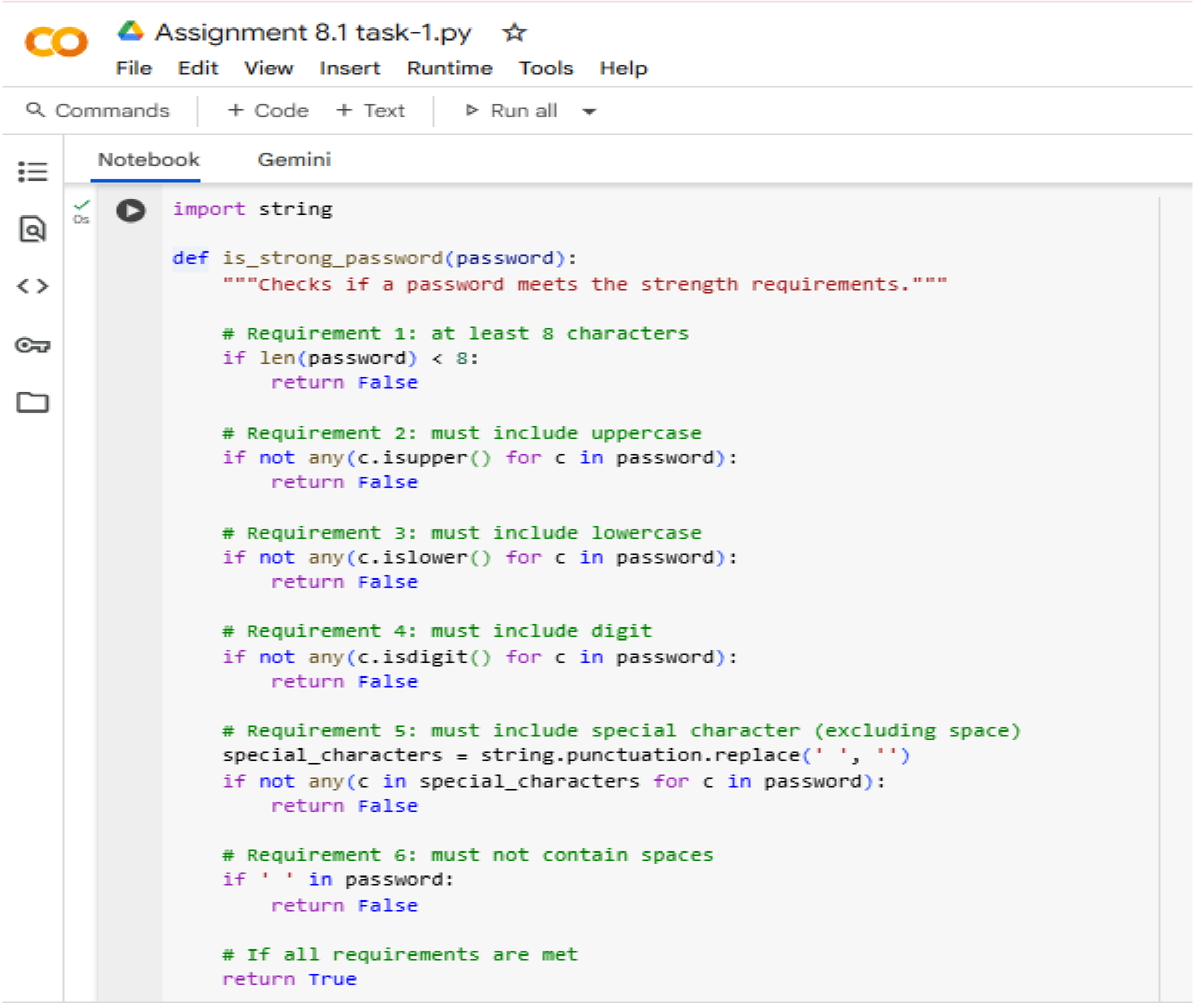
ASSIGNMENT-8.1

TASK-1:

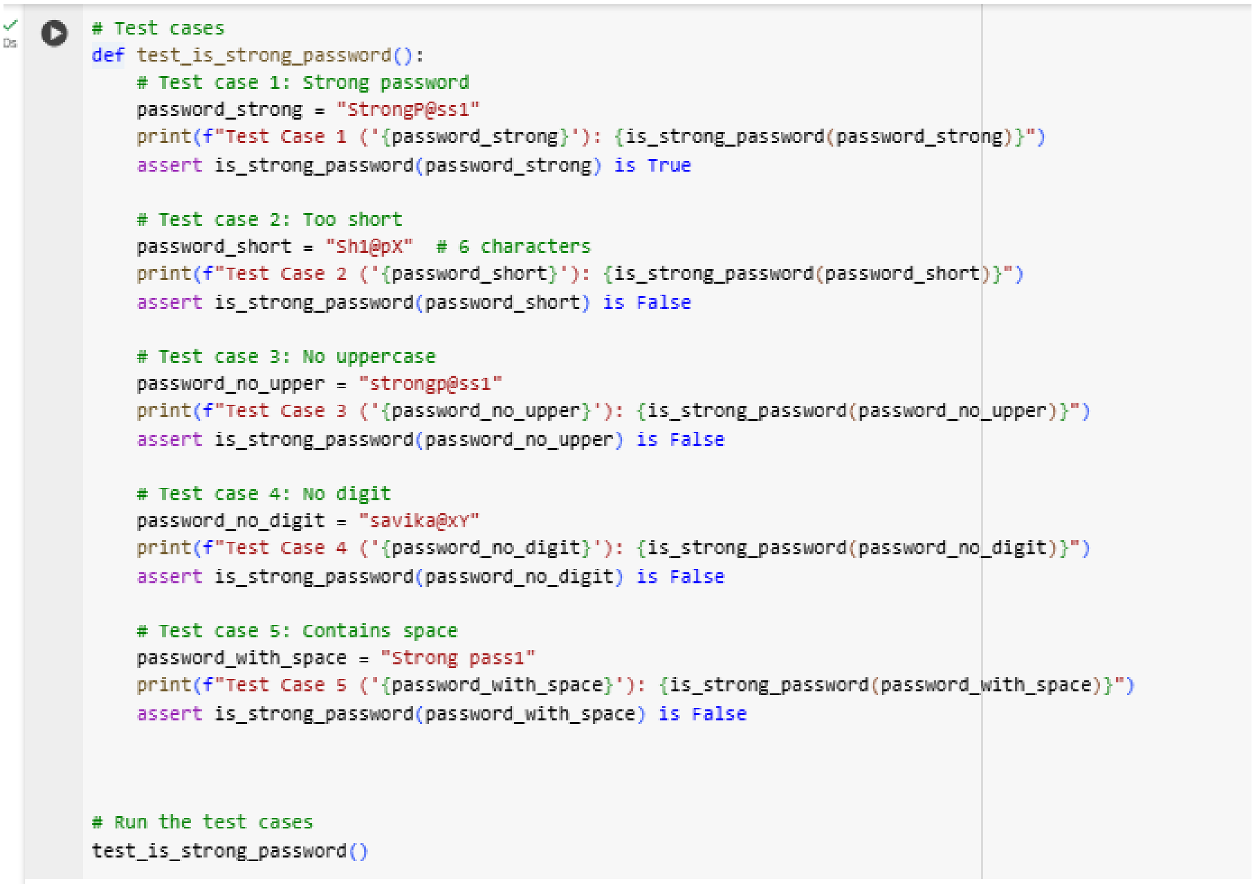
Prompt:

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

Code and Output:

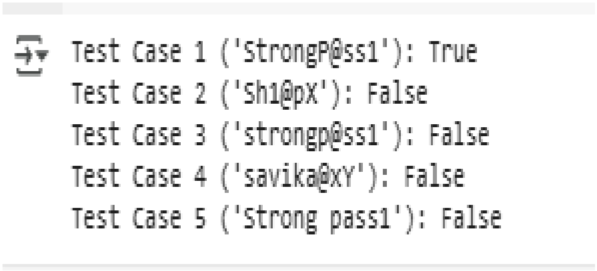

Assignment 8.1 task-1.py
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Notebook Gemini
0s

```
import string

def is_strong_password(password):
    """Checks if a password meets the strength requirements."""

    # Requirement 1: at least 8 characters
    if len(password) < 8:
        return False

    # Requirement 2: must include uppercase
    if not any(c.isupper() for c in password):
        return False

    # Requirement 3: must include lowercase
    if not any(c.islower() for c in password):
        return False

    # Requirement 4: must include digit
    if not any(c.isdigit() for c in password):
        return False

    # Requirement 5: must include special character (excluding space)
    special_characters = string.punctuation.replace(' ', '')
    if not any(c in special_characters for c in password):
        return False

    # Requirement 6: must not contain spaces
    if ' ' in password:
        return False

    # If all requirements are met
    return True
```



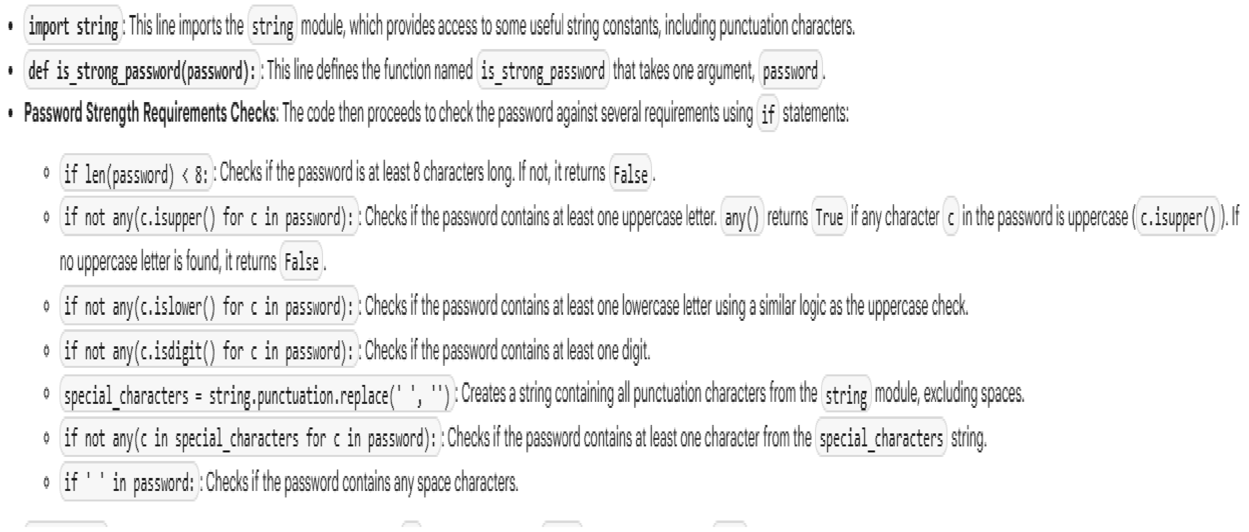
```python
# Test cases
def test_is_strong_password():
    # Test case 1: Strong password
    password_strong = "StrongP@ss1"
    print(f"Test Case 1 ('{password_strong}'): {is_strong_password(password_strong)}")
    assert is_strong_password(password_strong) is True

    # Test case 2: Too short
    password_short = "Sh1@pX"  # 6 characters
    print(f"Test Case 2 ('{password_short}'): {is_strong_password(password_short)}")
    assert is_strong_password(password_short) is False

    # Test case 3: No uppercase
    password_no_upper = "strongp@ss1"
    print(f"Test Case 3 ('{password_no_upper}'): {is_strong_password(password_no_upper)}")
    assert is_strong_password(password_no_upper) is False

    # Test case 4: No digit
    password_no_digit = "savika@xY"
    print(f"Test Case 4 ('{password_no_digit}'): {is_strong_password(password_no_digit)}")
    assert is_strong_password(password_no_digit) is False

    # Test case 5: Contains space
    password_with_space = "Strong pass1"
    print(f"Test Case 5 ('{password_with_space}'): {is_strong_password(password_with_space)}")
    assert is_strong_password(password_with_space) is False



# Run the test cases
test_is_strong_password()
```

Output:

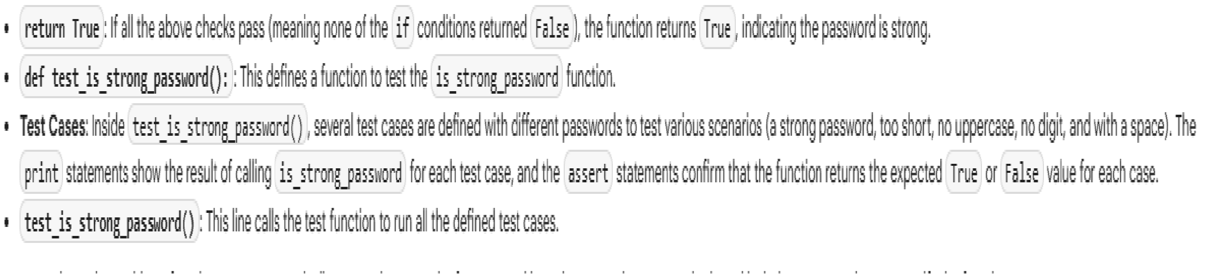


```
Test Case 1 ('StrongP@ss1'): True
Test Case 2 ('Sh1@pX'): False
Test Case 3 ('strongp@ss1'): False
Test Case 4 ('savika@xY'): False
Test Case 5 ('Strong pass1'): False
```

Explanation:



- `import string`: This line imports the `string` module, which provides access to some useful string constants, including punctuation characters.
- `def is_strong_password(password):` : This line defines the function named `is_strong_password` that takes one argument, `password`.
- **Password Strength Requirements Checks**: The code then proceeds to check the password against several requirements using `if` statements:
    - `if len(password) < 8:` : Checks if the password is at least 8 characters long. If not, it returns `False`.
    - `if not any(c.isupper() for c in password):` : Checks if the password contains at least one uppercase letter. `any()` returns `True` if any character `c` in the password is uppercase (`c.isupper()`). If no uppercase letter is found, it returns `False`.
    - `if not any(c.islower() for c in password):` : Checks if the password contains at least one lowercase letter using a similar logic as the uppercase check.
    - `if not any(c.isdigit() for c in password):` : Checks if the password contains at least one digit.
    - `special_characters = string.punctuation.replace(' ', '')` : Creates a string containing all punctuation characters from the `string` module, excluding spaces.
    - `if not any(c in special_characters for c in password):` : Checks if the password contains at least one character from the `special_characters` string.
    - `if ' ' in password:` : Checks if the password contains any space characters.



