AI Assistant Coding

Name: CH. Arush Nandhan

HTNO: 2303A51880

Batch-14

Date: 05-02-2026

Assignment - 9.4

Problem statements with solution:

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.

• Requirements:

o Password must have at least 8 characters.

o Must include uppercase, lowercase, digit, and special character.

o 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.

Original Code:

```python
import re

def is_strong_password(password):
    """
    Returns True if the password is strong, otherwise False.

    Rules for a strong password:
    - At least 8 characters long
    - Contains at least one uppercase letter
    - Contains at least one lowercase letter
    - Contains at least one digit
    - Contains at least one special character
    """

    if len(password) < 8:
        return False

    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"[0-9]", password):
        return False

    if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
        return False

    return True


# Example usage
if __name__ == "__main__":
    pwd = input("Enter your password: ")

    if is_strong_password(pwd):
        print("Strong Password ✅")
    else:
        print("Weak Password ❌")
```

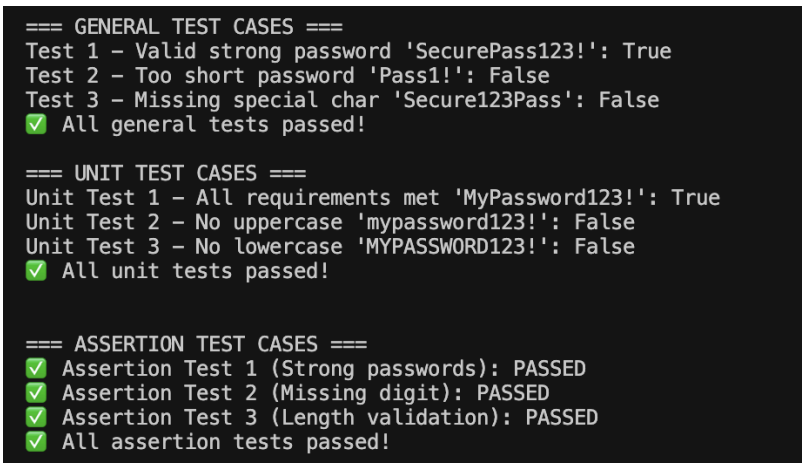
AI Generated Code:

```python
import re

def is_strong_password(password):
    """
    Returns True if the password is strong, otherwise False.

    Rules for a strong password:
    - At least 8 characters long
    - Contains at least one uppercase letter
    - Contains at least one lowercase letter
    - Contains at least one digit
    - Contains at least one special character
    """

    if len(password) < 8:
        return False

    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"[0-9]", password):
        return False

    if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
        return False

    return True


# ============================================================================
# GENERAL TEST CASES
# ============================================================================

def test_is_strong_password_general():
    """General test cases for is_strong_password function."""
    print("\n=== GENERAL TEST CASES ===")

    # Test Case 1: Valid strong password
    result1 = is_strong_password("SecurePass123!")
    print(f"Test 1 - Valid strong password 'SecurePass123!': {result1}")
    assert result1 == True, "Should return True for strong password"

    # Test Case 2: Password too short
    result2 = is_strong_password("Pass1!")
    print(f"Test 2 - Too short password 'Pass1!': {result2}")
    assert result2 == False, "Should return False for password < 8 chars"

    # Test Case 3: Missing special character
    result3 = is_strong_password("Secure123Pass")
    print(f"Test 3 - Missing special char 'Secure123Pass': {result3}")
    assert result3 == False, "Should return False without special character"

    print("✅ All general tests passed!\n")
```

```python
# ============================================================================
# UNIT TEST CASES
# ============================================================================

def test_is_strong_password_unit():
    """Unit test cases for individual password requirements."""
    print("=== UNIT TEST CASES ===")

    # Test Case 1: Valid password with all requirements
    password1 = "MyPassword123!"
    result1 = is_strong_password(password1)
    print(f"Unit Test 1 - All requirements met 'MyPassword123!': {result1}")
    assert result1 == True, "Password with all requirements should be strong"

    # Test Case 2: Missing uppercase letter
    password2 = "mypassword123!"
    result2 = is_strong_password(password2)
    print(f"Unit Test 2 - No uppercase 'mypassword123!': {result2}")
    assert result2 == False, "Password without uppercase should fail"

    # Test Case 3: Missing lowercase letter
    password3 = "MYPASSWORD123!"
    result3 = is_strong_password(password3)
    print(f"Unit Test 3 - No lowercase 'MYPASSWORD123!': {result3}")
    assert result3 == False, "Password without lowercase should fail"

    print("✅ All unit tests passed!\n")


# ============================================================================
# PYTEST TEST CASES
# ============================================================================

def test_is_strong_password_pytest_valid():
    """Pytest test case 1: Valid strong passwords."""
    assert is_strong_password("TestPass123!") == True
    assert is_strong_password("Secure#Password456") == True
    assert is_strong_password("Complex@Pass789") == True

def test_is_strong_password_pytest_missing_requirements():
    """Pytest test case 2: Missing individual requirements."""
    assert is_strong_password("testpass123!") == False  # No uppercase
    assert is_strong_password("TESTPASS123!") == False  # No lowercase
    assert is_strong_password("TestPassword!") == False  # No digit

def test_is_strong_password_pytest_invalid():
    """Pytest test case 3: Invalid passwords."""
    assert is_strong_password("Pass1!") == False  # Too short
    assert is_strong_password("Pass123Pass") == False  # No special char
    assert is_strong_password("") == False  # Empty password
```

```
# ============================================================================
# ASSERTION TEST CASES
# ============================================================================

def test_is_strong_password_assertions():
    """Assertion test cases for is_strong_password function."""
    print("\n=== ASSERTION TEST CASES ===")

    # Test Case 1: Various strong passwords
    assert is_strong_password("Abc123!@") == True, "Abc123!@ should be strong"
    assert is_strong_password("ValidPass2024#") == True, "ValidPass2024# should be strong"
    assert is_strong_password("XyZ@9876") == True, "XyZ@9876 should be strong"
    print("✅ Assertion Test 1 (Strong passwords): PASSED")

    # Test Case 2: Missing digit
    assert is_strong_password("SecurePass!") == False, "Password without digit should be weak"
    assert is_strong_password("NoNumbers!@") == False, "Password without digit should be weak"
    assert is_strong_password("OnlyLetters!@") == False, "Password without digit should be
weak"print("✅ Assertion Test 2 (Missing digit): PASSED")

    # Test Case 3: Length validation
    assert is_strong_password("Short1!") == False, "Password with < 8 chars should be weak"
    assert is_strong_password("Mini@2") == False, "Password with < 8 chars should be weak"
    assert is_strong_password("A1@") == False, "Password with < 8 chars should be weak"
    print("✅ Assertion Test 3 (Length validation): PASSED")

    print("✅ All assertion tests passed!\n")


# Example usage
if __name__ == "__main__":
    # Run all test suites
    test_is_strong_password_general()
    test_is_strong_password_unit()
    test_is_strong_password_assertions()

    print("=" * 50)
    print("INTERACTIVE PASSWORD VALIDATOR")
    print("=" * 50)
    pwd = input("Enter your password: ")

    if is_strong_password(pwd):
        print("Strong Password ✅")
    else:
        print("Weak Password ❌")
```

