# **Name: Abdurrahman Qureshi**

# **Roll No: 242466**

Practical No: 12

**Q) To demonstrate following OOP concept in python**

1. **Polymorphism.**
2. **Constructor with Inheritance**
3. **Method overloading**
4. **Method overriding**
5. **Polymorphism**

**Explain the above concepts with example [ Theory ]**

**Explain the code in your own words**

a) Constructor with Inheritance

CODE:

class Character:

    def \_\_init\_\_(self, name, role, weapon):

        self.name = name

        self.role = role

        self.weapon = weapon

    def show\_info(self):

        print(f"Name: {self.name}")

        print(f"Role: {self.role}")

        print(f"Weapon: {self.weapon}")

class Human(Character):

    def \_\_init\_\_(self, name, role, weapon, tribe):

        super().\_\_init\_\_(name, role, weapon)

        self.tribe = tribe

    def show\_info(self):

        super().show\_info()

        print(f"Tribe: {self.tribe}")

class Machine(Character):

    def \_\_init\_\_(self, name, role, weapon, weakness):

        super().\_\_init\_\_(name, role, weapon)

        self.weakness = weakness

    def show\_info(self):

        super().show\_info()

        print(f"Weakness: {self.weakness}")

class Tribe(Character):

    def \_\_init\_\_(self, name, role, weapon, territory):

        super().\_\_init\_\_(name, role, weapon)

        self.territory = territory

    def show\_info(self):

        super().show\_info()

        print(f"Territory: {self.territory}")

aloy = Human("Aloy", "Hunter", "Bow", "Nora")

thunderjaw = Machine("Thunderjaw", "Apex Predator", "Disc Launcher", "Tearblast Arrows")

tenakth = Tribe("Tenakth", "Warrior Tribe", "Spear", "The Clan Lands")

print("--- Human ---")

aloy.show\_info()

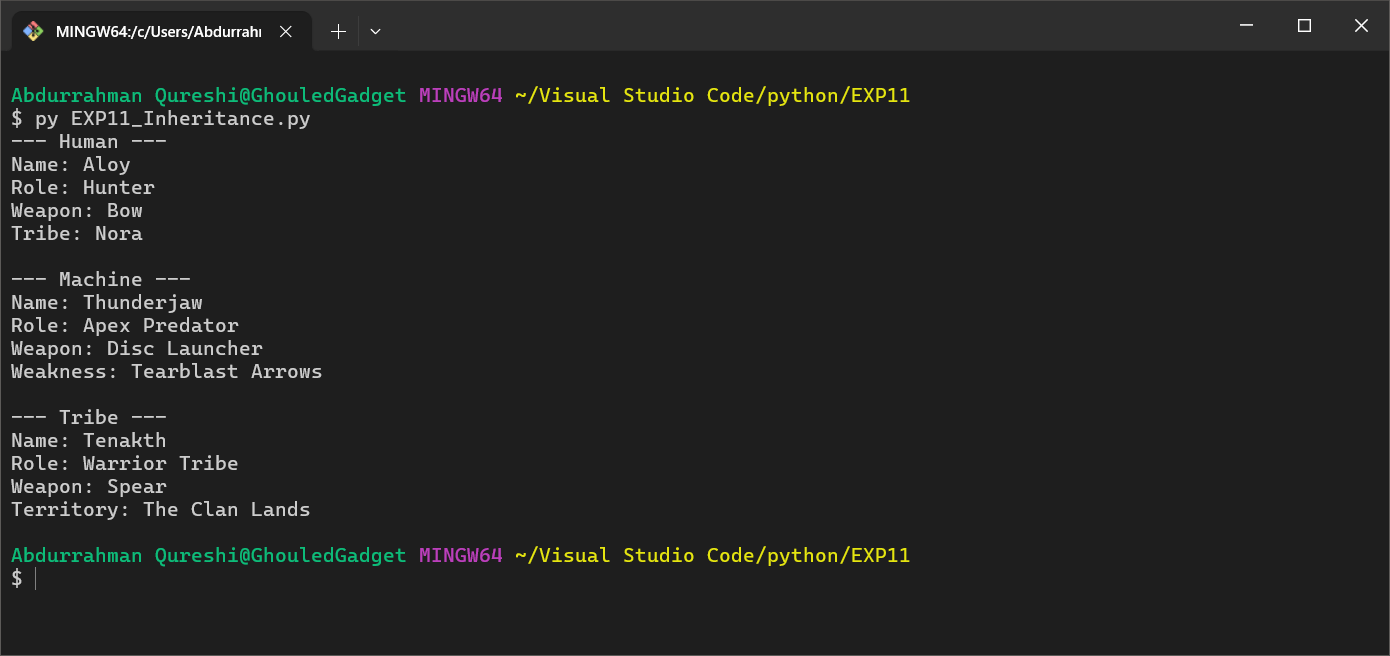
print("\n--- Machine ---")