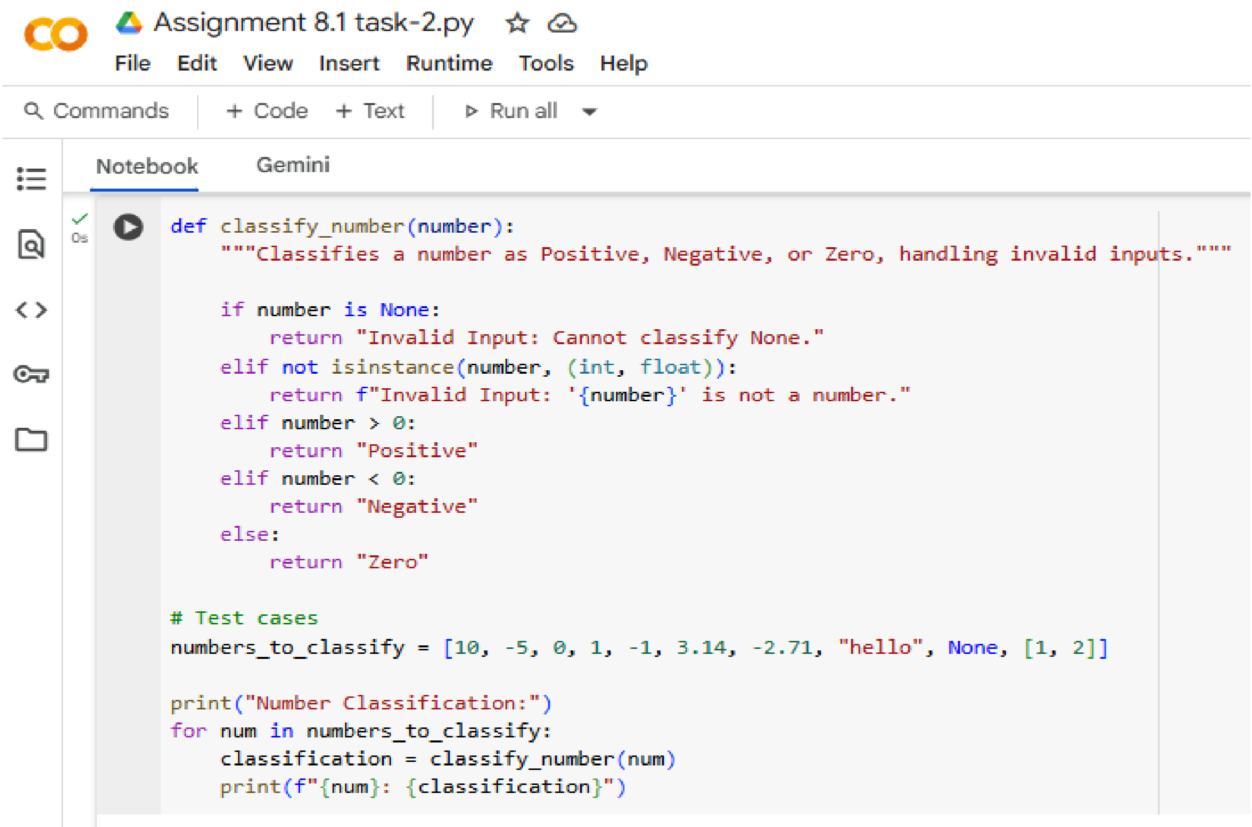
- `return True` : If all the above checks pass (meaning none of the `if` conditions returned `False`), the function returns `True`, indicating the password is strong.
- `def test_is_strong_password():` : This defines a function to test the `is_strong_password` function.
- **Test Cases**: Inside `test_is_strong_password()`, several test cases are defined with different passwords to test various scenarios (a strong password, too short, no uppercase, no digit, and with a space). The `print` statements show the result of calling `is_strong_password` for each test case, and the `assert` statements confirm that the function returns the expected `True` or `False` value for each case.
- `test_is_strong_password()` : This line calls the test function to run all the defined test cases.

TASK-2:

Prompt:

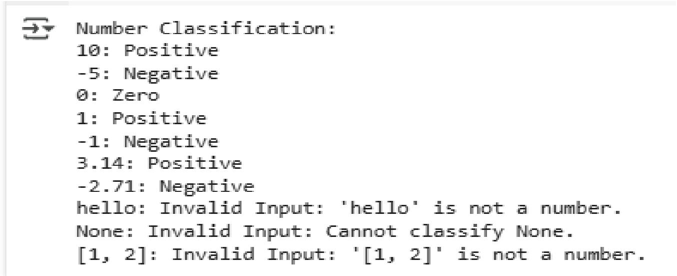
# write a python program to Number Classification with Loops. Requirements are Classify numbers as Positive, Negative or Zero and Handle invalid inputs like strings and none and include boundary conditions (-1, 0, 1).

Code and Output:



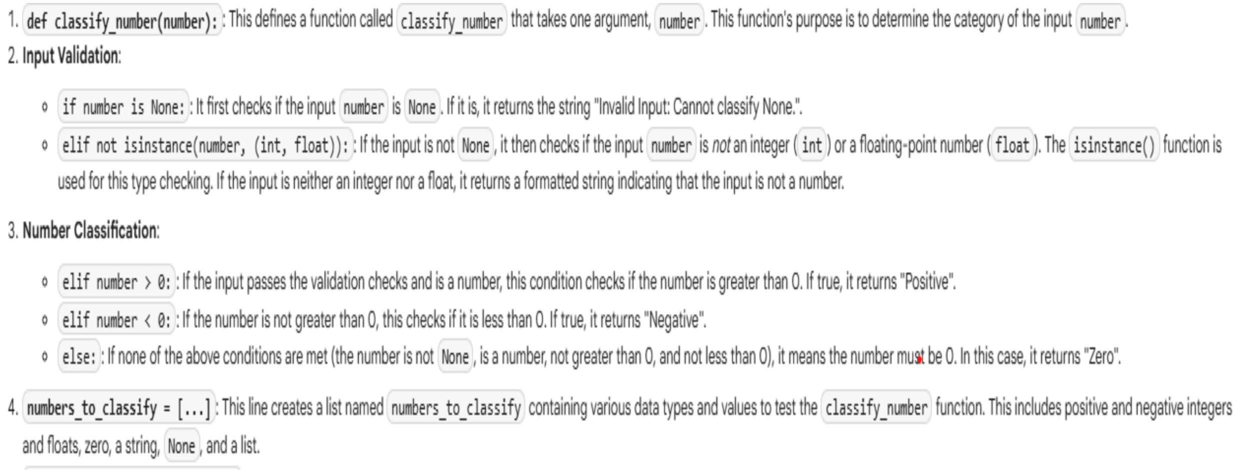
```python
def classify_number(number):
    """Classifies a number as Positive, Negative, or Zero, handling invalid inputs."""

    if number is None:
        return "Invalid Input: Cannot classify None."
    elif not isinstance(number, (int, float)):
        return f"Invalid Input: '{number}' is not a number."
    elif number > 0:
        return "Positive"
    elif number < 0:
        return "Negative"
    else:
        return "Zero"

# Test cases
numbers_to_classify = [10, -5, 0, 1, -1, 3.14, -2.71, "hello", None, [1, 2]]

print("Number Classification:")
for num in numbers_to_classify:
    classification = classify_number(num)
    print(f"{num}: {classification}")
```

Output:

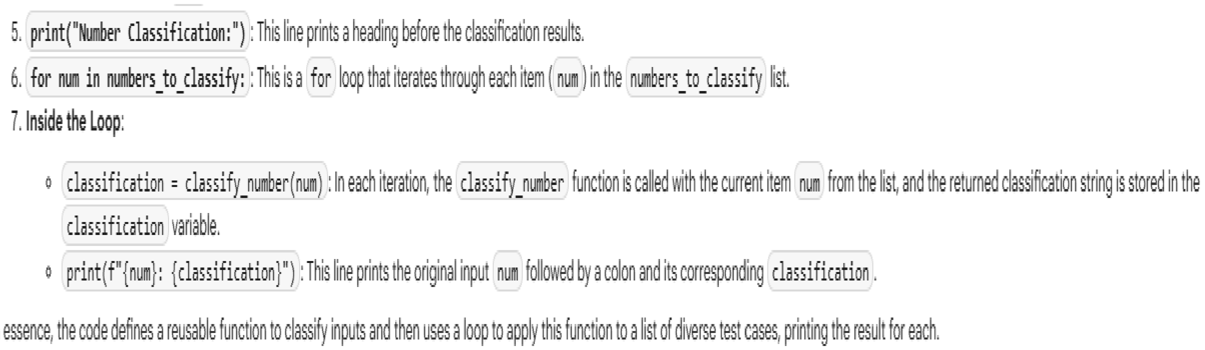


```
Number Classification:
10: Positive
-5: Negative
0: Zero
1: Positive
-1: Negative
3.14: Positive
-2.71: Negative
hello: Invalid Input: 'hello' is not a number.
None: Invalid Input: Cannot classify None.
[1, 2]: Invalid Input: '[1, 2]' is not a number.
```

Explanation:



1. `def classify_number(number):` : This defines a function called `classify_number` that takes one argument, `number`. This function's purpose is to determine the category of the input `number`.
2. **Input Validation**:
    - `if number is None:` : It first checks if the input `number` is `None`. If it is, it returns the string "Invalid Input: Cannot classify None.".
    - `elif not isinstance(number, (int, float)):` : If the input is not `None`, it then checks if the input `number` is *not* an integer (`int`) or a floating-point number (`float`). The `isinstance()` function is used for this type checking. If the input is neither an integer nor a float, it returns a formatted string indicating that the input is not a number.
3. **Number Classification**:
    - `elif number > 0:` : If the input passes the validation checks and is a number, this condition checks if the number is greater than 0. If true, it returns "Positive".
    - `elif number < 0:` : If the number is not greater than 0, this checks if it is less than 0. If true, it returns "Negative".
    - `else:` : If none of the above conditions are met (the number is not `None`, is a number, not greater than 0, and not less than 0), it means the number must be 0. In this case, it returns "Zero".
4. `numbers_to_classify = [...]` : This line creates a list named `numbers_to_classify` containing various data types and values to test the `classify_number` function. This includes positive and negative integers and floats, zero, a string, `None`, and a list.



