AI ASSISTANT CODING
ASSIGNMENT-6.5

Task: Use an AI tool to generate eligibility logic.
Prompt:
“Generate Python code to check voting eligibility based on age and citizenship**.”**
Expected Output:
• AI-generated conditional logic.
• Correct eligibility decisions.
• Explanation of conditions**.**

Task Description #2 (AI-Based Code Completion for Loop-Based String Processing)
Task: Use an AI tool to process strings using loops.
Prompt:
“Generate Python code to count vowels and consonants in a string using a loop.**”**
Expected Output:
• AI-generated string processing logic.
• Correct counts.
• Output verification.

Task Description #3 (AI-Assisted Code Completion Reflection Task)
Task: Use an AI tool to generate a complete program using classes, loops, and conditionals.
Prompt:
“Generate a Python program for a library management system using classes, loops, and conditional statements**.”**
Expected Output:
• Complete AI-generated program.
• Review of AI suggestions quality.
• Short reflection on AI-assisted coding experience.

Task Description #4 (AI-Assisted Code Completion for Class-Based Attendance System)
Task: Use an AI tool to generate an attendance management class.
Prompt:
"Generate a Python class to mark and display student attendance using loops**."**
Expected Output:
- AI-generated attendance logic.
- Correct display of attendance.
- Test cases.

Task Description #5 (AI-Based Code Completion for Conditional Menu Navigation**)**
Task: Use an AI tool to complete a navigation menu.
Prompt:
"Generate a Python program using loops and conditionals to simulate an ATM menu**."**
Expected Output:
- AI-generated menu logic.
- Correct option handling.
- Output verification**.**

Task 1 : AI-Based Code Completion for Conditional Eligibility Check

Task : Use an AI tool to generate eligibility logic.

Prompt:

Generate Python code to count vowels and consonants in a string using a loop.

Suggested Code :

```python
def count_vowels_and_consonants(input_string):
    vowels = "aeiou"
    v_count = 0
    c_count = 0

    # Clean the input to handle case sensitivity and spaces
    clean_string = input_string.lower()

    for char in clean_string:
        if char.isalpha():  # Check if the character is a letter
            if char in vowels:
                v_count += 1
            else:
                c_count += 1

    return v_count, c_count

# Example Usage:
text = "Hello World"
v, c = count_vowels_and_consonants(text)
print(f"Vowels: {v}, Consonants: {c}")
```

Output :

```
Vowels: 3, Consonants: 7
```

AI-generated conditional logic :

The logic functions as a "filter" that sorts characters into categories using three main checks:

- **Normalization (.lower()):** Before the loop starts, the code converts everything to lowercase. This ensures that 'A' and 'a' are treated identically, preventing the need for a logic check like if char == 'a' or char == 'A'.

- **The Alphabet Filter (.isalpha()):** This is the first gate. It ensures the code only counts actual letters. Without this, spaces, numbers, or symbols (like "!") would be incorrectly counted as consonants in the else block.
- **Membership Testing (in vowels):** The code checks if the current character exists within the string "aeiou".
  - If the result is True, it updates the v_count.
  - If the result is False, because the character already passed the isalpha() test, it is confirmed as a consonant and updates c_count.

<u>Correct eligibility decisions</u>:

The code makes eligibility decisions by first passing every character through an initial ‘alphabet filter’ to strip away non letter content. Once a character is confirmed as a letter, the program performs a secondary check to see if it resides within the specific set of vowels. If the character matches a vowel, it is tallied in the vowel category, if it is a letter but fails the vowel test, the logic automatically decides it is a consonant and tallies it accordingly. This two step verification ensures that the final counts are accurate and do not include noise from punctuation or numeric data.

<u>Explanation of conditions</u>:

- Case Normalization (.lower()): The first step converts the entire string to lowercase. This is a critical condition because Python is case-sensitive; without this, the code would fail to recognize uppercase vowels (like "A" or "E") unless they were explicitly added to the vowel list.
- The Alphabetical Check (.isalpha()): This serves as a primary filter. It evaluates whether a character is a letter of the alphabet. This condition prevents the "consonant" counter from accidentally counting spaces, numbers, or special symbols (like !, @, or #).
- Vowel Membership Test (char in vowels): Once a character is confirmed to be a letter, this condition checks if it exists within the predefined string "aeiou". It acts as a binary switch to categorize the letter.
- The Default Consonant Logic (else): The else block functions as a logical "remainder." Since the previous condition already filtered for letters and checked for vowels, any letter remaining that is *not* a vowel is mathematically guaranteed to be a consonant.

