SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE			DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
ProgramName:B. Tech		**Assignment Type: Lab**		**AcademicYear:**2025-2026
CourseCoordinatorName		Venkataramana Veeramsetty		
Instructor(s)Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)		
CourseCode	24CS002PC215	**CourseTitle**	AI Assisted Coding	
Year/Sem	II/I	**Regulation**	R24	
Date and Day of Assignment	Week3 - Wednesday	**Time(s)**		
Duration	2 Hours	**Applicableto Batches**		
AssignmentNumber:6.3(Present assignment number)/**24**(Total number of assignments)				

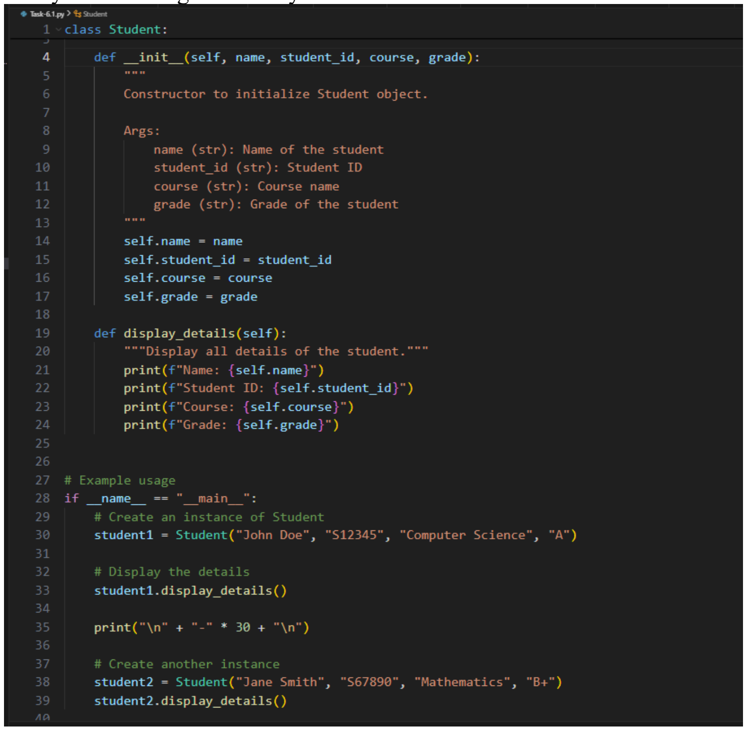
Q.No.	**Question**	***ExpectedTime to complete***
1	Lab 6: AI-Based Code Completion – Classes, Loops, and Conditionals **Lab Objectives:** • To explore AI-powered auto-completion features for core Python constructs. • To analyze how AI suggests logic for class definitions, loops, and conditionals. • To evaluate the completeness and correctness of code generated by AI assistants. **Lab Outcomes (LOs):**	Week3 - Wednesday

After completing this lab, students will be able to:

- Use AI tools to generate and complete class definitions and methods.
- Understand and assess AI-suggested loops for iterative tasks.
- Generate conditional statements through prompt-driven suggestions.
- Critically evaluate AI-assisted code for correctness and clarity.

Task Description#1 (Classes)

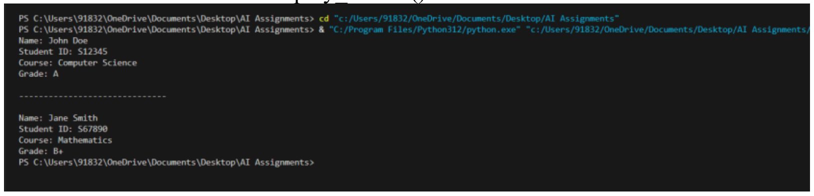
- Use AI to complete a Student class with attributes and a method.
- Check output
- Analyze the code generated by AI tool



```python
class Student:

    def __init__(self, name, student_id, course, grade):
        """
        Constructor to initialize Student object.

        Args:
            name (str): Name of the student
            student_id (str): Student ID
            course (str): Course name
            grade (str): Grade of the student
        """
        self.name = name
        self.student_id = student_id
        self.course = course
        self.grade = grade

    def display_details(self):
        """Display all details of the student."""
        print(f"Name: {self.name}")
        print(f"Student ID: {self.student_id}")
        print(f"Course: {self.course}")
        print(f"Grade: {self.grade}")


# Example usage
if __name__ == "__main__":
    # Create an instance of Student
    student1 = Student("John Doe", "S12345", "Computer Science", "A")

    # Display the details
    student1.display_details()

    print("\n" + "-" * 30 + "\n")

    # Create another instance
    student2 = Student("Jane Smith", "S67890", "Mathematics", "B+")
    student2.display_details()
```

Expected Output#1

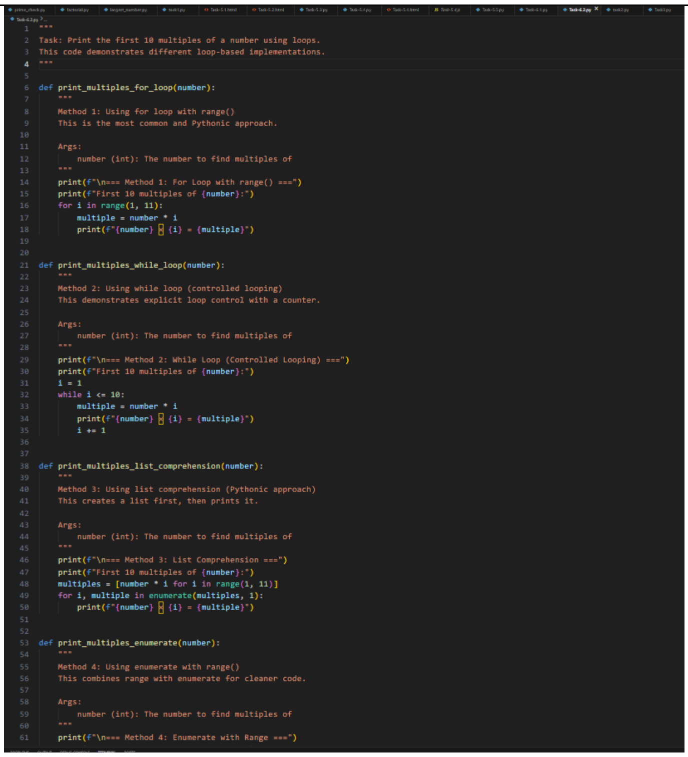
- Class with constructor and display_details() method



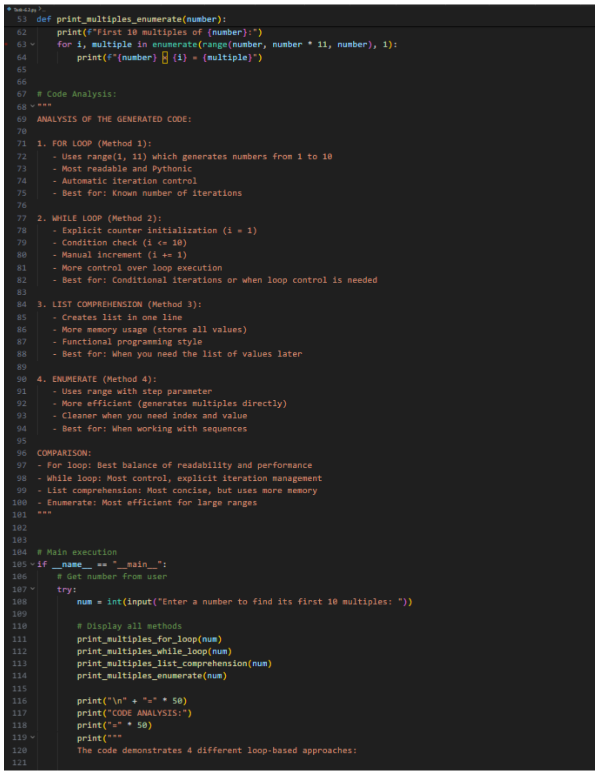
```
PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments> cd "c:/Users/91832/OneDrive/Documents/Desktop/AI Assignments"
PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments> & "C:/Program Files/Python312/python.exe" "c:/Users/91832/OneDrive/Documents/Desktop/AI Assignments/
Name: John Doe
Student ID: S12345
Course: Computer Science
Grade: A

------------------------------

Name: Jane Smith
Student ID: S67890
Course: Mathematics
Grade: B+
PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments>
```

Task Description#2 (Loops)

- Prompt AI to complete a function that prints the first 10 multiples of a number using a loop.
- Analyze the generated code
- Ask AI to generate code using other controlled looping



