

AI Assisted Coding Lab ASS-8.1

CH. VENUGOPAL-2303A51844-B13

Task Description #1 (Password Strength Validator – Apply AI in Security Context)

- Task: Apply AI to generate at least 3 assert test cases for `is_strong_password(password)` and implement the validator function.

PROMPT: Write a Python function called `is_strong_password(password)` to validate password strength.

Conditions:

- Password must have at least 8 characters
- Must contain uppercase, lowercase, digit, and special character
- Must not contain spaces

Generate 4 test cases and show their outputs using:

1. General test cases using `print()`
2. Assert test cases
3. Pytest test cases
4. Unit test cases

Display the output for each test type.

Keep the code simple and clear.

```
[m]
✓ On
import re

def is_strong_password(password):
    """
    Validates password strength based on the following conditions:
    - At least 8 characters
    - Contains uppercase, lowercase, digit, and special character
    - No spaces
    """

    # Condition 1: At least 8 characters
    if len(password) < 8:
        return False

    # Condition 2: Must contain uppercase, lowercase, digit, and special character
    has_uppercase = re.search(r'[A-Z]', password)
    has_lowercase = re.search(r'[a-z]', password)
    has_digit = re.search(r'[0-9]', password)
    # Escaped single quote in regex: \'
    has_special_char = re.search(r'!@#$%^&*()_+!~\[\]\{\};\'":.,<.>/?~', password)

    if not (has_uppercase and has_lowercase and has_digit and has_special_char):
        return False

    # Condition 3: Must not contain spaces
    if ' ' in password:
        return False

    return True
```

1. General Test Cases with print()

```
[m]
✓ On
print("--- General Test Cases (using print()) ---")

# Test Case 1: Strong password
password_1 = 'Strong@ss1'
print(f'{password_1} is strong: {is_strong_password(password_1)} (Expected: True)')

# Test Case 2: Too short
password_2 = 'Short!'
print(f'{password_2} is strong: {is_strong_password(password_2)} (Expected: False)')

# Test Case 3: Missing digit
password_3 = 'NoDigit@ss'
print(f'{password_3} is strong: {is_strong_password(password_3)} (Expected: False)')

# Test Case 4: Contains space
password_4 = 'Password With Space!'
print(f'{password_4} is strong: {is_strong_password(password_4)} (Expected: False)')

# Test Case 5: Missing special character
password_5 = 'NoSpecial123'
print(f'{password_5} is strong: {is_strong_password(password_5)} (Expected: False)')

# Test Case 6: Missing uppercase
password_6 = 'weak@ss1'
print(f'{password_6} is strong: {is_strong_password(password_6)} (Expected: False)')

# Test Case 7: Missing lowercase
password_7 = 'WEAK@SS1'
print(f'{password_7} is strong: {is_strong_password(password_7)} (Expected: False)')

--- General Test Cases (using print()) ---
'Strong@ss1' is strong: True (Expected: True)
'Short!' is strong: False (Expected: False)
'NoDigit@ss' is strong: False (Expected: False)
'Password With Space!' is strong: False (Expected: False)
'NoSpecial123' is strong: False (Expected: False)
'weak@ss1' is strong: False (Expected: False)
'WEAK@SS1' is strong: False (Expected: False)
```

2. Assert Test Cases

```
print('\n--- Assert Test Cases ---')

try:
    assert is_strong_password('StrongP@ss1') is True, "Test Case 1 Failed: Strong password"
    assert is_strong_password('Short1!') is False, "Test Case 2 Failed: Too short"
    assert is_strong_password('NoDigitP@ss') is False, "Test Case 3 Failed: Missing digit"
    assert is_strong_password('Password With Space1@') is False, "Test Case 4 Failed: Contains space"
    assert is_strong_password('NoSpecial123') is False, "Test Case 5 Failed: Missing special character"
    assert is_strong_password('weakp@ss1') is False, "Test Case 6 Failed: Missing uppercase"
    assert is_strong_password('WEAKP@SS1') is False, "Test Case 7 Failed: Missing lowercase"
    print("All assert tests passed!")
except AssertionError as e:
    print(f"Assert test failed: {e}")
```

```
---
--- Assert Test Cases ---
All assert tests passed!
```

