k.Harini sri
2303A51602
Batch: 25

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.
1 • Critically analyze AI-assisted code for correctness, clarity, and efficiency. Week3 - Wednesday

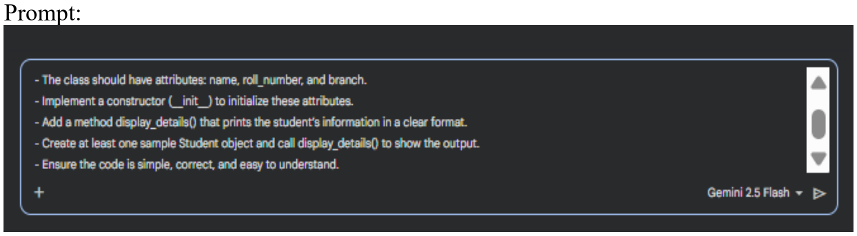
Task Description #1: Classes (Student Class)

Scenario
You are developing a simple student information management module.

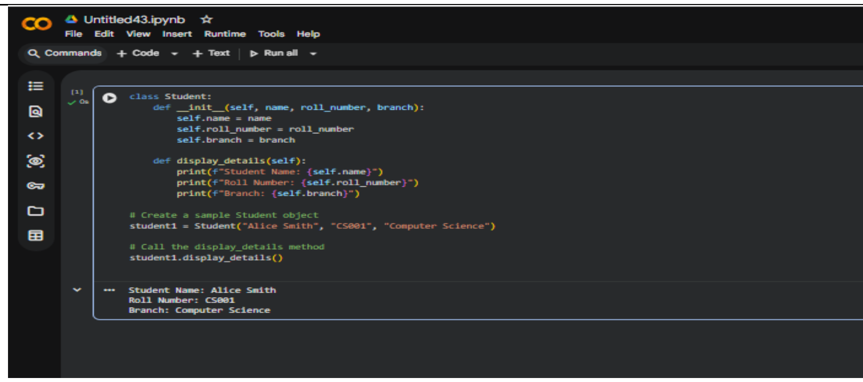
Task
• Use an AI tool (GitHub Copilot / Cursor AI / Gemini) to complete a Student class.
• The class should include attributes such as name, roll number, and branch.
• Add a method display_details() to print student information.
• Execute the code and verify the output.
• Analyze the code generated by 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.

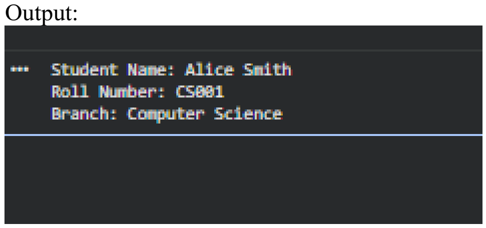
Prompt:



Code:



Output:



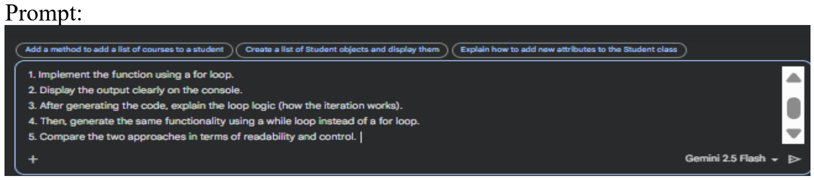
Task Description #2: Loops (Multiples of a Number)

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:



Code for for loop:

Output:

```
Multiples of 7:
7
14
21
28
35
42
49
56
63
70
```

Code for while loop:

```
def print_multiples_while_loop(number):
    print(f"Multiples of {number} (using while loop):")
    count = 1
    while count <= 10:
        multiple = number * count
        print(multiple)
        count += 1

# Call the function with a sample number (7)
print_multiples_while_loop(7)
```

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

Output:

```
... Multiples of 7 (using while loop):
    7
    14
    21
    28
    35
    42
    49
    56
    63
    70
```

Comparision of both loops:
Compare loop approaches
Subtask:
Compare the for loop and while loop implementations in terms of their readability, control, and suitability for this specific task.
Both for and while loops can achieve the task of printing the first 10 multiples of a number, but they differ in their structure and typical use cases:
Readability

- **for loop (print_multiples_for_loop)**: For iterating a fixed number of times or over a sequence, the for loop is generally more readable. The range(1, 11) clearly indicates that the loop will run 10 times, from 1 up to 10. The iteration variable i directly represents the multiplier.
- **while loop (print_multiples_while_loop)**: While functional, the while loop requires explicit management of the counter (count = 1, count <= 10, count += 1). This adds a bit more boilerplate code and mental overhead to understand the loop's exact iteration count compared to the for loop's range function.

