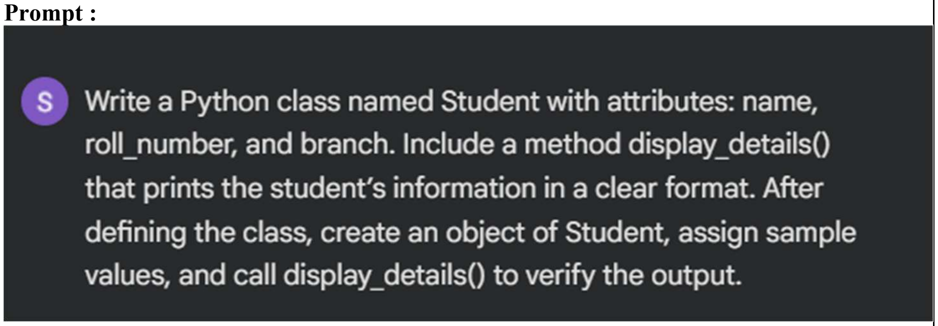
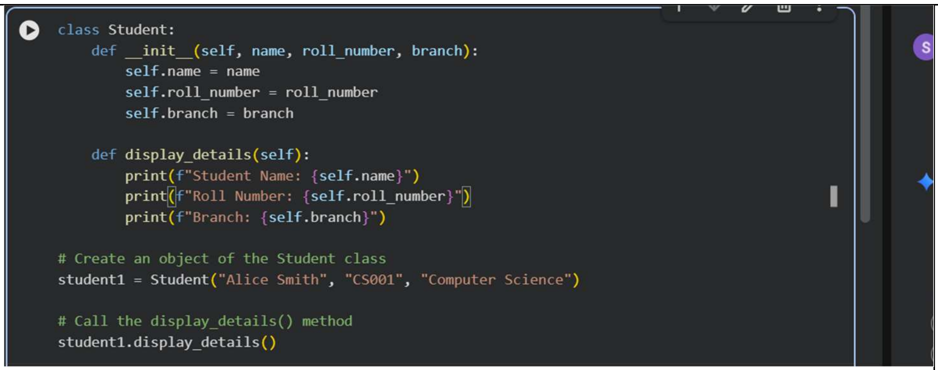
K.Akshay
2303A51817
B-26

Assignment: 6.3

Q.No.	Question	*Expected Time to complete*
1	**Lab 6: AI-Based Code Completion – Classes, Loops, and Conditionals** **Lab Objectives** • To explore AI-powered auto-completion features for core Python constructs such as classes, loops, and conditional statements. • To analyze how AI tools suggest logic for object-oriented programming and control structures. • To evaluate the correctness, readability, and completeness of AI-generated Python code. **Lab Outcomes (LOs)** After completing this lab, students will be able to: • Use AI tools to generate and complete Python class definitions and methods. • Understand and assess AI-suggested loop constructs for iterative tasks. • Generate and evaluate conditional statements using AI-driven prompts. • Critically analyze AI-assisted code for correctness, clarity, and efficiency. **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 :**  S Write a Python class named Student with attributes: name, roll_number, and branch. Include a method display_details() that prints the student's information in a clear format. After defining the class, create an object of Student, assign sample values, and call display_details() to verify the output.	Week3 - Wednesday



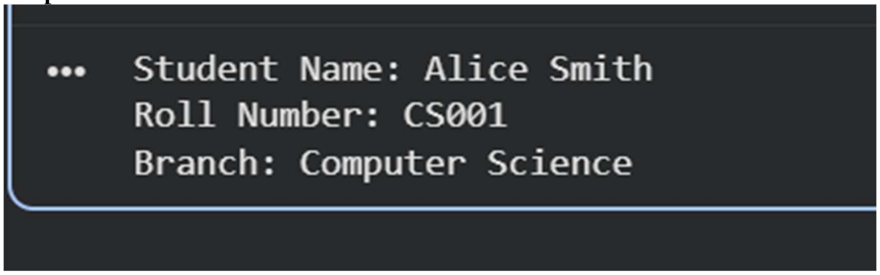
```
class Student:
    def __init__(self, name, roll_number, branch):
        self.name = name
        self.roll_number = roll_number
        self.branch = branch

    def display_details(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")

# Create an object of the Student class
student1 = Student("Alice Smith", "CS001", "Computer Science")

# Call the display_details() method
student1.display_details()
```

