SR UNIVERSITY

AI ASSIST CODING

Lab-8: Test-Driven Development with AI – Generating and Working with Test Cases

ROLL NO:2503A51L11

NAME:P.Susmija

BATCH: 24BTCAICSB19

Lab Objectives:

• To introduce students to test-driven development (TDD) using AI code generation tools.

• To enable the generation of test cases before writing code implementations.

• To reinforce the importance of testing, validation, and error handling.

• To encourage writing clean and reliable code based on AI-generated test expectations

Lab Outcomes (LOs):

After completing this lab, students will be able to:

• Use AI tools to write test cases for Python functions and classes.

• Implement functions based on test cases in a test-first development style.

• Use unittest or pytest to validate code correctness.

• Analyze the completeness and coverage of AI-generated tests.

• Compare AI-generated and manually written test cases for quality and logics.

TASK#1

PROMPT

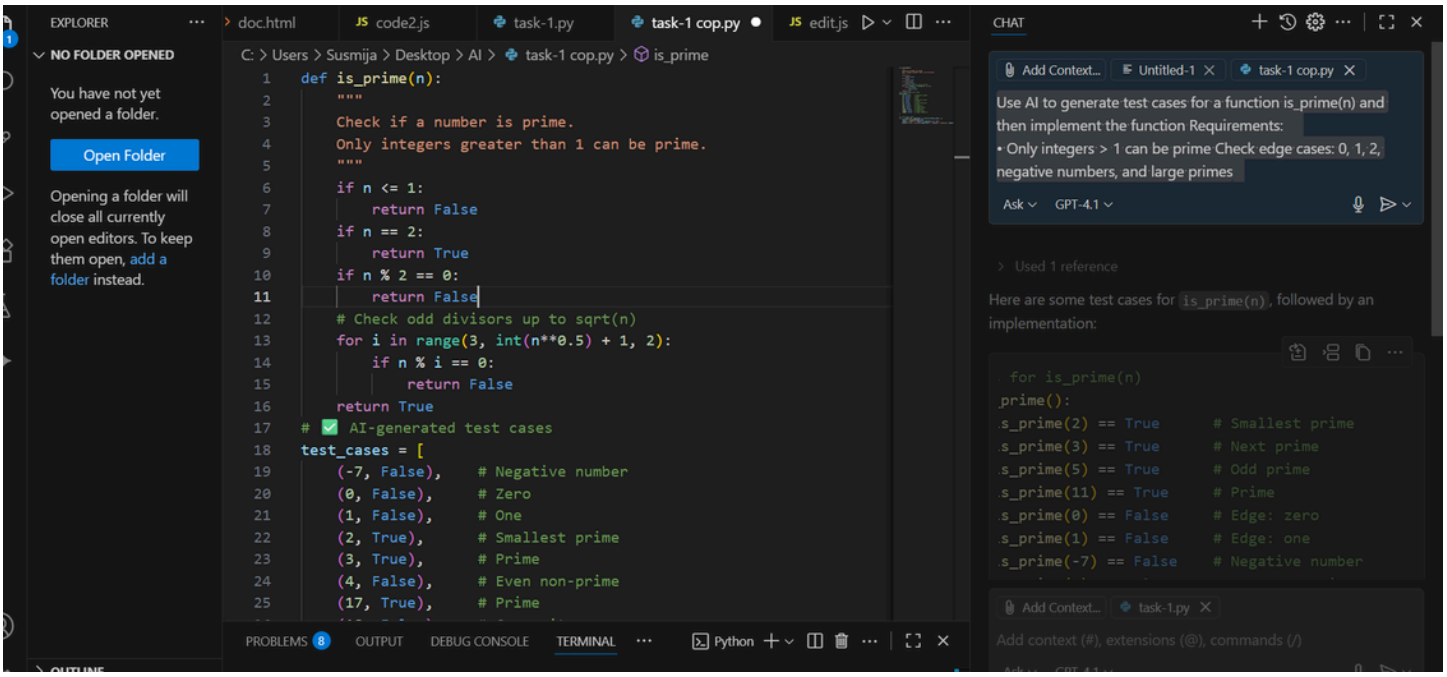
- Use AI to generate test cases for a function is_prime(n) and then implement the function

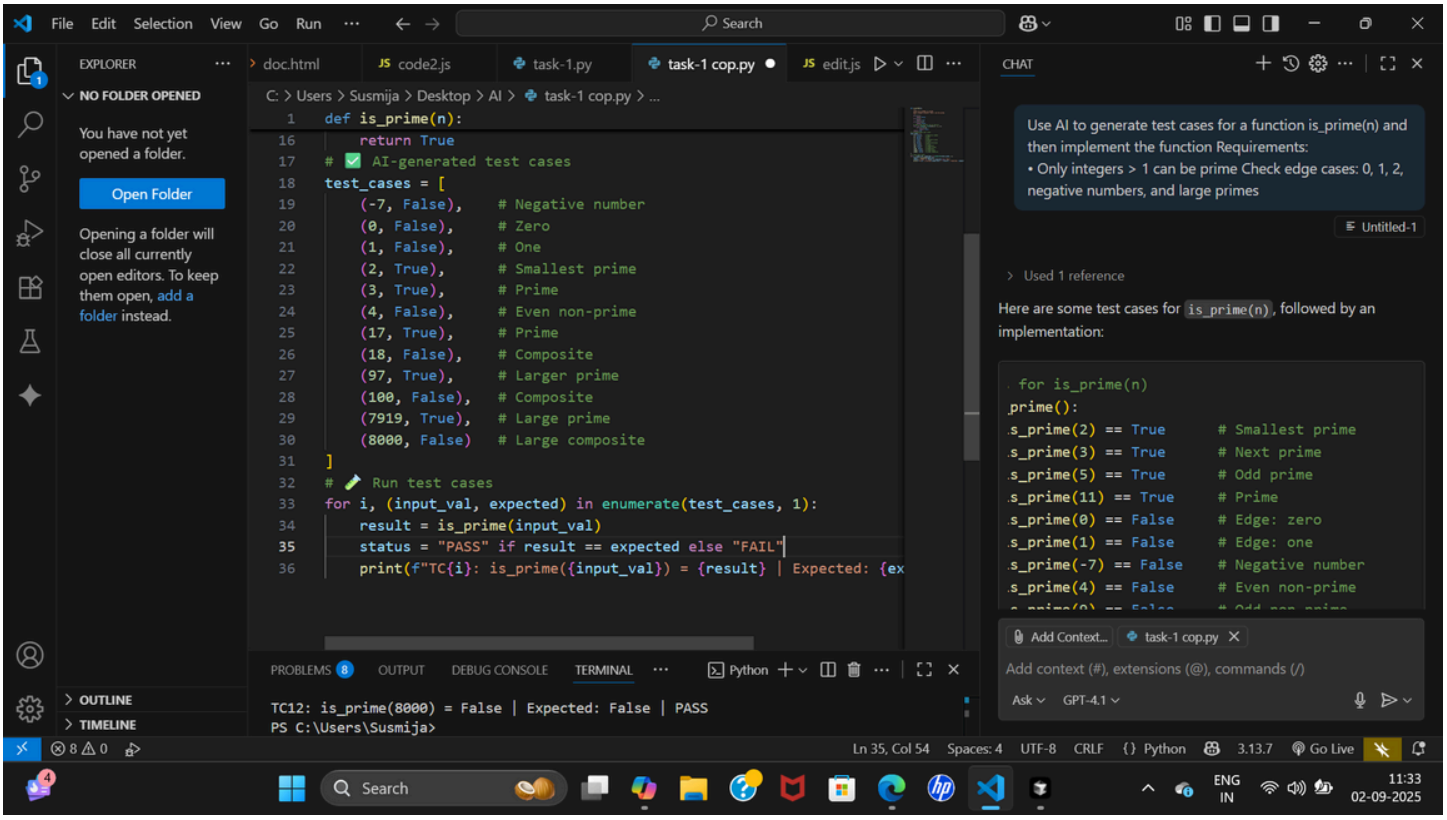
  Requirements:

  • Only integers > 1 can be prime.

