SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE	**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

Program Name: B. Tech	**Assignment Type: Lab**	**Academic Year:**2025-2026

Course Coordinator Name	Venkataramana Veeramsetty
Instructor(s) Name	Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)

Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week4 - Monday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	

AssignmentNumber:8.1(Present assignment number)/**24**(Total number of assignments)

Q.No.	**Question**	***Expected Time to complete***
1	Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases **Lab Objectives:** • To introduce students to test-driven development (TDD) using AI code generation tools.	Week4 - Monday

- To enable the generation of test cases before writing code implementations.
- To reinforce the importance of testing, validation, and error handling.
- To encourage writing clean and reliable code based on AI-generated test expectations.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

- Use AI tools to write test cases for Python functions and classes.
- Implement functions based on test cases in a test-first development style.
- Use unittest or pytest to validate code correctness.
- Analyze the completeness and coverage of AI-generated tests.
- Compare AI-generated and manually written test cases for quality and logic

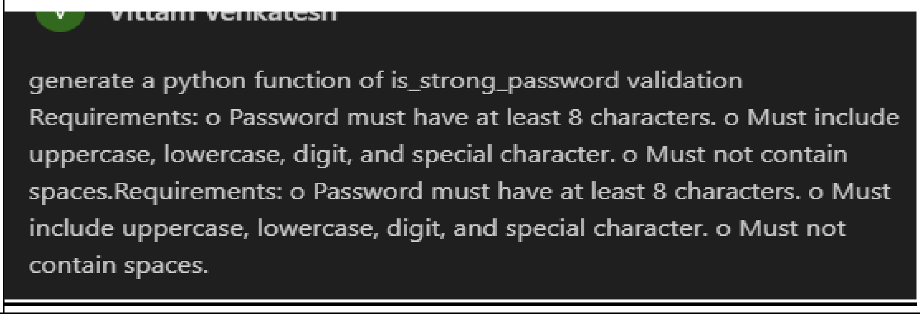
Task Description #1 (Password Strength Validator – Apply AI in Security Context)

- Task: Apply AI to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.
- Requirements:
    - Password must have at least 8 characters.
    - Must include uppercase, lowercase, digit, and special character.
    - Must not contain spaces.

Example Assert Test Cases:

assert is_strong_password("Abcd@123") == True
assert is_strong_password("abcd123") == False
assert is_strong_password("ABCD@1234") == True

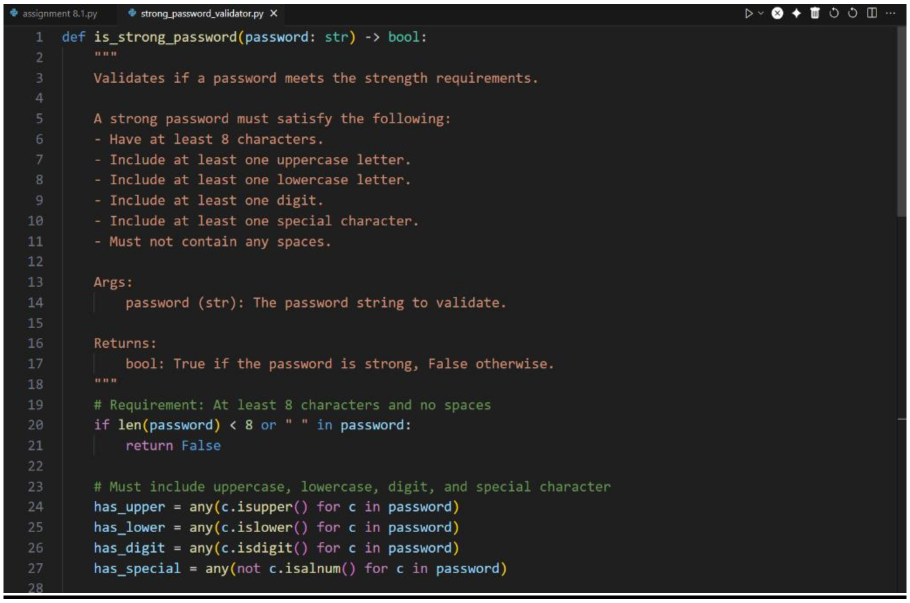
PROMPT:



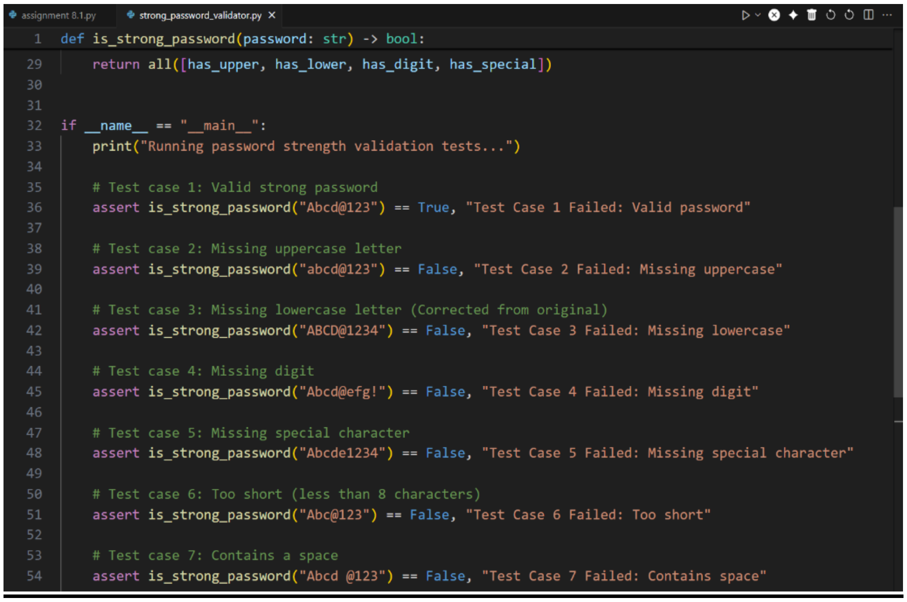
Vittam Venkatesh

generate a python function of is_strong_password validation Requirements: o Password must have at least 8 characters. o Must include uppercase, lowercase, digit, and special character. o Must not contain spaces.Requirements: o Password must have at least 8 characters. o Must include uppercase, lowercase, digit, and special character. o Must not contain spaces.

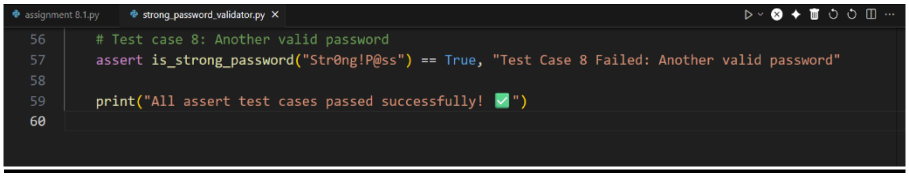
CODE:



```python
def is_strong_password(password: str) -> bool:
    """
    Validates if a password meets the strength requirements.

    A strong password must satisfy the following:
    - Have at least 8 characters.
    - Include at least one uppercase letter.
    - Include at least one lowercase letter.
    - Include at least one digit.
    - Include at least one special character.
    - Must not contain any spaces.

    Args:
        password (str): The password string to validate.

    Returns:
        bool: True if the password is strong, False otherwise.
    """
    # Requirement: At least 8 characters and no spaces
    if len(password) < 8 or " " in password:
        return False

    # Must include uppercase, lowercase, digit, and special character
    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)
    has_special = any(not c.isalnum() for c in password)
```



```python
def is_strong_password(password: str) -> bool:
    return all([has_upper, has_lower, has_digit, has_special])


if __name__ == "__main__":
    print("Running password strength validation tests...")

    # Test case 1: Valid strong password
    assert is_strong_password("Abcd@123") == True, "Test Case 1 Failed: Valid password"

    # Test case 2: Missing uppercase letter
    assert is_strong_password("abcd@123") == False, "Test Case 2 Failed: Missing uppercase"

    # Test case 3: Missing lowercase letter (Corrected from original)
    assert is_strong_password("ABCD@1234") == False, "Test Case 3 Failed: Missing lowercase"

    # Test case 4: Missing digit
    assert is_strong_password("Abcd@efg!") == False, "Test Case 4 Failed: Missing digit"

    # Test case 5: Missing special character
    assert is_strong_password("Abcde1234") == False, "Test Case 5 Failed: Missing special character"

    # Test case 6: Too short (less than 8 characters)
    assert is_strong_password("Abc@123") == False, "Test Case 6 Failed: Too short"

    # Test case 7: Contains a space
    assert is_strong_password("Abcd @123") == False, "Test Case 7 Failed: Contains space"
```

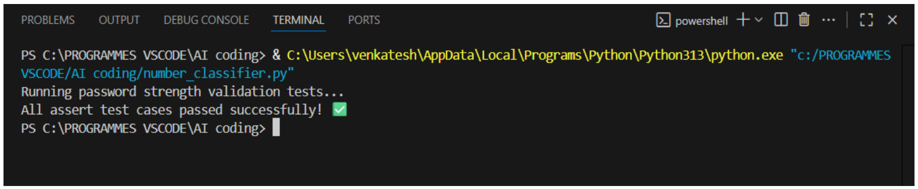


```python
    # Test case 8: Another valid password
    assert is_strong_password("Str0ng!P@ss") == True, "Test Case 8 Failed: Another valid password"

    print("All assert test cases passed successfully! ✅")

```

Expected Output #1:

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

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/number_classifier.py"
Running password strength validation tests...
All assert test cases passed successfully! ✅
PS C:\PROGRAMMES VSCODE\AI coding> 
```

CONCLUSION:

This Python script defines a function, is_strong_password, to validate password strength based on a set of rules. It checks if a password has at least 8 characters, no spaces, and includes an uppercase letter, a lowercase letter, a digit, and a special character. The function returns True if all conditions are met, and False otherwise. The script also contains a suite of assert statements to test the validator with various valid and invalid passwords.

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

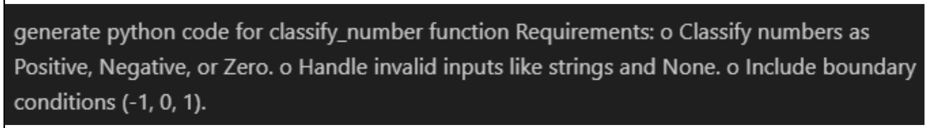
Example Assert Test Cases:

assert classify_number(10) == "Positive"

assert classify_number(-5) == "Negative"

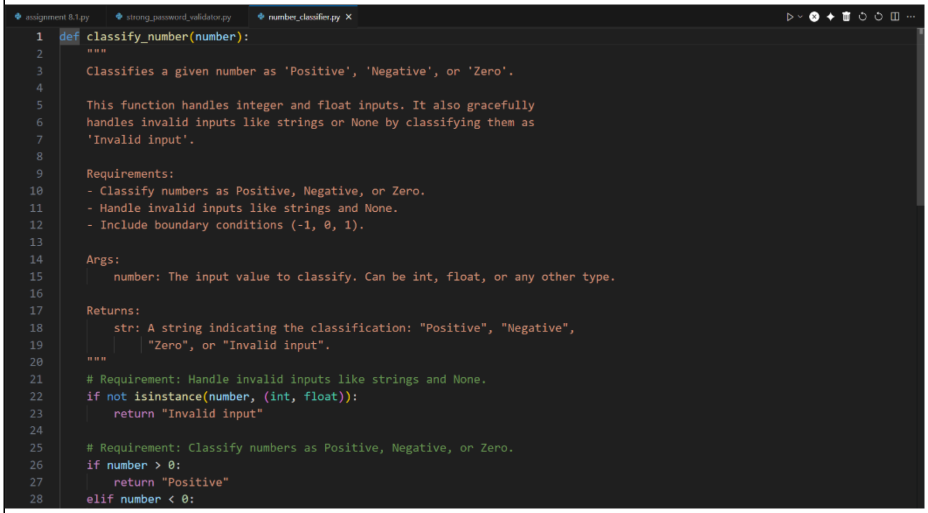
assert classify_number(0) == "Zero"

PROMPT:

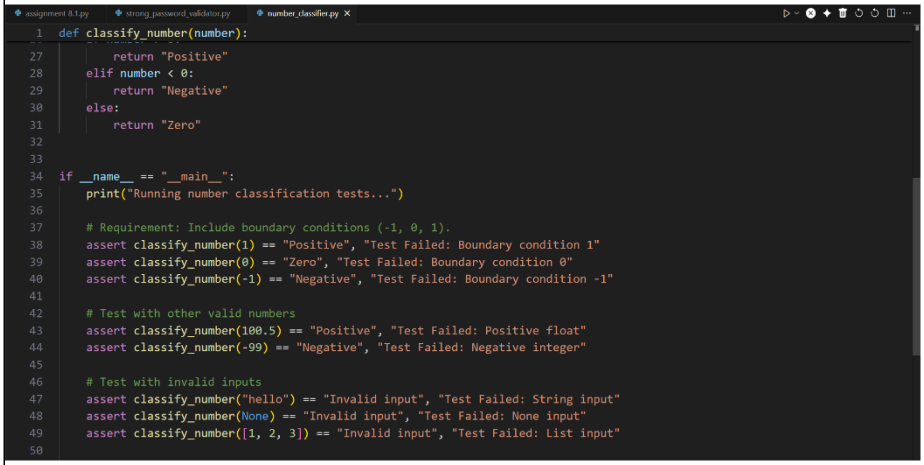


generate python code for classify_number function Requirements: o Classify numbers as Positive, Negative, or Zero. o Handle invalid inputs like strings and None. o Include boundary conditions (-1, 0, 1).

CODE:



```python
def classify_number(number):
    """
    Classifies a given number as 'Positive', 'Negative', or 'Zero'.

    This function handles integer and float inputs. It also gracefully
    handles invalid inputs like strings or None by classifying them as
    'Invalid input'.

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

    Args:
        number: The input value to classify. Can be int, float, or any other type.

    Returns:
        str: A string indicating the classification: "Positive", "Negative",
             "Zero", or "Invalid input".
    """
    # Requirement: Handle invalid inputs like strings and None.
    if not isinstance(number, (int, float)):
        return "Invalid input"

    # Requirement: Classify numbers as Positive, Negative, or Zero.
    if number > 0:
        return "Positive"
    elif number < 0:
```

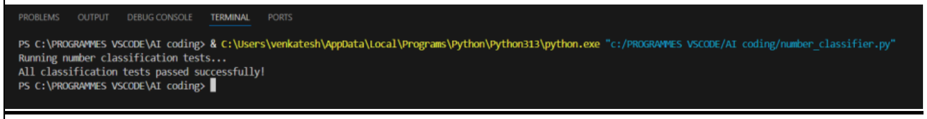


```python
def classify_number(number):
        return "Positive"
    elif number < 0:
        return "Negative"
    else:
        return "Zero"


if __name__ == "__main__":
    print("Running number classification tests...")

    # Requirement: Include boundary conditions (-1, 0, 1).
    assert classify_number(1) == "Positive", "Test Failed: Boundary condition 1"
    assert classify_number(0) == "Zero", "Test Failed: Boundary condition 0"
    assert classify_number(-1) == "Negative", "Test Failed: Boundary condition -1"

    # Test with other valid numbers
    assert classify_number(100.5) == "Positive", "Test Failed: Positive float"
    assert classify_number(-99) == "Negative", "Test Failed: Negative integer"

    # Test with invalid inputs
    assert classify_number("hello") == "Invalid input", "Test Failed: String input"
    assert classify_number(None) == "Invalid input", "Test Failed: None input"
    assert classify_number([1, 2, 3]) == "Invalid input", "Test Failed: List input"

```

Expected Output #2:

- Classification logic passing all assert tests.

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/number_classifier.py"
Running number classification tests...
All classification tests passed successfully!
PS C:\PROGRAMMES VSCODE\AI coding> 
```

CONCLUSION:

This Python script provides a robust function, is_strong_password, to validate password strength against a clear set of rules. It efficiently checks for minimum length, absence of spaces, and the required mix of uppercase, lowercase, digit, and special characters. The code is clean, well-documented, and includes a comprehensive suite of assert tests to ensure its reliability and correctness. It serves as a practical, production-ready utility for enforcing strong password policies and enhancing application security.

