ASSIGNMENT-6.3

M.HARIKRISHNA

2203A52100

B-50

TASK-1

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.

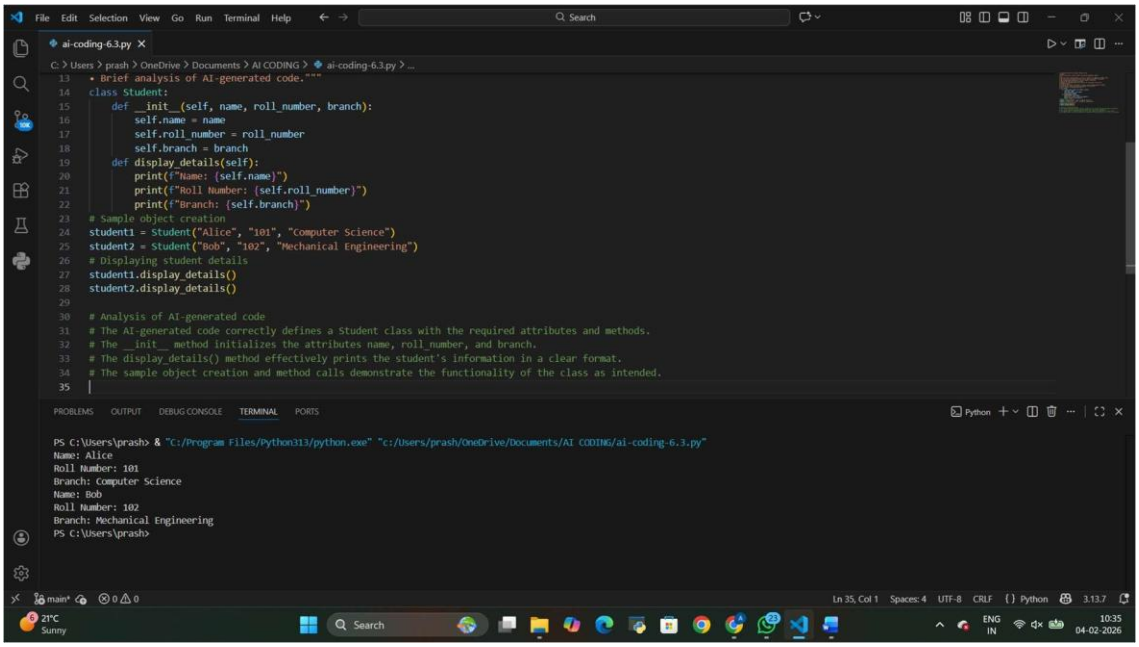
PROMPT:

Create a Python program to implement a Student class with attributes name, roll number, and branch using a constructor. Add a method display_details() to print student information. Create two student objects and display their details.

CODE:

```
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"Name: {self.name}")
print(f"Roll Number: {self.roll_number}")
print(f"Branch: {self.branch}")
# Sample object creation student1 = Student("Alice", "101",
"Computer Science")   student2 = Student("Bob", "102",
"Mechanical Engineering")
# Displaying student details
student1.display_details()
student2.display_details()
```



EXPLAINATION:

This program defines a class named Student, which is used to store and manage student information. A class is a blueprint used to create objects. Inside the class, the init method is used as a constructor. The constructor automatically runs when a new object is created and initializes the attributes name, roll number, and branch using the self keyword. The self keyword refers to the current object and helps store values inside that specific object.

The class also contains a method called display_details(). This method prints the student's name, roll number, and branch using formatted strings. When this method is called using an object, it displays that particular student's details.

After defining the class, two objects (student1 and student2) are created using the Student class. Each object stores different student data. Then the display_details() method is called for each object to print their information. This demonstrates how classes help organize data and methods together, making programs more structured and reusable.