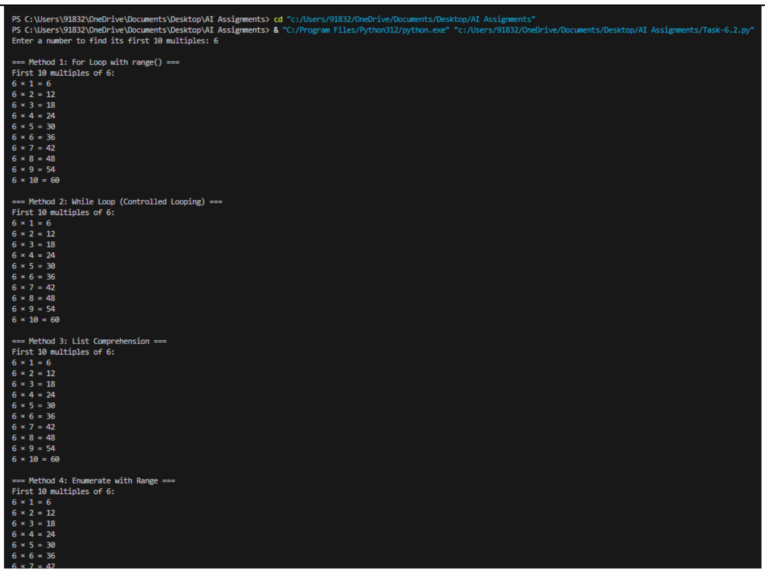
```python
"""
Task: Print the first 10 multiples of a number using loops.
This code demonstrates different loop-based implementations.
"""

def print_multiples_for_loop(number):
    """
    Method 1: Using for loop with range()
    This is the most common and Pythonic approach.

    Args:
        number (int): The number to find multiples of
    """
    print(f"\n=== Method 1: For Loop with range() ===")
    print(f"First 10 multiples of {number}:")
    for i in range(1, 11):
        multiple = number * i
        print(f"{number} × {i} = {multiple}")


def print_multiples_while_loop(number):
    """
    Method 2: Using while loop (controlled looping)
    This demonstrates explicit loop control with a counter.

    Args:
        number (int): The number to find multiples of
    """
    print(f"\n=== Method 2: While Loop (Controlled Looping) ===")
    print(f"First 10 multiples of {number}:")
    i = 1
    while i <= 10:
        multiple = number * i
        print(f"{number} × {i} = {multiple}")
        i += 1


def print_multiples_list_comprehension(number):
    """
    Method 3: Using list comprehension (Pythonic approach)
    This creates a list first, then prints it.

    Args:
        number (int): The number to find multiples of
    """
    print(f"\n=== Method 3: List Comprehension ===")
    print(f"First 10 multiples of {number}:")
    multiples = [number * i for i in range(1, 11)]
    for i, multiple in enumerate(multiples, 1):
        print(f"{number} × {i} = {multiple}")


def print_multiples_enumerate(number):
    """
    Method 4: Using enumerate with range()
    This combines range with enumerate for cleaner code.

    Args:
        number (int): The number to find multiples of
    """
    print(f"\n=== Method 4: Enumerate with Range ===")
```



```python
def print_multiples_enumerate(number):
    print(f"First 10 multiples of {number}:")
    for i, multiple in enumerate(range(number, number * 11, number), 1):
        print(f"{number} × {i} = {multiple}")


# Code Analysis:
"""
ANALYSIS OF THE GENERATED CODE:

1. FOR LOOP (Method 1):
   - Uses range(1, 11) which generates numbers from 1 to 10
   - Most readable and Pythonic
   - Automatic iteration control
   - Best for: Known number of iterations

2. WHILE LOOP (Method 2):
   - Explicit counter initialization (i = 1)
   - Condition check (i <= 10)
   - Manual increment (i += 1)
   - More control over loop execution
   - Best for: Conditional iterations or when loop control is needed

3. LIST COMPREHENSION (Method 3):
   - Creates list in one line
   - More memory usage (stores all values)
   - Functional programming style
   - Best for: When you need the list of values later

4. ENUMERATE (Method 4):
   - Uses range with step parameter
   - More efficient (generates multiples directly)
   - Cleaner when you need index and value
   - Best for: When working with sequences

COMPARISON:
- For loop: Best balance of readability and performance
- While loop: Most control, explicit iteration management
- List comprehension: Most concise, but uses more memory
- Enumerate: Most efficient for large ranges
"""


# Main execution
if __name__ == "__main__":
    # Get number from user
    try:
        num = int(input("Enter a number to find its first 10 multiples: "))

        # Display all methods
        print_multiples_for_loop(num)
        print_multiples_while_loop(num)
        print_multiples_list_comprehension(num)
        print_multiples_enumerate(num)

        print("\n" + "=" * 50)
        print("CODE ANALYSIS:")
        print("=" * 50)
        print("""
        The code demonstrates 4 different loop-based approaches:

```

Expected Output#2

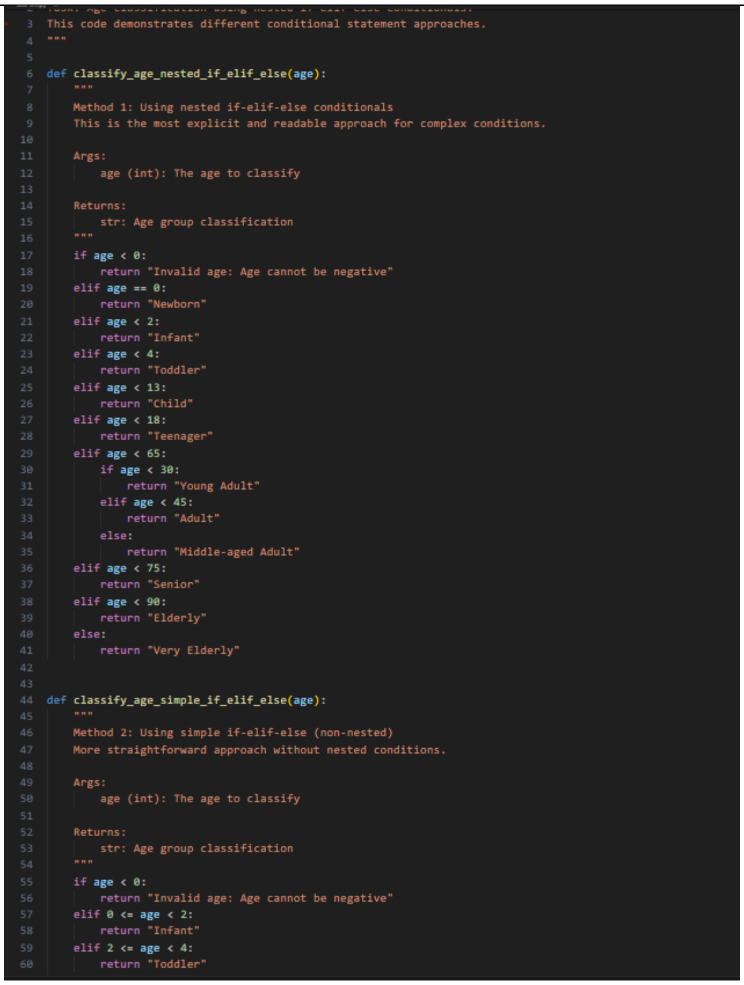
- Correct loop-based implementation



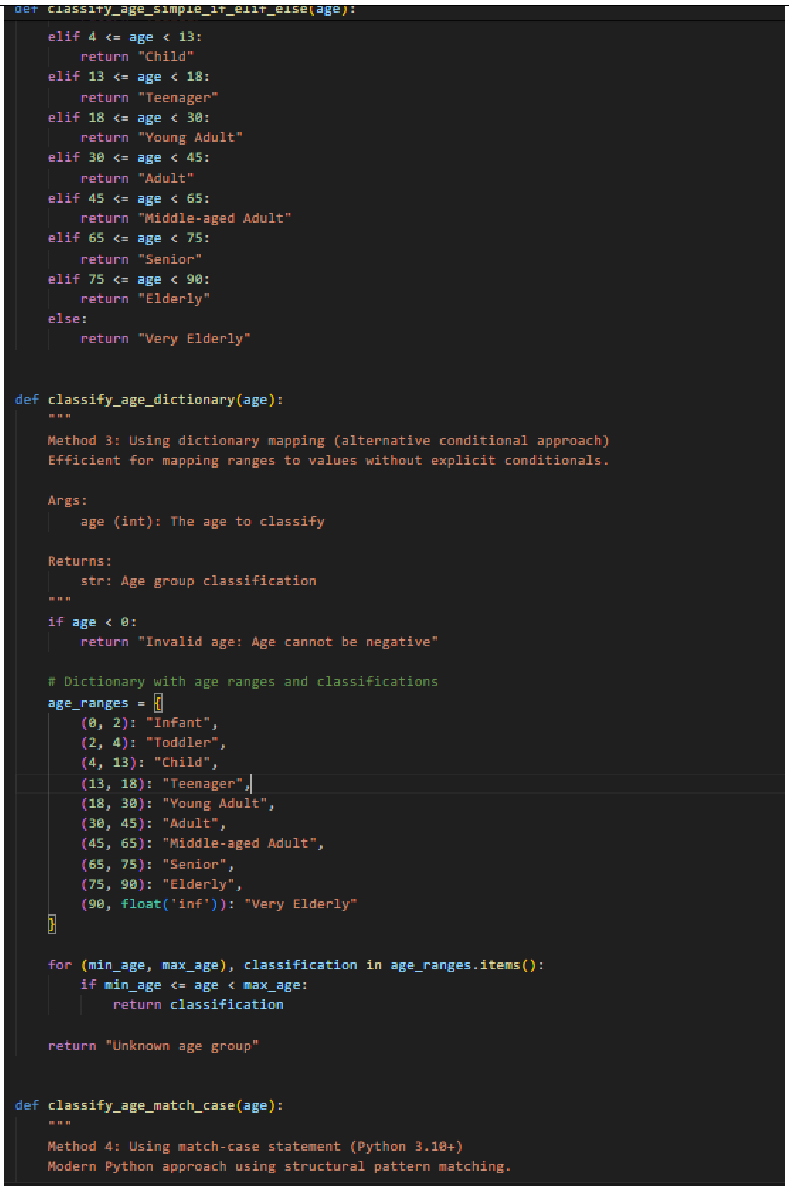
```
PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments> cd "c:/Users/91832/OneDrive/Documents/Desktop/AI Assignments"
PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments> & "C:/Program Files/Python312/python.exe" "c:/Users/91832/OneDrive/Documents/Desktop/AI Assignments/Task-6.2.py"
Enter a number to find its first 10 multiples: 6

=== Method 1: For Loop with range() ===
First 10 multiples of 6:
6 × 1 = 6
6 × 2 = 12
6 × 3 = 18
6 × 4 = 24
6 × 5 = 30
6 × 6 = 36
6 × 7 = 42
6 × 8 = 48
6 × 9 = 54
6 × 10 = 60

=== Method 2: While Loop (Controlled Looping) ===
First 10 multiples of 6:
6 × 1 = 6
6 × 2 = 12
6 × 3 = 18
6 × 4 = 24
6 × 5 = 30
6 × 6 = 36
6 × 7 = 42
6 × 8 = 48
6 × 9 = 54
6 × 10 = 60

=== Method 3: List Comprehension ===
First 10 multiples of 6:
6 × 1 = 6
6 × 2 = 12
6 × 3 = 18
6 × 4 = 24
6 × 5 = 30
6 × 6 = 36
6 × 7 = 42
6 × 8 = 48
6 × 9 = 54
6 × 10 = 60

=== Method 4: Enumerate with Range ===
First 10 multiples of 6:
6 × 1 = 6
6 × 2 = 12
6 × 3 = 18
6 × 4 = 24
6 × 5 = 30
6 × 6 = 36
6 × 7 = 42
```

Task Description#3 (Conditional Statements)