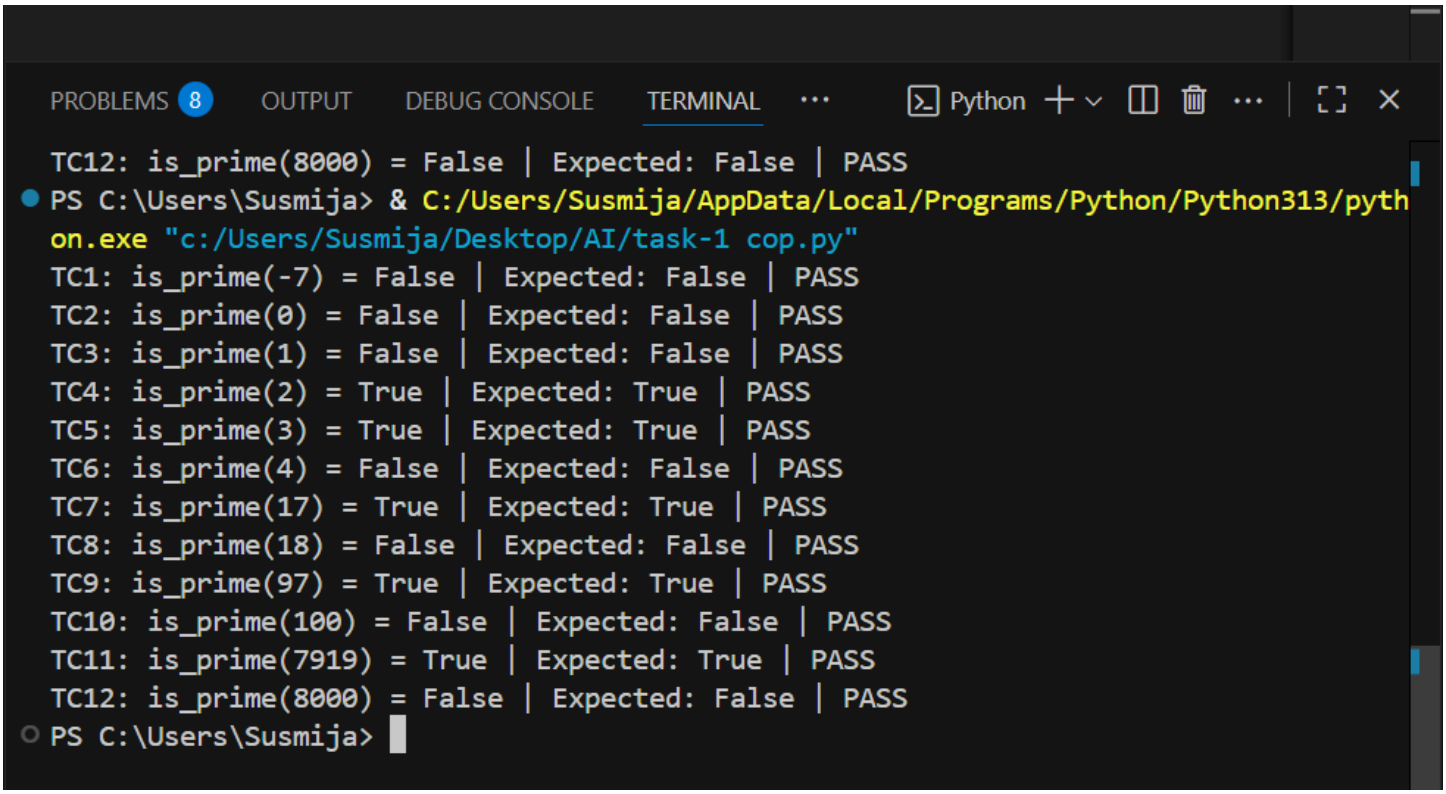
  •Check edge cases: 0, 1, 2, negative numbers, and large primes

CODE GENERATED:





OUTPUT OF THE CODE:



OBSERVATIONS:

Purpose of the Function

- The function is_prime(n) is designed to determine if a given integer n is a prime number.
- A prime number is defined as a number greater than 1 that has no divisors other than 1 and itself.

Edge Case Handling:

- Handles:
  - Negative numbers (e.g., -7)
  - Zero (0)
  - One (1)
  - Two (2) as the smallest and only even prime

Even Number Optimization:

- Early exit for even numbers (n % 2 == 0) improves efficiency.

Efficient Loop for Checking Factors:

- Iterates only through **odd divisors up to √n**, skipping even numbers after 2. This optimizes performance for large inputs

TASK#2

PROMPT

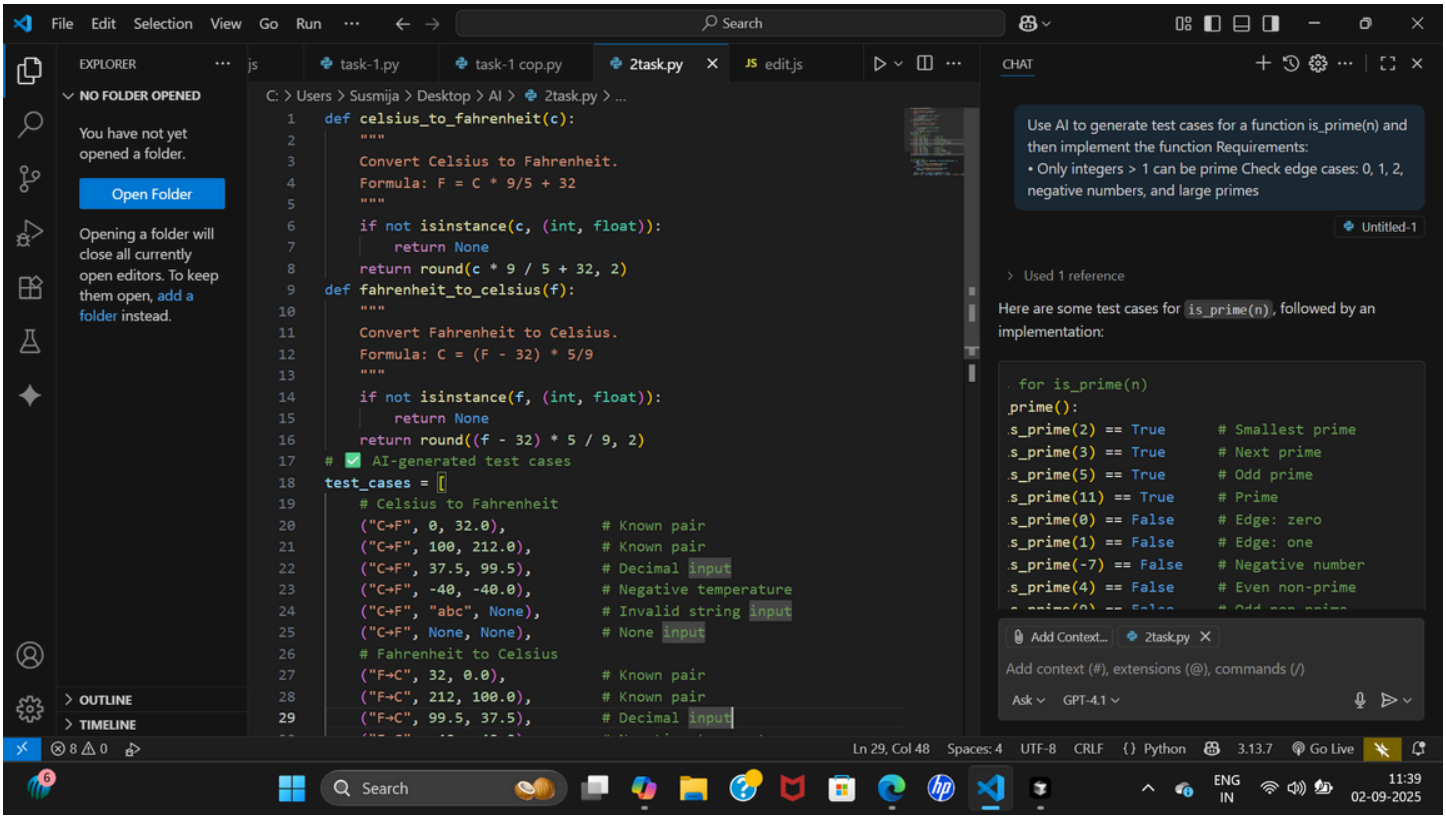
- Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f)