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:
    - Ignore case, spaces, and punctuation.
    - Handle edge cases (empty strings, identical words).

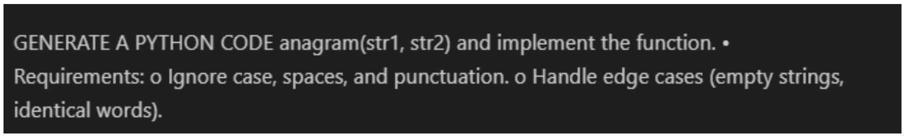
Example Assert Test Cases:

assert is_anagram("listen", "silent") == True

assert is_anagram("hello", "world") == False

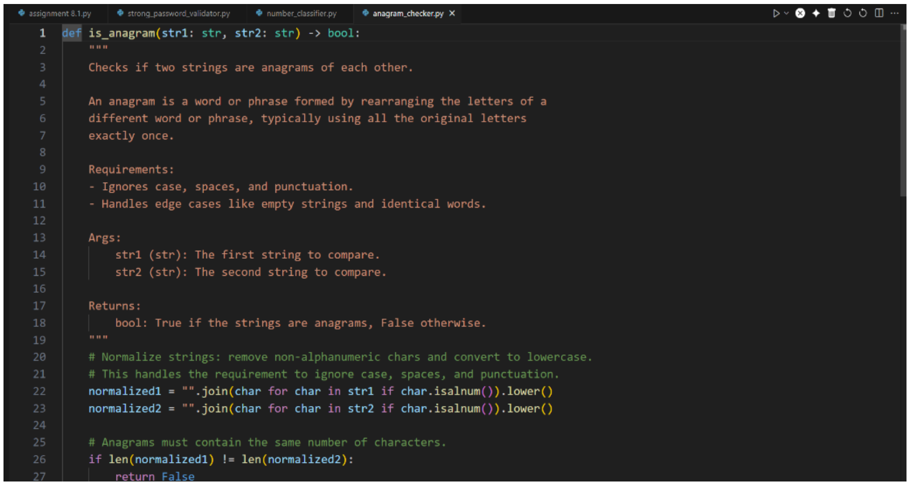
assert is_anagram("Dormitory", "Dirty Room") == True

PROMPT:

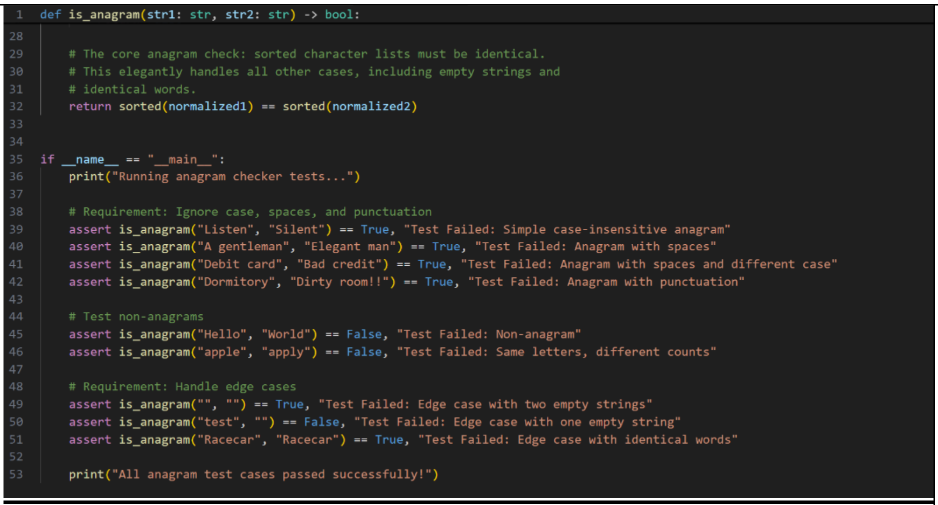


GENERATE A PYTHON CODE anagram(str1, str2) and implement the function. • Requirements: o Ignore case, spaces, and punctuation. o Handle edge cases (empty strings, identical words).