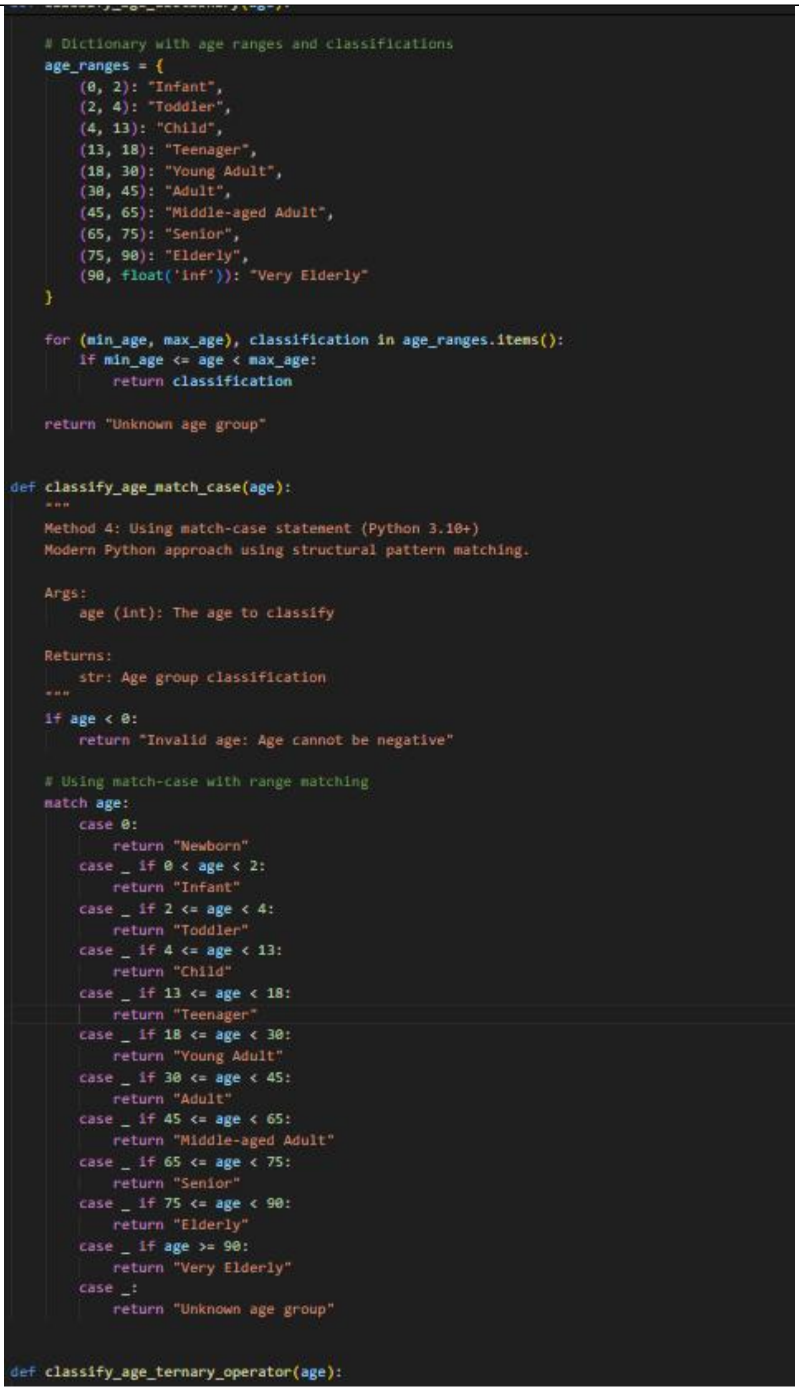
- Ask AI to write nested if-elif-else conditionals to classify age groups.
- Analyze the generated code
- Ask AI to generate code using other conditional statements



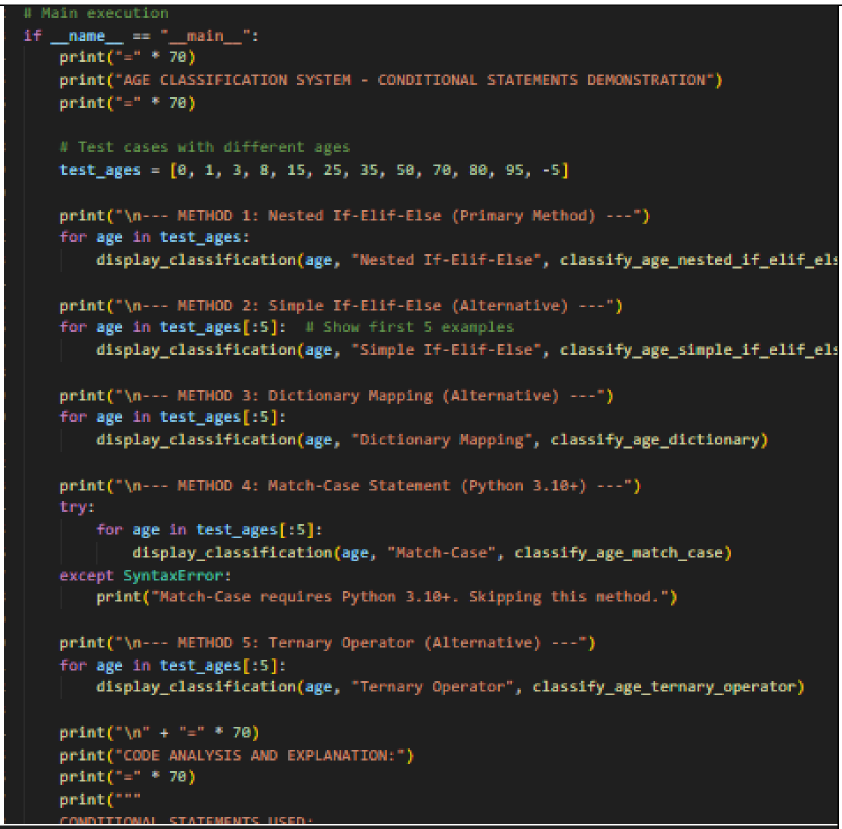
```python
This code demonstrates different conditional statement approaches.
"""

def classify_age_nested_if_elif_else(age):
    """
    Method 1: Using nested if-elif-else conditionals
    This is the most explicit and readable approach for complex conditions.

    Args:
        age (int): The age to classify

    Returns:
        str: Age group classification
    """
    if age < 0:
        return "Invalid age: Age cannot be negative"
    elif age == 0:
        return "Newborn"
    elif age < 2:
        return "Infant"
    elif age < 4:
        return "Toddler"
    elif age < 13:
        return "Child"
    elif age < 18:
        return "Teenager"
    elif age < 65:
        if age < 30:
            return "Young Adult"
        elif age < 45:
            return "Adult"
        else:
            return "Middle-aged Adult"
    elif age < 75:
        return "Senior"
    elif age < 90:
        return "Elderly"
    else:
        return "Very Elderly"


def classify_age_simple_if_elif_else(age):
    """
    Method 2: Using simple if-elif-else (non-nested)
    More straightforward approach without nested conditions.

    Args:
        age (int): The age to classify

    Returns:
        str: Age group classification
    """
    if age < 0:
        return "Invalid age: Age cannot be negative"
    elif 0 <= age < 2:
        return "Infant"
    elif 2 <= age < 4:
        return "Toddler"
```



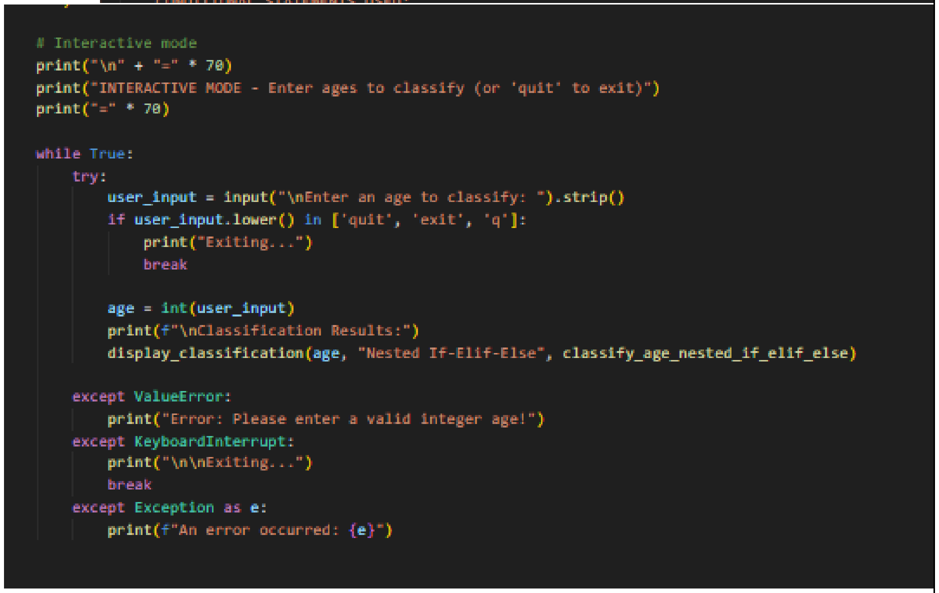
```python
def classify_age_simple_if_elif_else(age):
    elif 4 <= age < 13:
        return "Child"
    elif 13 <= age < 18:
        return "Teenager"
    elif 18 <= age < 30:
        return "Young Adult"
    elif 30 <= age < 45:
        return "Adult"
    elif 45 <= age < 65:
        return "Middle-aged Adult"
    elif 65 <= age < 75:
        return "Senior"
    elif 75 <= age < 90:
        return "Elderly"
    else:
        return "Very Elderly"


def classify_age_dictionary(age):
    """
    Method 3: Using dictionary mapping (alternative conditional approach)
    Efficient for mapping ranges to values without explicit conditionals.

    Args:
        age (int): The age to classify

    Returns:
        str: Age group classification
    """
    if age < 0:
        return "Invalid age: Age cannot be negative"

    # Dictionary with age ranges and classifications
    age_ranges = {
        (0, 2): "Infant",
        (2, 4): "Toddler",
        (4, 13): "Child",
        (13, 18): "Teenager",
        (18, 30): "Young Adult",
        (30, 45): "Adult",
        (45, 65): "Middle-aged Adult",
        (65, 75): "Senior",
        (75, 90): "Elderly",
        (90, float('inf')): "Very Elderly"
    }

    for (min_age, max_age), classification in age_ranges.items():
        if min_age <= age < max_age:
            return classification

    return "Unknown age group"


def classify_age_match_case(age):
    """
    Method 4: Using match-case statement (Python 3.10+)
    Modern Python approach using structural pattern matching.
```



```python
    # Dictionary with age ranges and classifications
    age_ranges = {
        (0, 2): "Infant",
        (2, 4): "Toddler",
        (4, 13): "Child",
        (13, 18): "Teenager",
        (18, 30): "Young Adult",
        (30, 45): "Adult",
        (45, 65): "Middle-aged Adult",
        (65, 75): "Senior",
        (75, 90): "Elderly",
        (90, float('inf')): "Very Elderly"
    }

    for (min_age, max_age), classification in age_ranges.items():
        if min_age <= age < max_age:
            return classification

    return "Unknown age group"


def classify_age_match_case(age):
    """
    Method 4: Using match-case statement (Python 3.10+)
    Modern Python approach using structural pattern matching.

    Args:
        age (int): The age to classify

    Returns:
        str: Age group classification
    """
    if age < 0:
        return "Invalid age: Age cannot be negative"

    # Using match-case with range matching
    match age:
        case 0:
            return "Newborn"
        case _ if 0 < age < 2:
            return "Infant"
        case _ if 2 <= age < 4:
            return "Toddler"
        case _ if 4 <= age < 13:
            return "Child"
        case _ if 13 <= age < 18:
            return "Teenager"
        case _ if 18 <= age < 30:
            return "Young Adult"
        case _ if 30 <= age < 45:
            return "Adult"
        case _ if 45 <= age < 65:
            return "Middle-aged Adult"
        case _ if 65 <= age < 75:
            return "Senior"
        case _ if 75 <= age < 90:
            return "Elderly"
        case _ if age >= 90:
            return "Very Elderly"
        case _:
            return "Unknown age group"


def classify_age_ternary_operator(age):
```



