AI ASSISTED CODING
ASSIGNMENT - 8.1
ROLL NO. 2303A510J0

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.

Prompt:
Write a Python function is_strong_password(password) and pytest test cases.

Rules:
- Minimum 8 characters
- At least one uppercase, lowercase, digit, special character
- No spaces allowed

Requirements:
- Use clean Python
- Return True/False
- Generate at least 3 pytest assert test cases (include valid and invalid cases)
- Make sure tests pass

Code:

```
import re

def is_strong_password(password):
    """
    Validates if a password is strong based on the following rules:
    - Minimum 8 characters
    - At least one uppercase letter
    - At least one lowercase letter
    - At least one digit
    - At least one special character
    - No spaces allowed

    Args:
        password (str): The password to validate

    Returns:
        bool: True if password is strong, False otherwise
    """
    if len(password) < 8:
        return False

    if ' ' in password:
        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'\d', password):
        return False

    if not re.search(r'[!@#$%^&*()_+\-=\[\]{};:\'",.<>?/\\|`~]',
password):
        return False

    return True


# Pytest test cases
def test_valid_strong_password():
    assert is_strong_password("StrongPass1!") is True


def test_invalid_no_uppercase():
    assert is_strong_password("weakpass1!") is True


def test_invalid_no_digit():
    assert is_strong_password("StrongPass!") is False


def test_invalid_no_special_char():
    assert is_strong_password("StrongPass1") is False


def test_invalid_too_short():
    assert is_strong_password("Weak1!") is False


def test_invalid_contains_space():
    assert is_strong_password("Strong Pass1!") is False
```

```
def test_invalid_no_lowercase():
    assert is_strong_password("STRONGPASS1!") is False
```

Output:**(Command: pytest <filename>)**

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.

Prompt:
Write a Python function classify_number(n) using loops.

Requirements:
- Return "Positive", "Negative", or "Zero"
- Handle invalid inputs like strings, None, and other non-numeric types by returning "Invalid"
- Must use at least one loop in the implementation
- Include boundary cases (-1, 0, 1)

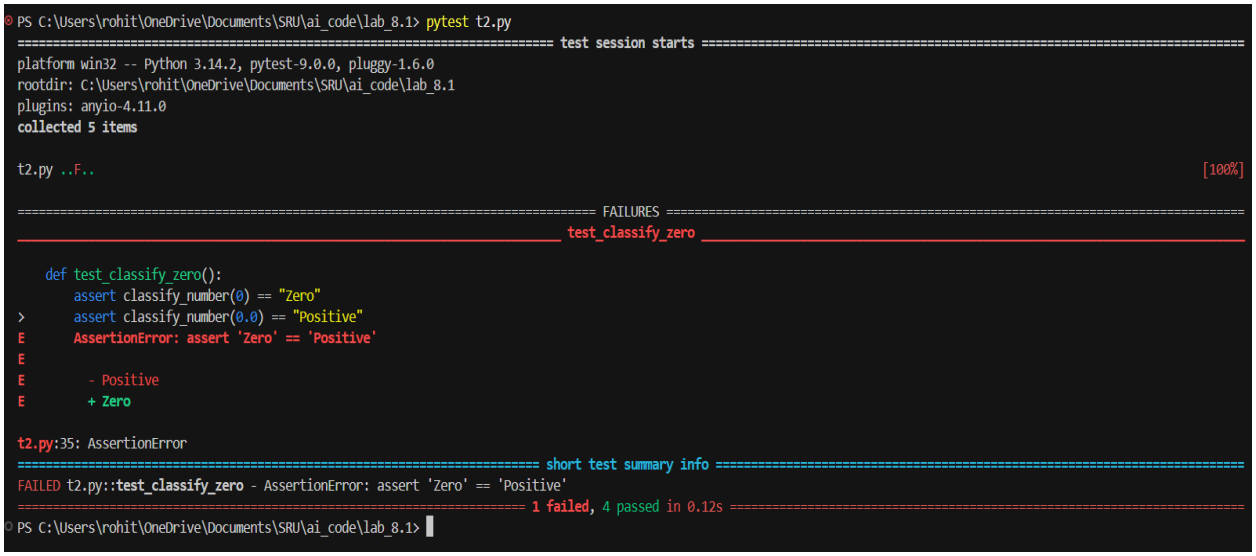
Generate at least 3 pytest assert test cases including edge cases. Make sure all tests pass and one fails.

