

ASSIGNMENT-06

NAME:B.Manishwar

HALLTICKET NO:2303A51276

BATCH:05

Q) Task Description #1: Classes (Student Class)

Scenario

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.

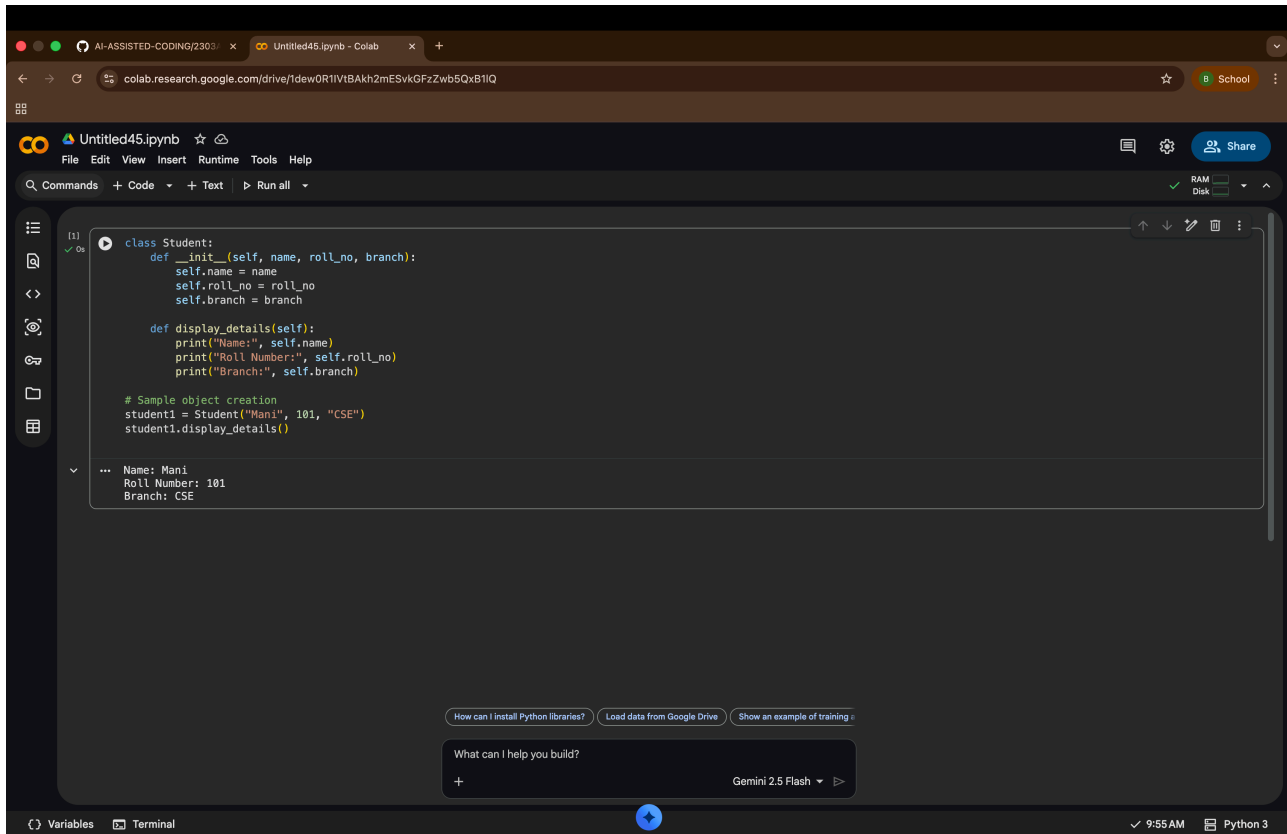
Expected Output #1

- A Python class with a constructor (`__init__`) and a `display_details()` method.
- Sample object creation and output displayed on the console.
- Brief analysis of AI-generated code.

PROMPT:

Generate a Python Student class with attributes name, roll number, and branch. Add a method `display_details()` to print student information. Also create a sample student object and display the output.