Output :



```
Student Name: Alice Smith
Roll Number: CS001
Branch: Computer Science
```

Explanation : Your program defines a simple Student class with attributes for name, roll number, and branch. The constructor initializes these values, and the display_details() method prints them neatly. Creating an object like student1 shows how classes encapsulate data and behavior, producing clear output when the method is called.

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

Scenario

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

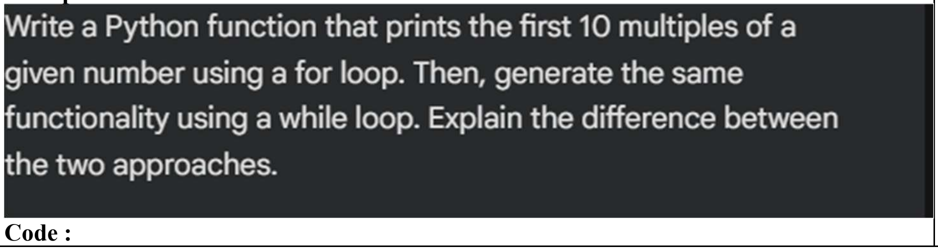
Task

- 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

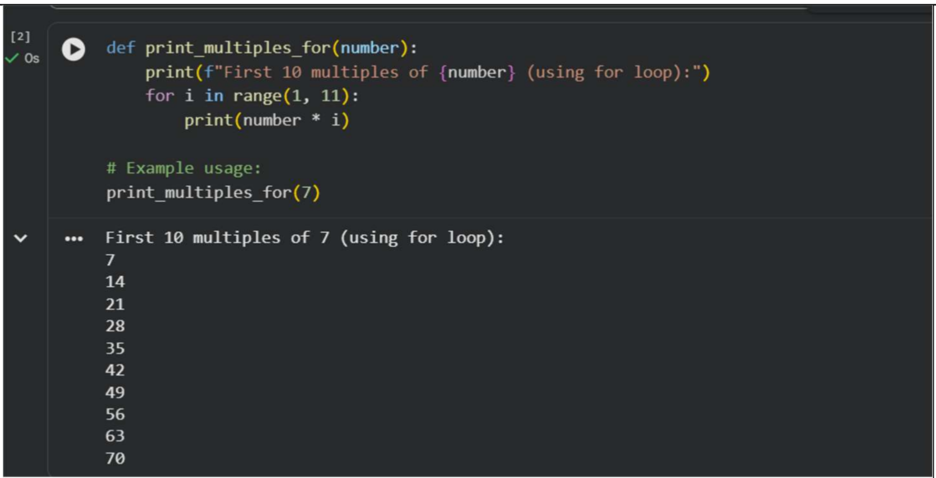
- Correct loop-based Python implementation.
- Output showing the first 10 multiples of a number.
- Comparison and analysis of different looping approaches.

Prompt :



Write a Python function that prints the first 10 multiples of a given number using a for loop. Then, generate the same functionality using a while loop. Explain the difference between the two approaches.

