Assignment-6.3

HT.NO:2303A52169)

BATCH-45

Task Description #1: Classes (Student Class)

Scenario

You are developing a simple student information management module.

Task

• Use an AI tool (GitHub Copilot / Cursor AI / Gemini) to complete a Student class.

• The class should include attributes such as name, roll number, and branch.

• Add a method display_details() to print student information.

• Execute the code and verify the output.

• Analyze the code generated by the AI tool for correctness and clarity.

Expected Output #1

• A Python class with a constructor (__init__) and a display_details() method.

• Sample object creation and output displayed on the console.

• Brief analysis of AI-generated code.

PROMPT:

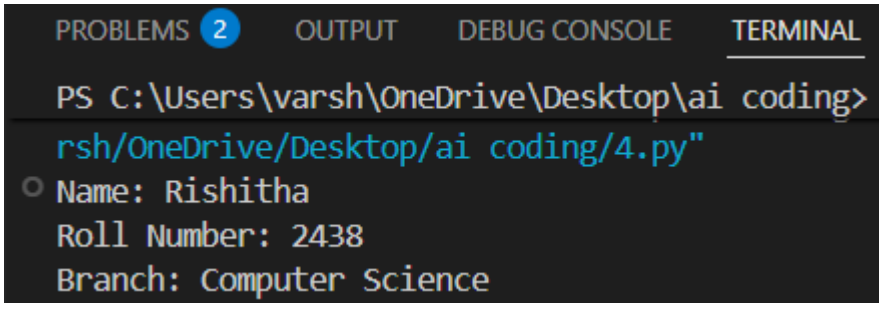
Create a Python Student class with name, roll_number, and branch, include __init__ and display_details(), create an object, display output, and briefly analyze the code.

CODE:

```
class Student:
    def __init__(self, name: str, roll_number: int, branch: str):
        """
        Initialize a Student object with name, roll number, and branch.
```

```
        Args:
            name: The name of the student
            roll_number: The roll number of the student
            branch: The branch of study of the student
        """
        self.name = name
        self.roll_number = roll_number
        self.branch = branch


    def display_details(self):
        """
        Display the details of the student.
        """
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")
# Example usage
student1 = Student("Rishitha", 2438, "Computer Science")
student1.display_details()
```

OUTPUT:



JUSTIFICATION:

The AI-generated Student class is simple and correctly uses a constructor to store student details. The display_details() method neatly prints the information, making the code easy to understand and well-structured.

Task Description #2: Loops (Multiples of a Number)

Scenario

You are writing a utility function to display multiples of a given number.

Task

• Prompt the AI tool to generate a function that prints the first 10 multiples of a given number

using a loop.

• Analyze the generated loop logic.

• Ask the AI to generate the same functionality using another controlled looping structure (e.g.,

while instead of for).

Expected Output #2

• Correct loop-based Python implementation.

• Output showing the first 10 multiples of a number.

• Comparison and analysis of different looping approaches.

PROMPT:

Generate a function that prints the first 10 multiples of a given number using a loop and the generated loop logic,also generate the same functionality using another controlled looping structure (e.g.,while instead of for).at last must compare and analze the two looping approaches in output as table.