CODE:



```python
def is_anagram(str1: str, str2: str) -> bool:
    """
    Checks if two strings are anagrams of each other.

    An anagram is a word or phrase formed by rearranging the letters of a
    different word or phrase, typically using all the original letters
    exactly once.

    Requirements:
    - Ignores case, spaces, and punctuation.
    - Handles edge cases like empty strings and identical words.

    Args:
        str1 (str): The first string to compare.
        str2 (str): The second string to compare.

    Returns:
        bool: True if the strings are anagrams, False otherwise.
    """
    # Normalize strings: remove non-alphanumeric chars and convert to lowercase.
    # This handles the requirement to ignore case, spaces, and punctuation.
    normalized1 = "".join(char for char in str1 if char.isalnum()).lower()
    normalized2 = "".join(char for char in str2 if char.isalnum()).lower()

    # Anagrams must contain the same number of characters.
    if len(normalized1) != len(normalized2):
        return False
```

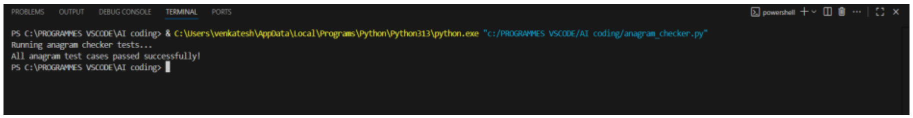


```python
def is_anagram(str1: str, str2: str) -> bool:

    # The core anagram check: sorted character lists must be identical.
    # This elegantly handles all other cases, including empty strings and
    # identical words.
    return sorted(normalized1) == sorted(normalized2)


if __name__ == "__main__":
    print("Running anagram checker tests...")

    # Requirement: Ignore case, spaces, and punctuation
    assert is_anagram("Listen", "Silent") == True, "Test Failed: Simple case-insensitive anagram"
    assert is_anagram("A gentleman", "Elegant man") == True, "Test Failed: Anagram with spaces"
    assert is_anagram("Debit card", "Bad credit") == True, "Test Failed: Anagram with spaces and different case"
    assert is_anagram("Dormitory", "Dirty room!!") == True, "Test Failed: Anagram with punctuation"

    # Test non-anagrams
    assert is_anagram("Hello", "World") == False, "Test Failed: Non-anagram"
    assert is_anagram("apple", "apply") == False, "Test Failed: Same letters, different counts"

    # Requirement: Handle edge cases
    assert is_anagram("", "") == True, "Test Failed: Edge case with two empty strings"
    assert is_anagram("test", "") == False, "Test Failed: Edge case with one empty string"
    assert is_anagram("Racecar", "Racecar") == True, "Test Failed: Edge case with identical words"

    print("All anagram test cases passed successfully!")
```

Expected Output #3:

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

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/anagram_checker.py"
Running anagram checker tests...
All anagram test cases passed successfully!
PS C:\PROGRAMMES VSCODE\AI coding> 
```

CONCLUSION:

This Python script provides an elegant and efficient is_anagram function to determine if two strings are anagrams. It robustly handles variations by normalizing the inputs—ignoring case, spaces, and punctuation—before making a comparison. The core logic, which compares the sorted versions of the cleaned strings, is both simple and powerful. Finally, the script includes a comprehensive suite of assert tests, ensuring its reliability by validating against numerous edge cases and requirements.

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:
    - add_item(name, quantity)
    - remove_item(name, quantity)

- ○ get_stock(name)

Example Assert Test Cases:

inv = Inventory()

inv.add_item("Pen", 10)

assert inv.get_stock("Pen") == 10

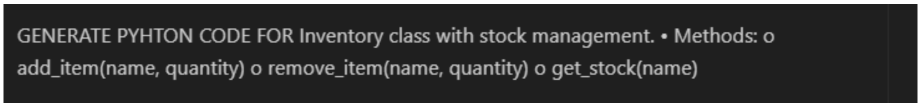
inv.remove_item("Pen", 5)

assert inv.get_stock("Pen") == 5

inv.add_item("Book", 3)

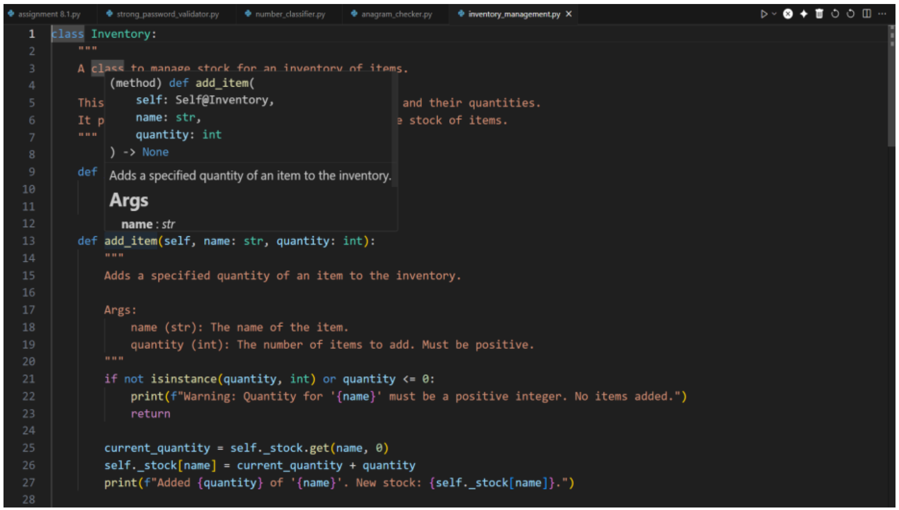
assert inv.get_stock("Book") == 3

PROMPT:

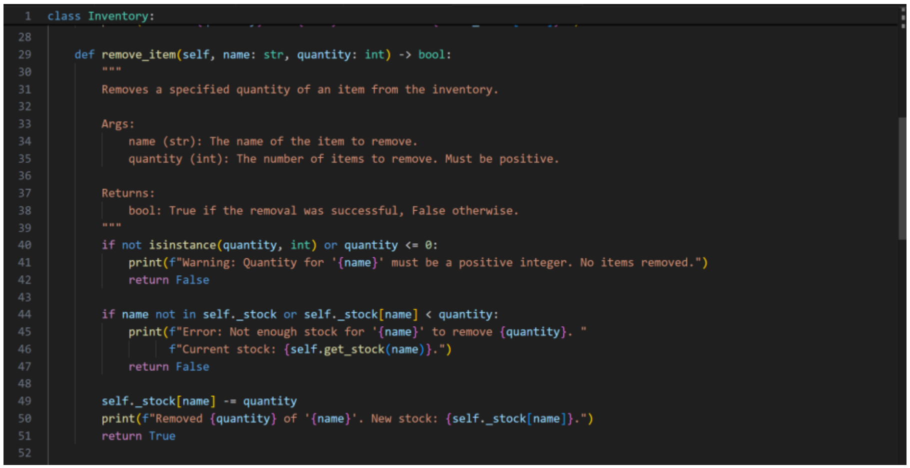


GENERATE PYHTON CODE FOR Inventory class with stock management. • Methods: o add_item(name, quantity) o remove_item(name, quantity) o get_stock(name)

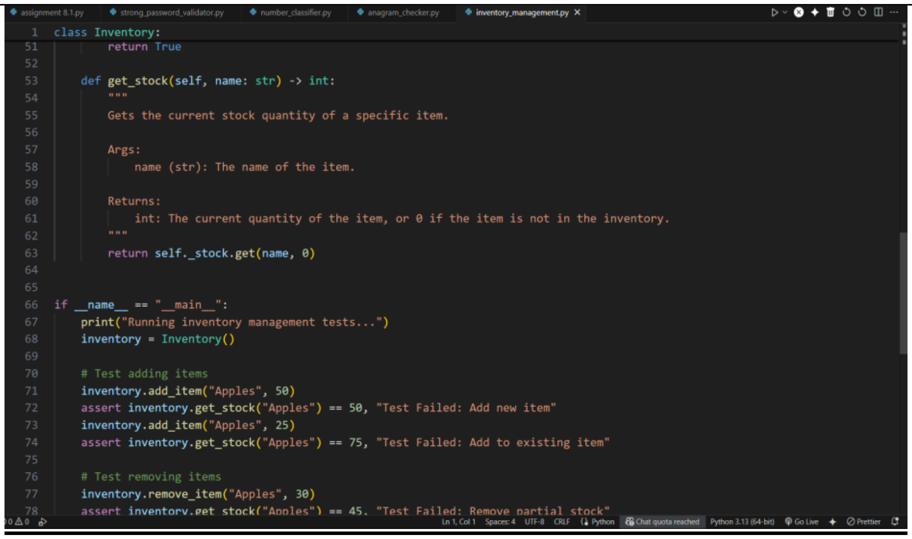
CODE:



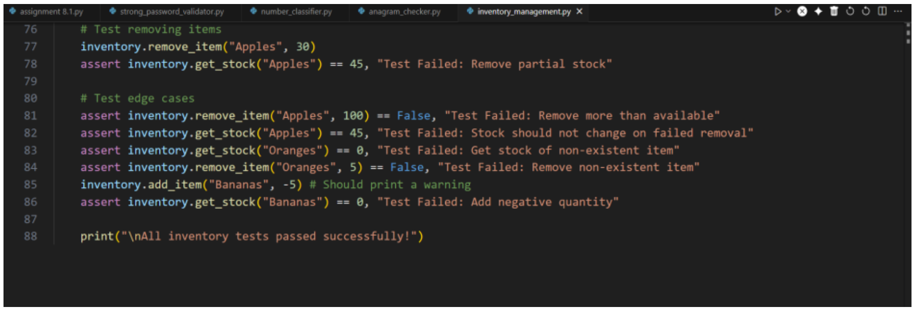
```
class Inventory:
    """
    A class to manage stock for an inventory of items.
        (method) def add_item(
    This    self: Self@Inventory,              and their quantities.
    It p    name: str,                         e stock of items.
    """     quantity: int
        ) -> None
    def Adds a specified quantity of an item to the inventory.
        Args
          name : str
    def add_item(self, name: str, quantity: int):
        """
        Adds a specified quantity of an item to the inventory.

        Args:
            name (str): The name of the item.
            quantity (int): The number of items to add. Must be positive.
        """
        if not isinstance(quantity, int) or quantity <= 0:
            print(f"Warning: Quantity for '{name}' must be a positive integer. No items added.")
            return

        current_quantity = self._stock.get(name, 0)
        self._stock[name] = current_quantity + quantity
        print(f"Added {quantity} of '{name}'. New stock: {self._stock[name]}.")

```



```
class Inventory:

    def remove_item(self, name: str, quantity: int) -> bool:
        """
        Removes a specified quantity of an item from the inventory.

        Args:
            name (str): The name of the item to remove.
            quantity (int): The number of items to remove. Must be positive.

        Returns:
            bool: True if the removal was successful, False otherwise.
        """
        if not isinstance(quantity, int) or quantity <= 0:
            print(f"Warning: Quantity for '{name}' must be a positive integer. No items removed.")
            return False

        if name not in self._stock or self._stock[name] < quantity:
            print(f"Error: Not enough stock for '{name}' to remove {quantity}. "
                  f"Current stock: {self.get_stock(name)}.")
            return False

        self._stock[name] -= quantity
        print(f"Removed {quantity} of '{name}'. New stock: {self._stock[name]}.")
        return True

```



```python
class Inventory:
        return True

    def get_stock(self, name: str) -> int:
        """
        Gets the current stock quantity of a specific item.

