AI Assisted Coding

Assignment:8.1

TASK-1:

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.**

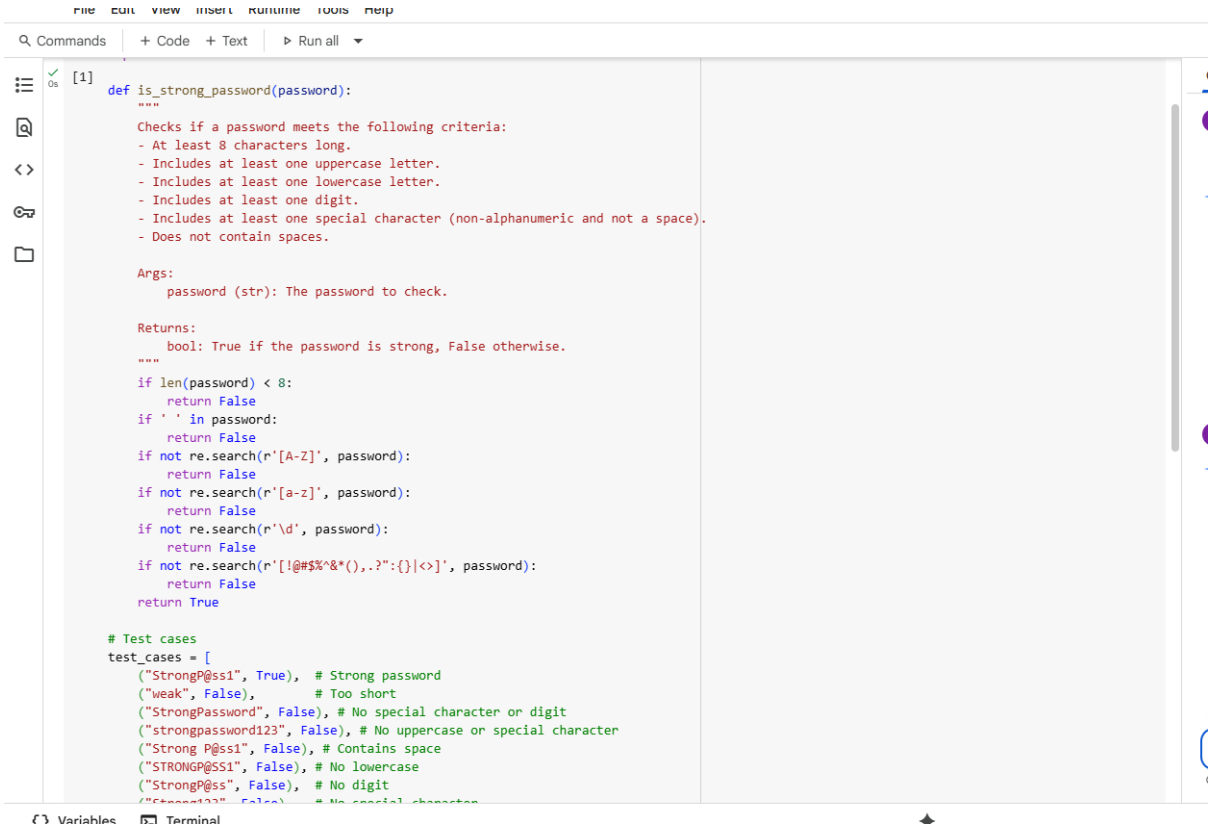
Example Assert Test Cases:

assert is_strong_password("Abcd@123") == True

assert is_strong_password("abcd123") == False

assert is_strong_password("ABCD@1234") == True

code output screenshots:



Code explanation:

Here's a breakdown of the code:

1. `import re`: This line imports the regular expression module in Python, which is used for pattern matching in strings.
2. `def is_strong_password(password):`: This defines the function `is_strong_password` that takes one argument, `password`.
3. `if len(password) < 8:`: This checks if the length of the password is less than 8 characters. If it is, the function returns `False` because the password is too short.
4. `if ' ' in password:`: This checks if the password contains any spaces. If it does, the function returns `False` because spaces are not allowed.
5. `if not re.search(r'[A-Z]', password):`: This uses a regular expression to check if the password contains at least one uppercase letter (A-Z). If it doesn't, the function returns `False`.
6. `if not re.search(r'[a-z]', password):`: This checks if the password contains at least one lowercase letter (a-z) using a regular expression. If it doesn't, the function returns `False`.
7. `if not re.search(r'\d', password):`: This checks if the password contains at least one digit (0-9) using a regular expression (`\d` is a shorthand for digits). If it doesn't, the function returns `False`.
8. `if not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):`: This checks if the password contains at least one special character from the defined set using a regular expression. If it doesn't, the function returns `False`.
9. `return True`: If all the above checks pass (meaning the password meets all the criteria), the function returns `True`, indicating that the password is strong.
10. `test_cases = [...]`: This is a list of tuples, where each tuple contains a test password and the expected boolean result (True if strong, False if not).
11. `for password, expected_result in test_cases:`: This loop iterates through each test case in the `test_cases` list.
12. `actual_result = is_strong_password(password)`: Inside the loop, it calls the

13. `print(...)`: This line prints the test password, the actual result from the function, and the expected result for comparison.

In summary, the code defines a function to validate password strength based on several criteria and then tests this function with a set of predefined examples.

Task-2:

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"

code output screenshots:

Commands | + Code | + Text | Run all

```python
class Inventory:
    """
    A simple inventory management class.
    """
    def __init__(self):
        """
        Initializes an empty inventory.
        """
        self.stock = {}

    def add_item(self, name, quantity):
        """
        Adds a specified quantity of an item to the inventory.
        If the item already exists, its quantity is increased.
        If the item does not exist, it is added.

        Args:
            name (str): The name of the item.
            quantity (int): The quantity to add. Must be a positive integer.

        Raises:
            ValueError: If quantity is not a positive integer.
        """
        if not isinstance(quantity, int) or quantity <= 0:
            raise ValueError("Quantity must be a positive integer.")

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

    def remove_item(self, name, quantity):
        """
        Removes a specified quantity of an item from the inventory.
        If the item quantity becomes zero or less, the item is removed from the inventory.

        Args:
            name (str): The name of the item.
            quantity (int): The quantity to remove. Must be a positive integer.
```

