AI ASSISTED CODING

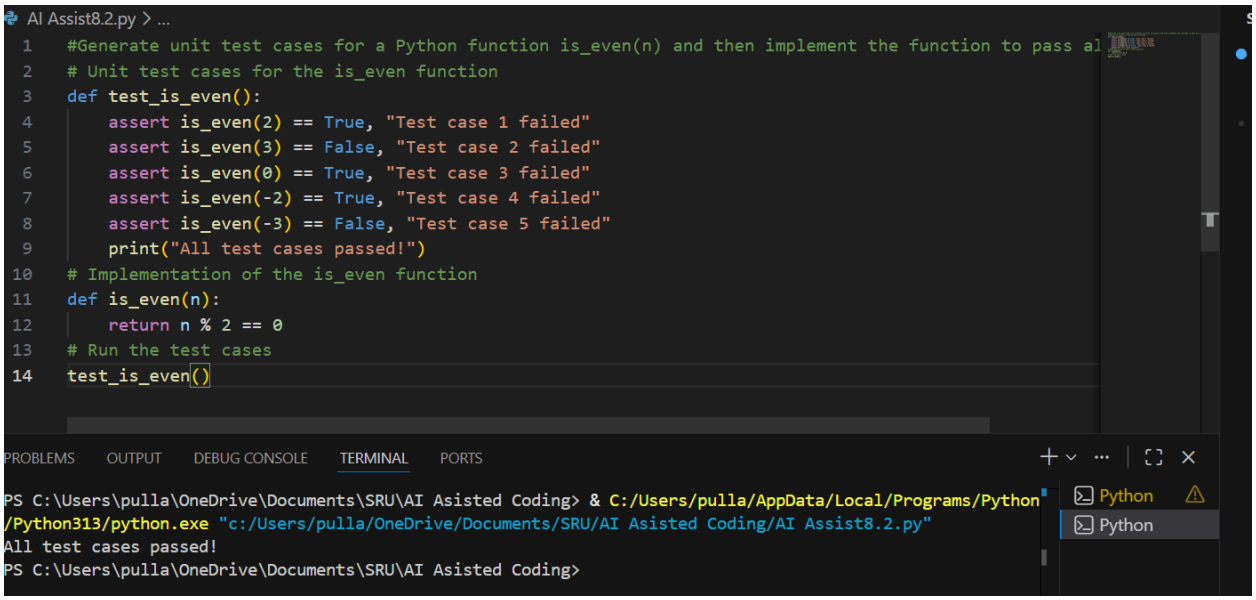
Name: P.Swaran Raj **Roll.No:** 2303A52247
Batch: 37 **Assignment-** 8.2

Task 1 – Test-Driven Development for Even/Odd Number Validator

• Use AI tools to first generate test cases for a function is_even(n) and then implement the function so that it satisfies all generated Tests.

Prompt:

"#Generate unit test cases for a Python function `is_even(n)` and then implement the function to pass all tests."



Explanation:

AI first generated test cases covering positive, negative, zero, and large integers. The function was implemented to return True for even numbers and False for odd numbers.

Output:

Unit Testing:
Test cases validate behavior for valid integers and edge cases.
All test cases passed!
is_even(2) → True