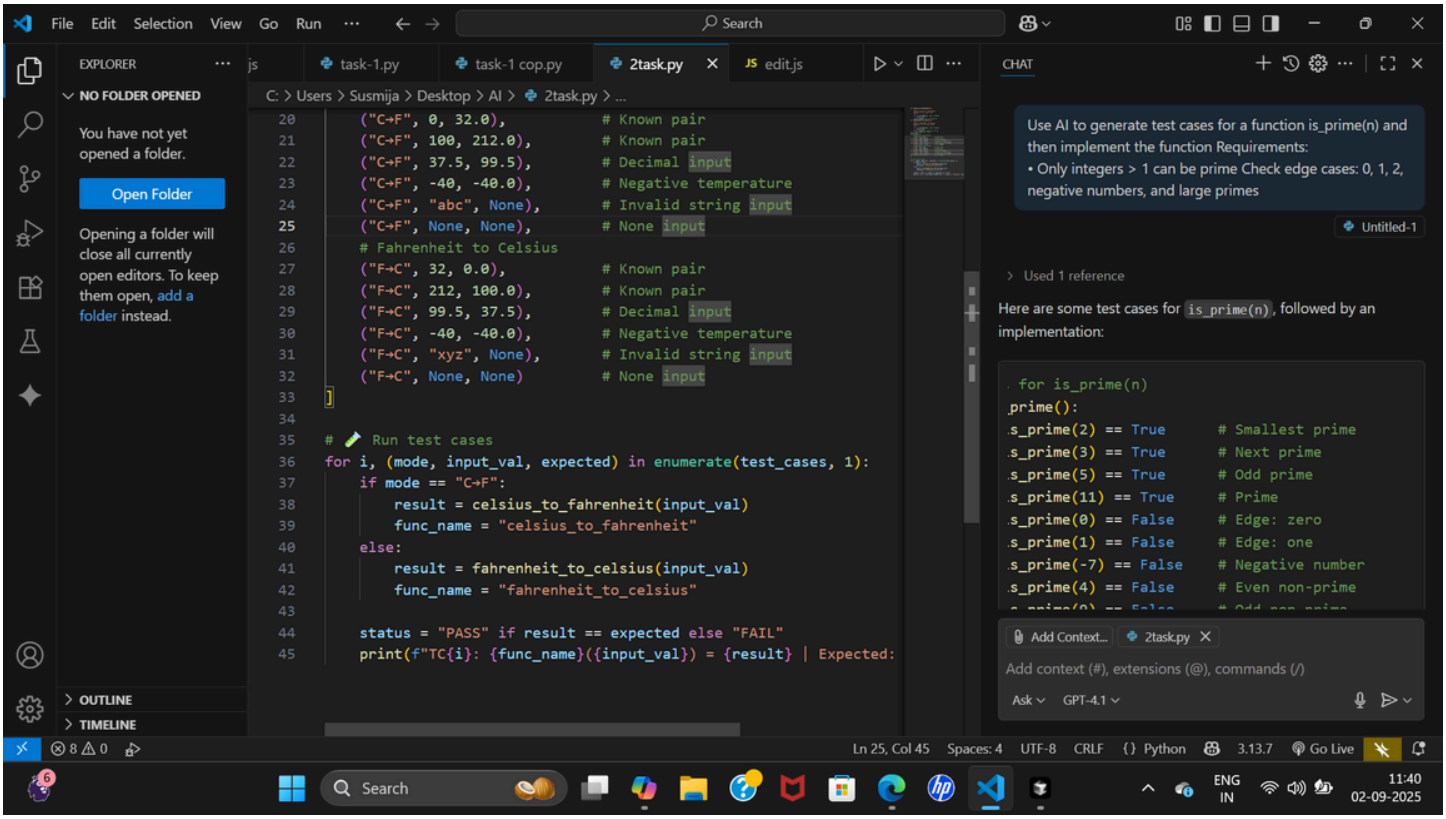
Requirements

• Validate known pairs: 0°C = 32°F, 100°C = 212°F.

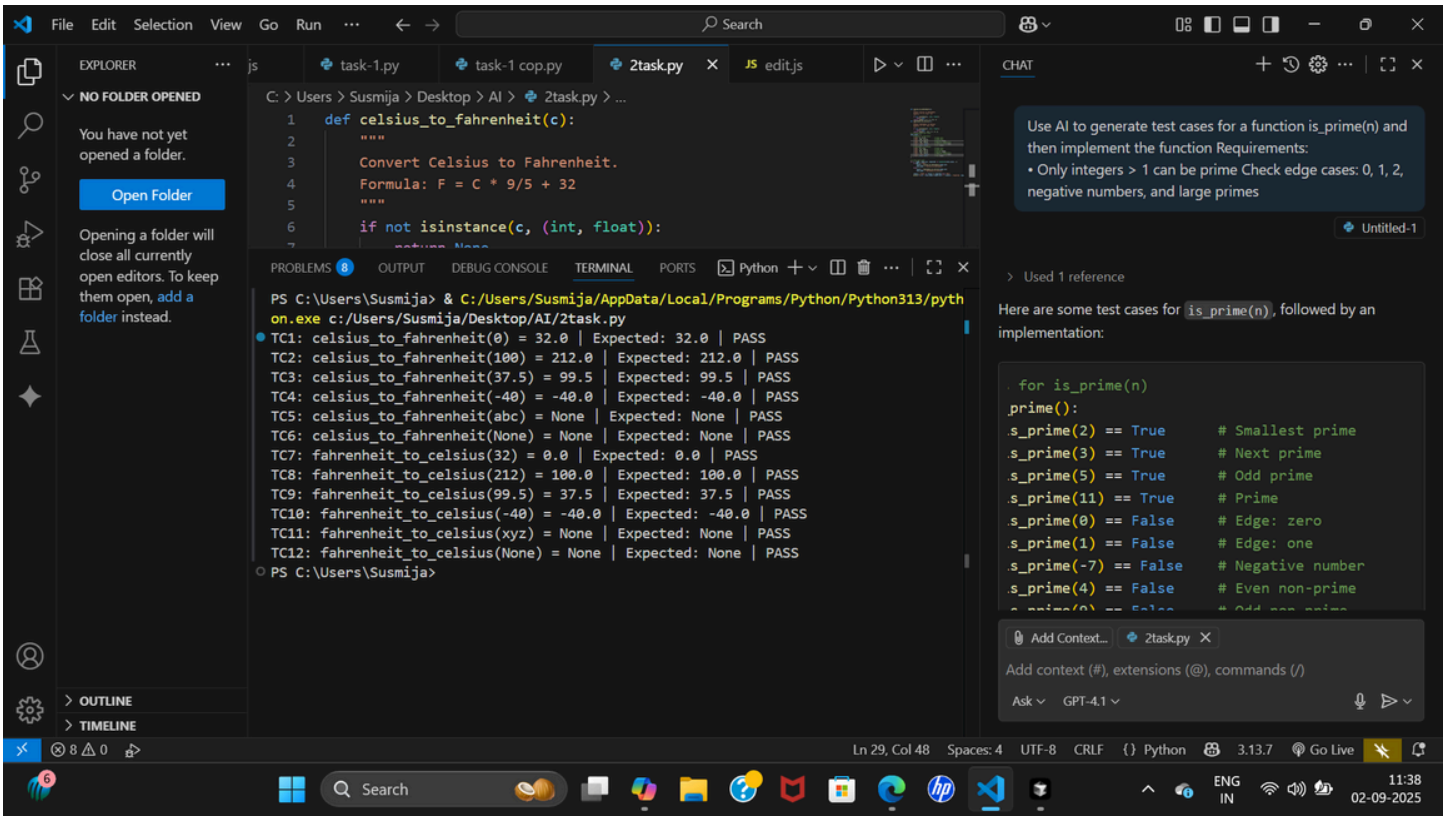
• Include decimals and invalid inputs like strings or None

CODE GENERATED:





OUTPUT OF THE CODE:



OBSERVATIONS:

Test Cases Defined in test_cases List

- Clearly structured as tuples in the format:
- ("ConversionType", input_value, expected_output)

🔹 Coverage:

- **Valid Inputs:**
  - Known conversion pairs (e.g., 0°C → 32°F, 212°F → 100°C)
  - Decimal temperatures (e.g., 37.5°C)
  - Negative values (e.g., -40°C)
- **Invalid Inputs**:
  - Strings (e.g., "abc")
  - None values

Consider Handling Edge Cases Explicitly:

- e.g., math.inf, float('nan'), very large/small values.