CODE AND OUTPUT:



The screenshot shows a Google Colab notebook titled 'Untitled45.ipynb'. The code cell contains the following Python code:

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

    def display_details(self):
        print("Name:", self.name)
        print("Roll Number:", self.roll_no)
        print("Branch:", self.branch)

# Sample object creation
student1 = Student("Mani", 101, "CSE")
student1.display_details()
```

The output of the code cell shows the following text:

```
Name: Mani
Roll Number: 101
Branch: CSE
```

The interface includes a left sidebar with icons for file explorer, search, and other tools. At the bottom, there is a prompt area with the text 'What can I help you build?' and a 'Gemini 2.5 Flash' model selector.

CODE EXPLANATION:

This code defines a `Student` class with a constructor to initialize name, roll number, and branch. The `display_details()` method prints the student information. A student object is created and the details are displayed on the console. The code is correct, clear, and easy to understand.

Q) Task Description #2: Loops (Multiples of a Number)

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`).

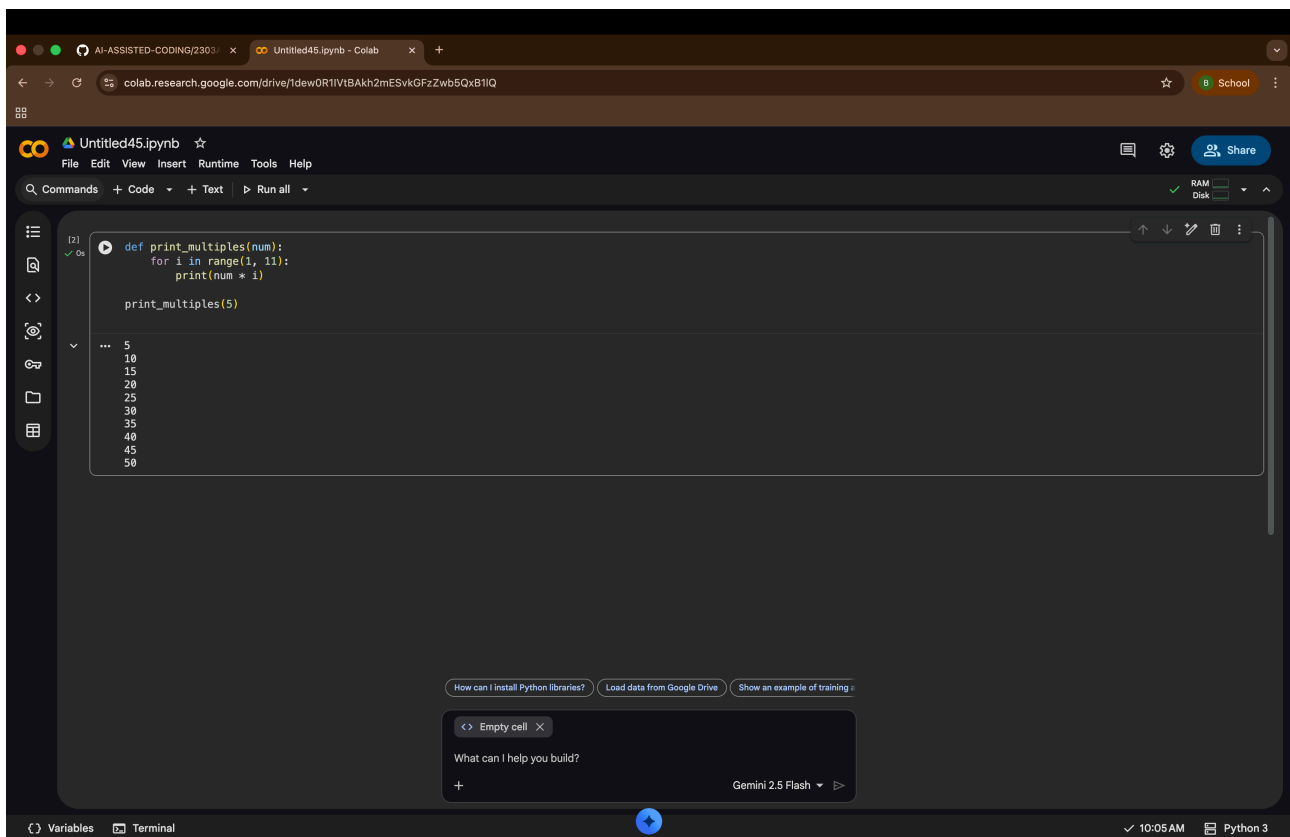
Expected Output #2

- Correct loop-based Python implementation.
- Output showing the first 10 multiples of a number.
- Comparison and analysis of different looping approaches.

PROMPT:

Generate a Python function that prints the first 10 multiples of a given number using a `for` loop.

CODE AND OUTPUT:



The screenshot shows a Google Colab notebook titled 'Untitled45.ipynb'. The code cell contains a function `print_multiples(num)` that uses a `for` loop to print multiples of `num` from 1 to 11. The function is called with `print_multiples(5)`. The output shows the numbers 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for file operations, and a bottom status bar showing 'Variables', 'Terminal', '10:05 AM', and 'Python 3'.