thunderjaw.show\_info()

print("\n--- Tribe ---")

tenakth.show\_info()

OUTPUT:



EXPLANATION:

This example demonstrates inheritance with Human, Machine, and Tribe inheriting from Character. Each subclass overrides the show\_info method to add specific details like tribe, weakness, or territory, showcasing encapsulation and polymorphism through unique behaviours.

b) Polymorphism – (Method Overloading)

CODE:

class Troop:

    def attack(self, target, weapon=None):

        if weapon:

            print(f"Attacking {target} with {weapon}!")

        else:

            print(f"Attacking {target} with a default weapon!")

class Hero(Troop):

    def attack(self, target, weapon=None, ability=None):

        if ability:

            print(f"Attacking {target} with {weapon} and using {ability}!")

        else:

            super().attack(target, weapon)

archer = Troop()

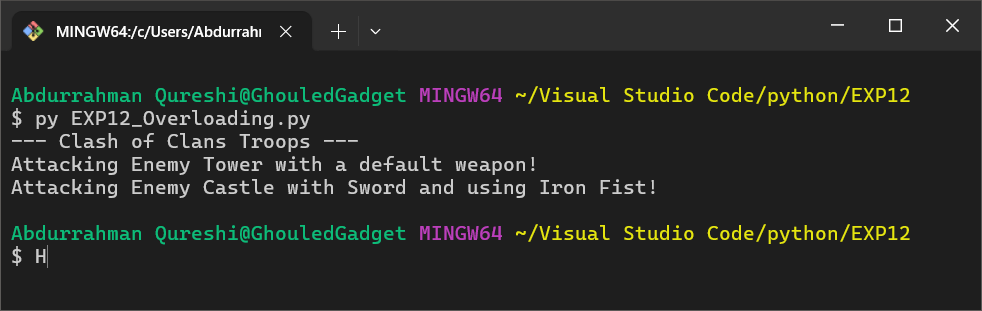
barbarian\_king = Hero()

print("--- Clash of Clans Troops ---")

archer.attack("Enemy Tower")

barbarian\_king.attack("Enemy Castle", "Sword", "Iron Fist")

OUTPUT:



EXPLANATION:

This example shows inheritance with Hero extending Troop. It simulates method overloading in Hero’s attack method using optional parameters (weapon, ability). Polymorphism is demonstrated as attack behaves differently for Troop and Hero objects.

c) Polymorphism - (Method Overriding)

CODE:

class Survivor:

    def \_\_init\_\_(self, name, health):

        self.name = name

        self.health = health

    def use\_ability(self):

        print(f"{self.name} uses a generic ability.")

class Coach(Survivor):

    def use\_ability(self):

        print(f"{self.name} uses a powerful melee attack to clear zombies!")

class Zoey(Survivor):

    def use\_ability(self):

        print(f"{self.name} uses precise shooting to take down the special infected!")

coach = Coach("Coach", 100)

