ASSIGNMENT – 6.3

HT.NO: 2303A52185

BATCH: 45

TASK-01:

Question:

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.

Prompt:

Create a Python Student class with attributes name, roll_number, and branch.

Add a constructor (__init__) and a method display_details() to print student information.

Generated Code:

```
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"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")
# Example usage
student1 = Student("Alice", "101", "Computer Science")
student1.display_details()
```

Output:

```
Name: Alice
Roll Number: 101
Branch: Computer Science
```

Explanation:

This code defines a Student class that stores a student's name, roll number, and branch.
The __init__ method sets these values when a student object is created.
The display_details() method prints the student's information, and the example shows how to create and display one student's details.

TASK-02:

Question:

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).

Prompt (2.1):

Write a Python function that prints the first 10 multiples of a given number using a loop.

Generated Code:

```
def print_multiples(number):
    print(f"First 10 multiples of {number}:")
    for i in range(1, 11):
        multiple = number * i
        print(multiple)
# Example usage
print_multiples(5)
```

Output (2.1):

```
5
10
15
20
25
30
35
40
45
50
```

Explanation:

This function prints the first 10 multiples of a given number.
It uses a for loop from 1 to 10 and multiplies the number with each value.
Calling print_multiples(5) displays multiples of 5 from 5 to 50.

Prompt (2.2):

Write a Python function that prints the first 10 multiples of a given number using while loop.

Generated Code:

```
def print_multiples_while(number):
    print(f"First 10 multiples of {number} using while loop:")
    i = 1
    while i <= 10:
        multiple = number * i
        print(multiple)
        i += 1
# Example usage
print_multiples_while(5)
```

Output (2.2):

```
First 10 multiples of 5 using while loop:
5
10
15
20
25
30
35
40
45
50
```

Explanation:

This function prints the first 10 multiples of a given number using a while loop.
It starts with i = 1 and keeps multiplying the number until i reaches 10.
Each multiple is printed, and i is increased by 1 in every loop iteration.

TASK-03:

Question:

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).

Prompt (3.1):

Write a Python function that classifies a person's age into groups such as child, teenager, adult, and senior using if-elif-else statements.

Generated Code:

```
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"
# Example usage
ages = [5, 15, 30, 65, -3]
```

```
for age in ages:
    category = classify_age(age)
    print(f"Age: {age} -> Category: {category}")
```

Output (3.1):

```
Age: 5 -> Category: Child
Age: 15 -> Category: Teenager
Age: 30 -> Category: Adult
Age: 65 -> Category: Senior
Age: -3 -> Category: Invalid age
```

Explanation:

This function checks a person's age and returns a category like Child, Teenager, Adult, or Senior.
It uses if-elif-else conditions to compare the age with different ranges.
The example tests multiple ages and prints their corresponding categories.

Prompt (3.2):

Implement the same age classification using an alternative approach such as simplified conditional statements or dictionary-based logic.

Explain how this approach works and how it differs from the if-elif-else method.

Generated Code:

```
def classify_age_dict(age):
    age_groups = {
        (0, 12): "Child",
        (13, 19): "Teenager",
        (20, 59): "Adult",
        (60, float('inf')): "Senior"
    }
    for (min_age, max_age), group in age_groups.items():
        if min_age <= age <= max_age:
            return group
    return "Invalid age"
# Example usage of the alternative approach
```

```
ages = [5, 15, 30, 65, -3]

for age in ages:

    category = classify_age_dict(age)

    print(f"Age: {age} -> Category: {category}")

# Explanation:

# In this alternative approach, we use a dictionary to map age ranges to their corresponding categories.

# The function iterates through the dictionary items and checks which range the given age falls into.

# This method separates the data (age ranges and categories) from the logic (classification),

# making it easier to modify or extend the age groups in the future.
```

Output (3.2):

```
Age: 5 -> Category: Child
Age: 15 -> Category: Teenager
Age: 30 -> Category: Adult
Age: 65 -> Category: Senior
Age: -3 -> Category: Invalid age
```

Explanation:

This function classifies a person's age using a **dictionary of age ranges** instead of multiple if-elif statements.
It checks which age range the given value falls into and returns the matching category.
This approach is flexible and easy to update because age ranges and categories are stored separately from the logic.