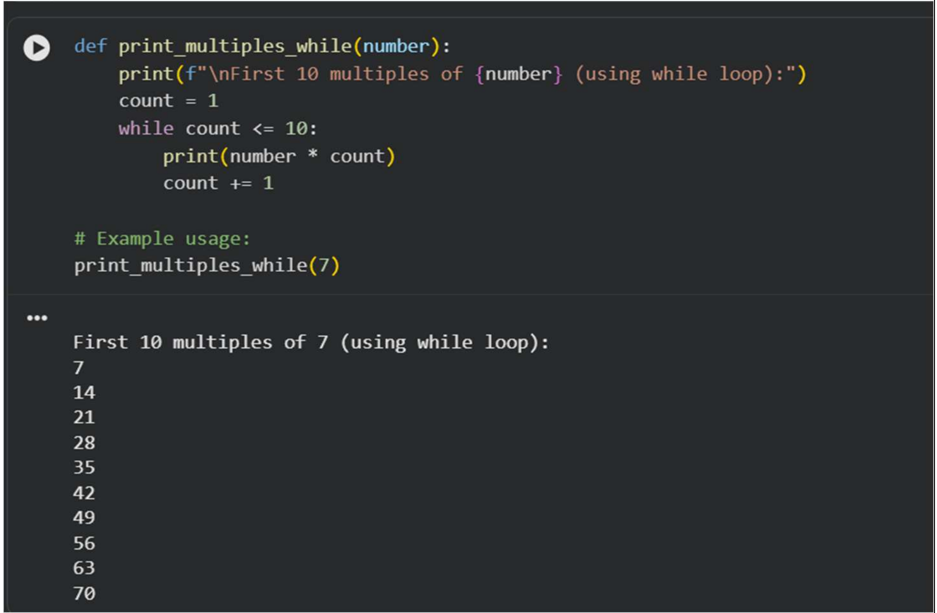
Code :



```
def print_multiples_for(number):
    print(f"First 10 multiples of {number} (using for loop):")
    for i in range(1, 11):
        print(number * i)

# Example usage:
print_multiples_for(7)
```

```
First 10 multiples of 7 (using for loop):
7
14
21
28
35
42
49
56
63
70
```



```
def print_multiples_while(number):
    print(f"\nFirst 10 multiples of {number} (using while loop):")
    count = 1
    while count <= 10:
        print(number * count)
        count += 1

# Example usage:
print_multiples_while(7)
```

```

First 10 multiples of 7 (using while loop):
7
14
21
28
35
42
49
56
63
70
```

Difference between for and while loops:

- for loop: Best used when you know the number of iterations in advance, or when iterating over a sequence (like a list, tuple, string, or range). It automatically handles the iteration variable (e.g., i in range(1, 11)).
- while loop: Best used when the number of iterations is not known beforehand, and the loop continues as long as a certain condition is true. You need to explicitly manage the loop control variable (e.g., count += 1) to avoid infinite loops.

Explanation : A for loop is best when the number of iterations is fixed or when iterating over a sequence—it automatically manages the loop variable (like i in range(1, 11)). A while loop is better when the number of iterations isn't known in advance—it runs as long as a condition is true, but you must manually update the loop variable (like count += 1) to prevent infinite loops.

Task Description #3: Conditional Statements (Age Classification)

Scenario

You are building a basic classification system based on age.