Variables | Terminal

33°C

File Edit View Insert Runtime Tools Help

Commands | + Code | + Text | Run all

```python
def classify_number(n):
    """
    Classifies a number as Positive, Negative, or Zero.
    Handles invalid inputs like strings and None.

    Args:
        n: The input to classify.

    Returns:
        str: "Positive", "Negative", "Zero", or "Invalid Input".
    """
    if isinstance(n, (int, float)):
        if n > 0:
            return "Positive"
        elif n < 0:
            return "Negative"
        else:
            return "Zero"
    else:
        return "Invalid Input"

# Assert test cases
test_cases = [
    (10, "Positive"),
    (-5, "Negative"),
    (0, "Zero"),
    (1, "Positive"),  # Boundary condition
    (-1, "Negative"), # Boundary condition
    ("abc", "Invalid Input"), # Invalid input (string)
    (None, "Invalid Input"),  # Invalid input (None)
    (3.14, "Positive"), # Float input
    (-2.7, "Negative")  # Float input
]

# Run test cases using a loop
for input_value, expected_output in test_cases:
    actual_output = classify_number(input_value)
    assert actual_output == expected_output, f"Input: {input_value}, Expected: {expected_output}, Got: {actual_output}"
    print(f"Test case passed for input: {input_value}")

print("\nAll test cases passed!")
```

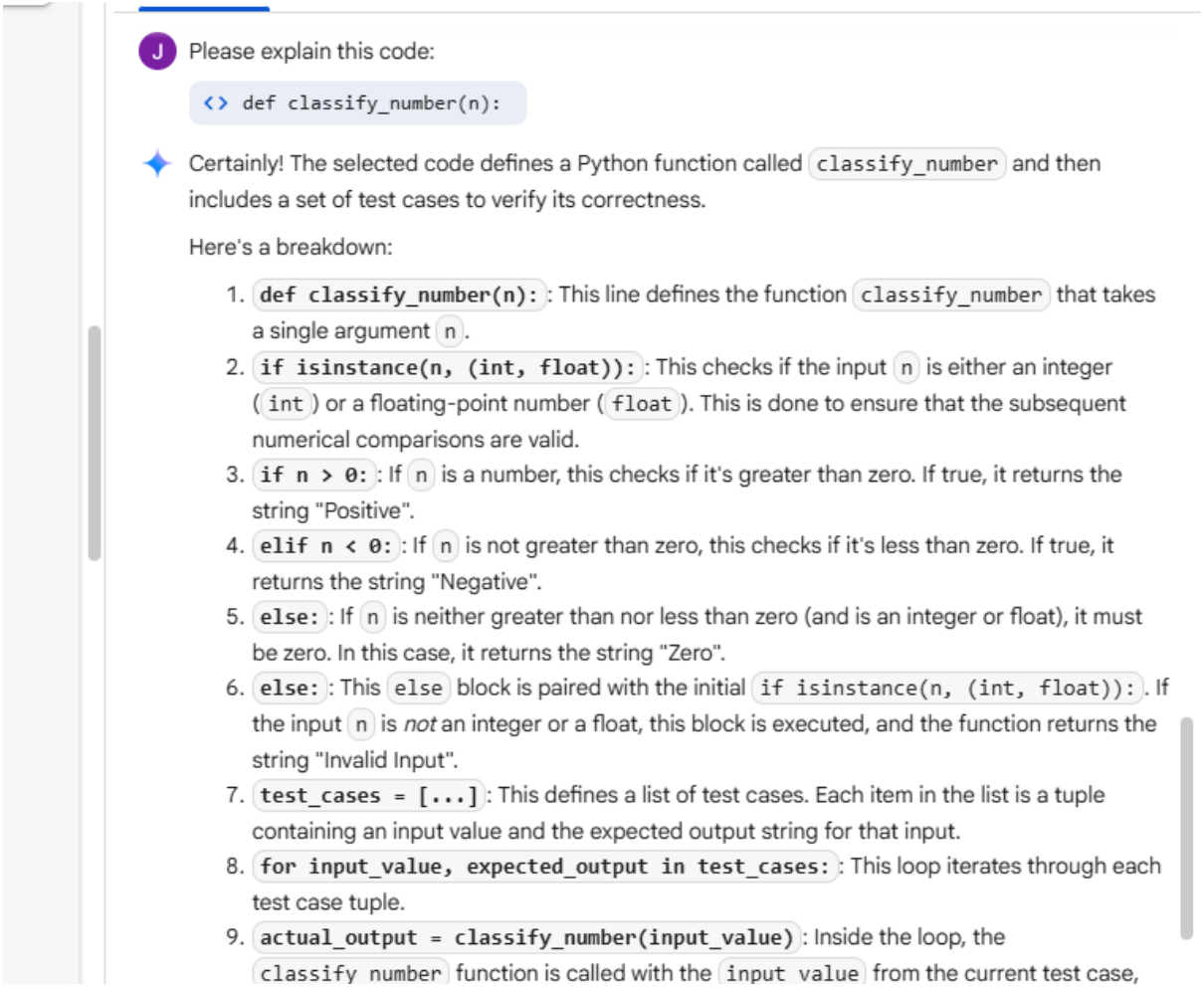
Variables | Terminal

Thunderstorm w...
In effect

Search

```
Test case passed for input: 10
Test case passed for input: -5
Test case passed for input: 0
Test case passed for input: 1
Test case passed for input: -1
Test case passed for input: abc
Test case passed for input: None
Test case passed for input: 3.14
Test case passed for input: -2.7

All test cases passed!
```

Code explanation :



Please explain this code:

```
def classify_number(n):
```

Certainly! The selected code defines a Python function called `classify_number` and then includes a set of test cases to verify its correctness.

Here's a breakdown:

1. `def classify_number(n):`: This line defines the function `classify_number` that takes a single argument `n`.
2. `if isinstance(n, (int, float)):`: This checks if the input `n` is either an integer (`int`) or a floating-point number (`float`). This is done to ensure that the subsequent numerical comparisons are valid.
3. `if n > 0:`: If `n` is a number, this checks if it's greater than zero. If true, it returns the string "Positive".
4. `elif n < 0:`: If `n` is not greater than zero, this checks if it's less than zero. If true, it returns the string "Negative".
5. `else:`: If `n` is neither greater than nor less than zero (and is an integer or float), it must be zero. In this case, it returns the string "Zero".
6. `else:`: This `else` block is paired with the initial `if isinstance(n, (int, float)):`. If the input `n` is *not* an integer or a float, this block is executed, and the function returns the string "Invalid Input".
7. `test_cases = [...]`: This defines a list of test cases. Each item in the list is a tuple containing an input value and the expected output string for that input.
8. `for input_value, expected_output in test_cases:`: This loop iterates through each test case tuple.
9. `actual_output = classify_number(input_value)`: Inside the loop, the `classify_number` function is called with the `input_value` from the current test case,

be zero. In this case, it returns the string "Zero".

