1. **Write what is meant by operator overloading and method overriding with examples.**

**Operator Overloading:**

Operator overloading is a feature in object-oriented programming that allows operators to be used with user-defined data types in addition to their standard use with built-in data types. In other words, you can redefine the behavior of operators for your custom classes.

Example:

```
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, team) -> None:
        self.team = team
        super().__init__(name, age, height, weight)


    def __add__(self, other):
        return self.age + other.age

s=Cricketer('sakib',40,12,16,'BD')
c=Cricketer('Mushi',34,10,14,'BD')
print(s+c)
```

### Method Overriding:

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

Example:

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

    def eat(self):
        print('vat')

class Cricketer(Person):
    def __init__(self,name,weight,height,team):
        self.team=team

        super().__init__(name,weight,height)

       #OVerride

    def eat(self):
        print('Vegetables')
```

2. **Write down 4 differences between the class method and static method with proper examples.**

(i)***Access to Class and Instance Variables:***

Class Method: Has access to the class and its class-level variables but not to the instance-specific variables. It takes the class (cls) as its first parameter.
Static Method: Does not have access to the class or instance-specific variables. It behaves like a regular function but is defined within the class for organization.

Example:

```
class MyClass:
    class_variable = "I am a class variable"

    @classmethod
    def class_method(cls):
        print(cls.class_variable)

    @staticmethod
    def static_method():
        print("I am a static method")

MyClass.class_method()  # Output: I am a class variable
MyClass.static_method()  # Output: I am a static method
```

(ii)
**Accessing the Instance:**

- Class Method: Cannot access or modify instance-specific variables directly. It works with the class and its class-level variables.
- Static Method: Has no access to the instance itself. It's similar to a regular function within the class.

Example:

```
class MyClass:
    instance_variable = "I am an instance variable"

    @classmethod
    def class_method(cls):
        # Cannot access instance_variable directly
        print(cls.instance_variable)  # Output: I am an instance variable

    @staticmethod
    def static_method():
        # Cannot access instance_variable directly
        print("I am a static method")

obj = MyClass()
obj.class_method()
obj.static_method()
```

(iii)

**Decorator Usage:**

- Class Method: Decorated with `@classmethod`.
- Static Method: Decorated with `@staticmethod`.

Example:

```
class MyClass:
    @classmethod
    def class_method(cls):
        pass

    @staticmethod
    def static_method():
        Pass
```

(iV)

**Usage of Parameters:**

- Class Method: Takes the class (`cls`) as its first parameter, allowing it to work with class-level variables.
- Static Method: Does not take the class or instance as its first parameter, treating it like a standalone function within the class.

Example

```
class MyClass:
    class_variable = "I am a class variable"
    @classmethod
    def class_method(cls):
        print(cls.class_variable)

    @staticmethod
    def static_method():
        print("I am a static method")

MyClass.class_method()  # Output: I am a class variable
MyClass.static_method()  # Output: I am a static method
```

### 3. Write what are getter and setter with proper examples

```
# read only --> you can not set the value. value can not be changed
# getter --> get a value of a property through a method. Most of the time, you will
get the value of a private attribute.
# setter --> set a value of a property through a method. Most of the time, you will
set the value of a private property.

class User:
    def __init__(self,name,age,money) -> None:
        self._name=name
        self._age=age
        self.__money=money

# getter without any setter is a read-only attribute
    @property
    def age(self):
        return self._age

    #getter
    @property
    def salary(self):   #this is private
        return self .__money
    #setter
    @salary.setter
    def salary(self,value):
        if value<0:
            return 'salary cannot be negative'
        self.__money+=value


samsu=User('kopa',34,1000)
print(samsu.age)
print(samsu.salary)
samsu.salary=1200
print(samsu.salary)
```

4. Explain the difference between inheritance and composition with proper examples.

```
# inheritance provides you "is a" relation

class Animal:
    pass

# Dog is an animal
```

```
class Dog(Animal):
    pass

# Tiger is an animal
class Tiger(Animal):
    pass



class Furniture:
    pass

# chair is a furniture
class Chair(Furniture):
    pass

# table is a furniture
class Table(Furniture):
    pass

# bed is a furniture
class Bed(Furniture):
    pass


# inheritance vs composition
class CPU:
    def __init__(self, cores) -> None:
        self.cores = cores

class RAM:
    def __init__(self, size) -> None:
        self.size = size

class HardDrive:
    def __init__(self, capacity) -> None:
        self.capacity = capacity
```

```
# computer has a cpu
# computer has a ram
# computer has a hard drive
class Computer:
    def __init__(self, cores, ram_size, hd_capacity) -> None:
        self.cpu = CPU(cores)
        self.ram = RAM(ram_size)
        self.hard_disc = HardDrive(hd_capacity)


mac = Computer(8, 16, 512)
```

# ANS
# Operator overloading

```
class GridPoint:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):      # Overloading + operator
        return GridPoint(self.x + other.x, self.y + other.y)

    def __str__(self):             # Overloading "to string" (for
printing)
        string = str(self.x)
        string = string + ", " + str(self.y)
        return string
      def __gt__(self, other):     # Overloading > operator (Greater Than)
            return self.x > other.x
point1 = GridPoint(3, 5)
point2 = GridPoint(-1, 4)
point3 = point1 + point2     # Add two points using __add__() method
```

```
print(point3)                      # Print the attributes using __str__()
method
if point1 > point2:          # Compares using __gt__() method
   print('point1 is greater than point2')
```

## Method overloading

```
class A:
    def first(self):
        print("First function of class A")

    def second(self):
        print('Second function of class A')

# Derived Class
class B(A):
    # Overriden Function
    def first(self):
        print("Redefined function of class A in class B")

    def display(self):
        print('Display Function of Child class')

# Driver Code
if(__name__ == "__main__"):
    # Creating child class object
    child_obj = B()

    # Calling the overridden method
    print("Method Overriding\n")
    child_obj.first()

    # Calling the original Parent class method
    # Using parent class object.
    A().first()
```