AI Assisted Coding
Assignment - 6
M.Rohan || 2303A51545 || Batch:- 8

Task Description #1: Classes (Student Class)
Scenario
You are developing a simple student information management module.

Code:

```
In [1]: 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"Student Name: {self.name}")
                print(f"Roll Number: {self.roll_number}")
                print(f"Branch: {self.branch}")

        # Sample object creation and output verification
        print("--- Task 1: Student Class ---")
        student1 = Student("Alice Smith", "A101", "Computer Science")
        student1.display_details()

        student2 = Student("Bob Johnson", "B202", "Electrical Engineering")
        student2.display_details()

        print("\n--- Analysis of AI-Generated Code ---")
        print("The AI-generated code successfully created a 'Student' class with a constructor (__init__) to initialize 'name', 'roll_number', and 'branch' attributes. It also includes a 'display_details' method

--- Task 1: Student Class ---
Student Name: Alice Smith
Roll Number: A101
Branch: Computer Science
Student Name: Bob Johnson
Roll Number: B202
Branch: Electrical Engineering

--- Analysis of AI-Generated Code ---
The AI-generated code successfully created a 'Student' class with a constructor (__init__) to initialize 'name', 'roll_number', and 'branch' attributes. It also includes a 'display_details' method that corr
ectly prints the student's information in a readable format. The code is clear, follows standard Python class definition practices, and is functionally correct for the specified task.
```

Task Description #2: Loops (Multiples of a Number)
Scenario
You are writing a utility function to display multiples of a given number.

Code:

```
In [...
def print_multiples_for_loop(number):
    print(f"--- Multiples of {number} (using for loop) ---")
    for i in range(1, 11): # Loop from 1 to 10 to get the first 10 multiples
        multiple = number * i
        print(f"{number} x {i} = {multiple}")

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

# Sample usage for for loop
print_multiples_for_loop(7)
print_multiples_for_loop(3)

# Sample usage for while loop
print_multiples_while_loop(7)
print_multiples_while_loop(3)

print("\n--- Analysis of 'for' loop logic ---")
print("The 'print_multiples_for_loop' function uses a 'for' loop with the 'range(1, 11)' function. 'range(1, 11)' generates numbers from 1 up to (but not including) 11, effectively 1, 2, ..., 10. In each iteration, 'i' takes on one of these values, and the mul

print("\n--- Analysis of 'while' loop logic ---")
print("The 'print_multiples_while_loop' function uses a 'while' loop. It initializes a 'count' variable to 1 and continues looping as long as 'count' is less than or equal to 10. Inside the loop, it calculates and prints the multiple, then increments 'count'. "

File display

--- Multiples of 7 (using for loop) ---
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
--- Multiples of 3 (using for loop) ---
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
--- Multiples of 7 (using while loop) ---
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
--- Multiples of 3 (using while loop) ---
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

Task Description #3: Conditional Statements (Age Classification)
Scenario
You are building a basic classification system based on age.

Code:

```
In [ ]:
def classify_age_nested_if(age):
    print(f"--- Classifying age: {age} (using nested if-elif-else) ---")
    if age < 0:
        category = "Invalid Age"
    else:
        if age <= 12:
            category = "Child"
        elif age <= 17:
            category = "Teenager"
        elif age <= 64:
            category = "Adult"
        else:
            category = "Senior"
    print(f"Age {age} is classified as: {category}")
    return category

def classify_age_simplified(age):
    print(f"--- Classifying age: {age} (using simplified if-elif-else) ---")
    if age < 0:
        category = "Invalid Age"
    elif age <= 12:
        category = "Child"
    elif age <= 17:
        category = "Teenager"
    elif age <= 64:
        category = "Adult"
    else:
        category = "Senior"
    print(f"Age {age} is classified as: {category}")
    return category

