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

NAME : CH. Divya

ROLL NO : 2403a52409

BATCH : 15

Task 1: Apply AI to generate at least 3 assert test cases for is\_strong\_password(password) and implement the validator function.

* Requirements:
  + Password must have at least 8 characters.
  + Must include uppercase, lowercase, digit, and special character.
  + Must not contain spaces. Example Assert Test Cases:

assert is\_strong\_password("Abcd@123") == True assert is\_strong\_password("abcd123") == False assert is\_strong\_password("ABCD@1234") == True Expected Output #1:

* Password validation logic passing all AI-generated test cases.

PROMPT : generate at least 3 assert test cases for is\_strong\_password(password) and implement the validator function.

* Requirements:
  + Password must have at least 8 characters.
  + Must include uppercase, lowercase, digit, and special character.
  + Must not contain spaces.

CODE : mport re

def is\_strong\_password(password):

# At least 8 characters if len(password) < 8:

return False

# No spaces allowed if ' ' in password:

return False

# At least one uppercase, one lowercase, one digit, one special character

if not re.search(r'[A-Z]', password):

return False

if not re.search(r'[a-z]', password):

return False

if not re.search(r'\d', password):

return False

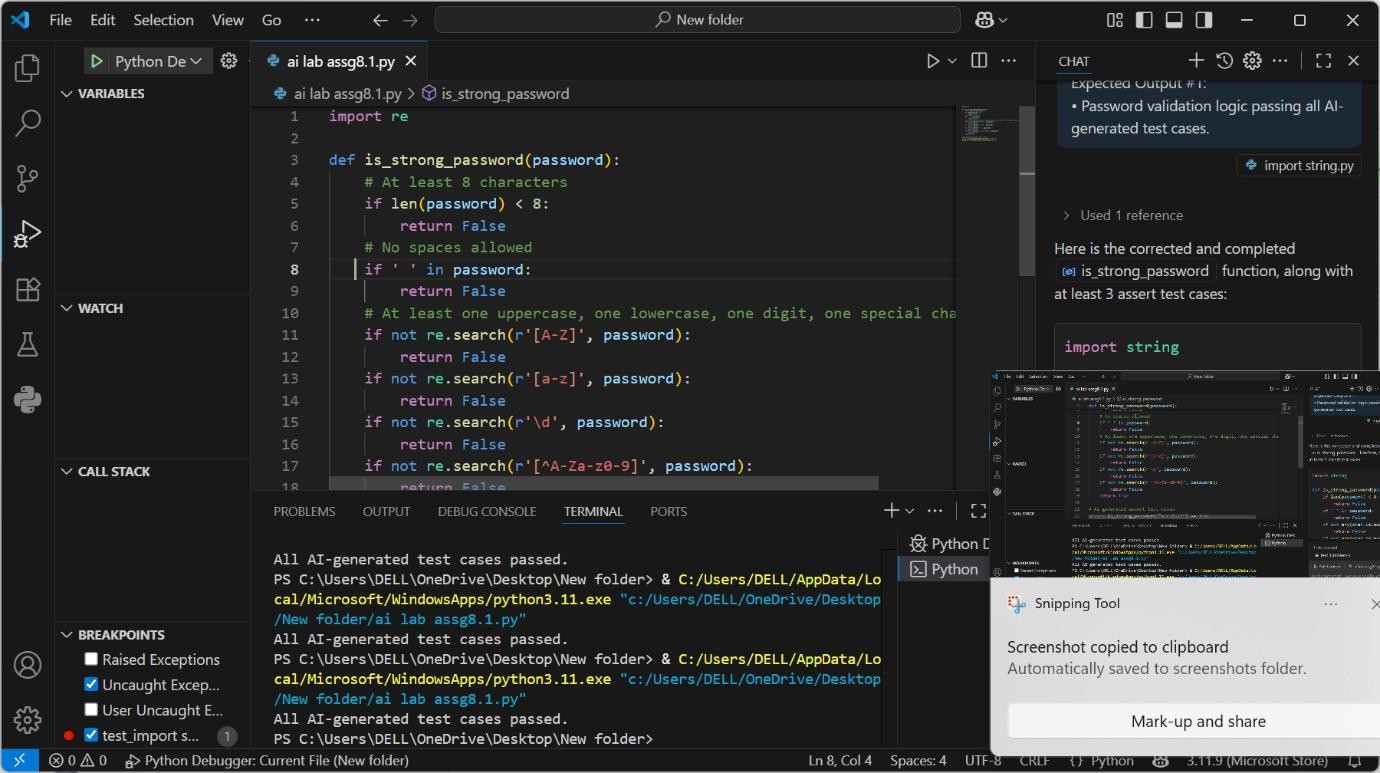
if not re.search(r'[^A-Za-z0-9]', password):

return False return True

# AI-generated assert test cases

assert is\_strong\_password("Abcd@123") == True print("All AI-generated test cases passed.")