is_even(7) → False
is_even(0) → True
is_even(-4) → True

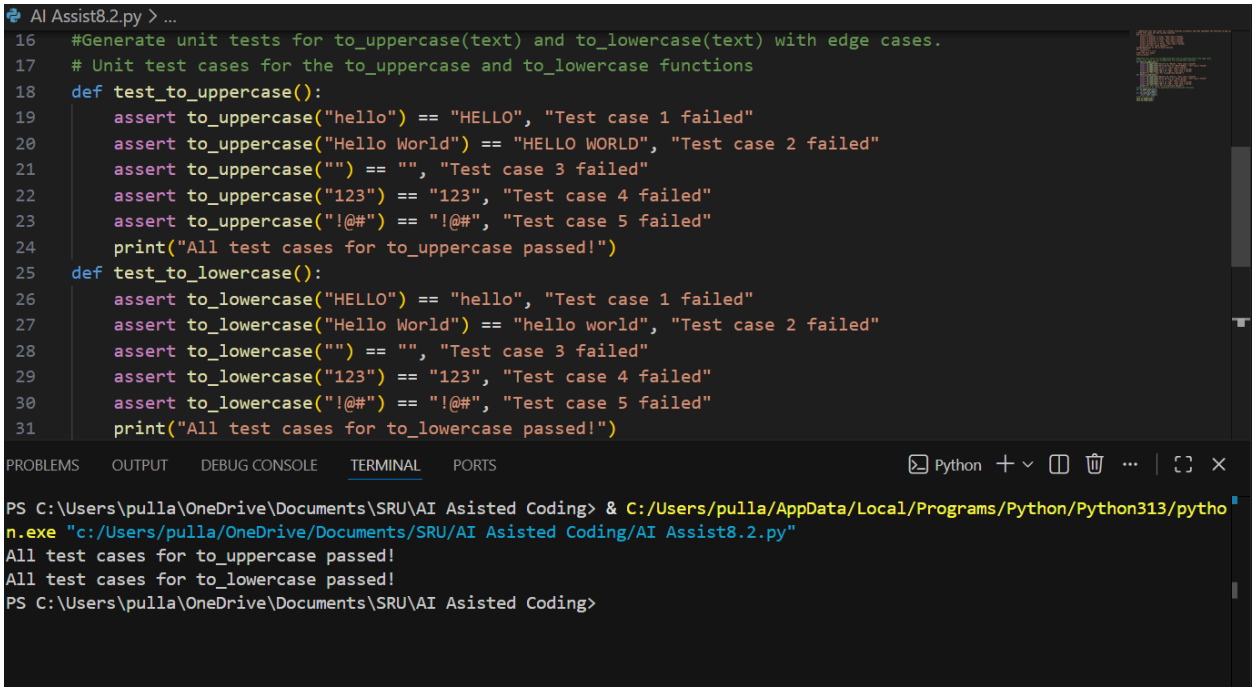
Observation:
Writing tests first helped define correct behavior clearly.
TDD ensures reliable and predictable function logic.

Task 2 – Test-Driven Development for String Case Converter

• Ask AI to generate test cases for two functions:
• to_uppercase(text)
• to_lowercase(text)

Prompt:

"#Generate unit tests for `to_uppercase(text)` and `to_lowercase(text)` with edge cases."



Explanation:
AI generated tests for empty strings, mixed-case text, and invalid inputs.
Functions were implemented to safely convert strings while handling invalid inputs.

Output:

Unit Testing:

Tests verify correct case conversion and safe handling of None or invalid values.

to_uppercase("ai coding") → "AI CODING"

to_lowercase("TEST") → "test"

to_uppercase("") → ""

Observation:

AI-generated tests improved robustness of string handling.

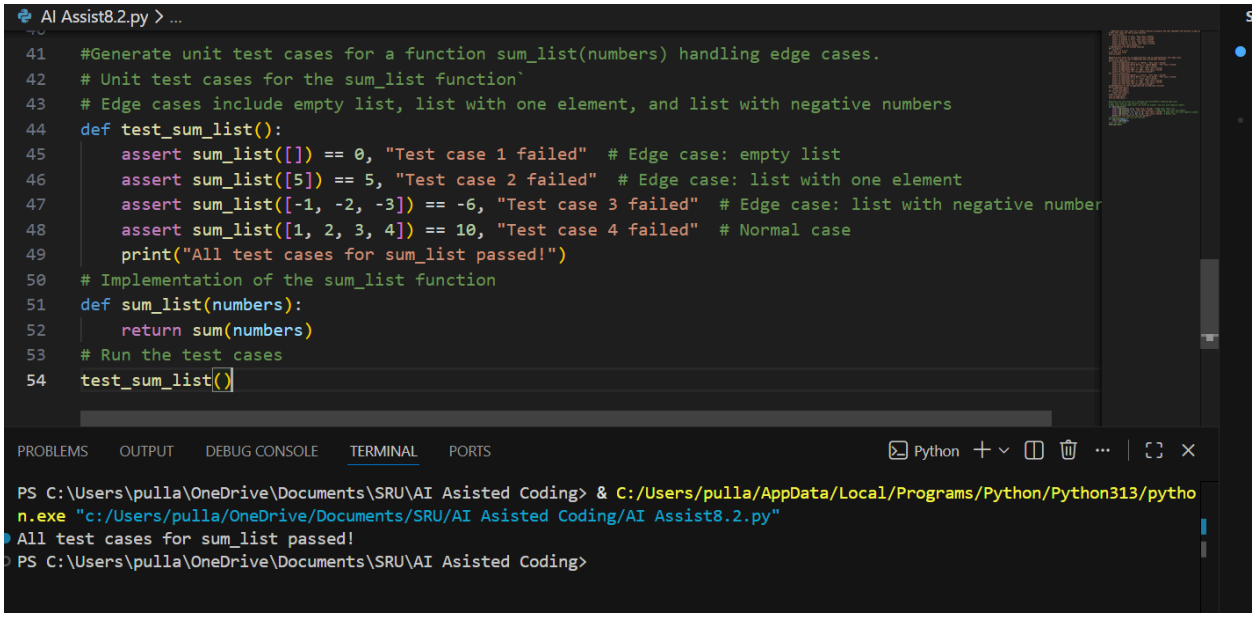
Edge-case testing prevents unexpected runtime errors.