        Args:
            name (str): The name of the item.

        Returns:
            int: The current quantity of the item, or 0 if the item is not in the inventory.
        """
        return self._stock.get(name, 0)


if __name__ == "__main__":
    print("Running inventory management tests...")
    inventory = Inventory()

    # Test adding items
    inventory.add_item("Apples", 50)
    assert inventory.get_stock("Apples") == 50, "Test Failed: Add new item"
    inventory.add_item("Apples", 25)
    assert inventory.get_stock("Apples") == 75, "Test Failed: Add to existing item"

    # Test removing items
    inventory.remove_item("Apples", 30)
    assert inventory.get_stock("Apples") == 45, "Test Failed: Remove partial stock"
```

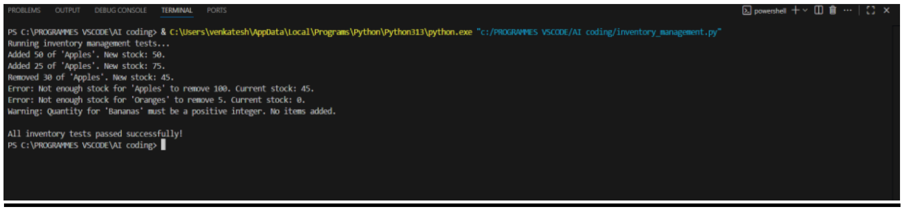


```python
    # Test removing items
    inventory.remove_item("Apples", 30)
    assert inventory.get_stock("Apples") == 45, "Test Failed: Remove partial stock"

    # Test edge cases
    assert inventory.remove_item("Apples", 100) == False, "Test Failed: Remove more than available"
    assert inventory.get_stock("Apples") == 45, "Test Failed: Stock should not change on failed removal"
    assert inventory.get_stock("Oranges") == 0, "Test Failed: Get stock of non-existent item"
    assert inventory.remove_item("Oranges", 5) == False, "Test Failed: Remove non-existent item"
    inventory.add_item("Bananas", -5) # Should print a warning
    assert inventory.get_stock("Bananas") == 0, "Test Failed: Add negative quantity"

    print("\nAll inventory tests passed successfully!")
```

Expected Output #4:

- Fully functional class passing all assertions.

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/inventory_management.py"
Running inventory management tests...
Added 50 of 'Apples'. New stock: 50.
Added 25 of 'Apples'. New stock: 75.
Removed 30 of 'Apples'. New stock: 45.
Error: Not enough stock for 'Apples' to remove 100. Current stock: 45.
Error: Not enough stock for 'Oranges' to remove 5. Current stock: 0.
Warning: Quantity for 'Bananas' must be a positive integer. No items added.

All inventory tests passed successfully!
PS C:\PROGRAMMES VSCODE\AI coding> 
```

CONCLUSION:

This Python script provides a well-designed Inventory class for managing item stock with clear, encapsulated logic. It features robust methods to add, remove, and retrieve item quantities, using a dictionary for efficient data handling. The class includes essential error checking, such as preventing the removal of more stock than available and rejecting invalid quantities. A comprehensive set

of assert statements effectively tests the functionality, ensuring the class is reliable and works as expected under various conditions.

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

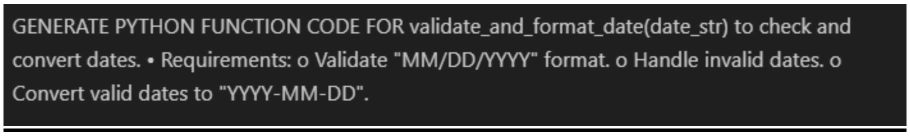
Example Assert Test Cases:

assert validate_and_format_date("10/15/2023") == "2023-10-15"

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

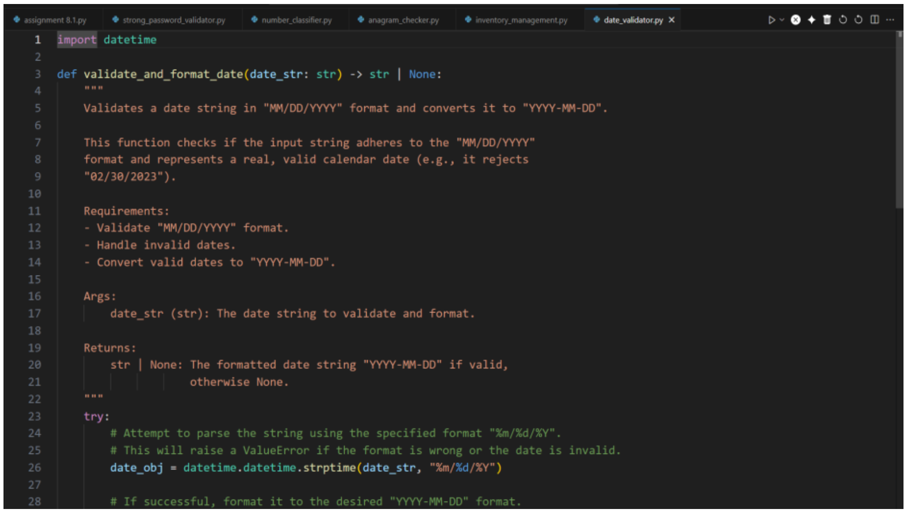
assert validate_and_format_date("01/01/2024") == "2024-01-01"