Use of Constants for Precision:

- Optionally define rounding precision as a constant for easy adjustments.

TASK#3

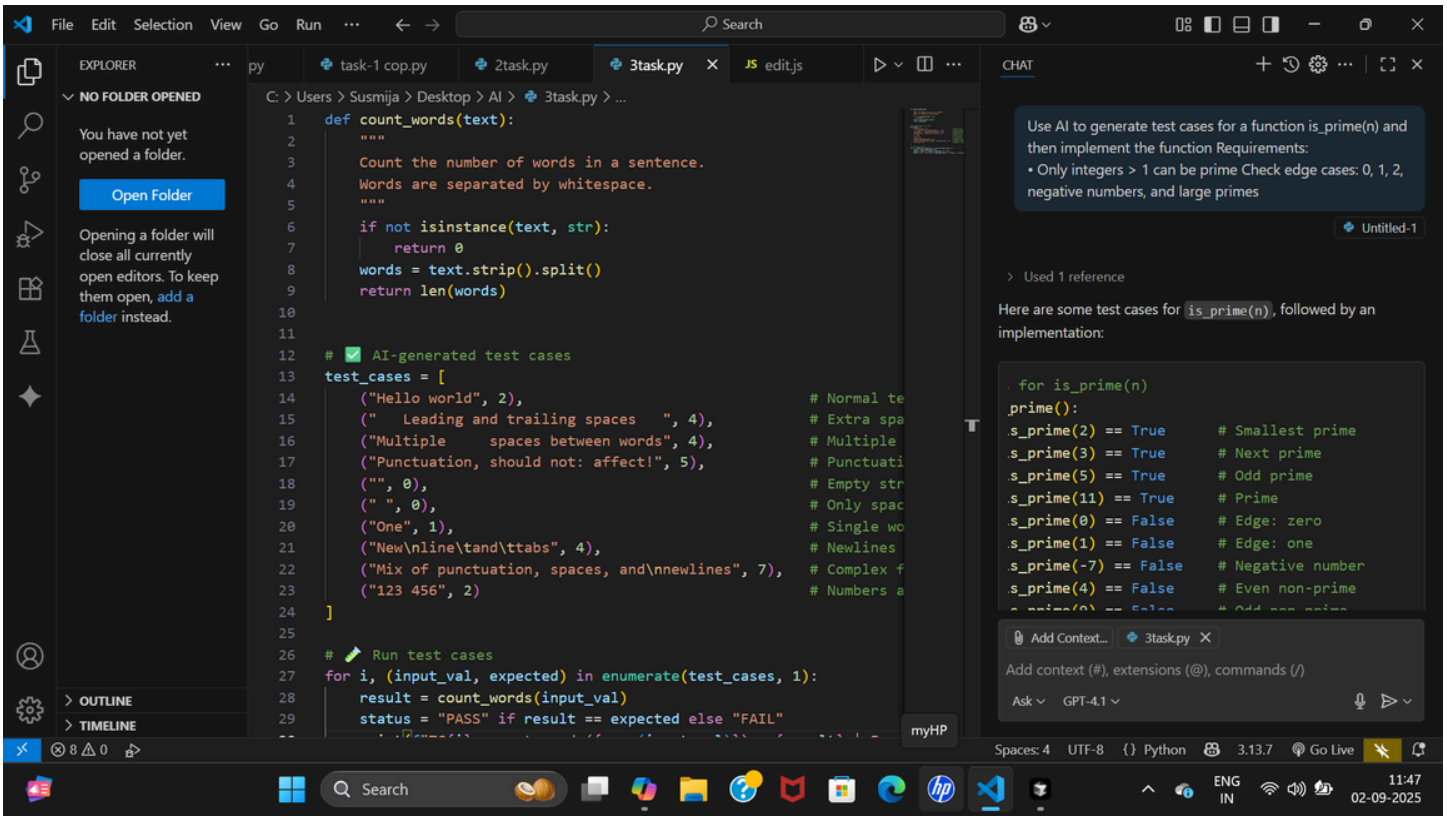
PROMPT

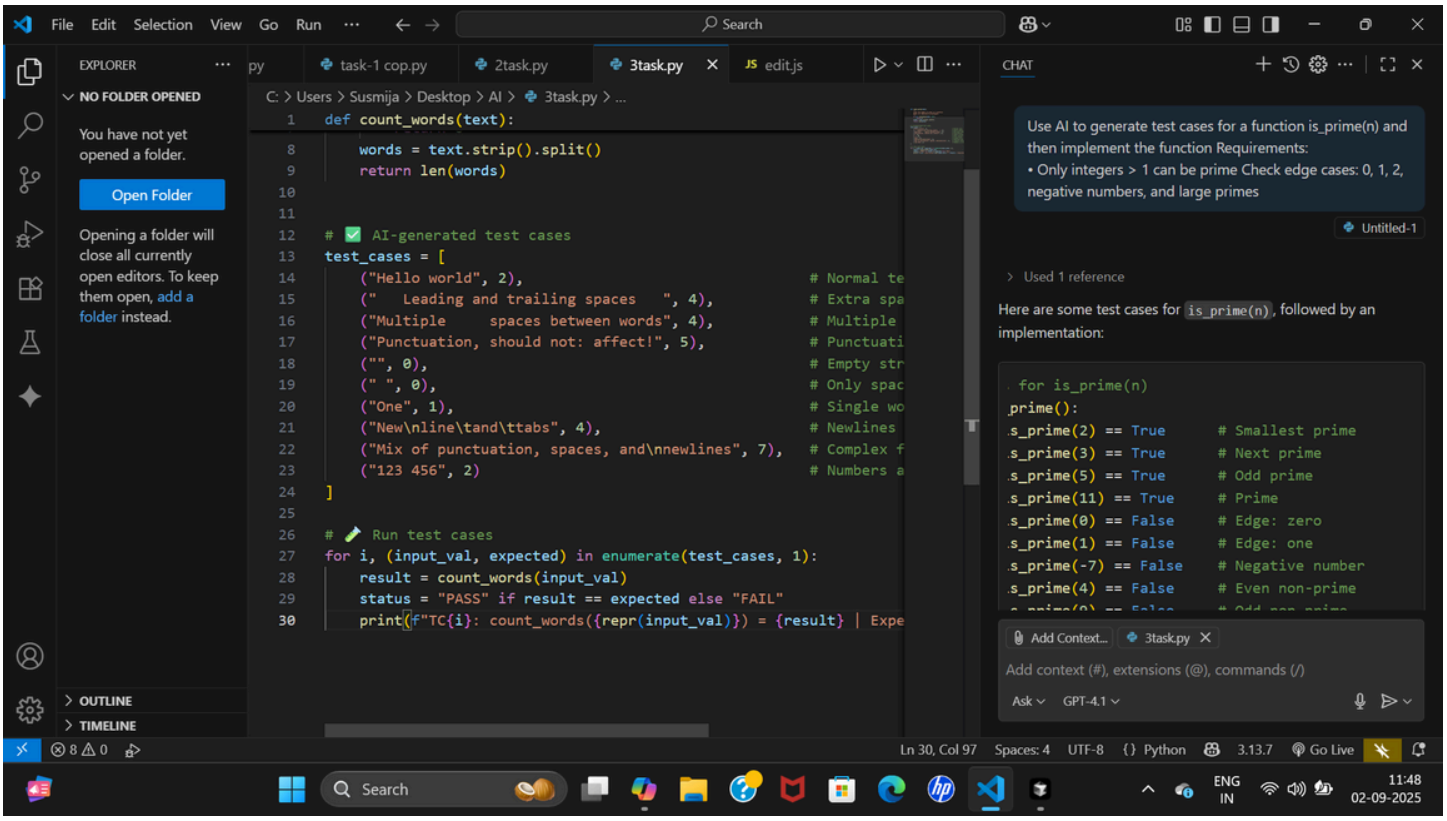
- Use AI to write test cases for a function count_words(text) that returns the number of words in sentences

