

## **ASSIGNMENT-06**

**NAME:**M.SHIVA SHANKAR

**HALLTICKET NO:**2303A51294

**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.

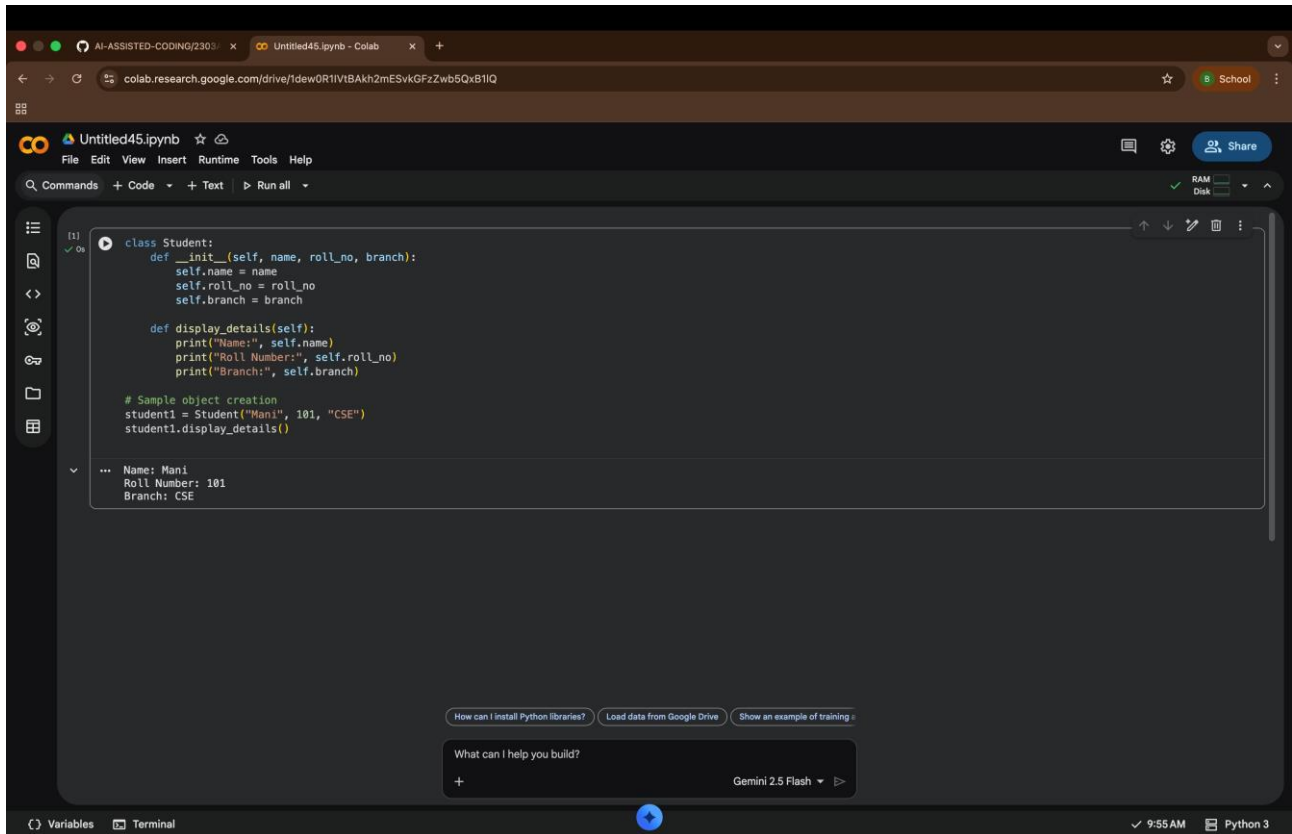
#### **Expected Output #1**

- 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 defines a `Student` class with an `__init__` method to initialize `name`, `roll_no`, and `branch`, and a `display_details` method to print these attributes. A sample object `student1` is created with the values 'Mani', 101, and 'CSE'. The output of the code execution is displayed in a collapsed cell, showing the printed details: 'Name: Mani', 'Roll Number: 101', and 'Branch: CSE'.

```
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()
```

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

## 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.

### PROMT:

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 the following Python code:

```
def print_multiples(num):  
    for i in range(1, 11):  
        print(num * i)  
  
print_multiples(5)
```

The output of the code is displayed below the cell, showing the multiples of 5 from 5 to 50:

```
5  
10  
15  
20  
25  
30  
35  
40  
45  
50
```

The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for commands, code, text, and running all cells, and a sidebar with icons for file explorer, search, and other tools. At the bottom, there is a terminal and a status bar showing 'Python 3'.