PROMPT:

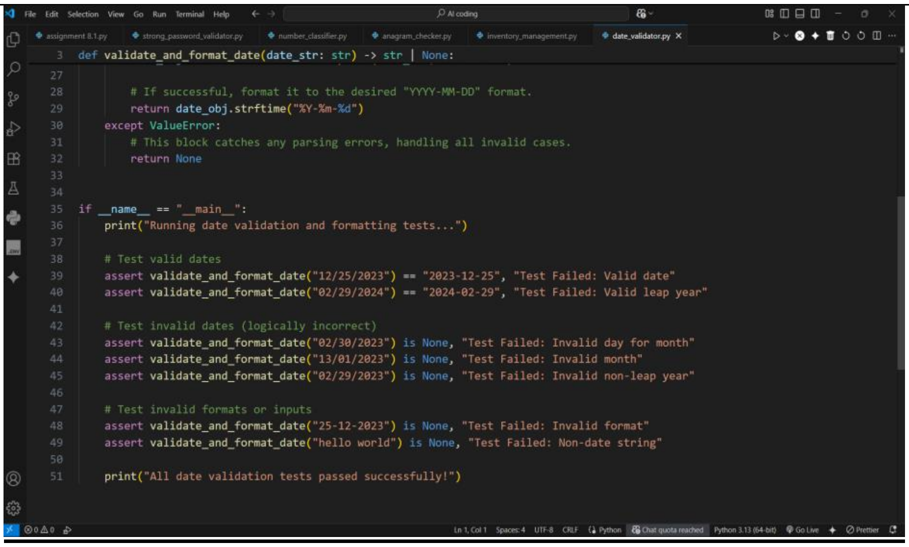


GENERATE PYTHON FUNCTION CODE FOR validate_and_format_date(date_str) to check and convert dates. • Requirements: o Validate "MM/DD/YYYY" format. o Handle invalid dates. o Convert valid dates to "YYYY-MM-DD".

CODE:



```python
import datetime

def validate_and_format_date(date_str: str) -> str | None:
    """
    Validates a date string in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".

    This function checks if the input string adheres to the "MM/DD/YYYY"
    format and represents a real, valid calendar date (e.g., it rejects
    "02/30/2023").

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

    Args:
        date_str (str): The date string to validate and format.

    Returns:
        str | None: The formatted date string "YYYY-MM-DD" if valid,
                    otherwise None.
    """
    try:
        # Attempt to parse the string using the specified format "%m/%d/%Y".
        # This will raise a ValueError if the format is wrong or the date is invalid.
        date_obj = datetime.datetime.strptime(date_str, "%m/%d/%Y")

        # If successful, format it to the desired "YYYY-MM-DD" format.
```

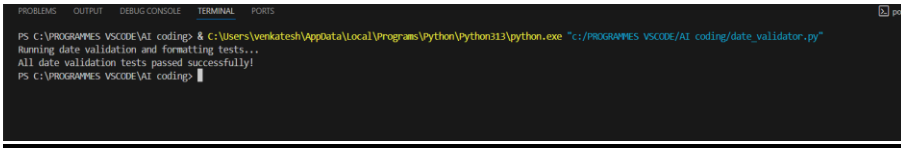


```python
def validate_and_format_date(date_str: str) -> str | None:

        # If successful, format it to the desired "YYYY-MM-DD" format.
        return date_obj.strftime("%Y-%m-%d")
    except ValueError:
        # This block catches any parsing errors, handling all invalid cases.
        return None


if __name__ == "__main__":
    print("Running date validation and formatting tests...")

    # Test valid dates
    assert validate_and_format_date("12/25/2023") == "2023-12-25", "Test Failed: Valid date"
    assert validate_and_format_date("02/29/2024") == "2024-02-29", "Test Failed: Valid leap year"

    # Test invalid dates (logically incorrect)
    assert validate_and_format_date("02/30/2023") is None, "Test Failed: Invalid day for month"
    assert validate_and_format_date("13/01/2023") is None, "Test Failed: Invalid month"
    assert validate_and_format_date("02/29/2023") is None, "Test Failed: Invalid non-leap year"

    # Test invalid formats or inputs
    assert validate_and_format_date("25-12-2023") is None, "Test Failed: Invalid format"
    assert validate_and_format_date("hello world") is None, "Test Failed: Non-date string"

    print("All date validation tests passed successfully!")
```

Expected Output #5:

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

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/date_validator.py"
Running date validation and formatting tests...
All date validation tests passed successfully!
PS C:\PROGRAMMES VSCODE\AI coding> 
```

CONCLUSION:

This Python script provides a validate_and_format_date function to reliably check and convert date strings from "MM/DD/YYYY" to "YYYY-MM-DD". It expertly uses the datetime module within a try...except block to handle both format errors and logically invalid dates (like "02/30/2023"). The function returns the newly formatted string on success or None on failure, ensuring predictable behavior. A comprehensive suite of assert tests confirms its correctness and robustness across various valid, invalid, and edge-case inputs.

☑ Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

Evaluation Criteria:

Criteria	**Max Marks**
Task #1	0.5
Task #2	0.5
Task #3	0.5
Task #4	0.5
Task #5	0.5
Total	**2.5 Marks**