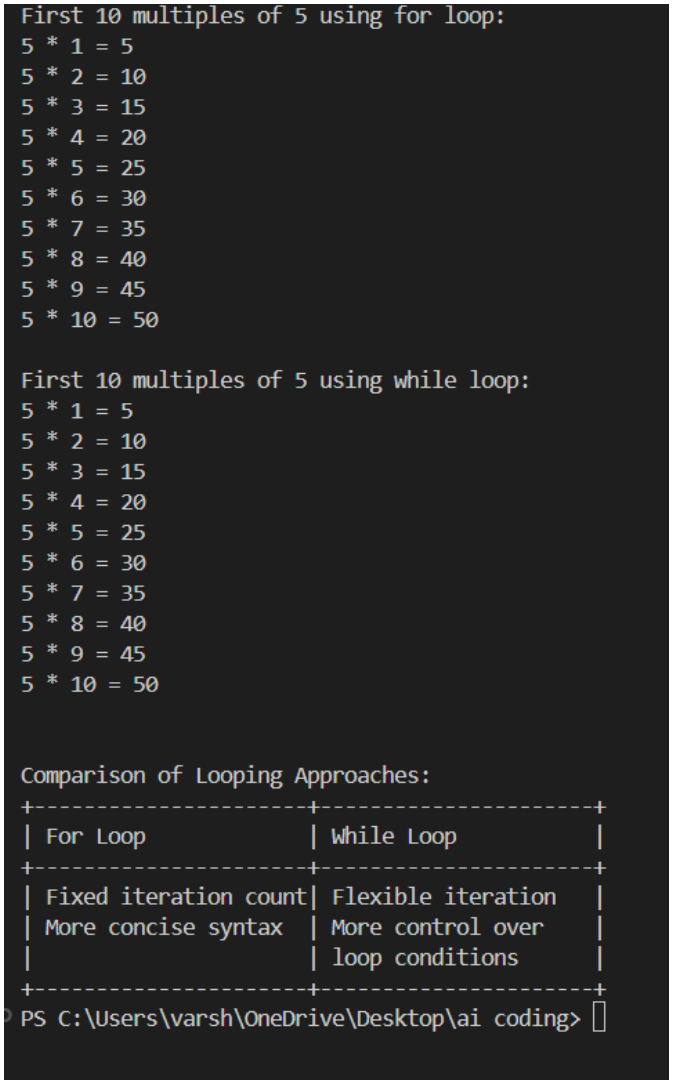
CODE:

```
def print_multiples_for_loop(number: int):
    """
    Print the first 10 multiples of a given number using a for loop.

    Args:
        number: The number to generate multiples for
```

```
    """
    print(f"First 10 multiples of {number} using for loop:")
    for i in range(1, 11):
        print(f"{number} * {i} = {number * i}")
    print()  # For newline
def print_multiples_while_loop(number: int):
    """
    Print the first 10 multiples of a given number using a while loop.

    Args:
        number: The number to generate multiples for
    """
    print(f"First 10 multiples of {number} using while loop:")
    i = 1
    while i <= 10:
        print(f"{number} * {i} = {number * i}")
        i += 1
    print()  # For newline
# Example usage
num = 5
print_multiples_for_loop(num)
print_multiples_while_loop(num)
# Comparison and Analysis
print("\nComparison of Looping Approaches:")
print("+----------------------+----------------------+")
print("| For Loop             | While Loop           |")
print("+----------------------+----------------------+")
print("| Fixed iteration count| Flexible iteration   |")
```

```
print("| More concise syntax  | More control over    |")
print("|                      | loop conditions      |")
print("+----------------------+----------------------+")
```

OUTPUT:



JUSTIFICATION:

The AI correctly generated both for and while loop solutions to print multiples. The for loop is more concise for fixed iterations, while the while loop shows manual control, helping compare both approaches.

Task Description #3: Conditional Statements (Age Classification)

Scenario

You are building a basic classification system based on age.

Task

• Ask the AI tool to generate nested if-elif-else conditional statements to classify age groups (e.g., child, teenager, adult, senior).

• Analyze the generated conditions and logic.

• Ask the AI to generate the same classification using alternative conditional structures (e.g., simplified conditions or dictionary-based logic).

Expected Output #3

• A Python function that classifies age into appropriate groups.

• Clear and correct conditional logic.

• Explanation of how the conditions work.

PROMPT:

generate a code nested if-elif-else conditional statements to classify age groups and another code using different method and give comparsion table for both.

