AI ASSISTED CODING

Assignment-8:

Test-Driven Development with AI – Generating and Working with Test Cases

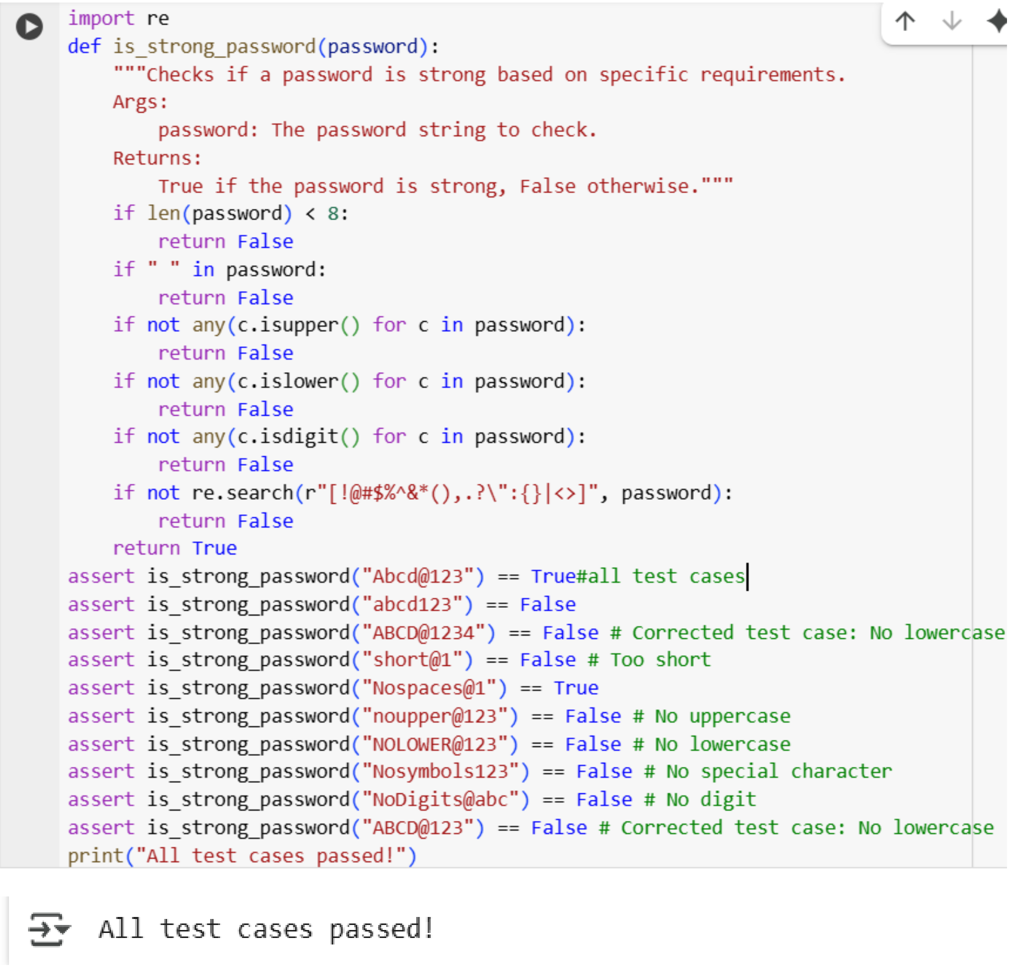
2403A52107

Pavani Voddepalli

Batch : 24BTCAIAI05

TASK-1 : Generate 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.
Example Assert Test Cases:
assert is_strong_password("Abcd@123") == True
assert is_strong_password("abcd123") == False
assert is_strong_password("ABCD@1234") == True

