AIAssistantCoding

Assignment-6.3

Name: T.Harivarshith

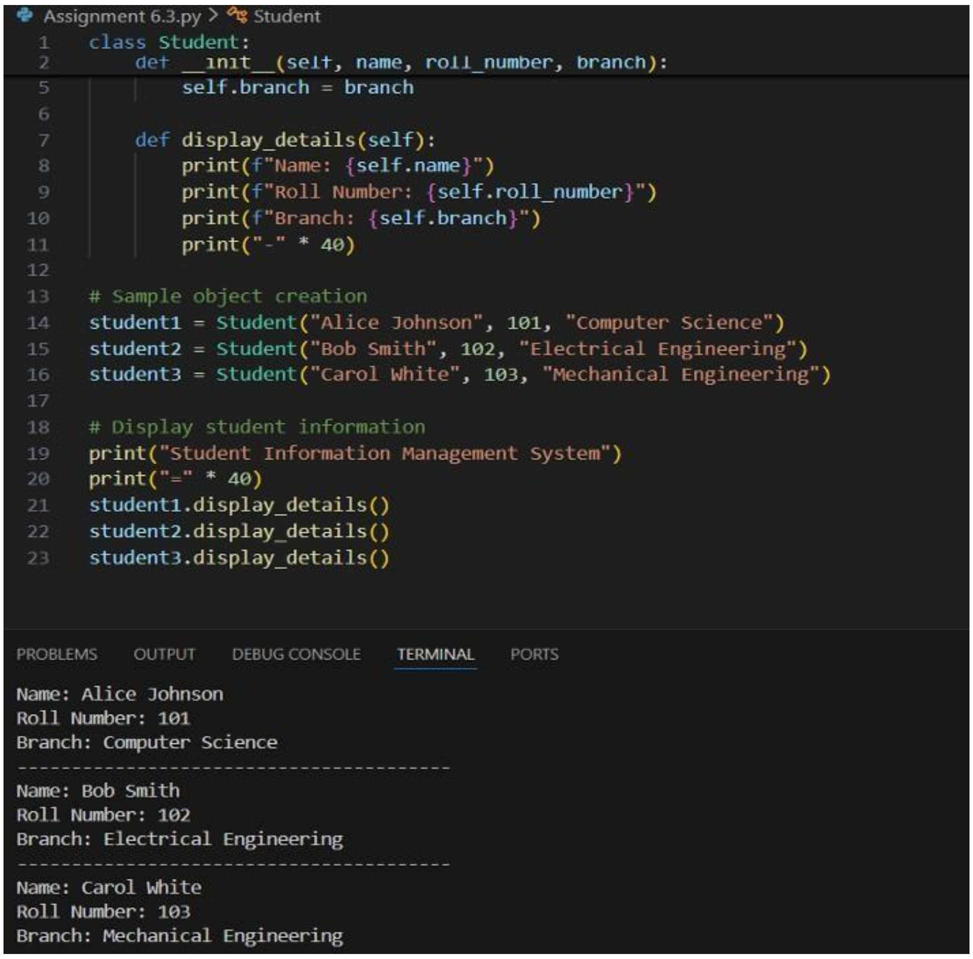
HT NO:2303A51351

Batch:20

TaskDescription 1:Classes (StudentClass)

Prompt: Create a Python program for a simple student information management module. Define a Student class with attributes name, roll_number, and branch. Implement a constructor (init) to initialize these attributes and a method display_details() to print the student’s information in a readable format.
Include sample object creation, execute the code, and show the output. After the code, provide a brief analysis explaining the correctness and clarity of the generated code.

Code & Output:



```python
class Student:
    def __init__(self, name, roll_number, branch):
        self.branch = branch

    def display_details(self):
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Branch: {self.branch}")
        print("-" * 40)

# Sample object creation
student1 = Student("Alice Johnson", 101, "Computer Science")
student2 = Student("Bob Smith", 102, "Electrical Engineering")
student3 = Student("Carol White", 103, "Mechanical Engineering")

# Display student information
print("Student Information Management System")
print("=" * 40)
student1.display_details()
student2.display_details()
student3.display_details()
```

```
Name: Alice Johnson
Roll Number: 101
Branch: Computer Science
----------------------------------------
Name: Bob Smith
Roll Number: 102
Branch: Electrical Engineering
----------------------------------------
Name: Carol White
Roll Number: 103
Branch: Mechanical Engineering
```

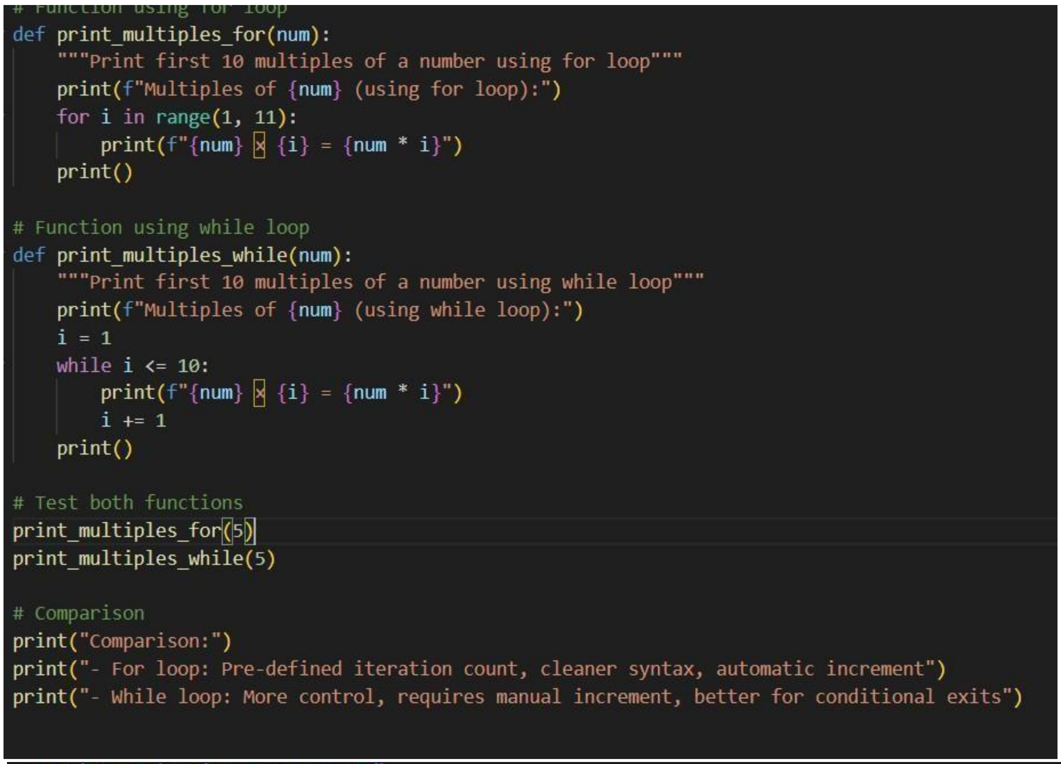
BriefAnalysis ofAI-Generated Code

- The Student class is correctly defined using object-oriented principles.
- The constructor (init) properly initializes the student attributes: name, roll_number, and branch.
- Thedisplay_details() method clearly formats andprints student information, improving readability.
- Sample object creation demonstrates correct usage of the class.
- The code is clean, well-structured, easy to understand, and follows Python best practices.
- Overall, the AI-generated code is correct, clear, and suitable for a basic student information management module