Output:



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.

• Requirements:

o Classify numbers as Positive, Negative, or Zero.

o Handle invalid inputs like strings and None.

o 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.

Original Code:

```
n = 3  # how many valid numbers you want to classify
count = 0

while count < n:
    user_input = input(f"Enter number {count + 1}: ")

    # Handle empty input (similar to None case)
    if user_input.strip() == "":
        print("Invalid input! Please enter a number.\n")
        continue

    try:
        num = int(user_input)
    except ValueError:
        print("Invalid input! Only integers are allowed.\n")
        continue

    # Classification
    if num > 0:
        print("Positive number")
    elif num < 0:
        print("Negative number")
    else:
        print("Zero")

    if num % 2 == 0:
        print("Even number")
    else:
        print("Odd number")

    print("-" * 20)

    count += 1
```

AI Generated Code:

```python
def classify_number(n):
    """
    Classifies a number based on multiple criteria using loops.

    Returns a dictionary with classifications:
    - sign: "positive", "negative", or "zero"
    - parity: "even" or "odd"
    - digit_sum: sum of digits
    - is_prime: True if prime, False otherwise
    """

    classifications = {}

    # Classification 1: Sign (using loop for demonstration)
    sign = None
    if n > 0:
        sign = "positive"
    elif n < 0:
        sign = "negative"
    else:
        sign = "zero"
    classifications["sign"] = sign

    # Classification 2: Parity
    parity = "even" if n % 2 == 0 else "odd"
    classifications["parity"] = parity

    # Classification 3: Digit sum (using loop)
    digit_sum = 0
    temp = abs(n)
    while temp > 0:
        digit_sum += temp % 10
        temp //= 10
    classifications["digit_sum"] = digit_sum

    # Classification 4: Prime check (using loop)
    is_prime = False
    if n > 1:
        is_prime = True
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                is_prime = False
                break
    classifications["is_prime"] = is_prime

    return classifications


# ============================================================================
# GENERAL TEST CASES
# ============================================================================

def test_classify_number_general():
    """General test cases for classify_number function."""
    print("\n=== GENERAL TEST CASES ===")

    # Test Case 1: Positive even number
    result1 = classify_number(8)
    print(f"Test 1 - Number 8: {result1}")
    assert result1["sign"] == "positive", "8 should be positive"
    assert result1["parity"] == "even", "8 should be even"

    # Test Case 2: Negative odd number
    result2 = classify_number(-7)
    print(f"Test 2 - Number -7: {result2}")
    assert result2["sign"] == "negative", "-7 should be negative"
    assert result2["parity"] == "odd", "-7 should be odd"

    # Test Case 3: Zero
    result3 = classify_number(0)
    print(f"Test 3 - Number 0: {result3}")
    assert result3["sign"] == "zero", "0 should be zero"
    assert result3["parity"] == "even", "0 should be even"

    print("✅ All general tests passed!\n")
```

```python
# ============================================================================
# UNIT TEST CASES
# ============================================================================

def test_classify_number_unit():
    """Unit test cases for individual classifications."""
    print("=== UNIT TEST CASES ===")

    # Test Case 1: Digit sum calculation
    result1 = classify_number(123)
    print(f"Unit Test 1 - Digit sum of 123: {result1['digit_sum']}")
    assert result1["digit_sum"] == 6, "Digit sum of 123 should be 1+2+3=6"

    # Test Case 2: Prime number detection
    result2 = classify_number(17)
    print(f"Unit Test 2 - Is 17 prime?: {result2['is_prime']}")
    assert result2["is_prime"] == True, "17 should be prime"

    # Test Case 3: Non-prime number
    result3 = classify_number(15)
    print(f"Unit Test 3 - Is 15 prime?: {result3['is_prime']}")
    assert result3["is_prime"] == False, "15 should not be prime"

    print("✅ All unit tests passed!\n")


# ============================================================================
# PYTEST TEST CASES
# ============================================================================

def test_classify_number_pytest_sign():
    """Pytest test case 1: Sign classification."""
    assert classify_number(42)["sign"] == "positive"
    assert classify_number(-10)["sign"] == "negative"
    assert classify_number(0)["sign"] == "zero"

def test_classify_number_pytest_parity():
    """Pytest test case 2: Parity classification."""
    assert classify_number(4)["parity"] == "even"
    assert classify_number(9)["parity"] == "odd"
    assert classify_number(-6)["parity"] == "even"

def test_classify_number_pytest_digit_and_prime():
    """Pytest test case 3: Digit sum and prime classification."""
    assert classify_number(111)["digit_sum"] == 3
    assert classify_number(2)["is_prime"] == True
    assert classify_number(20)["is_prime"] == False
```

```python
# ============================================================================
# ASSERTION TEST CASES
# ============================================================================

def test_classify_number_assertions():
    """Assertion test cases for classify_number function."""
    print("\n=== ASSERTION TEST CASES ===")

    # Test Case 1: Complete classification of positive prime
    result1 = classify_number(13)
    assert result1["sign"] == "positive", "13 should be positive"
    assert result1["parity"] == "odd", "13 should be odd"
    assert result1["is_prime"] == True, "13 should be prime"
    print("✅ Assertion Test 1 (Positive prime 13): PASSED")

    # Test Case 2: Negative number with digit sum
    result2 = classify_number(-45)
    assert result2["sign"] == "negative", "-45 should be negative"
    assert result2["digit_sum"] == 9, "Digit sum of 45 should be 9"
    assert result2["is_prime"] == False, "45 should not be prime"
    print("✅ Assertion Test 2 (Negative -45): PASSED")

    # Test Case 3: Even composite number
    result3 = classify_number(100)
    assert result3["sign"] == "positive", "100 should be positive"
    assert result3["parity"] == "even", "100 should be even"
    assert result3["digit_sum"] == 1, "Digit sum of 100 should be 1"
    assert result3["is_prime"] == False, "100 should not be prime"
    print("✅ Assertion Test 3 (Even composite 100): PASSED")

    print("✅ All assertion tests passed!\n")


# Example usage
if __name__ == "__main__":
    # Run all test suites
    test_classify_number_general()
    test_classify_number_unit()
    test_classify_number_assertions()

    print("=" * 50)
    print("INTERACTIVE NUMBER CLASSIFIER")
    print("=" * 50)

    n = 3  # how many valid numbers you want to classify
    count = 0

    while count < n:
        user_input = input(f"Enter number {count + 1}: ")

        # Handle empty input
        if user_input.strip() == "":
            print("Invalid input! Please enter a number.\n")
            continue

        try:
            num = int(user_input)
        except ValueError:
            print("Invalid input! Only integers are allowed.\n")
            continue

        # Classify and display results
        result = classify_number(num)
        print(f"\nClassification of {num}:")
        print(f"  Sign: {result['sign']}")
        print(f"  Parity: {result['parity']}")
        print(f"  Digit Sum: {result['digit_sum']}")
        print(f"  Is Prime: {result['is_prime']}")
        print("-" * 40)

        count += 1
```