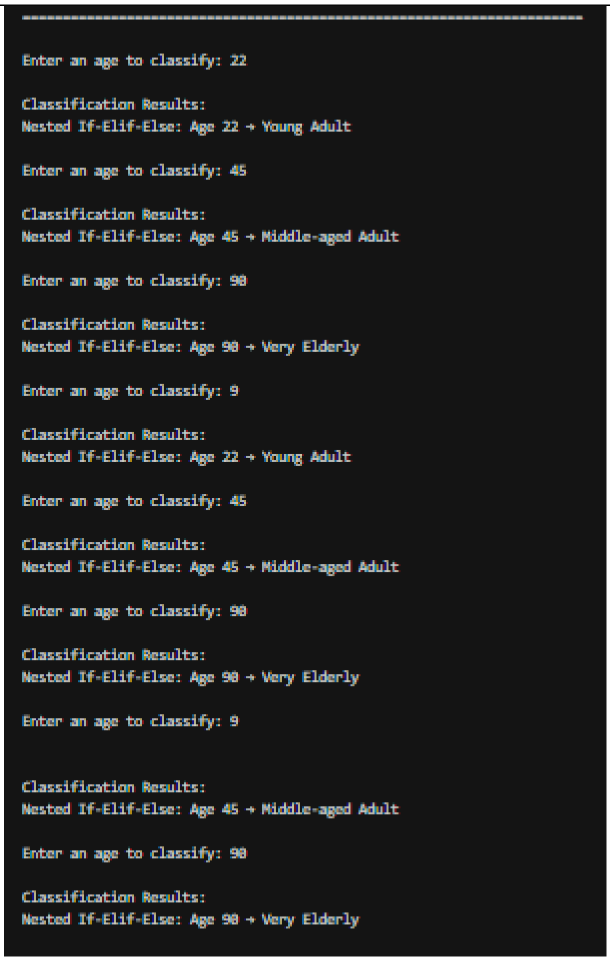
```
# Main execution
if __name__ == "__main__":
    print("=" * 70)
    print("AGE CLASSIFICATION SYSTEM - CONDITIONAL STATEMENTS DEMONSTRATION")
    print("=" * 70)

    # Test cases with different ages
    test_ages = [0, 1, 3, 8, 15, 25, 35, 50, 70, 80, 95, -5]

    print("\n--- METHOD 1: Nested If-Elif-Else (Primary Method) ---")
    for age in test_ages:
        display_classification(age, "Nested If-Elif-Else", classify_age_nested_if_elif_els

    print("\n--- METHOD 2: Simple If-Elif-Else (Alternative) ---")
    for age in test_ages[:5]:  # Show first 5 examples
        display_classification(age, "Simple If-Elif-Else", classify_age_simple_if_elif_els

    print("\n--- METHOD 3: Dictionary Mapping (Alternative) ---")
    for age in test_ages[:5]:
        display_classification(age, "Dictionary Mapping", classify_age_dictionary)

    print("\n--- METHOD 4: Match-Case Statement (Python 3.10+) ---")
    try:
        for age in test_ages[:5]:
            display_classification(age, "Match-Case", classify_age_match_case)
    except SyntaxError:
        print("Match-Case requires Python 3.10+. Skipping this method.")

    print("\n--- METHOD 5: Ternary Operator (Alternative) ---")
    for age in test_ages[:5]:
        display_classification(age, "Ternary Operator", classify_age_ternary_operator)

    print("\n" + "=" * 70)
    print("CODE ANALYSIS AND EXPLANATION:")
    print("=" * 70)
    print("""
    CONDITIONAL STATEMENTS USED:
```



```
    # Interactive mode
    print("\n" + "=" * 70)
    print("INTERACTIVE MODE - Enter ages to classify (or 'quit' to exit)")
    print("=" * 70)

    while True:
        try:
            user_input = input("\nEnter an age to classify: ").strip()
            if user_input.lower() in ['quit', 'exit', 'q']:
                print("Exiting...")
                break

            age = int(user_input)
            print(f"\nClassification Results:")
            display_classification(age, "Nested If-Elif-Else", classify_age_nested_if_elif_else)

        except ValueError:
            print("Error: Please enter a valid integer age!")
        except KeyboardInterrupt:
            print("\n\nExiting...")
            break
        except Exception as e:
            print(f"An error occurred: {e}")
```

Expected Output#3

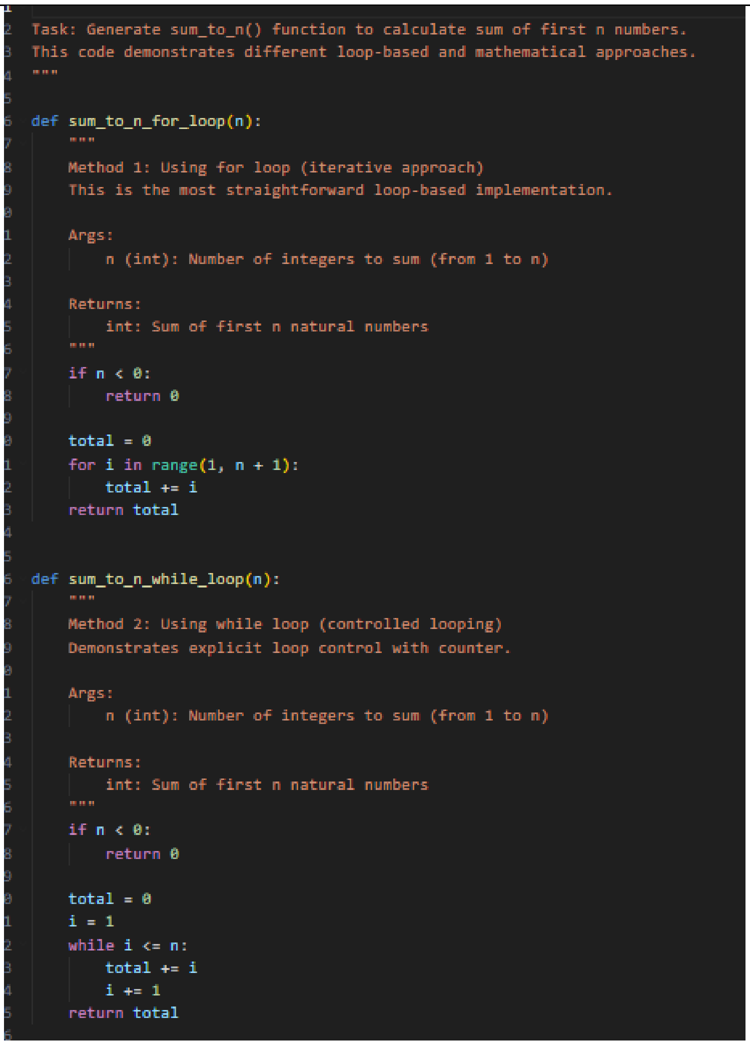
- Age classification function with appropriate conditions and with explanation



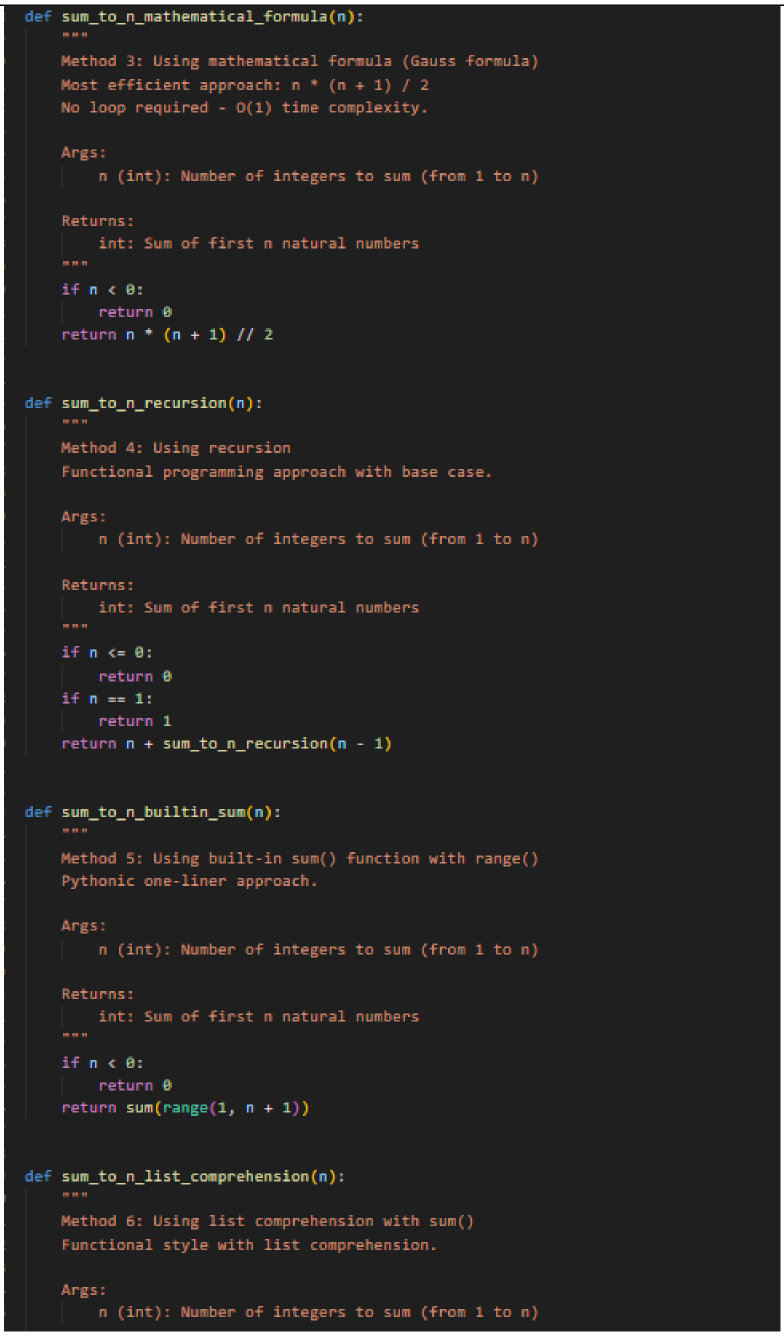
```
======================================================================

Enter an age to classify: 22

Classification Results:
Nested If-Elif-Else: Age 22 → Young Adult

Enter an age to classify: 45

Classification Results:
Nested If-Elif-Else: Age 45 → Middle-aged Adult

Enter an age to classify: 90

Classification Results:
Nested If-Elif-Else: Age 90 → Very Elderly

Enter an age to classify: 9

Classification Results:
Nested If-Elif-Else: Age 22 → Young Adult

Enter an age to classify: 45

Classification Results:
Nested If-Elif-Else: Age 45 → Middle-aged Adult

Enter an age to classify: 90

Classification Results:
Nested If-Elif-Else: Age 90 → Very Elderly

Enter an age to classify: 9


Classification Results:
Nested If-Elif-Else: Age 45 → Middle-aged Adult

Enter an age to classify: 90

Classification Results:
Nested If-Elif-Else: Age 90 → Very Elderly
```