3. Pytest Test Cases

To run these `pytest` cases, save the following code into a file named `test_password.py` (e.g., in the `/content/` directory). Then, you'll need to install `pytest` and run it from a terminal or another code cell.

```
%%writefile test_password.py
import pytest
import re

def is_strong_password(password):
    # Re-define the function or ensure it's imported correctly for pytest
    # For this example, re-defining it to make the test file self-contained
    if len(password) < 8:
        return False
    has_uppercase = re.search(r'[A-Z]', password)
    has_lowercase = re.search(r'[a-z]', password)
    has_digit = re.search(r'[0-9]', password)
    has_special_char = re.search(r'[@$%^&*()_+!~\-\[\]\{\};\':"\".,<.>/?~]', password)
    if not (has_uppercase and has_lowercase and has_digit and has_special_char):
        return False
    if ' ' in password:
        return False
    return True

def test_strong_password():
    assert is_strong_password('StrongP@ss1') == True

def test_password_too_short():
    assert is_strong_password('Short1!') == False

def test_password_missing_digit():
    assert is_strong_password('NoDigitP@ss') == False

def test_password_with_space():
    assert is_strong_password('Password With Space1@') == False

def test_password_missing_special_char():
    assert is_strong_password('NoSpecial123') == False

def test_password_missing_uppercase():
    assert is_strong_password('weakp@ss1') == False

def test_password_missing_lowercase():
    assert is_strong_password('WEAKP@SS1') == False
```

```
--- Overwriting test_password.py
```

```
bash$
# install pytest if you haven't already
$pip install pytest

# run pytest
print('p--- Pytest test cases output ---')
$pytest test_password.py

Requirement already satisfied: pytest in /usr/local/lib/python3.12/dist-packages (8.6.2)
Requirement already satisfied: iniconfig==1 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.3.0)
Requirement already satisfied: packaging==20 in /usr/local/lib/python3.12/dist-packages (from pytest) (20.0)
Requirement already satisfied: pluggy<2,>=1.6 in /usr/local/lib/python3.12/dist-packages (from pytest) (1.6.0)
Requirement already satisfied: pygments==2.7.2 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.19.2)

--- Pytest test cases output ---
===== test session starts =====
platform linux -- Python 3.12.12, pytest-8.6.2, pluggy-1.6.0
rootdir: /content
plugins: anyio-4.12.1, langsmith-0.6.0, typeguard-4.6.0
collected 7 items

test_password.py ..... [100%]

===== 7 passed in 0.02s =====

Unittest Test Cases

older to 'pytest', save the following code into a file named 'test_password_unittest.py'. Then, you can run it from a code cell using
there's '-m unittest' command.

$writefile test_password_unittest.py
import unittest
import re

def is_strong_password(password):
    # Re-define the function or ensure it's imported correctly for unittest
    # For this example, re-defining it to make the test file self-contained
    if len(password) < 8:
        return False
    has_uppercase = re.search(r'[A-Z]', password)
    has_lowercase = re.search(r'[a-z]', password)
    has_digit = re.search(r'[0-9]', password)
    has_special_char = re.search(r'[!@#$%^&*()_+=-~\|;\|:|'|',.<?/?-]', password)
    if not (has_uppercase and has_lowercase and has_digit and has_special_char):
        return False
    if ' ' in password:
        return False
    return True

class TestStrongPassword(unittest.TestCase):

    def test_strong_password(self):
        self.assertTrue(is_strong_password('StrongP@ss1'))

    def test_password_too_short(self):
        self.assertFalse(is_strong_password('Short1'))

    def test_password_missing_digit(self):
        self.assertFalse(is_strong_password('NoDigit@ss1'))

    def test_password_with_space(self):
        self.assertFalse(is_strong_password('Password with space1'))

    def test_password_missing_special_char(self):
        self.assertFalse(is_strong_password('Nosp@cial123'))

    def test_password_missing_uppercase(self):
        self.assertFalse(is_strong_password('nosp@ss1'))

    def test_password_missing_lowercase(self):
        self.assertFalse(is_strong_password('SH0RP@ss1'))

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

Overwriting test_password_unittest.py

print('p--- Unittest test cases output ---')
$python -m unittest test_password_unittest.py

--- Unittest test cases output ---
-----
Run 7 tests in 0.000s

OK
```

Task Description #2 (Number Classification with Loops – Apply AI for Edge Case Handling)

• Task: Use AI to generate at least 3 assert test cases for a `classify_number(n)` function. Implement using loops.

PROMPT: Write a Python function `classify_number(n)`.

Conditions:

- Return Positive, Negative, or Zero
- Handle invalid inputs like string and None

- Use loop

Generate 4 test cases and show outputs using:

1. General test cases

2. Assert test cases

3. Pytest test cases

4. Unit test cases

```
def classify_number(n):
    # Condition 2: Check if n is not a numeric type
    if not isinstance(n, (int, float)):
        return 'Invalid Input'

    # Condition 3: Classify numeric types
    if n > 0:
        return 'Positive'
    elif n < 0:
        return 'Negative'
    else:
        return 'Zero'

print('--- classify_number Test Cases ---')

# Test Case 1: Positive number
result_1 = classify_number(10)
print(f"classify_number(10): {result_1} (Expected: 'Positive')")
assert result_1 == 'Positive', "Test Case 1 Failed: Positive number"

# Test Case 2: Negative number
result_2 = classify_number(-5)
print(f"classify_number(-5): {result_2} (Expected: 'Negative')")
assert result_2 == 'Negative', "Test Case 2 Failed: Negative number"

# Test Case 3: Zero
result_3 = classify_number(0)
print(f"classify_number(0): {result_3} (Expected: 'Zero')")
assert result_3 == 'Zero', "Test Case 3 Failed: Zero"

# Test Case 4: Float positive
result_4 = classify_number(3.14)
print(f"classify_number(3.14): {result_4} (Expected: 'Positive')")
assert result_4 == 'Positive', "Test Case 4 Failed: Float positive"

# Test Case 5: Float negative
result_5 = classify_number(-0.001)
print(f"classify_number(-0.001): {result_5} (Expected: 'Negative')")
assert result_5 == 'Negative', "Test Case 5 Failed: Float negative"

# Test Case 6: Invalid input (string)
result_6 = classify_number('hello')
print(f"classify_number('hello'): {result_6} (Expected: 'Invalid Input')")
assert result_6 == 'Invalid Input', "Test Case 6 Failed: Invalid input (string)"

# Test Case 7: Invalid input (list)
result_7 = classify_number([1, 2])
print(f"classify_number([1, 2]): {result_7} (Expected: 'Invalid Input')")
assert result_7 == 'Invalid Input', "Test Case 7 Failed: Invalid input (list)"

# Test Case 8: Invalid input (None)
result_8 = classify_number(None)
print(f"classify_number(None): {result_8} (Expected: 'Invalid Input')")
assert result_8 == 'Invalid Input', "Test Case 8 Failed: Invalid input (None)"

print("All classify_number tests passed!")

--- classify_number Test Cases ---
classify_number(10): Positive (Expected: 'Positive')
classify_number(-5): Negative (Expected: 'Negative')
classify_number(0): Zero (Expected: 'Zero')
classify_number(3.14): Positive (Expected: 'Positive')
classify_number(-0.001): Negative (Expected: 'Negative')
classify_number('hello'): Invalid Input (Expected: 'Invalid Input')
classify_number([1, 2]): Invalid Input (Expected: 'Invalid Input')
classify_number(None): Invalid Input (Expected: 'Invalid Input')
All classify_number tests passed!
```

```

❏ %%writefile test_classify_number.py
import pytest

def classify_number(n):
    if not isinstance(n, (int, float)):
        return 'Invalid Input'
    if n > 0:
        return 'Positive'
    elif n < 0:
        return 'Negative'
    else:
        return 'Zero'

def test_positive_number():
    assert classify_number(10) == 'Positive'

def test_negative_number():
    assert classify_number(-5) == 'Negative'

def test_zero():
    assert classify_number(0) == 'Zero'

def test_float_positive():
    assert classify_number(3.14) == 'Positive'

def test_float_negative():
    assert classify_number(-0.001) == 'Negative'

def test_invalid_string_input():
    assert classify_number('hello') == 'Invalid Input'

def test_invalid_list_input():
    assert classify_number([1, 2]) == 'Invalid Input'

def test_invalid_none_input():
    assert classify_number(None) == 'Invalid Input'