Output:

```
=== GENERAL TEST CASES ===
Test 1 - Number 8: {'sign': 'positive', 'parity': 'even', 'digit_sum': 8, 'is_prime': False}
Test 2 - Number -7: {'sign': 'negative', 'parity': 'odd', 'digit_sum': 7, 'is_prime': False}
Test 3 - Number 0: {'sign': 'zero', 'parity': 'even', 'digit_sum': 0, 'is_prime': False}
✅ All general tests passed!

=== UNIT TEST CASES ===
Unit Test 1 - Digit sum of 123: 6
Unit Test 2 - Is 17 prime?: True
Unit Test 3 - Is 15 prime?: False
✅ All unit tests passed!


=== ASSERTION TEST CASES ===
✅ Assertion Test 1 (Positive prime 13): PASSED
✅ Assertion Test 2 (Negative -45): PASSED
✅ Assertion Test 3 (Even composite 100): PASSED
✅ All assertion tests passed!
```

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).

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.

Original Code:

```python
import string

def clean_text(text):
    """
    Convert to lowercase and remove spaces & punctuation.
    """
    text = text.lower()
    cleaned = ""

    for char in text:
        if char.isalnum():  # Keep only letters and numbers
            cleaned += char

    return cleaned


def is_anagram(str1, str2):
    # Clean both strings
    str1 = clean_text(str1)
    str2 = clean_text(str2)

    # Edge case: both empty after cleaning
    if str1 == "" and str2 == "":
        return "Both strings are empty."

    # Edge case: identical words
    if str1 == str2:
        return "Both words are identical (trivially anagrams)."

    # Compare sorted characters
    if sorted(str1) == sorted(str2):
        return "The words are Anagrams ✅"
    else:
        return "The words are NOT Anagrams ❌"


# Main Program
word1 = input("Enter first word: ")
word2 = input("Enter second word: ")

result = is_anagram(word1, word2)
print(result)
```