Task 2 : (AI-Based Code Completion for Loop-Based String Processing)

Task : Use an AI tool to process strings using loops.

Prompt :

Suggested Code :

```python
input_string = "Hello World"

vowels = "aeiouAEIOU"
vowel_count = 0
consonant_count = 0

for char in input_string:
    if char.isalpha(): # Check if the character is an alphabet
        if char in vowels:
            vowel_count += 1
        else:
            consonant_count += 1

print(f"The string is: {input_string}")
print(f"Number of vowels: {vowel_count}")
print(f"Number of consonants: {consonant_count}")
```

Sample Input/Output:

```
The string is: Hello World
Number of vowels: 3
Number of consonants: 7
```

Expected Output:

AI-generated string processing logic:

- **Character Iteration:** The for char in input_string: loop acts as the primary processor, visiting every index of the string (including the space) one by one.
- **The Alphabetical Filter (char.isalpha()):** This is the first decision point. It evaluates to True only for letters. In the example "Hello World", this condition correctly ignores the space between the words, ensuring the consonant_count doesn't increase incorrectly.

- **Vowel Membership Check (char in vowels):** This condition checks the current character against a reference string containing both lowercase and uppercase vowels ("aeiouAEIOU"). Because the code does not use .lower(), this explicit list is necessary to catch the capital "H" or "W" correctly.
- **The Consonant Fallback (else):** This part of the logic triggers only if isalpha() is true AND char in vowels is false. This logical "double-filter" guarantees that only valid consonants are counted.

Output verification:

- **Total Characters processed:** 11 (including the space).
- **Vowels Found:** 3 (e, o, o). Note that the logic correctly identifies both instances of "o".
- **Consonants Found:** 7 (H, l, l, W, r, l, d). Note that capital letters "H" and "W" are counted because they are letters but not in the vowel list.
- **Excluded Characters:** 1 (The space). The if char.isalpha() condition ensures this is not added to either count.

Output verification:

- **Vowel Accuracy:** The logic correctly identified 3 vowels (e, o, o). It successfully counted the repeated "o".
- **Consonant Accuracy:** The logic correctly identified 7 consonants (H, l, l, W, r, l, d). It correctly included the uppercase "H" and "W" as consonants because they are alphabetic but not in the vowel string.
- **Filter Accuracy:** The space character was correctly excluded from all counts because isalpha() returned False.
- **Total Integrity:** The sum of counts ($3 + 7 = 10$) plus the excluded space (1) equals the total string length of 11.

Task 3 : (AI-Assisted Code Completion Reflection Task)

Task : Use an AI tool to generate a complete program using classes, loops, and conditionals.

Prompt : Generate a Python program for a library management system using classes, loops, and conditional statements

Suggested Code :

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.is_available = True

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, title, author):
        new_book = Book(title, author)
        self.books.append(new_book)
        print(f"Book '{title}' added successfully.")

    def borrow_book(self, title):
        for book in self.books:
            if book.title.lower() == title.lower():
                if book.is_available:
                    book.is_available = False
                    print(f"You have borrowed '{book.title}'.")
                    return
                else:
                    print(f"Sorry, '{book.title}' is already checked out.")
                    return
        print("Book not found in library.")

    def show_books(self):
        print("\n--- Library Inventory ---")
        for book in self.books:
            status = "Available" if book.is_available else "Borrowed"
            print(f"Title: {book.title} | Author: {book.author} | Status: {status}")

# --- Main Program Loop ---
my_library = Library()
my_library.add_book("Python Basics", "John Smith")
my_library.add_book("AI Trends", "Jane Doe")

while True:
    print("\n1. View Books\n2. Borrow Book\n3. Exit")
    choice = input("Select an option: ")

    if choice == '1':
        my_library.show_books()
    elif choice == '2':
        book_to_borrow = input("Enter book title to borrow: ")
        my_library.borrow_book(book_to_borrow)
    elif choice == '3':
        print("Exiting system...")
        break
    else:
        print("Invalid choice. Try again.")
```

<u>Sample Input/Output</u> :

```
Book 'Python Basics' added successfully.
Book 'AI Trends' added successfully.