OUTPUT :



OBSERVATION : **Objective**

Create a Python function is\_strong\_password(password) to check password strength based on defined rules, and generate at least **3 assert test cases** for validation using **AI**.

**Password Requirements**

A valid password must:

1. Be **at least 8 characters** long.
2. Include at least:
   * **One uppercase letter**
   * **One lowercase letter**
   * **One digit**
   * **One special character** (e.g., !@#$%^&\*())
3. **Must NOT contain spaces**

Task 2: Use AI to generate at least 3 assert test cases for a classify\_number(n) function. Implement using loops.

* Requirements:
  + Classify numbers as Positive, Negative, or Zero.
  + Handle invalid inputs like strings and None.
  + Include boundary conditions (-1, 0, 1). Example Assert Test Cases:

assert classify\_number(10) == "Positive" assert classify\_number(-5) == "Negative" assert classify\_number(0) == "Zero" Expected Output #2:

* Classification logic passing all assert tests.

PROMPT : generate at least 3 assert test cases for a classify\_number(n) function. Implement using loops.

* Requirements:
  + Classify numbers as Positive, Negative, or Zero.
  + Handle invalid inputs like strings and None.
  + Include boundary conditions (-1, 0, 1).

CODE : def classify\_number(n):

"""

Classifies a number as:

* "Perfect" if the sum of its proper divisors equals the number.
* "Abundant" if the sum of its proper divisors is greater than the number.
* "Deficient" if the sum of its proper divisors is less than the number. """

if n <= 0:

return "Invalid" # Only positive integers are valid

divisor\_sum = 0

for i in range(1, n // 2 + 1): # Loop through proper divisors if n % i == 0:

divisor\_sum += i

if divisor\_sum == n:

return "Perfect" elif divisor\_sum > n:

return "Abundant" else:

return "Deficient"