TASK-2

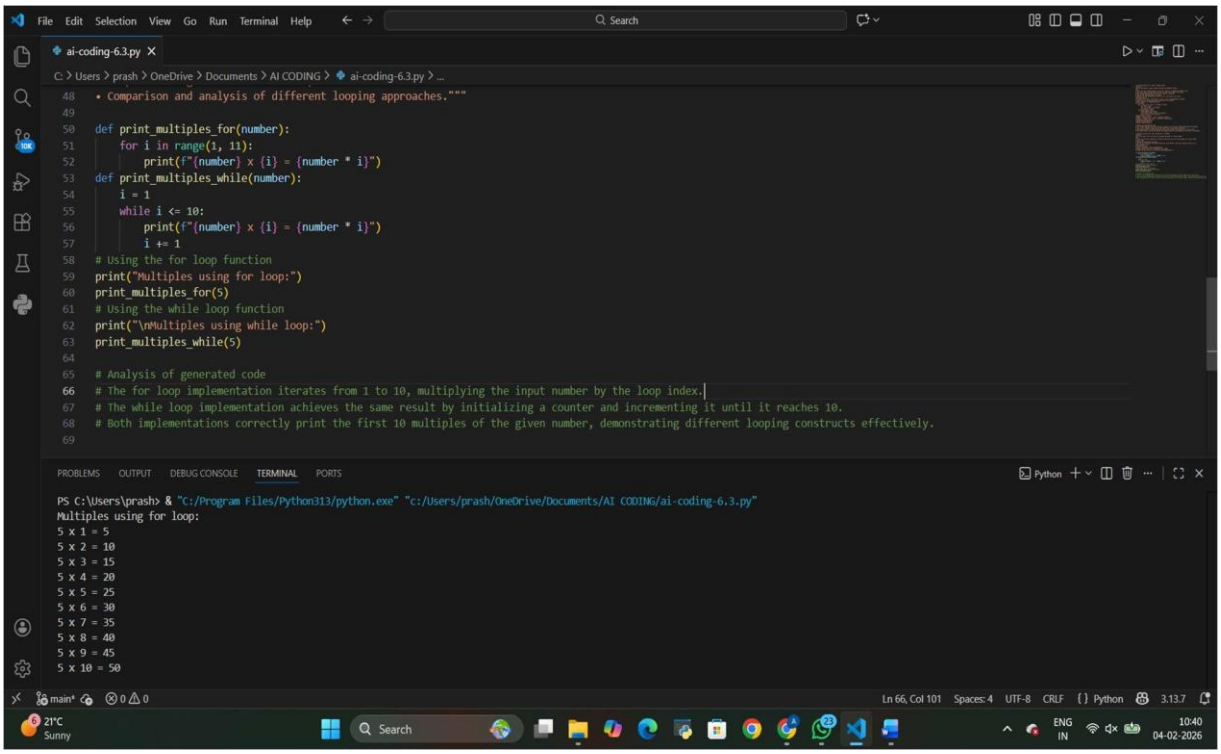
PROMPT:

Generate a Python function to print the first 10 multiples of a number using a for loop and then implement the same using a while loop, showing output and comparing both approaches.

CODE:

```
def print_multiples_for(number):
    for i in range(1, 11):
        print(f"{number} x {i} = {number * i}")
def print_multiples_while(number):
    i = 1     while
i <= 10:
        print(f"{number} x {i} = {number * i}")
        i += 1
# Using the for loop function
print("Multiples using for loop:")
print_multiples_for(5) # Using the
while loop function
print("\nMultiples using while
loop:") print_multiples_while(5)
```

OUTPUT:



EXPLAINATION:

Loop Logic Analysis

The for loop version is simpler and shorter because it automatically controls the loop using range(). It is best when the number of iterations is known. The while loop version gives more control because the loop runs based on a condition, but it requires manual initialization and increment of the counter variable. Both loops produce the same output, but the for loop is more readable, while the while loop is more flexible for complex conditions.

TASK-3

PROMPT:

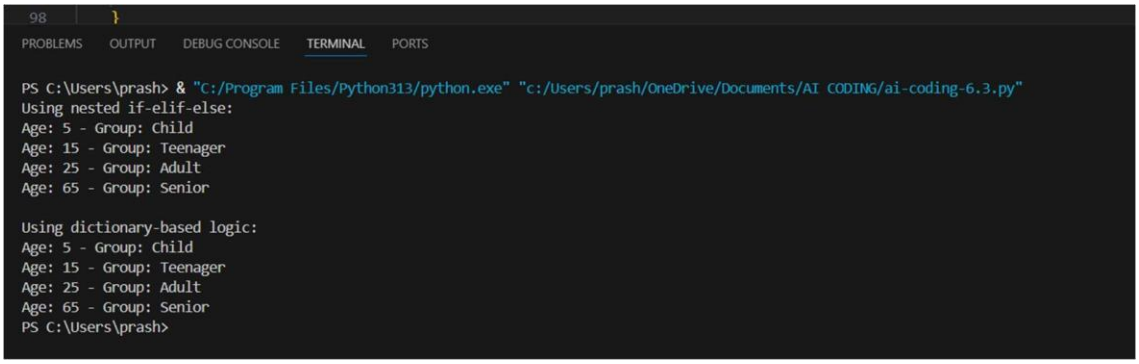
Generate a Python function to classify age into groups (child, teenager, adult, senior) using nested if-elif-else, and then create another version using simplified or dictionary-based conditional logic with explanation.

