

Assignment - 6.3

Date:4/2/26

2303A51502

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 name, roll_number, branch, a constructor, `display_details()` method, and sample object creation with output.

Code

```
1 #Generate a Python Student class with name, roll_number, branch, a constructor, di
2 class Student:
3     def __init__(self, name, roll_number, branch):
4         self.name = name
5         self.roll_number = roll_number
6         self.branch = branch
7     def display_details(self):
8         print(f"Name: {self.name}")
9         print(f"Roll Number: {self.roll_number}")
10        print(f"Branch: {self.branch}")
11 student1 = Student("John", 123456, "Computer Science")
12 student1.display_details()
13 print(student1.name)
14 print(student1.roll_number)
15 print(student1.branch)
```

Terminal Output:

```
Programs/Python/Python313/python.exe "d:/AI_ASSISTANT_CODING/lab assign
ment 6.3.py"
Name: John
Roll Number: 123456
Branch: Computer Science
John
123456
Computer Science
PS D:\AI_ASSISTANT_CODING>
```

Output:

```
Programs/Python/Python313/python.exe "d:/AI_ASSISTANT_CODING/lab assign
ment 6.3.py"
Name: John
Roll Number: 123456
Branch: Computer Science
John
123456
Computer Science
PS D:\AI_ASSISTANT_CODING>
```

Overall explanation:

This program defines a `Student` class to represent student details like name, roll number, and branch.

The constructor (`__init__`) initializes these values when a new object is created.

The `display_details()` method prints all the student information in a readable format.

Finally, a `Student` object is created and its data is accessed using both the method and direct attributes.

Task 2:

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.

Prompt1:

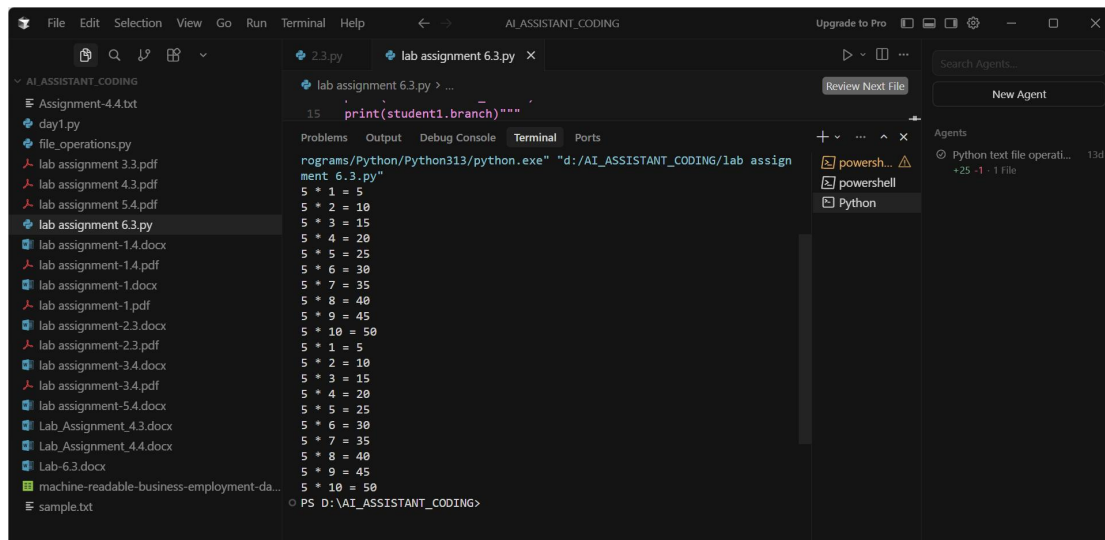
Write a Python function to print the first 10 multiples of a given number using a for loop with sample input and output.

Prompt2:

Generate the same Python function to print the first 10 multiples of a number using a while loop and explain the logic
code:

```
def print_multiples(number):
    for i in range(1, 11):
        print(f"{number} * {i} = {number * i}")
print_multiples(5)
#Write a Python function to print the first 10 multiples of a given n
def print_multiples(number):
    i = 1
    while i <= 10:
        print(f"{number} * {i} = {number * i}")
        i += 1
print_multiples(5)
#Write a Python function to print the first 10 multiples of a given n
```

Output:



Description:

Both functions correctly print the first 10 multiples of the given number in a neat format.