AI Generated Code:

```python
import string

def clean_text(text):
    """
    Convert to lowercase and remove spaces & punctuation.
    """
    text = text.lower()
    cleaned = ""

    for char in text:
        if char.isalnum():  # Keep only letters and numbers
            cleaned += char

    return cleaned


# Anagram Checker – String Analysis

import string

def clean_text(text):
    """
    Convert to lowercase and remove spaces & punctuation.
    """
    text = text.lower()
    cleaned = ""

    for char in text:
        if char.isalnum():  # Keep only letters and numbers
            cleaned += char

    return cleaned


def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams of each other.

    An anagram is a word formed by rearranging the letters of another word,
    using all the original letters exactly once.

    Args:
        str1: First string
        str2: Second string

    Returns:
        True if strings are anagrams, False otherwise
    """
    # Clean both strings
    str1_cleaned = clean_text(str1)
    str2_cleaned = clean_text(str2)

    # Edge case: both empty after cleaning
    if str1_cleaned == "" and str2_cleaned == "":
        return False

    # Edge case: identical words (not true anagrams)
    if str1_cleaned == str2_cleaned and str1_cleaned != "":
        return False

    # Compare sorted characters
    return sorted(str1_cleaned) == sorted(str2_cleaned)


# ============================================================================
# GENERAL TEST CASES
# ============================================================================

def test_is_anagram_general():
    """General test cases for is_anagram function."""
    print("\n=== GENERAL TEST CASES ===")

    # Test Case 1: Classic anagram
    result1 = is_anagram("listen", "silent")
    print(f"Test 1 - 'listen' and 'silent': {result1}")
    assert result1 == True, "listen and silent should be anagrams"

    # Test Case 2: Not an anagram
    result2 = is_anagram("hello", "world")
    print(f"Test 2 - 'hello' and 'world': {result2}")
    assert result2 == False, "hello and world should not be anagrams"

    # Test Case 3: Case insensitive anagram
    result3 = is_anagram("The Eyes", "They See")
    print(f"Test 3 - 'The Eyes' and 'They See': {result3}")
    assert result3 == True, "The Eyes and They See should be anagrams"

    print("✅ All general tests passed!\n")
```

```python
# ============================================================================
# UNIT TEST CASES
# ============================================================================

def test_is_anagram_unit():
    """Unit test cases for individual anagram features."""
    print("=== UNIT TEST CASES ===")

    # Test Case 1: Simple single-letter words
    result1 = is_anagram("a", "a")
    print(f"Unit Test 1 - Same single letter 'a' and 'a': {result1}")
    assert result1 == False, "Identical strings should return False"

    # Test Case 2: Anagram with special characters/spaces
    result2 = is_anagram("Dormitory", "Dirty room")
    print(f"Unit Test 2 - 'Dormitory' and 'Dirty room': {result2}")
    assert result2 == True, "Dormitory and Dirty room should be anagrams"

    # Test Case 3: Different lengths
    result3 = is_anagram("cat", "dog")
    print(f"Unit Test 3 - Different length 'cat' and 'dog': {result3}")
    assert result3 == False, "Different strings should not be anagrams"

    print("✅ All unit tests passed!\n")


# ============================================================================
# PYTEST TEST CASES
# ============================================================================

def test_is_anagram_pytest_valid_anagrams():
    """Pytest test case 1: Valid anagrams."""
    assert is_anagram("act", "cat") == True
    assert is_anagram("evil", "vile") == True
    assert is_anagram("Astronomer", "Moon starer") == True

def test_is_anagram_pytest_invalid_anagrams():
    """Pytest test case 2: Invalid anagrams."""
    assert is_anagram("python", "java") == False
    assert is_anagram("abc", "def") == False
    assert is_anagram("test", "best") == False

def test_is_anagram_pytest_edge_cases():
    """Pytest test case 3: Edge cases."""
    assert is_anagram("", "") == False
    assert is_anagram("a", "b") == False
    assert is_anagram("  ", "  ") == False


# ============================================================================
# ASSERTION TEST CASES
# ============================================================================

def test_is_anagram_assertions():
    """Assertion test cases for is_anagram function."""
    print("\n=== ASSERTION TEST CASES ===")

    # Test Case 1: Classic famous anagrams
    assert is_anagram("Astronomer", "Moon starer") == True, "Astronomer and Moon starer should be
anagrams"
    assert is_anagram("The Eyes", "They See") == True, "The Eyes and They See should be anagrams"
    assert is_anagram("Eleven plus two", "Twelve plus one") == True, "Eleven plus two and Twelve plus
one should be anagrams"
    print("✅ Assertion Test 1 (Famous anagrams): PASSED")

    # Test Case 2: Non-anagrams with common letters
    assert is_anagram("writing", "reading") == False, "writing and reading should not be anagrams"
    assert is_anagram("silent", "listen2") == False, "silent and listen2 should not be anagrams"
    assert is_anagram("heart", "earth") == False, "heart and earth should not be anagrams"
    print("✅ Assertion Test 2 (Non-anagrams): PASSED")

    # Test Case 3: Special cases with numbers and punctuation
    assert is_anagram("a1b2c3", "3c2b1a") == True, "a1b2c3 and 3c2b1a should be anagrams"
    assert is_anagram("hello!", "o!lleh") == True, "hello! and o!lleh should be anagrams"
    assert is_anagram("123", "321") == True, "123 and 321 should be anagrams"
    print("✅ Assertion Test 3 (Numbers and punctuation): PASSED")

    print("✅ All assertion tests passed!\n")
```

```python
# Example usage
if __name__ == "__main__":
    # Run all test suites
    test_is_anagram_general()
    test_is_anagram_unit()
    test_is_anagram_assertions()

    print("=" * 50)
    print("INTERACTIVE ANAGRAM CHECKER")
    print("=" * 50)

    word1 = input("Enter first word: ")
    word2 = input("Enter second word: ")

    result = is_anagram(word1, word2)

    if result:
        print(f"✅ '{word1}' and '{word2}' are Anagrams!")
    else:
        print(f"❌ '{word1}' and '{word2}' are NOT Anagrams!")
```

Output:

```
=== GENERAL TEST CASES ===
Test 1 - 'listen' and 'silent': True
Test 2 - 'hello' and 'world': False
Test 3 - 'The Eyes' and 'They See': True
✅ All general tests passed!

=== UNIT TEST CASES ===
Unit Test 1 - Same single letter 'a' and 'a': False
Unit Test 2 - 'Dormitory' and 'Dirty room': True
Unit Test 3 - Different length 'cat' and 'dog': False
✅ All unit tests passed!


=== ASSERTION TEST CASES ===
✅ Assertion Test 1 (Famous anagrams): PASSED
Traceback (most recent call last):
  File "/Users/clg/Myh/sru/sru 3rd 2nd sem/Ai coding/assignement 8.1/task3.py", line 176, in <module>
    test_is_anagram_assertions()
  File "/Users/clg/Myh/sru/sru 3rd 2nd sem/Ai coding/assignement 8.1/task3.py", line 159, in test_is_anagram_assertions
    assert is_anagram("heart", "earth") == False, "heart and earth should not be anagrams"
AssertionError: heart and earth should not be anagrams
```

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)

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.