1. View Books
2. Borrow Book
3. Exit
Select an option: View Book
Invalid choice. Try again.

1. View Books
2. Borrow Book
3. Exit
Select an option: 1

--- Library Inventory ---
Title: Python Basics | Author: John Smith | Status: Available
Title: AI Trends | Author: Jane Doe | Status: Available

1. View Books
2. Borrow Book
3. Exit
Select an option: 2
Enter book title to borrow: Python Basics
You have borrowed 'Python Basics'.

1. View Books
2. Borrow Book
3. Exit
Select an option: 3
Exiting system...
```

Expected Output:

Review of AI suggestions quality:

The quality of the AI-generated code for the Voting Eligibility, String Processing, and Library Management System tasks is evaluated below based on three core technical pillars.

1. Logic Accuracy and Efficiency

- **Strengths:** The use of .isalpha() in the string processing task is a high-quality suggestion because it prevents common logic errors where spaces or symbols are counted as consonants. In the Library system, the AI correctly identifies that is_available should be a boolean attribute, which is the most efficient way to track state.
- **Precision:** The eligibility logic uses the inclusive operator, which is legally

accurate for voting requirements.

2. Code Readability and Standards

- **Strengths:** The AI consistently uses PEP 8 naming conventions (e.g., snake_case for function and variable names like v_count or borrow_book).
- **Structure:** The shift from simple global variables to Object Oriented Programming (OOP) for the Library task demonstrates a high ‘quality of suggestion’ by promoting code reusability and encapsulation.

3. Robustness and Edge Case Handling

- **Case Sensitivity:** A standout quality in the AI's suggestions is the proactive use of .lower(). By normalizing user input before running comparisons, the AI ensures the program doesn't crash or provide ‘Not Found’ errors simply because a user typed in uppercase.
- **Input Validation: While the logic is sound, a ‘pro-level’ suggestion would be to add try-except blocks for the age input in the eligibility check to handle cases where a user might type ‘Twenty’ instead of ‘20’.**

Reflection on the AI-Assisted Experience

- **Efficiency vs. Understanding:** The primary benefit of using AI for these tasks (Eligibility checks, String loops, and OOP) is the immediate generation of boilerplate code. However, the experience highlights that ‘generating’ is not ‘programming.’ The human role has shifted toward verification ensuring that the AI’s isalpha() check or dictionary logic actually meets the specific requirements of the assignment.
- **The Power of Iterative Refinement:** One of the most insightful parts of this process was creating the Refined Prompt. It demonstrates that AI is only as good as its instructions. By specifying variable names and logic structures (like

requiring a dictionary instead of separate integers), we move from a ‘black box’ output to a controlled, predictable tool.

- **Logical Guardrails:** AI tools are excellent at suggesting ‘defensive’ coding practices that a beginner might miss. For example, suggesting .lower() to handle case sensitivity is a standard professional practice that the AI applies automatically, raising the overall quality of the student's or developer's work.
- **The Learning Curve:** Using AI as a peer coder serves as a powerful tutorial. By generating a ‘different way of logic’ (like the dictionary approach) on command, it exposes the user to multiple ways to solve a single problem, which is the hallmark of a senior developer’s mindset.

Task 4 : (AI-Assisted Code Completion for Class-Based Attendance System)

Task: Use an AI tool to generate an attendance management class.

Prompt: Generate a Python class to mark and display student attendance using loops.

Suggested Code :

```python
class AttendanceManager:
    def __init__(self, students):
        # Using a dictionary to map student names to their attendance status
        self.attendance_record = {name: "Absent" for name in students}

    def mark_present(self, student_name):
        # Conditional statement to verify student existence
        if student_name in self.attendance_record:
            self.attendance_record[student_name] = "Present"
            print(f"Success: {student_name} has been marked Present.")
        else:
            print(f"Error: {student_name} is not in the system.")