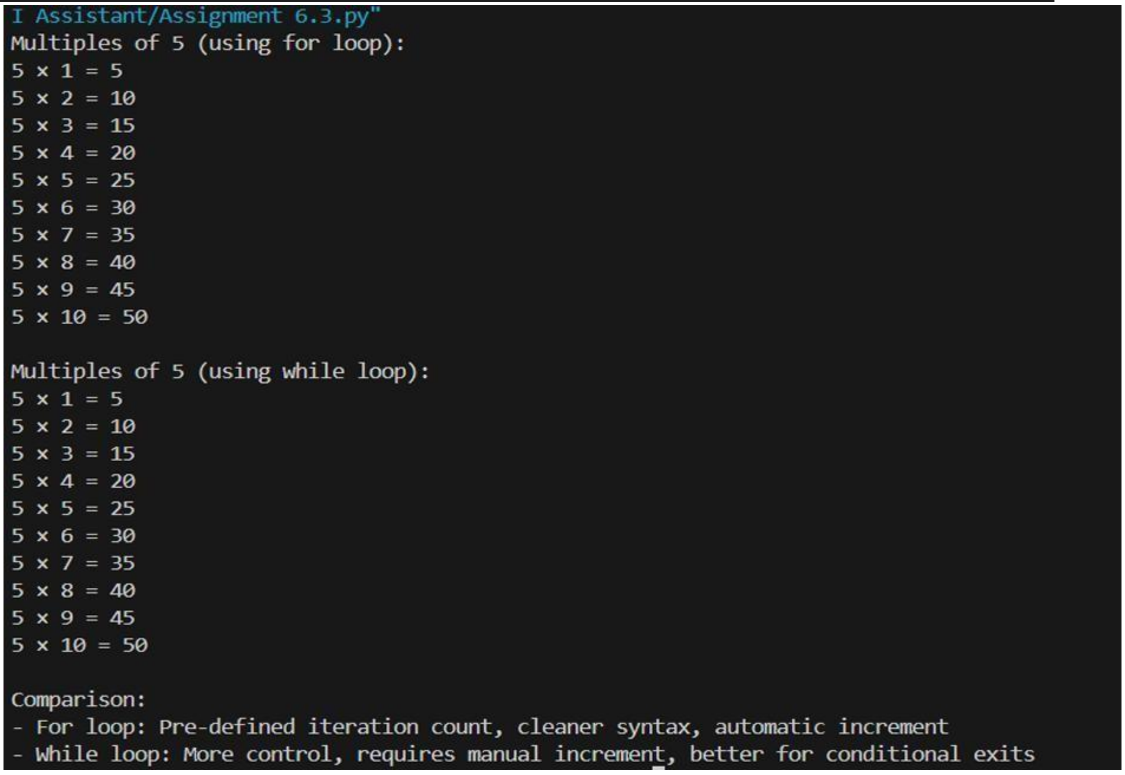
TaskDescription 2: Loops (Multiples of a Number)

Prompt: Write a Python utility function that takes an integer as input and prints the first 10 multiples of that number using a loop.
First, implement the solution using a for loop and display the output.
Analyze the loop logic used in the function for correctness and clarity.
Then, generate the same functionality using a different controlled looping structure, such as a while loop.
Finally, compare both looping approaches and briefly explain their differences.

Code& Output:



```
# Function using for loop
def print_multiples_for(num):
    """Print first 10 multiples of a number using for loop"""
    print(f"Multiples of {num} (using for loop):")
    for i in range(1, 11):
        print(f"{num} × {i} = {num * i}")
    print()

# Function using while loop
def print_multiples_while(num):
    """Print first 10 multiples of a number using while loop"""
    print(f"Multiples of {num} (using while loop):")
    i = 1
    while i <= 10:
        print(f"{num} × {i} = {num * i}")
        i += 1
    print()

# Test both functions
print_multiples_for(5)
print_multiples_while(5)

# Comparison
print("Comparison:")
print("- For loop: Pre-defined iteration count, cleaner syntax, automatic increment")
print("- While loop: More control, requires manual increment, better for conditional exits")
```



```
I Assistant/Assignment 6.3.py"
Multiples of 5 (using for loop):
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
5 × 5 = 25
5 × 6 = 30
5 × 7 = 35
5 × 8 = 40
5 × 9 = 45
5 × 10 = 50

Multiples of 5 (using while loop):
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
5 × 5 = 25
5 × 6 = 30
5 × 7 = 35
5 × 8 = 40
5 × 9 = 45
5 × 10 = 50

Comparison:
- For loop: Pre-defined iteration count, cleaner syntax, automatic increment
- While loop: More control, requires manual increment, better for conditional exits
```

AnalysisofLoopLogic For Loop Analysis

- Uses range(1, 11) to define a fixed number of iterations.
- Automatically handles initialization, condition checking, and increment.
- Best suited when the number of iterations is known in advance.
- Code is concise and easy to read.

While Loop Analysis

- Uses a manually controlled counter (i).
- Offersmore flexibilityfor complex or condition-based looping.
- Requirescareful increment to avoid infinite loops.
- Slightly more verbose but useful when loop conditions may change dynamically.

