Q1: Concept of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain both data (attributes) and functions (methods) that operate on the data. OOP provides a way to structure software that models real-world entities and their interactions.

The four main pillars of OOP are:

1. **Encapsulation**: Encapsulation is the process of bundling data (attributes) and methods (functions) that operate on the data within one unit, typically a class. It restricts direct access to some of the object's components, ensuring that data is hidden from outside interference and misuse.
   - **Benefit**: It improves data security and hides implementation details from the outside world.
2. **Abstraction**: Abstraction focuses on exposing only the essential features of an object while hiding unnecessary details. This allows for the creation of simple interfaces while managing complexity.
   - **Benefit**: It simplifies complex systems by allowing the developer to focus on interactions and essential properties.
3. **Inheritance**: Inheritance allows one class (child class) to inherit the properties and behaviours (methods) of another class (parent class). It promotes code reusability.
   - **Benefit**: It reduces redundancy, allowing for easier updates and maintenance since changes made to a parent class propagate to child classes.
4. **Polymorphism**: Polymorphism means "many forms" and allows objects of different classes to be treated as objects of a common parent class. It also enables methods to perform differently based on the object calling them.
   - **Benefit**: It allows for flexible and dynamic method invocation, enhancing extensibility and scalability.

Q2: Purpose of a Constructor in Python

In Python, a constructor is a special method used to initialize the attributes of an object when it is created. The __init__ method is the constructor in Python, which gets called automatically when an object of the class is instantiated.

The __init__ method allows you to set up or initialize the state of the object by assigning values to its attributes.

Example:

```
class Person:
    def __init__(self, name, age):
        # Initializing instance attributes
        self.name = name
        self.age = age

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


# Creating an object of the Person class
person1 = Person("John", 30)
person1.display_info()  # Output: Name: John, Age: 30
```

In this example, when person1 is created, the __init__ method initializes its attributes name and age.

Q3: Class Variables vs. Instance Variables

- **Instance Variables**: These are variables that are unique to each instance of a class. Each object has its own copy of instance variables.
- **Class Variables**: These are shared among all instances of a class. There is only one copy of a class variable, and any change made to it affects all instances.

Example:

python

```
class Car:
    # Class variable (shared among all instances)
    wheels = 4

    def __init__(self, color):
        # Instance variable (unique to each instance)
        self.color = color

# Creating two objects of the Car class
car1 = Car("Red")
car2 = Car("Blue")

print(car1.wheels)  # Output: 4 (class variable)
print(car2.wheels)  # Output: 4 (class variable)

# Changing the class variable
```

```
Car.wheels = 6


print(car1.wheels)  # Output: 6 (reflects the change for all instances)

print(car2.wheels)  # Output: 6
```

In this example, the wheels class variable is shared by both car1 and car2, while the color instance variable is unique to each object.

Q4: Class Method and Static Method

- **Class Method**: A class method is a method that is bound to the class and not the object of the class. It takes cls as its first parameter, representing the class itself.
  - **Use Case**: Class methods are used to access or modify class-level data.
- **Static Method**: A static method is a method that does not take self or cls as its first parameter. It behaves like a regular function but belongs to the class's namespace.
  - **Use Case**: Static methods are used when functionality is related to the class but doesn't need access to class or instance-level data.

Example:

python

```
    @classmethod

    def square(cls, x):

        return x * x



    @staticmethod
    def add(a, b):
        return a + b


# Using class method
print(MathOperations.square(5))  # Output: 25


# Using static method
print(MathOperations.add(3, 7))  # Output: 10
```

Here, square is a class method that takes a parameter x, and add is a static method that directly performs addition.

Q5: Real-World Scenario for Class Variables

Scenario: Imagine building a school management system where every student has a unique name and grade, but all students share a common school name.

In this case, the school name can be a class variable since it is the same for all students, whereas the student name and grade should be instance variables, unique to each object.

Example

```
class Student:
    # Class variable (shared by all instances)
    school_name = "ABC High School"


    def __init__(self, name, grade):
        # Instance variables (unique to each instance)
        self.name = name

        self.grade = grade


# Creating instances of the Student class

student1 = Student("Alice", "A")

student2 = Student("Bob", "B")


# Accessing class variable

print(student1.school_name)  # Output: ABC High School

print(student2.school_name)  # Output: ABC High School
```

In this scenario, the school_name is shared among all instances, while each student has a unique name and grade.