Control

- **for loop**: Control is handled implicitly by the range object. It initializes i at 1, checks if it's less than 11, and increments i automatically after each iteration. This concise control makes it ideal for definite iteration.
- **while loop**: Control is explicit. You must initialize count before the loop, define the termination condition (count <= 10), and manually increment count within the loop

body. Forgetting to increment count would result in an infinite loop.

Suitability for this specific task

- For tasks that involve iterating a **known number of times** (like printing the first 10 multiples), the for loop, especially with range(), is generally the more idiomatic and preferred choice in Python. It's designed for definite iteration.
- The while loop is better suited for **indefinite iteration**, where the number of repetitions is not known beforehand and depends on a condition being met (e.g., looping until a user enters 'quit', or processing items from a queue until it's empty).

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:

Implement the `for` loop function for printing multiples | Write the `while` loop function for printing multiples

Write a Python program that classifies people into age groups.

Requirements:
1. Create a function classify_age(age) that uses nested if-elif-else statements to categorize:
  - Child (0–12)

Implement the `for` loop function for printing multiples | Write the `while` loop function for printing multiples

  - Teenager (13–19)
  - Adult (20–59)
  - Senior (60 and above)
2. Print the classification result clearly.
3. After generating the code, explain how the conditional logic works.

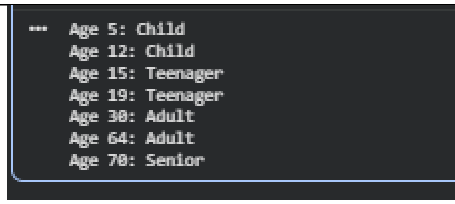
Code:

```python
def classify_age_if_elif_else(age):
    if age <= 12:
        return 'Child'
    elif age <= 19:
        return 'Teenager'
    elif age <= 64:
        return 'Adult'
    else:
        return 'Senior'

# Demonstrate the function with examples
print(f"Age 5: {classify_age_if_elif_else(5)}")
print(f"Age 12: {classify_age_if_elif_else(12)}")
print(f"Age 15: {classify_age_if_elif_else(15)}")
print(f"Age 19: {classify_age_if_elif_else(19)}")
print(f"Age 30: {classify_age_if_elif_else(30)}")
print(f"Age 64: {classify_age_if_elif_else(64)}")
print(f"Age 70: {classify_age_if_elif_else(70)}")
```

```
Age 5: Child
Age 12: Child
Age 15: Teenager
Age 19: Teenager
Age 30: Adult
Age 64: Adult
Age 70: Senior
```

Output:


```
Age 5: Child
Age 12: Child
Age 15: Teenager
Age 19: Teenager
Age 30: Adult
Age 64: Adult
Age 70: Senior
```

Condition explanation:
Provide a clear explanation of the conditional logic within the classify_age_if_elif_else function, describing how each condition is evaluated and how the age groups are determined.
The classify_age_if_elif_else function categorizes an age into one of four groups ('Child', 'Teenager', 'Adult', 'Senior') using a sequence of if-elif-else statements. The key aspect of this structure is that conditions are evaluated sequentially, and only the *first* condition that evaluates to True is executed, after which the function returns.