CODE:

```
def classify_age_nested(age):
    if age < 13:
```

```
        return "Child"
elif 13 <= age < 20:
        return "Teenager"
elif 20 <= age < 60:
return "Adult"     else:
        return "Senior"  def
classify_age_dict(age):
age_groups = {         range(0,
13): "Child",         range(13,
20): "Teenager",
range(20, 60): "Adult",
range(60, 150): "Senior"
    }
    for age_range, group in
age_groups.items():         if age in age_range:
return group     return "Unknown" # Testing
the functions ages = [5, 15, 25, 65]
print("Using nested if-elif-else:") for age in
ages:
    print(f"Age: {age} - Group:
{classify_age_nested(age)}") print("\nUsing dictionary-
based logic:") for age in ages:
    print(f"Age: {age} - Group: {classify_age_dict(age)}")
```

OUTPUT:



EXPLAINATION:

Conditional Logic

In the first version, the program uses nested if-elif-else statements to check age ranges step by step. If the age is less than 13, it is classified as child. If not, it checks if age is less than 20 for teenager, then less than 60 for adult, otherwise senior. This method is simple and easy to understand.

In the second version, a dictionary is used to store age group names with their conditions. The program loops through the dictionary and returns the group when the condition becomes true. This method is more flexible and easier to extend if more age groups are added.

TASK-4

PROMPT:

Generate a Python function to calculate the sum of first n natural numbers using a for loop, then provide alternative implementations using a while loop and mathematical formula with explanation and comparison.

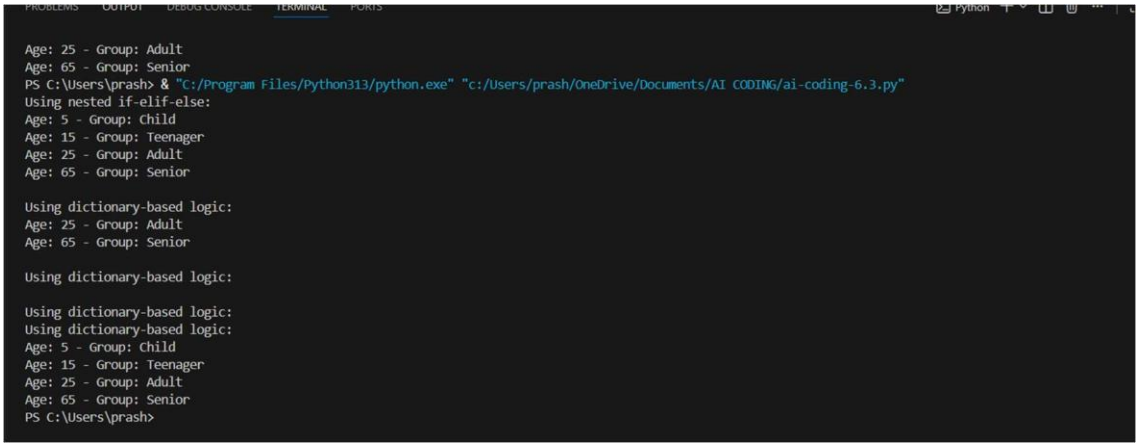
CODE:

```
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

# Testing the functions n = 10   print(f"Sum of first {n} numbers using
for loop: {sum_to_n_for(n)}") print(f"Sum of first {n} numbers using
while loop: {sum_to_n_while(n)}") print(f"Sum of first {n} numbers
using formula: {sum_to_n_formula(n)}")
```

OUTPUT:



EXPLAINATON:

The for loop method adds numbers from 1 to n using iteration and is easy to understand and commonly used. The while loop method does the same but gives more control over loop conditions, although it requires manual counter handling. The mathematical formula method is the fastest because it calculates the sum directly using the formula n(n+1)/2 without looping. For small values of n, all methods work well, but for large values, the formula method is most efficient.

TASK-5

PROMPT:

Generate a Python BankAccount class with methods deposit(), withdraw(), and check_balance(), including comments, sample usage, and explanation of how the class works.

CODE:

```
class BankAccount:     def __init__(self,
account_holder, initial_balance=0):
        """Initialize the bank account with account holder's name and an optional
initial balance."""          self.account_holder = account_holder          self.balance =
initial_balance


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


    def withdraw(self, amount):
        """Withdraw a specified amount from the account if sufficient funds are
available."""          if amount > 0:              if amount <= self.balance:
self.balance -= amount                  print(f"Withdrew: ${amount:.2f}")
            else:                print("Insufficient funds for this
withdrawal.")          else:
            print("Withdrawal amount must be positive.")


    def check_balance(self):
```

```
        """Check and return the current balance of the account."""
print(f"Current  balance:  ${self.balance:.2f}")              return
self.balance
# Sample usage account = BankAccount("John Doe",
1000)  account.check_balance()  # Check initial balance
account.deposit(500)     # Deposit money
account.withdraw(200)    # Withdraw money
account.check_balance()  # Check balance after
transactions
```

OUTPUT:

EXPLAINATION:

This program creates a BankAccount class to simulate basic banking operations. The constructor init initializes the account holder name and starting balance. The deposit() method adds money to the balance after checking if the amount is valid. The withdraw() method subtracts money only if enough balance is available, preventing overdrawing. The check_balance() method displays the current account balance. This class shows how objectoriented programming helps organize banking operations into one structured unit.