#1
Ans:

1Definition and Usage:

- Class Method: A class method is a method that is bound to the class and not the instance of the class. It can access and modify class-level variables but not instance-level variables. Class methods are defined using the **@classmethod** decorator.

  class MyClass:

  @classmethod

  def class_method(cls, arg1, arg2):

  # Code here

  Static Method: A static method is a method that is not bound to the class or the instance. It cannot access or modify class or instance variables. Static methods are defined using the **@staticmethod** decorator.

  class MyClass:

  @staticmethod

  def static_method(arg1, arg2):

  # Code here

  2 Accessing Variables:

- Class Method: Class methods have access to class-level variables through the **cls** parameter passed to the method. They can modify class-level variables and call other class methods.

  class MyClass:

  count = 0

  @classmethod

  def increment_count(cls):

  cls.count += 1

```
MyClass.increment_count()  # Calling the class method

print(MyClass.count)  # Output: 1
```

Static Method: Static methods cannot access class or instance variables directly. They are primarily used for utility functions that do not depend on the state of the class or instance.

```
class MathUtils:

    @staticmethod

    def add_numbers(num1, num2):

        return num1 + num2


result = MathUtils.add_numbers(3, 5)  # Calling the static method

print(result)  # Output: 8
```

3 Invocation:

- Class Method: Class methods are typically called using the class name, and the first parameter is automatically passed as the class itself.

```
MyClass.class_method(arg1, arg2)  # Calling the class method
```

Static Method: Static methods can be called using either the class name or an instance of the class. The method does not have access to the instance or class.

```
MyClass.static_method(arg1, arg2)  # Calling the static method

my_instance.static_method(arg1, arg2)  # Calling the static method using an instance
```

These are the main differences between class methods and static methods in Python. Class methods are bound to the class and have access to class-level variables, while static methods are not bound to the class or instance and cannot access class or instance variables.

#2
Ans:

1Method Overriding: Method overriding occurs when a subclass provides a different implementation of a method that is already defined in its superclass. The method in the

subclass has the same name and parameters as the method in the superclass but provides a different implementation. This allows different objects to exhibit different behaviors for the same method call.

```
class Animal:

    def sound(self):

        print("Animal makes a sound.")


class Dog(Animal):

    def sound(self):

        print("Dog barks.")


class Cat(Animal):

    def sound(self):

        print("Cat meows.")


# Polymorphism in action

dog = Dog()

cat = Cat()


dog.sound()  # Output: Dog barks.

cat.sound()  # Output: Cat meows.
```

2 Method Overloading (Emulated): Method overloading typically involves defining multiple methods with the same name but different parameters in a class. However, Python does not natively support method overloading based on different parameter signatures. However, polymorphism can be emulated by using default values or conditional statements inside a method to handle different parameter cases.

```
class MathUtils:
```

```
    def add(self, num1, num2=None):
    if num2 is None:
    return num1
    else:
    return num1 + num2


# Polymorphism in action
math = MathUtils()


result1 = math.add(5)        # Output: 5
result2 = math.add(2, 3)     # Output: 5
```

1. In the example above, the **add** method in the **MathUtils** class emulates method overloading. When called with only one argument, it returns the single argument itself. When called with two arguments, it performs the addition.

Polymorphism allows different objects to respond to the same method call in different ways, enhancing flexibility and modularity in object-oriented programming. It enables code reuse, as objects of different classes can be treated uniformly when they share a common interface or base class.

#3
Ans:

variables **a**, **b**, and **c**, along with the **sum()** and **factorial()** methods:

```
class MyClass:
    def __init__(self, a, b, c):
    self.a = a
    self.b = b
       self.c = c
```

```
    def sum(self):

    return self.a + self.b + self.c


    def factorial(self):

    result = 1

    for num in range(1, self.b + 1):

    result *= num

    return result
```

In the above code:

- The **__init__()** method is a constructor that initializes the instance variables **a**, **b**, and **c** with the provided values.
- The **sum()** method calculates and returns the sum of **a**, **b**, and **c** using the **+** operator.
- The **factorial()** method calculates and returns the factorial of **b** using a loop.

You can create an instance of the class and test these methods like this:

```
# Create an instance of MyClass

my_object = MyClass(2, 4, 6)


# Test the sum() method

print(my_object.sum())  # Output: 12


# Test the factorial() method

print(my_object.factorial())  # Output: 24
```

In the above code, we create an instance **my_object** of the **MyClass** class with values **2**, **4**, and **6** for **a**, **b**, and **c** respectively. Then we call the **sum()** and **factorial()** methods on **my_object** and print the results. The output will be **12** for the sum and **24** for the factorial.

#4
Ans:

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

        def get_brand(self):
        return self.brand

        def drive(self):
    print("Driving the vehicle.")


