Assignment-6.3

2303A51304

Batch-05

Task-01:

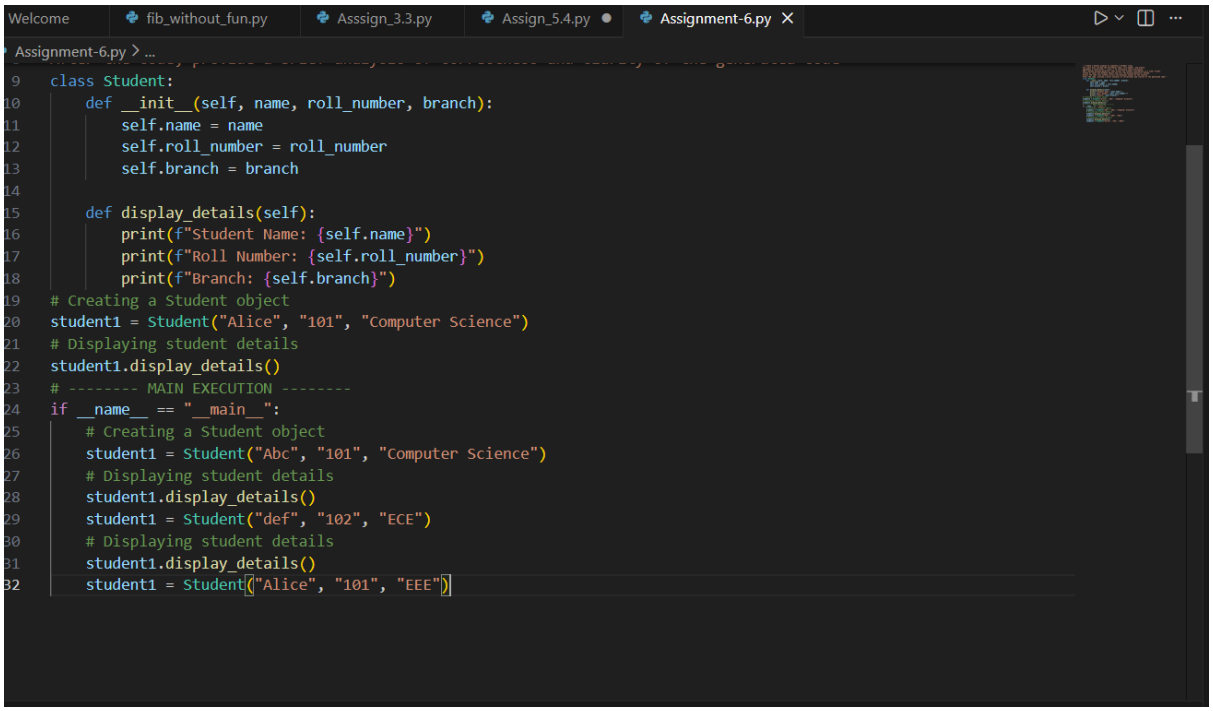
Prompt: Create a Python program to complete a Student class.

The class must include the attributes name, roll_number, and branch.

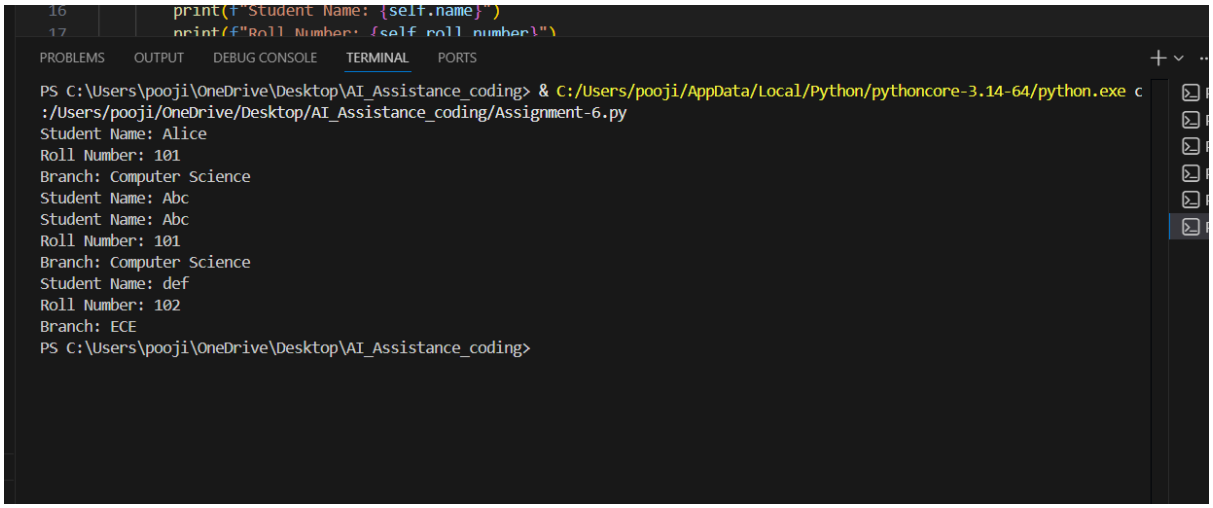
Implement a constructor (__init__) to initialize these attributes.