```

... Writing test_classify_number.py

Reasoning: To execute the pytest tests, I need to first install `pytest` and then run the test file using the `!pytest` command.

```

# Install pytest if you haven't already
%pip install pytest

# Run pytest
print('\n--- Pytest Test Cases Output ---')
!pytest test_classify_number.py

```

```

Requirement already satisfied: pytest in /usr/local/lib/python3.12/dist-packages (8.4.2)
Requirement already satisfied: iniconfig>=1 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.3.0)
Requirement already satisfied: packaging>=20 in /usr/local/lib/python3.12/dist-packages (from pytest) (26.0)
Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.12/dist-packages (from pytest) (1.6.0)
Requirement already satisfied: pygments>=2.7.2 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.19.2)

--- Pytest Test Cases Output ---
===== test session starts =====
platform linux -- Python 3.12.12, pytest-8.4.2, pluggy-1.6.0
rootdir: /content
plugins: anyio-4.12.1, langsmith-0.6.9, typeguard-4.4.4
collected 8 items

test_classify_number.py ..... [100%]

===== 8 passed in 0.03s =====

```

```
[32] %%writefile test_classify_number_unittest.py
✓ Os import unittest

def classify_number(n):
    if not isinstance(n, (int, float)):
        return 'Invalid Input'
    if n > 0:
        return 'Positive'
    elif n < 0:
        return 'Negative'
    else:
        return 'Zero'

class TestClassifyNumber(unittest.TestCase):

    def test_positive_number(self):
        self.assertEqual(classify_number(10), 'Positive')

    def test_negative_number(self):
        self.assertEqual(classify_number(-5), 'Negative')

    def test_zero(self):
        self.assertEqual(classify_number(0), 'Zero')

    def test_float_positive(self):
        self.assertEqual(classify_number(3.14), 'Positive')

    def test_float_negative(self):
        self.assertEqual(classify_number(-0.001), 'Negative')

    def test_invalid_string_input(self):
        self.assertEqual(classify_number('hello'), 'Invalid Input')

    def test_invalid_list_input(self):
        self.assertEqual(classify_number([1, 2]), 'Invalid Input')

    def test_invalid_none_input(self):
        self.assertEqual(classify_number(None), 'Invalid Input')

    if __name__ == '__main__':
        unittest.main(argv=['first-arg-is-ignored'], exit=False)

--- Writing test_classify_number_unittest.py

[33] print('\n--- Unittest Test Cases Output ---')
✓ Os !python -m unittest test_classify_number_unittest.py

---
--- Unittest Test Cases Output ---
-----
Ran 8 tests in 0.000s

OK
```

Task Description #3 (Anagram Checker – Apply AI for String

Analysis)

- Task: Use AI to generate at least 3 assert test cases for

is_anagram(str1, str2) and implement the function.

- Requirements:

o Ignore case, spaces, and punctuation.

o Handle edge cases (empty strings, identical words).

PROMPT: Write a Python function `is_anagram(str1, str2)`.

Conditions:

- Ignore case, spaces, punctuation
- Return True or False

Generate 4 test cases and show outputs using:

General, Assert, Pytest, Unit Test


```

def is_anagram(str1, str2):
    # 2. Convert both strings to lowercase
    str1_lower = str1.lower()
    str2_lower = str2.lower()

    # 3. Remove all non-alphabetic characters
    # Helper function to filter out non-alphabetic characters
    def filter_alphabetic(s):
        return ''.join(char for char in s if char.isalpha())

    processed_str1 = filter_alphabetic(str1_lower)
    processed_str2 = filter_alphabetic(str2_lower)

    # 4. Sort the characters of the processed strings
    sorted_str1 = sorted(processed_str1)
    sorted_str2 = sorted(processed_str2)

    # 5. Compare the sorted lists of characters
    return sorted_str1 == sorted_str2

print('--- is_anagram General Test Cases (using print()) ---')

# Test Case 1: Simple anagrams
str1_1 = 'listen'
str2_1 = 'silent'
print(f'{str1_1} and {str2_1} are anagrams: {is_anagram(str1_1, str2_1)} (Expected: True)')

# Test Case 2: Anagrams with different cases
str1_2 = 'Debit Card'
str2_2 = 'Bad Credit'
print(f'{str1_2} and {str2_2} are anagrams: {is_anagram(str1_2, str2_2)} (Expected: True)')

# Test Case 3: Anagrams with punctuation and spaces
str1_3 = 'A gentleman.'
str2_3 = 'Elegant man'
print(f'{str1_3} and {str2_3} are anagrams: {is_anagram(str1_3, str2_3)} (Expected: True)')

# Test Case 4: Not anagrams (different length)
str1_4 = 'hello'
str2_4 = 'world'
print(f'{str1_4} and {str2_4} are anagrams: {is_anagram(str1_4, str2_4)} (Expected: False)')

# Test Case 5: Not anagrams (same length, different characters)
str1_5 = 'apple'
str2_5 = 'apply'
print(f'{str1_5} and {str2_5} are anagrams: {is_anagram(str1_5, str2_5)} (Expected: False)')

# Test Case 6: Anagrams with numbers and special characters (should be ignored)
str1_6 = 'earth 123'
str2_6 = 'heart 456'
print(f'{str1_6} and {str2_6} are anagrams: {is_anagram(str1_6, str2_6)} (Expected: True)')

# Test Case 7: Empty strings (considered anagrams of each other)
str1_7 = ''
str2_7 = ''
print(f'{str1_7} and {str2_7} are anagrams: {is_anagram(str1_7, str2_7)} (Expected: True)')

# Test Case 8: One empty, one non-empty
str1_8 = ''
str2_8 = 'a'
print(f'{str1_8} and {str2_8} are anagrams: {is_anagram(str1_8, str2_8)} (Expected: False)')

--- is_anagram General Test Cases (using print()) ---
'listen' and 'silent' are anagrams: True (Expected: True)
'Debit Card' and 'Bad Credit' are anagrams: True (Expected: True)
'A gentleman.' and 'Elegant man' are anagrams: True (Expected: True)
'hello' and 'world' are anagrams: False (Expected: False)
'apple' and 'apply' are anagrams: False (Expected: False)
'earth 123' and 'heart 456' are anagrams: True (Expected: True)
'' and '' are anagrams: True (Expected: True)
'' and 'a' are anagrams: False (Expected: False)

```

```

print('\n--- is_anagram Assert Test Cases ---')

try:
    assert is_anagram('listen', 'silent') is True, "Test Case 1 Failed: Simple anagrams"
    assert is_anagram('Debit Card', 'Bad Credit') is True, "Test Case 2 Failed: Anagrams with different cases"
    assert is_anagram('A gentleman.', 'Elegant man') is True, "Test Case 3 Failed: Anagrams with punctuation and spaces"
    assert is_anagram('hello', 'world') is False, "Test Case 4 Failed: Not anagrams (different length)"
    assert is_anagram('apple', 'apply') is False, "Test Case 5 Failed: Not anagrams (same length, different characters)"
    assert is_anagram('earth 123', 'heart 456') is True, "Test Case 6 Failed: Anagrams with numbers and special characters (should be ignored)"
    assert is_anagram('', '') is True, "Test Case 7 Failed: Empty strings"
    assert is_anagram('', 'a') is False, "Test Case 8 Failed: One empty, one non-empty"
    print("All assert tests passed!")
except AssertionError as e:
    print(f"Assert test failed: {e}")

--- is_anagram Assert Test Cases ---
All assert tests passed!

```



Writefile test_anagram.py

```

import pytest

def is_anagram(str1, str2):
    str1_lower = str1.lower()
    str2_lower = str2.lower()

    def filter_alphabetic(s):
        return ''.join(char for char in s if char.isalpha())

    processed_str1 = filter_alphabetic(str1_lower)
    processed_str2 = filter_alphabetic(str2_lower)

    sorted_str1 = sorted(processed_str1)
    sorted_str2 = sorted(processed_str2)

    return sorted_str1 == sorted_str2

def test_simple_anagrams():
    assert is_anagram('listen', 'silent') is True

def test_case_insensitive_anagrams():
    assert is_anagram('Debit Card', 'Bad Credit') is True

def test_anagrams_with_punctuation_and_spaces():
    assert is_anagram('A gentleman.', 'Elegant man') is True

def test_not_anagrams_different_length():
    assert is_anagram('hello', 'world') is False

def test_not_anagrams_same_length_different_chars():
    assert is_anagram('apple', 'apply') is False

def test_anagrams_ignore_numbers_special_chars():
    assert is_anagram('earth 123', 'heart 456') is True

def test_empty_strings():
    assert is_anagram('', '') is True

def test_one_empty_one_non_empty():
    assert is_anagram('', 'a') is False

```

Overwriting test_anagram.py

```

# Install pytest if you haven't already
$pip install pytest

# Run pytest
print('\n--- Pytest Test Cases Output ---')
!pytest test_anagram.py

```

```

Requirement already satisfied: pytest in /usr/local/lib/python3.12/dist-packages (8.4.2)
Requirement already satisfied: iniconfig>=1 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.3.0)
Requirement already satisfied: packaging>=20 in /usr/local/lib/python3.12/dist-packages (from pytest) (26.0)
Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.12/dist-packages (from pytest) (1.6.0)
Requirement already satisfied: pygments>=2.7.2 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.19.2)

```

```

--- Pytest Test Cases Output ---
===== test session starts =====
platform linux -- Python 3.12.12, pytest-8.4.2, pluggy-1.6.0
rootdir: /content
plugins: anyio-4.12.1, langsmith-0.6.9, typeguard-4.4.4
collected 8 items

test_anagram.py ..... [100%]

===== 8 passed in 0.03s =====