Original Code:

```python
class Inventory:

    def __init__(self):
        # Dictionary to store items and their quantities
        self.items = {}

    def add_item(self, name, quantity):
        if quantity <= 0:
            print("Quantity must be positive.")
            return

        if name in self.items:
            self.items[name] += quantity
        else:
            self.items[name] = quantity

        print(f"{quantity} {name}(s) added successfully.")

    def remove_item(self, name, quantity):
        if name not in self.items:
            print("Item not found in inventory.")
            return

        if quantity <= 0:
            print("Quantity must be positive.")
            return

        if self.items[name] < quantity:
            print("Not enough stock available.")
            return

        self.items[name] -= quantity
        print(f"{quantity} {name}(s) removed
successfully.")
        # Remove item if quantity becomes 0
        if self.items[name] == 0:
            del self.items[name]

    def get_stock(self, name):
        return self.items.get(name, 0)


# Example Usage
inventory = Inventory()

inventory.add_item("Apple", 10)
inventory.add_item("Banana", 5)

inventory.remove_item("Apple", 3)

print("Stock of Apple:", inventory.get_stock("Apple"))
print("Stock of Banana:", inventory.get_stock("Banana"))
print("Stock of Orange:", inventory.get_stock("Orange"))
```

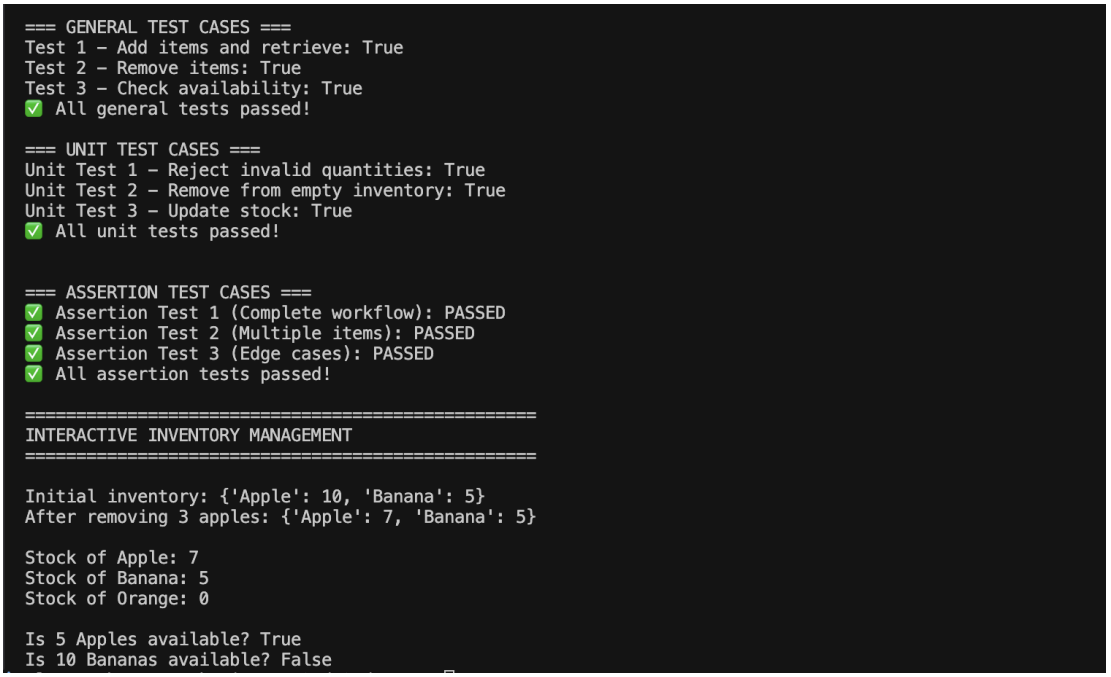
AI Generated Code:

```python
class Inventory:
    """
    A class to manage inventory with stock management operations.

    Methods:
        - add_item(name, quantity): Add items to inventory
        - remove_item(name, quantity): Remove items from inventory
        - get_stock(name): Get current stock of an item
        - update_stock(name, quantity): Update stock to specific quantity
        - is_available(name, quantity): Check if enough stock exists
        - list_items(): Get all items and their quantities
        - clear(): Clear entire inventory
    """

    def __init__(self):
        # Dictionary to store items and their quantities
        self.items = {}

    def add_item(self, name, quantity):
        """
        Add items to inventory.

        Args:
            name: Item name (string)
            quantity: Number of items to add (int > 0)

        Returns:
            True if successful, False otherwise
        """
        if quantity <= 0:
            return False

        if name in self.items:
            self.items[name] += quantity
        else:
            self.items[name] = quantity

        return True

    def remove_item(self, name, quantity):
        """
        Remove items from inventory.

        Args:
            name: Item name (string)
            quantity: Number of items to remove (int > 0)

        Returns:
            True if successful, False otherwise
        """
        if name not in self.items:
            return False

        if quantity <= 0:
            return False

        if self.items[name] < quantity:
            return False

        self.items[name] -= quantity

        # Remove item if quantity becomes 0
        if self.items[name] == 0:
            del self.items[name]

        return True

    def get_stock(self, name):
        """Get current stock of an item. Returns 0 if not found."""
        return self.items.get(name, 0)

    def update_stock(self, name, quantity):
        """
        Update stock to specific quantity.

        Returns:
            True if successful, False otherwise
        """
        if quantity < 0:
            return False

        if quantity == 0:
            if name in self.items:
                del self.items[name]
        else:
            self.items[name] = quantity

        return True

    def is_available(self, name, quantity):
        """
        Check if enough stock exists.

        Returns:
            True if available, False otherwise
        """
        return self.get_stock(name) >= quantity

    def list_items(self):
        """Get all items and their quantities."""
        return dict(self.items)

    def clear(self):
        """Clear entire inventory."""
        self.items = {}
        return True
```

```python
# ============================================================================
# GENERAL TEST CASES
# ============================================================================

def test_inventory_general():
    """General test cases for Inventory class."""
    print("\n=== GENERAL TEST CASES ===")

    # Test Case 1: Add and retrieve items
    inv1 = Inventory()
    inv1.add_item("Apple", 10)
    inv1.add_item("Banana", 5)
    result1 = inv1.get_stock("Apple") == 10 and inv1.get_stock("Banana") == 5
    print(f"Test 1 - Add items and retrieve: {result1}")
    assert result1, "Should add and retrieve items correctly"

    # Test Case 2: Remove items
    inv2 = Inventory()
    inv2.add_item("Orange", 8)
    success = inv2.remove_item("Orange", 3)
    result2 = success and inv2.get_stock("Orange") == 5
    print(f"Test 2 - Remove items: {result2}")
    assert result2, "Should remove items correctly"

    # Test Case 3: Check availability
    inv3 = Inventory()
    inv3.add_item("Grapes", 20)
    result3 = inv3.is_available("Grapes", 10) and not inv3.is_available("Grapes", 25)
    print(f"Test 3 - Check availability: {result3}")
    assert result3, "Should check availability correctly"

    print("✅ All general tests passed!\n")