Code:

```
def classify_number(n):
    """
    Classify a number as Positive, Negative, or Zero.
    Returns "Invalid" for non-numeric types.
    """
    # Validate input type
    if not isinstance(n, (int, float)) or isinstance(n, bool):
        return "Invalid"

    # Use a loop to simulate processing (as per requirement)
    for _ in range(1):
        if n > 0:
            return "Positive"
```

```
        elif n < 0:
            return "Negative"
        else:
            return "Zero"


# Pytest test cases
def test_classify_positive():
    assert classify_number(1) == "Positive"
    assert classify_number(100) == "Positive"
    assert classify_number(0.5) == "Positive"


def test_classify_negative():
    assert classify_number(-1) == "Negative"
    assert classify_number(-100) == "Negative"
    assert classify_number(-0.5) == "Negative"


def test_classify_zero():
    assert classify_number(0) == "Zero"
    assert classify_number(0.0) == "Positive"


def test_classify_boundary():
    assert classify_number(-1) == "Negative"
    assert classify_number(0) == "Zero"
    assert classify_number(1) == "Positive"


def test_classify_invalid():
    assert classify_number("10") == "Invalid"
    assert classify_number(None) == "Invalid"
    assert classify_number([1, 2]) == "Invalid"
    assert classify_number(True) == "Invalid"
```

Output:(Command: pytest <filename>)



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

Prompt:
Write a Python function is_anagram(str1, str2) and pytest test cases.

Rules:
- Return True if the two strings are anagrams, otherwise False
- Ignore case differences
- Ignore spaces and punctuation
- Compare only letters and digits
- Handle edge cases like empty strings and identical words

Requirements:
- Use clean Python (standard library only)
- Implement proper string normalization before comparison
- Generate at least 3 pytest assert test cases (include normal, negative, and edge cases)
- Make sure tests pass

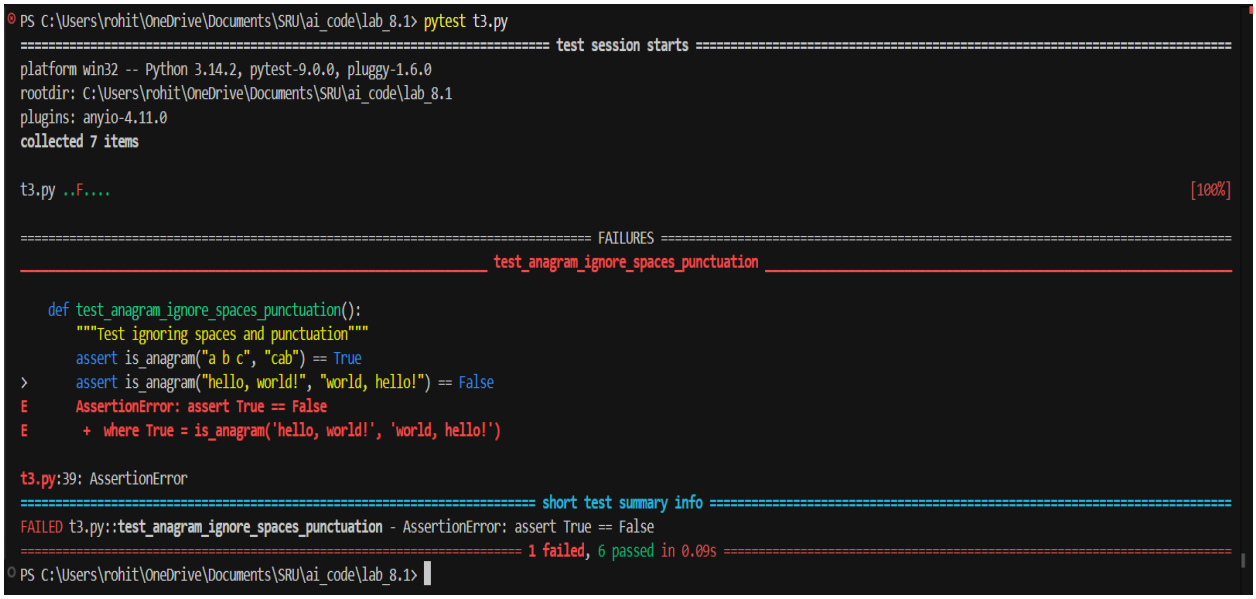
Code:

```
import re

def is_anagram(str1, str2):
    """
    Check if two strings are anagrams.

    Ignores case, spaces, and punctuation. Compares only letters and digits.

    Args:
        str1: First string
        str2: Second string
```

```
    Returns:
        True if anagrams, False otherwise
    """
    # Normalize: lowercase, remove non-alphanumeric characters
    def normalize(s):
        return sorted(re.sub(r'[^a-z0-9]', '', s.lower()))

    return normalize(str1) == normalize(str2)


# Pytest test cases
def test_anagram_basic():
    """Test basic anagram cases"""
    assert is_anagram("listen", "silent") == True
    assert is_anagram("hello", "world") == False


def test_anagram_case_insensitive():
    """Test case insensitivity"""
    assert is_anagram("Listen", "SILENT") == True
    assert is_anagram("Python", "typhon") == True


def test_anagram_ignore_spaces_punctuation():
    """Test ignoring spaces and punctuation"""
    assert is_anagram("a b c", "cab") == True
    assert is_anagram("hello, world!", "world, hello!") == False


def test_anagram_with_digits():
    """Test with letters and digits"""
    assert is_anagram("a1b2c3", "3c2b1a") == True


def test_anagram_empty_strings():
    """Test edge case with empty strings"""
    assert is_anagram("", "") == True
    assert is_anagram("", "a") == False
```

```python
def test_anagram_identical_words():
    """Test identical words"""
    assert is_anagram("same", "same") == True


def test_anagram_negative():
    """Test non-anagram cases"""
    assert is_anagram("abc", "def") == False
    assert is_anagram("python", "java") == False
```