Requirements

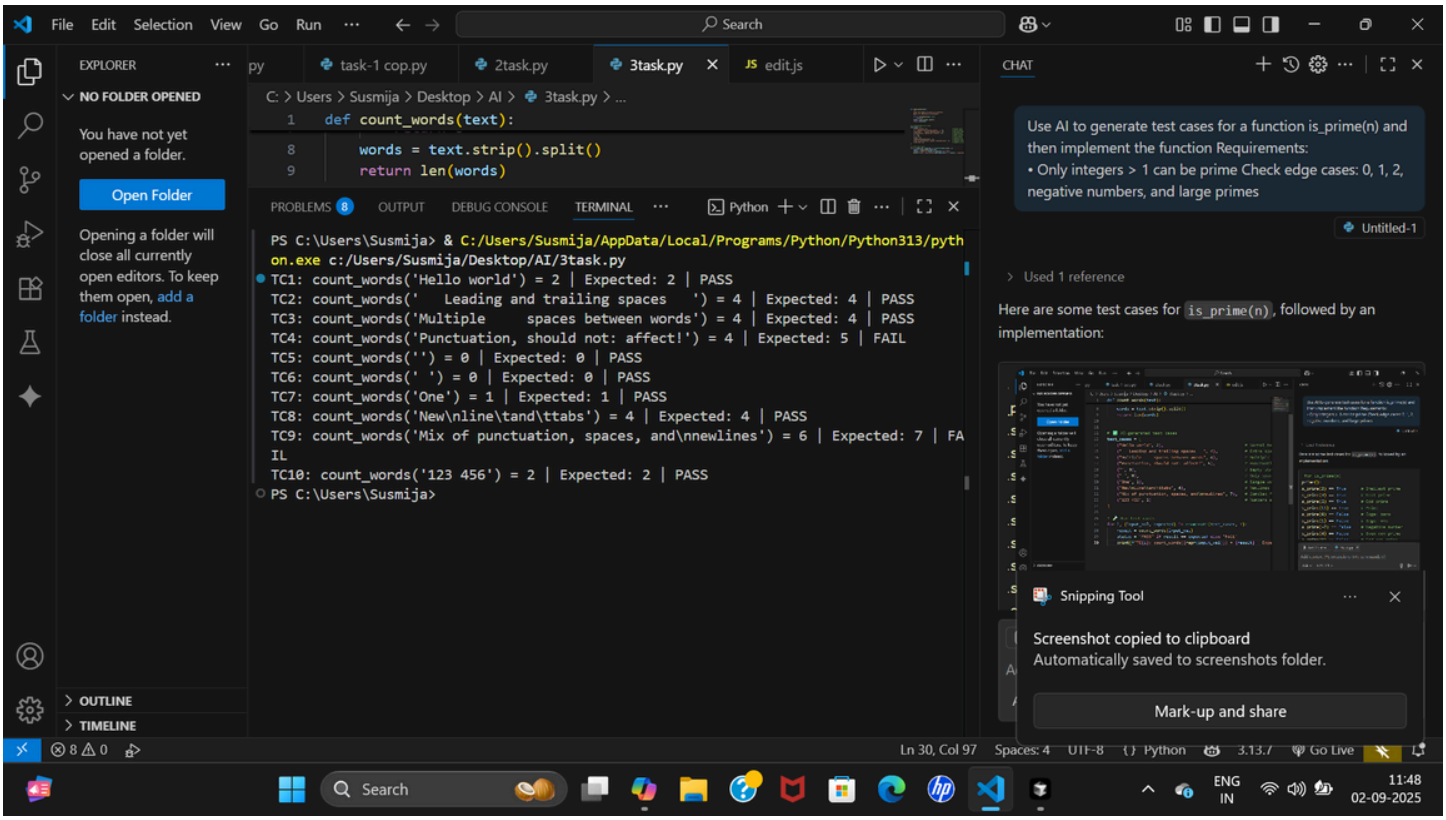
Handle normal text, multiple spaces, punctuation, and empty strings.

CODE GENERATED:





OUTPUT OF THE CODE:



OBSERVATIONS:

Function: count_words(text)

- Goal: Counts the number of words in a string.
- Logic:
  - Removes leading and trailing spaces using strip().
  - Splits the string into words using whitespace as a delimiter (split()).
  - Returns the count of words using len().
- If the input text is not a string, it returns 0. This protects the function from invalid types (like numbers or None).
- Efficient Use of Built-in Functions:
- Uses strip() to clean up extra spaces.
- Uses split() to split based on any whitespace (handles spaces, tabs, newlines).

Handles Edge Cases Gracefully:

- Empty strings, strings with only spaces, newline and tab characters — all are handled.

TASK#4

PROMPT

- Generate test cases for a BankAccount class with:

  Methods:

  deposit(amount)

  withdraw(amount)

check_balance()

Requirements:

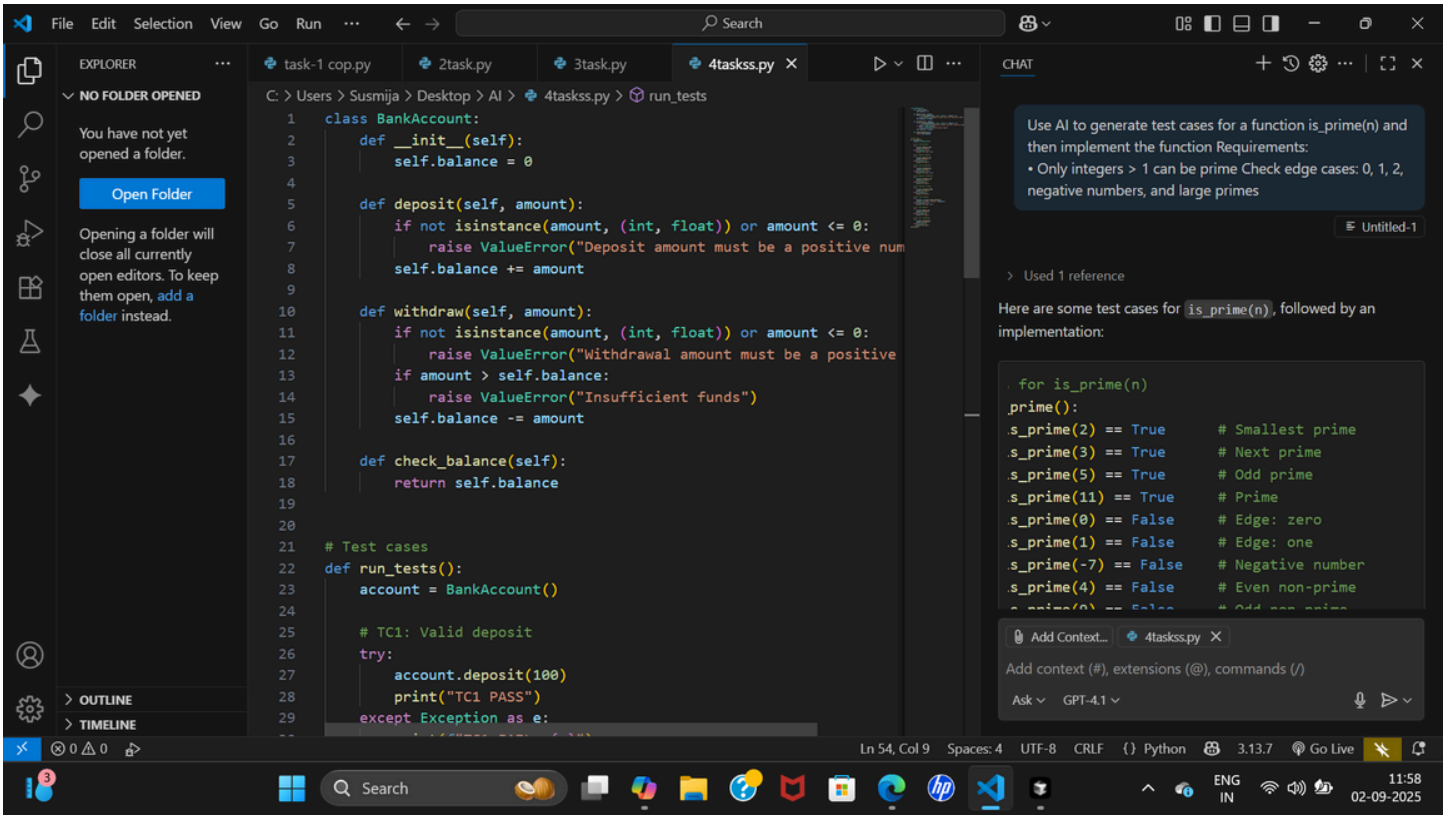
• Negative deposits/withdrawals should raise an error.