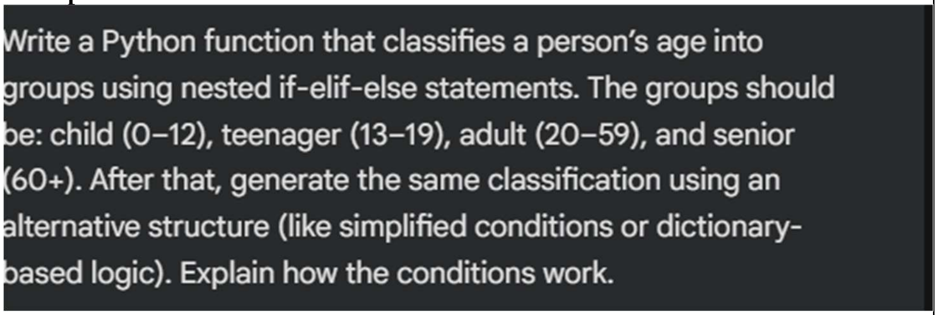
Task

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

Task

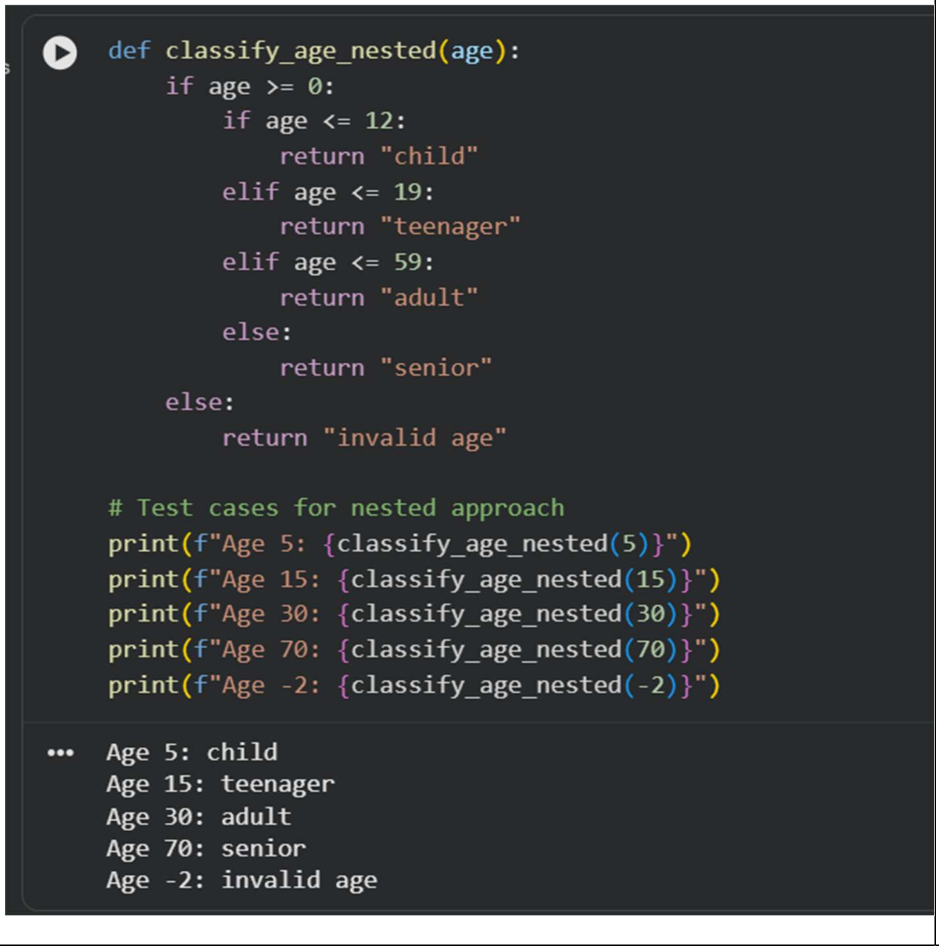
- Ask the AI tool to generate nested if-elif-else conditional statements to classify age groups (e.g., child, teenager, adult, senior).
- Analyze the generated conditions and logic.
- Ask the AI to generate the same classification using alternative conditional structures (e.g., simplified conditions or dictionary-based logic).

Prompt :



Write a Python function that classifies a person’s age into groups using nested if-elif-else statements. The groups should be: child (0–12), teenager (13–19), adult (20–59), and senior (60+). After that, generate the same classification using an alternative structure (like simplified conditions or dictionary-based logic). Explain how the conditions work.