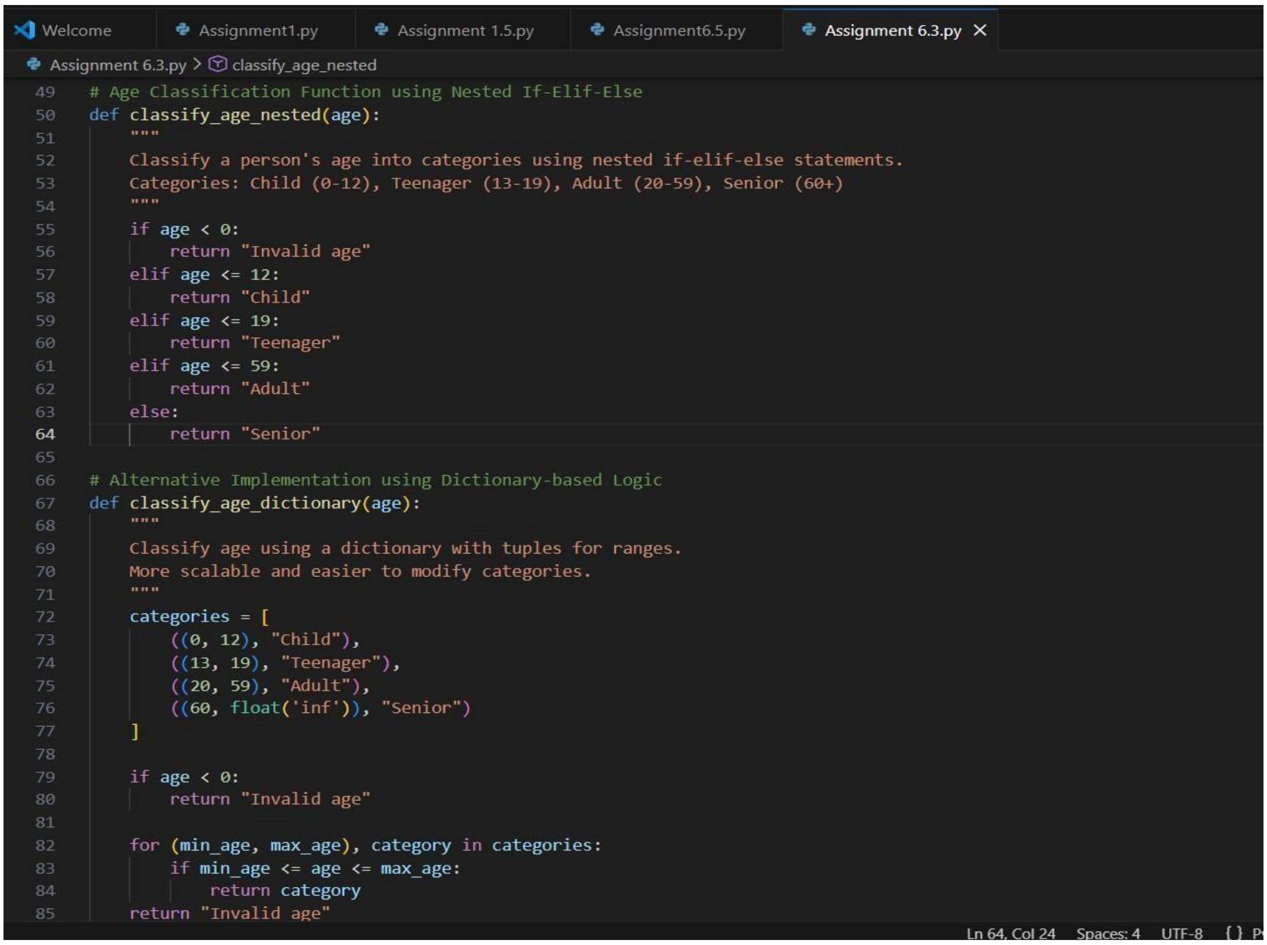
Comparison Summary

- Both approaches correctly generate the first 10 multiples of a number.
- The for loop is simpler and cleaner for fixed iterations.
- The while loop provides greater control and flexibility.
- Choosing between them depends on the problem requirements.

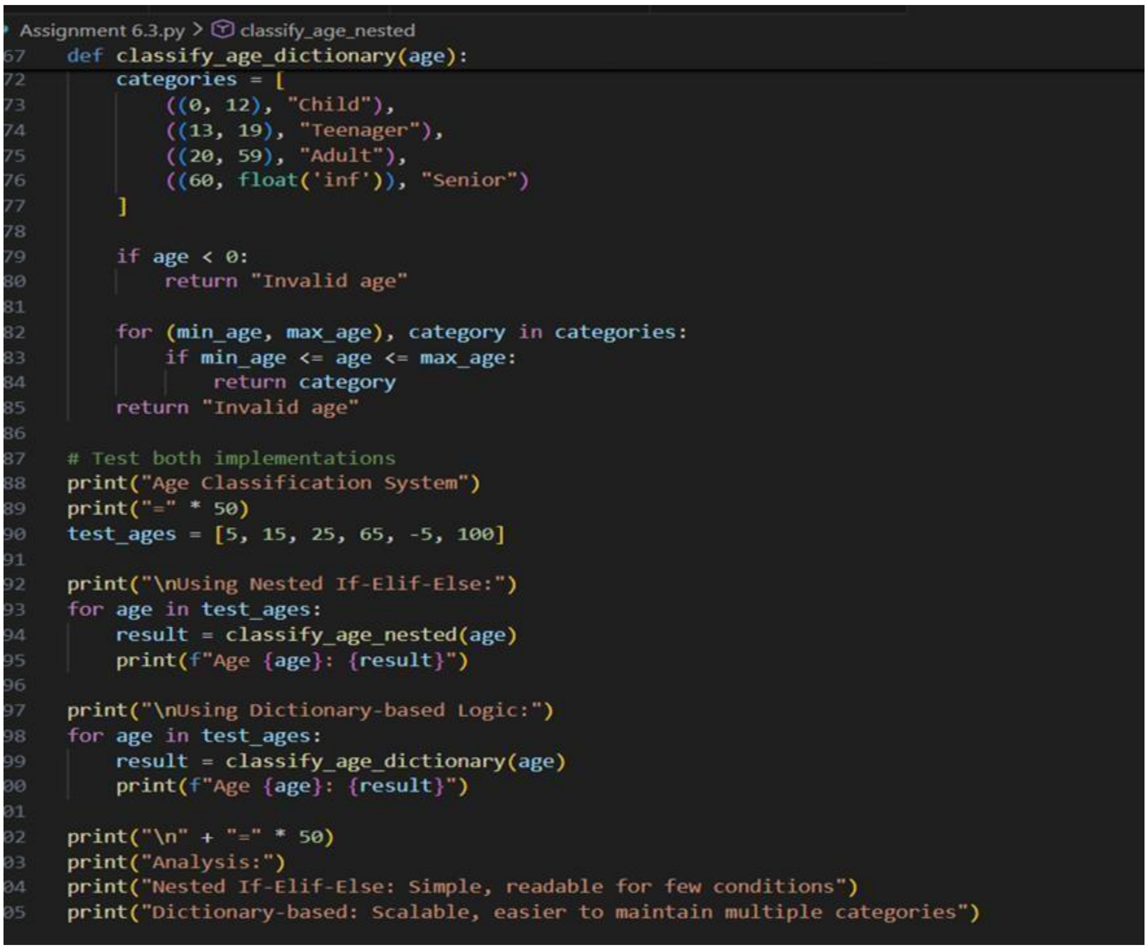
Task Description 3: Conditional Statements (Age Classification)

Prompt: Create a Python function that classifies a person's age into categories such as child, teenager, adult, and senior using nested if-elif-else conditional statements.
Analyze the conditional logic used and explain how each condition works.
Then, generate an alternative implementation of the same age classification using a different conditional approach, such as simplified conditions or a dictionary-based logic. Ensure the output is clear, correct, and easy to understand.