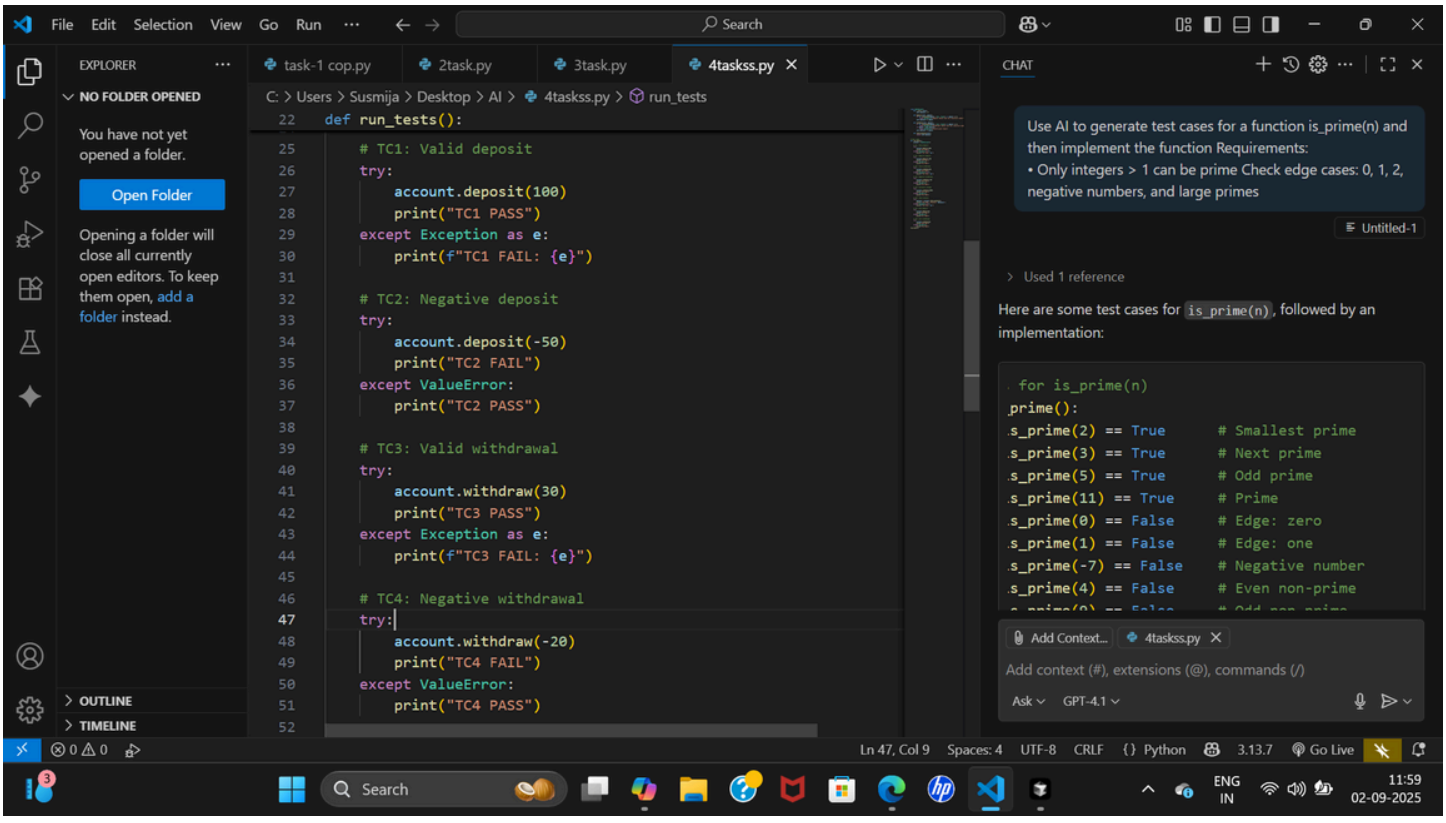
• Cannot withdraw more than balance.

