribution to software design

Encapsulation provides a clear interface for interacting with objects while protecting the integrity of data by **controlling access to it.**

Inheritance:

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reuse.

Contribution to software design:

Inheritance reduces redundancy by allowing shared code between related classes. It also supports a hierarchical relationship, making it easier to maintain and extend the code.

Polymorphism:

Polymorphism allows methods to take many forms. A method can have the same name in different classes, and the method behavior can be overridden to suit the specific class.

Contribution to software design:

Polymorphism promotes flexibility and the ability to use objects of different types interchangeably, which makes code more general and easier to extend.

Abstraction:

Abstraction focuses on hiding the complexity of the system and exposing only the necessary parts. It defines the essential properties and behaviors of an object and hides the implementation details.

Contribution to software design:

Abstraction simplifies complex systems by providing a simpler interface, which makes the code more maintainable and easier to understand.

QUESTION 2

Below is the purpose of a constructor in python, how the \_init\_ method is used to initialize an object’s attributes with example.

In Python, a **constructor** is a special method called \_\_init\_\_ that is automatically called when an object is created from a class. Its primary purpose is to initialize the object’s attributes and set their initial values when the object is instantiated.

Example

python

class Car:

def \_\_init\_\_(self, make, model, year)

self. make = make

self. model = model

self. year = year

def car\_info(self):

return f"{self. year} {self.make} {self.model}"

my\_car = Car("Toyota", "Camry", 2020)

print(my\_car.car\_info()) # Output: 2020 Toyota Camry

The \_\_init\_\_ method takes three arguments (make, model, and year).

These values are assigned to the instance variables (self. make, self. model, self. year)

QUESTION 3

Below is the difference between class variables and instance variables with their examples that illustrate how class variables are shared among all instance of a class.

Instance Variables: Are variables defined within methods like \_\_init\_\_ and are unique to each object. Every instance of a class has its own copy of the instance variables.

Class Variables: Are variables that are shared across all instances of the class. They are defined outside any instance methods and are shared among all objects of that class.

Example:

class Employee:

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

self.name = name

self.age = age

emp1 = Employee("Alice", 30)

emp2 = Employee("Bob", 25)

print(emp1.name) # Output: Alice

print(emp2.name) # Output: Bob

print(emp1.company) # Output: TechCorp

print(emp2.company) # Output: TechCorp

Employee.company = "NewTechCorp"

print(emp1.company) # Output: NewTechCorp

print(emp2.company) # Output: NewTechCorp

QUESTION 4

Instance Methods:

These operate on an instance of the class and require a reference to the instance (self) to access instance-specific data.

Class Methods:

These operate on the class itself rather than on instances. They require a reference to the class (cls) and are marked with the @classmethod decorator.

Static Methods:

These don’t operate on the class or instance; they are simply methods that belong to the class. They do not take self or cls as the first argument and are marked with the @staticmethod decorator.

Example:

class Calculator:

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

self.num = num # Instance variable

# Instance method

def multiply(self):

return self.num \* Calculator.factor

# Class method

@classmethod

def update\_factor(cls, new\_factor):

cls.factor = new\_factor

# Static method

@staticmethod

def add(x, y):

return x + y

# Using the methods

calc = Calculator(5)

print(calc.multiply()) # Instance method: Output: 50

Calculator.update\_factor(20) # Class method: modifies class variable

print(calc.multiply()) # Output: 100

print(Calculator.add(10, 15)) # Static method: Output: 25

Differences:

Instance methods need access to individual instance data (self).

Class methods work on the class level (cls) and can modify class variables.

Static methods don’t depend on class or instance-specific data and behave like regular functions but belong to the class.

QUESTION 5

Below is areal-world scenario where using class variables would be more appropriate than instance variables.

A real-world scenario where class variables are more appropriate is when you need to maintain data that is shared among all instances of a class. For example, imagine a library system where every book shares the same lending policy:

Scenario: Library System

Class: Book

Class Variable: lending\_period (shared by all books, say 14 days)

Instance Variables: title, author (specific to each book)

Example

class Book:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

# Accessing the class variable

print(Book.lending\_period) # Output: 14

book1 = Book("1984", "George Orwell")

book2 = Book("To Kill a Mockingbird", "Harper Lee")

# Both books have the same lending period

print(book1.lending\_period) # Output: 14

print(book2.lending\_period) # Output: 14

# Changing the lending period for all books

Book.lending\_period = 21

print(book1.lending\_period) # Output: 21

print(book2.lending\_period) # Output: 21

In this scenario, it makes sense to use a class variable for the lending\_period since it applies to all books. Changes to this policy affect all instances, whereas book-specific details like title and author are stored as instance variables.