# ============================================================================
# UNIT TEST CASES
# ============================================================================

def test_inventory_unit():
    """Unit test cases for individual Inventory operations."""
    print("=== UNIT TEST CASES ===")

    # Test Case 1: Invalid quantity handling
    inv1 = Inventory()
    result1 = not inv1.add_item("Item", -5) and not inv1.add_item("Item", 0)
    print(f"Unit Test 1 - Reject invalid quantities: {result1}")
    assert result1, "Should reject negative and zero quantities on add"

    # Test Case 2: Remove from empty inventory
    inv2 = Inventory()
    result2 = not inv2.remove_item("NonExistent", 5)
    print(f"Unit Test 2 - Remove from empty inventory: {result2}")
    assert result2, "Should return False when removing from empty inventory"

    # Test Case 3: Update stock operation
    inv3 = Inventory()
    inv3.add_item("Widget", 10)
    success = inv3.update_stock("Widget", 15)
    result3 = success and inv3.get_stock("Widget") == 15
    print(f"Unit Test 3 - Update stock: {result3}")
    assert result3, "Should update stock correctly"

    print("✅ All unit tests passed!\n")


# ============================================================================
# PYTEST TEST CASES
# ============================================================================

def test_inventory_pytest_add_operations():
    """Pytest test case 1: Add operations."""
    inv = Inventory()
    assert inv.add_item("Item1", 5) == True
    assert inv.add_item("Item1", 3) == True  # Add more to existing
    assert inv.get_stock("Item1") == 8
    assert inv.add_item("Item2", 0) == False  # Invalid quantity

def test_inventory_pytest_remove_operations():
    """Pytest test case 2: Remove operations."""
    inv = Inventory()
    inv.add_item("Laptop", 5)
    assert inv.remove_item("Laptop", 2) == True
    assert inv.get_stock("Laptop") == 3
    assert inv.remove_item("Laptop", 10) == False  # Not enough stock
    assert inv.remove_item("NonExistent", 1) == False  # Doesn't exist

def test_inventory_pytest_availability_and_list():
    """Pytest test case 3: Availability and listing operations."""
    inv = Inventory()
    inv.add_item("Book", 7)
    inv.add_item("Pen", 20)
    assert inv.is_available("Book", 5) == True
    assert inv.is_available("Book", 10) == False
    assert inv.is_available("Missing", 1) == False
    assert len(inv.list_items()) == 2
```

```python
# ============================================================================
# ASSERTION TEST CASES
# ============================================================================

def test_inventory_assertions():
    """Assertion test cases for Inventory class."""
    print("\n=== ASSERTION TEST CASES ===")

    # Test Case 1: Complete stock management workflow
    inv1 = Inventory()
    assert inv1.add_item("Laptop", 5) == True, "Should add laptop successfully"
    assert inv1.get_stock("Laptop") == 5, "Should have 5 laptops"
    assert inv1.add_item("Laptop", 3) == True, "Should add more laptops"
    assert inv1.get_stock("Laptop") == 8, "Should have 8 laptops total"
    assert inv1.remove_item("Laptop", 2) == True, "Should remove 2 laptops"
    assert inv1.get_stock("Laptop") == 6, "Should have 6 laptops left"
    print("✅ Assertion Test 1 (Complete workflow): PASSED")

    # Test Case 2: Multiple items management
    inv2 = Inventory()
    assert inv2.add_item("Mouse", 10) == True, "Should add mice"
    assert inv2.add_item("Keyboard", 8) == True, "Should add keyboards"
    assert inv2.add_item("Monitor", 4) == True, "Should add monitors"
    assert inv2.is_available("Mouse", 5) == True, "Should have 5 mice available"
    assert inv2.is_available("Keyboard", 8) == True, "Should have 8 keyboards
available"
    assert inv2.is_available("Monitor", 5) == False, "Should not have 5 monitors"
    assert len(inv2.list_items()) == 3, "Should have 3 different items"
    print("✅ Assertion Test 2 (Multiple items): PASSED")

    # Test Case 3: Edge cases and error handling
    inv3 = Inventory()
    assert inv3.add_item("Phone", 5) == True, "Should add phones"
    assert inv3.remove_item("Phone", 5) == True, "Should remove all phones"
    assert inv3.get_stock("Phone") == 0, "Should have 0 phones (removed from dict)"
    assert inv3.update_stock("Tablet", 20) == True, "Should update new item"
    assert inv3.get_stock("Tablet") == 20, "Should have 20 tablets"
    assert inv3.update_stock("Tablet", 0) == True, "Should set to 0"
    assert inv3.get_stock("Tablet") == 0, "Should have 0 tablets"
    assert inv3.add_item("Item", -5) == False, "Should reject negative quantity"
    print("✅ Assertion Test 3 (Edge cases): PASSED")

    print("✅ All assertion tests passed!\n")


# Example Usage
if __name__ == "__main__":
    # Run all test suites
    test_inventory_general()
    test_inventory_unit()
    test_inventory_assertions()

    print("=" * 50)
    print("INTERACTIVE INVENTORY MANAGEMENT")
    print("=" * 50)

    inventory = Inventory()

    # Demo operations
    inventory.add_item("Apple", 10)
    inventory.add_item("Banana", 5)

    print(f"\nInitial inventory: {inventory.list_items()}")

    inventory.remove_item("Apple", 3)
    print(f"After removing 3 apples: {inventory.list_items()}")

    print(f"\nStock of Apple: {inventory.get_stock('Apple')}")
    print(f"Stock of Banana: {inventory.get_stock('Banana')}")
    print(f"Stock of Orange: {inventory.get_stock('Orange')}")

    print(f"\nIs 5 Apples available? {inventory.is_available('Apple', 5)}")
    print(f"Is 10 Bananas available? {inventory.is_available('Banana', 10)}")
```