Add a method display_details() that prints the student information in a clear format. Create at least one Student object and call the display_details() method. Ensure the code runs without errors and displays output on the console. After the code, provide a brief analysis of correctness and clarity of the generated code

Code:



Output:



Explanation:

The Student class is defined with three attributes: name, roll_number, and branch to store student information.

The __init__ constructor initializes these attributes whenever a new Student object is created.

The display_details() method prints the student details in a readable and structured format.

Student objects are created using different values to demonstrate reusability of the class.

The program executes correctly without errors and clearly displays the output on the console.

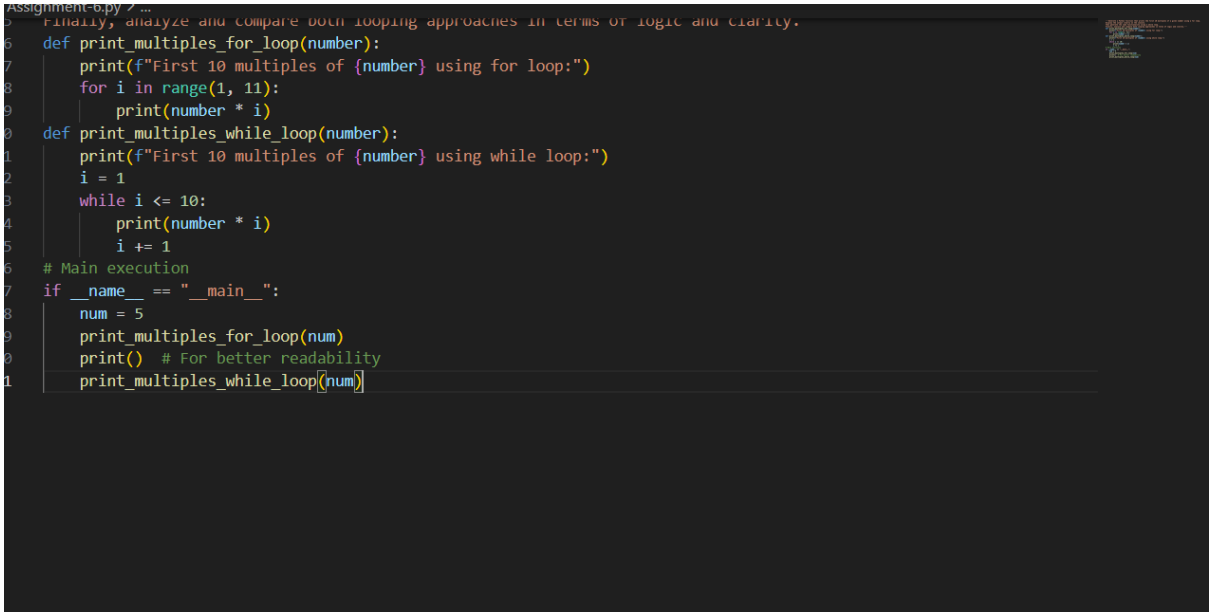
Task-02:

Prompt:

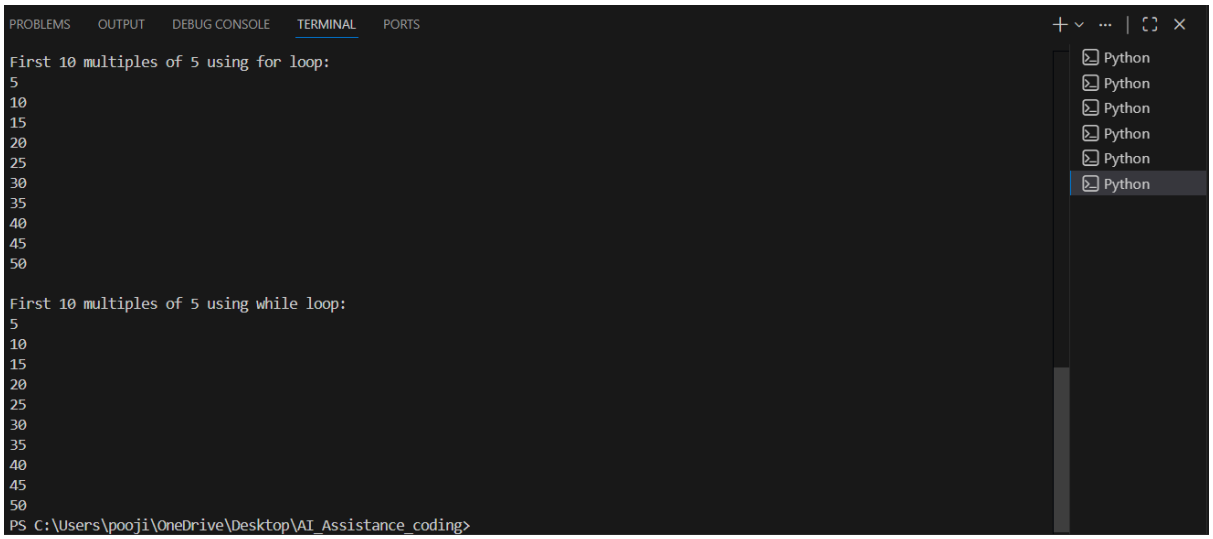
Generate a Python function that prints the first 10 multiples of a given number using a for loop. Display the output clearly on the console. Then generate the same functionality using a while

loop.
Finally, analyze and compare both looping approaches in terms of logic and clarity.

Code:



Output:



Explanation:

Both loops correctly generate the first 10 multiples of a number.
The for loop is simpler and more readable for fixed iterations.
The while loop provides better control but requires careful condition handling. Both approaches produce the same output.
Choice of loop depends on problem requirements.

Task-03:

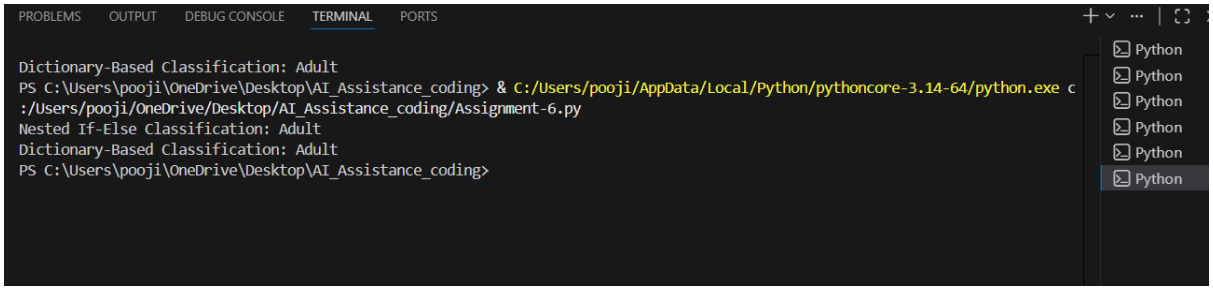
Prompt:

Generate a Python function that classifies a person into age groups using nested if-elif-else conditions. The age groups should be: Child, Teenager, Adult, and Senior. Display the classification result on the console. Then generate the same age classification using an alternative conditional approach such as simplified conditions or dictionary-based logic. Finally, explain and analyze the conditional logic used in both approaches

Code:

```python
def classify_age_nested(age):
    """Classify age into groups using nested if-elif-else conditions."""
    if age < 0:
        return "Invalid age"
    else:
        if age <= 12:
            return "Child"
        else:
            if age <= 19:
                return "Teenager"
            else:
                if age <= 64:
                    return "Adult"
                else:
                    return "Senior"
def classify_age_dict(age):
    """Classify age into groups using dictionary-based logic."""
    if age < 0:
        return "Invalid age"

    age_groups = {
        range(0, 13): "Child",
        range(13, 20): "Teenager",
        range(20, 65): "Adult",
        range(65, 150): "Senior"
    }

    for age_range, group in age_groups.items():
        if age in age_range:
            return group
# -------- MAIN EXECUTION --------
if __name__ == "__main__":
    age = 45

    # Using nested if-elif-else conditions
    result_nested = classify_age_nested(age)
    print("Nested If-Else Classification:", result_nested)

    # Using dictionary-based logic
    result_dict = classify_age_dict(age)
    print("Dictionary-Based Classification:", result_dict)
```

Output:



Explanation:

The age classification is implemented using conditional statements.
The if–elif–else approach provides clear and direct logic.
The alternative approach separates data from logic for flexibility.
Both methods correctly classify age groups.
Choice of approach depends on system complexity.