Task Description#4 (For and While loops)

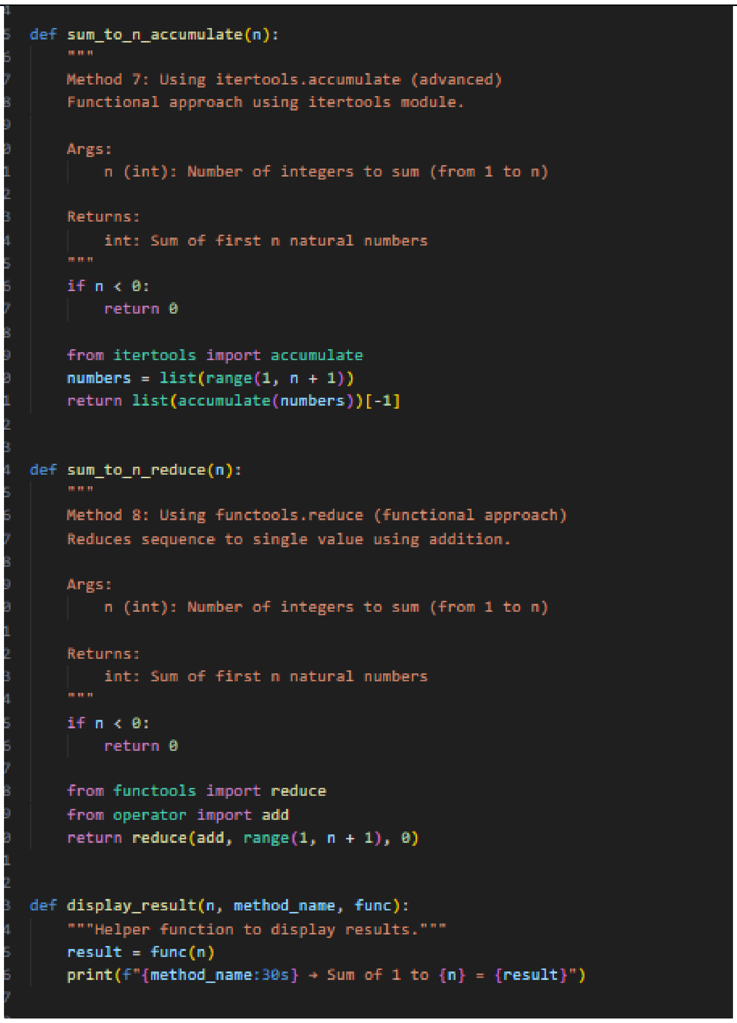
- Generate a sum_to_n() function to calculate sum of first n numbers
- Analyze the generated code
- Get suggestions from AI with other controlled looping



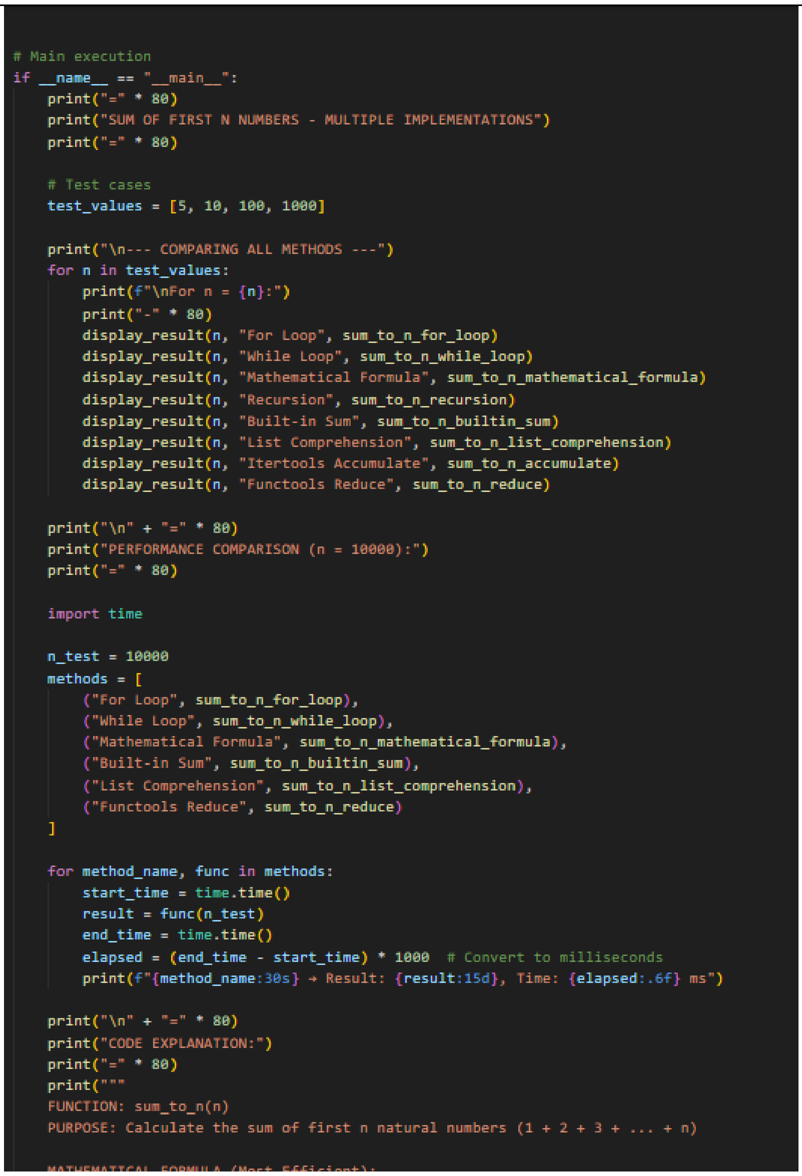
```python
Task: Generate sum_to_n() function to calculate sum of first n numbers.
This code demonstrates different loop-based and mathematical approaches.
"""


def sum_to_n_for_loop(n):
    """
    Method 1: Using for loop (iterative approach)
    This is the most straightforward loop-based implementation.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0

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


def sum_to_n_while_loop(n):
    """
    Method 2: Using while loop (controlled looping)
    Demonstrates explicit loop control with counter.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0

    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total
```



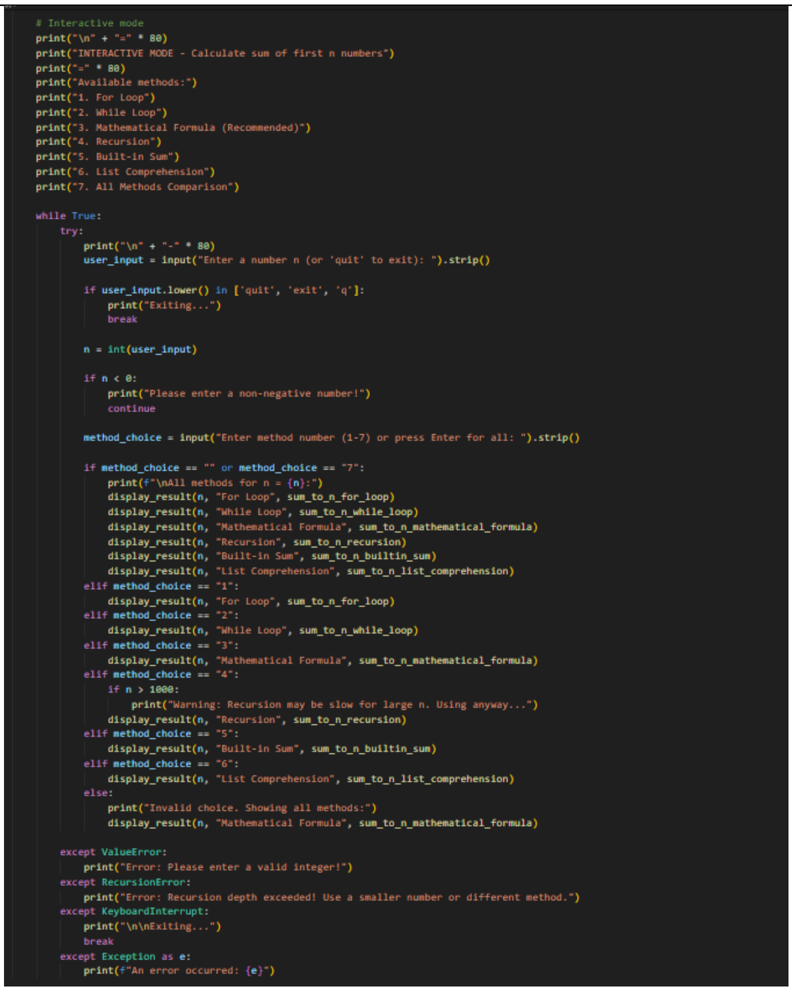
```python
def sum_to_n_mathematical_formula(n):
    """
    Method 3: Using mathematical formula (Gauss formula)
    Most efficient approach: n * (n + 1) / 2
    No loop required - O(1) time complexity.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0
    return n * (n + 1) // 2


def sum_to_n_recursion(n):
    """
    Method 4: Using recursion
    Functional programming approach with base case.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n <= 0:
        return 0
    if n == 1:
        return 1
    return n + sum_to_n_recursion(n - 1)


def sum_to_n_builtin_sum(n):
    """
    Method 5: Using built-in sum() function with range()
    Pythonic one-liner approach.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0
    return sum(range(1, n + 1))


def sum_to_n_list_comprehension(n):
    """
    Method 6: Using list comprehension with sum()
    Functional style with list comprehension.

    Args:
        n (int): Number of integers to sum (from 1 to n)
```