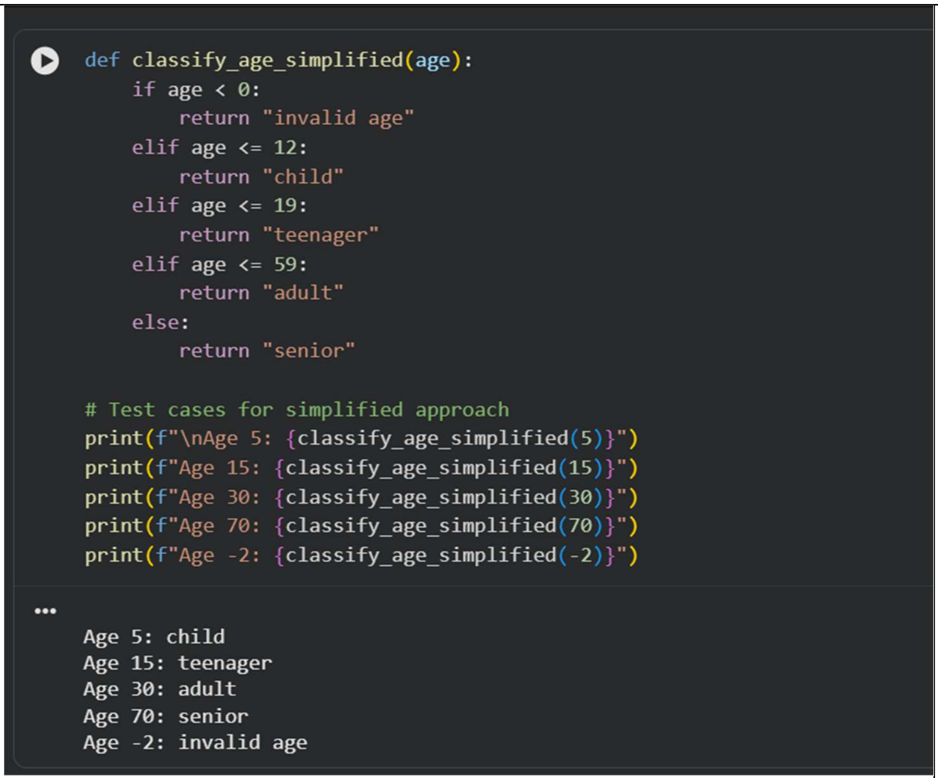
Code :



```
def classify_age_nested(age):
    if age >= 0:
        if age <= 12:
            return "child"
        elif age <= 19:
            return "teenager"
        elif age <= 59:
            return "adult"
        else:
            return "senior"
    else:
        return "invalid age"

# Test cases for nested approach
print(f"Age 5: {classify_age_nested(5)}")
print(f"Age 15: {classify_age_nested(15)}")
print(f"Age 30: {classify_age_nested(30)}")
print(f"Age 70: {classify_age_nested(70)}")
print(f"Age -2: {classify_age_nested(-2)}")
```

```
Age 5: child
Age 15: teenager
Age 30: adult
Age 70: senior
Age -2: invalid age
```



```python
def classify_age_simplified(age):
    if age < 0:
        return "invalid age"
    elif age <= 12:
        return "child"
    elif age <= 19:
        return "teenager"
    elif age <= 59:
        return "adult"
    else:
        return "senior"

# Test cases for simplified approach
print(f"\nAge 5: {classify_age_simplified(5)}")
print(f"Age 15: {classify_age_simplified(15)}")
print(f"Age 30: {classify_age_simplified(30)}")
print(f"Age 70: {classify_age_simplified(70)}")
print(f"Age -2: {classify_age_simplified(-2)}")
```

```

Age 5: child
Age 15: teenager
Age 30: adult
Age 70: senior
Age -2: invalid age
```

Explanation : The **nested if-elif-else approach** first checks if age is valid (non-negative) and then uses step-by-step conditions to classify into child, teenager, adult, or senior. The **simplified approach** removes redundant checks by ordering conditions so that lower bounds are implied by previous failures, making the code shorter and cleaner while still producing the same classifications

---

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.