Code & Output:

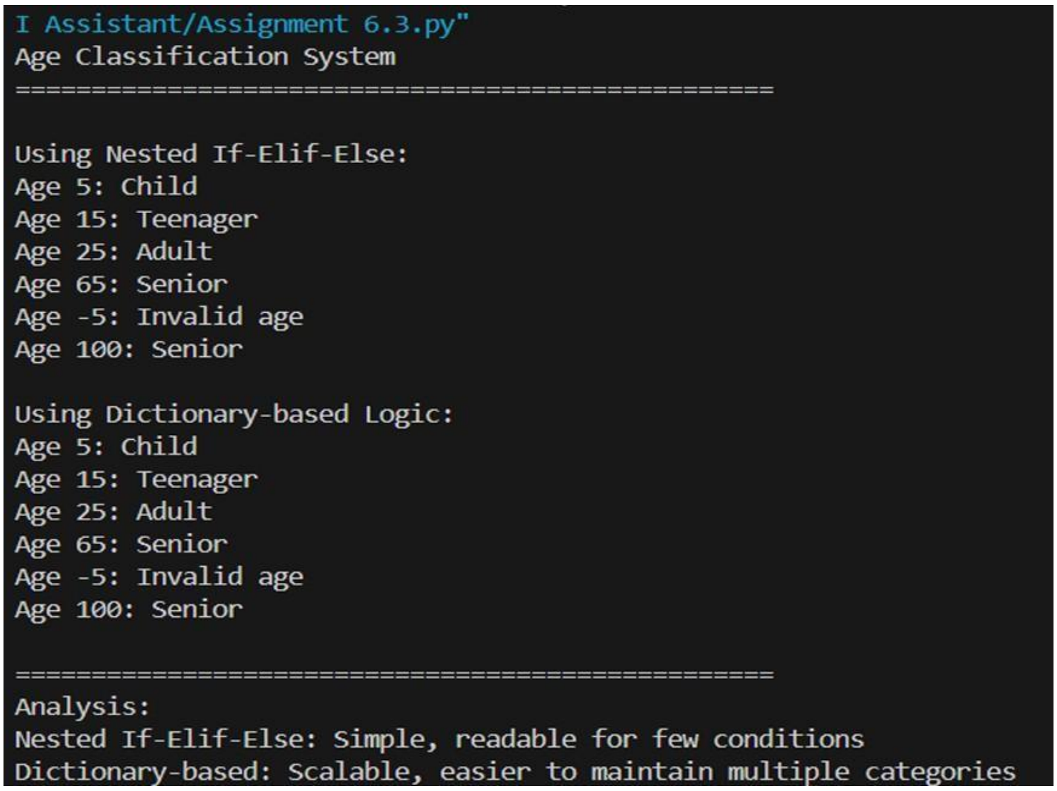


```python
# Age Classification Function using Nested If-Elif-Else
def classify_age_nested(age):
    """
    Classify a person's age into categories using nested if-elif-else statements.
    Categories: Child (0-12), Teenager (13-19), Adult (20-59), Senior (60+)
    """
    if age < 0:
        return "Invalid age"
    elif age <= 12:
        return "Child"
    elif age <= 19:
        return "Teenager"
    elif age <= 59:
        return "Adult"
    else:
        return "Senior"

# Alternative Implementation using Dictionary-based Logic
def classify_age_dictionary(age):
    """
    Classify age using a dictionary with tuples for ranges.
    More scalable and easier to modify categories.
    """
    categories = [
        ((0, 12), "Child"),
        ((13, 19), "Teenager"),
        ((20, 59), "Adult"),
        ((60, float('inf')), "Senior")
    ]

    if age < 0:
        return "Invalid age"

    for (min_age, max_age), category in categories:
        if min_age <= age <= max_age:
            return category
    return "Invalid age"
```



Assignment 6.3.py > classify_age_nested

```python
def classify_age_dictionary(age):
    categories = [
        ((0, 12), "Child"),
        ((13, 19), "Teenager"),
        ((20, 59), "Adult"),
        ((60, float('inf')), "Senior")
    ]

    if age < 0:
        return "Invalid age"

    for (min_age, max_age), category in categories:
        if min_age <= age <= max_age:
            return category
    return "Invalid age"

# Test both implementations
print("Age Classification System")
print("=" * 50)
test_ages = [5, 15, 25, 65, -5, 100]

print("\nUsing Nested If-Elif-Else:")
for age in test_ages:
    result = classify_age_nested(age)
    print(f"Age {age}: {result}")

print("\nUsing Dictionary-based Logic:")
for age in test_ages:
    result = classify_age_dictionary(age)
    print(f"Age {age}: {result}")

print("\n" + "=" * 50)
print("Analysis:")
print("Nested If-Elif-Else: Simple, readable for few conditions")
print("Dictionary-based: Scalable, easier to maintain multiple categories")
```



```
I Assistant/Assignment 6.3.py"
Age Classification System
==================================================

Using Nested If-Elif-Else:
Age 5: Child
Age 15: Teenager
Age 25: Adult
Age 65: Senior
Age -5: Invalid age
Age 100: Senior

Using Dictionary-based Logic:
Age 5: Child
Age 15: Teenager
Age 25: Adult
Age 65: Senior
Age -5: Invalid age
Age 100: Senior

==================================================
Analysis:
Nested If-Elif-Else: Simple, readable for few conditions
Dictionary-based: Scalable, easier to maintain multiple categories
```

Python Functions for Age Classification

Nested if-elif-else Implementation

- Classifies age into Child, Teenager, Adult, and Senior
- Handles invalid (negative) ages

Alternative Dictionary-Based Implementation

- Uses age ranges stored as tuples
- More scalable and easier to modify or extend

ExplanationofConditionalLogic

Nested if-elif-else

- Checks conditions sequentially from lowest age to highest.
- Each elif narrows the age range: ○ age <= 12 → Child ○ age <= 19 → Teenager ○ age <= 59 → Adult
    - else → Senior
- Easy to read and ideal for a small number of conditions.