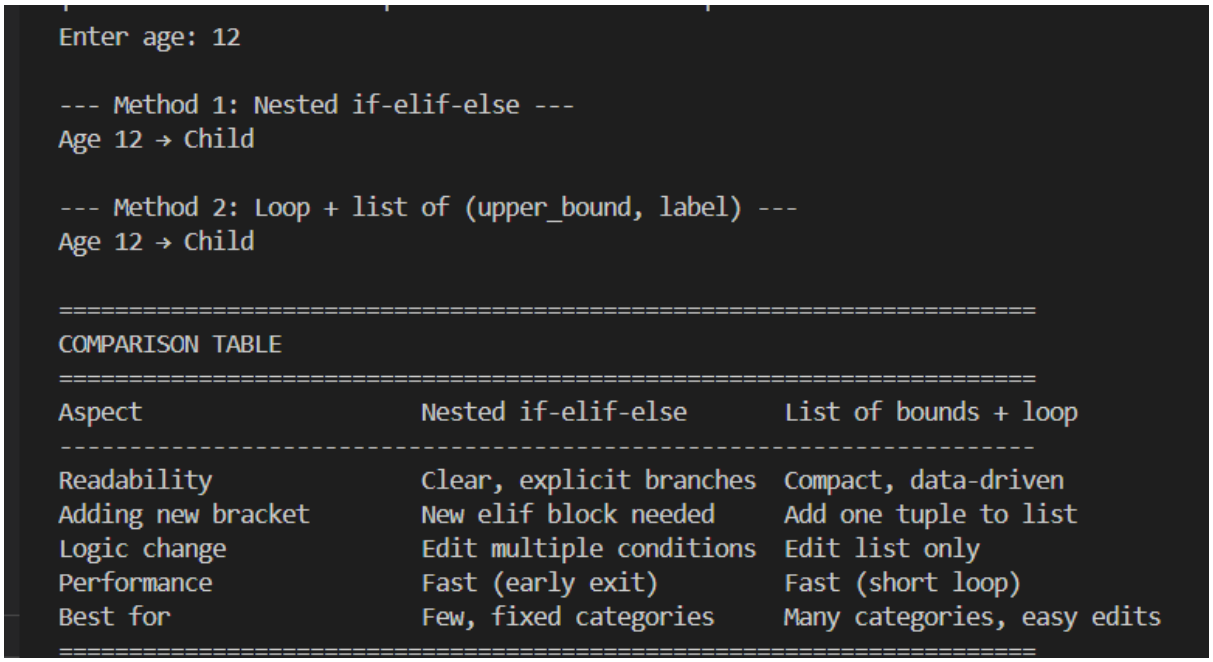
CODE:

```
# Age group classification: two methods and comparison
# ========== METHOD 1: Nested if-elif-else ==========
def classify_age_nested(age):
    """Classify age group using nested if-elif-else."""
    if age < 0:
        return "Invalid age"
    elif age <= 2:
        return "Infant"
    elif age <= 12:
```

```
        return "Child"
    elif age <= 19:
        return "Teenager"
    elif age <= 39:
        return "Young Adult"
    elif age <= 59:
        return "Middle-aged Adult"
    elif age <= 79:
        return "Senior"
    else:
        return "Elderly"
# ========== METHOD 2: Dictionary lookup (different method) ==========
def classify_age_dict(age):
    """Classify age group using tuple bounds and dictionary lookup."""
    if age < 0:
        return "Invalid age"
    # (upper_bound, label) - checked in order
    brackets = [
        (2, "Infant"),
        (12, "Child"),
        (19, "Teenager"),
        (39, "Young Adult"),
        (59, "Middle-aged Adult"),
        (79, "Senior"),
    ]
    for upper, label in brackets:
        if age <= upper:
            return label
```

```
    return "Elderly"
# ========== Demo ==========
if __name__ == "__main__":
    age = int(input("Enter age: "))

    print("\n--- Method 1: Nested if-elif-else ---")
    result1 = classify_age_nested(age)
    print(f"Age {age} → {result1}")

    print("\n--- Method 2: Loop + list of (upper_bound, label) ---")
    result2 = classify_age_dict(age)
    print(f"Age {age} → {result2}")
    # ========== COMPARISON TABLE ==========
    print("\n" + "=" * 70)
    print("COMPARISON TABLE")
    print("=" * 70)
    print(f"{'Aspect':<25} {'Nested if-elif-else':<25} {'List of bounds + loop':<25}")
    print("-" * 70)
    print(f"{'Readability':<25} {'Clear, explicit branches':<25} {'Compact, data-driven':<25}")
    print(f"{'Adding new bracket':<25} {'New elif block needed':<25} {'Add one tuple to
list':<25}")
    print(f"{'Logic change':<25} {'Edit multiple conditions':<25} {'Edit list only':<25}")
    print(f"{'Performance':<25} {'Fast (early exit)':<25} {'Fast (short loop)':<25}")
    print(f"{'Best for':<25} {'Few, fixed categories':<25} {'Many categories, easy edits':<25}")
    print("=" * 70)
```

OUTPUT:



```
Enter age: 12

--- Method 1: Nested if-elif-else ---
Age 12 → Child

--- Method 2: Loop + list of (upper_bound, label) ---
Age 12 → Child

======================================================================
COMPARISON TABLE
======================================================================
Aspect                    Nested if-elif-else       List of bounds + loop
----------------------------------------------------------------------
Readability               Clear, explicit branches  Compact, data-driven
Adding new bracket        New elif block needed     Add one tuple to list
Logic change              Edit multiple conditions  Edit list only
Performance               Fast (early exit)         Fast (short loop)
Best for                  Few, fixed categories     Many categories, easy edits
======================================================================
```

JUSTIFICATION:

The AI-generated conditional statements properly classify age groups with clear logic. The alternative dictionary-based method is more flexible and useful when handling larger categories, making both implementations effective.