# Test cases

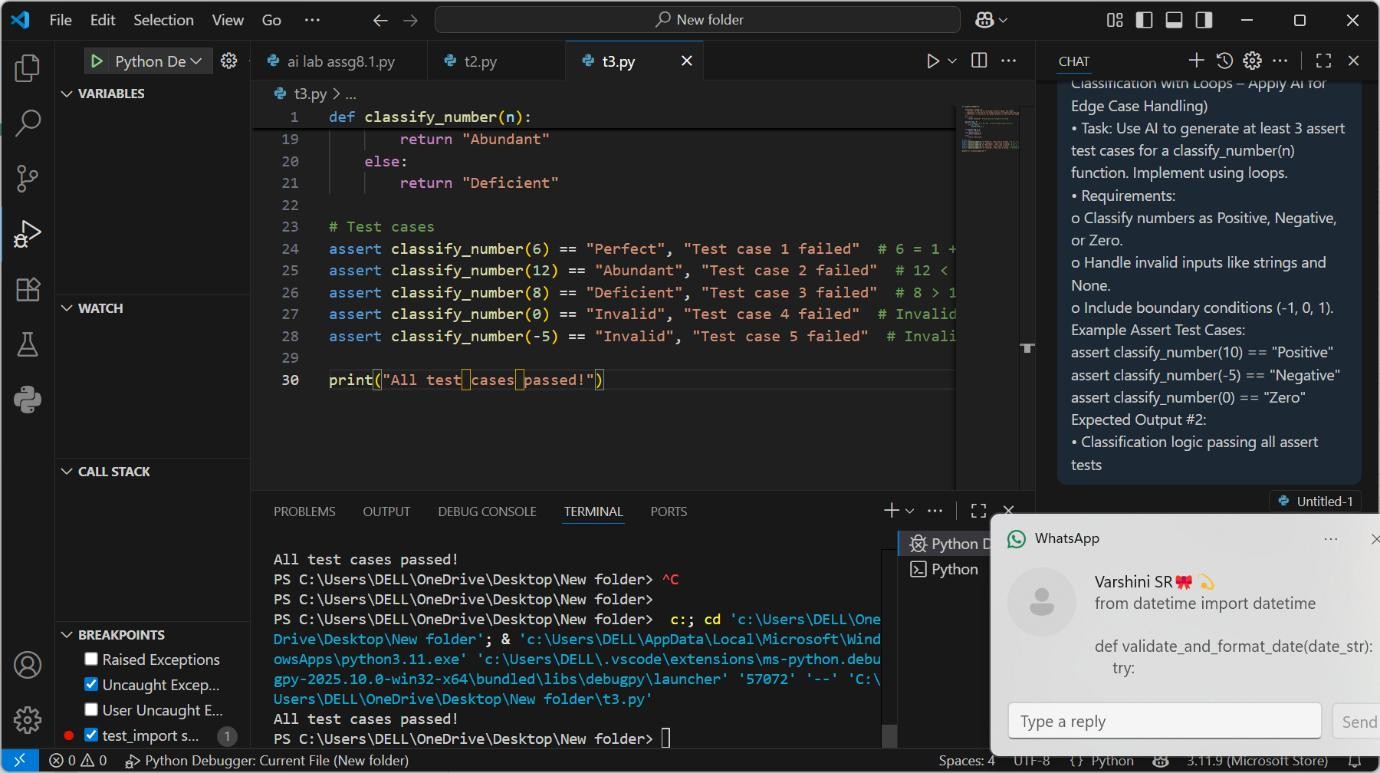
assert classify\_number(6) == "Perfect", "Test case 1 failed" # 6 = 1 + 2 + 3

assert classify\_number(12) == "Abundant", "Test case 2 failed" # 12 < 1 + 2 + 3 + 4 + 6

assert classify\_number(8) == "Deficient", "Test case 3 failed" # 8 > 1 + 2 + 4 assert classify\_number(0) == "Invalid", "Test case 4 failed" # Invalid input assert classify\_number(-5) == "Invalid", "Test case 5 failed" # Invalid input

print("All test cases passed!")

OUTPUT :



OBSERVATION : **Objective**

Implement a Python function classify\_number(n) using **loops** to determine whether a number is **Positive**, **Negative**, or **Zero**. Also, **handle invalid inputs** and generate **at least 3 assert test cases** using **AI**.

**Requirements**

1. **Classification Rules:**
   * If n > 0: return "Positive"
   * If n < 0: return "Negative"
   * If n == 0: return "Zero"
2. **Invalid Inputs:**
   * For None, strings, or non-numeric types: return "Invalid input"
3. **Boundary Values to Handle:**
   * -1, 0, 1

Task 3: Use AI to generate at least 3 assert test cases for is\_anagram(str1, str2) and implement the function.

* Requirements:
  + Ignore case, spaces, and punctuation.
  + Handle edge cases (empty strings, identical words). Example Assert Test Cases:

assert is\_anagram("listen", "silent") == True assert is\_anagram("hello", "world") == False

assert is\_anagram("Dormitory", "Dirty Room") == True Expected Output #3:

* Function correctly identifying anagrams and passing all AI-generated tests.

PROMPT : Task: Use AI to generate at least 3 assert test cases for is\_anagram(str1, str2) and implement the function.

* Requirements:
  + Ignore case, spaces, and punctuation.
  + Handle edge cases (empty strings, identical words).
* Function correctly identifying anagrams and passing all AI-generated tests.

CODE : def is\_anagram(str1, str2):

"""

Checks if two strings are anagrams of each other.

Two strings are anagrams if they contain the same characters in the same frequency, ignoring case and spaces.