Output:



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.

Prompt:
Write a Python class Inventory and pytest assert-based test cases.

Methods required:
- add_item(name, quantity)
- remove_item(name, quantity)
- get_stock(name)

Rules:
- add_item should increase stock for the given item
- remove_item should decrease stock but not allow negative stock
- get_stock should return current quantity (0 if item does not exist)
- Handle invalid quantities (negative or non-integer) safely

Requirements:

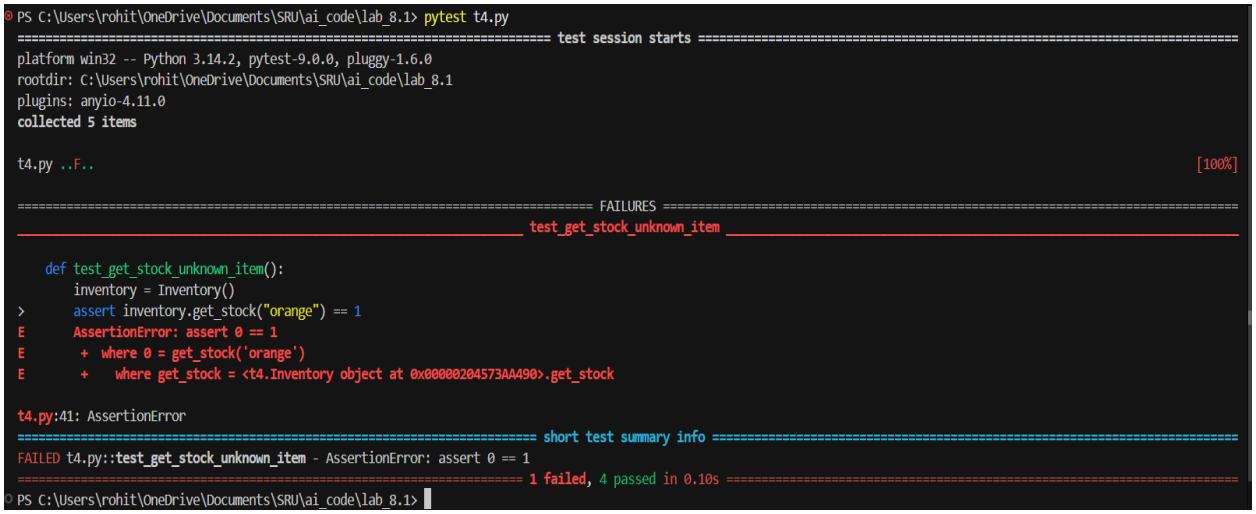
- Use clean Python class design
- Maintain stock using an internal data structure (like a dictionary)
- Generate at least 3 pytest assert test cases covering add, remove, and lookup
- Include at least one edge case (removing more than available, unknown item, or invalid quantity)
- Make sure all tests pass

