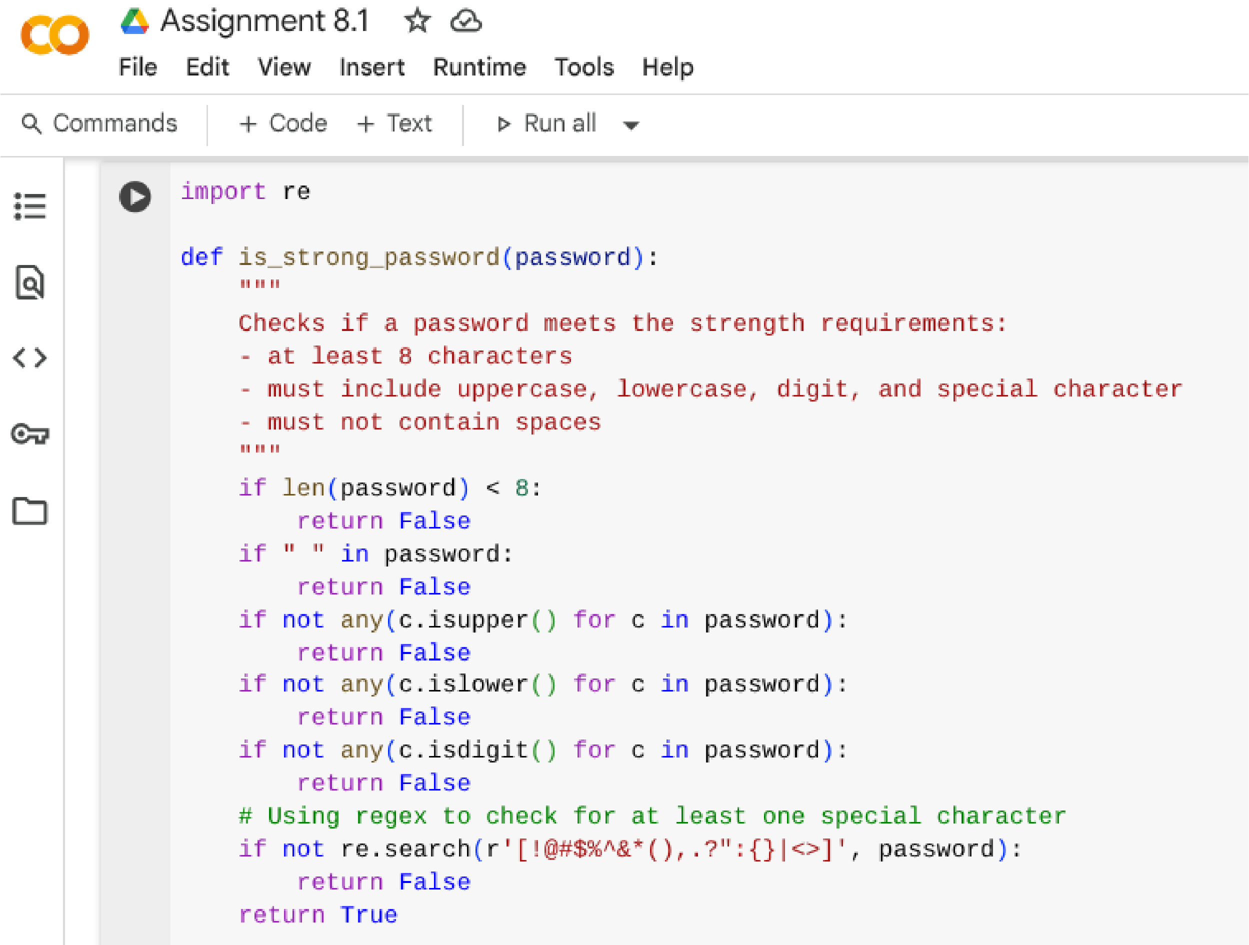
ASSIGNMENT-8.1

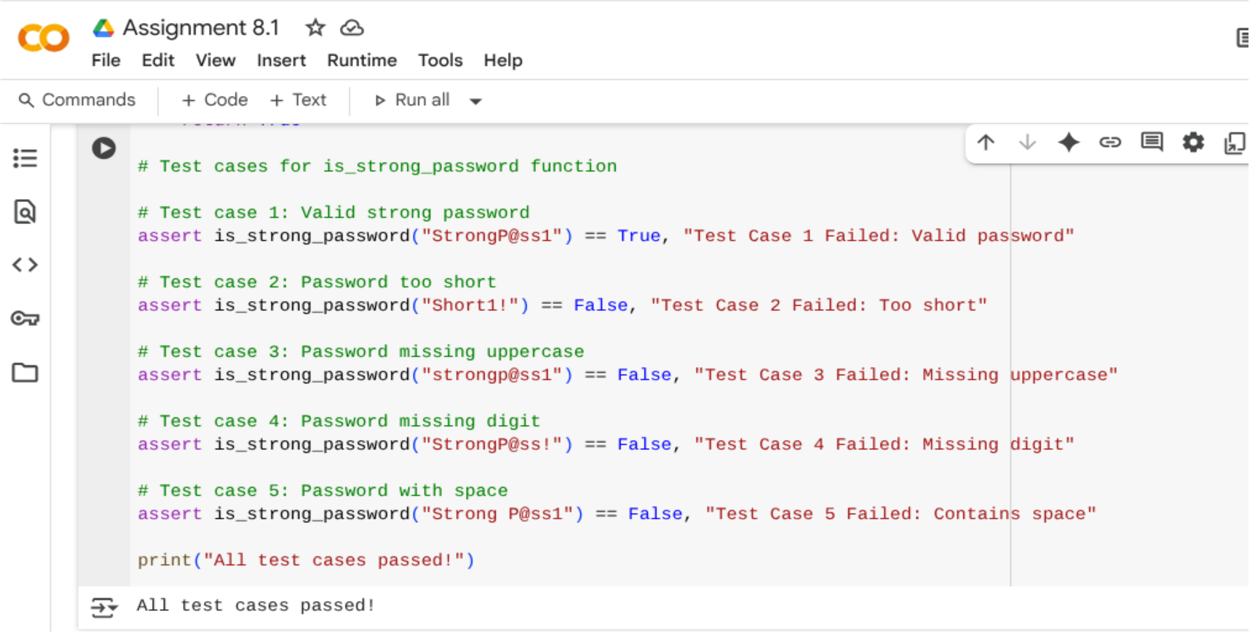
Task 1:

write a python function is_strong_password(password)that checks password strength.Requirements:at least 8 characters,must include uppercase,lowercase,digit and special character and must not contain spaces. give 5 test cases.

Code and Output:



```python
import re

def is_strong_password(password):
    """
    Checks if a password meets the strength requirements:
    - at least 8 characters
    - must include uppercase, lowercase, digit, and special character
    - must not contain spaces
    """
    if len(password) < 8:
        return False
    if " " in password:
        return False
    if not any(c.isupper() for c in password):
        return False
    if not any(c.islower() for c in password):
        return False
    if not any(c.isdigit() for c in password):
        return False
    # Using regex to check for at least one special character
    if not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        return False
    return True
```



```python
# Test cases for is_strong_password function

# Test case 1: Valid strong password
assert is_strong_password("StrongP@ss1") == True, "Test Case 1 Failed: Valid password"

# Test case 2: Password too short
assert is_strong_password("Short1!") == False, "Test Case 2 Failed: Too short"

# Test case 3: Password missing uppercase
assert is_strong_password("strongp@ss1") == False, "Test Case 3 Failed: Missing uppercase"

# Test case 4: Password missing digit
assert is_strong_password("StrongP@ss!") == False, "Test Case 4 Failed: Missing digit"

# Test case 5: Password with space
assert is_strong_password("Strong P@ss1") == False, "Test Case 5 Failed: Contains space"

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