```

```

❏ _XXwritefile test_anagram_unittest.py
import unittest

def is_anagram(str1, str2):
    str1_lower = str1.lower()
    str2_lower = str2.lower()

    def filter_alphabetic(s):
        return ''.join(char for char in s if char.isalpha())

    processed_str1 = filter_alphabetic(str1_lower)
    processed_str2 = filter_alphabetic(str2_lower)

    sorted_str1 = sorted(processed_str1)
    sorted_str2 = sorted(processed_str2)

    return sorted_str1 == sorted_str2

class TestIsAnagram(unittest.TestCase):

    def test_simple_anagrams(self):
        self.assertTrue(is_anagram('listen', 'silent'))

    def test_case_insensitive_anagrams(self):
        self.assertTrue(is_anagram('Debit Card', 'Bad Credit'))

    def test_anagrams_with_punctuation_and_spaces(self):
        self.assertTrue(is_anagram('A gentleman.', 'Elegant man'))

    def test_not_anagrams_different_length(self):
        self.assertFalse(is_anagram('hello', 'world'))

    def test_not_anagrams_same_length_different_chars(self):
        self.assertFalse(is_anagram('apple', 'apply'))

    def test_anagrams_ignore_numbers_special_chars(self):
        self.assertTrue(is_anagram('earth 123', 'heart 456'))

    def test_empty_strings(self):
        self.assertTrue(is_anagram('', ''))

    def test_one_empty_one_non_empty(self):
        self.assertFalse(is_anagram('', 'a'))

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

```

```

*** Writing test_anagram_unittest.py

```

```

print('\n--- Unittest Test Cases Output ---')
!python -m unittest test_anagram_unittest.py

```

```

--- Unittest Test Cases Output ---

```

```

.....

```

```

-----
Ran 8 tests in 0.000s

```

```

OK

```

Task Description #4 (Inventory Class – Apply AI to Simulate Real-

World Inventory System)

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

- Methods:

- o add_item(name, quantity)

- o remove_item(name, quantity)

- o get_stock(name)

PROMPT: Write Inventory class with:

add_item()

remove_item()

get_stock()

Generate 4 test cases and show outputs using:

General, Assert, Pytest, Unit Test

```

class Inventory:
    def __init__(self):
        self.stock = {}

    def add_item(self, item_name, quantity):
        if not isinstance(quantity, int) or quantity <= 0:
            print("Error: Quantity must be a positive integer.")
            return

        self.stock[item_name] = self.stock.get(item_name, 0) + quantity
        print(f"Added {quantity} of {item_name}. Current stock: {self.stock[item_name]}")

    def remove_item(self, item_name, quantity):
        if not isinstance(quantity, int) or quantity <= 0:
            print("Error: Quantity to remove must be a positive integer.")
            return

        if item_name not in self.stock or self.stock[item_name] == 0:
            print(f"Error: {item_name} not in stock or quantity is zero.")
            return

        current_stock = self.stock[item_name]
        if quantity >= current_stock:
            print(f"Removing all {current_stock} of {item_name}. Stock is now 0.")
            self.stock[item_name] = 0
        else:
            self.stock[item_name] -= quantity
            print(f"Removed {quantity} of {item_name}. Current stock: {self.stock[item_name]}")

    def get_stock(self, item_name):
        return self.stock.get(item_name, 0)

```

```

print('--- Inventory Class General Test Cases (using print()) ---')
# 1. Create an instance of the Inventory class
inventory = Inventory()
print("Inventory initialized.")
# 2. Call the add_item method multiple times
inventory.add_item('Apples', 10)
inventory.add_item('Bananas', 5)
inventory.add_item('Apples', 3) # Add more apples
inventory.add_item('Oranges', 7)
print(f"Current inventory after additions: {inventory.stock}")
# 3. Call the get_stock method to check the stock of existing and non-existing items
print(f"Stock of Apples: {inventory.get_stock('Apples')} (Expected: 13)")
print(f"Stock of Bananas: {inventory.get_stock('Bananas')} (Expected: 5)")
print(f"Stock of Grapes: {inventory.get_stock('Grapes')} (Expected: 0)")
# 4. Call the remove_item method for existing items with sufficient quantities
inventory.remove_item('Apples', 5)
print(f"Stock of Apples after removing 5: {inventory.get_stock('Apples')} (Expected: 8)")
# 5. Call the remove_item method for quantities that would result in zero stock and exceeding current stock
inventory.remove_item('Bananas', 5) # Remove all bananas
print(f"Stock of Bananas after removing all: {inventory.get_stock('Bananas')} (Expected: 0)")
inventory.remove_item('Oranges', 10) # Remove more than available
print(f"Stock of Oranges after removing 10 (was 7): {inventory.get_stock('Oranges')} (Expected: 0)")
# 6. Call the remove_item method for non-existent items
inventory.remove_item('Grapes', 2) # Grapes not in stock
print(f"Stock of Grapes (should be 0): {inventory.get_stock('Grapes')} (Expected: 0)")
# 7. Add an item with a non-positive integer quantity
inventory.add_item('Pears', 0) # Invalid quantity
inventory.add_item('Pears', -5) # Invalid quantity
inventory.add_item('Pears', 8) # Valid quantity
print(f"Stock of Pears: {inventory.get_stock('Pears')} (Expected: 8)")
# 8. Remove an item with a non-positive integer quantity
inventory.remove_item('Pears', 0) # Invalid quantity
inventory.remove_item('Pears', -3) # Invalid quantity
inventory.remove_item('Pears', 3) # Valid removal
print(f"Stock of Pears after removing 3: {inventory.get_stock('Pears')} (Expected: 5)")
print("---- General Test Cases Complete ----")