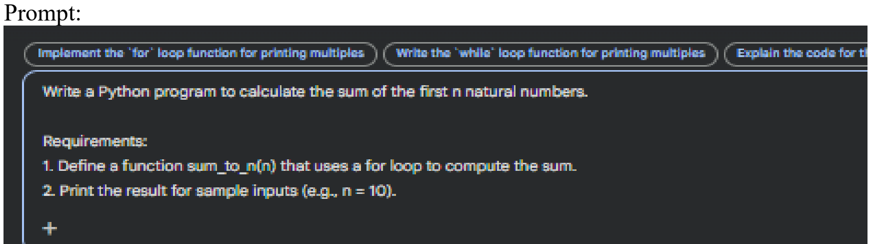
1. **if age <= 12:**: This is the initial check. If the input age is 12 or less, the function immediately returns 'Child'. All subsequent elif and else blocks are skipped.
2. **elif age <= 19:**: This condition is only evaluated if the first if condition (age <= 12) was False. This means that at this point, we know age is greater than 12. If age is now found to be 19 or less (and implicitly, greater than 12), the function returns 'Teenager'.
3. **elif age <= 64:**: This condition is only reached if both the if and the first elif conditions were False. Therefore, we know age is greater than 19. If age is now found to be 64 or less (and implicitly, greater than 19), the function returns 'Adult'.
4. **else:**: This else block is executed only if all preceding if and elif conditions were False. This means age is not <= 12, not <= 19, and not <= 64. Consequently, age must be greater than 64, and the function returns 'Senior'.

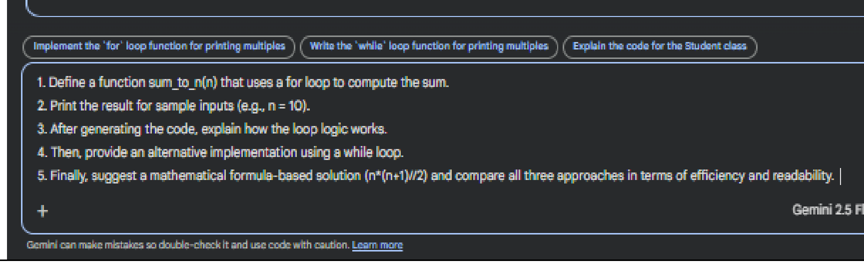
Task Description #4: For and While Loops (Sum of First n Numbers)

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:





Code for forloop:

```
def sum_to_n_for_loop(n):
    total_sum = 0
    for i in range(1, n + 1):
        total_sum += i
    return total_sum

# Demonstrate the function with examples
num_1 = 10
print(f"Sum of first {num_1} natural numbers: {sum_to_n_for_loop(num_1)}")

num_2 = 5
print(f"Sum of first {num_2} natural numbers: {sum_to_n_for_loop(num_2)}")

Sum of first 10 natural numbers: 55
Sum of first 5 natural numbers: 15
```

Output:

```
Sum of first 10 natural numbers: 55
Sum of first 5 natural numbers: 15
```

Code for while loop using mathematical formula:

```
def sum_to_n_while_loop(n):
    total_sum = 0
    i = 1
    while i <= n:
        total_sum += i
        i += 1
    return total_sum

# Demonstrate the function with examples
num_1 = 10
print(f"Sum of first {num_1} natural numbers (using while loop): {sum_to_n_while_loop(num_1)}")

num_2 = 5
print(f"Sum of first {num_2} natural numbers (using while loop): {sum_to_n_while_loop(num_2)}")

Sum of first 10 natural numbers (using while loop): 55
Sum of first 5 natural numbers (using while loop): 15
```

Output:

```
Sum of first 10 natural numbers (using while loop): 55
Sum of first 5 natural numbers (using while loop): 15
```

Comparision of both approaches:

- **Readability**: The for loop is generally more readable for definite iteration due to its concise syntax (range(1, n + 1)). The while loop is less concise as it requires explicit counter management. The formula is highly concise but assumes familiarity with the underlying mathematical principle.
- **Control**: The for loop offers implicit control over iteration (initialization, condition, increment) via the range object, making it less prone to errors. The while loop demands explicit management of these aspects, increasing the risk of errors like infinite loops if not handled carefully.
- **Efficiency**: The for loop and while loop approaches both have a time complexity of O(n), meaning their execution time scales linearly with n. In contrast, the formula-based approach has a time complexity of O(1), as it performs a constant number of operations regardless of n, making it significantly more efficient for large values of n.
- **Suitability**: For calculating the sum of the first n natural numbers, the formula-based method is the most suitable and efficient choice due to its O(1) complexity. The for loop is a very suitable and Pythonic choice for demonstrating definite iteration and general summing tasks where the number of iterations is known, while the while loop is generally less suitable for definite iteration but better for indefinite iteration.