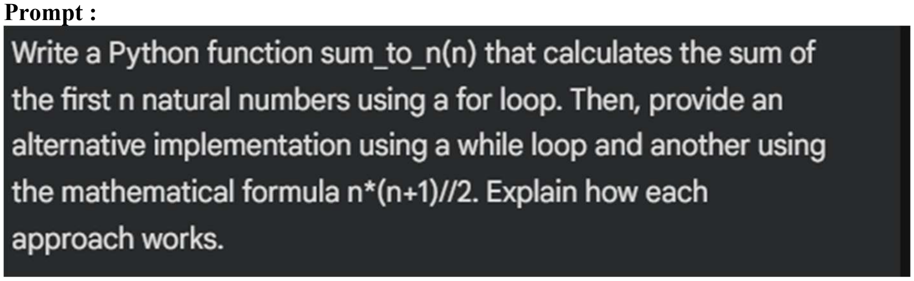
Task

- 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

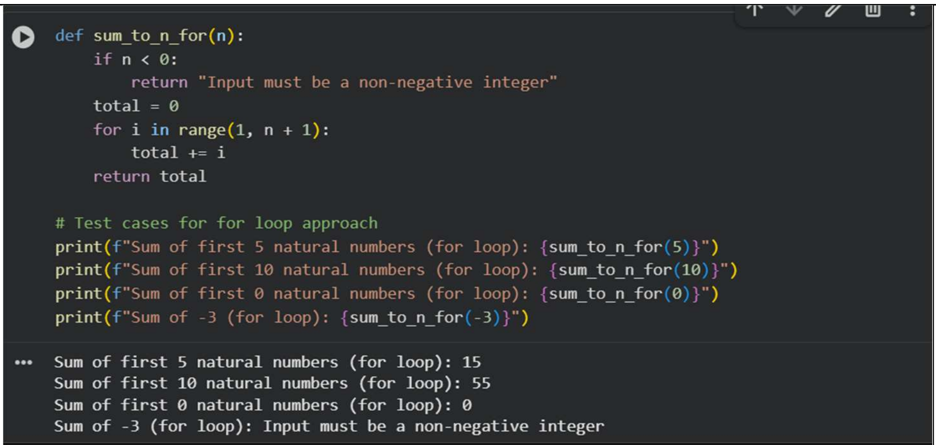
- Python function to compute the sum of first n numbers.
- Correct output for sample inputs.
- Explanation and comparison of different approaches.

Prompt :



Write a Python function sum_to_n(n) that calculates the sum of the first n natural numbers using a for loop. Then, provide an alternative implementation using a while loop and another using the mathematical formula n*(n+1)//2. Explain how each approach works.

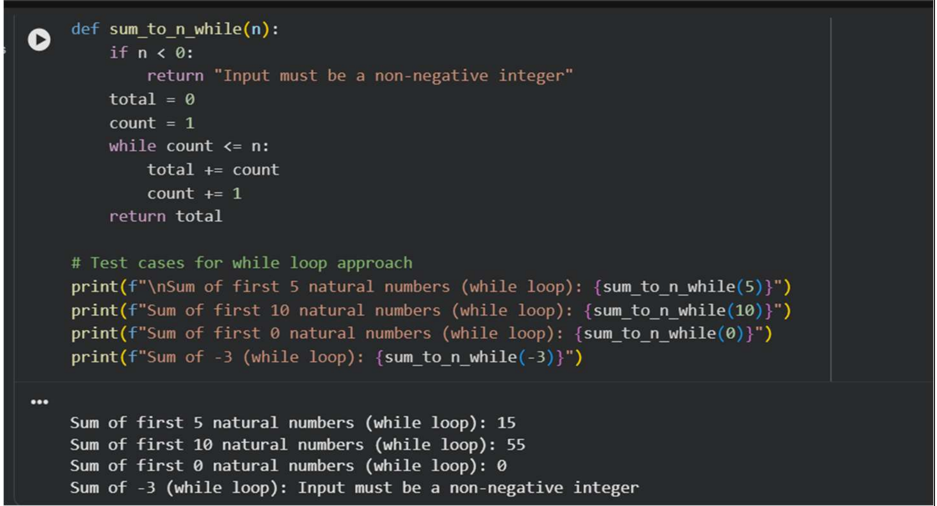
Code :



```
def sum_to_n_for(n):
    if n < 0:
        return "Input must be a non-negative integer"
    total = 0
    for i in range(1, n + 1):
        total += i
    return total

# Test cases for for loop approach
print(f"Sum of first 5 natural numbers (for loop): {sum_to_n_for(5)}")
print(f"Sum of first 10 natural numbers (for loop): {sum_to_n_for(10)}")
print(f"Sum of first 0 natural numbers (for loop): {sum_to_n_for(0)}")
print(f"Sum of -3 (for loop): {sum_to_n_for(-3)}")
```

```
Sum of first 5 natural numbers (for loop): 15
Sum of first 10 natural numbers (for loop): 55
Sum of first 0 natural numbers (for loop): 0
Sum of -3 (for loop): Input must be a non-negative integer
```



```
def sum_to_n_while(n):
    if n < 0:
        return "Input must be a non-negative integer"
    total = 0
    count = 1
    while count <= n:
        total += count
        count += 1
    return total

# Test cases for while loop approach
print(f"\nSum of first 5 natural numbers (while loop): {sum_to_n_while(5)}")
print(f"Sum of first 10 natural numbers (while loop): {sum_to_n_while(10)}")
print(f"Sum of first 0 natural numbers (while loop): {sum_to_n_while(0)}")
print(f"Sum of -3 (while loop): {sum_to_n_while(-3)}")
```

```

Sum of first 5 natural numbers (while loop): 15
Sum of first 10 natural numbers (while loop): 55
Sum of first 0 natural numbers (while loop): 0
Sum of -3 (while loop): Input must be a non-negative integer
```