# Sample usage for nested if-elif-else
classify_age_nested_if(5)
classify_age_nested_if(15)
classify_age_nested_if(30)
classify_age_nested_if(70)
classify_age_nested_if(-2)
classify_age_nested_if(12)
classify_age_nested_if(17)
classify_age_nested_if(64)

print("\n--- Analysis of Nested if-elif-else Logic ---")
print("The 'classify_age_nested_if' function uses a hierarchical structure of 'if', 'elif', and 'else' statements. It first checks for invalid age (less than 0). If the age is valid, it proceeds to a nested set of conditions. Each 'elif' condition builds upon

# Sample usage for simplified if-elif-else
print("\n--- Sample usage for Simplified if-elif-else ---")
classify_age_simplified(5)
classify_age_simplified(15)
classify_age_simplified(30)
classify_age_simplified(70)
classify_age_simplified(-2)

print("\n--- Analysis of Simplified if-elif-else Logic ---")
print("The 'classify_age_simplified' function achieves the same result using a linear sequence of 'if-elif-else' statements without explicit nesting. The key to this simplification is the order of conditions. By placing the most restrictive conditions first (e
```

```
--- Classifying age: 5 (using nested if-elif-else) ---
Age 5 is classified as: Child
--- Classifying age: 15 (using nested if-elif-else) ---
Age 15 is classified as: Teenager
--- Classifying age: 30 (using nested if-elif-else) ---
Age 30 is classified as: Adult
--- Classifying age: 70 (using nested if-elif-else) ---
Age 70 is classified as: Senior
--- Classifying age: -2 (using nested if-elif-else) ---
Age -2 is classified as: Invalid Age
--- Classifying age: 12 (using nested if-elif-else) ---
Age 12 is classified as: Child
--- Classifying age: 17 (using nested if-elif-else) ---
Age 17 is classified as: Teenager
--- Classifying age: 64 (using nested if-elif-else) ---
Age 64 is classified as: Adult

--- Analysis of Nested if-elif-else Logic ---
The 'classify_age_nested_if' function uses a hierarchical structure of 'if', 'elif', and 'else' statements. It first checks for invalid age (less than 0). If the age is valid, it proceeds to a nested set of conditions. Each 'elif' condition builds upon the previ
us one; for example, 'elif age <= 17' only executes if 'age <= 12' was false, effectively checking for ages between 13 and 17. This nested approach ensures that each age falls into exactly one category based on the defined thresholds, ordered from youngest to ol
est.

--- Sample usage for Simplified if-elif-else ---
--- Classifying age: 5 (using simplified if-elif-else) ---
Age 5 is classified as: Child
--- Classifying age: 15 (using simplified if-elif-else) ---
Age 15 is classified as: Teenager
--- Classifying age: 30 (using simplified if-elif-else) ---
Age 30 is classified as: Adult
--- Classifying age: 70 (using simplified if-elif-else) ---
Age 70 is classified as: Senior
--- Classifying age: -2 (using simplified if-elif-else) ---
Age -2 is classified as: Invalid Age

--- Analysis of Simplified if-elif-else Logic ---
The 'classify_age_simplified' function achieves the same result using a linear sequence of 'if-elif-else' statements without explicit nesting. The key to this simplification is the order of conditions. By placing the most restrictive conditions first (e.g., 'age
< 0', then 'age <= 12'), subsequent 'elif' blocks implicitly handle the ranges not covered by previous conditions. For instance, 'elif age <= 17' is only reached if 'age > 12', effectively checking for ages between 13 and 17. This approach is generally more read
ble and less prone to logical errors than deeply nested structures when conditions are mutually exclusive and ordered logically.
```

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.

Code:

```
In [ ]:
def sum_to_n_for_loop(n):
    """Calculates the sum of the first n natural numbers using a for loop."""
    print(f"--- Sum of first {n} numbers (using for loop) ---")
    total_sum = 0
    if n < 0:
        print("Invalid input: n should be a non-negative integer.")
        return 0
    for i in range(1, n + 1):
        total_sum += i
    print(f"The sum of the first {n} natural numbers is: {total_sum}")
    return total_sum

def sum_to_n_while_loop(n):
    """Calculates the sum of the first n natural numbers using a while loop."""
    print(f"--- Sum of first {n} numbers (using while loop) ---")
    total_sum = 0
    current_num = 1
    if n < 0:
        print("Invalid input: n should be a non-negative integer.")
        return 0
    while current_num <= n:
        total_sum += current_num
        current_num += 1
    print(f"The sum of the first {n} natural numbers is: {total_sum}")
    return total_sum

def sum_to_n_formula(n):
    """Calculates the sum of the first n natural numbers using the mathematical formula."""
    print(f"--- Sum of first {n} numbers (using formula) ---")
    if n < 0:
        print("Invalid input: n should be a non-negative integer.")
        return 0
    total_sum = (n * (n + 1)) // 2 # Using integer division
    print(f"The sum of the first {n} natural numbers is: {total_sum}")
    return total_sum

# Sample usage for for loop
sum_to_n_for_loop(5)
sum_to_n_for_loop(10)
sum_to_n_for_loop(1)
sum_to_n_for_loop(0)
sum_to_n_for_loop(-3)

# Sample usage for while loop
print("\n--- Sample usage for while loop ---")
sum_to_n_while_loop(5)
sum_to_n_while_loop(10)
sum_to_n_while_loop(1)
sum_to_n_while_loop(0)
sum_to_n_while_loop(-3)

# Sample usage for formula
print("\n--- Sample usage for mathematical formula ---")
sum_to_n_formula(5)
sum_to_n_formula(10)
sum_to_n_formula(1)
sum_to_n_formula(0)
sum_to_n_formula(-3)

print("\n--- Analysis of 'for' loop implementation ---")
print("The `sum_to_n_for_loop` function initializes `total_sum` to 0. It then uses a `for` loop with `range(1, n + 1)` to iterate through each natural number from 1 up to and including `n`. In each iteration, the current number `i` is added to `total_sum`. This

print("\n--- Analysis of 'while' loop implementation ---")
print("The `sum_to_n_while_loop` function achieves the same result as the `for` loop using a `while` loop. It initializes `total_sum` to 0 and `current_num` to 1. The loop continues as long as `current_num` is less than or equal to `n`. Inside the loop, `curre

print("\n--- Analysis of 'mathematical formula' implementation ---")
print("The `sum_to_n_formula` function calculates the sum using the direct mathematical formula for the sum of an arithmetic series: `n * (n + 1) / 2`. This method is the most efficient for this specific problem as it avoids iteration entirely, providing a cons
```

File display

```
--- Sum of first 5 numbers (using for loop) ---
The sum of the first 5 natural numbers is: 15
--- Sum of first 10 numbers (using for loop) ---
The sum of the first 10 natural numbers is: 55
--- Sum of first 1 numbers (using for loop) ---
The sum of the first 1 natural numbers is: 1
--- Sum of first 0 numbers (using for loop) ---
The sum of the first 0 natural numbers is: 0
--- Sum of first -3 numbers (using for loop) ---
Invalid input: n should be a non-negative integer.

--- Sample usage for while loop ---
--- Sum of first 5 numbers (using while loop) ---
The sum of the first 5 natural numbers is: 15
--- Sum of first 10 numbers (using while loop) ---
The sum of the first 10 natural numbers is: 55
--- Sum of first 1 numbers (using while loop) ---
The sum of the first 1 natural numbers is: 1
--- Sum of first 0 numbers (using while loop) ---
The sum of the first 0 natural numbers is: 0
--- Sum of first -3 numbers (using while loop) ---
Invalid input: n should be a non-negative integer.

--- Sample usage for mathematical formula ---
--- Sum of first 5 numbers (using formula) ---
The sum of the first 5 natural numbers is: 15
--- Sum of first 10 numbers (using formula) ---
The sum of the first 10 natural numbers is: 55
--- Sum of first 1 numbers (using formula) ---
The sum of the first 1 natural numbers is: 1
--- Sum of first 0 numbers (using formula) ---
The sum of the first 0 natural numbers is: 0
--- Sum of first -3 numbers (using formula) ---
Invalid input: n should be a non-negative integer.