5. `print("Number Classification:")` : This line prints a heading before the classification results.
6. `for num in numbers_to_classify:` : This is a `for` loop that iterates through each item (`num`) in the `numbers_to_classify` list.
7. **Inside the Loop**:
    - `classification = classify_number(num)` : In each iteration, the `classify_number` function is called with the current item `num` from the list, and the returned classification string is stored in the `classification` variable.
    - `print(f"{num}: {classification}")` : This line prints the original input `num` followed by a colon and its corresponding `classification`.

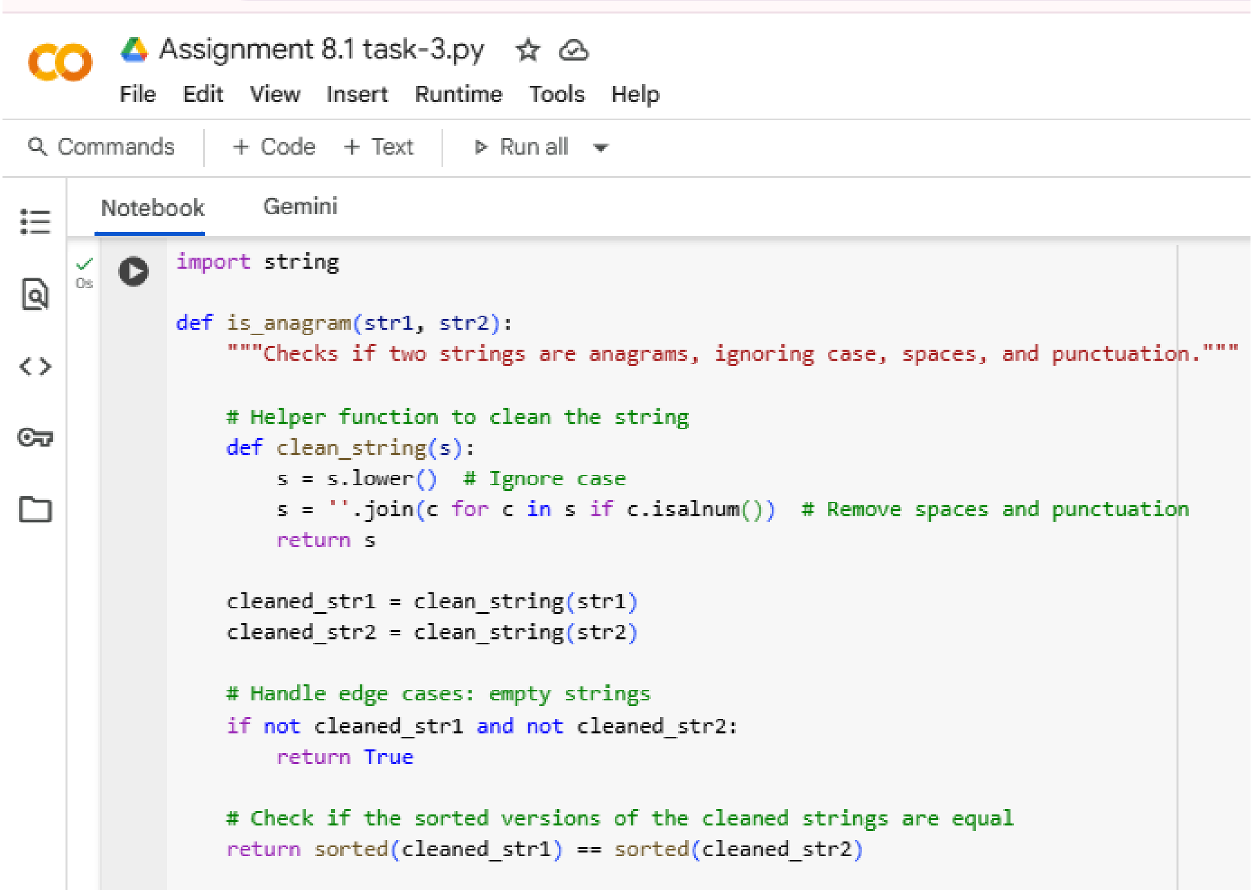
In essence, the code defines a reusable function to classify inputs and then uses a loop to apply this function to a list of diverse test cases, printing the result for each.

TASK-3:

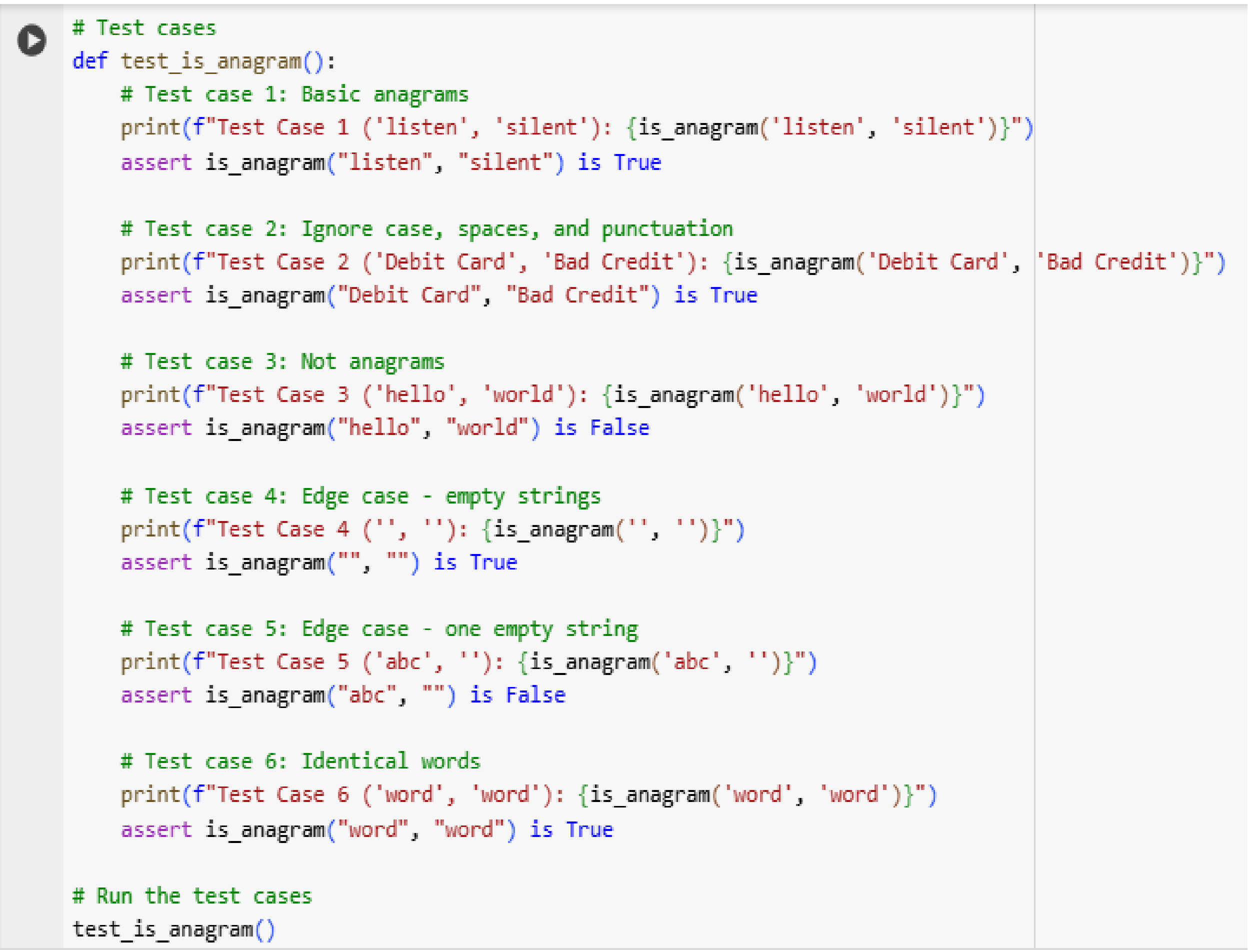
Prompt:

#write a python program generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function. Requirements are Ignore case, spaces, punctuation and Handle edge cases (empty strings, identical words).

Code and output:

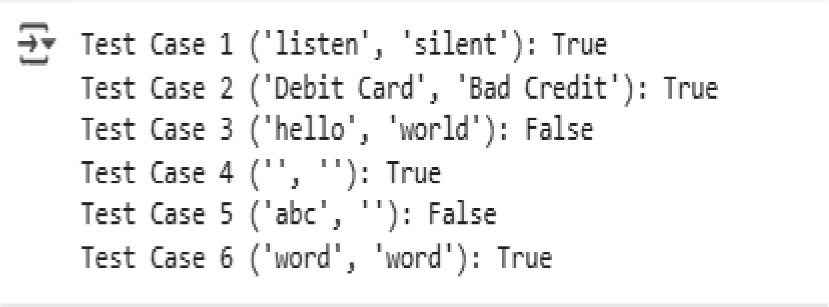


```python
import string

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

    # Helper function to clean the string
    def clean_string(s):
        s = s.lower()  # Ignore case
        s = ''.join(c for c in s if c.isalnum())  # Remove spaces and punctuation
        return s

    cleaned_str1 = clean_string(str1)
    cleaned_str2 = clean_string(str2)

    # Handle edge cases: empty strings
    if not cleaned_str1 and not cleaned_str2:
        return True

    # Check if the sorted versions of the cleaned strings are equal
    return sorted(cleaned_str1) == sorted(cleaned_str2)
```



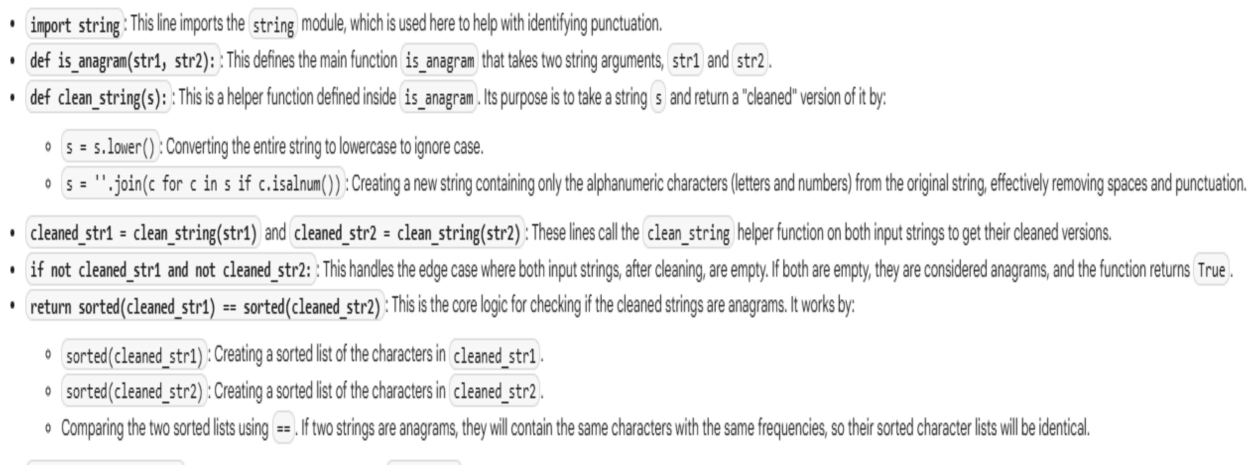
```
# Test cases
def test_is_anagram():
    # Test case 1: Basic anagrams
    print(f"Test Case 1 ('listen', 'silent'): {is_anagram('listen', 'silent')}")
    assert is_anagram("listen", "silent") is True

    # Test case 2: Ignore case, spaces, and punctuation
    print(f"Test Case 2 ('Debit Card', 'Bad Credit'): {is_anagram('Debit Card', 'Bad Credit')}")
    assert is_anagram("Debit Card", "Bad Credit") is True

    # Test case 3: Not anagrams
    print(f"Test Case 3 ('hello', 'world'): {is_anagram('hello', 'world')}")
    assert is_anagram("hello", "world") is False

    # Test case 4: Edge case - empty strings
    print(f"Test Case 4 ('', ''): {is_anagram('', '')}")
    assert is_anagram("", "") is True

    # Test case 5: Edge case - one empty string
    print(f"Test Case 5 ('abc', ''): {is_anagram('abc', '')}")
    assert is_anagram("abc", "") is False

    # Test case 6: Identical words
    print(f"Test Case 6 ('word', 'word'): {is_anagram('word', 'word')}")
    assert is_anagram("word", "word") is True

# Run the test cases
test_is_anagram()
```

Output:

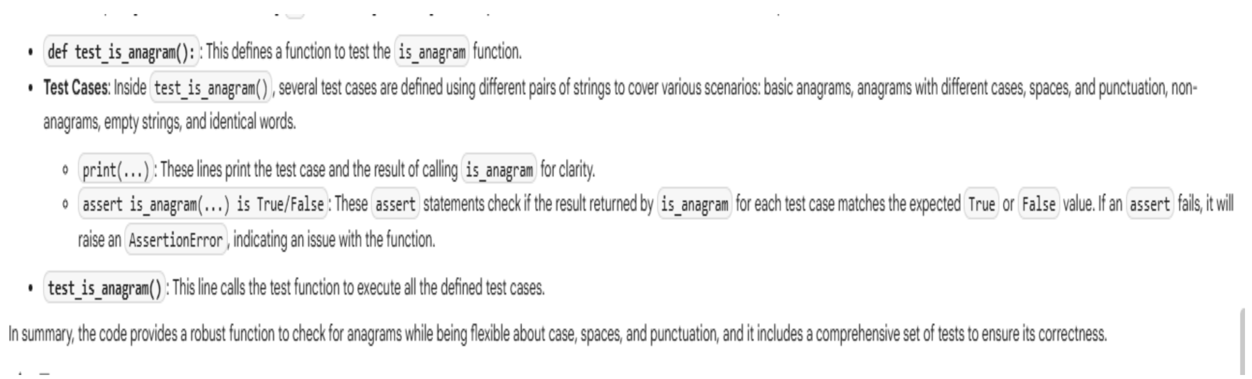


```
Test Case 1 ('listen', 'silent'): True
Test Case 2 ('Debit Card', 'Bad Credit'): True
Test Case 3 ('hello', 'world'): False
Test Case 4 ('', ''): True
Test Case 5 ('abc', ''): False
Test Case 6 ('word', 'word'): True
```

Explanation:



- `import string`: This line imports the `string` module, which is used here to help with identifying punctuation.
- `def is_anagram(str1, str2):`: This defines the main function `is_anagram` that takes two string arguments, `str1` and `str2`.
- `def clean_string(s):`: This is a helper function defined inside `is_anagram`. Its purpose is to take a string `s` and return a "cleaned" version of it by:
  - `s = s.lower()`: Converting the entire string to lowercase to ignore case.
  - `s = ''.join(c for c in s if c.isalnum())`: Creating a new string containing only the alphanumeric characters (letters and numbers) from the original string, effectively removing spaces and punctuation.
- `cleaned_str1 = clean_string(str1)` and `cleaned_str2 = clean_string(str2)`: These lines call the `clean_string` helper function on both input strings to get their cleaned versions.
- `if not cleaned_str1 and not cleaned_str2:`: This handles the edge case where both input strings, after cleaning, are empty. If both are empty, they are considered anagrams, and the function returns `True`.
- `return sorted(cleaned_str1) == sorted(cleaned_str2)`: This is the core logic for checking if the cleaned strings are anagrams. It works by:
  - `sorted(cleaned_str1)`: Creating a sorted list of the characters in `cleaned_str1`.
  - `sorted(cleaned_str2)`: Creating a sorted list of the characters in `cleaned_str2`.
  - Comparing the two sorted lists using `==`. If two strings are anagrams, they will contain the same characters with the same frequencies, so their sorted character lists will be identical.



- `def test_is_anagram():`: This defines a function to test the `is_anagram` function.
- **Test Cases**: Inside `test_is_anagram()`, several test cases are defined using different pairs of strings to cover various scenarios: basic anagrams, anagrams with different cases, spaces, and punctuation, non-anagrams, empty strings, and identical words.
  - `print(...)`: These lines print the test case and the result of calling `is_anagram` for clarity.
  - `assert is_anagram(...) is True/False`: These `assert` statements check if the result returned by `is_anagram` for each test case matches the expected `True` or `False` value. If an `assert` fails, it will raise an `AssertionError`, indicating an issue with the function.
- `test_is_anagram()`: This line calls the test function to execute all the defined test cases.

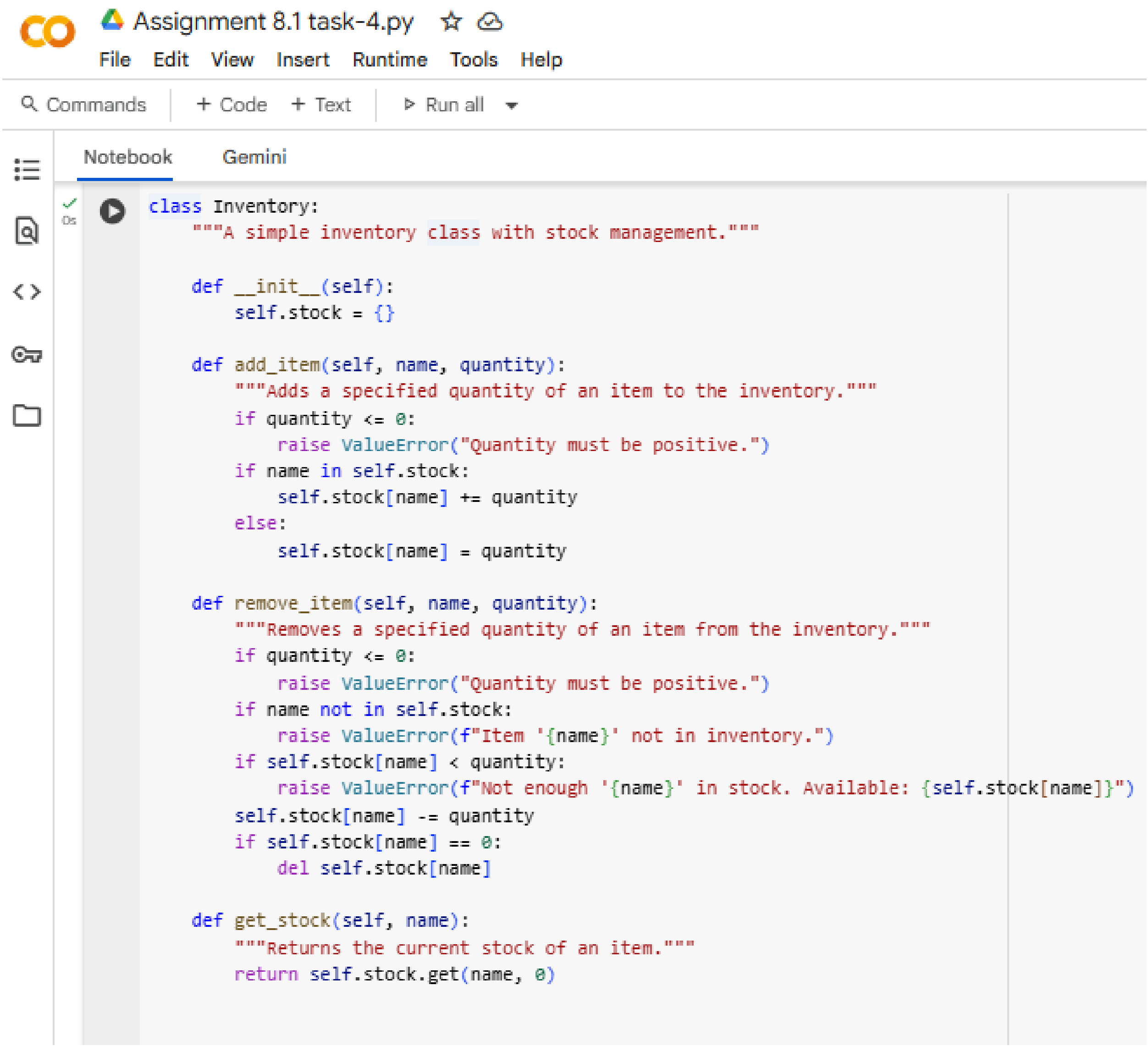
In summary, the code provides a robust function to check for anagrams while being flexible about case, spaces, and punctuation, and it includes a comprehensive set of tests to ensure its correctness.

Task-4:

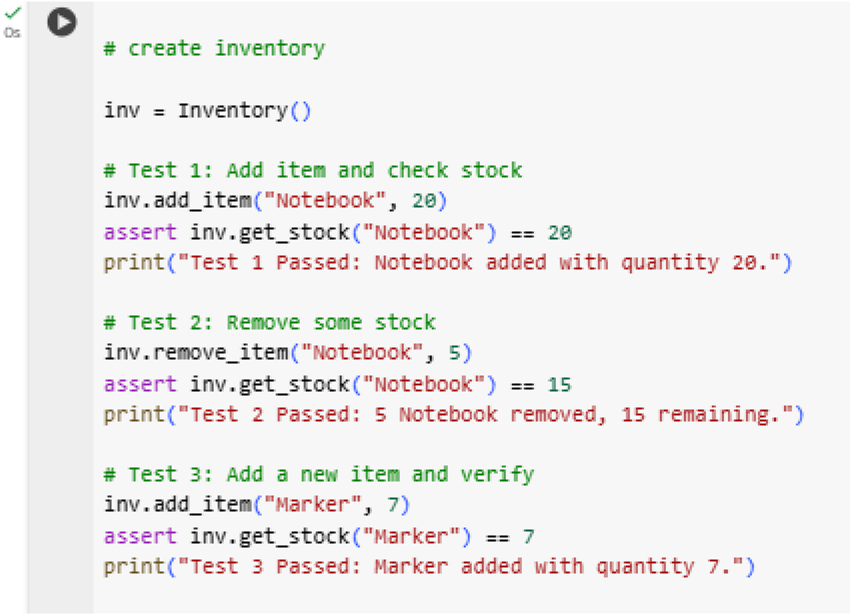
Prompt:

# write a python program to generate at least 3 assert-based tests for an Inventory class with stock management. Methods are add_item(name, quantity),remove_item(name, quantity) and get_stock(name).