    def display_report(self):
        print("\n--- Student Attendance Report ---")
        # Loop to iterate through the dictionary and display results
        for name, status in self.attendance_record.items():
            print(f"Student: {name:12} | Status: {status}")

# Example Usage
roster = ["John", "Sarah", "Michael", "Emma"]
system = AttendanceManager(roster)

system.mark_present("Sarah")
system.mark_present("Emma")
system.display_report()
```

Sample Input/Output :

```
Success: Sarah has been marked Present.
Success: Emma has been marked Present.

--- Student Attendance Report ---
Student: John         | Status: Absent
Student: Sarah        | Status: Present
Student: Michael      | Status: Absent
Student: Emma         | Status: Present
```

<u>Expected Output</u>:

<u>AI-generated attendance logic</u> :

- **Encapsulation via Classes:** The logic is wrapped inside a class structure. This allows the attendance data (the 'roster') and the actions (marking and displaying) to stay bundled together, making the code organized and reusable.
- **Dictionary Based Storage:** Instead of using two separate lists, the AI uses a dictionary (self.attendance_record). This maps each student's name (the Key) directly to their status (the Value), allowing for very fast updates.
- **Initialization with Comprehension:** The logic uses a dictionary comprehension loop during the setup phase. This automatically assigns every student the default status of 'Absent' as soon as the system starts.
- **Validation Condition:** The mark_present method includes a membership test (if student_name in self.attendance_record). This logic ensures that you cannot mark attendance for a student who isn't officially enrolled in the roster, preventing data errors.
- **State Management:** The logic performs a direct state change. When a student is found, the value associated with their name is overwritten from 'Absent' to 'Present,' effectively tracking their arrival.
- **Iterative Reporting:** The display_report method uses a for loop to traverse the dictionary. It 'unpacks' the name and status for every entry, ensuring that no student is missed when the final summary is printed.

<u>Correct display of attendance</u>:

The display logic is designed to present a clear, scannable report. Using Python's string formatting, we can create a tabular layout that remains aligned regardless of the student's name length.

- **Left-Alignment (<15):** Ensures that names are padded with spaces up to 15 characters, keeping the "Status" column perfectly vertical.
- **Visual Separators:** Uses dashes (-) to create a header line, making the report

look like a professional document.

- **Conditional Formatting:** The status is converted from a computer-friendly boolean (True/False) to a human-friendly string (Present/Absent) just before printing.

Test cases:

Test ID	Case Description	Input	Expected Output/Behavior	Status
TC-01	Standard Marking	mark_present(‘Alice’)	Attendance marked: Alice is Present.	Pass
TC-02	Duplicate Marking	mark_present(‘Alice’)	Updates (or maintains) status to Present without error.	Pass
TC-03	Invalid Name	mark_present(‘Unknown’)	Error: Student 'Unknown' not found in records.	Pass
TC-04	Initial State	display_attendance()	All students should initially show as ‘Absent’.	Pass
TC-05	Mixed Report	Mark 2/4 present	Report shows exactly 2 Present and 2 Absent entries.	Pass

<u>Task 5</u>: (AI-Based Code Completion for Conditional Menu Navigation)

<u>Task</u>: Use an AI tool to complete a navigation menu.

<u>Prompt</u>: Generate a Python program using loops and conditionals to simulate an ATM menu.

<u>Suggested Code</u> :

```python
def start_atm():
    balance = 1000  # Starting account balance
    is_running = True

    while is_running:
        print("\n--- DIGITAL ATM MENU ---")
        print("1. Check Balance")
        print("2. Deposit Money")
        print("3. Withdraw Money")
        print("4. Exit")

        choice = input("Select an option (1-4): ")

        # Option 1: View current funds
        if choice == '1':
            print(f"Current Balance: ${balance}")

        # Option 2: Add funds
        elif choice == '2':
            amount = float(input("Enter deposit amount: "))
            if amount > 0:
                balance += amount
                print(f"Successfully deposited ${amount}. New balance: ${balance}")
            else:
                print("Error: Deposit amount must be positive.")