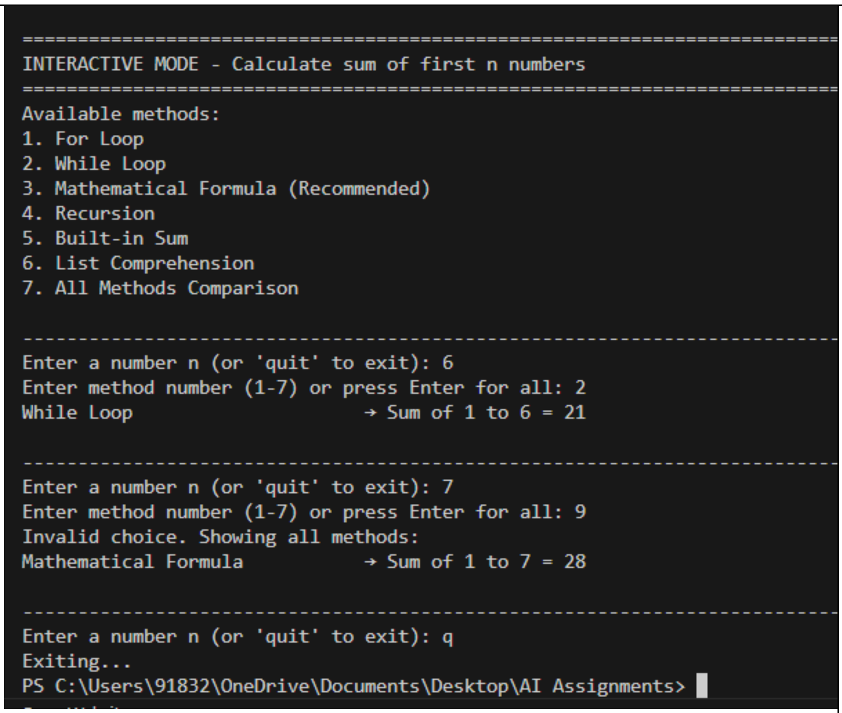


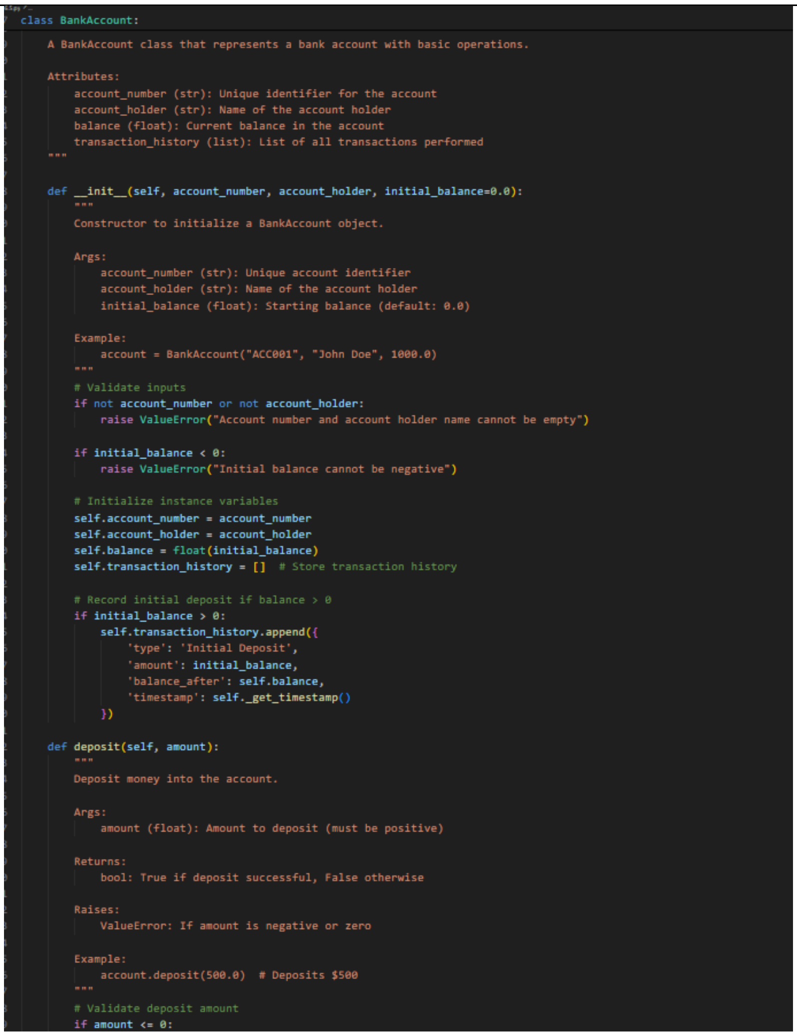
```python
def sum_to_n_accumulate(n):
    """
    Method 7: Using itertools.accumulate (advanced)
    Functional approach using itertools module.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0

    from itertools import accumulate
    numbers = list(range(1, n + 1))
    return list(accumulate(numbers))[-1]


def sum_to_n_reduce(n):
    """
    Method 8: Using functools.reduce (functional approach)
    Reduces sequence to single value using addition.

    Args:
        n (int): Number of integers to sum (from 1 to n)

    Returns:
        int: Sum of first n natural numbers
    """
    if n < 0:
        return 0

    from functools import reduce
    from operator import add
    return reduce(add, range(1, n + 1), 0)


def display_result(n, method_name, func):
    """Helper function to display results."""
    result = func(n)
    print(f"{method_name:30s} → Sum of 1 to {n} = {result}")
```



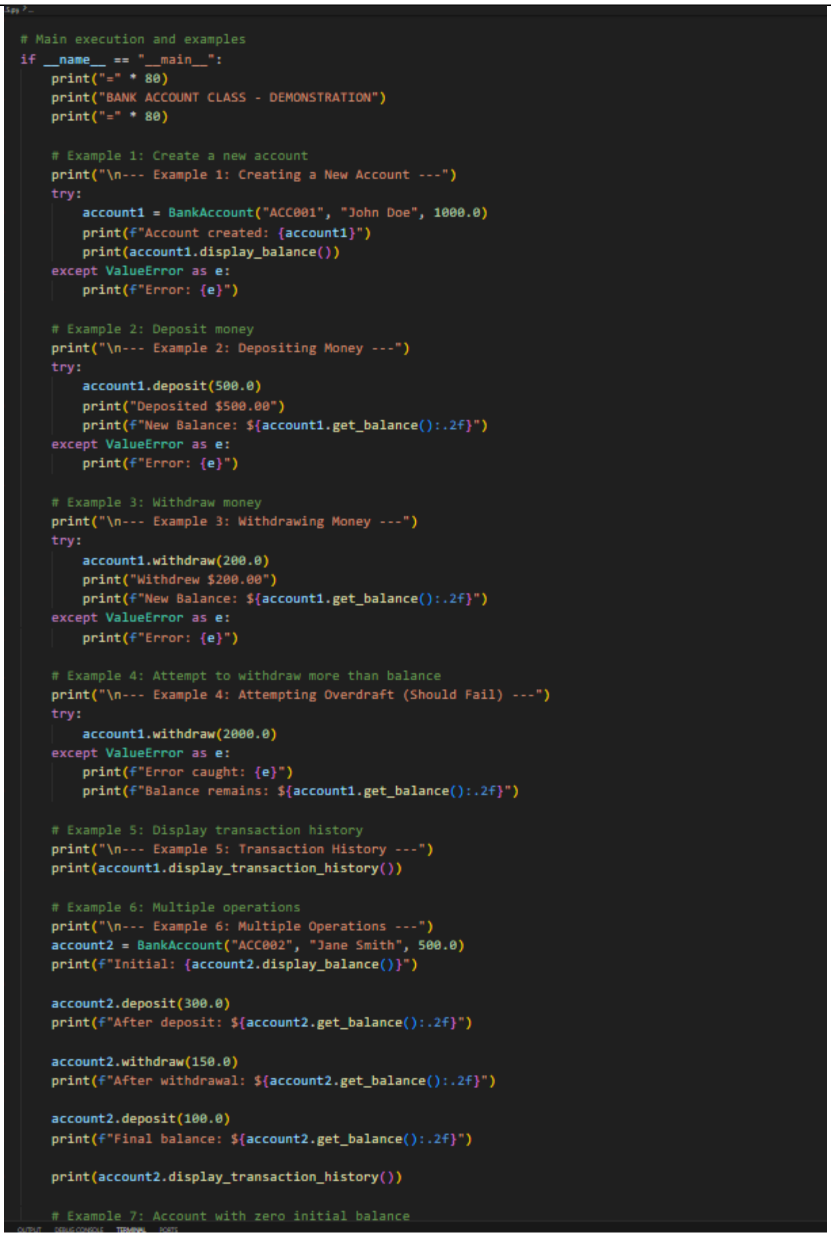
```python
# Main execution
if __name__ == "__main__":
    print("=" * 80)
    print("SUM OF FIRST N NUMBERS - MULTIPLE IMPLEMENTATIONS")
    print("=" * 80)

    # Test cases
    test_values = [5, 10, 100, 1000]

    print("\n--- COMPARING ALL METHODS ---")
    for n in test_values:
        print(f"\nFor n = {n}:")
        print("-" * 80)
        display_result(n, "For Loop", sum_to_n_for_loop)
        display_result(n, "While Loop", sum_to_n_while_loop)
        display_result(n, "Mathematical Formula", sum_to_n_mathematical_formula)
        display_result(n, "Recursion", sum_to_n_recursion)
        display_result(n, "Built-in Sum", sum_to_n_builtin_sum)
        display_result(n, "List Comprehension", sum_to_n_list_comprehension)
        display_result(n, "Itertools Accumulate", sum_to_n_accumulate)
        display_result(n, "Functools Reduce", sum_to_n_reduce)

    print("\n" + "=" * 80)
    print("PERFORMANCE COMPARISON (n = 10000):")
    print("=" * 80)

    import time

    n_test = 10000
    methods = [
        ("For Loop", sum_to_n_for_loop),
        ("While Loop", sum_to_n_while_loop),
        ("Mathematical Formula", sum_to_n_mathematical_formula),
        ("Built-in Sum", sum_to_n_builtin_sum),
        ("List Comprehension", sum_to_n_list_comprehension),
        ("Functools Reduce", sum_to_n_reduce)
    ]

    for method_name, func in methods:
        start_time = time.time()
        result = func(n_test)
        end_time = time.time()
        elapsed = (end_time - start_time) * 1000  # Convert to milliseconds
        print(f"{method_name:30s} → Result: {result:15d}, Time: {elapsed:.6f} ms")

    print("\n" + "=" * 80)
    print("CODE EXPLANATION:")
    print("=" * 80)
    print("""
    FUNCTION: sum_to_n(n)
    PURPOSE: Calculate the sum of first n natural numbers (1 + 2 + 3 + ... + n)

    MATHEMATICAL FORMULA (Most Efficient):
```