6. `else:` : This `else` block is paired with the initial `if isinstance(n, (int, float)):`. If the input `n` is *not* an integer or a float, this block is executed, and the function returns the string "Invalid Input".
7. `test_cases = [...]` : This defines a list of test cases. Each item in the list is a tuple containing an input value and the expected output string for that input.
8. `for input_value, expected_output in test_cases:` : This loop iterates through each test case tuple.
9. `actual_output = classify_number(input_value)` : Inside the loop, the `classify_number` function is called with the `input_value` from the current test case, and the result is stored in `actual_output`.
10. `assert actual_output == expected_output, f"..."` : This is an `assert` statement, which is used for debugging and testing. It checks if the `actual_output` matches the `expected_output`. If they do *not* match, an `AssertionError` is raised, and the program stops, displaying the provided error message (which includes the input, expected output, and actual output). If they match, the assertion passes, and the program continues.
11. `print(f"Test case passed for input: {input_value}")` : If the assertion passes, this line prints a message indicating that the test case for the current input value was successful.
12. `print("\nAll test cases passed!")` : After the loop finishes without any assertions failing, this line is printed, indicating that all the provided test cases passed.

In essence, this code provides a function to categorize inputs as "Positive", "Negative", "Zero", or "Invalid Input" and then includes a set of self-checking tests to ensure the function works as intended for various inputs, including boundary conditions and non-numeric types.

TASK-3:

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).**

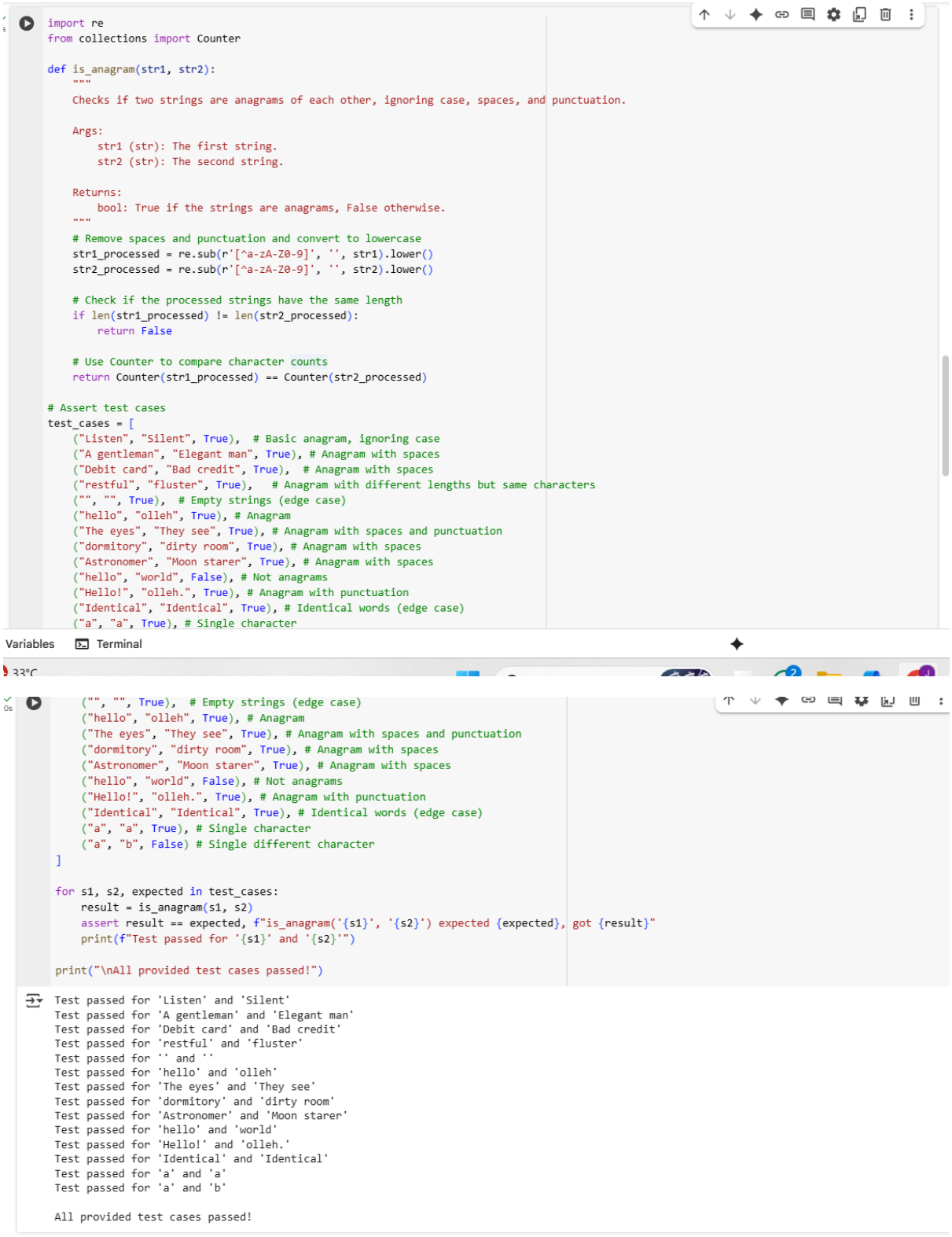
Example Assert Test Cases:

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

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

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

code output screenshots:



```python
import re
from collections import Counter

def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams of each other, ignoring case, spaces, and punctuation.

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

    Returns:
        bool: True if the strings are anagrams, False otherwise.
    """
    # Remove spaces and punctuation and convert to lowercase
    str1_processed = re.sub(r'[^a-zA-Z0-9]', '', str1).lower()
    str2_processed = re.sub(r'[^a-zA-Z0-9]', '', str2).lower()

    # Check if the processed strings have the same length
    if len(str1_processed) != len(str2_processed):
        return False

    # Use Counter to compare character counts
    return Counter(str1_processed) == Counter(str2_processed)

# Assert test cases
test_cases = [
    ("Listen", "Silent", True),  # Basic anagram, ignoring case
    ("A gentleman", "Elegant man", True), # Anagram with spaces
    ("Debit card", "Bad credit", True),  # Anagram with spaces
    ("restful", "fluster", True),   # Anagram with different lengths but same characters
    ("", "", True),  # Empty strings (edge case)
    ("hello", "olleh", True), # Anagram
    ("The eyes", "They see", True), # Anagram with spaces and punctuation
    ("dormitory", "dirty room", True), # Anagram with spaces
    ("Astronomer", "Moon starer", True), # Anagram with spaces
    ("hello", "world", False), # Not anagrams
    ("Hello!", "olleh.", True), # Anagram with punctuation
    ("Identical", "Identical", True), # Identical words (edge case)
    ("a", "a", True), # Single character
```