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:

Implement the `for` loop function for printing multiples | Write the `while` loop function for printing multiples

Write a Python program that defines a BankAccount class.

Requirements:
1. The class should have an attribute balance initialized in the constructor (__init__).
2. Implement the following methods:

+

Implement the `for` loop function for printing multiples | Write the `while` loop function for printing multiples | Explain the code for the Student class

2. Implement the following methods:
- deposit(amount): adds money to the balance.
- withdraw(amount): subtracts money if sufficient balance exists, otherwise print an error message.
- check_balance(): prints the current balance.
3. Demonstrate the class by creating a BankAccount object, performing deposit and withdrawal operations, and displaying the updated balance.

+ Gemini 2.5 Fl

- withdraw(amount): subtracts money if sufficient balance exists, otherwise print an error message.
- check_balance(): prints the current balance.
3. Demonstrate the class by creating a BankAccount object, performing deposit and withdrawal operations, and displaying the updated balance.
4. Add meaningful comments explaining each part of the code.
5. After generating the code, provide a brief analysis of the class structure and logic for clarity.

Code:

```python
class BankAccount:
    # Constructor to initialize the bank account with an initial balance
    def __init__(self, initial_balance=0):
        if initial_balance < 0:
            print("Error: Initial balance cannot be negative. Setting to 0.")
            self.balance = 0
        else:
            self.balance = initial_balance
        print(f"Account 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}. New balance: ${self.balance:.2f}")
        else:
            print("Error: Deposit amount must be positive.")

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

    # Method to check and print the current balance
    def check_balance(self):
        print(f"Current balance: ${self.balance:.2f}")

# --- Demonstrate the BankAccount class ---

# 1. Create a BankAccount object
print("\n--- Creating Account ---")
my_account = BankAccount(100.00)
my_account.check_balance()

# 2. Perform deposit operations
print("\n--- Deposit Operations ---")
my_account.deposit(50.75)
my_account.deposit(0) # Attempt invalid deposit
my_account.check_balance()

# 3. Perform withdrawal operations
print("\n--- Withdrawal Operations ---")
my_account.withdraw(30.50)
my_account.withdraw(200.00) # Attempt invalid withdrawal
my_account.withdraw(-10) # Attempt invalid withdrawal amount
my_account.check_balance()

# 4. Final balance check
print("\n--- Final Check ---")
my_account.check_balance()
```

Output:

```
--- Creating Account ---
Account created with initial balance: $100.00
Current balance: $100.00

--- Deposit Operations ---
Deposited: $50.75. New balance: $150.75
Error: Deposit amount must be positive.
Current balance: $150.75

--- Withdrawal Operations ---
Withdrew: $30.50. New balance: $120.25
Error: Insufficient balance. Current balance: $120.25. Attempted withdrawal: $200.00
Error: Withdrawal amount must be positive.
Current balance: $120.25

--- Final Check ---
Current balance: $120.25
```

Code explanation:

The given Python program defines a **BankAccount class** that models a simple banking system with essential operations such as depositing, withdrawing, and checking balance. The constructor (__init__) initializes the account with an optional starting balance, ensuring that negative balances are not allowed by resetting them to zero. The deposit() method adds money to the account only if the amount is positive, while the withdraw() method deducts money if the withdrawal amount is valid and does not exceed the current balance, otherwise it prints appropriate error messages. The check_balance() method displays the current balance in a formatted way. In the demonstration section, an account is created with an initial balance, followed by deposit and withdrawal operations, including invalid transactions to test error handling. Throughout execution, the program provides clear feedback messages, making the logic easy to follow and ensuring that the account balance is updated correctly after each operation.

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.