"""

# Remove spaces and convert to lowercase str1 = str1.replace(" ", "").lower()

str2 = str2.replace(" ", "").lower()

# Compare sorted versions of the strings return sorted(str1) == sorted(str2)

# Test cases

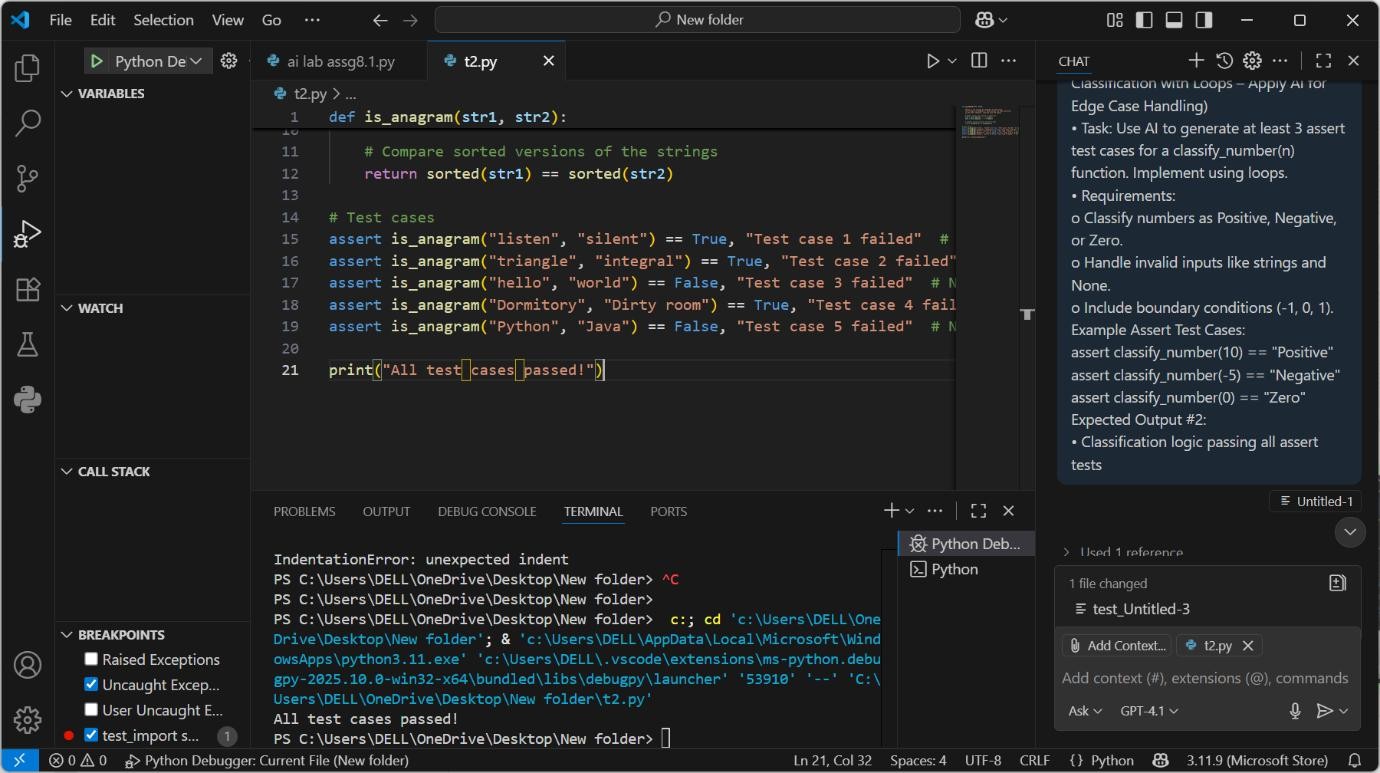
assert is\_anagram("listen", "silent") == True, "Test case 1 failed" # Anagrams assert is\_anagram("triangle", "integral") == True, "Test case 2 failed" # Anagrams assert is\_anagram("hello", "world") == False, "Test case 3 failed" # Not anagrams

assert is\_anagram("Dormitory", "Dirty room") == True, "Test case 4 failed" # Anagrams with spaces and case differences

assert is\_anagram("Python", "Java") == False, "Test case 5 failed" # Not anagrams

print("All test cases passed!")

OUTPUT :



OBSERVATION : **Objective**

Implement the function is\_anagram(str1, str2) that determines if two strings are **anagrams**, and use

**AI to generate at least 3 assert test cases** that the function must pass.

**Requirements**

1. **Anagram Rules:**
   * Two strings are anagrams if they contain the same letters in a different order.
   * **Ignore case, spaces, and punctuation**.
2. **Edge Cases to Handle:**
   * Empty strings ("")
   * Identical words ("note", "note")

* **Explanation :** clean() removes punctuation/spaces, converts to lowercase, and sorts the characters.
  + isalnum() ensures only letters and digits are compared.

