

ques1. What are the five key concepts of Object-Oriented Programming (OOP)? The five key concepts of Object-Oriented Programming (OOP) are:

1. **Encapsulation:** This principle involves bundling the data (attributes) and methods (functions) that operate on that data into a single unit, or class. It also restricts direct access to some of the object's components, which helps protect the integrity of the data.
2. **Abstraction:** Abstraction focuses on hiding complex implementation details and exposing only the necessary features of an object. It allows programmers to interact with objects at a high level without needing to understand their inner workings.
3. **Inheritance:** Inheritance enables a new class (subclass or derived class) to inherit attributes and methods from an existing class (superclass or base class). This promotes code reusability and establishes a natural hierarchy between classes.
4. **Polymorphism:** Polymorphism allows methods to do different things based on the object that it is acting upon. This can be achieved through method overriding (where a subclass provides a specific implementation of a method defined in its superclass) or method overloading (where multiple methods have the same name but different parameters).
5. **Composition:** Composition is the concept of building complex types by combining objects (instances of classes) rather than inheriting from a base class. This approach allows for greater flexibility and promotes a "has-a" relationship, where one object contains or is composed of other objects.

These concepts work together to enable developers to create modular, reusable, and maintainable code.

ques 2. Write a Python class for a Car with attributes for make, model, and year. Include a method to display the car's information.

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

    def display_info(self):
        print(f"Car Make: {self.make}")
        print(f"Car Model: {self.model}")
        print(f"Car Year: {self.year}")

# Example usage:
my_car = Car("Toyota", "Camry", 2021)
my_car.display_info()
```

```
Car Make: Toyota
Car Model: Camry
Car Year: 2021
```

ques 3. Explain the difference between instance methods and class methods. Provide an example of each.

In Python, instance methods and class methods serve different purposes and are called in different ways.

Instance Methods Definition: Instance methods are functions that operate on an instance of a class. They take the instance (self) as their first parameter, allowing them to access instance attributes and other instance methods.

Usage: They are used to manipulate or retrieve the data specific to an object.

Class Methods Definition: Class methods are functions that operate on the class itself rather than on instances of the class. They take the class (cls) as their first parameter and are defined using the @classmethod decorator.

Usage: They are used for operations that pertain to the class as a whole, such as factory methods or keeping track of class-level data.

```
#example of instance method
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return f"{self.name} says Woof!"

# Example usage:
my_dog = Dog("Buddy")
print(my_dog.bark()) # Output: Buddy says Woof!

#class methods
```

```

class Dog:
    count = 0 # Class variable to keep track of the number of dogs

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

    @classmethod
    def get_dog_count(cls):
        return cls.count

# Example usage:
my_dog1 = Dog("Buddy")
my_dog2 = Dog("Max")
print(Dog.get_dog_count()) # Output: 2

```

```

→ Buddy says Woof!
2

```

Instance Methods: Operate on individual instances and can access instance-specific data.

Class Methods: Operate on the class level, and are used for class-related operations or to modify class-level data.

ques 4. How does Python implement method overloading? Give an example method overloading is the practice of invoking the same method more than once with different parameters. method overloading is not supported by python. even if you overload the method, python only takes into account the most recent definition. if you overload in python, a typeerror will be raised.

```

def mul(x,y):
    z=x*y
    print (z)
def mul(p,q,r):
    a=p*q*r
    print("output:",a)
    print("output:",z)
    mul(5,6,3)

#Example of Method Overloading using Default Arguments
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c

# Example usage:
math_op = MathOperations()

print(math_op.add(5))          # Output: 5 (5 + 0 + 0)
print(math_op.add(5, 3))      # Output: 8 (5 + 3 + 0)
print(math_op.add(5, 3, 2))   # Output: 10 (5 + 3 + 2)

```

```

→ 5
8
10

```

```

#Example of Method Overloading using Variable-Length Arguments
class MathOperations:
    def add(self, *args):
        return sum(args)

# Example usage:
math_op = MathOperations()

print(math_op.add(5))          # Output: 5
print(math_op.add(5, 3))      # Output: 8
print(math_op.add(5, 3, 2, 4, 1)) # Output: 15

```

```

→ 5
8
15

```

ques 5.What are the three types of access modifiers in Python? How are they denoted? Access Modifier in python: A class's data members and methods can be made private or protected in order to achieve encapsulation. Direct access modifiers like public, private, and protected don't exist in Python, though. Single and double underscores can be used to accomplish this.

Modifiers for access control restrict use of a class's variables and methods. Private, public, and protected are the three different access modifier types that Python offers.

Public Member: from outside of class, anywhere accessible.

Private Member: Within the class, accessible.

Protected Member: Within the class and its subclasses, accessible.

#Public Member: Both inside and outside of a class, public data members are accessible.
#By default, the class member variables are all public.