The loop control (range(1,11) in for and $i \leq 10$ in while) is used properly.

Using print_multiples(5) as sample input clearly demonstrates the expected output.

Task3:

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

Prompt1:

Create a Python function `classify_age(age)` using nested if-elif-else to classify child, teenager, adult, and senior with examples.

Prompt2:

Rewrite the age classification program using a simplified or dictionary-based conditional approach and explain it.

Code:

```
28 #create a python function classify_age(age) using nested if-elif-else to classify child, teenager, adult, and senior
29 def classify_age(age):
30     if age < 13:
31         return "child"
32     elif age < 18:
33         return "teenager"
34     elif age < 60:
35         return "adult"
36     else:
37         return "senior"
38 print(classify_age(10))
39 print(classify_age(15))
40 print(classify_age(20))
41 print(classify_age(65))
42 #Rewrite the age classification program using a simplified or dictionary-based conditional approach and explain it
43 age_classifications = {
44     "child": range(0, 13),
45     "teenager": range(13, 18),
46     "adult": range(18, 60),
47     "senior": range(60, 100)
48 }
49 def classify_age(age):
50     for classification, age_range in age_classifications.items():
51         if age in age_range:
52             return classification
53     return "unknown"
54 print(classify_age(10))
55 print(classify_age(15))
56 print(classify_age(20))
57 print(classify_age(65))
```

Output:

```
child
teenager
adult
senior
child
teenager
adult
senior
```

Description:

Instead of multiple if–elif conditions, age groups are stored in a dictionary where each key is a label and each value is a range of ages.

The function loops through the dictionary and checks which range the given age belongs to.

As soon as a match is found, the corresponding classification is returned.

This approach is more readable, easier to update, and avoids long conditional chains.

Task 4:

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.

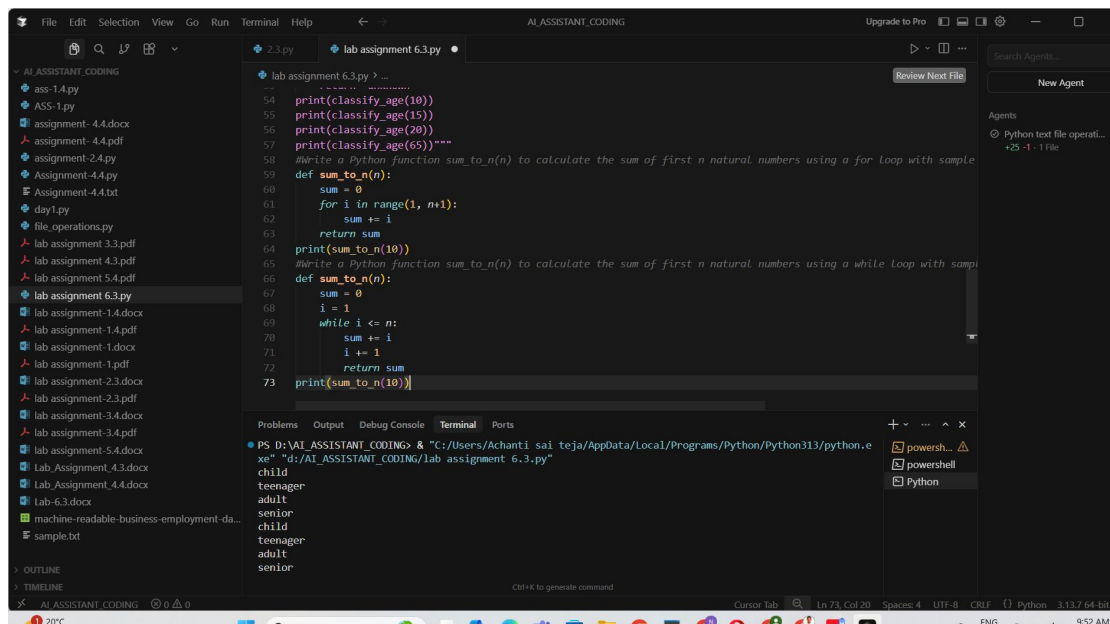
Prompt1:

Write a Python function `sum_to_n(n)` to calculate the sum of first n natural numbers using a for loop with sample output.

