Topic:

- Method Over-Riding

Method Over-Riding

In a class Animal, it has a constructor, def __init__(self):

```
class Animal:
    def __init__(self):
        _----___----____            this is the constructor for class Animal
        _----___----____
```

Let’s suppose animal has a method, walk

```
    Def walk(self):
        _----___----____            this is the method for class Animal
        _----___----____
```

```
class Dog(Animal)          → this is inherting from Animal
    def __init__(self):
        super().__init__(self):
            _----___----____
            _----___----____

    def bark(self)
        _----___----____
        _----___----____
```

If I want to over ride walk,

```
class Animal:
    def __init__(self):
        print("Animal constructor is called")

    def walk(self):
        print("Animal is walking")

class Dog(Animal):
    def __init__(self):
        super().__init__()
        print("Dog constructor is called")

    def bark(self):
        print("Dog is barking")

    def walk(self):
        print("Dog is walking")

if __name__ == "__main__":
    d = Dog()
    d.walk()
```

```
OUTPUT:
Animal constructor is called
Dog constructor is called
Dog is walking
```

If you want to call the parent also in a method, then use the *super().walk()* in the method and it will give the output for both.

INPUT:

```
class Animal:
    def __init__(self):
        print("Animal constructor is called")

    def walk(self):
        print("Animal is walking")

class Dog(Animal):
    def __init__(self):
        super().__init__()
        print("Dog constructor is called")

    def bark(self):
        print("Dog is barking")

    def walk(self):
        super().walk()
        print("Dog is walking")

if __name__ == "__main__":
    d = Dog()
    d.walk()
```

OUTPUT:

```
Animal constructor is called
Dog constructor is called
Animal is walking
Dog is walking
```

This is called Method Over-Riding

OOPs → Object Oriented Programming S.

1. **Inheritance**: parent children relationship.

    - Children have all the attributes and methods of the parents and you can do method over riding in this.
2. **Encapsulation**: Anything inside a class should not be directly accessible outside the class.
3. **Abstraction**: hiding the details of a method or a class.
4. **Polymorphism**: poly means meaning and morphism means behavior or forms. You code can take different forms.

Dog is a type of Animal. if we say d=Dog(). So d is a type of dog and animal.
Animal – Dog – Alsatian.
Alsatian will have attributes of dog and since dog has attrbutes of animal, Alsatian will also have attributes of animal.

Inheritance

```
#Inheritance

class Bank:
    def __init__(self, name):
        self.name = name
        self.amount = 0

    def add_amount(self, x):
        self.amount += x

class HDFC(Bank):
    def __init__(self):
        name = "HDFC"
        super().__init__(name)

if __name__ == "__main__":
    bank = HDFC()
    bank.add_amount(100)
    print(bank.amount)
```

OUTPUT:

```
100
```

Encapsulation

```
class Bank:
    def __init__(self, name):
        self.name = name
        self.amount = 0

    def add_amount(self, x):
        self._amount += x

    def get_amount(self):
        return self._amount

class HDFC(Bank):
    def __init__(self):
        name = "HDFC"
        super().__init__(name)

if __name__ == "__main__":
    bank = HDFC()
    bank.add_amount(100)
    print(bank.amount)
```

Since we cannot use the add_amount, because it is made private, so we will create a method which is public, and through it, we are able to get the add_amount.

Abstraction

Bank is hiding the get_amount. HDFC is inheriting from bank and hiding get_amount.

Polymorphism

Same type of objects, but different behavior.

```
#polymorphism

class Shape:
    def __init__(self, name):
        self.name = name


class Square(Shape):
    def __init__(self, name, side):
        super().__init__(name)
        self.side = side

    def area(self):
        return self.side * self.side


class Rectangle(Shape):
    def __init__(self, name, x, y):
        super().__init__(name)
        self.x = x
        self.y = y

    def area(self):
        return self.x * self.y


if __name__ == "__main__":
    square = Square("square", 6)
    print(f"The area of {square.name} is {square.area()}")

    rectangle = Rectangle("rectangle", 5, 3)
    print(f"The area of {rectangle.name} is {rectangle.area()}")
```

In this example, the attributes of the class Shape are obtained by the children classes, Square and Rectangle.

NOTE: if your constructor is not doing anything, you can delete the constructor.

Example:

```
class Payment:
    def __init__(self, amount):
        self.amount = amount


    def make_payment(self):
        print("initiating payment")
        self.pay()

    def pay(self):
        print("default payment")

class COD(Payment):
    def __init__(self,amount):
        self.amount = amount

    def pay(self):
        print("default payment")

class CreditCar(Payment):
    def __init__(self,amount):
        self.amount = amount

    def pay(self):
        print("default payment")

if __name__ == "__main__":
    cod = COD(1000)
    cod.make_payment()
```

OUTPUT:

```
initiating payment
default payment
```