## 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:

```
[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.

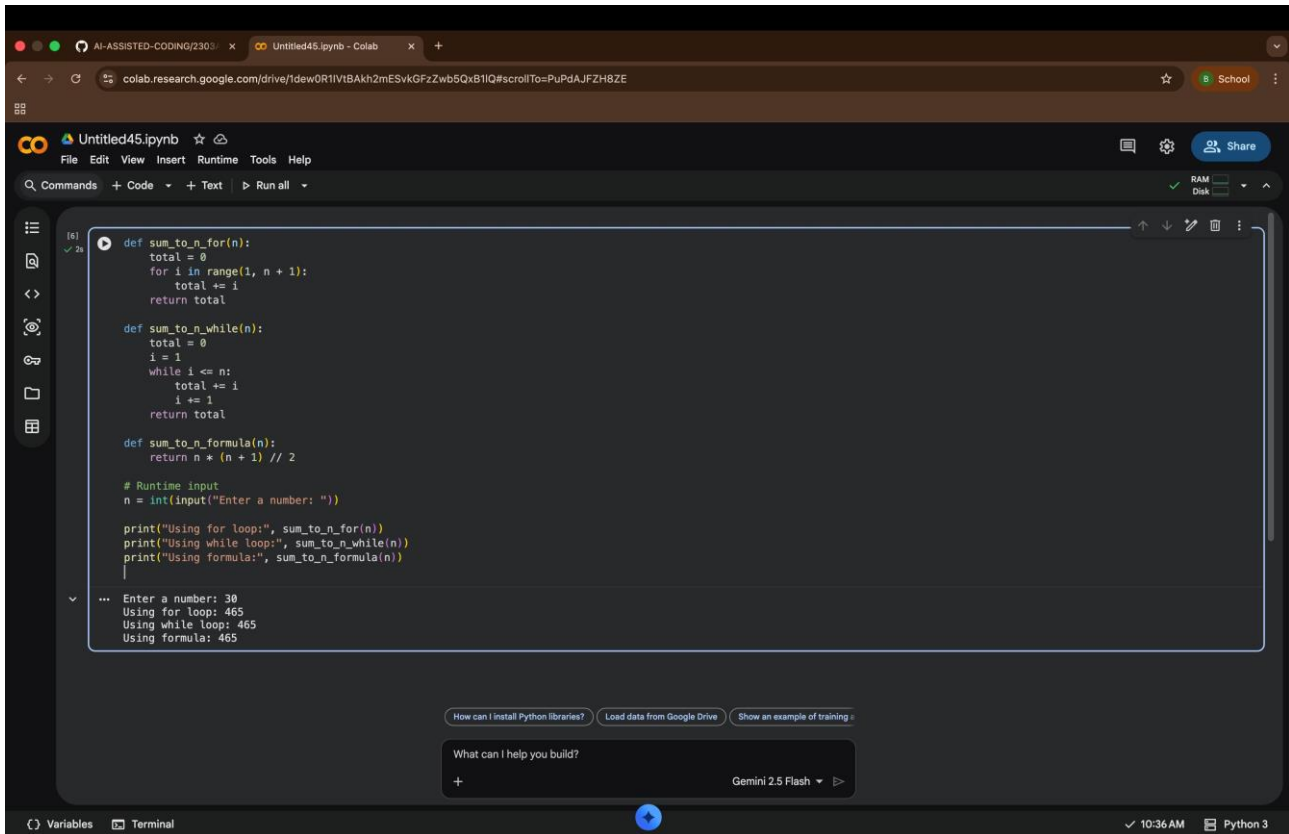
#### Expected Output #4

- 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:

The screenshot shows a Google Colab notebook interface. The browser address bar displays the URL: `colab.research.google.com/drive/1dew0R1VtBAkh2mESvkGFzZwb5Qx81lQ#scrollTo=PuPdAJFZH8ZE`. The notebook is titled "Untitled45.ipynb" and has a menu bar with options: File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu bar is a toolbar with icons for Commands, Code, Text, and Run all. The main code cell contains the following Python code:

```
[7] ✓ 3s
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()
```

Below the code cell, there are three links: "How can I install Python libraries?", "Load data from Google Drive", and "Show an example of training:". The output of the code cell is displayed in a terminal-like box:

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

At the bottom of the notebook, there is a search bar with the text "What can I help you build?" and a button labeled "Gemini 2.5 Flash". The status bar at the bottom shows "Variables", "Terminal", "10:37 AM", and "Python 3".

## 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.