```

Inventory Class General Test Cases (using print())

```

--- Inventory Class General Test Cases (using print()) ---
Inventory initialized.
Added 10 of Apples. Current stock: 10
Added 5 of Bananas. Current stock: 5
Added 3 of Apples. Current stock: 13
Added 7 of Oranges. Current stock: 7
Current inventory after additions: {'Apples': 13, 'Bananas': 5, 'Oranges': 7}
Stock of Apples: 13 (Expected: 13)
Stock of Bananas: 5 (Expected: 5)
Stock of Grapes: 0 (Expected: 0)
Removed 5 of Apples. Current stock: 8
Stock of Apples after removing 5: 8 (Expected: 8)
Removing all 5 of Bananas. Stock is now 0.
Stock of Bananas after removing all: 0 (Expected: 0)
Removing all 7 of Oranges. Stock is now 0.
Stock of Oranges after removing 10 (was 7): 0 (Expected: 0)
Error: Grapes not in stock or quantity is zero.
Stock of Grapes (should be 0): 0 (Expected: 0)
Error: Quantity must be a positive integer.
Error: Quantity must be a positive integer.
Added 8 of Pears. Current stock: 8
Stock of Pears: 8 (Expected: 8)
Error: Quantity to remove must be a positive integer.
Error: Quantity to remove must be a positive integer.
Removed 3 of Pears. Current stock: 5
Stock of Pears after removing 3: 5 (Expected: 5)
--- General Test Cases Complete ---

```

```

print("\n--- Inventory Class Assert Test Cases ---")

try:
    # Initialize inventory
    inventory = Inventory()

    # Test add_item method
    inventory.add_item('Laptops', 5)
    assert inventory.get_stock('Laptops') == 5, "Test Case 1 Failed: Initial add_item"

    inventory.add_item('Keyboards', 10)
    assert inventory.get_stock('Keyboards') == 10, "Test Case 2 Failed: Initial add_item another item"

    inventory.add_item('Laptops', 3)
    assert inventory.get_stock('Laptops') == 8, "Test Case 3 Failed: Add more to existing item"

    # Test get_stock method
    assert inventory.get_stock('Monitors') == 0, "Test Case 4 Failed: Get stock of non-existent item"
    assert inventory.get_stock('Keyboards') == 10, "Test Case 5 Failed: Get stock of existing item"

    # Test remove_item method
    inventory.remove_item('Laptops', 2)
    assert inventory.get_stock('Laptops') == 6, "Test Case 6 Failed: Partial remove_item"

    inventory.remove_item('Keyboards', 10) # Remove entire quantity
    assert inventory.get_stock('Keyboards') == 0, "Test Case 7 Failed: Remove entire quantity"

    # Attempt to remove more than available
    inventory.remove_item('Laptops', 10) # Should remove all 6, stock becomes 0
    assert inventory.get_stock('Laptops') == 0, "Test Case 8 Failed: Remove more than available"

    # Attempt to remove a non-existent item
    inventory.remove_item('Mice', 5) # Should not affect stock, remains 0
    assert inventory.get_stock('Mice') == 0, "Test Case 9 Failed: Remove non-existent item"

    # Test invalid quantities for add_item
    initial_stock_printers = inventory.get_stock('Printers')
    inventory.add_item('Printers', 0) # Invalid
    assert inventory.get_stock('Printers') == initial_stock_printers, "Test Case 10 Failed: add_item with zero quantity should not change stock"
    inventory.add_item('Printers', -2) # Invalid
    assert inventory.get_stock('Printers') == initial_stock_printers, "Test Case 11 Failed: add_item with negative quantity should not change stock"
    inventory.add_item('Printers', 5) # Valid
    assert inventory.get_stock('Printers') == initial_stock_printers + 5, "Test Case 12 Failed: add_item with valid quantity"

    # Test invalid quantities for remove_item
    initial_stock_printers = inventory.get_stock('Printers')
    inventory.remove_item('Printers', 0) # Invalid
    assert inventory.get_stock('Printers') == initial_stock_printers, "Test Case 13 Failed: remove_item with zero quantity should not change stock"
    inventory.remove_item('Printers', -2) # Invalid
    assert inventory.get_stock('Printers') == initial_stock_printers, "Test Case 14 Failed: remove_item with negative quantity should not change stock"

    print("All Inventory assert tests passed!")
except AssertionError as e:
    print(f"Inventory assert test failed: {e}")

```

```

--- Inventory Class Assert Test Cases ---
Added 5 of Laptops. Current stock: 5
Added 10 of Keyboards. Current stock: 10
Added 3 of Laptops. Current stock: 8
Removed 2 of Laptops. Current stock: 6
Removing all 10 of Keyboards. Stock is now 0.
Removing all 6 of Laptops. Stock is now 0.
Error: Mice not in stock or quantity is zero.
Error: Quantity must be a positive integer.
Error: Quantity must be a positive integer.
Added 5 of Printers. Current stock: 5
Error: Quantity to remove must be a positive integer.
Error: Quantity to remove must be a positive integer.
All Inventory assert tests passed!