```
[2] def print_multiples(num):  
    for i in range(1, 11):  
        print(num * i)  
  
    print_multiples(5)  
  
... 5  
    10  
    15  
    20  
    25  
    30  
    35  
    40  
    45  
    50
```

CODE EXPLANATION:

The `for` loop is simpler and more readable for a fixed number of iterations. The `while` loop gives more control but needs manual counter management. Both implementations are correct and produce the same output.

Q)Task Description #3: Conditional Statements (Age Classification)

Scenario

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).

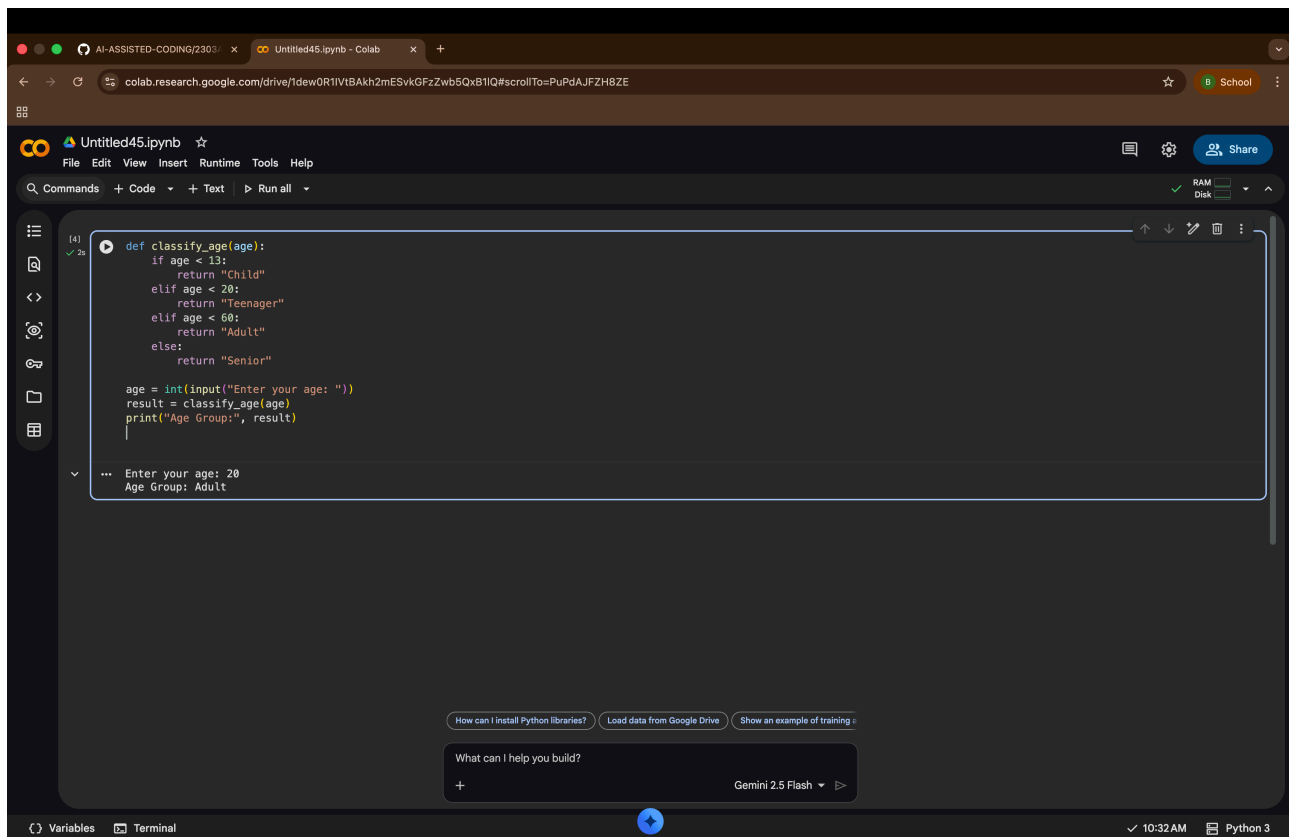
Expected Output #3

- A Python function that classifies age into appropriate groups.
- Clear and correct conditional logic.
- Explanation of how the conditions work.

PROMPT:

Generate a Python function using nested if-elif-else statements to classify age into child, teenager, adult, and senior.

CODE AND OUTPUT:



The screenshot shows a Google Colab notebook titled 'Untitled45.ipynb'. The code cell contains a Python function `classify_age` that uses nested if-elif-else statements to categorize an age into 'Child', 'Teenager', 'Adult', or 'Senior'. Below the code, the output shows the user entering '20' and the program outputting 'Age Group: Adult'. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for file operations, and a bottom status bar showing 'Variables', 'Terminal', and 'Python 3'.