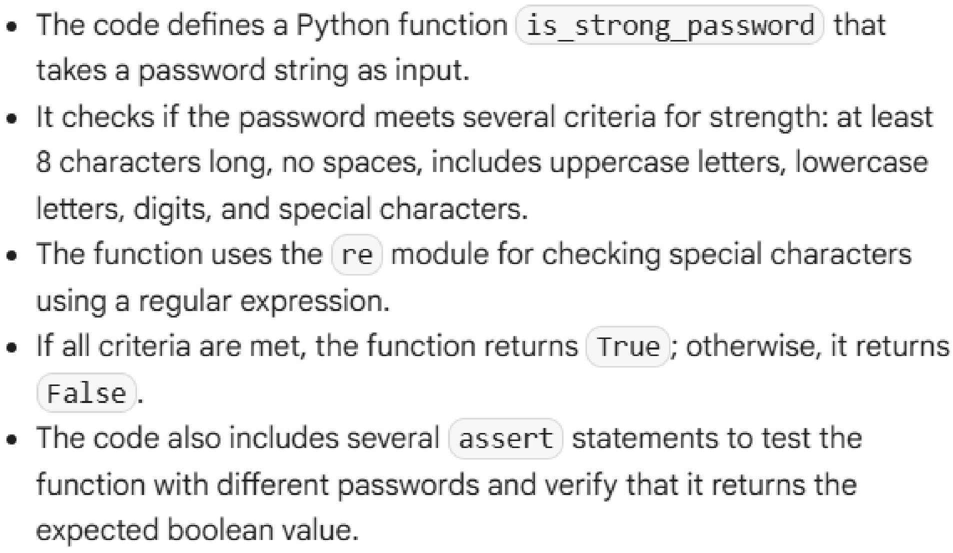
CODE & OUTPUT :



```python
import re
def is_strong_password(password):
    """Checks if a password is strong based on specific requirements.
    Args:
        password: The password string to check.
    Returns:
        True if the password is strong, False otherwise."""
    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
    if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
        return False
    return True
assert is_strong_password("Abcd@123") == True#all test cases
assert is_strong_password("abcd123") == False
assert is_strong_password("ABCD@1234") == False # Corrected test case: No lowercase
assert is_strong_password("short@1") == False # Too short
assert is_strong_password("Nospaces@1") == True
assert is_strong_password("noupper@123") == False # No uppercase
assert is_strong_password("NOLOWER@123") == False # No lowercase
assert is_strong_password("Nosymbols123") == False # No special character
assert is_strong_password("NoDigits@abc") == False # No digit
assert is_strong_password("ABCD@123") == False # Corrected test case: No lowercase
print("All test cases passed!")
```

```
All test cases passed!
```

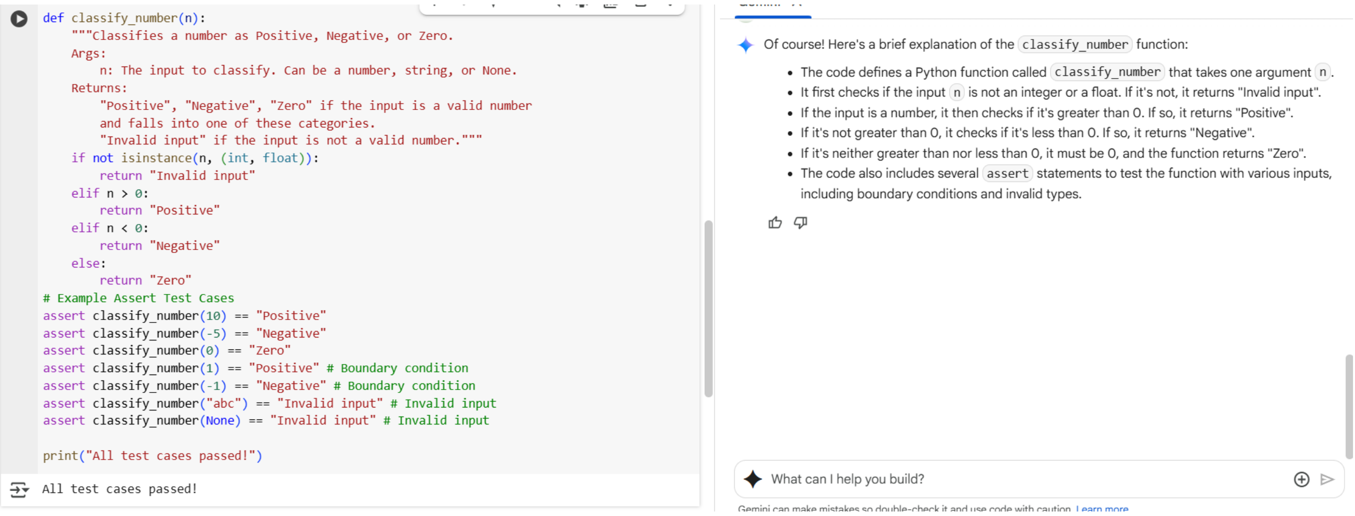
EXPLANATION:



- The code defines a Python function `is_strong_password` that takes a password string as input.
- It checks if the password meets several criteria for strength: at least 8 characters long, no spaces, includes uppercase letters, lowercase letters, digits, and special characters.
- The function uses the `re` module for checking special characters using a regular expression.
- If all criteria are met, the function returns `True`; otherwise, it returns `False`.
- The code also includes several `assert` statements to test the function with different passwords and verify that it returns the expected boolean value.

TASK-2 : Generate a python function classify_number(n) to Implement using loops. • Requirements: o Classify numbers as Positive, Negative, or Zero. o Handle invalid inputs like strings and None. o Include boundary conditions (-1, 0, 1).
Example Assert Test Cases:
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"

CODE , OUTPUT & EXPLANATION :



```
def classify_number(n):
    """Classifies a number as Positive, Negative, or Zero.
    Args:
        n: The input to classify. Can be a number, string, or None.
    Returns:
        "Positive", "Negative", "Zero" if the input is a valid number
        and falls into one of these categories.
        "Invalid input" if the input is not a valid number."""
    if not isinstance(n, (int, float)):
        return "Invalid input"
    elif n > 0:
        return "Positive"
    elif n < 0:
        return "Negative"
    else:
        return "Zero"
# Example Assert Test Cases
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
assert classify_number(1) == "Positive" # Boundary condition
assert classify_number(-1) == "Negative" # Boundary condition
assert classify_number("abc") == "Invalid input" # Invalid input
assert classify_number(None) == "Invalid input" # Invalid input

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

```
All test cases passed!
```

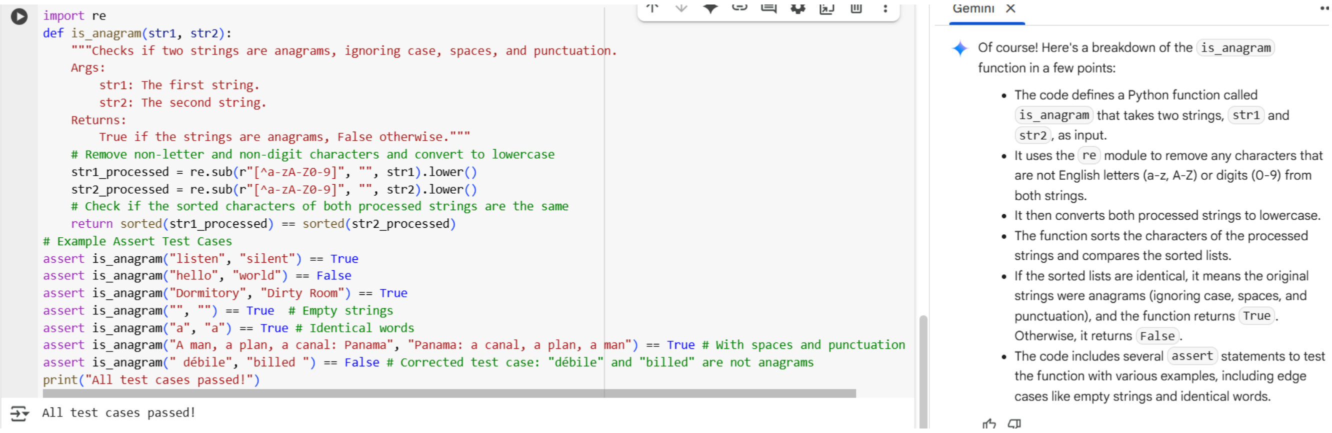
Of course! Here's a brief explanation of the `classify_number` function:

- The code defines a Python function called `classify_number` that takes one argument `n`.
- It first checks if the input `n` is not an integer or a float. If it's not, it returns "Invalid input".
- If the input is a number, it then checks if it's greater than 0. If so, it returns "Positive".
- If it's not greater than 0, it checks if it's less than 0. If so, it returns "Negative".
- If it's neither greater than nor less than 0, it must be 0, and the function returns "Zero".
- The code also includes several `assert` statements to test the function with various inputs, including boundary conditions and invalid types.

Gemini can make mistakes so double-check it and use code with caution. Learn more

TASK-3 : Generate a python function is_anagram(str1, str2) to 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 & EXPLANATION:



```python
import re
def is_anagram(str1, str2):
    """Checks if two strings are anagrams, ignoring case, spaces, and punctuation.
    Args:
        str1: The first string.
        str2: The second string.
    Returns:
        True if the strings are anagrams, False otherwise."""
    # Remove non-letter and non-digit characters 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 sorted characters of both processed strings are the same
    return sorted(str1_processed) == sorted(str2_processed)
# Example Assert Test Cases
assert is_anagram("listen", "silent") == True
assert is_anagram("hello", "world") == False
assert is_anagram("Dormitory", "Dirty Room") == True
assert is_anagram("", "") == True  # Empty strings
assert is_anagram("a", "a") == True # Identical words
assert is_anagram("A man, a plan, a canal: Panama", "Panama: a canal, a plan, a man") == True # With spaces and punctuation
assert is_anagram(" débile", "billed ") == False # Corrected test case: "débile" and "billed" are not anagrams
print("All test cases passed!")
```

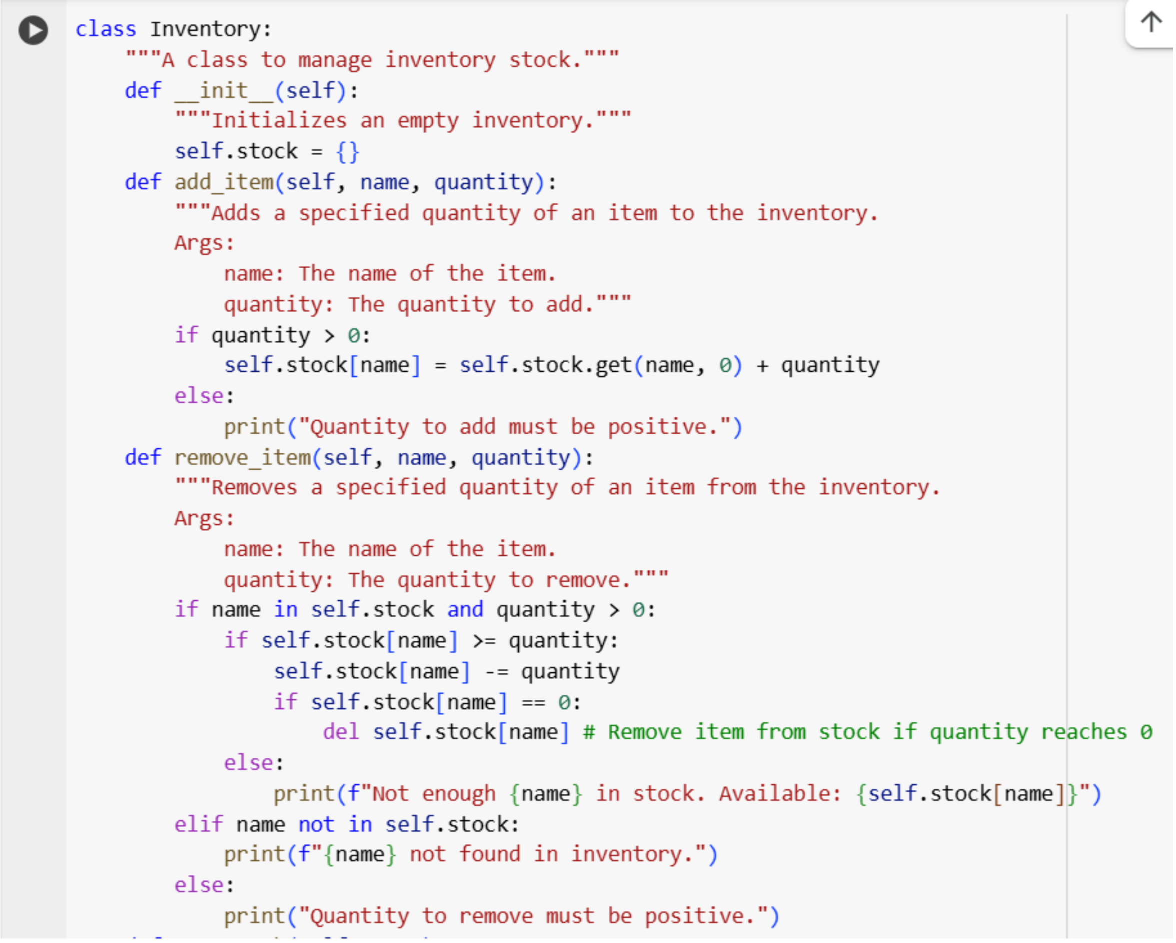
```
All test cases passed!
```

Of course! Here's a breakdown of the `is_anagram` function in a few points:

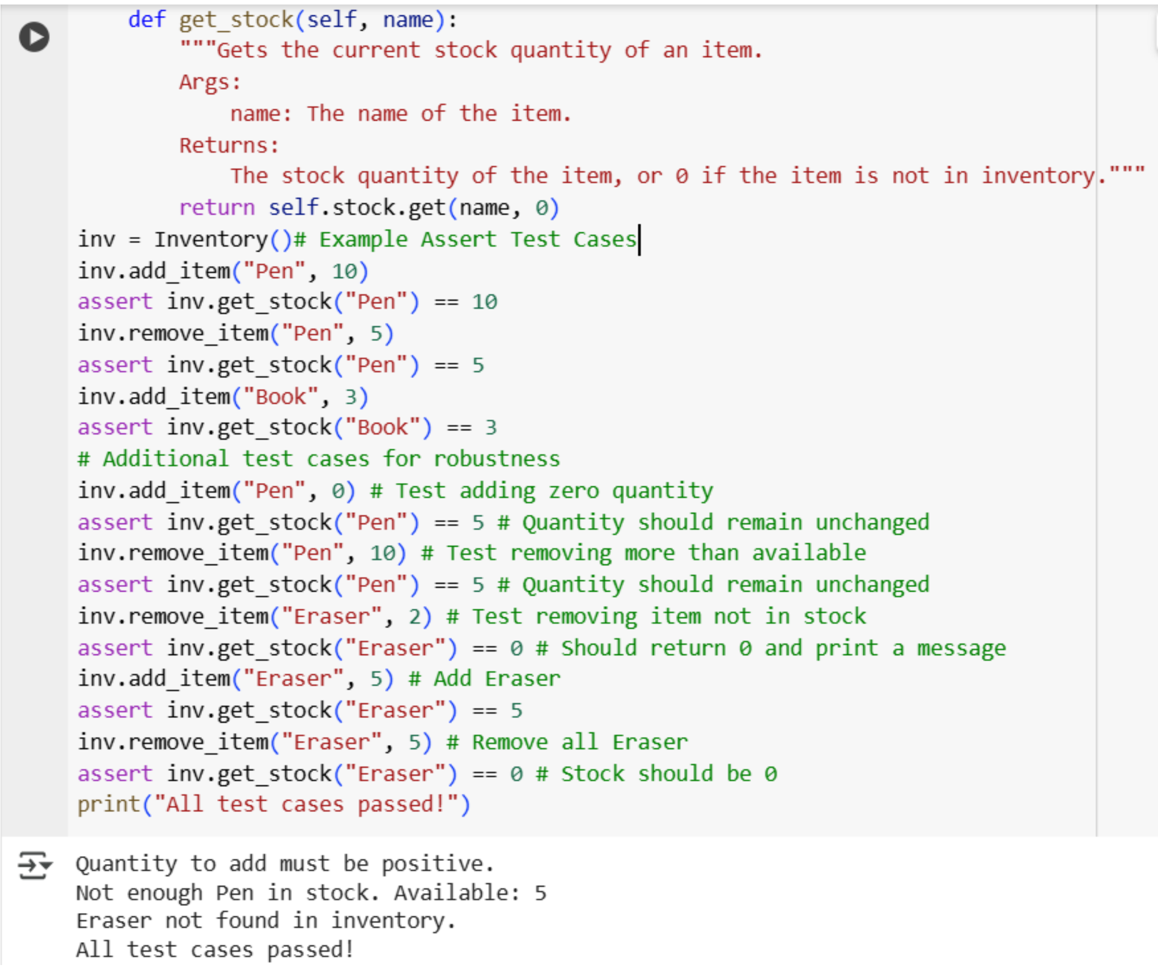
- The code defines a Python function called `is_anagram` that takes two strings, `str1` and `str2`, as input.
- It uses the `re` module to remove any characters that are not English letters (a-z, A-Z) or digits (0-9) from both strings.
- It then converts both processed strings to lowercase.
- The function sorts the characters of the processed strings and compares the sorted lists.
- If the sorted lists are identical, it means the original strings were anagrams (ignoring case, spaces, and punctuation), and the function returns `True`. Otherwise, it returns `False`.
- The code includes several `assert` statements to test the function with various examples, including edge cases like empty strings and identical words.

TASK-4 : Generate a python program to implement the Inventory class with stock management using 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 :



