Answer Script

Question No. 01

Write down the differences between class method and static method of Python with proper examples.(at least 3) **10**

Answer No. 01

Class method	Static method
A class method is a method that is bound to the class and not the instance of the class.	A static method is a method that belongs to the class, but does not have access to either the class or instance attributes.
Class methods are used when a method needs to access or modify the class attributes, but not the instance attributes.	Static methods are used when a method does not depend on any instance-specific or class-specific data.
Class methods are defined using the @classmethod decorator, which automatically passes the class as the first argument	Static methods are defined using the @staticmethod decorator and do not receive any automatic arguments related to the class.

Example:
class Phone:
brand = "samsung"
 	
@classmethod
def get_brand(cls):
return cls.brand
 	
brand = Phone.get_brand()
print(brand)
	Example:
	class Phone:
	brand = "samsung"
	
	@staticmethod
	def make_sound():
	print("Making a sound!")
	
	Phone.make_sound()

Question No. 02

Explain with proper examples what is meant by polymorphism in Python.**15**

Answer No. 02

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables objects to exhibit different behaviors while being accessed through a common interface. In Python, polymorphism is achieved through method overriding.

Example:

```
class Animal: # Here 'Animal' Class Serve as a superclass
        def __init__(self, name) -> None:
        self.name = name

        def make_sound(self):
        print('Animal making some sound')
class Cat(Animal): # Subclass inherit from 'Animal'
        def __init__(self, name) -> None:
        super().__init__(name)

        def make_sound(self):
        print('meow meow')
class Dog(Animal): # Subclass inherit from 'Animal'
        def __init__(self, name) -> None:
        super().__init__(name)

        def make_sound(self):
        print('Bark Bark')
class Goat(Animal): # Subclass inherit from 'Animal'
        def __init__(self, name) -> None:
        super().__init__(name)

        def make_sound(self):
        print('beh beh beh')
```

# Here each subclass overrides the make_sound() method with its own implementation.

Question No. 03

Write a class with three instance variables a,b and c.
Now add the following two methods in that class

a) **sum()** to get the sum of a,b and c.
b) **factorial()** to get the factorial of b.
   15

Answer No. 03

```
class MyMath:
        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):
        fact = 1
        for num in range(1, self.b + 1):
        fact *= num
        return fact

my_solution = MyMath(2, 3, 4)
print(my_solution.sum())
print(my_solution.factorial())
```

Question No. 04

Explain with proper examples what is meant by multilevel inheritance in Python. **15**

Answer No. 04

Multilevel inheritance in Python refers to a scenario where a derived class (subclass) inherits from another derived class, forming a hierarchy of classes with multiple levels. Each level of the hierarchy represents a separate level of inheritance. The subclasses inherit the attributes and methods from their immediate superclass and can also add their own attributes and methods.

Example:

```
class Vehicle:
        def __init__(self, name, price) -> None:
        self.name = name
        self.price = price
class Bus(Vehicle):
        def __init__(self, name, price, seat) -> None:
        self.seat = seat
        super().__init__(name, price)
class ACBus(Bus):
        def __init__(self, name, price, seat, temperature) -> None:
        self.temperature = temperature
        super().__init__(name, price, seat)
```

# Here 'ACBus' derived from 'Bus' class and 'Bus' class derived from 'Vehicle' class

Question No. 05

Write the advantages of using inner functions in Python OOP. Provide specific use cases where inner functions can enhance code readability and organization. (note: answer with proper examples). **15**

Answer No. 05

Advantages of using inner functions in Python OOP:

1) **Encapsulation:** Inner functions help in encapsulating functionality within a specific scope, allowing for better organization and limiting the visibility of functions to the outer scope. This can help prevent name clashes and improve code maintainability.

   Example:

   ```
   def double_decker():
           print('starting the double decker')
           def inner_fun():
           print('inside the inner')
           return 3000
           return inner_fun
   print(double_decker()())
   ```

2) **Readability**: Inner functions can improve code readability by providing a more concise and focused implementation. They can help in breaking down complex logic into smaller, more manageable parts.

   Example:

   ```
   def do_something(work):
           print('work started')
           work()
           print('work ended')
   def coding():
           print('coding in python')
   do_something(coding)
   def sleeping():
           print('sleeping and dreaming in python')
   do_something(sleeping)
   ```

Question No. 06

Write Python program to solve **Frequency Array** **15**

Answer No. 06

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

frequency = {}

for num in A:
        if num in frequency:
        frequency[num] += 1
        else:
        frequency[num] = 1

for i in range(1, M+1):
        if i in frequency:
        print(frequency[i])
        else:
        print(0)
```

Question No. 07

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


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)
```

Modify the Cricketer class to find the youngest player using the concept of operator overloading and lastly print his name. **15**

Answer No. 07

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

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

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

```
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)
```