```
All test cases passed!
```

Explanation:

1. **Import `re`:** The code starts by importing the regular expression module, which is used for pattern matching.
2. **Define `is_strong_password` function:** A function named `is_strong_password` is defined to take a `password` string as input.
3. **Check Minimum Length:** It first checks if the password is at least 8 characters long. If not, it's considered weak.
4. **Check for Spaces:** The function verifies that the password does not contain any spaces.
5. **Check for Uppercase:** It checks if there is at least one uppercase letter in the password.
6. **Check for Lowercase:** It checks if there is at least one lowercase letter in the password.
7. **Check for Digits:** It checks if there is at least one digit (0-9) in the password.
8. **Check for Special Characters:** It uses a regular expression to ensure the password contains at least one special character from a predefined set.
9. **Return True/False:** If all the above conditions are met, the function returns `True` (strong password); otherwise, it returns `False`.
10. **Test Cases:** The code includes `assert` statements to test the function with various examples, confirming its correctness.

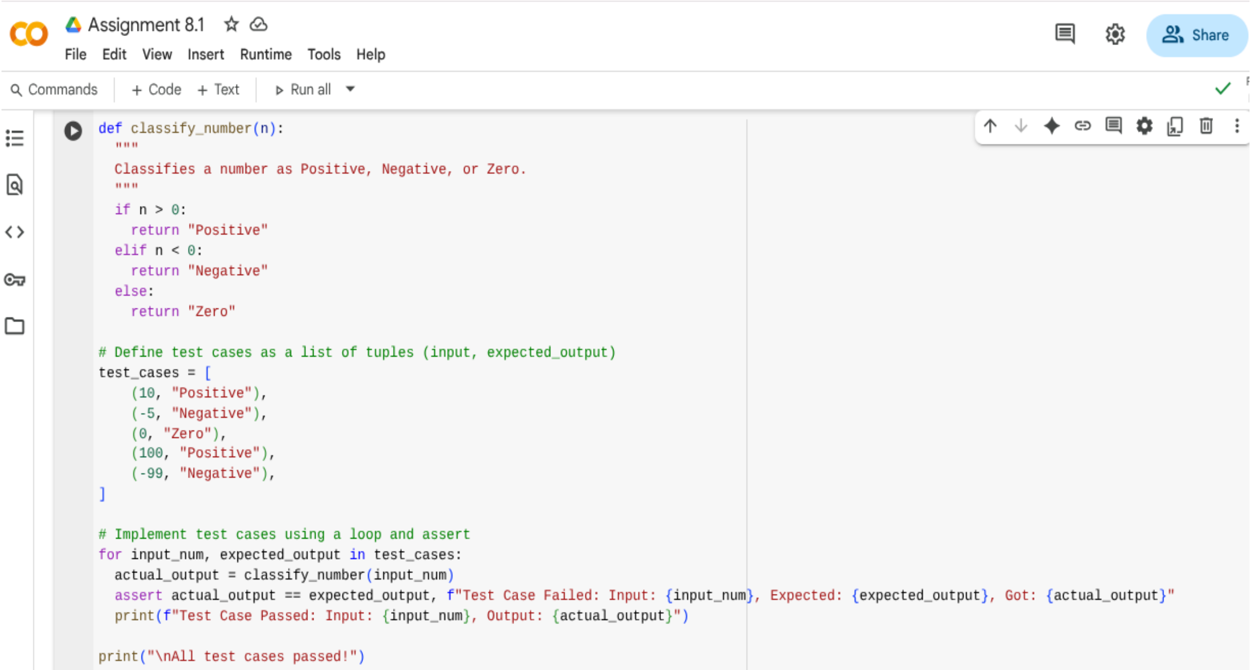
Task-2:

write a python program using functions to generate at least 3 assert test cases for a classify number(n) function. implement using loops examples Assert Test cases assert classify_number(10) == "Positive"

assert classify_number(-5) == "Negative"

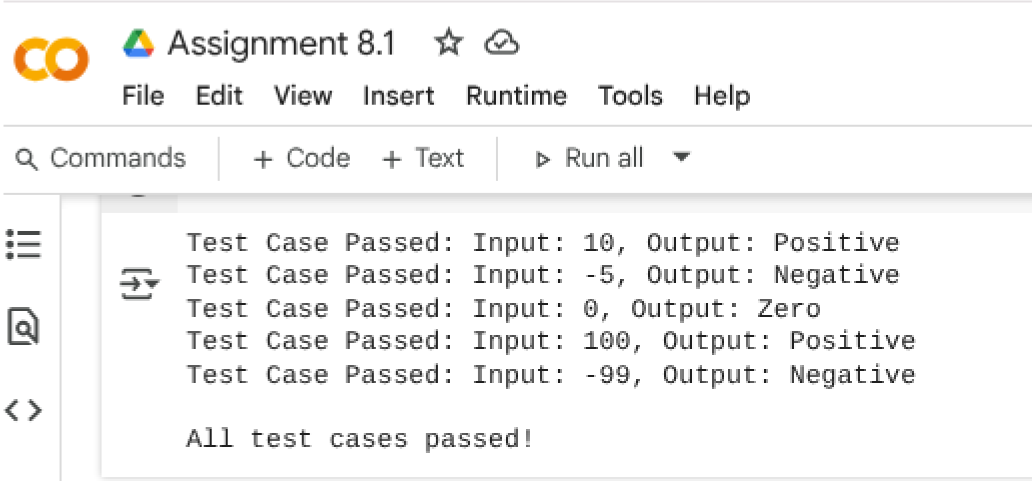
assert classify_number(0) == "Zero"

Code:



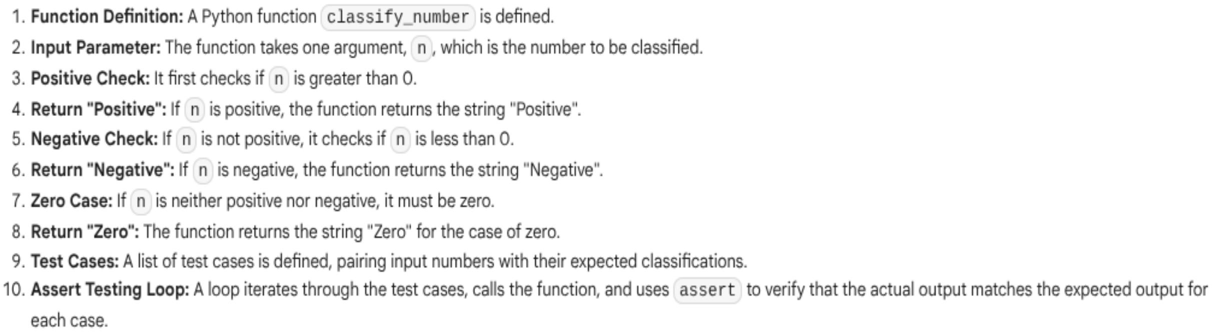
```python
def classify_number(n):
  """
  Classifies a number as Positive, Negative, or Zero.
  """
  if n > 0:
    return "Positive"
  elif n < 0:
    return "Negative"
  else:
    return "Zero"

# Define test cases as a list of tuples (input, expected_output)
test_cases = [
    (10, "Positive"),
    (-5, "Negative"),
    (0, "Zero"),
    (100, "Positive"),
    (-99, "Negative"),
]

# Implement test cases using a loop and assert
for input_num, expected_output in test_cases:
  actual_output = classify_number(input_num)
  assert actual_output == expected_output, f"Test Case Failed: Input: {input_num}, Expected: {expected_output}, Got: {actual_output}"
  print(f"Test Case Passed: Input: {input_num}, Output: {actual_output}")

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

Output:


Assignment 8.1
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

```
Test Case Passed: Input: 10, Output: Positive
Test Case Passed: Input: -5, Output: Negative
Test Case Passed: Input: 0, Output: Zero
Test Case Passed: Input: 100, Output: Positive
Test Case Passed: Input: -99, Output: Negative

All test cases passed!
```

Explanation:



1. **Function Definition:** A Python function `classify_number` is defined.
2. **Input Parameter:** The function takes one argument, `n`, which is the number to be classified.
3. **Positive Check:** It first checks if `n` is greater than 0.
4. **Return "Positive":** If `n` is positive, the function returns the string "Positive".
5. **Negative Check:** If `n` is not positive, it checks if `n` is less than 0.
6. **Return "Negative":** If `n` is negative, the function returns the string "Negative".
7. **Zero Case:** If `n` is neither positive nor negative, it must be zero.
8. **Return "Zero":** The function returns the string "Zero" for the case of zero.
9. **Test Cases:** A list of test cases is defined, pairing input numbers with their expected classifications.
10. **Assert Testing Loop:** A loop iterates through the test cases, calls the function, and uses `assert` to verify that the actual output matches the expected output for each case.

Task-3:

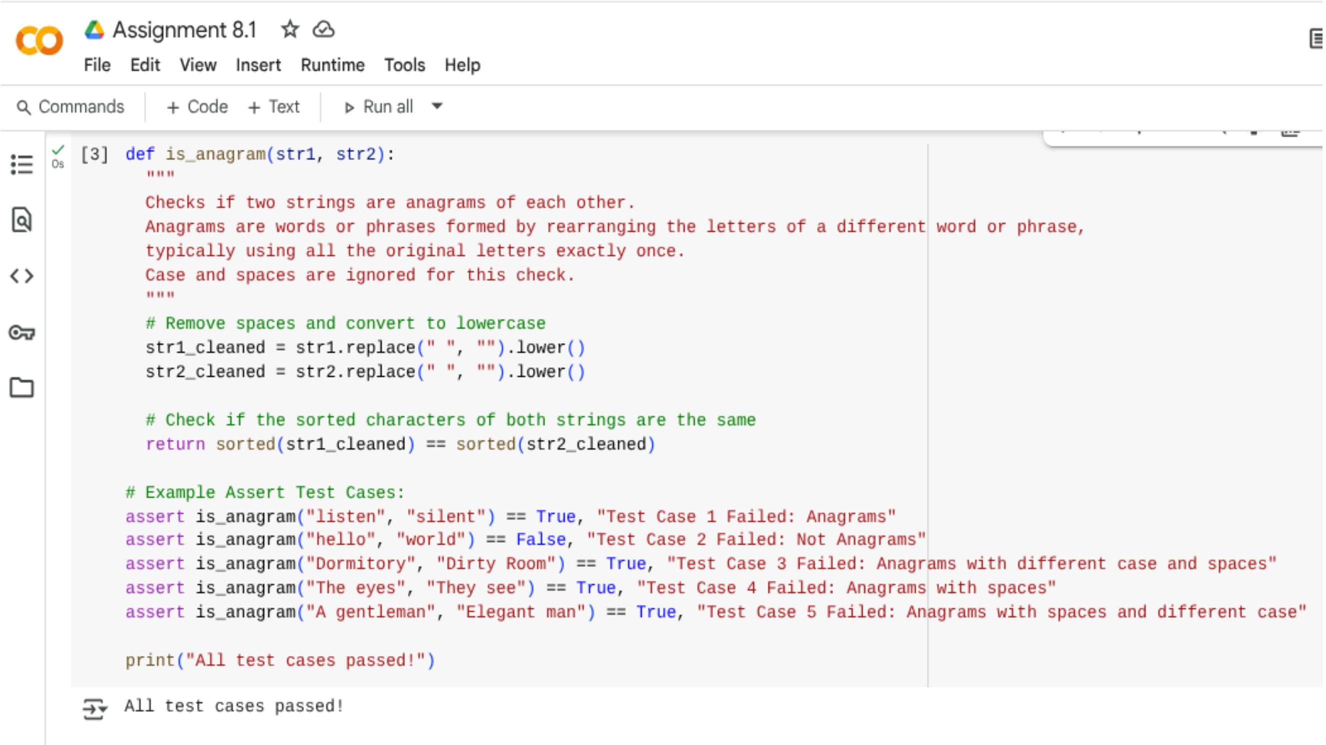
write a python program using functions to generate at least assert test cases for is anagrsm(str1,str2)Example Assert Test Cases:

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

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

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

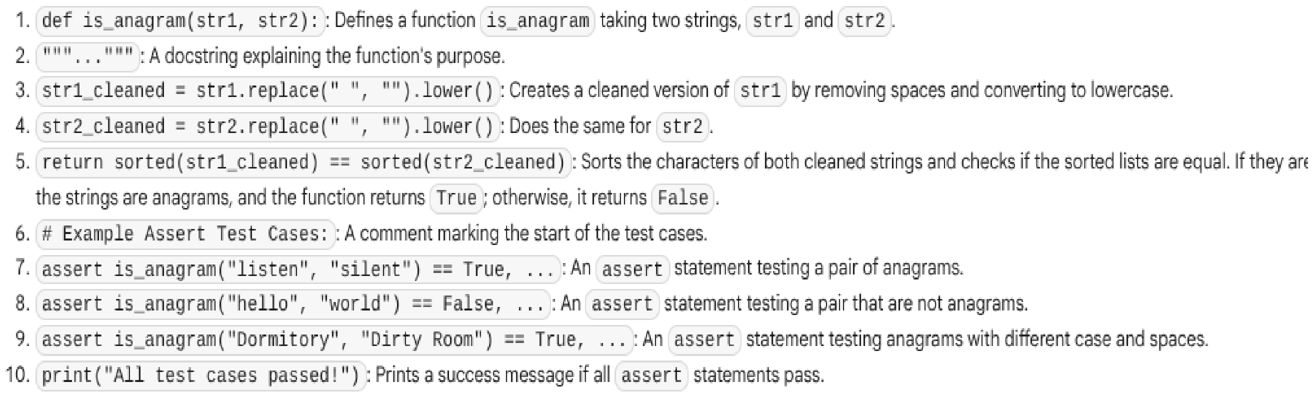
Code and Output:



```python
def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams of each other.
    Anagrams are words or phrases formed by rearranging the letters of a different word or phrase,
    typically using all the original letters exactly once.
    Case and spaces are ignored for this check.
    """
    # Remove spaces and convert to lowercase
    str1_cleaned = str1.replace(" ", "").lower()
    str2_cleaned = str2.replace(" ", "").lower()

    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)

# Example Assert Test Cases:
assert is_anagram("listen", "silent") == True, "Test Case 1 Failed: Anagrams"
assert is_anagram("hello", "world") == False, "Test Case 2 Failed: Not Anagrams"
assert is_anagram("Dormitory", "Dirty Room") == True, "Test Case 3 Failed: Anagrams with different case and spaces"
assert is_anagram("The eyes", "They see") == True, "Test Case 4 Failed: Anagrams with spaces"
assert is_anagram("A gentleman", "Elegant man") == True, "Test Case 5 Failed: Anagrams with spaces and different case"

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