```python
class Inventory:
    """A class to manage inventory stock."""
    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.
        Args:
            name: The name of the item.
            quantity: The quantity to add."""
        if quantity > 0:
            self.stock[name] = self.stock.get(name, 0) + quantity
        else:
            print("Quantity to add must be positive.")
    def remove_item(self, name, quantity):
        """Removes a specified quantity of an item from the inventory.
        Args:
            name: The name of the item.
            quantity: The quantity to remove."""
        if name in self.stock and quantity > 0:
            if self.stock[name] >= quantity:
                self.stock[name] -= quantity
                if self.stock[name] == 0:
                    del self.stock[name] # Remove item from stock if quantity reaches 0
            else:
                print(f"Not enough {name} in stock. Available: {self.stock[name]}")
        elif name not in self.stock:
            print(f"{name} not found in inventory.")
        else:
            print("Quantity to remove must be positive.")
```



```python
    def get_stock(self, name):
        """Gets the current stock quantity of an item.
        Args:
            name: The name of the item.
        Returns:
            The stock quantity of the item, or 0 if the item is not in inventory."""
        return self.stock.get(name, 0)
inv = Inventory()# Example Assert Test Cases
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
# Additional test cases for robustness
inv.add_item("Pen", 0) # Test adding zero quantity
assert inv.get_stock("Pen") == 5 # Quantity should remain unchanged
inv.remove_item("Pen", 10) # Test removing more than available
assert inv.get_stock("Pen") == 5 # Quantity should remain unchanged
inv.remove_item("Eraser", 2) # Test removing item not in stock
assert inv.get_stock("Eraser") == 0 # Should return 0 and print a message
inv.add_item("Eraser", 5) # Add Eraser
assert inv.get_stock("Eraser") == 5
inv.remove_item("Eraser", 5) # Remove all Eraser
assert inv.get_stock("Eraser") == 0 # Stock should be 0
print("All test cases passed!")
```

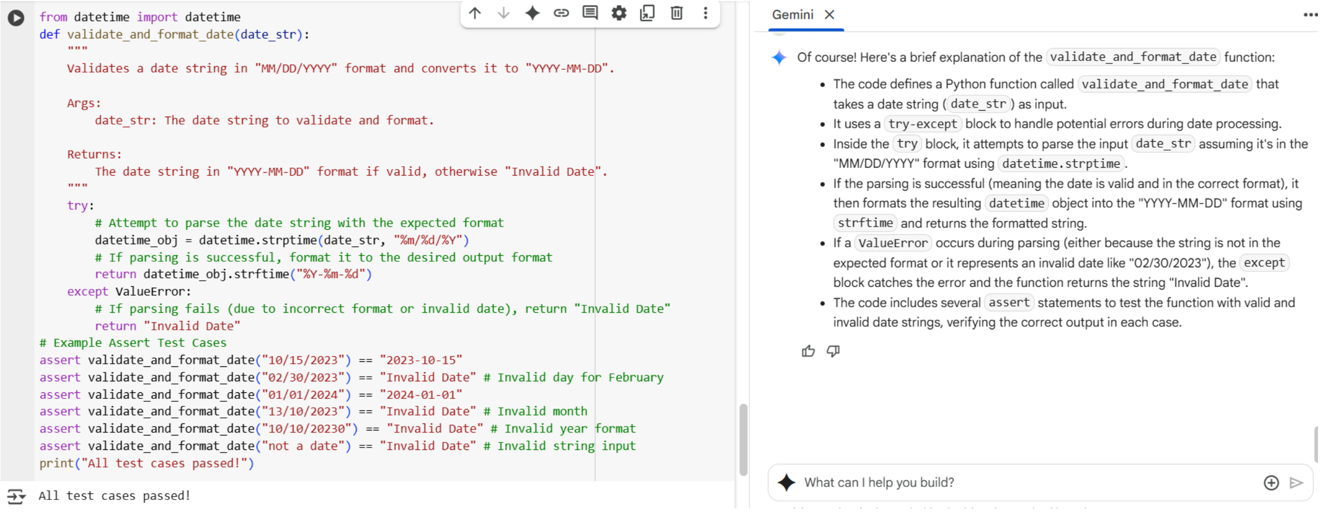
```
Quantity to add must be positive.
Not enough Pen in stock. Available: 5
Eraser not found in inventory.
All test cases passed!
```

EXPLANATION:

- The code defines a Python class named `Inventory` to manage stock.
- The `__init__` method initializes an empty dictionary called `self.stock` to store item names and their quantities.
- The `add_item` method takes an item `name` and `quantity` as input and adds the quantity to the item's stock. It ensures that only positive quantities are added.
- The `remove_item` method takes an item `name` and `quantity` and reduces the stock. It checks if the item exists and if there's enough stock before removing. If the stock reaches zero after removal, the item is removed from the dictionary. It also ensures that only positive quantities are removed.
- The `get_stock` method takes an item `name` and returns the current quantity of that item in stock. It returns 0 if the item is not found in the inventory.
- The code includes example test cases using `assert` to verify the functionality of the `add_item`, `remove_item`, and `get_stock` methods, including edge cases like adding zero quantity, removing more than available, and removing items not in stock.

TASK-5 : Generate a python function 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 & EXPLANATION :