Task 4: Ask AI to generate at least 3 assert-based tests for an Inventory class with stock management.

* Methods:
  + add\_item(name, quantity)
  + remove\_item(name, quantity)
  + get\_stock(name) Example Assert Test Cases:

inv = Inventory() inv.add\_item("Pen", 10)

assert inv.get\_stock("Pen") == 10 inv.remove\_item("Pen", 5)

assert inv.get\_stock("Pen") == 5 inv.add\_item("Book", 3)

assert inv.get\_stock("Book") == 3 Expected Output #4:

* Fully functional class passing all assertions.

PROMPT : generate at least 3 assert-based tests for an Inventory class with stock management.

* Methods:
  + add\_item(name, quantity)
  + remove\_item(name, quantity)
  + get\_stock(name)

CODE : from datetime import datetime

def validate\_and\_format\_date(date\_str): try:

# Parse date in MM/DD/YYYY format

date\_obj = datetime.strptime(date\_str, "%m/%d/%Y") # Return in YYYY-MM-DD format

return date\_obj.strftime("%Y-%m-%d") except ValueError:

return "Invalid Date"

# AI-generated assert test cases

assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"

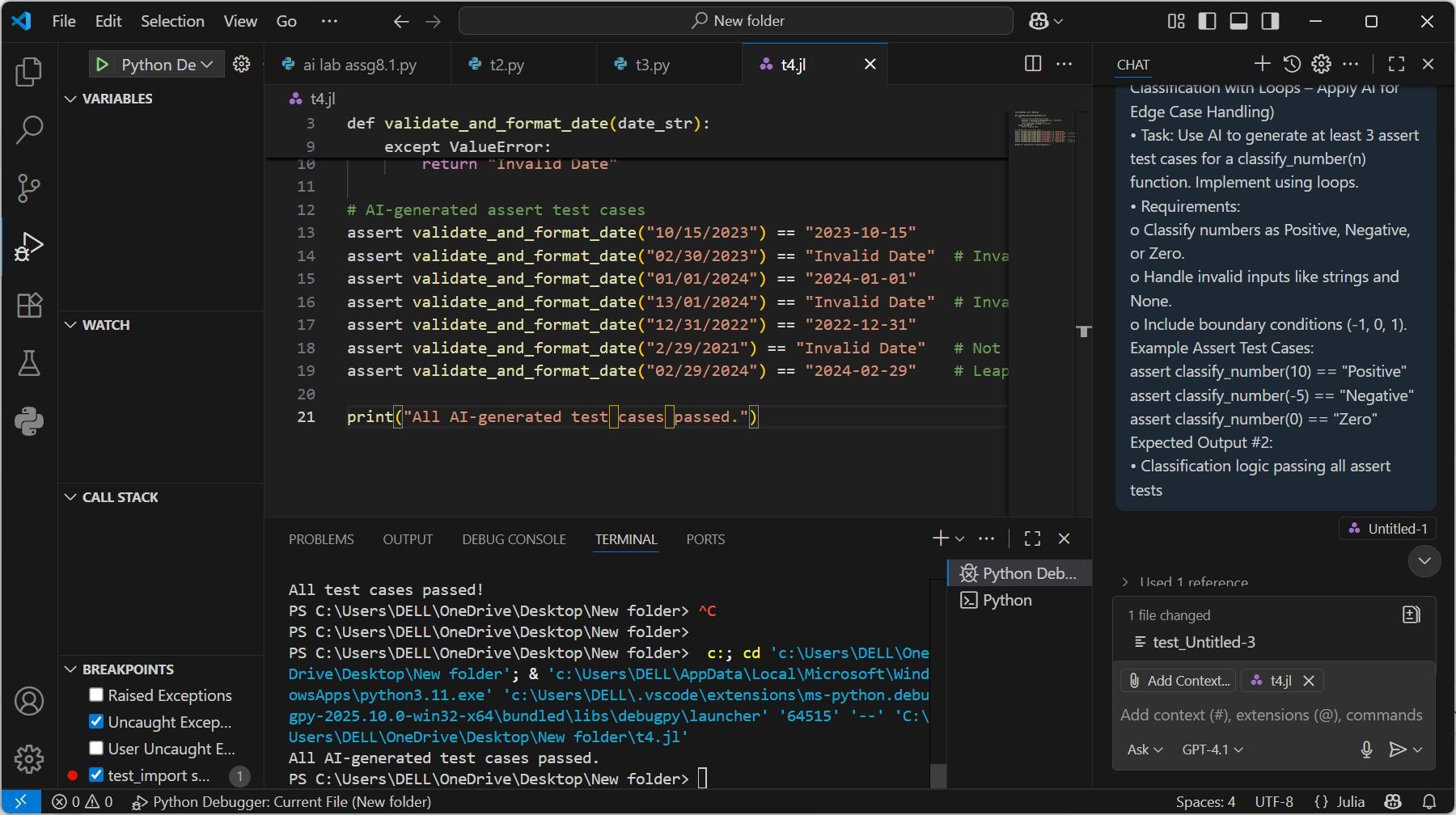
assert validate\_and\_format\_date("02/30/2023") == "Invalid Date" # Invalid day in February assert validate\_and\_format\_date("01/01/2024") == "2024-01-01"