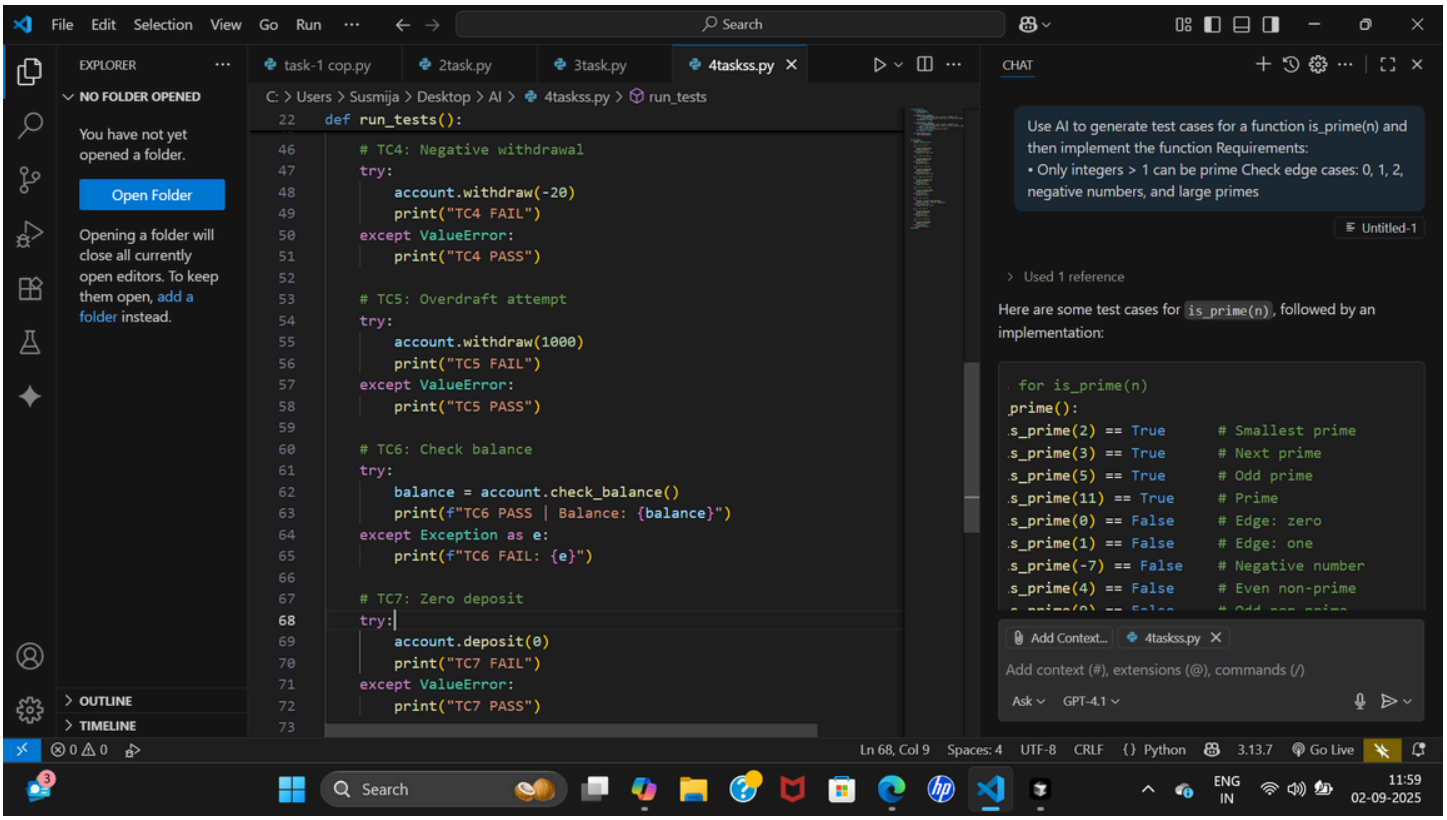
Expected Output#4

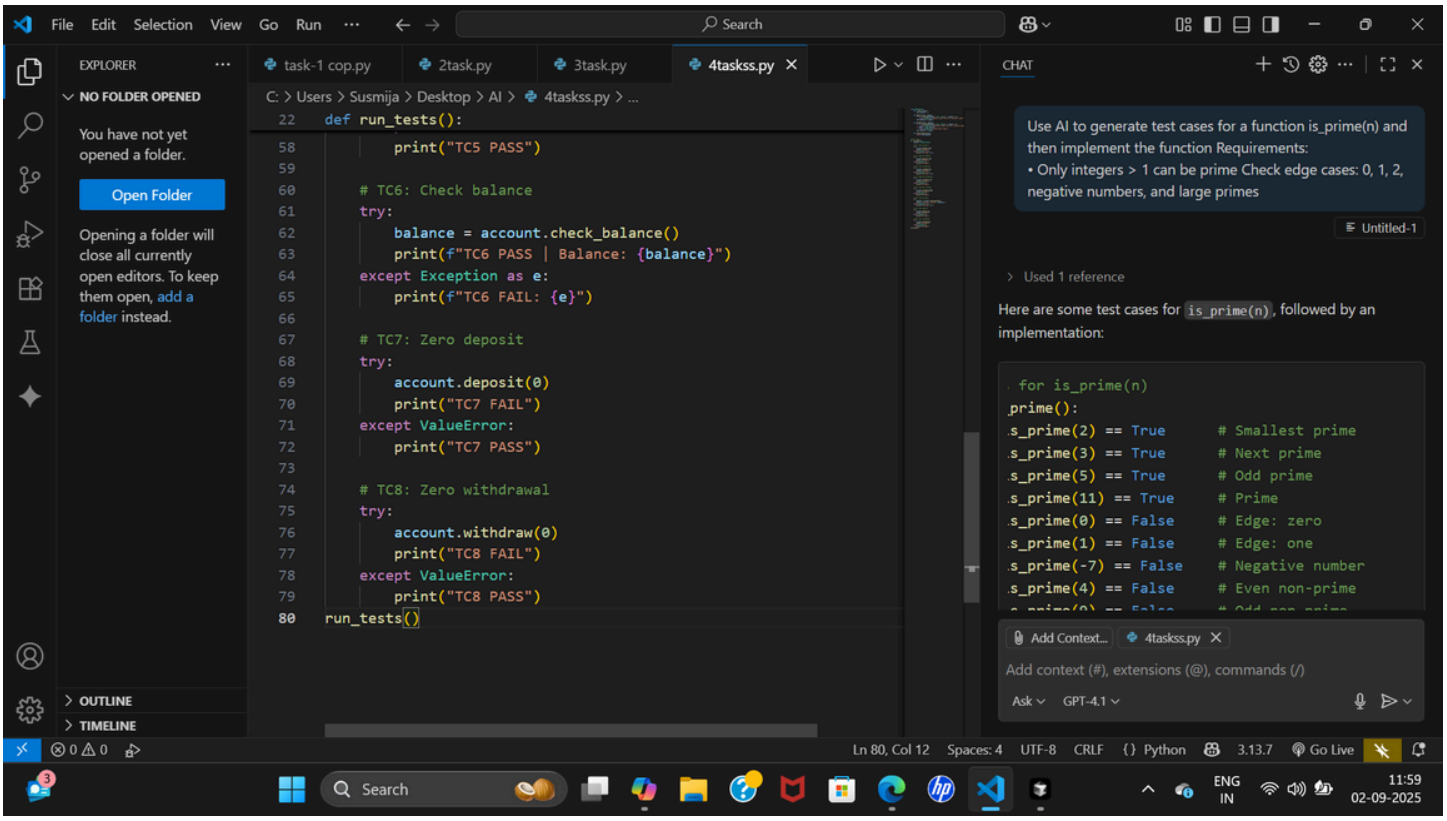
• AI-generated test suite with a robust class that handles all test cases.

CODE GENERATED:

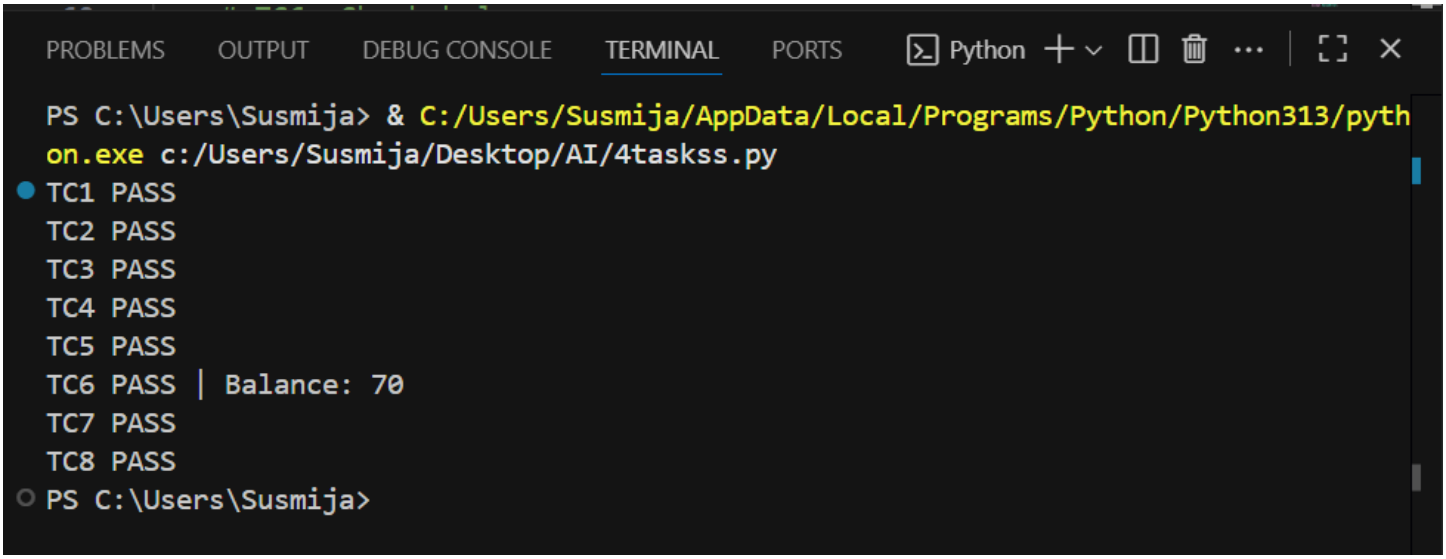








OUTPUT OF THE CODE:



OBSERVATIONS:

I have implemented a basic BankAccount class with the following features:

1. Constructor

- __init__(self) initializes the account with a balance of 0.

