AI-Assisted Coding

Name: P. Archith Roll Number: 2303A52258 Batch: 45

AssignmentNumber:6.3(Present assignment number)/**24**(Total number of assignments)

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: Create a Python code-class named Student.
The class should have attributes: name, roll_number, and branch.
Implement a constructor (__init__) to initialize these attributes using dynamic user input.
Add a method display_details() that prints the student information in a readable format.
Create an object of the class, call the method, and display the output.
Ensure the code is simple, readable, and correct.

Code:

```python
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("\n--- Student Details ---")
        print("Name       :", self.name)
        print("Roll No    :", self.roll_number)
        print("Branch     :", self.branch)
# Dynamic input from user
name = input("Enter student name: ")
roll_number = input("Enter roll number: ")
branch = input("Enter branch: ")
# Object creation
student1 = Student(name, roll_number, branch)
# Display details
student1.display_details()
```

Output:

Justification: **Correctness**

- The class correctly defines attributes using the constructor.
- Dynamic input ensures flexibility for multiple users.
- The display_details() method prints all required information correctly.

Readability

- Code is well-structured and easy to understand.
- Meaningful method and variable names are used.
- Output formatting improves clarity.

Completeness

- Includes class definition, object creation, and method invocation.
- Fully executable without modifications.
- Meets all task requirements mentioned in the lab.

Efficiency

- Uses simple and efficient logic.
- Suitable for beginner-level object-oriented programming tasks.

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 code that accepts a number as dynamic input and prints the first 10 multiples of that number using a for loop.
Ensure the code is readable, correct, and displays the output clearly.

Code: For Loop

```
def print_multiples_for(num):
    print(f"\nFirst 10 multiples of {num} (using for loop):")
    for i in range(1, 11):
        print(num * i)
# Dynamic input
number = int(input("Enter a number: "))
# Function call
print_multiples_for(number)
```

Output:

```
itant Coding\Task-6.5.py'
Enter a number: 15

First 10 multiples of 15 (using for loop):
15
30
45
60
75
90
105
120
135
150
PS D:\3-2\AI Assitant Coding>
```

Analysis of for Loop Logic

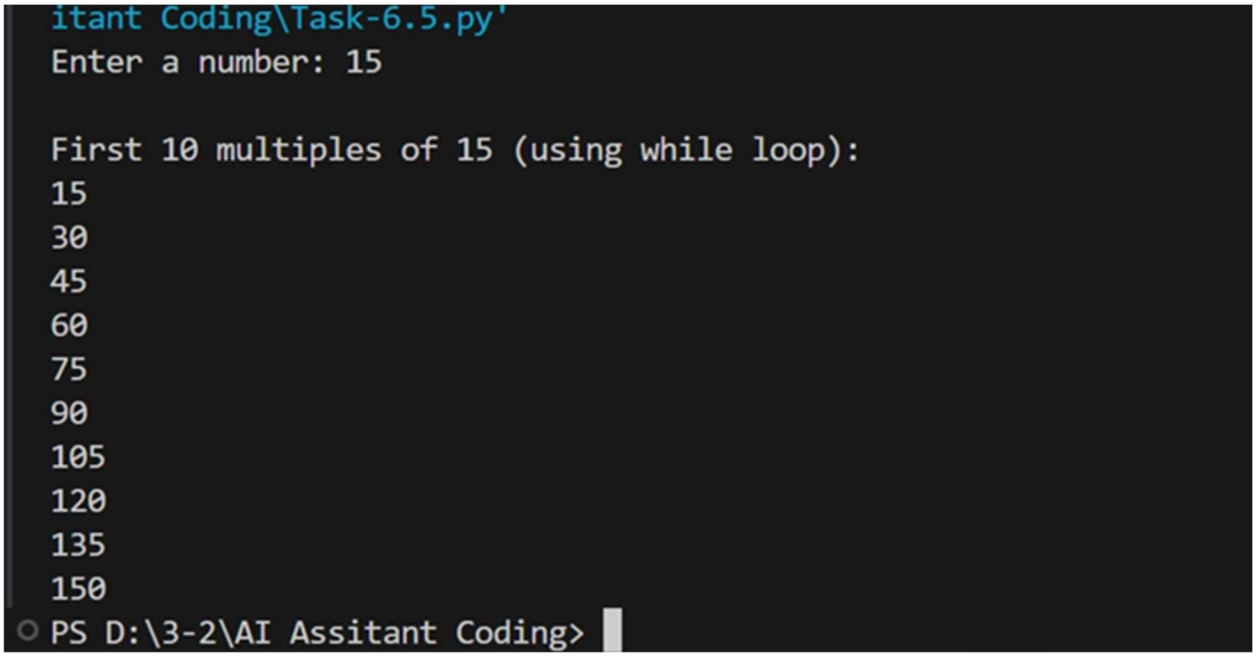
- The range (1, 11) generates values from 1 to 10.
- Each iteration multiplies the input number by the loop counter.
- The loop is concise and ideal when the number of iterations is known in advance.
- Readability and simplicity make it suitable for beginners.