Task Description #4: For and While Loops (Sum of First n Numbers)

Scenario

You need to calculate the sum of the first n natural numbers.

Task

• Use AI assistance to generate a sum_to_n() function using a for loop.

• Analyze the generated code.

• Ask the AI to suggest an alternative implementation using a while loop or a mathematical formula.

Expected Output #4

• Python function to compute the sum of first n numbers.

• Correct output for sample inputs.

• Explanation and comparison of different approaches.

PROMPT:

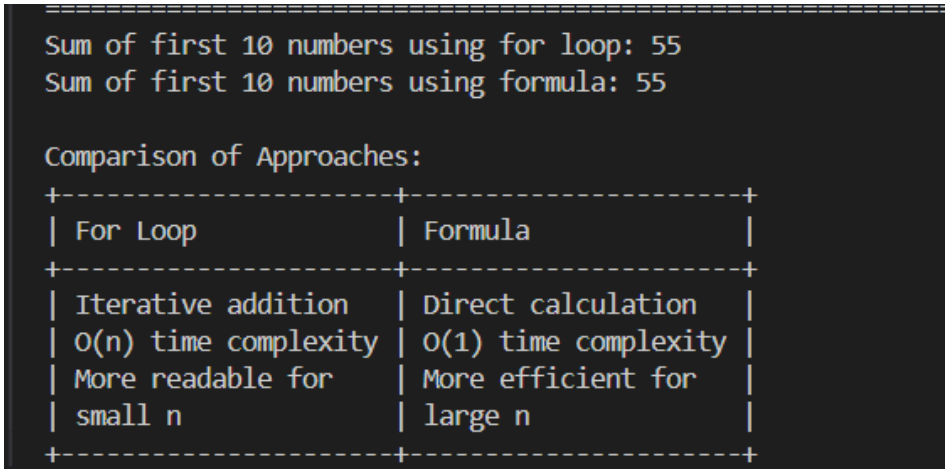
write a code to generate a sum_to_n() to find sum of first n numbers using for loop and another approach and give explanation and comparison of different approaches.

CODE:

```
def sum_to_n_for_loop(n: int) -> int:
    """
    Calculate the sum of the first n natural numbers using a for loop.

    Args:
        n: The upper limit number

    Returns:
        The sum of the first n natural numbers
    """
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
def sum_to_n_formula(n: int) -> int:
    """
    Calculate the sum of the first n natural numbers using the formula n(n + 1)/2.

    Args:
        n: The upper limit number

    Returns:
        The sum of the first n natural numbers
```

```
    """
    return n * (n + 1) // 2
# Example usage
n = 10
print(f"Sum of first {n} numbers using for loop: {sum_to_n_for_loop(n)}")
print(f"Sum of first {n} numbers using formula: {sum_to_n_formula(n)}")
# Comparison and Explanation
print("\nComparison of Approaches:")
print("+----------------------+----------------------+")
print("| For Loop             | Formula              |")
print("+----------------------+----------------------+")
print("| Iterative addition   | Direct calculation   |")
print("| O(n) time complexity | O(1) time complexity |")
print("| More readable for    | More efficient for   |")
print("| small n              | large n              |")
print("+----------------------+----------------------+")
```

OUTPUT:



JUSTIFICATION:

The AI provided both loop-based and formula-based solutions correctly. The loop method is helpful for learning iteration, while the mathematical formula is faster and more efficient for direct computation.

Task Description #5: Classes (Bank Account Class)

Scenario

You are designing a basic banking application.

Task

• Use AI tools to generate a Bank Account class with methods such as deposit(), withdraw(),

and check_balance().

• Analyze the AI-generated class structure and logic.

• Add meaningful comments and explain the working of the code.

Expected Output #5

• Complete Python Bank Account class.

• Demonstration of deposit and withdrawal operations with updated balance.

• Well-commented code with a clear explanation.

PROMPT:

write a code with class methods as deposit(),withdraw(), and check_balance() add meaningful comments and explain working of code.