--- Analysis of 'for' loop implementation ---
The `sum_to_n_for_loop` function initializes `total_sum` to 0. It then uses a `for` loop with `range(1, n + 1)` to iterate through each natural number from 1 up to and including `n`. In each iteration, the current number `i` is added to `total_sum`. This iterati
e approach correctly accumulates the sum. The `range` function is ideal here as it provides a clear sequence of numbers to iterate over for a predefined number of steps. A check for negative 'n' is added for robustness.

--- Analysis of 'while' loop implementation ---
The `sum_to_n_while_loop` function achieves the same result as the 'for' loop using a `while` loop. It initializes `total_sum` to 0 and `current_num` to 1. The loop continues as long as `current_num` is less than or equal to `n`. Inside the loop, `current_num` i
added to `total_sum`, and then `current_num` is incremented. This explicit management of the loop counter makes it suitable for cases where the termination condition is more dynamic, but for a fixed range, 'for' is often more concise. A check for negative 'n' is
also included.

--- Analysis of 'mathematical formula' implementation ---
The `sum_to_n_formula` function calculates the sum using the direct mathematical formula for the sum of an arithmetic series: `n * (n + 1) / 2`. This method is the most efficient for this specific problem as it avoids iteration entirely, providing a constant tim
complexity (O(1)). It's important to use integer division (`//`) if `n` is an integer to ensure the result is an integer. This approach is highly recommended for performance when applicable. A check for negative 'n' ensures correct behavior.
```

Task Description #5: Classes (Bank Account Class)
Scenario
You are designing a basic banking application.

Code:

In [ ]:

```python
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self.balance = initial_balance
        print(f"Account {self.account_number} created with initial balance: ${self.balance:.2f}")

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount:.2f} into account {self.account_number}. New balance: ${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                print(f"Withdrew ${amount:.2f} from account {self.account_number}. New balance: ${self.balance:.2f}")
            else:
                print(f"Insufficient funds for withdrawal. Current balance: ${self.balance:.2f}")
        else:
            print("Withdrawal amount must be positive.")

    def check_balance(self):
        print(f"Account {self.account_number} current balance: ${self.balance:.2f}")
        return self.balance

# --- Sample Usage and Verification ---
print("\n--- Task 5: Bank Account Class ---")

# Create an account
my_account = BankAccount("12345", 100)

# Check initial balance
my_account.check_balance()

# Perform deposits
my_account.deposit(50)
my_account.deposit(200.50)
my_account.deposit(-10) # Invalid deposit

# Perform withdrawals
my_account.withdraw(75)
my_account.withdraw(300) # Insufficient funds
my_account.withdraw(-20) # Invalid withdrawal

# Check final balance
my_account.check_balance()

print("\n--- Analysis of AI-Generated Code ---")
print("The AI-generated `BankAccount` class is well-structured and implements the requested functionalities correctly. The `__init__` method initializes the account with a number and an optional initial balance. The `deposit` method correctly adds funds, ensur
```

```
--- Task 5: Bank Account Class ---
Account 12345 created with initial balance: $100.00
Account 12345 current balance: $100.00
Deposited $50.00 into account 12345. New balance: $150.00
Deposited $200.50 into account 12345. New balance: $350.50
Deposit amount must be positive.
Withdrew $75.00 from account 12345. New balance: $275.50
Insufficient funds for withdrawal. Current balance: $275.50
Withdrawal amount must be positive.
Account 12345 current balance: $275.50

--- Analysis of AI-Generated Code ---
The AI-generated `BankAccount` class is well-structured and implements the requested functionalities correctly. The `__init__` method initializes the account with a number and an optional initial balance. The `deposit` method correctly adds funds, ensuring the a
ount is positive. The `withdraw` method handles withdrawals by checking for positive amounts and sufficient balance. The `check_balance` method simply displays the current balance. The code includes clear print statements for each operation, providing good feedb
ck on transactions. The class adheres to basic object-oriented principles and is easy to understand.
```