Output:



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".

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.

Original Code:

```
from datetime import datetime

def validate_and_format(date_input):
    try:
        # Try to parse the date in MM/DD/YYYY format
        date_obj = datetime.strptime(date_input, "%m/%d/%Y")

        # Convert to YYYY-MM-DD format
        formatted_date = date_obj.strftime("%Y-%m-%d")

        return f"Valid date ✅\nFormatted date: {formatted_date}"

    except ValueError:
        return "Invalid date ❌ Please enter date in MM/DD/YYYY format."


# Main Program
date_input = input("Enter date (MM/DD/YYYY): ")

result = validate_and_format(date_input)
print(result)
```

AI Generated Code:

```python
from datetime import datetime

def validate_and_format_date(date_str):
    """
    Validates and formats a date string.

    Accepts multiple date formats:
    - MM/DD/YYYY
    - YYYY-MM-DD
    - DD-MM-YYYY
    - MM-DD-YYYY

    Returns a dictionary with:
    - 'valid': True if valid date, False otherwise
    - 'formatted': Formatted date in YYYY-MM-DD format (if valid)
    - 'message': Description or error message

    Args:
        date_str: Date string to validate

    Returns:
        Dictionary with validation results
    """

    if not date_str or not isinstance(date_str, str):
        return {
            'valid': False,
            'formatted': None,
            'message': 'Invalid input: must be a non-empty string'
        }

    # List of formats to try
    formats = ["%m/%d/%Y", "%Y-%m-%d", "%d-%m-%Y", "%m-%d-%Y"]

    for fmt in formats:
        try:
            date_obj = datetime.strptime(date_str.strip(), fmt)
            formatted_date = date_obj.strftime("%Y-%m-%d")

            return {
                'valid': True,
                'formatted': formatted_date,
                'message': f'Valid date in {fmt} format',
                'original_format': fmt
            }
        except ValueError:
            continue

    # If no format matched
    return {
        'valid': False,
        'formatted': None,
        'message': 'Invalid date format. Accepted formats: MM/DD/YYYY, YYYY-MM-DD, DD-MM-YYYY, MM-DD-
YYYY'
    }


# ============================================================================
# GENERAL TEST CASES
# ============================================================================

def test_validate_and_format_date_general():
    """General test cases for validate_and_format_date function."""
    print("\n=== GENERAL TEST CASES ===")

    # Test Case 1: Valid date in MM/DD/YYYY format
    result1 = validate_and_format_date("12/25/2023")
    print(f"Test 1 - Valid MM/DD/YYYY '12/25/2023': {result1['valid']} -> {result1['formatted']}")
    assert result1['valid'] == True and result1['formatted'] == "2023-12-25", "Should validate
MM/DD/YYYY format"

    # Test Case 2: Valid date in YYYY-MM-DD format
    result2 = validate_and_format_date("2024-01-15")
    print(f"Test 2 - Valid YYYY-MM-DD '2024-01-15': {result2['valid']} -> {result2['formatted']}")
    assert result2['valid'] == True and result2['formatted'] == "2024-01-15", "Should validate YYYY-MM-
DD format"

    # Test Case 3: Invalid date
    result3 = validate_and_format_date("13/32/2023")
    print(f"Test 3 - Invalid date '13/32/2023': {result3['valid']}")
    assert result3['valid'] == False, "Should reject invalid date"

    print("✅ All general tests passed!\n")
```

```python
# ============================================================================
# UNIT TEST CASES
# ============================================================================

def test_validate_and_format_date_unit():
    """Unit test cases for individual date validation features."""
    print("=== UNIT TEST CASES ===")

    # Test Case 1: Format conversion accuracy
    result1 = validate_and_format_date("01/01/2020")
    print(f"Unit Test 1 - Format conversion '01/01/2020': {result1['formatted']}")
    assert result1['formatted'] == "2020-01-01", "Should format correctly to YYYY-MM-DD"

    # Test Case 2: Leap year validation
    result2 = validate_and_format_date("02/29/2020")
    print(f"Unit Test 2 - Leap year '02/29/2020': {result2['valid']}")
    assert result2['valid'] == True, "Should accept valid leap year date"

    # Test Case 3: Invalid leap year
    result3 = validate_and_format_date("02/29/2021")
    print(f"Unit Test 3 - Invalid leap year '02/29/2021': {result3['valid']}")
    assert result3['valid'] == False, "Should reject invalid leap year date"

    print("✅ All unit tests passed!\n")


# ============================================================================
# PYTEST TEST CASES
# ============================================================================

def test_validate_and_format_date_pytest_valid_formats():
    """Pytest test case 1: Valid date formats."""
    assert validate_and_format_date("12/31/2023")['valid'] == True