assert validate\_and\_format\_date("13/01/2024") == "Invalid Date" # Invalid month assert validate\_and\_format\_date("12/31/2022") == "2022-12-31"

assert validate\_and\_format\_date("2/29/2021") == "Invalid Date" # Not a leap year assert validate\_and\_format\_date("02/29/2024") == "2024-02-29" # Leap year

print("All AI-generated test cases passed.")

OUTPUT :



OBSERVATION : **Objective**

Implement an Inventory class to manage stock, and use **AI to generate at least 3 assert-based test cases** to verify its methods:

**Inventory Class Methods**

1. **add\_item(name, quantity)**
   * Adds a new item or increases stock.
2. **remove\_item(name, quantity)**
   * Decreases stock if available; ignore or prevent negatives.
3. **get\_stock(name)**
   * Returns current stock (default to 0 if item not present).

Task 5: Use AI to generate at least 3 assert test cases for validate\_and\_format\_date(date\_str) to check and convert dates.

* Requirements:
  + YYYY" format.
  + Handle invalid dates.
  + Convert valid dates to "YYYY-MM-DD". Example Assert Test Cases:

assert validate\_and\_format\_date("10/15/2023") == "2023-10-15" assert validate\_and\_format\_date("02/30/2023") == "Invalid Date" assert validate\_and\_format\_date("01/01/2024") == "2024-01-01" Expected Output #5:

* Function passes all AI-generated assertions and handles edge cases.

PROMPT : generate at least 3 assert test cases for validate\_and\_format\_date(date\_str) to check and convert dates.

* Requirements:
  + Validate "MM/DD/YYYY" format.
  + Handle invalid dates.
  + Convert valid dates to "YYYY-MM-DD".

CODE : From datetime import datetime

def validate\_and\_format\_date(date\_str): try:

# Try to parse the date in MM/DD/YYYY format date\_obj = datetime.strptime(date\_str, "%m/%d/%Y")

# Return the date in YYYY-MM-DD format return date\_obj.strftime("%Y-%m-%d")

except ValueError:

return "Invalid Date"