Explanation : The sum of the first *n* natural numbers can be calculated in three ways. Using a for loop, the function iterates from 1 to *n*, adding each number to a running total, which is straightforward when the number of iterations is known. Using a while loop, the same result is achieved by maintaining a counter that starts at 1 and increments until it reaches *n*, requiring explicit control of the loop variable. Finally, the mathematical formula n\cdot (n+1)//2 provides the most efficient solution, as it computes the sum directly in constant time without looping, making it ideal for large values of *n*.

Task Description #5: Classes (Bank Account Class)

Scenario

You are designing a basic banking application.

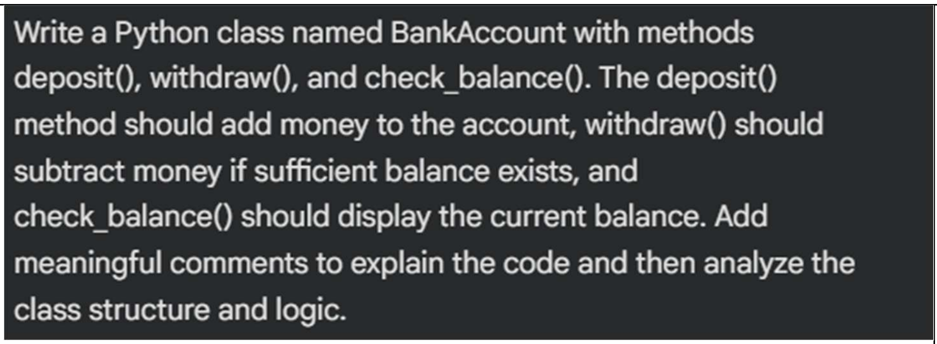
Task

- 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

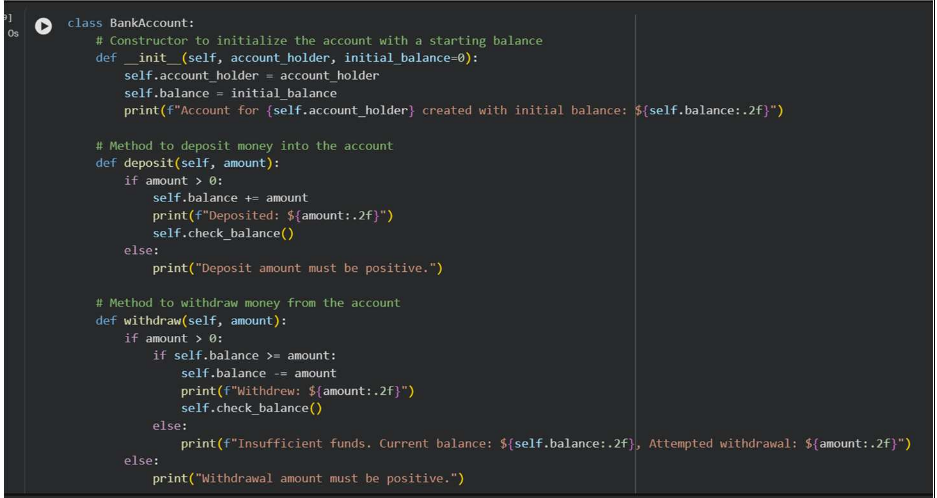
- Complete Python Bank Account class.
- Demonstration of deposit and withdrawal operations with updated balance.
- Well-commented code with a clear explanation.