Code and Output:

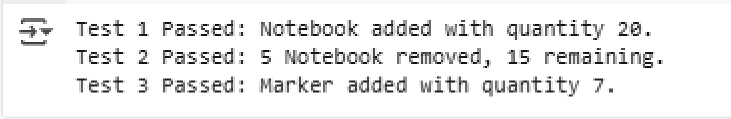


```
class Inventory:
    """A simple inventory class with stock management."""

    def __init__(self):
        self.stock = {}

    def add_item(self, name, quantity):
        """Adds a specified quantity of an item to the inventory."""
        if quantity <= 0:
            raise ValueError("Quantity must be positive.")
        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 quantity <= 0:
            raise ValueError("Quantity must be positive.")
        if name not in self.stock:
            raise ValueError(f"Item '{name}' not in inventory.")
        if self.stock[name] < quantity:
            raise ValueError(f"Not enough '{name}' in stock. Available: {self.stock[name]}")
        self.stock[name] -= quantity
        if self.stock[name] == 0:
            del self.stock[name]

    def get_stock(self, name):
        """Returns the current stock of an item."""
        return self.stock.get(name, 0)
```



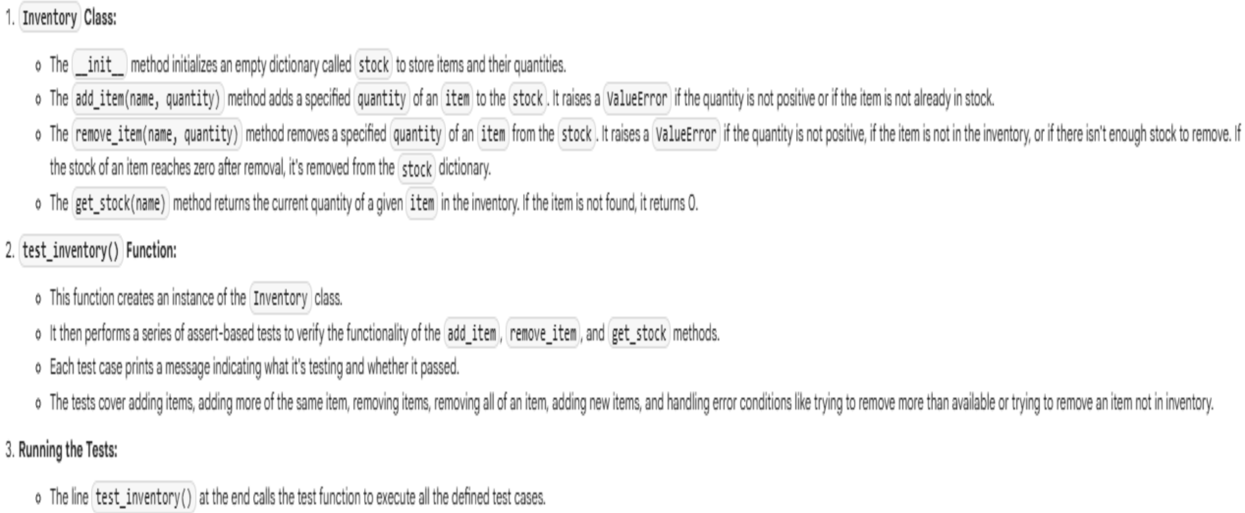
```python
# create inventory

inv = Inventory()

# Test 1: Add item and check stock
inv.add_item("Notebook", 20)
assert inv.get_stock("Notebook") == 20
print("Test 1 Passed: Notebook added with quantity 20.")

# Test 2: Remove some stock
inv.remove_item("Notebook", 5)
assert inv.get_stock("Notebook") == 15
print("Test 2 Passed: 5 Notebook removed, 15 remaining.")

# Test 3: Add a new item and verify
inv.add_item("Marker", 7)
assert inv.get_stock("Marker") == 7
print("Test 3 Passed: Marker added with quantity 7.")
```

Output:



```
Test 1 Passed: Notebook added with quantity 20.
Test 2 Passed: 5 Notebook removed, 15 remaining.
Test 3 Passed: Marker added with quantity 7.
```

Explanation:



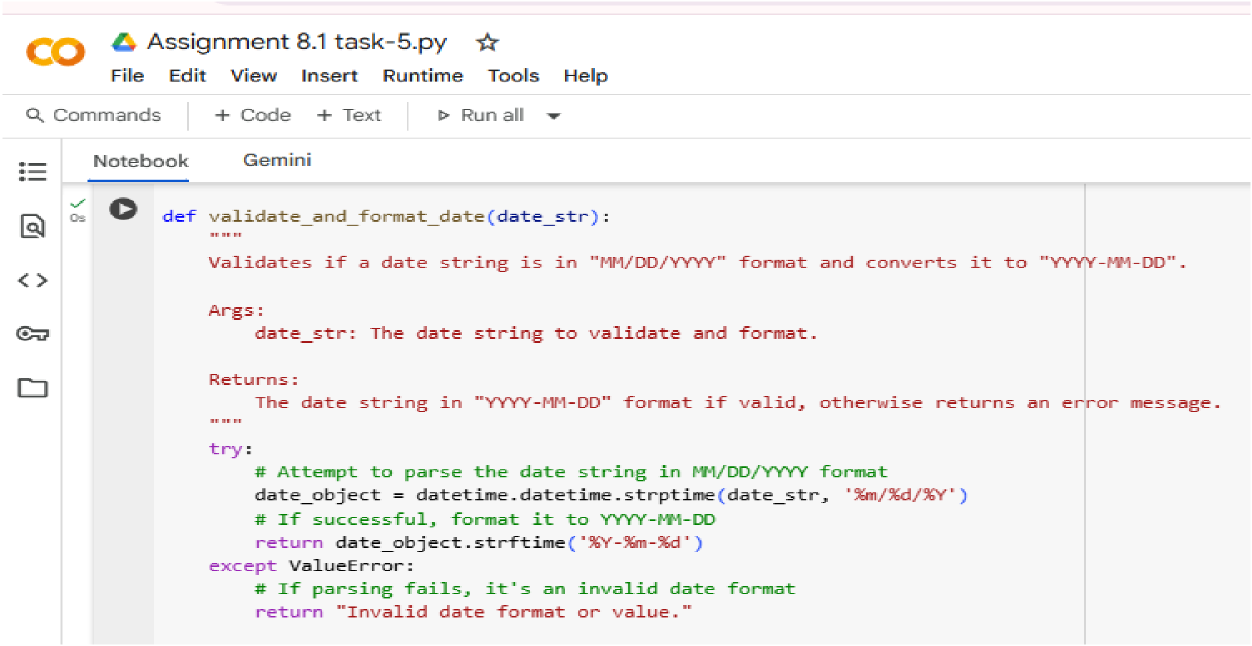
1. `Inventory` **Class:**
   - The `_init_` method initializes an empty dictionary called `stock` to store items and their quantities.
   - The `add_item(name, quantity)` method adds a specified `quantity` of an `item` to the `stock`. It raises a `ValueError` if the quantity is not positive or if the item is not already in stock.
   - The `remove_item(name, quantity)` method removes a specified `quantity` of an `item` from the `stock`. It raises a `ValueError` if the quantity is not positive, if the item is not in the inventory, or if there isn't enough stock to remove. If the stock of an item reaches zero after removal, it's removed from the `stock` dictionary.
   - The `get_stock(name)` method returns the current quantity of a given `item` in the inventory. If the item is not found, it returns 0.
2. `test_inventory()` **Function:**
   - This function creates an instance of the `Inventory` class.
   - It then performs a series of assert-based tests to verify the functionality of the `add_item`, `remove_item`, and `get_stock` methods.
   - Each test case prints a message indicating what it's testing and whether it passed.
   - The tests cover adding items, adding more of the same item, removing items, removing all of an item, adding new items, and handling error conditions like trying to remove more than available or trying to remove an item not in inventory.
3. **Running the Tests:**
   - The line `test_inventory()` at the end calls the test function to execute all the defined test cases.