Variables Terminal

33°C

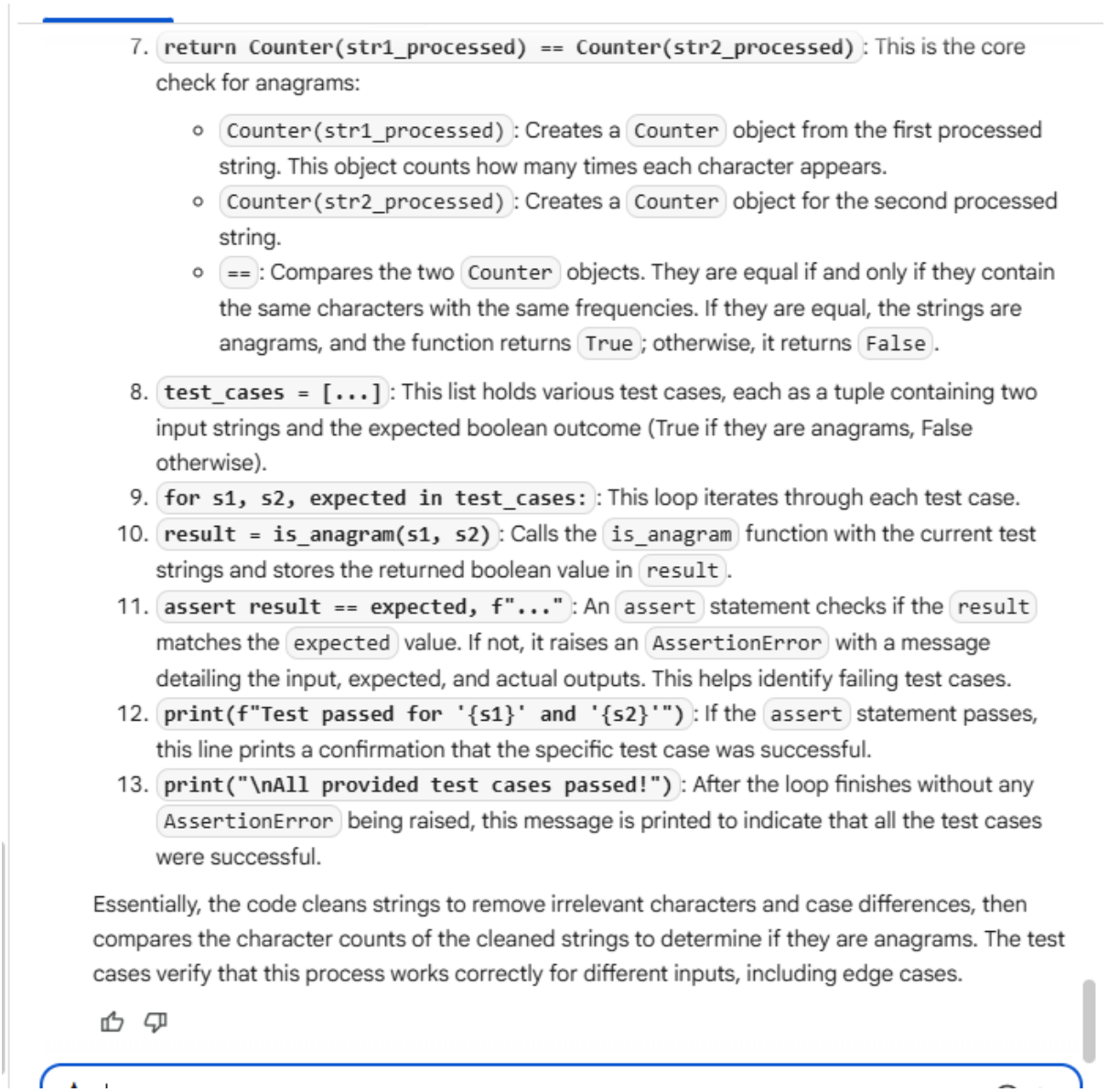
```python
    ("", "", True),  # Empty strings (edge case)
    ("hello", "olleh", True), # Anagram
    ("The eyes", "They see", True), # Anagram with spaces and punctuation
    ("dormitory", "dirty room", True), # Anagram with spaces
    ("Astronomer", "Moon starer", True), # Anagram with spaces
    ("hello", "world", False), # Not anagrams
    ("Hello!", "olleh.", True), # Anagram with punctuation
    ("Identical", "Identical", True), # Identical words (edge case)
    ("a", "a", True), # Single character
    ("a", "b", False) # Single different character
]

for s1, s2, expected in test_cases:
    result = is_anagram(s1, s2)
    assert result == expected, f"is_anagram('{s1}', '{s2}') expected {expected}, got {result}"
    print(f"Test passed for '{s1}' and '{s2}'")

print("\nAll provided test cases passed!")
```

```
Test passed for 'Listen' and 'Silent'
Test passed for 'A gentleman' and 'Elegant man'
Test passed for 'Debit card' and 'Bad credit'
Test passed for 'restful' and 'fluster'
Test passed for '' and ''
Test passed for 'hello' and 'olleh'
Test passed for 'The eyes' and 'They see'
Test passed for 'dormitory' and 'dirty room'
Test passed for 'Astronomer' and 'Moon starer'
Test passed for 'hello' and 'world'
Test passed for 'Hello!' and 'olleh.'
Test passed for 'Identical' and 'Identical'
Test passed for 'a' and 'a'
Test passed for 'a' and 'b'

All provided test cases passed!
```

Code explanation:

Here's a breakdown:

1. `import re`: Imports the regular expression module, which is used to remove characters that aren't letters or numbers.
2. `from collections import Counter`: Imports the `Counter` class, a specialized dictionary subclass for counting hashable objects.
3. `def is_anagram(str1, str2):`: Defines the function `is_anagram` that takes two strings, `str1` and `str2`, as input.
4. `str1_processed = re.sub(r'[^a-zA-Z0-9]', '', str1).lower()`: This line processes the first string:
    - `re.sub(r'[^a-zA-Z0-9]', '', str1)`: Uses a regular expression to find and remove any character that is *not* an uppercase letter (`A-Z`), a lowercase letter (`a-z`), or a digit (`0-9`). It replaces these characters with an empty string.
    - `.lower()`: Converts the resulting string to lowercase.
    - The cleaned-up string is stored in `str1_processed`.
5. `str2_processed = re.sub(r'[^a-zA-Z0-9]', '', str2).lower()`: This does the same cleaning and lowercasing for the second string, storing the result in `str2_processed`.
6. `if len(str1_processed) != len(str2_processed):`: After cleaning, if the lengths of the two processed strings are different, they cannot be anagrams, so the function immediately returns `False`.
7. `return Counter(str1_processed) == Counter(str2_processed)`: This is the core check for anagrams:
    - `Counter(str1_processed)`: Creates a `Counter` object from the first processed string. This object counts how many times each character appears.
    - `Counter(str2_processed)`: Creates a `Counter` object for the second processed string.
    - `==`: Compares the two `Counter` objects. They are equal if and only if they contain