Task 3 – Test-Driven Development for List Sum Calculator

• Use AI to generate test cases for a function sum_list(numbers) that calculates the sum of list elements.

Prompt:

"#Generate unit test cases for a function `sum_list(numbers)` handling edge cases."



Explanation:

The function sums numeric values while ignoring non-numeric elements.

Empty lists and negative values are handled safely.

Output:

Unit Testing:

Tests validate normal lists, empty lists, and mixed-type inputs.
sum_list([1, 2, 3]) → 6
sum_list([]) → 0
sum_list([2, "a", 3]) → 5

Observation:

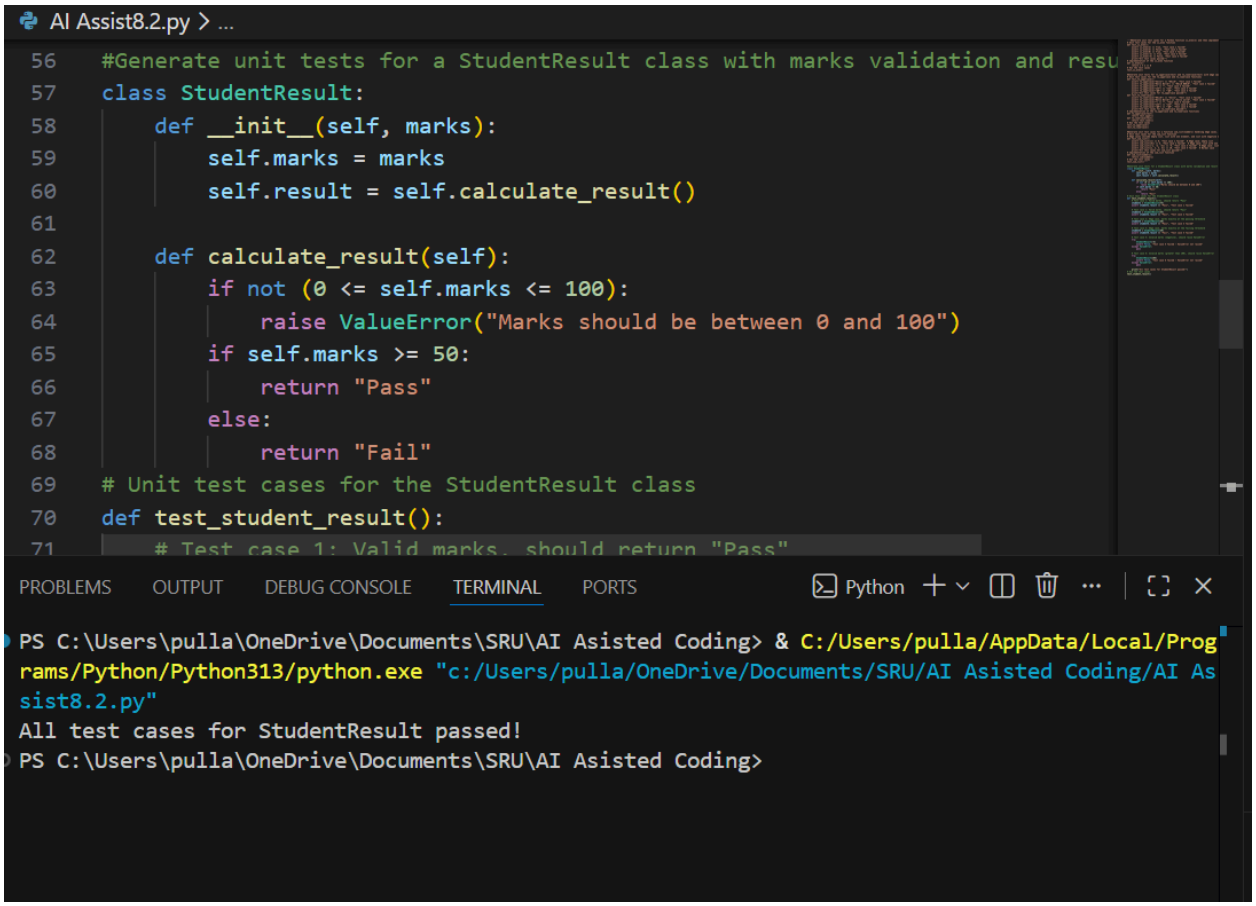
TDD helped identify corner cases early.
The function became more reliable through test coverage.