Prompt: Generate a Python function that prints the first 10 multiples of a given number using a while loop.
Use dynamic input and ensure correct loop termination.

Code:

```
"""While Loop"""
def print_multiples_while(num):
    print(f"\nFirst 10 multiples of {num} (using while loop):")
    i = 1
    while i <= 10:
        print(num * i)
        i += 1
# Dynamic input
number = int(input("Enter a number: "))
# Function call
print_multiples_while(number)
```

Output:



Justification:

Comparison and Analysis of Looping Approaches

Aspect	**For Loop**	**While Loop**
Loop Control	Automatic	Manual
Readability	High	Moderate
Best Use Case	Fixed number of iterations	Condition-based repetition
Risk of Error	Low	Higher (if counter not updated)

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: Write a Python Code that takes age as dynamic user input and classifies the person into age groups such as child, teenager, adult, or senior using nested if-elif-else statements.
Ensure the logic is correct, readable, and handles valid age ranges.

Code:

```
"""2303A52324"""
"""If--Elif-Else Statements"""

def classify_age(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 Citizen"
# Dynamic input
age = int(input("Enter age: "))
# Classification
result = classify_age(age)
print("Age Group:", result)
```

Output:

```
itant Coding\Task-6.5.py'
Enter age: 20
Age Group: Adult
itant Coding\Task-6.5.py'
Enter age: 60
Age Group: Senior Citizen
PS D:\3-2\AI Assitant Coding>
```

Analysis of Conditional Logic

- The conditions are checked **top-down**, ensuring correct classification.
- Age validation (age < 0) prevents invalid input.
- Each elif defines a clear age range without overlap.
- The structure is readable and easy to modify.

Prompt: Generate a Python Code age classification program using simplified conditional logic or a dictionary-based approach instead of multiple elif statements.
Use dynamic input and ensure clarity and correctness.

Code:

```
"""2303A52324"""
"""Multiple Elif Statements"""
def classify_age_simple(age):
    if age < 0:
        return "Invalid age"
    categories = {
        (0, 12): "Child",
        (13, 19): "Teenager",
        (20, 59): "Adult",
        (60, 150): "Senior Citizen"
    }
    for age_range, group in categories.items():
        if age_range[0] <= age <= age_range[1]:
            return group
# Dynamic input
age = int(input("Enter age: "))
# Classification
print("Age Group:", classify_age_simple(age))
```

Output:

Justification:

Aspect	if-elif-else	Dictionary-Based
Readability	Very high	Moderate
Scalability	Less flexible	More flexible
Best for	Small logic	Larger rule sets
Beginner Friendly	Yes	Intermediate

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 Code for sum_to_n() that takes a positive integer n as dynamic user input and calculates the sum of the first n natural numbers using a for loop.
Ensure the code is readable, correct, and handles basic input validation.

Code:

```
"""2303A52324"""
"""For Loop"""
def sum_to_n_for(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
# Dynamic input
n = int(input("Enter a positive integer: "))
if n > 0:
    print("Sum of first", n, "natural numbers (for loop):", sum_to_n_for(n))
else:
    print("Invalid input. Please enter a positive number.")
```

Output:

```
itant Coding\Task-6.5.py'
Enter a positive integer: 12
Sum of first 12 natural numbers (for loop): 78
PS D:\3-2\AI Assitant Coding>
```

Analysis of for Loop Implementation

- The loop iterates from 1 to n using range(1, n+1).
- Each number is added to total, accumulating the sum.
- Simple and readable logic.
- Best suited when the number of iterations is known.

Prompt: Generate an alternative implementation of the sum_to_n() function using a while loop or a mathematical formula.
Compare the logic and efficiency with the for loop version.