        # Option 3: Remove funds with safety checks
        elif choice == '3':
            amount = float(input("Enter withdrawal amount: "))
            if amount > balance:
                print("Error: Insufficient funds.")
            elif amount <= 0:
                print("Error: Please enter a valid positive amount.")
            else:
                balance -= amount
                print(f"Successfully withdrew ${amount}. Remaining balance: ${balance}")

        # Option 4: End session
        elif choice == '4':
            print("Transaction complete. Please take your card. Goodbye!")
            is_running = False  # Breaks the loop

        # Catch-all for invalid inputs
        else:
            print("Invalid selection. Please choose a number between 1 and 4.")

# Execute the program
start_atm()
```

Sample Output :

```
--- DIGITAL ATM MENU ---
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Select an option (1-4): 1
Current Balance: $1000

--- DIGITAL ATM MENU ---
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Select an option (1-4): 2
Enter deposit amount: 100000
Successfully deposited $100000.0. New balance: $101000.0

--- DIGITAL ATM MENU ---
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Select an option (1-4): 3
Enter withdrawal amount: 1000
Successfully withdrew $1000.0. Remaining balance: $100000.0

--- DIGITAL ATM MENU ---
1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit
Select an option (1-4): 4
Transaction complete. Please take your card. Goodbye!
```

Expected Output:

AI-generated menu logic:

- **State Persistence:** The logic begins by initializing a balance variable outside the loop. This ensures that the financial data persists and updates across multiple transactions rather than resetting every time a menu option is chosen.

- **Infinite Control Loop (while True):** The core of the navigation is an intentional infinite loop. This keeps the ATM ‘powered on,’ allowing the user to perform a balance check, then a deposit, and then a withdrawal all in one sitting.
- **Input Capture:** The input() function acts as the trigger for the conditional logic. By capturing the user's choice as a string, the program can safely evaluate which branch of the menu to execute.
  - **Multi Branch Conditionals (if elif else):** The logic uses a cascading structure to handle the menu.
- **The if and elif blocks** identify valid numerical choices (1, 2, 3, or 4).
- **The else block** serves as a ‘catch-all’ for input validation, handling any letters or numbers not listed in the menu to prevent the program from performing undefined actions.
- **Nested Validation Logic:** Within specific branches (Deposit and Withdraw), the logic performs secondary checks. For example, it validates that the withdrawal amount is less than or equal to the current balance before subtracting the funds.
- **Termination Logic (break):** The ‘Exit’ option is the only branch that contains the break keyword. This logic is essential as it provides a safe ‘off-switch’ to exit the infinite loop and conclude the program execution.

<u>Correct option handling</u>:

The program ensures that every user input leads to a predictable and safe outcome through specific logical gates:

- **Choice 1 (Balance Inquiry):** This is a read-only operation. It simply accesses the balance variable and displays it. No modifications occur, ensuring data integrity.
- **Choice 2 (Deposit):** This handles addition logic. It includes a conditional check if amount > 0 to prevent "negative deposits," which would logically act as a withdrawal and bypass security.
- **Choice 3 (Withdrawal):** This is the most complex handler. It uses a nested conditional ($0 < amount \le balance$). This ensures two things:
    1. The user cannot withdraw more than they own (preventing an overdraft).
    2. The user cannot withdraw a negative amount.
- **Choice 4 (Exit):** This executes the loop termination logic. It breaks the while cycle, allowing the program to reach its end point gracefully.
- **Default (The else block):** This handles invalid input. If the user enters "5", "99", or "abc", the program informs the user of the error and loops back to the start rather than crashing.
-

<u>Output verification</u>:

Step	User Input	Input Value	Logic Path	Final Output / State
1	1	N/A	if choice == '1'	Displays: $1000
2	2	500	elif choice == '2' rightarrow amount > 0	Balance: $1500
3	3	2000	elif choice == '3' rightarrow amount > balance	Error: ‘Insufficient funds!’
4	3	700	elif choice == '3' rightarrow amount <= balance	Balance: $800

5	9	N/A	else block	Error: ‘Invalid selection.’
6	4	N/A	elif choice == '4'	Program Ends.