```

```

❶ Writefile test_inventory.py
import pytest

# Define the Inventory class here to make the test file self-contained
class Inventory:
    def __init__(self):
        self.stock = {}

    def add_item(self, item_name, quantity):
        if not isinstance(quantity, int) or quantity <= 0:
            # print("Error: Quantity must be a positive integer.") # Suppress print for tests
            return

        self.stock[item_name] = self.stock.get(item_name, 0) + quantity
        # print(f"Added {quantity} of {item_name}. Current stock: {self.stock[item_name]}") # Suppress print for tests

    def remove_item(self, item_name, quantity):
        if not isinstance(quantity, int) or quantity <= 0:
            # print("Error: Quantity to remove must be a positive integer.") # Suppress print for tests
            return

        if item_name not in self.stock or self.stock[item_name] == 0:
            # print(f"Error: {item_name} not in stock or quantity is zero.") # Suppress print for tests
            return

        current_stock = self.stock[item_name]
        if quantity >= current_stock:
            # print(f"Removing all {current_stock} of {item_name}. Stock is now 0.") # Suppress print for tests
            self.stock[item_name] = 0
        else:
            self.stock[item_name] -= quantity
            # print(f"Removed {quantity} of {item_name}. Current stock: {self.stock[item_name]}") # Suppress print for tests

    def get_stock(self, item_name):
        return self.stock.get(item_name, 0)

# Pytest test functions

def test_add_item_new():
    inventory = Inventory()
    inventory.add_item('Laptops', 5)
    assert inventory.get_stock('Laptops') == 5

def test_add_item_existing():
    inventory = Inventory()
    inventory.add_item('Laptops', 5)
    inventory.add_item('Laptops', 3)
    assert inventory.get_stock('Laptops') == 8

def test_get_stock_existing():
    inventory = Inventory()
    inventory.add_item('Keyboards', 10)
    assert inventory.get_stock('Keyboards') == 10

def test_get_stock_non_existent():
    inventory = Inventory()
    assert inventory.get_stock('Monitors') == 0

def test_remove_item_partial():
    inventory = Inventory()
    inventory.add_item('Laptops', 10)
    inventory.remove_item('Laptops', 4)
    assert inventory.get_stock('Laptops') == 6

def test_remove_item_all():
    inventory = Inventory()
    inventory.add_item('Keyboards', 7)
    inventory.remove_item('Keyboards', 7)
    # print(f"Removing all {current_stock} of {item_name}. Stock is now 0.") # Suppress print for tests

```

```

        inventory.remove_item('Mice', 10)
        assert inventory.get_stock('Mice') == 0

def test_remove_item_non_existent():
    inventory = Inventory()
    inventory.remove_item('Printers', 5) # Should do nothing
    assert inventory.get_stock('Printers') == 0

def test_add_item_invalid_quantity_zero():
    inventory = Inventory()
    inventory.add_item('Monitors', 0) # Invalid
    assert inventory.get_stock('Monitors') == 0

def test_add_item_invalid_quantity_negative():
    inventory = Inventory()
    inventory.add_item('Monitors', -5) # Invalid
    assert inventory.get_stock('Monitors') == 0

def test_remove_item_invalid_quantity_zero():
    inventory = Inventory()
    inventory.add_item('Speakers', 10)
    inventory.remove_item('Speakers', 0) # Invalid
    assert inventory.get_stock('Speakers') == 10

def test_remove_item_invalid_quantity_negative():
    inventory = Inventory()
    inventory.add_item('Speakers', 10)
    inventory.remove_item('Speakers', -3) # Invalid
    assert inventory.get_stock('Speakers') == 10

```

Writing test_inventory.py



```

# Install pytest if you haven't already
%pip install pytest

# Run pytest
print('\n--- Pytest Test Cases Output ---')
!pytest test_inventory.py

```

```

--- Requirement already satisfied: pytest in /usr/local/lib/python3.12/dist-packages (8.4.2)
Requirement already satisfied: iniconfig>=1 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.3.0)
Requirement already satisfied: packaging>=20 in /usr/local/lib/python3.12/dist-packages (from pytest) (26.0)
Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.12/dist-packages (from pytest) (1.6.0)
Requirement already satisfied: pygments>=2.7.2 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.19.2)

```

```

--- Pytest Test Cases Output ---
===== test session starts =====
platform linux -- Python 3.12.12, pytest-8.4.2, pluggy-1.6.0
rootdir: /content
plugins: anyio-4.12.1, langsmith-0.6.9, typeguard-4.4.4
collected 12 items

test_inventory.py ..... [100%]

===== 12 passed in 0.03s =====

```

```

print('\n--- Unittest Test Cases Output ---')
!python -m unittest test_inventory_unittest.py

```

```

--- Unittest Test Cases Output ---
.....
-----
Ran 12 tests in 0.001s

OK

```

Task Description #5 (Date Validation & Formatting – Apply AI for Data Validation)

- Task: Use AI to generate at least 3 assert test cases for `validate_and_format_date(date_str)` to check and convert

dates.

- **Requirements:**

- o **Validate "MM/DD/YYYY" format.**

- o **Handle invalid dates.**

- o **Convert valid dates to "YYYY-MM-DD".**

PROMPT:

Write function `validate_and_format_date(date)`.