Dictionary-Based Logic

- Stores age ranges as (min_age, max_age) pairs.
- Iteratesthrough ranges and matches the age.
- More flexible and maintainable when adding or modifying categories.

Comparison Summary

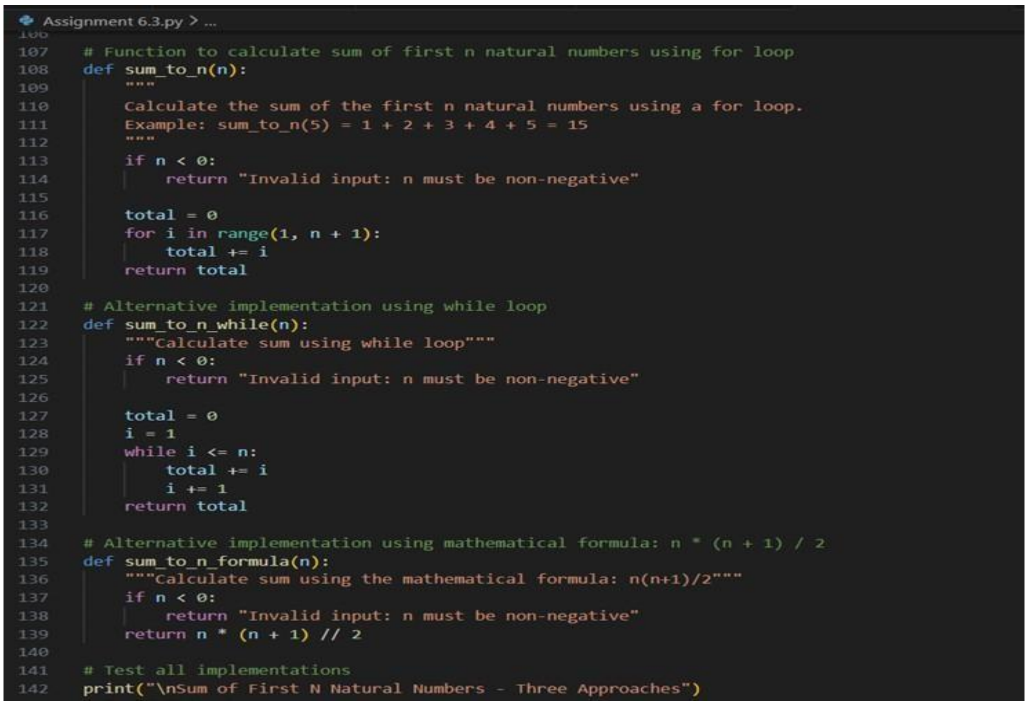
Approach	Advantage
Nested if-elif-else	Simple and straightforward
Dictionary-based	Scalable and easier to update

Task Description 4: For and While Loops (Sum of First n Numbers)

Prompt: Generate a Python function named sum_to_n(n) that calculates the sum of the first n natural numbers using a for loop.
Analyzethe generated code for correctness and clarity.
Then, provide an alternative implementation of the same functionality using either a while loop or a mathematical formula.

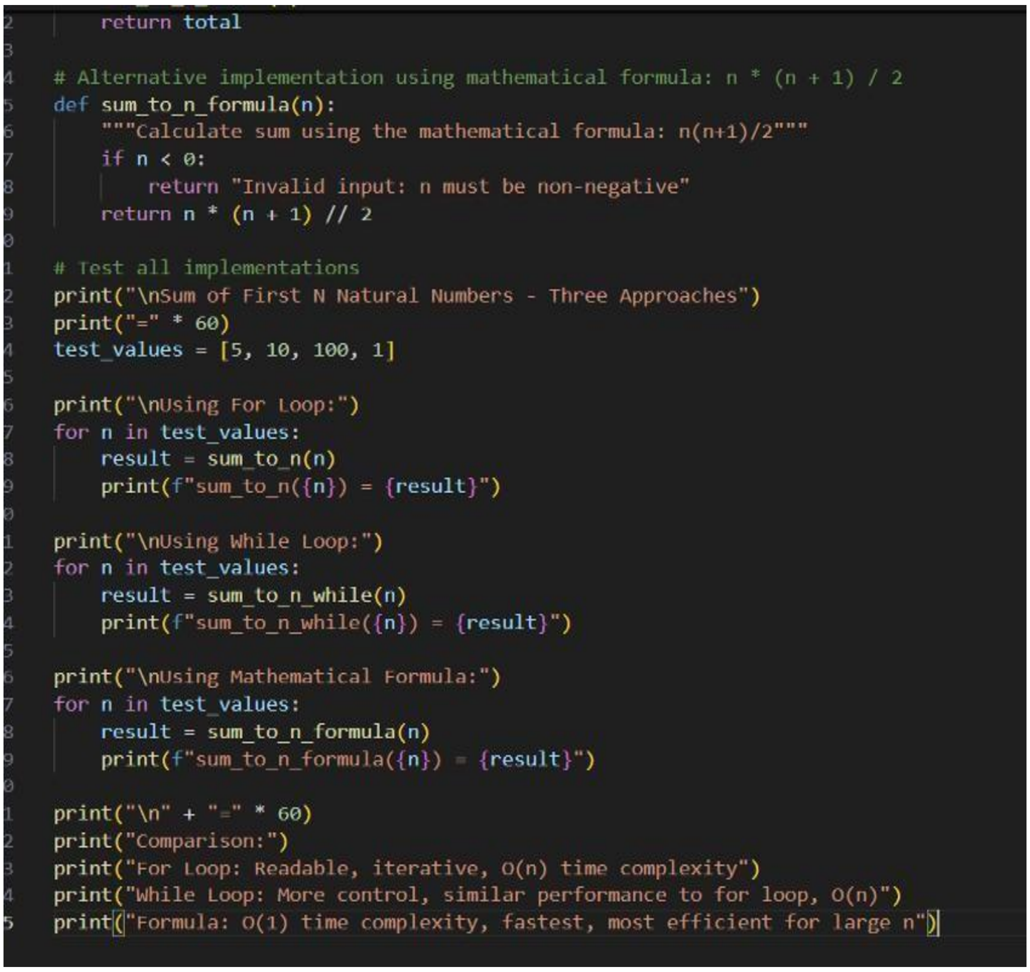
Include sample inputs, outputs, and a brief comparison explaining thedifferences between the approaches.