TASK-5:

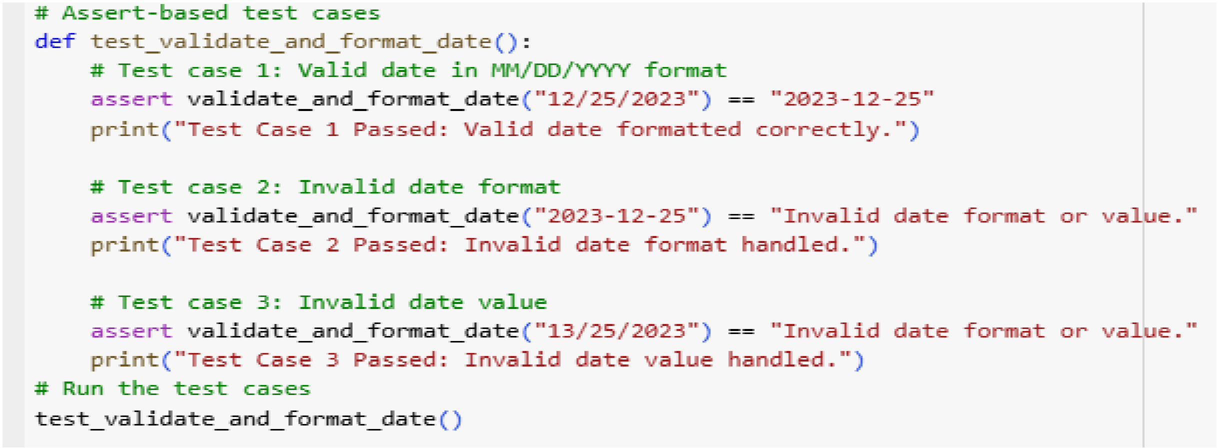
Prompt:

#write a python program to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates. Requirements are Validate "MM/DD/YYYY" format, Handle invalid dates and Convert valid dates to "YYYY-MM-DD".

Code and Output:

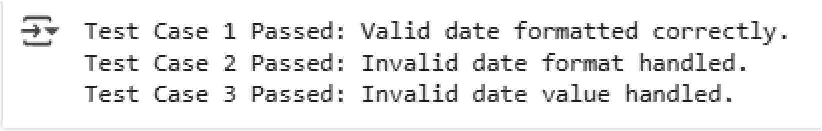


```python
def validate_and_format_date(date_str):
    """
    Validates if a date string is in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".

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

    Returns:
        The date string in "YYYY-MM-DD" format if valid, otherwise returns an error message.
    """
    try:
        # Attempt to parse the date string in MM/DD/YYYY format
        date_object = datetime.datetime.strptime(date_str, '%m/%d/%Y')
        # If successful, format it to YYYY-MM-DD
        return date_object.strftime('%Y-%m-%d')
    except ValueError:
        # If parsing fails, it's an invalid date format
        return "Invalid date format or value."
```



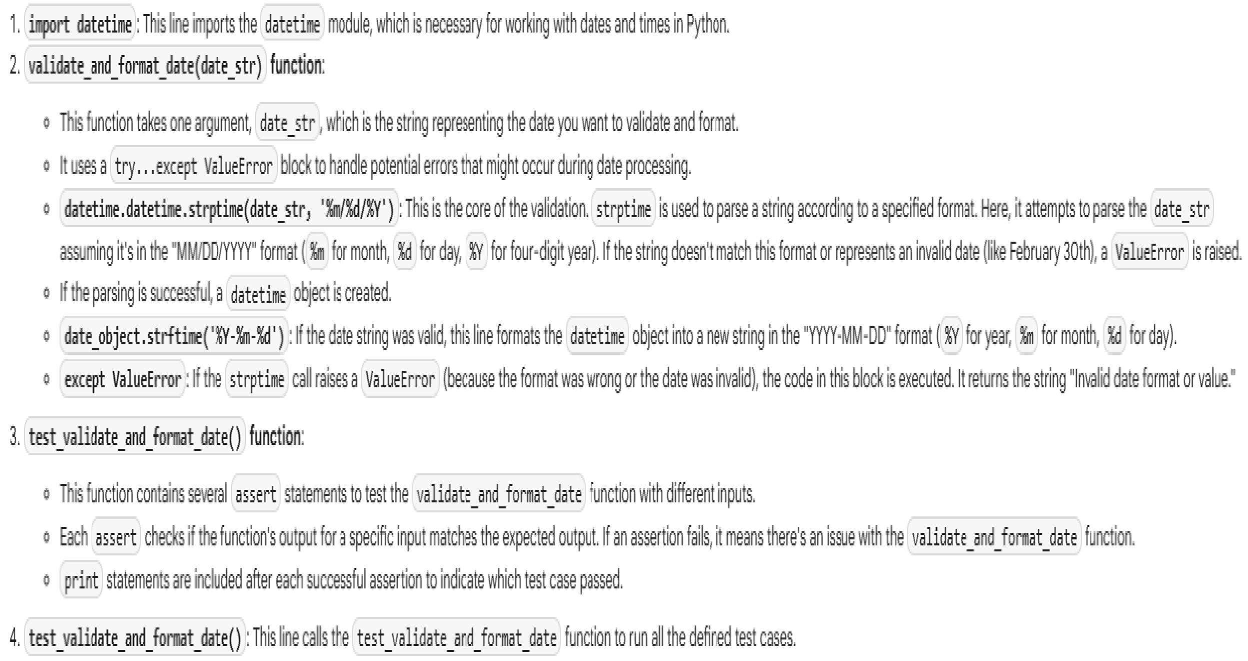
```python
# Assert-based test cases
def test_validate_and_format_date():
    # Test case 1: Valid date in MM/DD/YYYY format
    assert validate_and_format_date("12/25/2023") == "2023-12-25"
    print("Test Case 1 Passed: Valid date formatted correctly.")

    # Test case 2: Invalid date format
    assert validate_and_format_date("2023-12-25") == "Invalid date format or value."
    print("Test Case 2 Passed: Invalid date format handled.")

    # Test case 3: Invalid date value
    assert validate_and_format_date("13/25/2023") == "Invalid date format or value."
    print("Test Case 3 Passed: Invalid date value handled.")
# Run the test cases
test_validate_and_format_date()
```

Output:



```
Test Case 1 Passed: Valid date formatted correctly.
Test Case 2 Passed: Invalid date format handled.
Test Case 3 Passed: Invalid date value handled.
```

Explanation:



1. `import datetime` : This line imports the `datetime` module, which is necessary for working with dates and times in Python.
2. `validate_and_format_date(date_str)` **function**:
    - This function takes one argument, `date_str`, which is the string representing the date you want to validate and format.
    - It uses a `try...except ValueError` block to handle potential errors that might occur during date processing.
    - `datetime.datetime.strptime(date_str, '%m/%d/%Y')` : This is the core of the validation. `strptime` is used to parse a string according to a specified format. Here, it attempts to parse the `date_str` assuming it's in the "MM/DD/YYYY" format (`%m` for month, `%d` for day, `%Y` for four-digit year). If the string doesn't match this format or represents an invalid date (like February 30th), a `ValueError` is raised.
    - If the parsing is successful, a `datetime` object is created.
    - `date_object.strftime('%Y-%m-%d')` : If the date string was valid, this line formats the `datetime` object into a new string in the "YYYY-MM-DD" format (`%Y` for year, `%m` for month, `%d` for day).
    - `except ValueError` : If the `strptime` call raises a `ValueError` (because the format was wrong or the date was invalid), the code in this block is executed. It returns the string "Invalid date format or value."
3. `test_validate_and_format_date()` **function**:
    - This function contains several `assert` statements to test the `validate_and_format_date` function with different inputs.
    - Each `assert` checks if the function's output for a specific input matches the expected output. If an assertion fails, it means there's an issue with the `validate_and_format_date` function.
    - `print` statements are included after each successful assertion to indicate which test case passed.
4. `test_validate_and_format_date()` : This line calls the `test_validate_and_format_date` function to run all the defined test cases.