TASK-5:

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".**

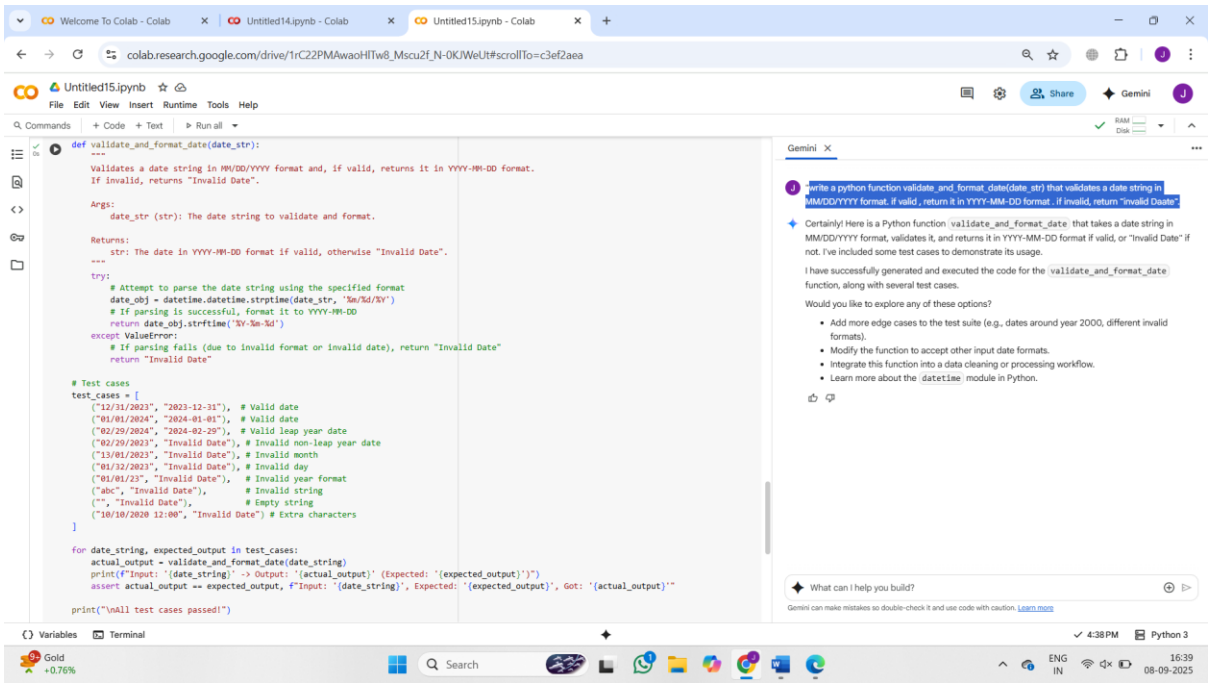
Example Assert Test Cases:

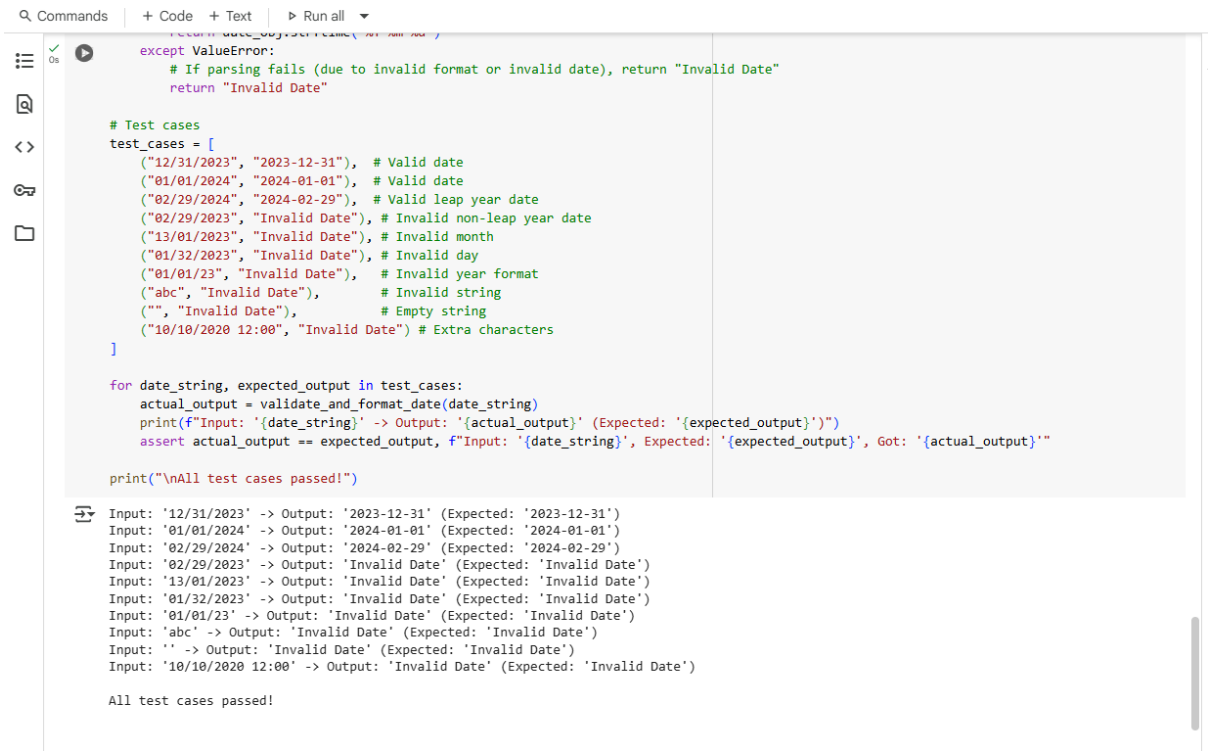
assert validate_and_format_date("10/15/2023") == "2023-10-15"

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

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

code output screenshots:





CODE EXPLANATION:

Gemini

J Please explain this code:

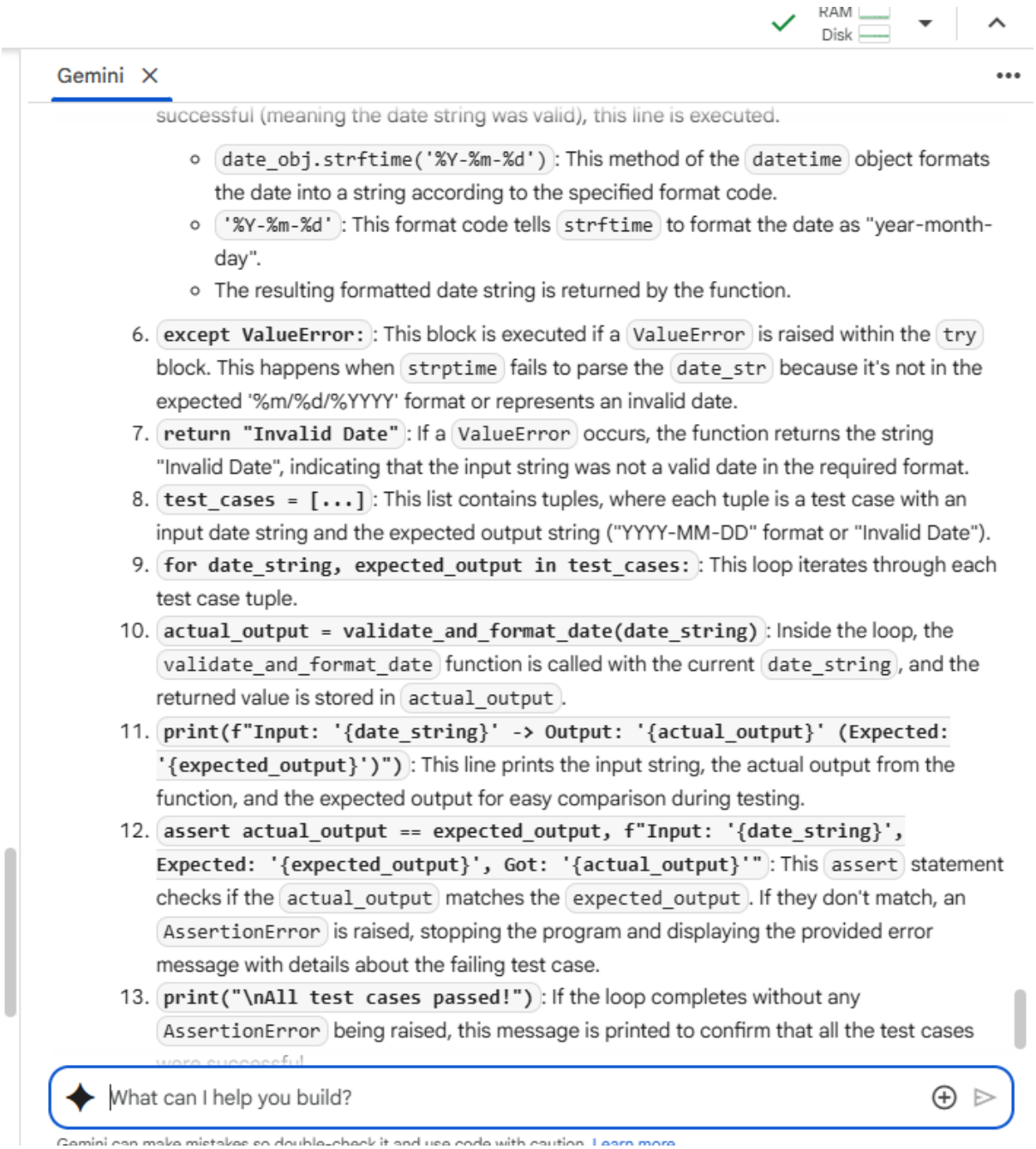
```
import datetime
```

Certainly! The selected code defines a Python function to validate and reformat a date string and includes test cases to check its correctness.

Here's a breakdown of the code:

1. `import datetime`: This line imports the `datetime` module, which provides classes for working with dates and times.
2. `def validate_and_format_date(date_str):`: This defines the function `validate_and_format_date` that takes one argument, `date_str`, which is expected to be a string representing a date.
3. `try:`: This block starts a `try...except` block, which is used to handle potential errors that might occur during the execution of the code within the `try` block. In this case, it's used to catch errors that might arise from trying to parse an invalid date string.
4. `date_obj = datetime.datetime.strptime(date_str, '%m/%d/%Y')`: This is the core of the validation and parsing.
    - `datetime.datetime.strptime()`: This is a class method from the `datetime` module that parses a string (`date_str`) according to a specified format code (`'%m/%d/%Y'`).
    - `'%m/%d/%Y'`: This format code tells `strptime` to expect the string in the format "month/day/year", where:
        - `%m` represents the month as a zero-padded decimal number (e.g., 01, 12).
        - `%d` represents the day of the month as a zero-padded decimal number (e.g., 01, 31).
        - `%Y` represents the year with century as a decimal number (e.g., 2023).

What can I help you build?



TASK-4:

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

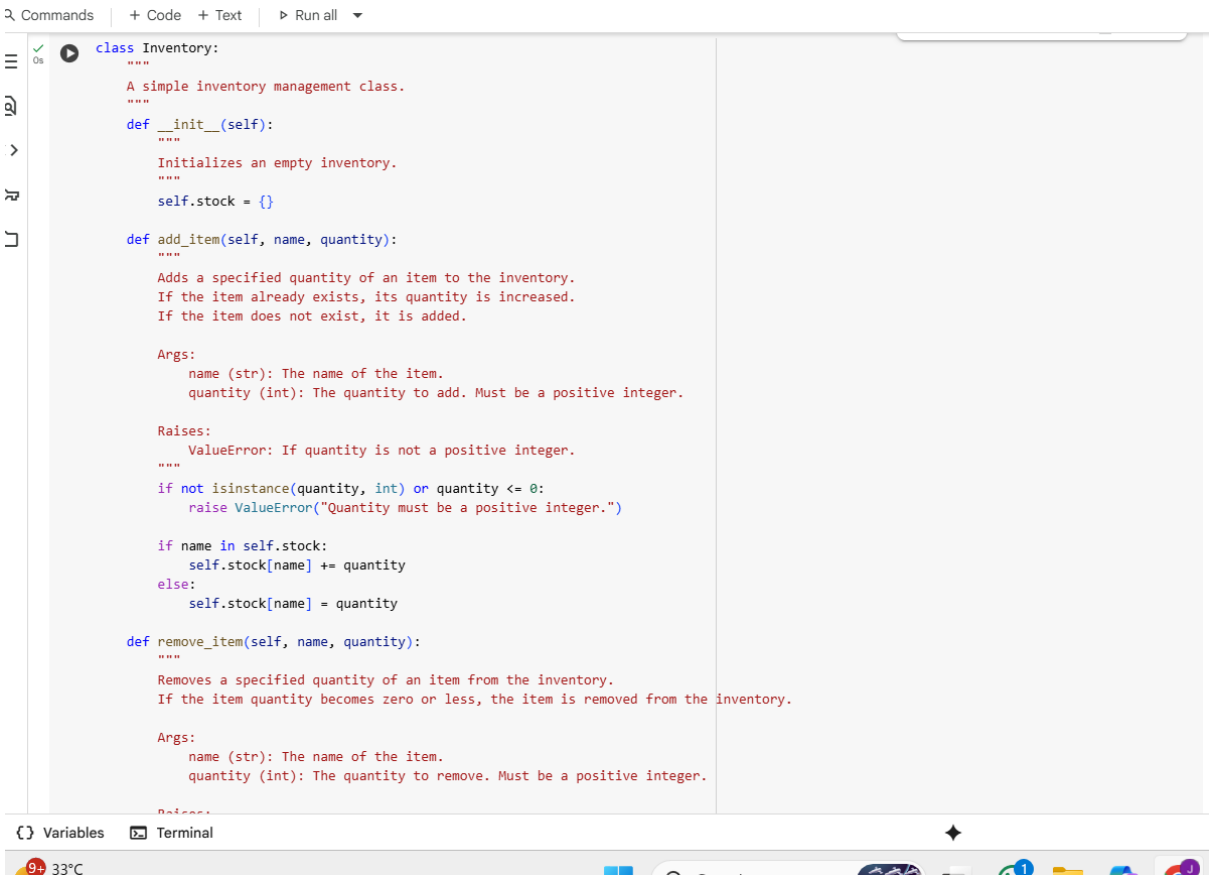
inv.remove_item("Pen", 5)

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

inv.add_item("Book", 3)

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

CODE OUTPUT SCREENSHOTS:



```
        Raises:
            ValueError: If quantity is not a positive integer.
            KeyError: If the item is not in the inventory.
            ValueError: If trying to remove more quantity than available.
        """
        if not isinstance(quantity, int) or quantity <= 0:
            raise ValueError("Quantity must be a positive integer.")

        if name not in self.stock:
            raise KeyError(f"Item '{name}' not found in inventory.")

        if self.stock[name] < quantity:
            raise ValueError(f"Not enough stock for item '{name}'. Available: {self.stock[name]}, Trying to remove: {quantity}")

        self.stock[name] -= quantity
        if self.stock[name] <= 0:
            del self.stock[name]

    def get_stock(self, name):
        """
        Gets the current stock quantity of an item.

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

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

# Assert-based tests

# Test Case 1: Adding and getting stock
inventory1 = Inventory()
inventory1.add_item("Apple", 10)
assert inventory1.get_stock("Apple") == 10, "Test Case 1 Failed: Add item and get stock"
assert inventory1.get_stock("Banana") == 0, "Test Case 1 Failed: Get stock for non-existent item"

# Test Case 2: Adding more of an existing item
inventory2 = Inventory()
```

Variables Terminal

Commands + Code + Text Run all

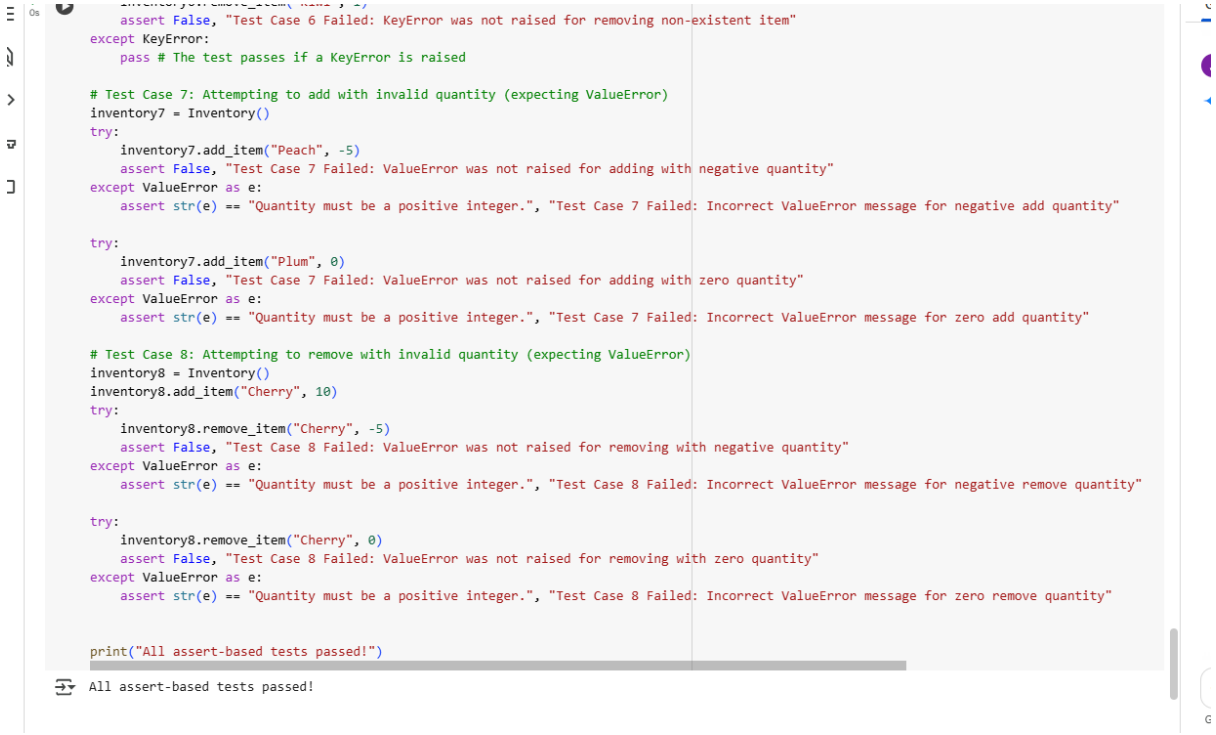
```
# Test Case 2: Adding more of an existing item
inventory2 = Inventory()
inventory2.add_item("Orange", 5)
inventory2.add_item("Orange", 7)
assert inventory2.get_stock("Orange") == 12, "Test Case 2 Failed: Add more of existing item"

# Test Case 3: Removing stock
inventory3 = Inventory()
inventory3.add_item("Grape", 20)
inventory3.remove_item("Grape", 5)
assert inventory3.get_stock("Grape") == 15, "Test Case 3 Failed: Remove stock"

# Test Case 4: Removing all stock
inventory4 = Inventory()
inventory4.add_item("Mango", 10)
inventory4.remove_item("Mango", 10)
assert inventory4.get_stock("Mango") == 0, "Test Case 4 Failed: Remove all stock"
assert "Mango" not in inventory4.stock, "Test Case 4 Failed: Item not removed from inventory"

# Test Case 5: Attempting to remove more than available (expecting ValueError)
inventory5 = Inventory()
inventory5.add_item("Pear", 8)
try:
    inventory5.remove_item("Pear", 10)
    assert False, "Test Case 5 Failed: ValueError was not raised for removing more than available"
except ValueError as e:
    assert str(e) == "Not enough stock for item 'Pear'. Available: 8, Trying to remove: 10", "Test Case 5 Failed: Incorrect ValueError message

# Test Case 6: Attempting to remove a non-existent item (expecting KeyError)
inventory6 = Inventory()
try:
    inventory6.remove_item("Kiwi", 1)
    assert False, "Test Case 6 Failed: KeyError was not raised for removing non-existent item"
except KeyError:
    pass # The test passes if a KeyError is raised

# Test Case 7: Attempting to add with invalid quantity (expecting ValueError)
inventory7 = Inventory()
try:
```