Code&Output:

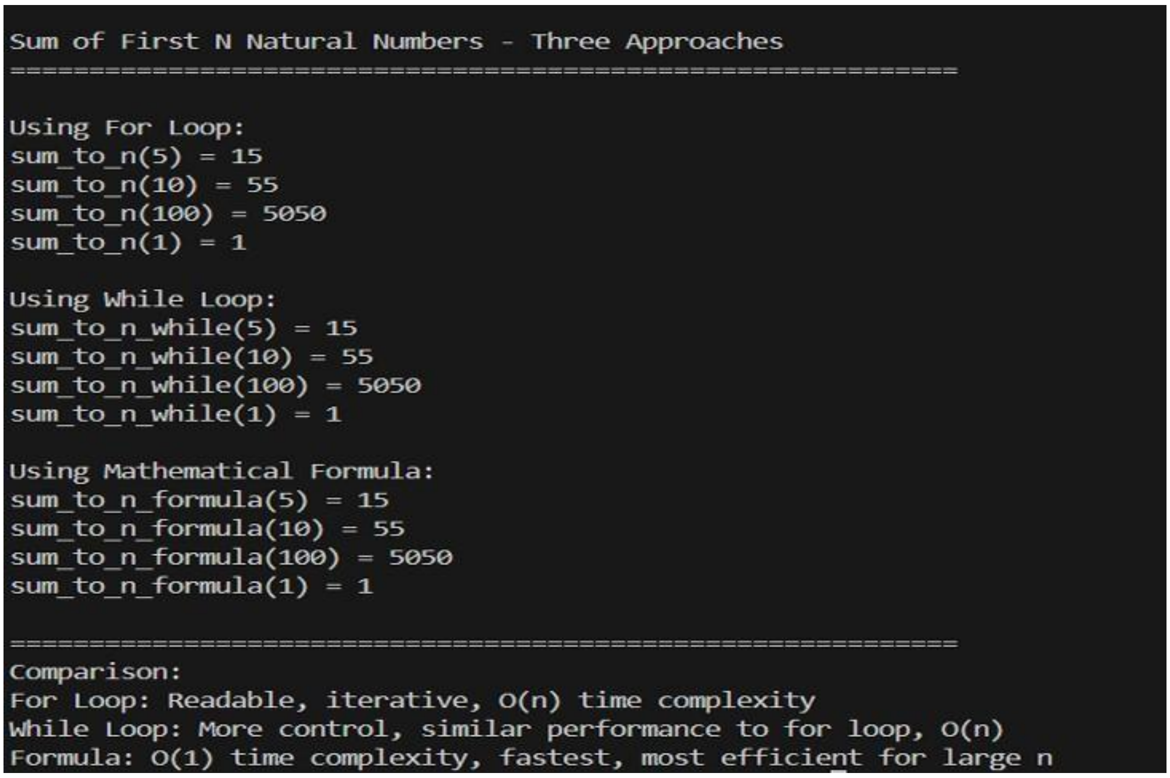


Assignment 6.3.py > ...

```python
# Function to calculate sum of first n natural numbers using for loop
def sum_to_n(n):
    """
    Calculate the sum of the first n natural numbers using a for loop.
    Example: sum_to_n(5) = 1 + 2 + 3 + 4 + 5 = 15
    """
    if n < 0:
        return "Invalid input: n must be non-negative"

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

# Alternative implementation using while loop
def sum_to_n_while(n):
    """Calculate sum using while loop"""
    if n < 0:
        return "Invalid input: n must be non-negative"

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

# Alternative implementation using mathematical formula: n * (n + 1) / 2
def sum_to_n_formula(n):
    """Calculate sum using the mathematical formula: n(n+1)/2"""
    if n < 0:
        return "Invalid input: n must be non-negative"
    return n * (n + 1) // 2

# Test all implementations
print("\nSum of First N Natural Numbers - Three Approaches")
```



```python
    return total

# Alternative implementation using mathematical formula: n * (n + 1) / 2
def sum_to_n_formula(n):
    """Calculate sum using the mathematical formula: n(n+1)/2"""
    if n < 0:
        return "Invalid input: n must be non-negative"
    return n * (n + 1) // 2

# Test all implementations
print("\nSum of First N Natural Numbers - Three Approaches")
print("=" * 60)
test_values = [5, 10, 100, 1]

print("\nUsing For Loop:")
for n in test_values:
    result = sum_to_n(n)
    print(f"sum_to_n({n}) = {result}")

print("\nUsing While Loop:")
for n in test_values:
    result = sum_to_n_while(n)
    print(f"sum_to_n_while({n}) = {result}")

print("\nUsing Mathematical Formula:")
for n in test_values:
    result = sum_to_n_formula(n)
    print(f"sum_to_n_formula({n}) = {result}")

print("\n" + "=" * 60)
print("Comparison:")
print("For Loop: Readable, iterative, O(n) time complexity")
print("While Loop: More control, similar performance to for loop, O(n)")
print("Formula: O(1) time complexity, fastest, most efficient for large n")
```



```
Sum of First N Natural Numbers - Three Approaches
============================================================

Using For Loop:
sum_to_n(5) = 15
sum_to_n(10) = 55
sum_to_n(100) = 5050
sum_to_n(1) = 1

Using While Loop:
sum_to_n_while(5) = 15
sum_to_n_while(10) = 55
sum_to_n_while(100) = 5050
sum_to_n_while(1) = 1

Using Mathematical Formula:
sum_to_n_formula(5) = 15
sum_to_n_formula(10) = 55
sum_to_n_formula(100) = 5050
sum_to_n_formula(1) = 1

============================================================
Comparison:
For Loop: Readable, iterative, O(n) time complexity
While Loop: More control, similar performance to for loop, O(n)
Formula: O(1) time complexity, fastest, most efficient for large n
```

ExplanationandComparisonof Approaches

For Loop Approach

- Iterates from 1 to *n* and accumulates the sum.
- Easy to read and understand.
- Time complexity: O(n).
- Suitable for learning and small input sizes.

While Loop Approach

- Uses a loop counter with manual control.
- Offers flexibility in complex conditions.
- Same time complexity as the for loop: O(n).
- Requirescareful handling to avoid infinite loops.