TASK-04:

Question:

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.

Prompt (4.1):

Write a Python function sum_to_n(n) that calculates the sum of the first n natural numbers using a for loop.

Generated Code:

```
def sum_to_n(n):
    if n < 1:
        return 0
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
# Example usage
n = 10
result = sum_to_n(n)
print(f"The sum of the first {n} natural numbers is: {result}")
```

Output (4.1):

```
The sum of the first 10 natural numbers is: 55
```

Explanation:

This function calculates the sum of the first n natural numbers.
If n is less than 1, it returns 0 to avoid invalid input.
Otherwise, it uses a for loop to add numbers from 1 to n and prints the final sum.

Prompt (4.2):

Rewrite the sum_to_n(n) function using an alternative approach such as a while loop or a mathematical formula.

Explain how this approach differs from the for loop implementation.

Generated Code:

```
def sum_to_n_formula(n):
    if n < 1:
        return 0
```

```
    return n * (n + 1) // 2

# Example usage of the alternative implementation

result_formula = sum_to_n_formula(n)

print(f"(Using formula) The sum of the first {n} natural numbers is: {result_formula}")

# Explanation:

# The first implementation uses a for loop to iterate through each natural number from 1 to n,

# accumulating the total sum. This approach is straightforward and easy to understand, but it has a

# time complexity of O(n).

# The alternative implementation uses the mathematical formula n(n + 1)/2, which provides a

# direct calculation of the sum. This method is more efficient with a time complexity of O(1)

# since it performs a constant number of operations regardless of the value of n.
```

Output (4.2):

```
(Using formula) The sum of the first 10 natural numbers is: 55
```

Explanation:

This code finds the sum of the first n natural numbers using a mathematical formula.
If n is less than 1, it returns 0; otherwise, it directly calculates the sum using n(n + 1)/2.
Compared to the loop-based method, this approach is faster and more efficient because it runs in constant time O(1).

TASK-05:

Question:

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.

Prompt:

Demonstrate deposit and withdrawal operations and display the updated balance.
Analyze the class structure and logic, add meaningful comments, and explain how the code works.

Generated Code:

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        """Deposit a specified amount into the account."""
        if amount > 0:
            self.balance += amount
            print(f"Deposited: {amount}. New balance: {self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """Withdraw a specified amount from the account if sufficient funds are available."""
        if amount > 0:
            if amount <= self.balance:
                self.balance -= amount
                print(f"Withdrew: {amount}. New balance: {self.balance}")
            else:
                print("Insufficient funds for this withdrawal.")
        else:
            print("Withdrawal amount must be positive.")

    def check_balance(self):
        """Return the current balance of the account."""
```

```
        return self.balance

# Example usage

account = BankAccount("John Doe", 1000)

account.deposit(500)

account.withdraw(200)

print(f"Current Balance: {account.check_balance()}")

# Explanation:

# The BankAccount class represents a simple bank account with basic functionalities.

# The constructor (__init__) initializes the account holder's name and the initial balance.

# The deposit() method allows adding funds to the account, ensuring the amount is positive.

# The withdraw() method allows removing funds from the account, checking for sufficient balance first.

# The check_balance() method returns the current balance of the account.
```

Output:

```
Deposited: 500. New balance: 1500
Withdrew: 200. New balance: 1300
Current Balance: 1300
```

Explanation:

This code defines a **BankAccount** class to manage basic banking operations.
It stores the account holder's name and balance, and provides methods to deposit money, withdraw money (with balance checks), and view the current balance.
The example shows creating an account, performing transactions, and printing the final balance.