Code:

```
"""2303A52324"""
"""While Loop"""
def sum_to_n_while(n):
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total
# Dynamic input
n = int(input("Enter a positive integer: "))
if n > 0:
    print("Sum of first", n, "natural numbers (while loop):", sum_to_n_while(n))
else:
    print("Invalid input. Please enter a positive number.")
```

Output:

```
itant Coding\Task-6.5.py'
Enter a positive integer: 15
Sum of first 15 natural numbers (while loop): 120
PS D:\3-2\AI Assitant Coding>
```

Code:

```
"""2303A52324"""
"""Mathematical Formula"""
def sum_to_n_formula(n):
    return n * (n + 1) // 2
# Dynamic input
n = int(input("Enter a positive integer: "))
if n > 0:
    print("Sum of first", n, "natural numbers (formula):", sum_to_n_formula(n))
else:
    print("Invalid input. Please enter a positive number.")
```

Output:

```
itant Coding\Task-6.5.py'
Enter a positive integer: 16
Sum of first 16 natural numbers (formula): 136
PS D:\3-2\AI Assitant Coding>
```

Justification: **Comparison of Different Approaches**

Approach	**Logic Used**	**Time Complexity**	**Readability**
For Loop	Iterative addition	O(n)	High
While Loop	Condition-based iteration	O(n)	Moderate
Formula	Direct computation	O(1)	Very High

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.

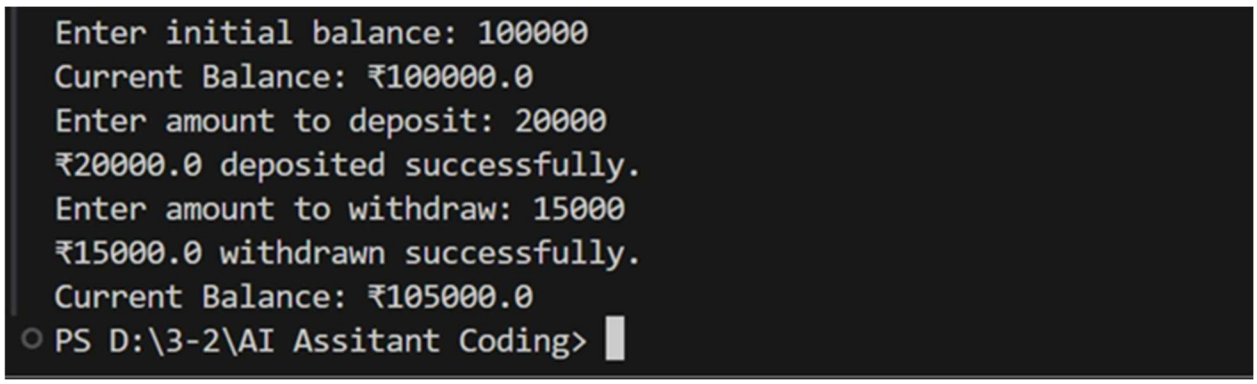
Expected Output #5
• Complete Python Bank Account class.
• Demonstration of deposit and withdrawal operations with updated balance.
• Well-commented code with a clear explanation.

Prompt: Create a Python Code-class named BankAccount.
The class should have attributes for account holder name and balance.
Implement methods deposit(), withdraw(), and check_balance().
Use dynamic user input, ensure input validation, and add meaningful comments.
Demonstrate deposit and withdrawal operations with updated balance output.

Code:

```
"""2303A52324"""
class BankAccount:
    def __init__(self, holder_name, initial_balance):
        # Initialize account holder name and balance
        self.holder_name = holder_name
        self.balance = initial_balance
    def deposit(self, amount):
        # Add money to the account if amount is valid
        if amount > 0:
            self.balance += amount
            print(f"₹{amount} deposited successfully.")
        else:
            print("Deposit amount must be positive.")
    def withdraw(self, amount):
        # Withdraw money if sufficient balance is available
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print("Insufficient balance.")
        else:
            self.balance -= amount
            print(f"₹{amount} withdrawn successfully.")
    def check_balance(self):
        # Display the current balance
        print(f"Current Balance: ₹{self.balance}")
# Dynamic input from user
name = input("Enter account holder name: ")
initial_balance = float(input("Enter initial balance: "))
# Create BankAccount object
account = BankAccount(name, initial_balance)
# Perform operations
account.check_balance()
deposit_amount = float(input("Enter amount to deposit: "))
account.deposit(deposit_amount)
withdraw_amount = float(input("Enter amount to withdraw: "))
account.withdraw(withdraw_amount)
# Final balance
account.check_balance()
```

Output:



Analysis of AI-Generated Class Structure:

Class Design

- Encapsulates data (holder_name, balance) within a class.
- Methods operate directly on object data, following OOP principles.

Method Logic

- deposit() validates positive amounts before updating balance.
- withdraw() checks both validity and sufficient balance.
- check_balance() provides a clear account summary.

Correctness

- Prevents invalid transactions.
- Ensures balance consistency after every operation.