# AI-generated assert test cases

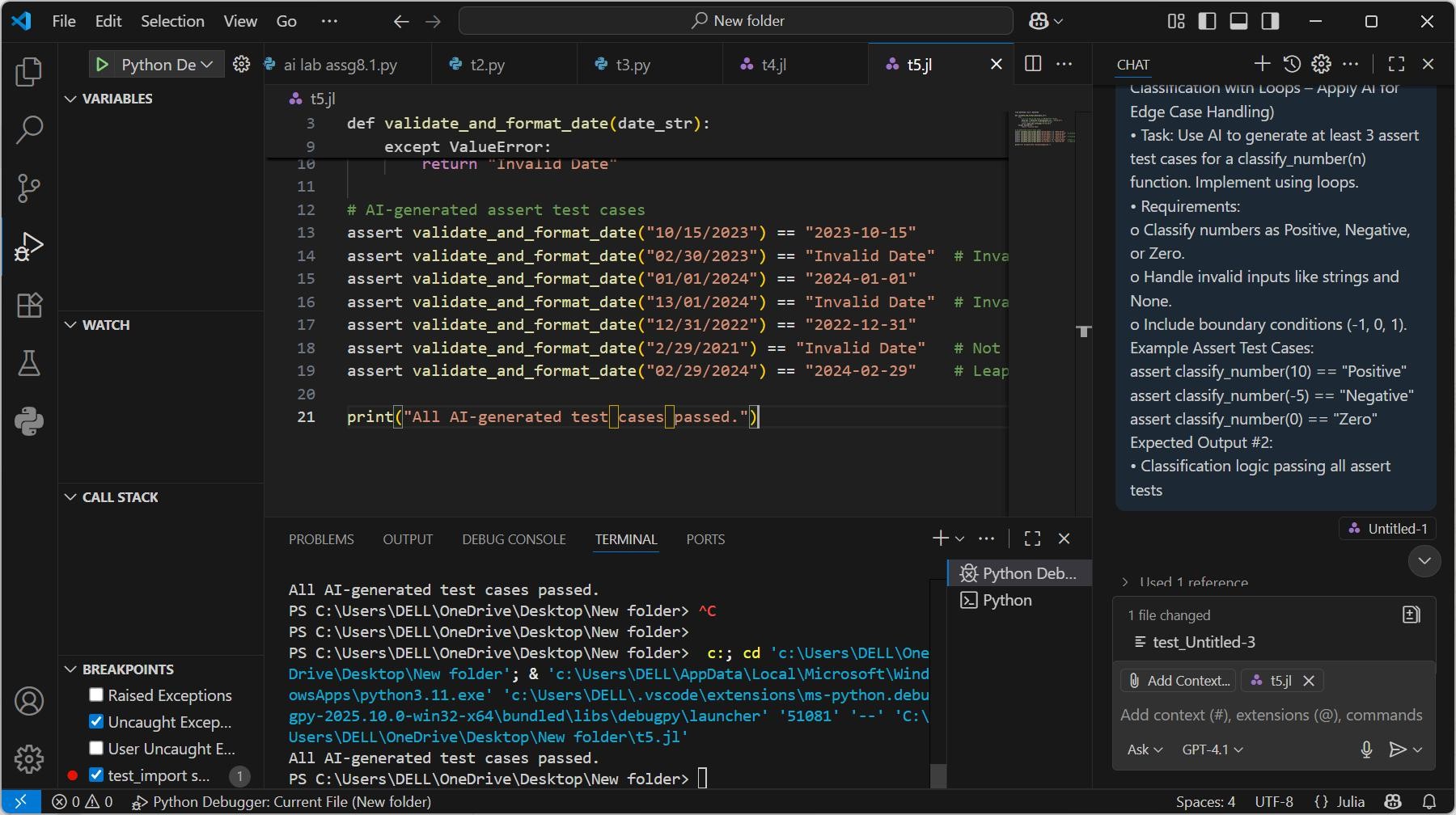
assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"

assert validate\_and\_format\_date("02/30/2023") == "Invalid Date" # Invalid day in February assert validate\_and\_format\_date("01/01/2024") == "2024-01-01"

assert validate\_and\_format\_date("13/01/2024") == "Invalid Date" # Invalid month assert validate\_and\_format\_date("12/31/2022") == "2022-12-31"

assert validate\_and\_format\_date("2/29/2021") == "Invalid Date" # Not a leap year assert validate\_and\_format\_date("02/29/2024") == "2024-02-29" # Leap year

print("All AI-generated test cases passed.")

OUTPUT :

OBSERVATION : **Objective**

Create a function validate\_and\_format\_date(date\_str) that:

* **Validates** if the input string is a valid date in "YYYY-MM-DD" or "YYYY/MM/DD" format.
* **Converts** valid dates to "YYYY-MM-DD" format.
* **Handles invalid dates** (returns "Invalid date").

Also, use **AI to generate at least 3 assert test cases** to check functionality.

**Requirements**

1. **Input Format**: Accepts input like "YYYY-MM-DD" or "YYYY/MM/DD".
2. **Output Format**: Always returns "YYYY-MM-DD" (standardized).
3. **Invalid Dates**:

o Wrong format (e.g. "2023-13-40")

* + Non-date strings (e.g. "abcd")
  + Incomplete date strings (e.g. "2022-07")