Conditions:

MM/DD/YYYY → YYYY-MM-DD

Generate 4 test cases and outputs using:

General, Assert, Pytest, Unit Test

```
1 from datetime import datetime
2
3 def validate_and_format_date(date_str):
4     """
5     Validates and formats a date string from MM/DD/YYYY to YYYY-MM-DD.
6     Returns the formatted date string or None if the input is invalid.
7     """
8     try:
9         # Attempt to parse the date string in MM/DD/YYYY format
10        # If parsing is successful, format it to YYYY-MM-DD
11        formatted_date = datetime.strptime(date_str, "%m/%d/%Y").strftime("%Y-%m-%d")
12        return formatted_date
13    except ValueError:
14        # If parsing fails, the date string is not in the expected format or is invalid
15        return None
16
17
18 # Install pytest if you haven't already
19 pip install pytest
20
21 # Run pytest
22 print("\n--- Pytest Test Cases Output ---")
23 pytest test_date_formatter.py
24
25 Requirement already satisfied: pytest in /usr/local/lib/python3.12/dist-packages (8.4.2)
26 Requirement already satisfied: iniconfig<1 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.3.0)
27 Requirement already satisfied: packaging>=20 in /usr/local/lib/python3.12/dist-packages (from pytest) (26.4)
28 Requirement already satisfied: pluggy<2, >=1.5 in /usr/local/lib/python3.12/dist-packages (from pytest) (1.6.0)
29 Requirement already satisfied: pygments>=2.7.2 in /usr/local/lib/python3.12/dist-packages (from pytest) (2.19.2)
30
31 --- Pytest Test Cases Output ---
32 ----- test session starts -----
33 platform linux -- Python 3.12.12, pytest-8.4.2, pluggy-1.6.0
34 rootdir: /content
35 plugins: myio-4.12.1, langsmith-0.6.9, typeguard-4.4.4
36 collected 9 items
37
38 test_date_formatter.py ..... [100%]
39
40 ===== 9 passed in 0.02s =====
41
42
43 print("\n--- Unittest Test Cases Output ---")
44 python -m unittest test_date_formatter_unittest.py
45
46 --- Unittest Test Cases Output ---
47 .....
48 Ran 9 tests in 0.004s
49
50 OK
```

```

64 print("---- General Test Cases for validate_and_format_date (using print()) ----")
65
66 # Test Case 1: Valid date
67 date_1 = "12/25/2023"
68 print(f"{date_1} formatted: {validate_and_format_date(date_1)} (Expected: '2023-12-25')")
69
70 # Test Case 2: Valid date with single digit month/day
71 date_2 = "01/01/2024"
72 print(f"{date_2} formatted: {validate_and_format_date(date_2)} (Expected: '2024-01-01')")
73
74 # Test Case 3: Invalid month
75 date_3 = "13/01/2023"
76 print(f"{date_3} formatted: {validate_and_format_date(date_3)} (Expected: None)")
77
78 # Test Case 4: Invalid day
79 date_4 = "02/29/2023" # February 2023 only has 28 days
80 print(f"{date_4} formatted: {validate_and_format_date(date_4)} (Expected: None)")
81
82 # Test Case 5: Incorrect format (DD-MM-YYYY)
83 date_5 = "25-12-2023"
84 print(f"{date_5} formatted: {validate_and_format_date(date_5)} (Expected: None)")
85
86 # Test Case 6: Incorrect format (YYYY/MM/DD)
87 date_6 = "2023/12/25"
88 print(f"{date_6} formatted: {validate_and_format_date(date_6)} (Expected: None)")
89
90 # Test Case 7: Non-date string
91 date_7 = "not a date"
92 print(f"{date_7} formatted: {validate_and_format_date(date_7)} (Expected: None)")
93
94 # Test Case 8: Leap year valid date
95 date_8 = "02/29/2024"
96 print(f"{date_8} formatted: {validate_and_format_date(date_8)} (Expected: '2024-02-29')")
97
98 # Test Case 9: Leap year invalid date
99 date_9 = "02/29/2023"
100 print(f"{date_9} formatted: {validate_and_format_date(date_9)} (Expected: None)")
101
102 --- General Test Cases for validate_and_format_date (using print()) ---
103 '12/25/2023' formatted: 2023-12-25 (Expected: '2023-12-25')
104 '01/01/2024' formatted: 2024-01-01 (Expected: '2024-01-01')
105 '13/01/2023' formatted: None (Expected: None)
106 '02/29/2023' formatted: None (Expected: None)
107 '25-12-2023' formatted: None (Expected: None)
108 '2023/12/25' formatted: None (Expected: None)
109 'not a date' formatted: None (Expected: None)
110 '02/29/2024' formatted: 2024-02-29 (Expected: '2024-02-29')
111 '02/29/2023' formatted: None (Expected: None)

```

```

102 print("\n---- Assert Test Cases for validate_and_format_date ----")
103
104 try:
105     assert validate_and_format_date("12/25/2023") == "2023-12-25", "Test Case 1 Failed: Valid date"
106     assert validate_and_format_date("01/01/2024") == "2024-01-01", "Test Case 2 Failed: Valid date single digit"
107     assert validate_and_format_date("13/01/2023") is None, "Test Case 3 Failed: Invalid month"
108     assert validate_and_format_date("02/29/2023") is None, "Test Case 4 Failed: Invalid day"
109     assert validate_and_format_date("25-12-2023") is None, "Test Case 5 Failed: Incorrect format DD-MM-YYYY"
110     assert validate_and_format_date("2023/12/25") is None, "Test Case 6 Failed: Incorrect format YYYY/MM/DD"
111     assert validate_and_format_date("not a date") is None, "Test Case 7 Failed: Non-date string"
112     assert validate_and_format_date("02/29/2024") == "2024-02-29", "Test Case 8 Failed: Leap year valid date"
113     assert validate_and_format_date("02/29/2023") is None, "Test Case 9 Failed: Leap year invalid date"
114     print("All assert tests passed!")
115 except AssertionError as e:
116     print(f"Assert test failed: {e}")
117
118 --- Assert Test Cases for validate_and_format_date ---
119 All assert tests passed!

```