Gemini

Variables Terminal

33°C Mostly sunny

Search



```
        assert False, "Test Case 6 Failed: KeyError was not raised for removing non-existent item"
    except KeyError:
        pass # The test passes if a KeyError is raised

    # Test Case 7: Attempting to add with invalid quantity (expecting ValueError)
    inventory7 = Inventory()
    try:
        inventory7.add_item("Peach", -5)
        assert False, "Test Case 7 Failed: ValueError was not raised for adding with negative quantity"
    except ValueError as e:
        assert str(e) == "Quantity must be a positive integer.", "Test Case 7 Failed: Incorrect ValueError message for negative add quantity"

    try:
        inventory7.add_item("Plum", 0)
        assert False, "Test Case 7 Failed: ValueError was not raised for adding with zero quantity"
    except ValueError as e:
        assert str(e) == "Quantity must be a positive integer.", "Test Case 7 Failed: Incorrect ValueError message for zero add quantity"

    # Test Case 8: Attempting to remove with invalid quantity (expecting ValueError)
    inventory8 = Inventory()
    inventory8.add_item("Cherry", 10)
    try:
        inventory8.remove_item("Cherry", -5)
        assert False, "Test Case 8 Failed: ValueError was not raised for removing with negative quantity"
    except ValueError as e:
        assert str(e) == "Quantity must be a positive integer.", "Test Case 8 Failed: Incorrect ValueError message for negative remove quantity"

    try:
        inventory8.remove_item("Cherry", 0)
        assert False, "Test Case 8 Failed: ValueError was not raised for removing with zero quantity"
    except ValueError as e:
        assert str(e) == "Quantity must be a positive integer.", "Test Case 8 Failed: Incorrect ValueError message for zero remove quantity"


    print("All assert-based tests passed!")
```

```
All assert-based tests passed!
```

CODE EXPLANATION:

Gemini ×

The `Inventory` Class:

1. `class Inventory:` : This line defines a new class named `Inventory`.
2. `__init__(self):` : This is the constructor method. It's called when you create a new `Inventory` object.
    - `self.stock = {}` : Inside the constructor, it initializes an empty dictionary called `stock`. This dictionary will store the inventory, where keys will be the item names (strings) and values will be their quantities (integers).
3. `add_item(self, name, quantity):` : This method is used to add items to the inventory or increase the quantity of existing items.
    - `if not isinstance(quantity, int) or quantity <= 0:` : It first checks if the `quantity` is a positive integer. If not, it raises a `ValueError`.
    - `if name in self.stock:` : It checks if the `name` of the item already exists as a key in the `stock` dictionary.
    - `self.stock[name] += quantity` : If the item exists, its current quantity is increased by the `quantity` being added.
    - `else: self.stock[name] = quantity` : If the item does not exist, it's added to the `stock` dictionary with the given `quantity`.
4. `remove_item(self, name, quantity):` : This method is used to remove items from the inventory or decrease their quantity.
    - `if not isinstance(quantity, int) or quantity <= 0:` : It checks if the `quantity` to remove is a positive integer. If not, it raises a `ValueError`.
    - `if name not in self.stock:` : It checks if the item exists in the inventory. If not, it raises a `KeyError`.
    - `if self.stock[name] < quantity:` : It checks if there is enough stock to remove the specified `quantity`. If not, it raises a `ValueError`.
    - `self.stock[name] -= quantity` : If the item exists and there's enough stock, the

What can I help you build?

becomes zero or less after removal, the item is completely removed from the `stock` dictionary.

5. `get_stock(self, name):` : This method returns the current stock quantity of a given item.
    - `return self.stock.get(name, 0)` : It uses the dictionary's `get()` method to retrieve the value (quantity) associated with the `name` key. If the `name` is not found in the dictionary, it returns the default value of `0` instead of raising a `KeyError`.

The Assert-based Tests:

The code then includes several assert-based test cases to verify that the methods of the `Inventory` class work as expected under various scenarios:

- **Test Case 1:** Tests adding an item and then getting its stock, as well as getting the stock of a non-existent item.
- **Test Case 2:** Tests adding more quantity to an item that already exists.
- **Test Case 3:** Tests removing a partial quantity of an item.
- **Test Case 4:** Tests removing the entire stock of an item, verifying that it's removed from the inventory.
- **Test Case 5:** Tests attempting to remove more quantity than available, expecting a `ValueError`. It uses a `try...except` block to catch the expected error and an `assert False` to ensure that a `ValueError` *was* indeed raised.
- **Test Case 6:** Tests attempting to remove a non-existent item, expecting a `KeyError`. It uses a `try...except KeyError:` block. The test passes if the `KeyError` is caught.
- **Test Case 7:** Tests attempting to add an item with an invalid quantity (negative or zero), expecting a `ValueError`.
- **Test Case 8:** Tests attempting to remove an item with an invalid quantity (negative or zero), expecting a `ValueError`.

Each test case uses `assert` statements to check if the actual output or behavior matches the expected outcome. If an `assert` statement's condition is false, it raises an `AssertionError`,