Mathematical FormulaApproach

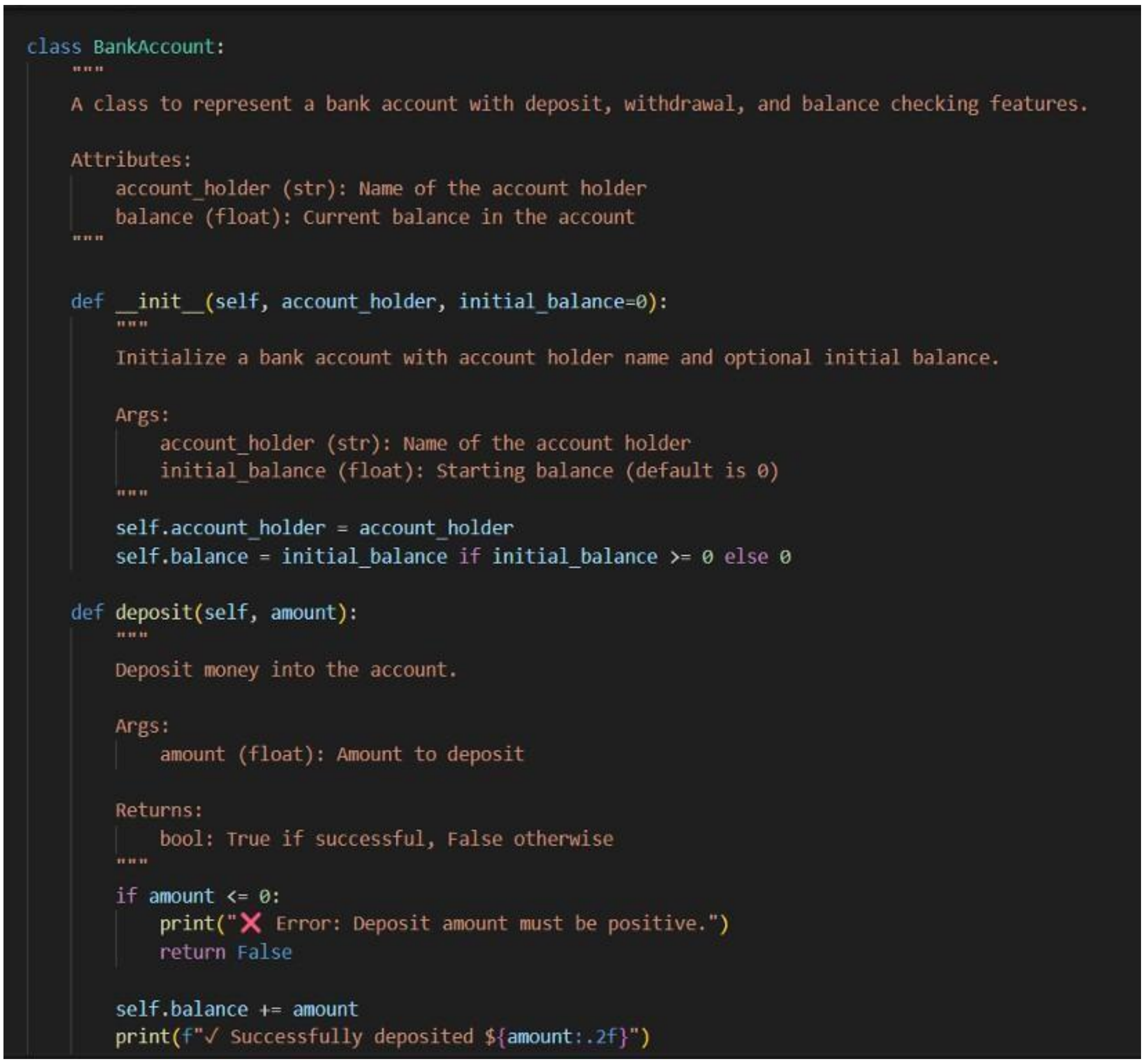
Uses the formula n(n + 1) / 2.

- No iteration required.
- Time complexity: O(1).
- Most efficient and best choice for large values of *n*.

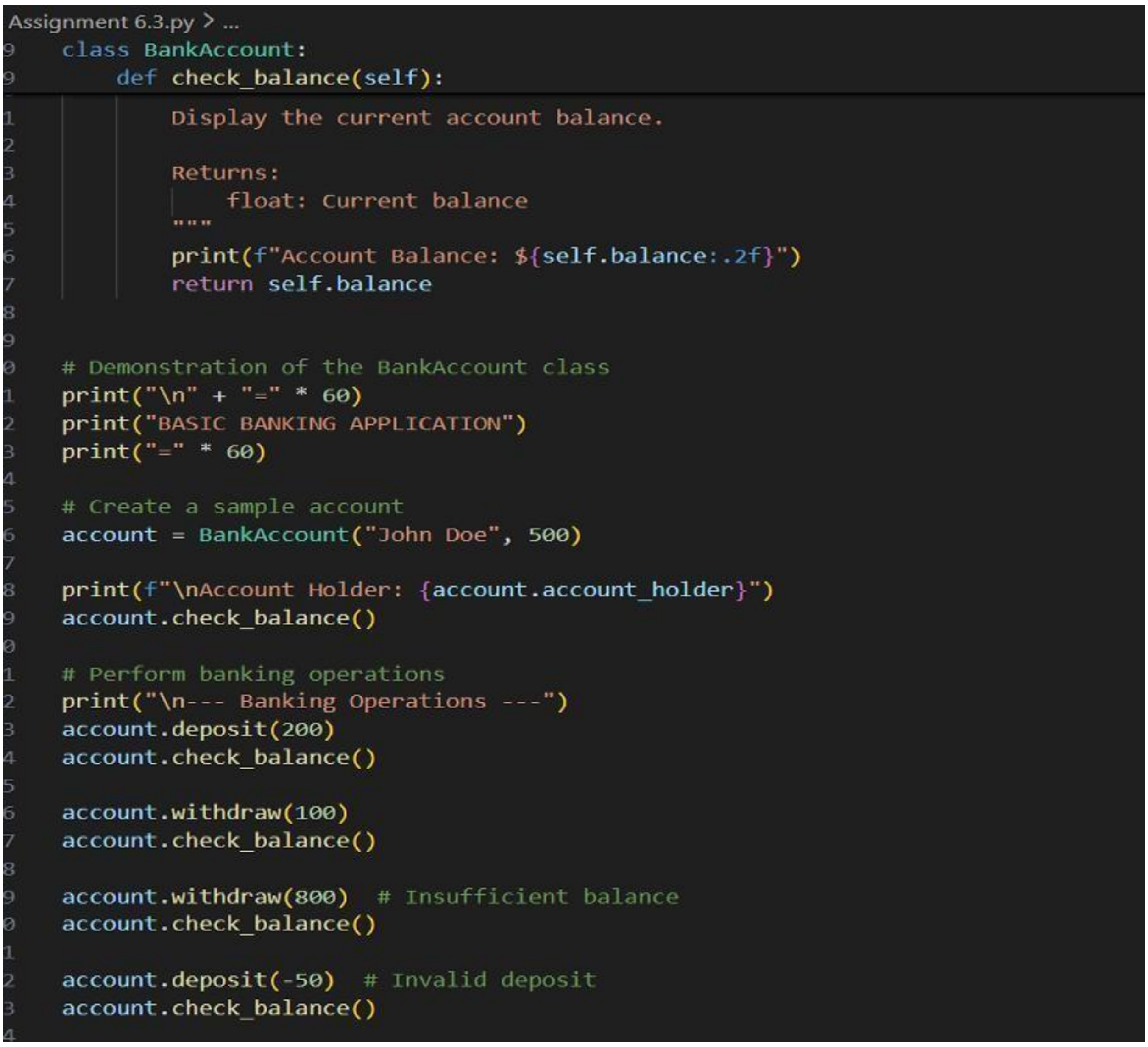
Task Description 5: Classes (Bank Account Class)