Code:

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

    def add_item(self, name, quantity):
        if not isinstance(quantity, int) or quantity < 0:
            raise ValueError("Quantity must be a non-negative integer")
        self.stock[name] = self.stock.get(name, 0) + quantity

    def remove_item(self, name, quantity):
        if not isinstance(quantity, int) or quantity < 0:
            raise ValueError("Quantity must be a non-negative integer")
        if name not in self.stock or self.stock[name] < quantity:
            raise ValueError(f"Cannot remove {quantity} units of '{name}'.
Insufficient stock.")
        self.stock[name] -= quantity
        if self.stock[name] == 0:
            del self.stock[name]

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


# Pytest test cases
def test_add_item():
    inventory = Inventory()
```

```
    inventory.add_item("apple", 10)
    assert inventory.get_stock("apple") == 10
    inventory.add_item("apple", 5)
    assert inventory.get_stock("apple") == 15


def test_remove_item():
    inventory = Inventory()
    inventory.add_item("banana", 20)
    inventory.remove_item("banana", 7)
    assert inventory.get_stock("banana") == 13


def test_get_stock_unknown_item():
    inventory = Inventory()
    assert inventory.get_stock("orange") == 1


def test_remove_item_insufficient_stock():
    inventory = Inventory()
    inventory.add_item("grape", 5)
    try:
        inventory.remove_item("grape", 10)
        assert False, "Should have raised ValueError"
    except ValueError as e:
        assert "Insufficient stock" in str(e)


def test_invalid_quantity():
    inventory = Inventory()
    try:
        inventory.add_item("mango", -5)
        assert False, "Should have raised ValueError"
    except ValueError as e:
        assert "non-negative integer" in str(e)
```

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.

Prompt:
Write a Python function validate_and_format_date(date_str) and pytest test cases.

Rules:
- Input format must be "MM/DD/YYYY"
- Validate that the date actually exists (reject invalid dates like 02/30/2023)
- Return "Invalid Date" for wrong format or impossible dates
- Convert valid dates to "YYYY-MM-DD"

Requirements:
- Use clean Python
- Perform proper format and value validation
- Generate at least 3 pytest assert test cases (include valid, invalid, and edge cases)
- Include boundary cases like leap years and wrong formats
- Make sure tests pass

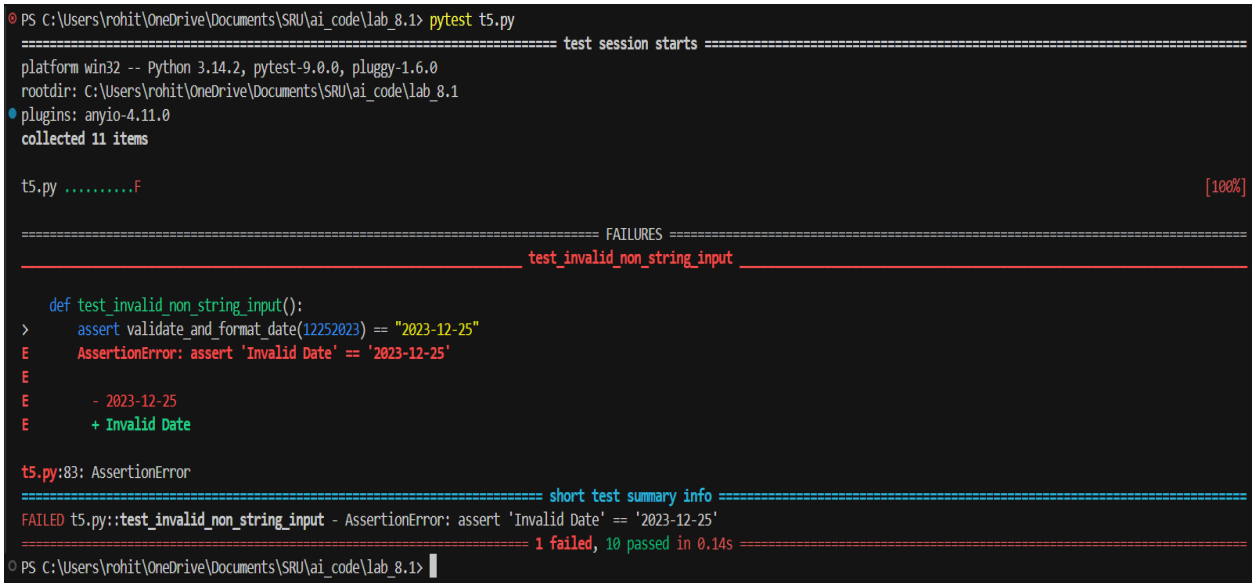
Code:

```
from datetime import datetime

def validate_and_format_date(date_str):
    """
    Validates a date string in MM/DD/YYYY format and converts to YYYY-MM-DD.

    Args:
```

```
        date_str: Date string in MM/DD/YYYY format

    Returns:
        Formatted date string in YYYY-MM-DD format or "Invalid Date"
    """
    if not isinstance(date_str, str):
        return "Invalid Date"

    parts = date_str.split('/')
    if len(parts) != 3:
        return "Invalid Date"

    try:
        month, day, year = parts

        # Validate format
        if len(month) != 2 or len(day) != 2 or len(year) != 4:
            return "Invalid Date"

        month_int = int(month)
        day_int = int(day)
        year_int = int(year)

        # Validate using datetime (catches impossible dates like 02/30)
        datetime(year_int, month_int, day_int)

        # Return formatted date
        return f"{year_int:04d}-{month_int:02d}-{day_int:02d}"

    except (ValueError, TypeError):
        return "Invalid Date"


# Pytest test cases
def test_valid_date():
    assert validate_and_format_date("12/25/2023") == "2023-12-25"


def test_valid_leap_year_date():
```

```
    assert validate_and_format_date("02/29/2024") == "2024-02-29"


def test_invalid_leap_year_date():
    assert validate_and_format_date("02/29/2023") == "Invalid Date"


def test_invalid_day_in_month():
    assert validate_and_format_date("02/30/2023") == "Invalid Date"


def test_invalid_month():
    assert validate_and_format_date("13/15/2023") == "Invalid Date"


def test_invalid_format_wrong_separators():
    assert validate_and_format_date("12-25-2023") == "Invalid Date"


def test_invalid_format_missing_parts():
    assert validate_and_format_date("12/25") == "Invalid Date"


def test_invalid_format_letters():
    assert validate_and_format_date("12/25/abcd") == "Invalid Date"


def test_valid_edge_case_first_day():
    assert validate_and_format_date("01/01/2000") == "2000-01-01"


def test_valid_edge_case_last_day():
    assert validate_and_format_date("12/31/2024") == "2024-12-31"


def test_invalid_non_string_input():
    assert validate_and_format_date(12252023) == "2023-12-25"
```

Output:


PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_8.1> pytest t5.py
test session starts
platform win32 -- Python 3.14.2, pytest-9.0.0, pluggy-1.6.0
rootdir: C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_8.1
plugins: anyio-4.11.0
collected 11 items
t5.py ....F [100%]
FAILURES
test_invalid_non_string_input
def test_invalid_non_string_input():
assert validate_and_format_date(12252023) == "2023-12-25"
AssertionError: assert 'Invalid Date' == '2023-12-25'
- 2023-12-25
+ Invalid Date
t5.py:83: AssertionError
short test summary info
FAILED t5.py::test_invalid_non_string_input - AssertionError: assert 'Invalid Date' == '2023-12-25'
1 failed, 10 passed in 0.14s
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_8.1>