	
	```
	# Interactive mode
	print("\n" + "=" * 80)
	print("INTERACTIVE MODE - Calculate sum of first n numbers")
	print("=" * 80)
	print("Available methods:")
	print("1. For Loop")
	print("2. While Loop")
	print("3. Mathematical Formula (Recommended)")
	print("4. Recursion")
	print("5. Built-in Sum")
	print("6. List Comprehension")
	print("7. All Methods Comparison")

	while True:
	    try:
	        print("\n" + "-" * 80)
	        user_input = input("Enter a number n (or 'quit' to exit): ").strip()

	        if user_input.lower() in ['quit', 'exit', 'q']:
	            print("Exiting...")
	            break

	        n = int(user_input)

	        if n < 0:
	            print("Please enter a non-negative number!")
	            continue

	        method_choice = input("Enter method number (1-7) or press Enter for all: ").strip()

	        if method_choice == "" or method_choice == "7":
	            print(f"\nAll methods for n = {n}:")
	            display_result(n, "For Loop", sum_to_n_for_loop)
	            display_result(n, "While Loop", sum_to_n_while_loop)
	            display_result(n, "Mathematical Formula", sum_to_n_mathematical_formula)
	            display_result(n, "Recursion", sum_to_n_recursion)
	            display_result(n, "Built-in Sum", sum_to_n_builtin_sum)
	            display_result(n, "List Comprehension", sum_to_n_list_comprehension)
	        elif method_choice == "1":
	            display_result(n, "For Loop", sum_to_n_for_loop)
	        elif method_choice == "2":
	            display_result(n, "While Loop", sum_to_n_while_loop)
	        elif method_choice == "3":
	            display_result(n, "Mathematical Formula", sum_to_n_mathematical_formula)
	        elif method_choice == "4":
	            if n > 1000:
	                print("Warning: Recursion may be slow for large n. Using anyway...")
	            display_result(n, "Recursion", sum_to_n_recursion)
	        elif method_choice == "5":
	            display_result(n, "Built-in Sum", sum_to_n_builtin_sum)
	        elif method_choice == "6":
	            display_result(n, "List Comprehension", sum_to_n_list_comprehension)
	        else:
	            print("Invalid choice. Showing all methods:")
	            display_result(n, "Mathematical Formula", sum_to_n_mathematical_formula)

	    except ValueError:
	        print("Error: Please enter a valid integer!")
	    except RecursionError:
	        print("Error: Recursion depth exceeded! Use a smaller number or different method.")
	    except KeyboardInterrupt:
	        print("\n\nExiting...")
	        break
	    except Exception as e:
	        print(f"An error occurred: {e}")
	```
	
	Expected Output#4
	• Python code with explanation	

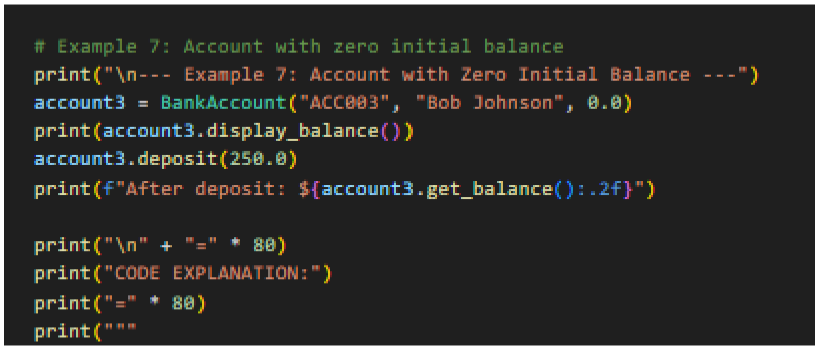
	
	```
	=====================================================================
	INTERACTIVE MODE - Calculate sum of first n numbers
	=====================================================================
	Available methods:
	1. For Loop
	2. While Loop
	3. Mathematical Formula (Recommended)
	4. Recursion
	5. Built-in Sum
	6. List Comprehension
	7. All Methods Comparison

	---------------------------------------------------------------------
	Enter a number n (or 'quit' to exit): 6
	Enter method number (1-7) or press Enter for all: 2
	While Loop                    → Sum of 1 to 6 = 21

	---------------------------------------------------------------------
	Enter a number n (or 'quit' to exit): 7
	Enter method number (1-7) or press Enter for all: 9
	Invalid choice. Showing all methods:
	Mathematical Formula          → Sum of 1 to 7 = 28

	---------------------------------------------------------------------
	Enter a number n (or 'quit' to exit): q
	Exiting...
	PS C:\Users\91832\OneDrive\Documents\Desktop\AI Assignments>
	```
	Task Description#5 (Class)
	• Use AI to build a BankAccount class with deposit, withdraw, and balance methods.
	• Analyze the generated code
	• Add comments and explain code	

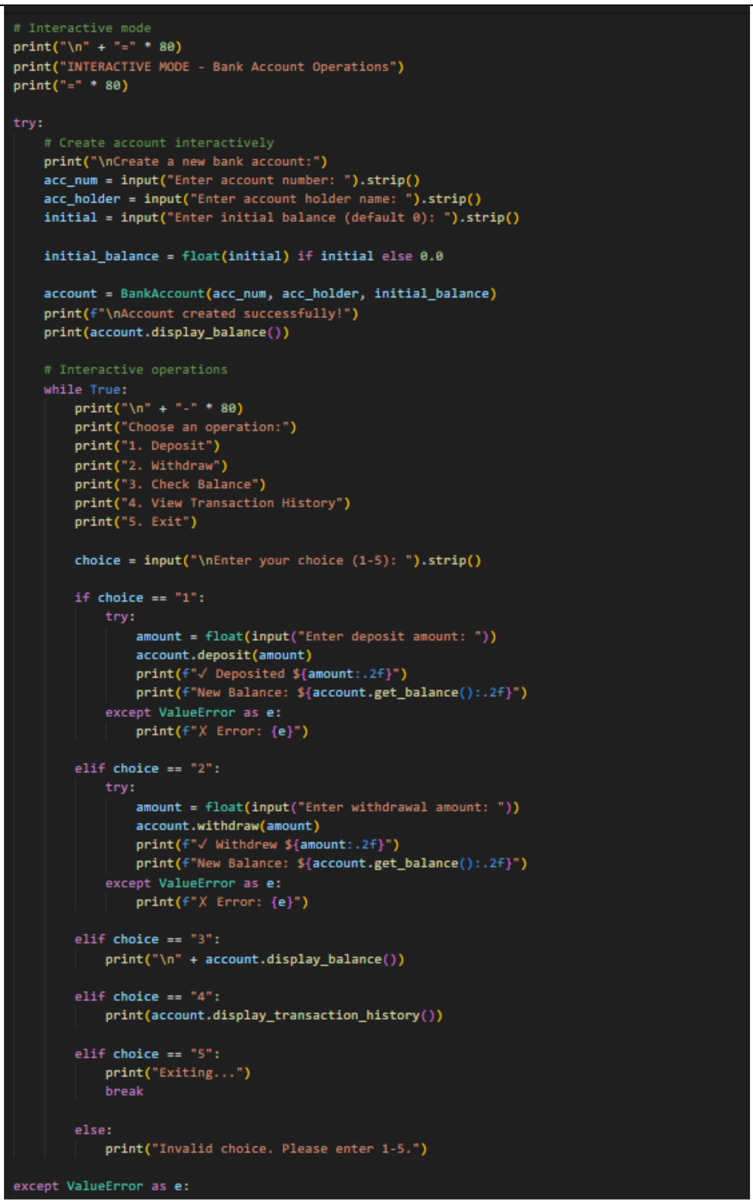
```python
class BankAccount:

    A BankAccount class that represents a bank account with basic operations.

    Attributes:
        account_number (str): Unique identifier for the account
        account_holder (str): Name of the account holder
        balance (float): Current balance in the account
        transaction_history (list): List of all transactions performed
    """

    def __init__(self, account_number, account_holder, initial_balance=0.0):
        """
        Constructor to initialize a BankAccount object.

        Args:
            account_number (str): Unique account identifier
            account_holder (str): Name of the account holder
            initial_balance (float): Starting balance (default: 0.0)

        Example:
            account = BankAccount("ACC001", "John Doe", 1000.0)
        """
        # Validate inputs
        if not account_number or not account_holder:
            raise ValueError("Account number and account holder name cannot be empty")

        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative")

        # Initialize instance variables
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = float(initial_balance)
        self.transaction_history = []  # Store transaction history

        # Record initial deposit if balance > 0
        if initial_balance > 0:
            self.transaction_history.append({
                'type': 'Initial Deposit',
                'amount': initial_balance,
                'balance_after': self.balance,
                'timestamp': self._get_timestamp()
            })

    def deposit(self, amount):
        """
        Deposit money into the account.

        Args:
            amount (float): Amount to deposit (must be positive)

        Returns:
            bool: True if deposit successful, False otherwise

        Raises:
            ValueError: If amount is negative or zero

        Example:
            account.deposit(500.0)  # Deposits $500
        """
        # Validate deposit amount
        if amount <= 0:
```