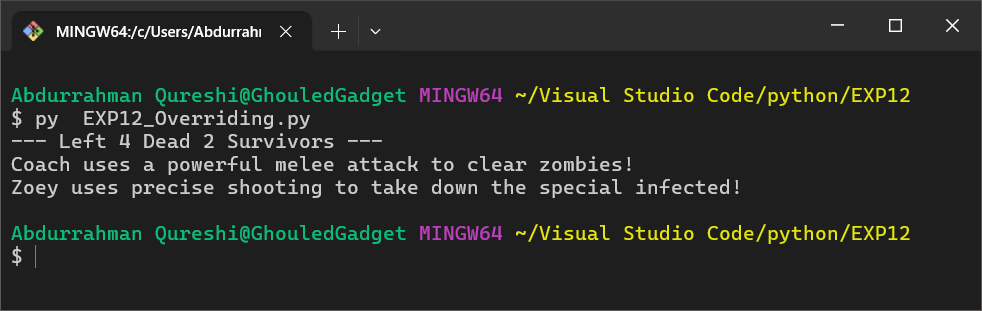
zoey = Zoey("Zoey", 80)

print("--- Left 4 Dead 2 Survivors ---")

coach.use\_ability()

zoey.use\_ability()

OUTPUT:



EXPLANATION:

This example highlights inheritance with Coach and Zoey inheriting from Survivor. Both subclasses override the use\_ability method to provide unique implementations, demonstrating polymorphism as the method behaves differently based on the object type.

d) Operator Overloading

CODE:

class OnePiece:

    def \_\_init\_\_(self, name, devil\_fruit, role, age, gender):

        self.name = name

        self.devil\_fruit = devil\_fruit

        self.role = role

        self.age = age

        self.gender = gender

    def \_\_ls\_\_(self, next):

        if self.age < next.age:

            return -1

        else:

            return 1

    def \_\_add\_\_(self, next):

        return self.age + next.age

nico\_robin = OnePiece("Nico Robin", "Flower Flower Fruit", "Archeologist", 32, "Female")

luffy = OnePiece("Monkey D. Luffy", "Gum Gum Fruit", "Captain", 21, "Male")

print("------------ Nico Robin ------------\n")

print("Name:", nico\_robin.name)

print("Gender:", nico\_robin.gender)

print("Age:", nico\_robin.age)

print("Role:", nico\_robin.role)

print("Devil Fruit:", nico\_robin.devil\_fruit)

print("\n------------ Monkey D. Luffy ------------\n")

print("Name:", luffy.name)

print("Gender:", luffy.gender)

print("Age:", luffy.age)

print("Role:", luffy.role)

print("Devil Fruit:", luffy.devil\_fruit)

print("\nCombined Age of both: ", nico\_robin + luffy)

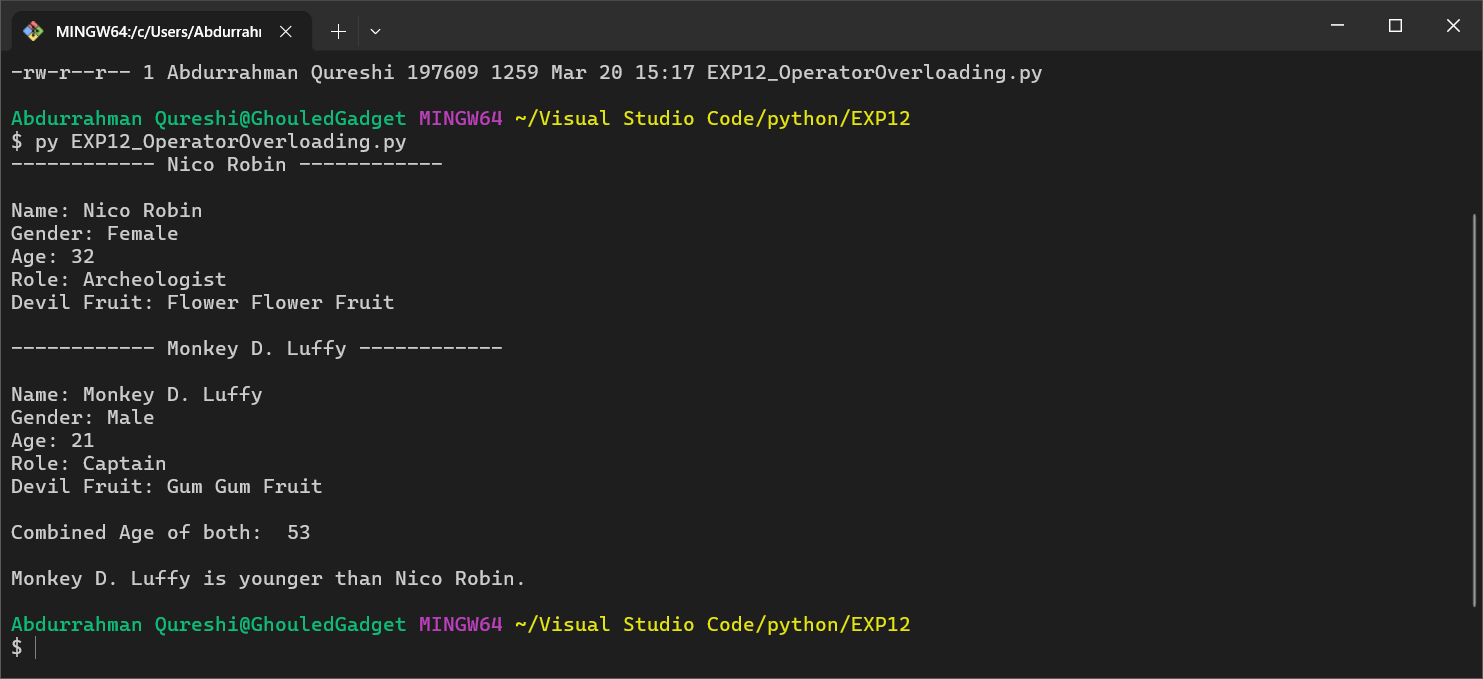
if nico\_robin.age < luffy.age:

    print("\nNico Robin is younger than Monkey D. Luffy.")

else:

    print("\nMonkey D. Luffy is younger than Nico Robin.")

OUTPUT:



EXPLANATION:

The code defines a OnePiece class with attributes like name, devil\_fruit, role, age, and gender. It includes methods \_\_ls\_\_ (unused) and \_\_add\_\_ to compare and sum ages. It creates two characters, displays their details, and compares their ages.

e) Multiple Inheritance

CODE:

class Daddy:

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def printDaddy(self):

        print('From Daddy')

class Mommy:

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def printMommy(self):

        print('From Mommy')

class Baby(Daddy,Mommy):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def printBaby(self):

        print('I am Baby')

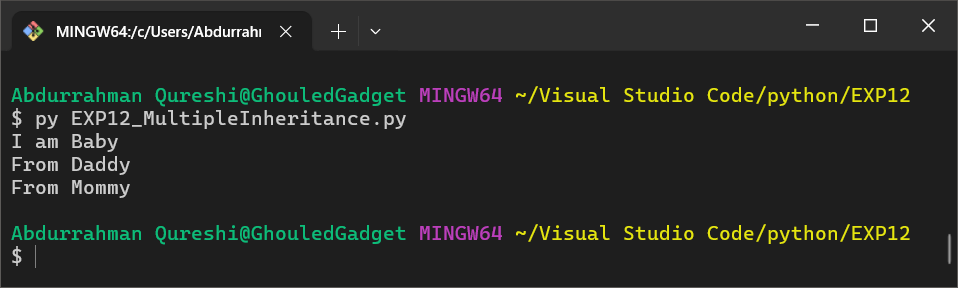
        super().printDaddy()

        super().printMommy()

obj = Baby()

obj.printBaby()

OUTPUT:



EXPLANATION:

The Baby class inherits from Daddy and Mommy. Its printBaby method prints "I am Baby", then uses super() to call printDaddy ("From Daddy") and printMommy ("From Mommy"). The Method Resolution Order (MRO) ensures proper execution.

f) Method Resolution Order

CODE:

class Daddy:

    def \_\_init\_\_(self):

        self.person='Daddy'

        print(self.person)

        super().\_\_init\_\_()

class Mommy:

    def \_\_init\_\_(self):

        self.person='Mommy'

        print(self.person)

        super().\_\_init\_\_()

class Baby(Daddy,Mommy):

    def \_\_init\_\_(self):

        self.person='Baby'

        print(self.person)

        super().\_\_init\_\_()

obj = Baby ()

OUTPUT:



EXPLANATION:

The Baby class inherits from Daddy and Mommy. When obj = Baby() is called, it prints "Baby", then "Daddy", and finally "Mommy" due to the super().\_\_init\_\_() calls and the Method Resolution Order (MRO).