**DATA SCIENCE – BWT – WEEK – 2**

**TASK – 6**

**SAIFULLAH KHAN**

**Modular Programming Approach In Python ( Classes , Inheritance , Encapsulation )**

**INTRODUCTION:**  
  
Modular programming in Python involves breaking down a program into smaller, manageable, and reusable modules. This approach enhances code readability, maintainability, and reusability. Here are key concepts and examples:  
  
**CLASSES:**  
  
Classes are the building blocks of object-oriented programming (OOP) in Python. They encapsulate data and behavior into a single unit called an object.

class CoffeeShop:

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

        self.name = name

        self.location = location

    def display\_info(self):

        print(f"Coffee Shop Name: {self.name}, Location: {self.location}")

**Inheritance**

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reuse and logical hierarchy.

class CoffeeShop:

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

        self.name = name

        self.location = location

    def display\_info(self):

        print(f"Coffee Shop Name: {self.name}, Location: {self.location}")

class SpecialtyCoffeeShop(CoffeeShop):

    def \_\_init\_\_(self, name, location, specialty\_drinks):

        super().\_\_init\_\_(name, location)

        self.specialty\_drinks = specialty\_drinks

    def display\_specialties(self):

        print(f"Specialty Drinks: {', '.join(self.specialty\_drinks)}")

# Creating an instance of SpecialtyCoffeeShop

specialty\_shop = SpecialtyCoffeeShop("Saif's Coffee", "Islamabad", ["Latte", "Espresso", "Cappuccino"])

specialty\_shop.display\_info()

specialty\_shop.display\_specialties()

### Encapsulation

Encapsulation involves restricting access to certain components of an object and can be achieved using private and protected attributes and methods. In Python, this is typically done using underscores.

class CoffeeShop:

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

        self.\_name = name

        self.\_\_location = location

    def display\_info(self):

        print(f"Coffee Shop Name: {self.\_name}, Location: {self.\_\_location}")

    def \_display\_protected\_info(self):

        print(f"Protected: {self.\_name}")

    def \_\_display\_private\_info(self):

        print(f"Private: {self.\_\_location}")

shop = CoffeeShop("Saif's Coffee", "Islamabad")

shop.display\_info()

shop.\_display\_protected\_info()

### Modular Approach Example

Combining these concepts, we can create a modular design for a simple coffee shop application:

# coffee\_shop.py

class CoffeeShop:

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

        self.name = name

        self.location = location

    def display\_info(self):

        print(f"Coffee Shop Name: {self.name}, Location: {self.location}")

# specialty\_coffee\_shop.py

from coffee\_shop import CoffeeShop

class SpecialtyCoffeeShop(CoffeeShop):

    def \_\_init\_\_(self, name, location, specialty\_drinks):

        super().\_\_init\_\_(name, location)

        self.specialty\_drinks = specialty\_drinks

    def display\_specialties(self):

        print(f"Specialty Drinks: {', '.join(self.specialty\_drinks)}")

# main.py

from specialty\_coffee\_shop import SpecialtyCoffeeShop

def main():

    specialty\_shop = SpecialtyCoffeeShop("Saif's Coffee", "Islamabad", ["Latte", "Espresso", "Cappuccino"])

    specialty\_shop.display\_info()

    specialty\_shop.display\_specialties()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**CONCLUSION:**  
Modular programming in Python, using classes, inheritance, and encapsulation, allows for building complex applications in a manageable and reusable way. By breaking down the program into separate modules, each with specific responsibilities, you can achieve a clean, organized, and maintainable codebase.