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

        def get_model(self):
        return self.model

        def drive(self):
    print(f"Driving the {self.brand} {self.model} car.")
```

```
class ElectricCar(Car):

    def __init__(self, brand, model, battery_capacity):

        super().__init__(brand, model)

        self.battery_capacity = battery_capacity


    def get_battery_capacity(self):

        return self.battery_capacity


    def drive(self):

        print(f"Driving the {self.brand} {self.model} electric car with {self.battery_capacity} kWh battery.")
```

In the above example, we have three classes: **Vehicle**, **Car**, and **ElectricCar**.

- The **Vehicle** class serves as the base class, which defines common properties and methods related to vehicles.
- The **Car** class is derived from **Vehicle** and adds additional properties and methods specific to cars.
- The **ElectricCar** class further inherits from **Car** and introduces new properties and methods specific to electric cars.

Through multilevel inheritance, the **ElectricCar** class inherits properties and methods from both the **Car** class and the **Vehicle** class, creating a chain of inheritance.

Let's create an instance of **ElectricCar** and observe the inherited functionality:

python
```
 my_car = ElectricCar("Tesla", "Model S", 100)


print(my_car.get_brand())          # Output: Tesla

print(my_car.get_model())          # Output: Model S

print(my_car.get_battery_capacity()) # Output: 100

my_car.drive()                     # Output: Driving the Tesla Model S electric car with 100 kWh battery.
```

In the above code, we create an instance of the **ElectricCar** class named **my_car**. We can use the inherited methods **get_brand()**, **get_model()**, and **get_battery_capacity()** to retrieve the respective information of the car. Additionally, when we call the **drive()** method, the overridden version defined in the **ElectricCar** class is executed, providing a specific implementation for electric cars.

Multilevel inheritance allows for the creation of a hierarchical structure of classes, where each derived class inherits the features of its immediate parent class. This enables code reuse, promotes modularity, and allows for specialization and customization of classes based on specific requirements.

#5
Ans:

Encapsulation and Information Hiding: Inner functions can be used to encapsulate functionality within a class, hiding it from the external scope. This promotes information hiding and helps maintain the principle of encapsulation in OOP.

```
class Calculator:

        def add(self, num1, num2):

        def validate_inputs():

            # Validation logic goes here

        pass



    validate_inputs()

        return num1 + num2
```

1. In the example above, the **validate_inputs()** function is an inner function within the **add()** method of the **Calculator** class. It encapsulates the input validation logic, ensuring that it is only accessible within the context of the **add()** method.
2. Code Reusability and Readability: Inner functions can improve code reusability and readability by providing a modular approach. They allow you to define helper functions that are specific to a class or a particular method, making the code more organized and easier to understand.

```
class Order:
```

```
    def calculate_total(self, items):

    def apply_discount(total):

    # Discount calculation logic goes here

    pass



    subtotal = sum(items)

    total = subtotal - apply_discount(subtotal)

    return total
```

1. In this example, the **apply_discount()** function is an inner function within the **calculate_total()** method of the **Order** class. It calculates the discount based on the subtotal of the items. By encapsulating the discount calculation logic in an inner function, it improves code readability and keeps the main logic of the **calculate_total()** method focused on the overall calculation.
2. Closure and State Preservation: Inner functions can access variables from their outer scope, even after the outer function has finished executing. This property, known as closure, can be useful for preserving state across multiple function calls.

```
def counter():

    count = 0



    def increment():

    nonlocal count

    count += 1

    return count



    return increment


counter1 = counter()

print(counter1())  # Output: 1
```

```
print(counter1())  # Output: 2
```

1. In this example, the **counter()** function returns an inner function **increment()**. The **count** variable is preserved in the closure of **increment()**, allowing it to maintain its state across multiple function calls. This can be handy in scenarios where you need to maintain and update a shared state within a class.

By using inner functions in Python OOP, you can achieve encapsulation, improve code organization and readability, promote information hiding, and leverage closure for state preservation. These advantages contribute to writing cleaner, more modular, and maintainable code.

#6
Ans:

```
# Read input values
N, M = map(int, input().split())
A = list(map(int, input().split()))

# Create a frequency dictionary
frequency = {}

# Count the occurrences of each number
for num in A:
    if num in frequency:
        frequency[num] += 1
    else:
        frequency[num] = 1

# Print the frequency of each number
for i in range(1, M+1):
    if i in frequency:
        print(frequency[i])
    else:
        print(0)
```

#7
Ans:

```
class Person:
    def __init__(self, name, age, height, weight) -> None:
        self.name = name
        self.age = age
        self.height = height
        self.weight = weight
```

```
    def __lt__(self, other):
        return self.age < other.age


class Cricketer(Person):
    def __init__(self, name, age, height, weight) -> None:
        super().__init__(name, age, height, weight)


Sakib = Cricketer('Sakib', 38, 68, 91)
Mushfiq = Cricketer('Mushfiq', 36, 55, 82)
Mustafiz = Cricketer('Mustafiz', 27, 69, 86)
Riyad = Cricketer('Riyad', 39, 72, 92)

youngest_player = min([Sakib,Mushfiq,Mustafiz,Riyad])
print(youngest_player.name)
```

we have overloaded the < operator in the Person class to compare two objects based on their age. We then created four objects of the Cricketer class and assigned them to variables. Finally we used the min() function to find the youngest player and printed his name.

The output of this code will be:

Mustafiz

So

Mustafiz is the youngest player among these four cricketers.