Prompt: Create a Python program for a basic banking application.
Define a BankAccount class with attributes such as account_holder and balance.
Implement methods deposit(amount), withdraw(amount), and check_balance() with proper validation (e.g., no negative deposits, insufficient balance checks).
Demonstrate the class by creating a sample account and performing deposit and withdrawal operations while displaying the updated balance.
Add meaningful comments to the code and provide a clear explanation of how the class and its methods work.

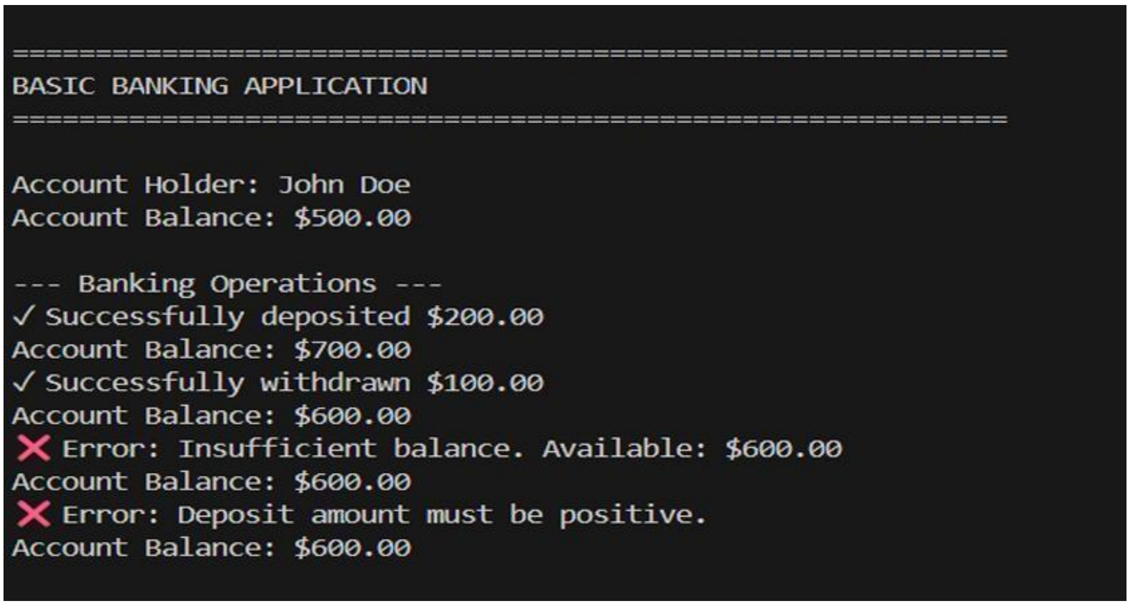
Code & Output



```python
class BankAccount:
    """
    A class to represent a bank account with deposit, withdrawal, and balance checking features.

    Attributes:
        account_holder (str): Name of the account holder
        balance (float): Current balance in the account
    """

    def __init__(self, account_holder, initial_balance=0):
        """
        Initialize a bank account with account holder name and optional initial balance.

        Args:
            account_holder (str): Name of the account holder
            initial_balance (float): Starting balance (default is 0)
        """
        self.account_holder = account_holder
        self.balance = initial_balance if initial_balance >= 0 else 0

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

        Args:
            amount (float): Amount to deposit

        Returns:
            bool: True if successful, False otherwise
        """
        if amount <= 0:
            print("❌ Error: Deposit amount must be positive.")
            return False

        self.balance += amount
        print(f"✓ Successfully deposited ${amount:.2f}")
```



```
Assignment 6.3.py > ...
class BankAccount:
    def check_balance(self):
        Display the current account balance.

        Returns:
            float: Current balance
        """
        print(f"Account Balance: ${self.balance:.2f}")
        return self.balance


# Demonstration of the BankAccount class
print("\n" + "=" * 60)
print("BASIC BANKING APPLICATION")
print("=" * 60)

# Create a sample account
account = BankAccount("John Doe", 500)

print(f"\nAccount Holder: {account.account_holder}")
account.check_balance()

# Perform banking operations
print("\n--- Banking Operations ---")
account.deposit(200)
account.check_balance()

account.withdraw(100)
account.check_balance()

account.withdraw(800)  # Insufficient balance
account.check_balance()

account.deposit(-50)  # Invalid deposit
account.check_balance()
```



```
============================================================
BASIC BANKING APPLICATION
============================================================

Account Holder: John Doe
Account Balance: $500.00

--- Banking Operations ---
✓ Successfully deposited $200.00
Account Balance: $700.00
✓ Successfully withdrawn $100.00
Account Balance: $600.00
❌ Error: Insufficient balance. Available: $600.00
Account Balance: $600.00
❌ Error: Deposit amount must be positive.
Account Balance: $600.00
```

Explanation of the Code Class Structure

- The BankAccount class represents a simple banking system.
- It stores the account holder’s name and current balance as attributes.

Constructor (__init__)

- Initializes the account with a holder name and optional initial balance.
- Preventsnegative starting balances by defaulting to zero**. deposit() Method**
- Allows adding money to the account.
- Validates that the deposit amount is positive.
- Updates and displays the new balance.

withdraw() Method

- Ensureswithdrawal amount is positive.
- Prevents overdrafts by checking available balance.
- Deducts the amount if valid and updates the balance. **check_balance() Method**
- Displays the current balance.
- Returns the balance for further use if needed.

Overall Analysis

The class structure is clean and well-organized
Input validation ensures safe banking operations
Methodsclearly reflect real-worldbanking behavior
Commentsand docstrings improve readability and understanding