CODE:

```
"""
Bank Account class with deposit(), withdraw(), and check_balance() methods.
Demonstrates encapsulation and basic banking operations.
"""
class BankAccount:
    """
    Represents a bank account with a balance.
```

```
Supports depositing money, withdrawing money, and checking the current balance.
"""
def __init__(self, owner_name, initial_balance=0.0):
    """
    Initialize a new bank account.
    :param owner_name: Name of the account holder
    :param initial_balance: Starting balance (default 0.0)
    """
    self.owner_name = owner_name
    self.balance = initial_balance  # Store balance as instance attribute
def deposit(self, amount):
    """
    Add money to the account.
    :param amount: Positive amount to deposit
    :return: Updated balance after deposit, or None if amount is invalid
    """
    if amount <= 0:
        print("Deposit failed: Amount must be positive.")
        return None
    self.balance += amount
    print(f"Deposited ₹{amount:.2f}. New balance: ₹{self.balance:.2f}")
    return self.balance
def withdraw(self, amount):
    """
    Remove money from the account if sufficient balance exists.
    :param amount: Positive amount to withdraw
    :return: Updated balance after withdrawal, or None if withdrawal failed
    """
```

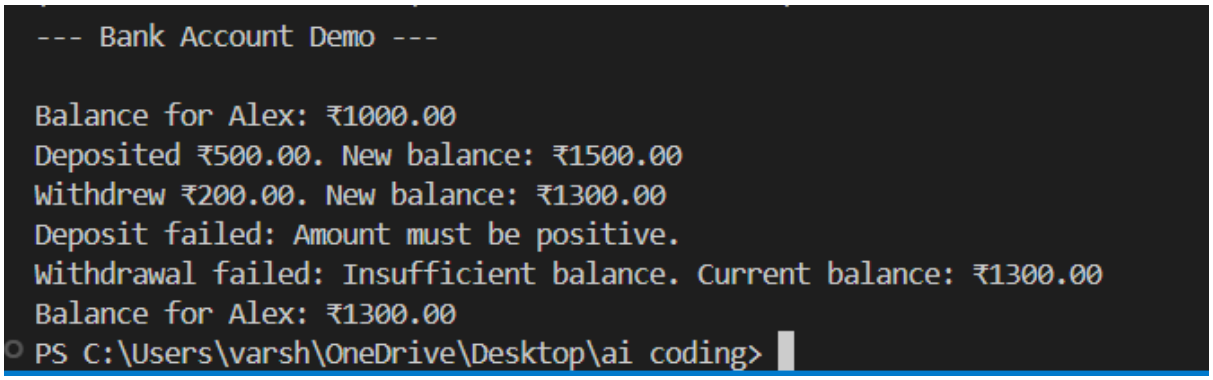
```
        if amount <= 0:
            print("Withdrawal failed: Amount must be positive.")
            return None
        if amount > self.balance:
            print(f"Withdrawal failed: Insufficient balance. Current balance: ₹{self.balance:.2f}")
            return None
        self.balance -= amount
        print(f"Withdrew ₹{amount:.2f}. New balance: ₹{self.balance:.2f}")
        return self.balance
    def check_balance(self):
        """
        Return and display the current account balance.
        :return: Current balance
        """
        print(f"Balance for {self.owner_name}: ₹{self.balance:.2f}")
        return self.balance
# ========== Example usage and explanation ==========
if __name__ == "__main__":
    # Create an account for "Alex" with initial balance ₹1000
    account = BankAccount("Alex", initial_balance=1000.0)
    print("--- Bank Account Demo ---\n")
    # Check initial balance
    account.check_balance()
    # Deposit ₹500
    account.deposit(500.0)
    # Withdraw ₹200
    account.withdraw(200.0)
    # Try invalid operations (negative amount, excess withdrawal)
```

```
    account.deposit(-100)
    account.withdraw(2000.0)
    # Final balance
    account.check_balance()
"""
HOW THE CODE WORKS
==================

1. CLASS BankAccount
  - A class is a blueprint. Each account is an object (instance) of this class.
  - __init__ runs when you create an account. It stores owner_name and balance
    as instance attributes (self.owner_name, self.balance) so every account
    has its own name and balance.

2. deposit(amount)
  - Checks that amount > 0. If not, prints error and returns None.
  - Otherwise adds amount to self.balance and prints the new balance.
  - Returns the updated balance so the caller can use it if needed.

3. withdraw(amount)
  - Checks amount > 0. If not, prints error and returns None.
  - Checks amount <= self.balance. If not, prints "Insufficient balance" and returns None.
  - Otherwise subtracts amount from self.balance, prints new balance, and returns it.

4. check_balance()
  - Does not change balance. Prints and returns the current self.balance.

5. ENCAPSULATION
```

- Balance is stored inside the object (self.balance). External code changes it only through deposit() and withdraw(), so invalid operations (negative amounts, over-withdrawal) can be rejected in one place.

"""

OUTPUT:



JUSTIFICATION:

The AI-generated Bank Account class correctly includes deposit, withdraw, and balance-check methods. It demonstrates real-world object-oriented programming with proper balance updates and basic validation.