```
class Student:
    def __init__(self, name, degree):
        self.name = name # Public attribute or data member
        self.degree = degree # Public attribute
    def show(self):
        print("name:",self.name,"degree:",self.degree)
stud=Student("ram", "b.tech")
stud.show()
print(stud.name)
print(stud.degree)
```

```
name: ram degree: b.tech
ram
b.tech
```

Private Member: By designating class variables as private, we may protect them. Add two underscores as a prefix to the beginning of a variable name to define it as a private variable.

Private members can only be accessed within the class; they are not directly available from class objects.

We can access private members from outside of a class using the following two approaches.

- Create a public method to access private members of a class.
- By using name mangling

```
class Student:
    def __init__(self, name, degree):
        self.name = name # Public attribute or data member
        #private
        self.__degree = degree
stud=Student("ram", "b.tech")
print("degree:", stud._Student__degree)
```

```
degree: b.tech
```

Protected Member: Within the class and to its sub-classes, protected members can be accessed. Add a single underscore (_) before the member name to define it as a protected member.

When you implement inheritance and wish to restrict access to data members to just child classes, you use protected data members

```
class College:
    def __init__(self):
        #protected member
        self._college_name="mdu"
    class col:
        def __init__(self,name):
            self.name=name
            College.__init__(self)
        def show(self):
            print("name:",self.name)
            print("college name:",self._college_name)
stud=College.col("ram")
stud.show()
```

```
name: ram
college name: mdu
```

ques 6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.

Single Inheritance A derived class inherits from a single base class.

Multiple Inheritance A derived class inherits from multiple base classes. This can lead to the diamond problem, which Python handles using the Method Resolution Order (MRO).

Multilevel Inheritance A derived class inherits from a base class, which in turn inherits from another class.

Hierarchical Inheritance Multiple derived classes inherit from a single base class.

Hybrid Inheritance A combination of two or more types of inheritance. This often involves multiple inheritance along with other types.

Single Inheritance: One base class.

Multiple Inheritance: Multiple base classes. Multilevel Inheritance: A chain of inheritance.

Hierarchical Inheritance: Multiple classes from one base class.

Hybrid Inheritance: A mix of the above types.

```
#multiple inheritance
class Canine:
    def bark(self):
        return "Barking"

class Pet:
    def play(self):
        return "Playing"

class Dog(Canine, Pet):
    def wag_tail(self):
        return "Wagging tail"

# Usage
dog = Dog()
print(dog.bark())
print(dog.play())
print(dog.wag_tail())
```

```
Barking
Playing
Wagging tail
```

ques 7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?

The Method Resolution Order (MRO) in Python defines the order in which base classes are searched when calling a method on an instance. This order is particularly important in cases of multiple inheritance, where a class inherits from more than one base class. The MRO ensures that the methods are called in a predictable and consistent order, preventing ambiguity.

How MRO Works

Python uses the C3 linearization algorithm (or C3 superclass linearization) to determine the MRO. This algorithm combines the order of base classes and resolves conflicts in a way that maintains the order of inheritance.

MRO defines the order of method resolution for classes in Python, especially in multiple inheritance scenarios.

You can access the MRO using the `mro()` method or the `mro` attribute of a class. This will return a list or a tuple of classes in the order they will be searched for methods.

```
#Retrieving the MRO Programmatically
#You can retrieve the MRO of a class using the mro() method or the __mro__ attribute.

class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

```
# Retrieve MRO
print(D.mro())
print(D.__mro__)
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

ques 8. Create an abstract base class `Shape` with an abstract method `area()`. Then create two subclasses `Circle` and `Rectangle` that implement the `area()` method.

```
from abc import ABC, abstractmethod
import math

# Abstract Base Class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

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

# Subclass for Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

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

print(f"Circle Area: {circle.area():.2f}")
print(f"Rectangle Area: {rectangle.area()}")
```

```
Circle Area: 78.54
Rectangle Area: 24
```

Abstract Base Class (`Shape`):

Inherits from `ABC`.

Contains an abstract method `area()` which must be implemented by any subclass.

Subclass (`Circle`):

Implements the `area()` method to calculate the area of a circle using the formula πradius^2

Subclass (`Rectangle`):

Implements the `area()` method to calculate the area of a rectangle using the formula $\text{width} \times \text{height}$

ques 9. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas.

Polymorphism allows functions to operate on objects of different classes as long as they implement the same method. In this case, we can create a function that takes any object of type `Shape` and prints its area. Here's how you can implement this:

```
from abc import ABC, abstractmethod
import math

# Abstract Base Class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
# Subclass for Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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


# Subclass for Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

# Function to print area of any Shape
def print_area(shape: Shape):
    print(f"The area of the shape is: {shape.area():.2f}")

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

print_area(circle)
print_area(rectangle)
```

 The area of the shape is: 78.54
The area of the shape is: 24.00

Abstract Base Class (Shape):

Defines the abstract method `area()` that must be implemented by subclasses.

Subclasses (Circle and Rectangle):

Both implement the `area()` method, allowing them to be treated as Shape objects.

Polymorphic Function (`print_area`):

This function accepts any object of type Shape.

Calls the `area()` method of the passed object, demonstrating polymorphism.

ques 11.. Write a class that overrides the `__str__` and `__add__` magic methods. What will these methods allow you to do?

What These Methods Allow You To Do

String Representation:

The **str** method allows instances of the class to be represented as a meaningful string when printed or converted to a string. This improves the readability and usability of your class.

Addition of Instances:

The **add** method allows instances of the class to be combined using the `+` operator, providing a natural syntax for operations between objects of the class. This can be particularly useful in mathematical or computational classes, like Vector.

ques 12.. Create a decorator that measures and prints the execution time of a function

```
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time() # Start time
        result = func(*args, **kwargs) # Call the original function
        end_time = time.time() # End time
        execution_time = end_time - start_time # Calculate execution time
        print(f"Execution time of '{func.__name__}': {execution_time:.4f} seconds")
        return result # Return the result of the function
    return wrapper

@timer_decorator
def slow_function():
```

```

time.sleep(2) # Simulating a slow operation
print("Function completed!")

@timer_decorator
def quick_function():
    print("Quick function executed!")

# Calling the functions
slow_function() # This will take around 2 seconds
quick_function() # This will execute almost instantly

Function completed!
Execution time of 'slow_function': 2.0023 seconds
Quick function executed!
Execution time of 'quick_function': 0.0000 seconds

```

Decorator Definition (timer_decorator):

Takes a function func as an argument.

Defines a nested function wrapper that will wrap the original function.

Records the start time before calling the original function.

Records the end time after the function call.

Calculates the execution time and prints it.

Returns the result of the original function.

Using the Decorator:

The @timer_decorator syntax is used to apply the decorator to slow_function and quick_function.

When these functions are called, they will execute with timing information printed.

ques 13.Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?

The Diamond Problem is a common issue that arises in multiple inheritance scenarios, where a class inherits from two or more classes that have a common ancestor. This can lead to ambiguity regarding which parent's method or attribute should be inherited by the child class.

```

  A
 / \
B   C
 \ /
  D

```

Class D inherits from both B and C, which both inherit from A.

If both B and C override a method from A, and D calls this method, it becomes unclear which version of the method should be executed—B's or C's.

Python's Resolution of the Diamond Problem:

Python resolves the Diamond Problem using the Method Resolution Order (MRO), which determines the order in which base classes are searched when calling a method. Python uses the C3 linearization algorithm to create a linearization of the classes involved, ensuring a consistent and predictable order.

```

class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(A):
    def greet(self):
        return "Hello from C"

class D(B, C):
    pass

```

```
d = D()
print(d.greet())
```

```
→ Hello from B
```

Explanation of the Output

Class Definitions:

A defines a greet() method. B and C both override the greet() method.

Class D:

Inherits from both B and C.

Calling greet() on an Instance of D:

The MRO for D can be determined using D.mro() or D.mro. In this case, the order is [D, B, C, A]. When d.greet() is called, Python looks for the greet() method in D, then B, and finds it in B first. Hence, it returns "Hello from B".

ques 15. Implement a static method in a class that checks if a given year is a leap year.

```
class YearChecker:
    @staticmethod
    def is_leap_year(year):
        """Return True if the specified year is a leap year, otherwise False."""
        # A year is a leap year if:
        # It is divisible by 4
        # It is not divisible by 100, unless it is also divisible by 400
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
            return True
        return False

if __name__ == "__main__":
    years_to_check = [2000, 2001, 2004, 1900, 2020, 2023]
    for year in years_to_check:
        if YearChecker.is_leap_year(year):
            print(f"{year} is a leap year.")
        else:
            print(f"{year} is not a leap year.")
```

```
→ 2000 is a leap year.
   2001 is not a leap year.
   2004 is a leap year.
   1900 is not a leap year.
   2020 is a leap year.
   2023 is not a leap year.
```