Prompt :



Write a Python class named BankAccount with methods deposit(), withdraw(), and check_balance(). The deposit() method should add money to the account, withdraw() should subtract money if sufficient balance exists, and check_balance() should display the current balance. Add meaningful comments to explain the code and then analyze the class structure and logic.

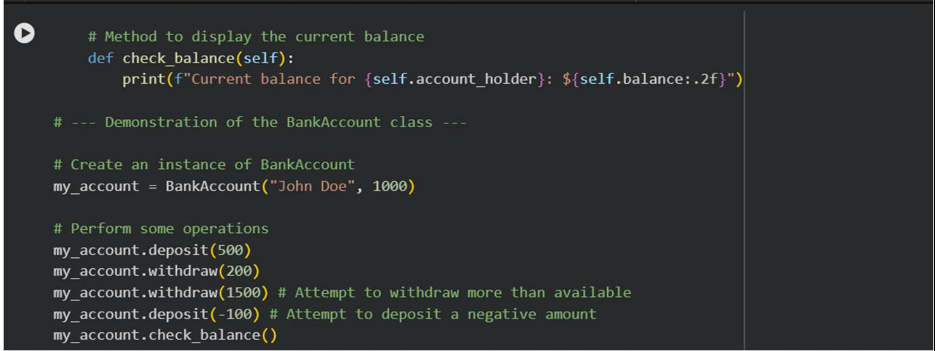
Code :



```
class BankAccount:
    # Constructor to initialize the account with a starting balance
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.balance = initial_balance
        print(f"Account for {self.account_holder} created with initial balance: ${self.balance:.2f}")

    # Method to deposit money into the account
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited: ${amount:.2f}")
            self.check_balance()
        else:
            print("Deposit amount must be positive.")

    # Method to withdraw money from the account
    def withdraw(self, amount):
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                print(f"Withdrew: ${amount:.2f}")
                self.check_balance()
            else:
                print(f"Insufficient funds. Current balance: ${self.balance:.2f}, Attempted withdrawal: ${amount:.2f}")
        else:
            print("Withdrawal amount must be positive.")
```



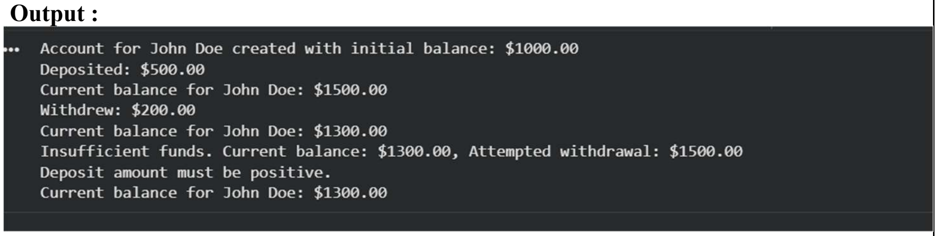
```
    # Method to display the current balance
    def check_balance(self):
        print(f"Current balance for {self.account_holder}: ${self.balance:.2f}")

# --- Demonstration of the BankAccount class ---

# Create an instance of BankAccount
my_account = BankAccount("John Doe", 1000)

# Perform some operations
my_account.deposit(500)
my_account.withdraw(200)
my_account.withdraw(1500) # Attempt to withdraw more than available
my_account.deposit(-100) # Attempt to deposit a negative amount
my_account.check_balance()
```

Output :



```
Account for John Doe created with initial balance: $1000.00
Deposited: $500.00
Current balance for John Doe: $1500.00
Withdrew: $200.00
Current balance for John Doe: $1300.00
Insufficient funds. Current balance: $1300.00, Attempted withdrawal: $1500.00
Deposit amount must be positive.
Current balance for John Doe: $1300.00
```

Explanation : The BankAccount class shows basic OOP in Python by combining data (account holder, balance) with behavior (deposit, withdraw, check_balance). The constructor sets up the account, deposits add funds if valid, withdrawals check for sufficient balance, and check_balance displays the current state. It ensures input validation, prevents overdrafts, and keeps account management organized and reusable.