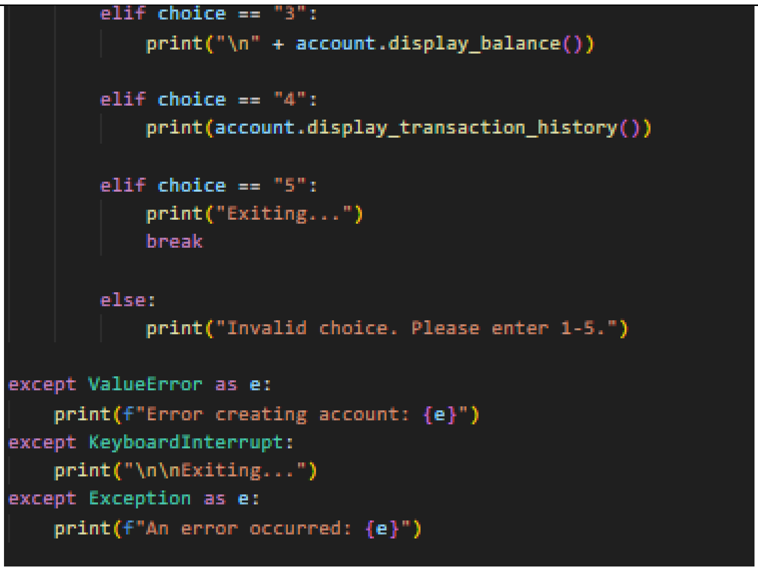
```python
# Main execution and examples
if __name__ == "__main__":
    print("=" * 80)
    print("BANK ACCOUNT CLASS - DEMONSTRATION")
    print("=" * 80)

    # Example 1: Create a new account
    print("\n--- Example 1: Creating a New Account ---")
    try:
        account1 = BankAccount("ACC001", "John Doe", 1000.0)
        print(f"Account created: {account1}")
        print(account1.display_balance())
    except ValueError as e:
        print(f"Error: {e}")

    # Example 2: Deposit money
    print("\n--- Example 2: Depositing Money ---")
    try:
        account1.deposit(500.0)
        print("Deposited $500.00")
        print(f"New Balance: ${account1.get_balance():.2f}")
    except ValueError as e:
        print(f"Error: {e}")

    # Example 3: Withdraw money
    print("\n--- Example 3: Withdrawing Money ---")
    try:
        account1.withdraw(200.0)
        print("Withdrew $200.00")
        print(f"New Balance: ${account1.get_balance():.2f}")
    except ValueError as e:
        print(f"Error: {e}")

    # Example 4: Attempt to withdraw more than balance
    print("\n--- Example 4: Attempting Overdraft (Should Fail) ---")
    try:
        account1.withdraw(2000.0)
    except ValueError as e:
        print(f"Error caught: {e}")
        print(f"Balance remains: ${account1.get_balance():.2f}")

    # Example 5: Display transaction history
    print("\n--- Example 5: Transaction History ---")
    print(account1.display_transaction_history())

    # Example 6: Multiple operations
    print("\n--- Example 6: Multiple Operations ---")
    account2 = BankAccount("ACC002", "Jane Smith", 500.0)
    print(f"Initial: {account2.display_balance()}")

    account2.deposit(300.0)
    print(f"After deposit: ${account2.get_balance():.2f}")

    account2.withdraw(150.0)
    print(f"After withdrawal: ${account2.get_balance():.2f}")

    account2.deposit(100.0)
    print(f"Final balance: ${account2.get_balance():.2f}")

    print(account2.display_transaction_history())

    # Example 7: Account with zero initial balance
```



```python
# Example 7: Account with zero initial balance
print("\n--- Example 7: Account with Zero Initial Balance ---")
account3 = BankAccount("ACC003", "Bob Johnson", 0.0)
print(account3.display_balance())
account3.deposit(250.0)
print(f"After deposit: ${account3.get_balance():.2f}")

print("\n" + "=" * 80)
print("CODE EXPLANATION:")
print("=" * 80)
print("""
```



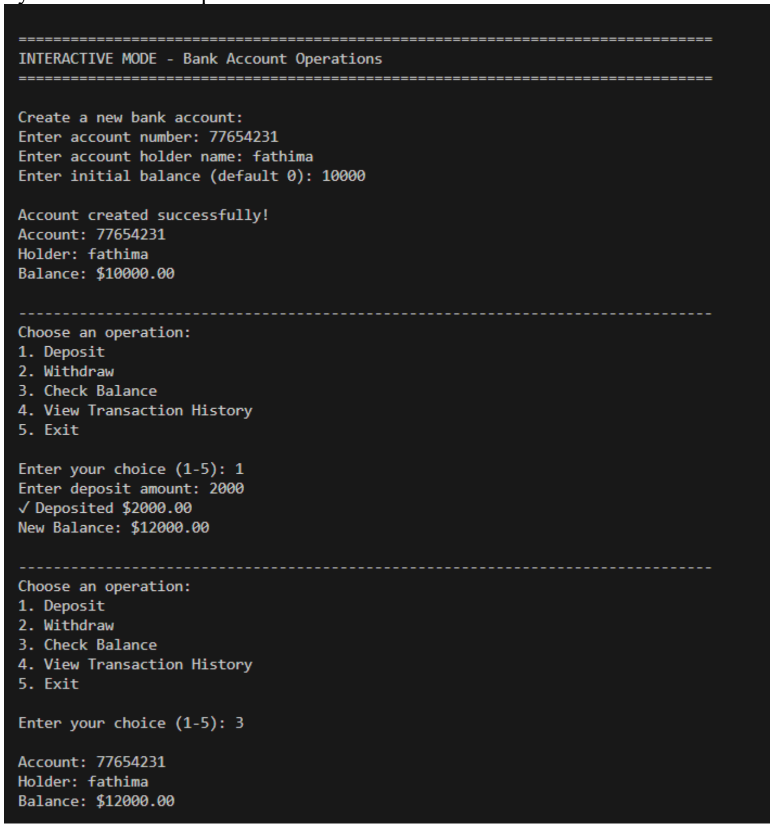
```python
# Interactive mode
print("\n" + "=" * 80)
print("INTERACTIVE MODE - Bank Account Operations")
print("=" * 80)

try:
    # Create account interactively
    print("\nCreate a new bank account:")
    acc_num = input("Enter account number: ").strip()
    acc_holder = input("Enter account holder name: ").strip()
    initial = input("Enter initial balance (default 0): ").strip()

    initial_balance = float(initial) if initial else 0.0

    account = BankAccount(acc_num, acc_holder, initial_balance)
    print(f"\nAccount created successfully!")
    print(account.display_balance())

    # Interactive operations
    while True:
        print("\n" + "-" * 80)
        print("Choose an operation:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Check Balance")
        print("4. View Transaction History")
        print("5. Exit")

        choice = input("\nEnter your choice (1-5): ").strip()

        if choice == "1":
            try:
                amount = float(input("Enter deposit amount: "))
                account.deposit(amount)
                print(f"✓ Deposited ${amount:.2f}")
                print(f"New Balance: ${account.get_balance():.2f}")
            except ValueError as e:
                print(f"✗ Error: {e}")

        elif choice == "2":
            try:
                amount = float(input("Enter withdrawal amount: "))
                account.withdraw(amount)
                print(f"✓ Withdrew ${amount:.2f}")
                print(f"New Balance: ${account.get_balance():.2f}")
            except ValueError as e:
                print(f"✗ Error: {e}")

        elif choice == "3":
            print("\n" + account.display_balance())

        elif choice == "4":
            print(account.display_transaction_history())

        elif choice == "5":
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please enter 1-5.")

except ValueError as e:
```



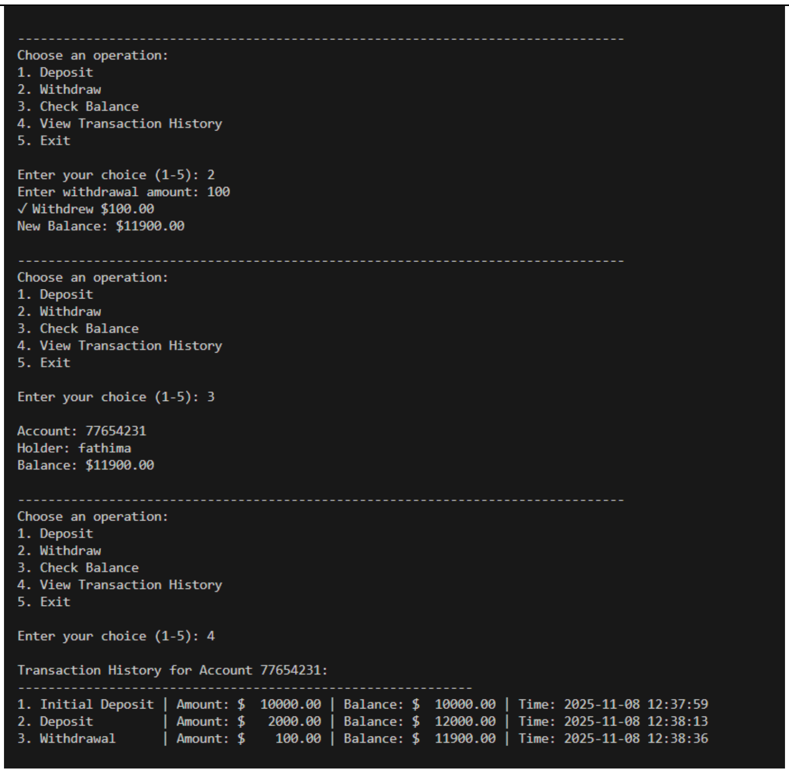
```
        elif choice == "3":
            print("\n" + account.display_balance())

        elif choice == "4":
            print(account.display_transaction_history())

        elif choice == "5":
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please enter 1-5.")

except ValueError as e:
    print(f"Error creating account: {e}")
except KeyboardInterrupt:
    print("\n\nExiting...")
except Exception as e:
    print(f"An error occurred: {e}")
```

Expected Output#5

- Python code with explanation



```
================================================================================
INTERACTIVE MODE - Bank Account Operations
================================================================================

Create a new bank account:
Enter account number: 77654231
Enter account holder name: fathima
Enter initial balance (default 0): 10000

Account created successfully!
Account: 77654231
Holder: fathima
Balance: $10000.00

--------------------------------------------------------------------------------
Choose an operation:
1. Deposit
2. Withdraw
3. Check Balance
4. View Transaction History
5. Exit

Enter your choice (1-5): 1
Enter deposit amount: 2000
✓ Deposited $2000.00
New Balance: $12000.00

--------------------------------------------------------------------------------
Choose an operation:
1. Deposit
2. Withdraw
3. Check Balance
4. View Transaction History
5. Exit

Enter your choice (1-5): 3

Account: 77654231
Holder: fathima
Balance: $12000.00
```



```
--------------------------------------------------------------------------------
Choose an operation:
1. Deposit
2. Withdraw
3. Check Balance
4. View Transaction History
5. Exit

Enter your choice (1-5): 2
Enter withdrawal amount: 100
✓ Withdrew $100.00
New Balance: $11900.00

--------------------------------------------------------------------------------
Choose an operation:
1. Deposit
2. Withdraw
3. Check Balance
4. View Transaction History
5. Exit

Enter your choice (1-5): 3

Account: 77654231
Holder: fathima
Balance: $11900.00

--------------------------------------------------------------------------------
Choose an operation:
1. Deposit
2. Withdraw
3. Check Balance
4. View Transaction History
5. Exit

Enter your choice (1-5): 4

Transaction History for Account 77654231:
------------------------------------------------------------
1. Initial Deposit | Amount: $  10000.00 | Balance: $  10000.00 | Time: 2025-11-08 12:37:59
2. Deposit         | Amount: $   2000.00 | Balance: $  12000.00 | Time: 2025-11-08 12:38:13
3. Withdrawal      | Amount: $    100.00 | Balance: $  11900.00 | Time: 2025-11-08 12:38:36
```

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

Evaluation Criteria:

Criteria	**Max Marks**
Class	1.0
Loops	1.0
Conditional Statements	0.5
Total	**2.5 Marks**