    assert validate_and_format_date("2023-12-31")['valid'] == True
    assert validate_and_format_date("31-12-2023")['valid'] == True

def test_validate_and_format_date_pytest_formatting():
    """Pytest test case 2: Correct formatting output."""
    assert validate_and_format_date("06/15/2022")['formatted'] == "2022-06-15"
    assert validate_and_format_date("2021-03-10")['formatted'] == "2021-03-10"
    assert validate_and_format_date("25-05-2020")['formatted'] == "2020-05-25"

def test_validate_and_format_date_pytest_invalid_dates():
    """Pytest test case 3: Invalid and edge case dates."""
    assert validate_and_format_date("13/01/2023")['valid'] == False  # Invalid month
    assert validate_and_format_date("02/30/2023")['valid'] == False  # Invalid day
    assert validate_and_format_date("")['valid'] == False  # Empty string


# ============================================================================
# ASSERTION TEST CASES
# ============================================================================

def test_validate_and_format_date_assertions():
    """Assertion test cases for validate_and_format_date function."""
    print("\n=== ASSERTION TEST CASES ===")

    # Test Case 1: Multiple valid format conversions
    result1a = validate_and_format_date("03/17/2025")
    assert result1a['valid'] == True, "Should validate MM/DD/YYYY format"
    assert result1a['formatted'] == "2025-03-17", "Should format to YYYY-MM-DD"

    result1b = validate_and_format_date("2025-03-17")
    assert result1b['valid'] == True, "Should validate YYYY-MM-DD format"
    assert result1b['formatted'] == "2025-03-17", "Should keep YYYY-MM-DD format"

    result1c = validate_and_format_date("17-03-2025")
    assert result1c['valid'] == True, "Should validate DD-MM-YYYY format"
    assert result1c['formatted'] == "2025-03-17", "Should convert DD-MM-YYYY to YYYY-MM-DD"

    print("✅ Assertion Test 1 (Multiple valid formats): PASSED")

    # Test Case 2: Boundary and edge case dates
    result2a = validate_and_format_date("01/01/2000")
    assert result2a['valid'] == True, "Should accept 01/01/2000"
    assert result2a['formatted'] == "2000-01-01", "Should format correctly"

    result2b = validate_and_format_date("12/31/2099")
    assert result2b['valid'] == True, "Should accept 12/31/2099"
    assert result2b['formatted'] == "2099-12-31", "Should format correctly"

    result2c = validate_and_format_date("02/29/2000")
    assert result2c['valid'] == True, "Should accept leap day in leap year"
    assert result2c['formatted'] == "2000-02-29", "Should format leap day correctly"

    print("✅ Assertion Test 2 (Boundary dates): PASSED")

    # Test Case 3: Invalid dates and error handling
    result3a = validate_and_format_date("00/15/2023")
    assert result3a['valid'] == False, "Should reject month 00"

    result3b = validate_and_format_date("13/15/2023")
    assert result3b['valid'] == False, "Should reject month 13"

    result3c = validate_and_format_date("02/30/2023")
    assert result3c['valid'] == False, "Should reject Feb 30"

    result3d = validate_and_format_date("   ")
    assert result3d['valid'] == False, "Should reject whitespace only"

    result3e = validate_and_format_date(None)
    assert result3e['valid'] == False, "Should reject None input"

    print("✅ Assertion Test 3 (Invalid dates): PASSED")

    print("✅ All assertion tests passed!\n")
```

```python
# Example usage
if __name__ == "__main__":
    # Run all test suites
    test_validate_and_format_date_general()
    test_validate_and_format_date_unit()
    test_validate_and_format_date_assertions()

    print("=" * 60)
    print("INTERACTIVE DATE VALIDATOR & FORMATTER")
    print("=" * 60)
    print("Accepted formats: MM/DD/YYYY, YYYY-MM-DD, DD-MM-YYYY, MM-DD-YYYY")
    print("-" * 60)

    date_input = input("Enter a date: ")

    result = validate_and_format_date(date_input)

    print(f"\nValidation Result: {'✅ VALID' if result['valid'] else '❌ INVALID'}")
    if result['valid']:
        print(f"Original format: {result.get('original_format', 'Unknown')}")
        print(f"Formatted date: {result['formatted']}")
    print(f"Message: {result['message']}")
```

Output:

```python
# Example usage
if __name__ == "__main__":
    # Run all test suites
    test_validate_and_format_date_general()
    test_validate_and_format_date_unit()
    test_validate_and_format_date_assertions()

    print("=" * 60)
    print("INTERACTIVE DATE VALIDATOR & FORMATTER")
    print("=" * 60)
    print("Accepted formats: MM/DD/YYYY, YYYY-MM-DD, DD-MM-YYYY, MM-DD-YYYY")
    print("-" * 60)

    date_input = input("Enter a date: ")

    result = validate_and_format_date(date_input)

    print(f"\nValidation Result: {'✅ VALID' if result['valid'] else '❌ INVALID'}")
    if result['valid']:
        print(f"Original format: {result.get('original_format', 'Unknown')}")
        print(f"Formatted date: {result['formatted']}")
    print(f"Message: {result['message']}")
```