```
All test cases passed!
```

Explanation:

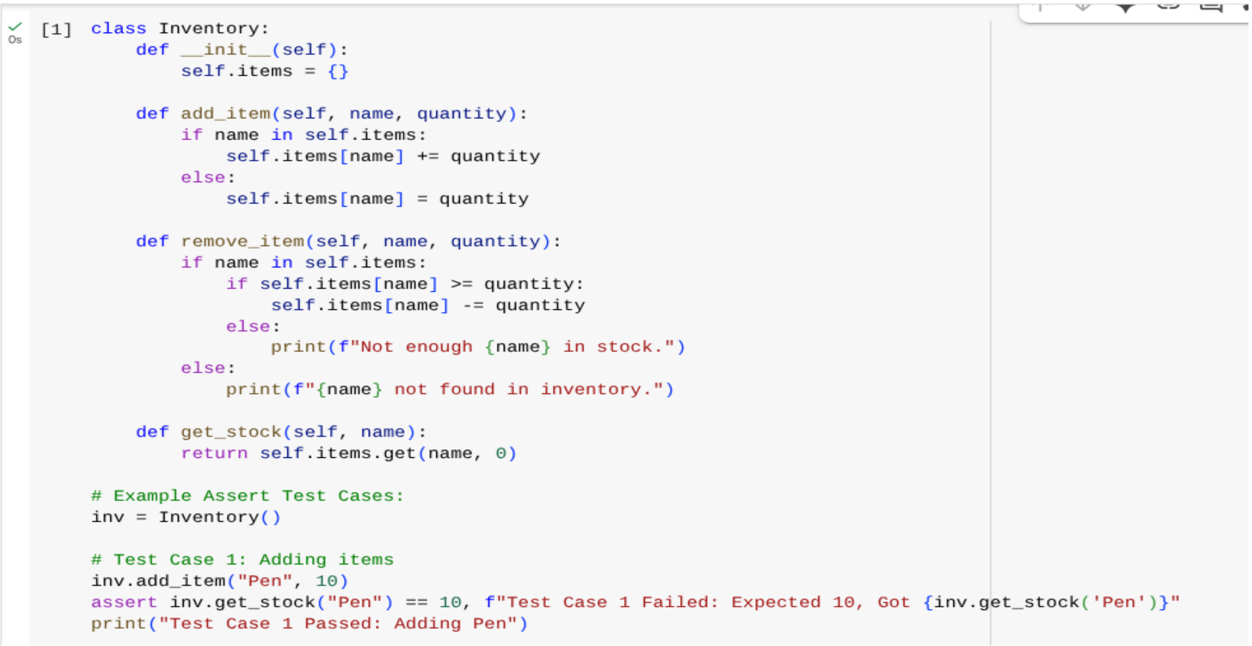


1. `def is_anagram(str1, str2):`: Defines a function `is_anagram` taking two strings, `str1` and `str2`.
2. `"""..."""`: A docstring explaining the function's purpose.
3. `str1_cleaned = str1.replace(" ", "").lower()`: Creates a cleaned version of `str1` by removing spaces and converting to lowercase.
4. `str2_cleaned = str2.replace(" ", "").lower()`: Does the same for `str2`.
5. `return sorted(str1_cleaned) == sorted(str2_cleaned)`: Sorts the characters of both cleaned strings and checks if the sorted lists are equal. If they are the strings are anagrams, and the function returns `True`; otherwise, it returns `False`.
6. `# Example Assert Test Cases:`: A comment marking the start of the test cases.
7. `assert is_anagram("listen", "silent") == True, ...`: An `assert` statement testing a pair of anagrams.
8. `assert is_anagram("hello", "world") == False, ...`: An `assert` statement testing a pair that are not anagrams.
9. `assert is_anagram("Dormitory", "Dirty Room") == True, ...`: An `assert` statement testing anagrams with different case and spaces.
10. `print("All test cases passed!")`: Prints a success message if all `assert` statements pass.

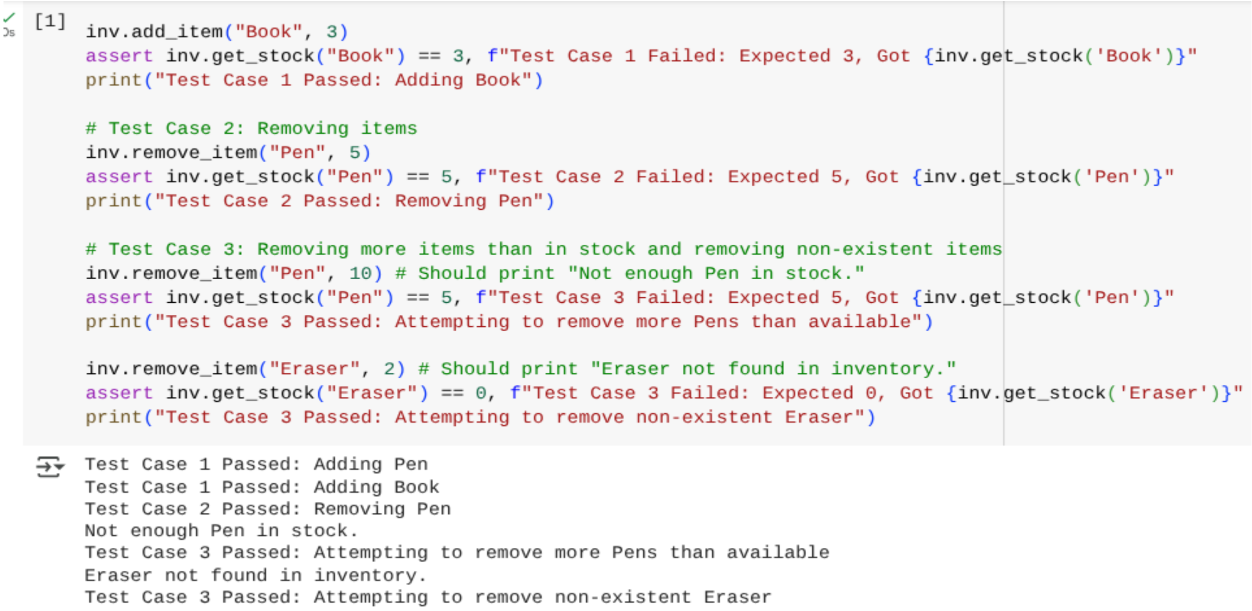
Task-4:

write a python function that generates inventory class using Methods: add_item(name, quantity) remove_item(name, quantity) get_stock(name).

Code and Output:



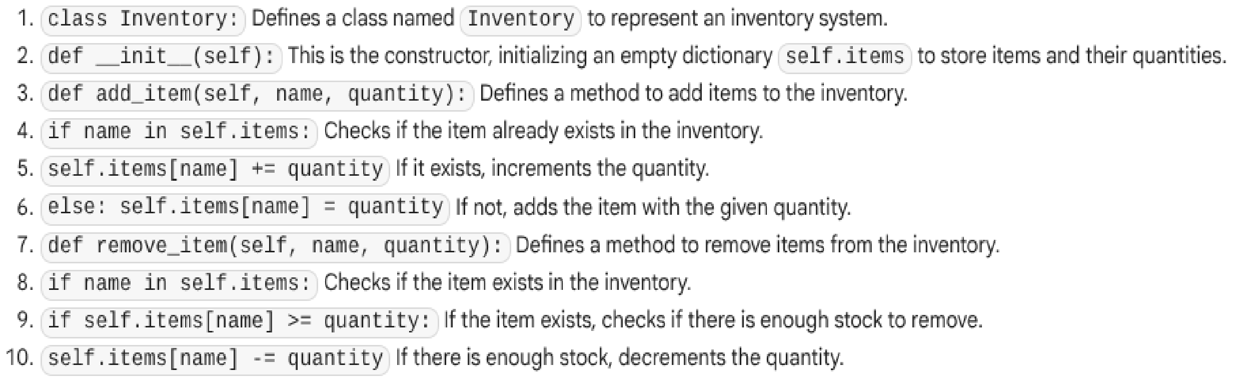
```python
class Inventory:
    def __init__(self):
        self.items = {}

    def add_item(self, name, quantity):
        if name in self.items:
            self.items[name] += quantity
        else:
            self.items[name] = quantity

    def remove_item(self, name, quantity):
        if name in self.items:
            if self.items[name] >= quantity:
                self.items[name] -= quantity
            else:
                print(f"Not enough {name} in stock.")
        else:
            print(f"{name} not found in inventory.")