```
from datetime 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 date string in "YYYY-MM-DD" format if valid, otherwise "Invalid Date".
    """
    try:
        # Attempt to parse the date string with the expected format
        datetime_obj = datetime.strptime(date_str, "%m/%d/%Y")
        # If parsing is successful, format it to the desired output format
        return datetime_obj.strftime("%Y-%m-%d")
    except ValueError:
        # If parsing fails (due to incorrect format or invalid date), return "Invalid Date"
        return "Invalid Date"
# 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" # Invalid day for February
assert validate_and_format_date("01/01/2024") == "2024-01-01"
assert validate_and_format_date("13/10/2023") == "Invalid Date" # Invalid month
assert validate_and_format_date("10/10/20230") == "Invalid Date" # Invalid year format
assert validate_and_format_date("not a date") == "Invalid Date" # Invalid string input
print("All test cases passed!")
```

```
All test cases passed!
```

Of course! Here's a brief explanation of the `validate_and_format_date` function:

- The code defines a Python function called `validate_and_format_date` that takes a date string (`date_str`) as input.
- It uses a `try-except` block to handle potential errors during date processing.
- Inside the `try` block, it attempts to parse the input `date_str` assuming it's in the "MM/DD/YYYY" format using `datetime.strptime`.
- If the parsing is successful (meaning the date is valid and in the correct format), it then formats the resulting `datetime` object into the "YYYY-MM-DD" format using `strftime` and returns the formatted string.
- If a `ValueError` occurs during parsing (either because the string is not in the expected format or it represents an invalid date like "02/30/2023"), the `except` block catches the error and the function returns the string "Invalid Date".
- The code includes several `assert` statements to test the function with valid and invalid date strings, verifying the correct output in each case.