Task-04:

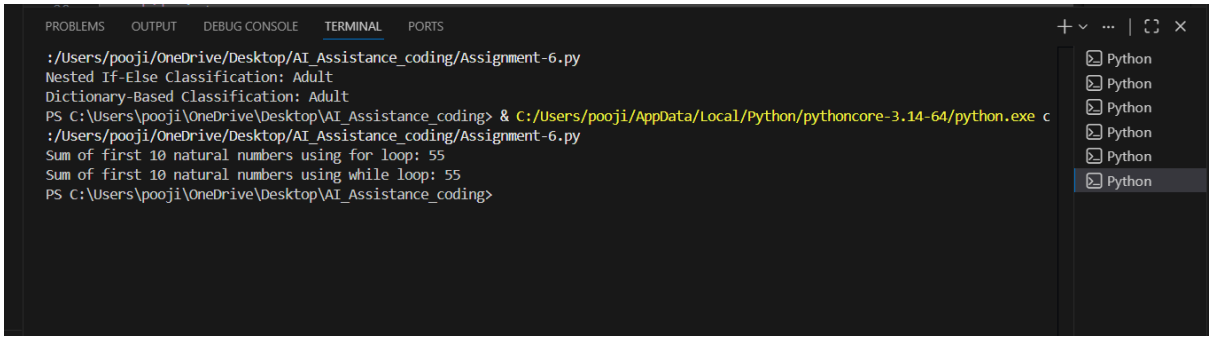
Prompt:

Generate a Python function sum_to_n() that calculates the sum of the first n natural numbers using a for loop. Show the output for a sample input. Analyze the generated loop logic. Then suggest an alternative implementation using a while loop and a mathematical formula. Finally, compare the different approaches

Code:

```
'''Generate a Python function sum_to_n() that calculates the sum of the first n natural numbers using a for loop.
Show the output for a sample input.
Analyze the generated loop logic.
Then suggest an alternative implementation using a while loop and a mathematical formula.
Finally, compare the different approaches'''
def sum_to_n_for_loop(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
# Sample input
n = 10
result_for_loop = sum_to_n_for_loop(n)
print("Sum of first", n, "natural numbers using for loop:", result_for_loop)
# Analysis of the for loop logic:
# The for loop iterates from 1 to n, adding each integer to the total sum.
def sum_to_n_while_loop(n):
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total
result_while_loop = sum_to_n_while_loop(n)
print("Sum of first", n, "natural numbers using while loop:", result_while_loop)
```

Output:



Explanation:

The sum of first n natural numbers is calculated using looping and formula-based methods.
The for loop iterates from 1 to n and accumulates the sum.
The while loop performs the same logic with explicit condition control. The formula method is the most efficient approach.

Task-05:

Prompt:

Generate a Python program to design a BankAccount class.
The class should include methods deposit(), withdraw(), and check_balance(). Initialize the account with an account holder name and initial balance using a constructor. Demonstrate deposit and withdrawal operations and display the updated balance. Add meaningful comments and explain the logic of the code clearly.

Code:

```python
signment-6.py > ...
class BankAccount:
    def __init__(self, name, balance=0):
        # Initialize account holder name and balance
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        # Deposit money if amount is positive
        if amount > 0:
            self.balance += amount
            print(f"Deposited ₹{amount}")
        else:
            print("Deposit amount must be positive")

    def withdraw(self, amount):
        # Withdraw money if sufficient balance is available
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrawn ₹{amount}")
        else:
            print("Insufficient balance")

    def check_balance(self):
        # Display current balance
        print(f"Current Balance: ₹{self.balance}")


# -------- MAIN EXECUTION --------
if __name__ == "__main__":
    # Creating a BankAccount object
    account = BankAccount("Poojitha", 5000)

    # Checking initial balance
    account.check_balance()

    # Depositing amount
    account.deposit(2000)
    account.check_balance()

    # Withdrawing amount
    account.withdraw(3000)
    account.check_balance()
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Curren
PS C:\                    op\AI_Assistance_coding> & C:/Users/pooji/AppData/Local/Python/pythoncore-3.14-64/pytho
n.exe c:/Users/pooji/OneDrive/Desktop/AI_Assistance_coding/Assignment-6.py
Current Balance: ₹5000
Deposited ₹2000
Current Balance: ₹7000
Withdrawn ₹3000
Current Balance: ₹4000
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding>
```

Open file in editor (ctrl + click)

Explanation:

The BankAccount class is designed using object-oriented principles.
It encapsulates account details and banking operations.
Input validation ensures correct deposit and withdrawal actions.
The program updates and displays the balance after each operation.
The code is clear, correct, and suitable for a basic banking application.