2. Deposit Method

- deposit(self, amount):
  - Only accepts positive int or float values.
  - Adds amount to balance.
  - Raises ValueError for invalid inputs.

3. Withdraw Method

- withdraw(self, amount):
  - Validates input type and ensures amount is > 0.
  - Checks for insufficient funds.
  - Deducts from balance if valid.

4. Balance Check

- check_balance(self):
  - Returns the current balance.

5. Test Cases

- run_tests() method performs basic functional tests:
  - TC1: Valid deposit test (100)
  - Uses try-except blocks to print pass/fail status.

TASK#5

PROMPT

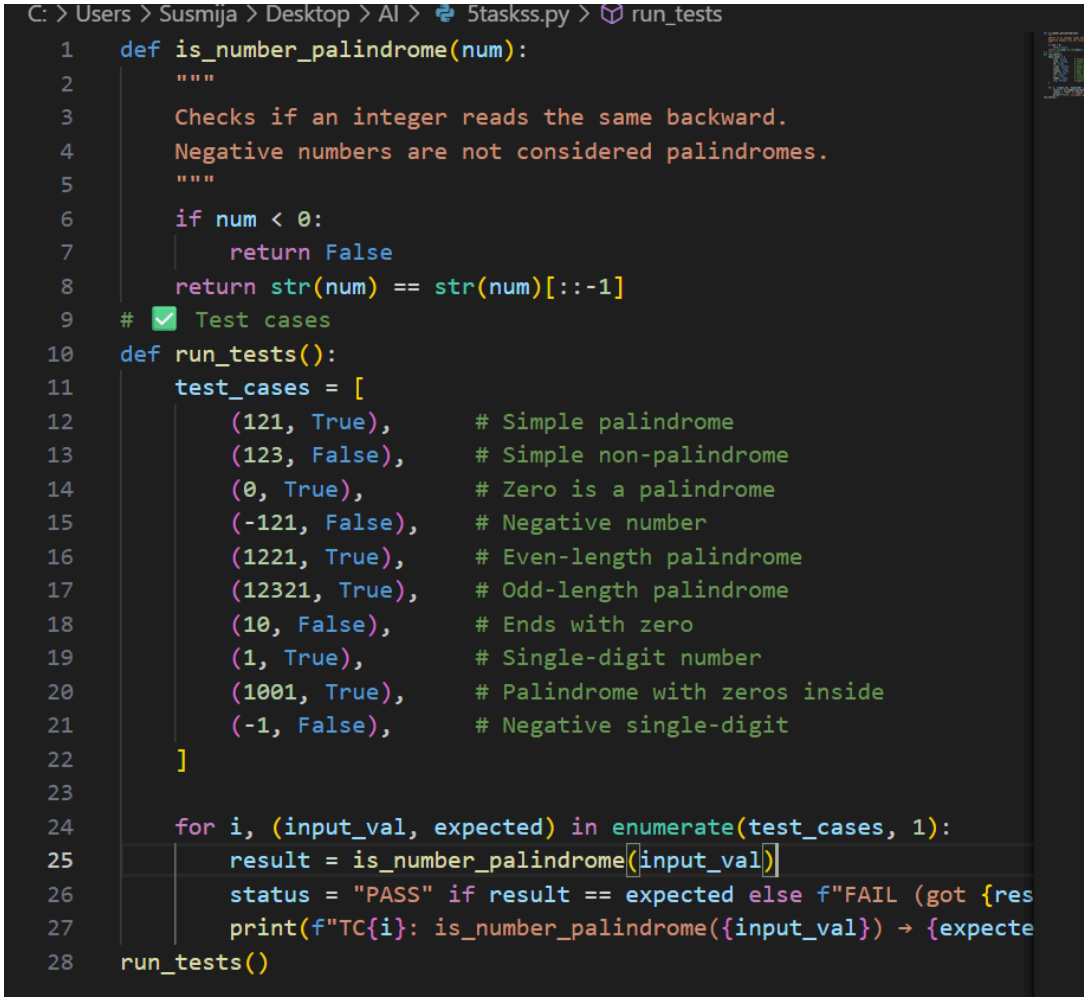
- Generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

  Examples:

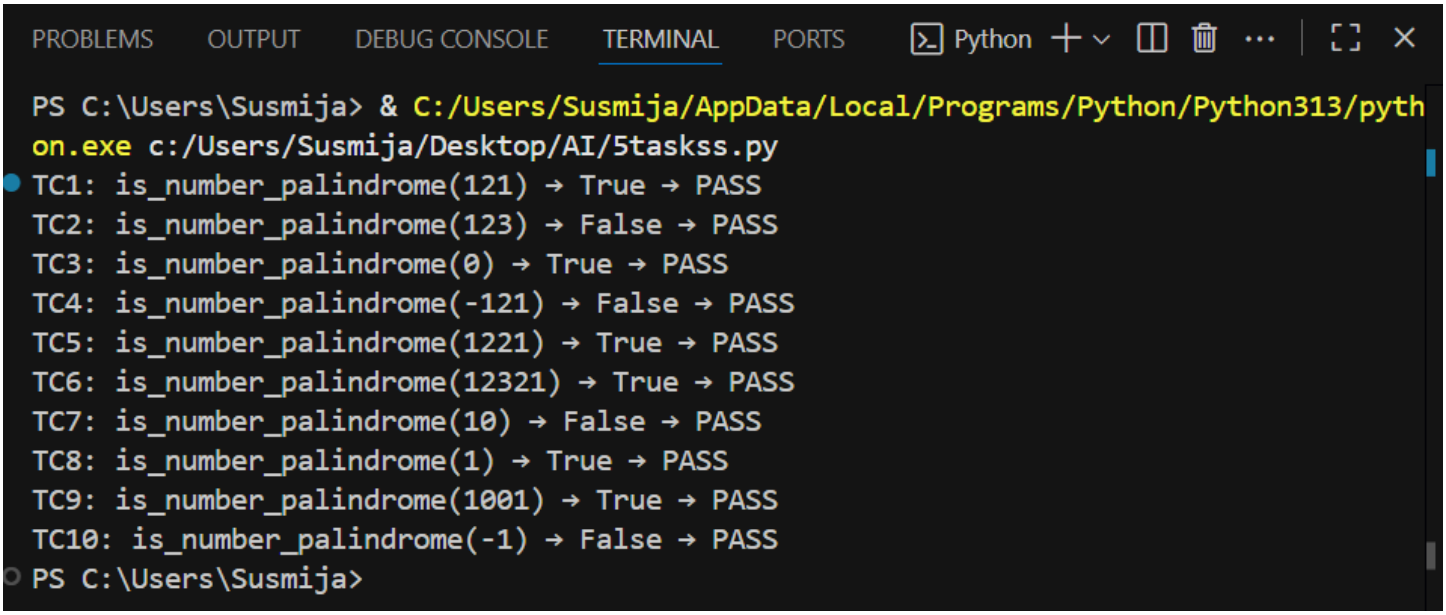
121 → True

123 → False
0, negative numbers → handled

CODE GENERATED:



OUTPUT OF THE CODE:



OBSERVATIONS:

1. **Negative numbers** are **not considered palindromes** because the minus sign (-) is not mirrored.

2. The function **converts the number to a string**, reverses it, and compares it with the original string.
3. If the reversed string matches the original, the function returns True; otherwise, it returns False.

➤ Function Logic:

- If the number is negative (num < 0), the function immediately returns False.
- Otherwise, the number is converted to a string, and the string is reversed using Python slicing ([::-1]).
- The original and reversed strings are compared:
    - If they are equal → return True (number is a palindrome).
    - If not → return False.

A list of test cases is created in the form of tuples: (input, expected_output). The function loops through each test case:

- Calls the palindrome-checking function with the test input.
- Compares the returned result with the expected value.
- Prints whether the test case passed or failed along with relevant information.