AI Assisted Coding Lab – Assignment 6.3

2303A51171
Name: T.Rohith
Batch-18

Task 1: Classes (Student Class)

Prompt Used:
Generate a Python class named Student with attributes name, roll number, and branch. Include a method display_details() to print the student information.

AI 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("Student Name:", self.name)
        print("Roll Number:", self.roll_number)
        print("Branch:", self.branch)


student1 = Student("Rohith", "23CS104", "CSE")
student1.display_details()
```

Output:

Student Name: Rohith

Roll Number: 23CS104
Branch: CSE

Analysis:
The AI-generated code is correct and follows object-oriented programming principles. The constructor initializes the attributes properly, and the display_details() method clearly prints the

student information. The structure is simple, readable, and easy to modify for future enhancements.

Task 2: Loops (Multiples of a Number)

Prompt Used:
Generate a Python function that prints the first 10 multiples of a number using both for loop and while loop.

AI Generated Code:

Using For Loop:

```
def multiples_for(n):
    for i in range(1, 11):
        print(n * i)

multiples_for(5)
```

Using While Loop:

```
def multiples_while(n):
    i = 1
    while i <= 10:
        print(n * i)
        i += 1

multiples_while(5)
```

Output:

5
10
15
20
25
30
35
40

45
50

Comparison and Analysis:
The for loop is easier to read and is best suited when the number of iterations is known in advance. The while loop provides more control over execution but requires manual initialization and increment, which increases the chance of logical errors such as infinite loops. Both implementations are correct and efficient for this task.

Task 3: Conditional Statements (Age Classification)

Prompt Used:
Generate nested if-elif-else statements to classify age groups and provide an alternative method.

AI Generated Code:

Using If-Elif-Else:

```
def classify_age(age):
    if age < 13:
        return "Child"
    elif age < 20:
        return "Teenager"
    elif age < 60:
        return "Adult"
    else:
        return "Senior"

print(classify_age(25))
```

Alternative Approach:

```
def classify_age_alt(age):
    categories = {
        "Child": age < 13,
        "Teenager": 13 <= age < 20,
        "Adult": 20 <= age < 60,
        "Senior": age >= 60
    }
```

```
    for category, condition in categories.items():
        if condition:
            return category

print(classify_age_alt(25))
```

Output:

child

Explanation:
The conditions are evaluated sequentially until one becomes true. The dictionary-based approach improves readability and makes the code easier to extend if additional categories are needed.

Task 4: Sum of First n Natural Numbers

Prompt Used:
Generate a Python function to compute the sum of first n numbers using for loop, while loop, and mathematical formula.

AI Generated Code:

Using For Loop:

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

print(sum_for(10))
```

Using While Loop:

```
def sum_while(n):
    total = 0
    i = 1
    while i <= n:
```

```
        total += i
        i += 1
    return total

print(sum_while(10))
```

Using Mathematical Formula:

```
def sum_formula(n):
    return n * (n + 1) // 2

print(sum_formula(10))
```

Output:

55
55
55

Comparison:
Both the for loop and while loop have a time complexity of O(n). The mathematical formula computes the result instantly with O(1) time complexity, making it the most efficient approach.

Task 5: Bank Account Class

Prompt Used:
Generate a BankAccount class with deposit, withdraw, and check_balance methods including comments.

AI Generated Code:

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

    def deposit(self, amount):
        self.balance += amount
        print("Deposited:", amount)
```

```
    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient balance!")
        else:
            self.balance -= amount
            print("Withdrawn:", amount)

    def check_balance(self):
        print("Current Balance:", self.balance)


acc = BankAccount("Rohith", 1000)

acc.deposit(500)
acc.withdraw(300)
acc.check_balance()
```

Output:

Deposited: 500
Withdrawn: 300
Current Balance: 1200

Explanation:
The class properly implements encapsulation by managing the balance through methods instead of allowing direct modification. The withdraw method includes a condition to prevent overdrawing, improving reliability. The structure is clear, logical, and suitable for basic banking applications.