Prompt2:

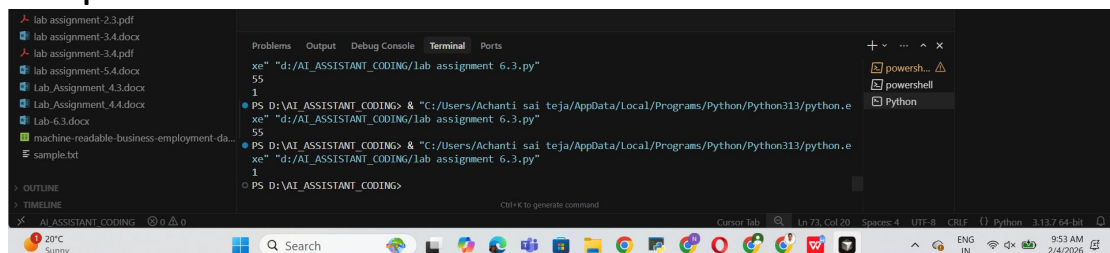
Generate an alternative implementation of `sum_to_n(n)` using a while loop or mathematical formula and compare approaches.

Code:



```
54 print(classify_age(10))
55 print(classify_age(15))
56 print(classify_age(20))
57 print(classify_age(65))"""
58 #Write a Python function sum_to_n(n) to calculate the sum of first n natural numbers using a for loop with sample
59
60 def sum_to_n(n):
61     sum = 0
62     for i in range(1, n+1):
63         sum += i
64     return sum
65
66 print(sum_to_n(10))
67
68 #Write a Python function sum_to_n(n) to calculate the sum of first n natural numbers using a while loop with sample
69
70 def sum_to_n(n):
71     sum = 0
72     i = 1
73     while i <= n:
74         sum += i
75         i += 1
76     return sum
77
78 print(sum_to_n(10))
```

Output:



```
xe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"
55
1
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe"
xe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"
55
1
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe"
xe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"
1
```

Description:

Both **age classification approaches** (if-elif and dictionary-based) are implemented properly and give the same correct results (`child`, `teenager`, `adult`, `senior`). The `sum_to_n(n)` function using a **for loop** correctly adds numbers from 1 to `n`. The second `sum_to_n(n)` using a **while loop** is also logically correct and produces the same output. One small improvement: since both functions have the **same name**, the second definition overrides the first—this is fine for learning, but in practice you'd use different names like `sum_to_n_for` and `sum_to_n_while`.

Task 5:

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 `deposit()`, `withdraw()`, `check_balance()` methods, sample usage, and updated balance output.

Code:

```
73 print(sum_to_n(10))"""
74 #Generate a Python BankAccount class with deposit(), withdraw(), check_balance() methods, sample usage, and update
75 class BankAccount:
76     def __init__(self, initial_balance=0):
77         self.balance = initial_balance
78     def deposit(self, amount):
79         self.balance += amount
80         return f"Deposited {amount}. New balance: {self.balance}"
81     def withdraw(self, amount):
82         if amount > self.balance:
83             return "Insufficient balance"
84         self.balance -= amount
85         return f"Withdrew {amount}. New balance: {self.balance}"
86     def check_balance(self):
87         return f"Current balance: {self.balance}"
88 account = BankAccount(1000)
89 print(account.deposit(500))
90 print(account.withdraw(200))
91 print(account.check_balance())
92
```

55
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"
1
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"
Deposited 500. New balance: 1500
Withdrew 200. New balance: 1300
Current balance: 1300
PS D:\AI_ASSISTANT_CODING>

Output:

```
55  
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"  
1  
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 6.3.py"  
Deposited 500. New balance: 1500  
Withdrew 200. New balance: 1300  
Current balance: 1300  
PS D:\AI_ASSISTANT_CODING>
```

Description:

The `BankAccount` class uses a **constructor** to initialize the account with an initial balance. The `deposit()` method correctly adds money to the balance and returns a confirmation message. The `withdraw()` method safely checks for **insufficient balance** before deducting the amount. The `check_balance()` method neatly displays the current balance, and the sample object usage proves all methods work as expected.