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

# Example Assert Test Cases:
inv = Inventory()

# Test Case 1: Adding items
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10, f"Test Case 1 Failed: Expected 10, Got {inv.get_stock('Pen')}"
print("Test Case 1 Passed: Adding Pen")
```



```python
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3, f"Test Case 1 Failed: Expected 3, Got {inv.get_stock('Book')}"
print("Test Case 1 Passed: Adding Book")

# Test Case 2: Removing items
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5, f"Test Case 2 Failed: Expected 5, Got {inv.get_stock('Pen')}"
print("Test Case 2 Passed: Removing Pen")

# Test Case 3: Removing more items than in stock and removing non-existent items
inv.remove_item("Pen", 10) # Should print "Not enough Pen in stock."
assert inv.get_stock("Pen") == 5, f"Test Case 3 Failed: Expected 5, Got {inv.get_stock('Pen')}"
print("Test Case 3 Passed: Attempting to remove more Pens than available")

inv.remove_item("Eraser", 2) # Should print "Eraser not found in inventory."
assert inv.get_stock("Eraser") == 0, f"Test Case 3 Failed: Expected 0, Got {inv.get_stock('Eraser')}"
print("Test Case 3 Passed: Attempting to remove non-existent Eraser")
```

```
Test Case 1 Passed: Adding Pen
Test Case 1 Passed: Adding Book
Test Case 2 Passed: Removing Pen
Not enough Pen in stock.
Test Case 3 Passed: Attempting to remove more Pens than available
Eraser not found in inventory.
Test Case 3 Passed: Attempting to remove non-existent Eraser
```

Explanation:

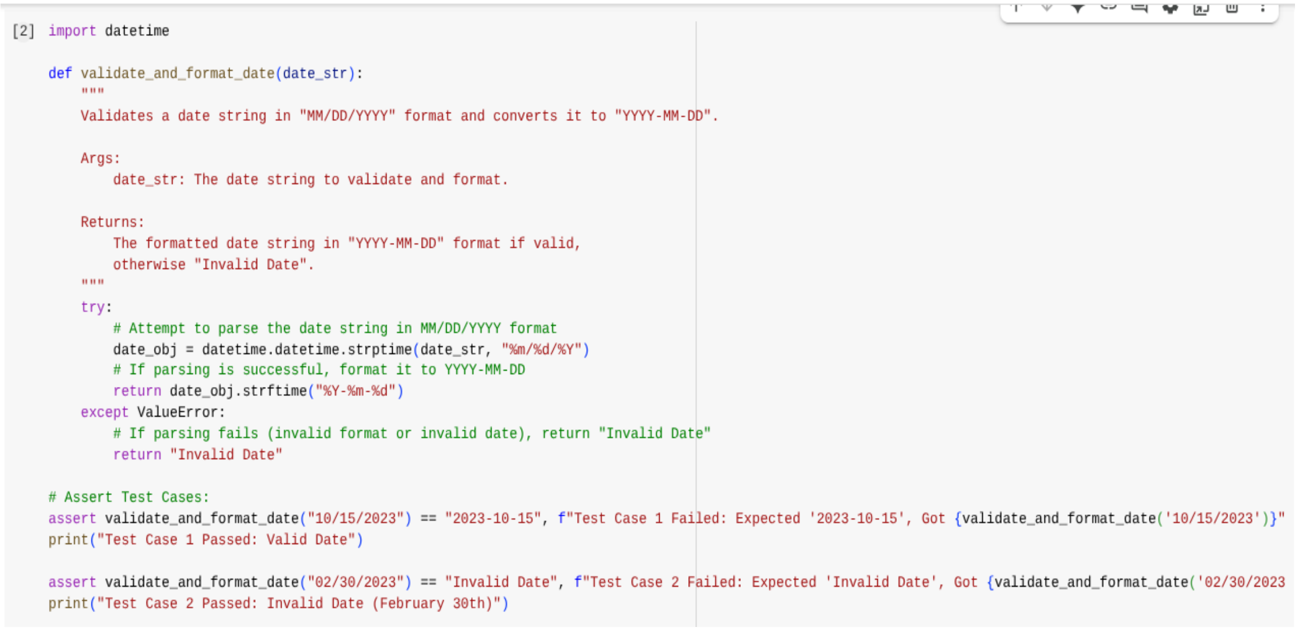


1. `class Inventory:` Defines a class named `Inventory` to represent an inventory system.
2. `def __init__(self):` This is the constructor, initializing an empty dictionary `self.items` to store items and their quantities.
3. `def add_item(self, name, quantity):` Defines a method to add items to the inventory.
4. `if name in self.items:` Checks if the item already exists in the inventory.
5. `self.items[name] += quantity` If it exists, increments the quantity.
6. `else: self.items[name] = quantity` If not, adds the item with the given quantity.
7. `def remove_item(self, name, quantity):` Defines a method to remove items from the inventory.
8. `if name in self.items:` Checks if the item exists in the inventory.
9. `if self.items[name] >= quantity:` If the item exists, checks if there is enough stock to remove.
10. `self.items[name] -= quantity` If there is enough stock, decrements the quantity.

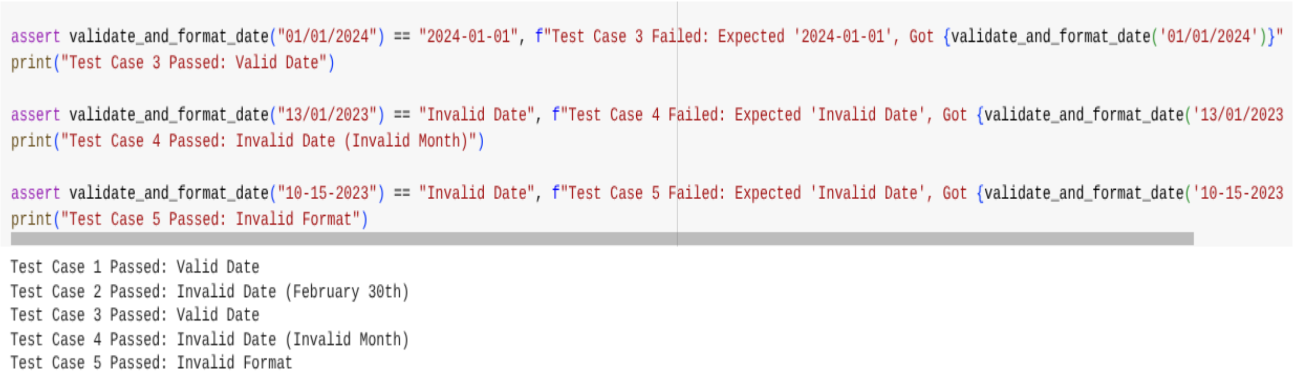
Task-5:

Date Validation & Formatting – for Data Validation 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" Make sure that Function passes all assertions and handles edge cases.

Code and Output:



```
[2] import datetime

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

        Args:
            date_str: The date string to validate and format.

        Returns:
            The formatted date string in "YYYY-MM-DD" format if valid,
            otherwise "Invalid Date".
        """
        try:
            # Attempt to parse the date string in MM/DD/YYYY format
            date_obj = datetime.datetime.strptime(date_str, "%m/%d/%Y")
            # If parsing is successful, format it to YYYY-MM-DD
            return date_obj.strftime("%Y-%m-%d")
        except ValueError:
            # If parsing fails (invalid format or invalid date), return "Invalid Date"
            return "Invalid Date"

    # Assert Test Cases:
    assert validate_and_format_date("10/15/2023") == "2023-10-15", f"Test Case 1 Failed: Expected '2023-10-15', Got {validate_and_format_date('10/15/2023')}"
    print("Test Case 1 Passed: Valid Date")

    assert validate_and_format_date("02/30/2023") == "Invalid Date", f"Test Case 2 Failed: Expected 'Invalid Date', Got {validate_and_format_date('02/30/2023
    print("Test Case 2 Passed: Invalid Date (February 30th)")
```



```
assert validate_and_format_date("01/01/2024") == "2024-01-01", f"Test Case 3 Failed: Expected '2024-01-01', Got {validate_and_format_date('01/01/2024')}"
print("Test Case 3 Passed: Valid Date")

assert validate_and_format_date("13/01/2023") == "Invalid Date", f"Test Case 4 Failed: Expected 'Invalid Date', Got {validate_and_format_date('13/01/2023
print("Test Case 4 Passed: Invalid Date (Invalid Month)")

assert validate_and_format_date("10-15-2023") == "Invalid Date", f"Test Case 5 Failed: Expected 'Invalid Date', Got {validate_and_format_date('10-15-2023
print("Test Case 5 Passed: Invalid Format")
```

```
Test Case 1 Passed: Valid Date
Test Case 2 Passed: Invalid Date (February 30th)
Test Case 3 Passed: Valid Date
Test Case 4 Passed: Invalid Date (Invalid Month)
Test Case 5 Passed: Invalid Format
```

Explanation:

1. `import datetime`: Imports the `datetime` module to work with dates and times.
2. `def validate_and_format_date(date_str):`: Defines a function named `validate_and_format_date` that takes a date string as input.
3. `try:` Starts a `try` block to handle potential errors during date parsing.
4. `date_obj = datetime.datetime.strptime(date_str, "%m/%d/%Y")`: Attempts to parse the input `date_str` assuming "MM/DD/YYYY" format.
5. `return date_obj.strftime("%Y-%m-%d")`: If parsing is successful, formats the date object into "YYYY-MM-DD" string and returns it.
6. `except ValueError:` Catches `ValueError` if `strptime` fails (due to invalid format or date).
7. `return "Invalid Date"`: If a `ValueError` occurs, returns the string "Invalid Date".
8. `assert validate_and_format_date("10/15/2023") == "2023-10-15", ...`: An assert statement to test a valid date.
9. `assert validate_and_format_date("02/30/2023") == "Invalid Date", ...`: An assert statement to test an invalid date (February 30th).
10. `assert validate_and_format_date("10-15-2023") == "Invalid Date", ...`: An assert statement to test an invalid format.