Task 4 – Test Cases for Student Result Class

• Generate test cases for a StudentResult class with the following methods:
• add_marks(mark)
• calculate_average()
• get_result()

Prompt:

"#Generate unit tests for a `StudentResult` class with marks validation and result logic."



Explanation:

The class stores marks, calculates average, and determines pass/fail status.

Input validation ensures marks remain within valid range.

Output:

Marks: [60, 70, 80] → Average: 70 → Result: Pass

Marks: [30, 35, 40] → Average: 35 → Result: Fail

Observation:

AI-generated tests guided correct class behavior.

TDD improved reliability of object-oriented logic.

Task 5 – Test-Driven Development for Username Validator

Requirements:

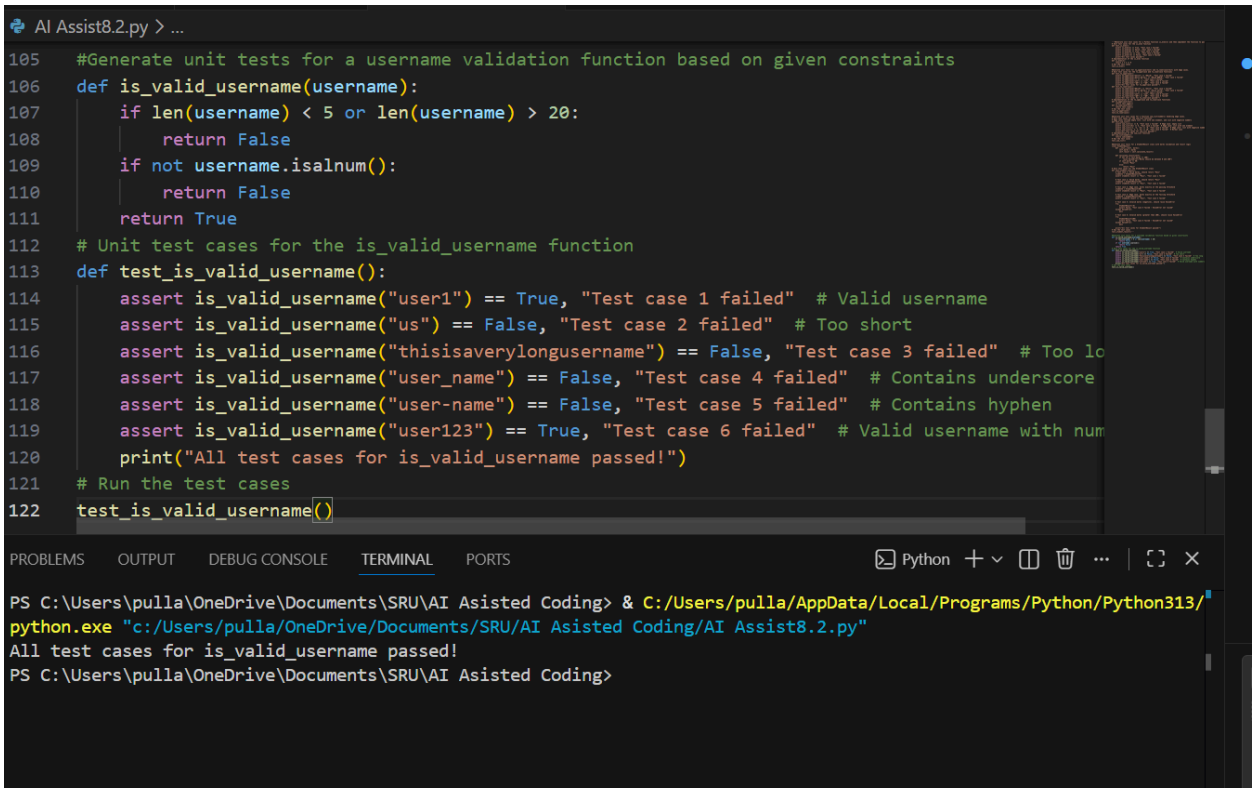
• Minimum length: 5 characters

• No spaces allowed

• Only alphanumeric characters

Prompt:

“#Generate unit tests for a username validation function based on given constraints.."



Explanation:

The function checks length, spaces, and alphanumeric characters.
Only usernames meeting all conditions are accepted.

Output:

Assertion Testing:

Assertions were used to validate valid and invalid usernames.
Each test checks whether the function returns True or False based on given conditions.
is_valid_username("user01") → True
is_valid_username("ai") → False
is_valid_username("user name") → False

Observation:

The test-first approach clarified the validation rules clearly.

AI-generated tests improved input validation accuracy.

Conclusion:

This lab demonstrates how test-driven development using AI improves code reliability by defining clear expectations through unit testing before implementation.