```
[4] def classify_age(age):  
    if age < 13:  
        return "Child"  
    elif age < 20:  
        return "Teenager"  
    elif age < 60:  
        return "Adult"  
    else:  
        return "Senior"  
  
    age = int(input("Enter your age: "))  
    result = classify_age(age)  
    print("Age Group:", result)
```

... Enter your age: 20
Age Group: Adult

CODE AND EXPLANATION:

The program takes age as input at runtime using `input()`. It then uses if-elif-else conditions to classify the age into groups. The logic is clear and correctly categorizes the user based on entered age.

Q)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.

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.

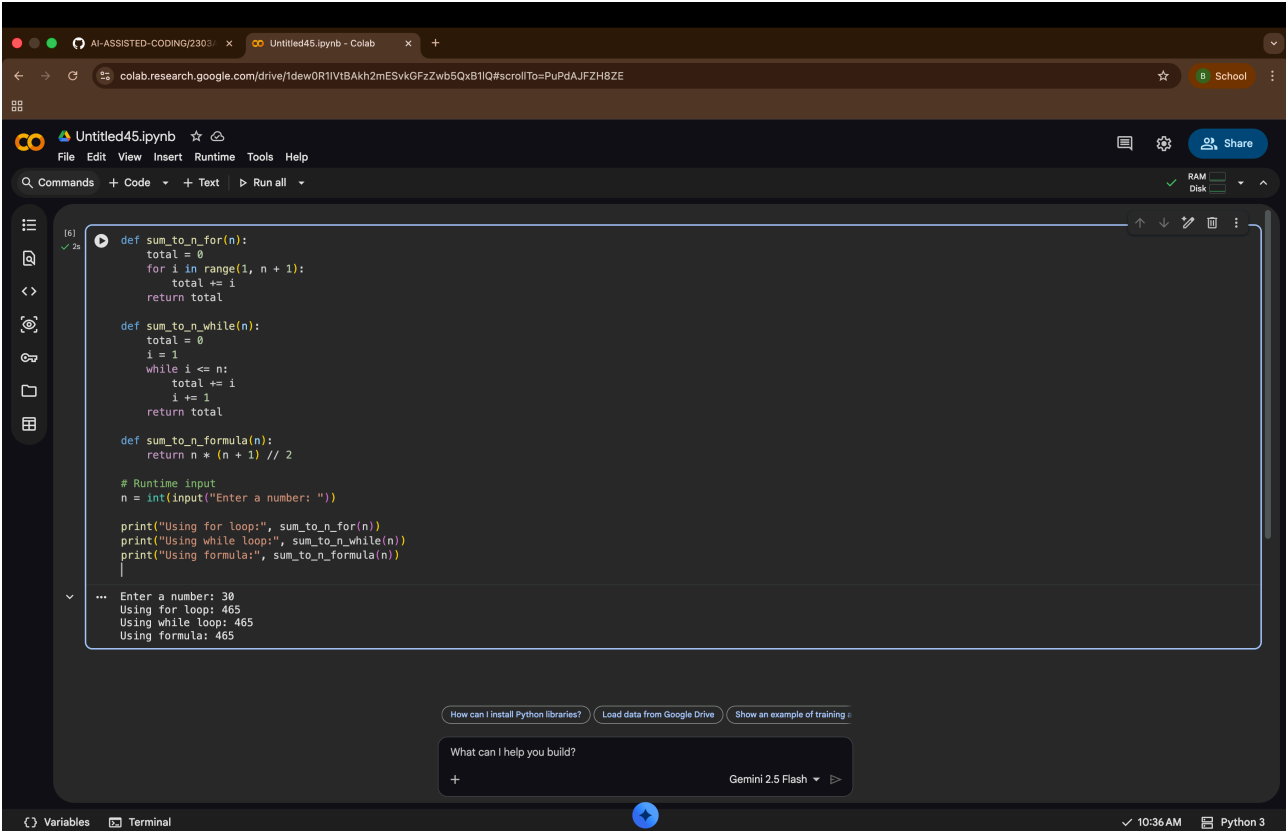
Expected Output #4

- Python function to compute the sum of first n numbers.
- Correct output for sample inputs.
- Explanation and comparison of different approaches.

PROMPT:

Generate a Python function that calculates the sum of the first n natural numbers using a for loop and takes input at runtime.

CODE AND OUTPUT:



```
(6) def sum_to_n_for(n):  
    total = 0  
    for i in range(1, n + 1):  
        total += i  
    return total  
  
def sum_to_n_while(n):  
    total = 0  
    i = 1  
    while i <= n:  
        total += i  
        i += 1  
    return total  
  
def sum_to_n_formula(n):  
    return n * (n + 1) // 2  
  
# Runtime input  
n = int(input("Enter a number: "))  
  
print("Using for loop:", sum_to_n_for(n))  
print("Using while loop:", sum_to_n_while(n))  
print("Using formula:", sum_to_n_formula(n))  
|  
... Enter a number: 30  
Using for loop: 465  
Using while loop: 465  
Using formula: 465
```

CODE EXPLANATION:

The for loop and while loop both add numbers one by one, which is easy to understand but slower for large n. The formula method is fastest and most efficient because it calculates the sum in one step. All methods are correct and give the same result.

Q)Task Description #5: Classes (Bank Account Class)

Scenario

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.

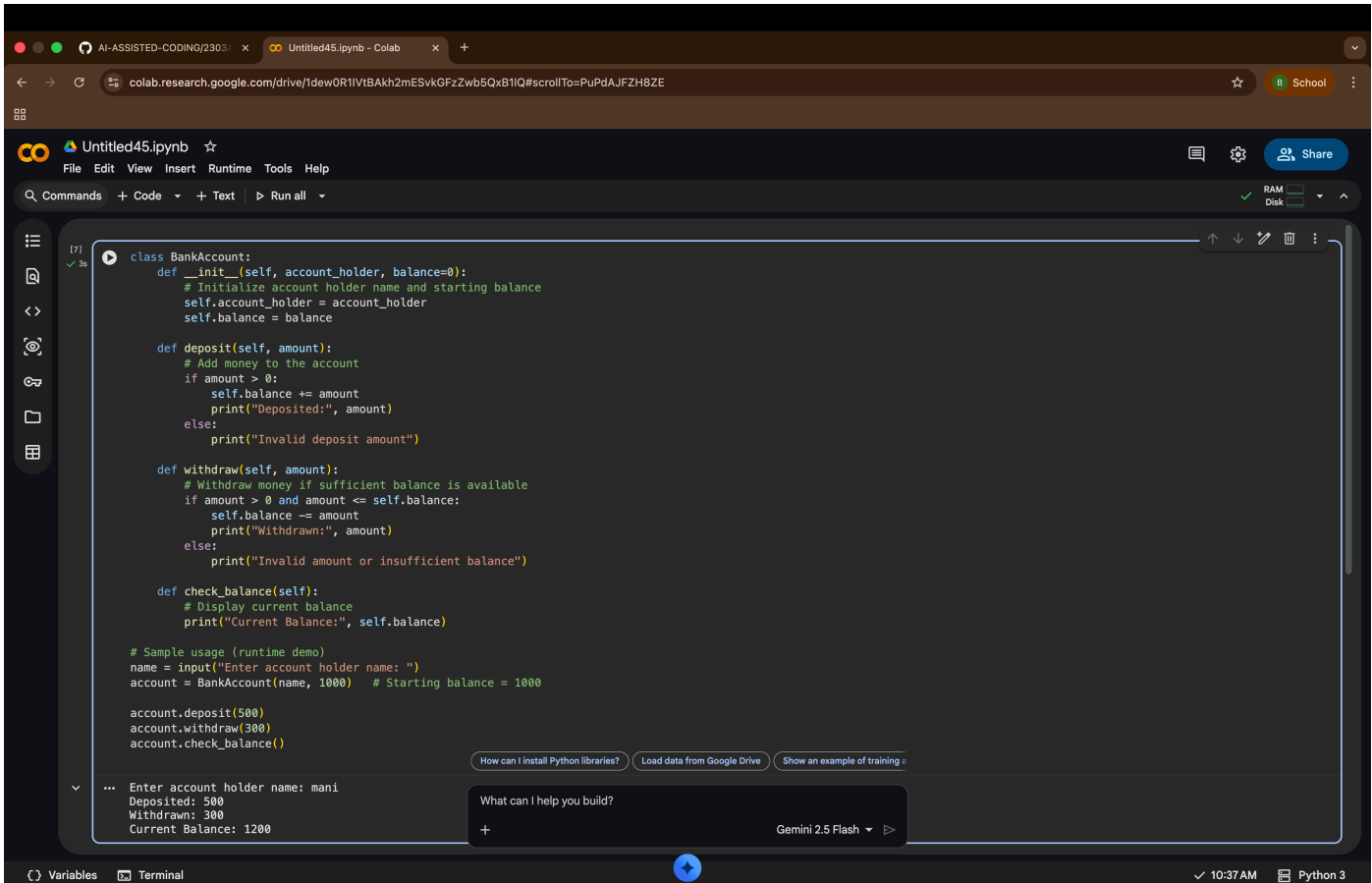
Expected Output #5

- Complete Python Bank Account class.
- Demonstration of deposit and withdrawal operations with updated balance.
- Well-commented code with a clear explanation.

PROMPT:

Generate a Python BankAccount class with methods deposit(), withdraw(), and check_balance(). Include sample usage and add meaningful comments.

CODE AND OUTPUT:



```
[7] class BankAccount:
    def __init__(self, account_holder, balance=0):
        # Initialize account holder name and starting balance
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        # Add money to the account
        if amount > 0:
            self.balance += amount
            print("Deposited:", amount)
        else:
            print("Invalid deposit amount")

    def withdraw(self, amount):
        # Withdraw money if sufficient balance is available
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            print("Withdrawn:", amount)
        else:
            print("Invalid amount or insufficient balance")

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

# Sample usage (runtime demo)
name = input("Enter account holder name: ")
account = BankAccount(name, 1000) # Starting balance = 1000

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

... Enter account holder name: mani
Deposited: 500
Withdrawn: 300
Current Balance: 1200

What can I help you build?

Gemini 2.5 Flash

CODE EXPLANATION:

This BankAccount class stores the account holder name and balance. The deposit() method adds money, withdraw() removes money only if sufficient balance exists, and check_balance() displays the current balance. The logic is correct, secure, and easy to understand. Comments improve clarity and make the code suitable for learning and maintenance.

