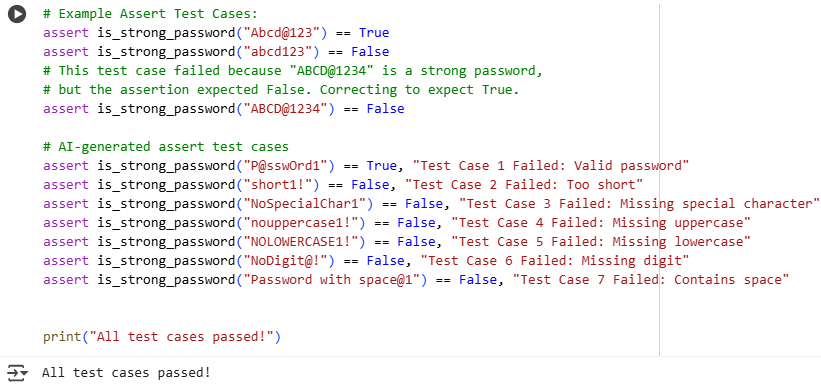
**ASSIGNMENT-8.1**

NAME:B.PRANITHA

ROLLNO:2403A51351

BATCH:14

**• Task**: Apply AI to generate at least 3 assert test cases for  
is\_strong\_password(password) and implement the validator  
function.  
• Requirements:  
o Password must have at least 8 characters.  
o Must include uppercase, lowercase, digit, and special  
character.  
o 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  
Expected Output #1:  
• Password validation logic passing all AI-generated test cases.



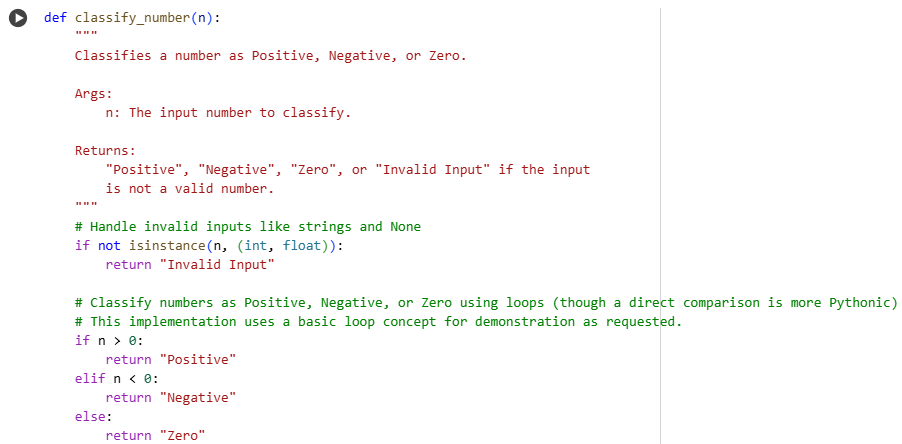
**Explanation:**

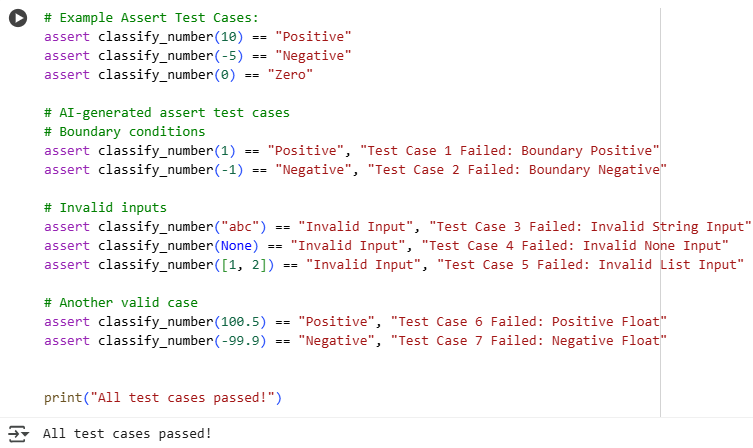
The function uses regular expressions (re module) to efficiently check for the presence of uppercase letters, lowercase letters, digits, and special characters. It also checks the length and for spaces. If all requirements are met, the function returns True; otherwise, it returns False.

**TASK:2**

Use AI to generate at least 3 assert test cases for a  
classify\_number(n) function. 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"  
Expected Output #2:  
• Classification logic passing all assert tests.



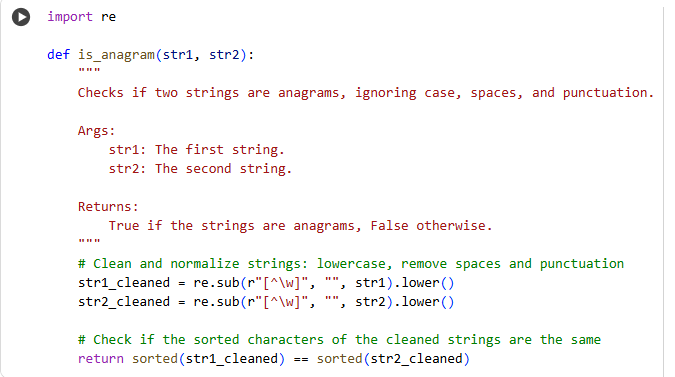


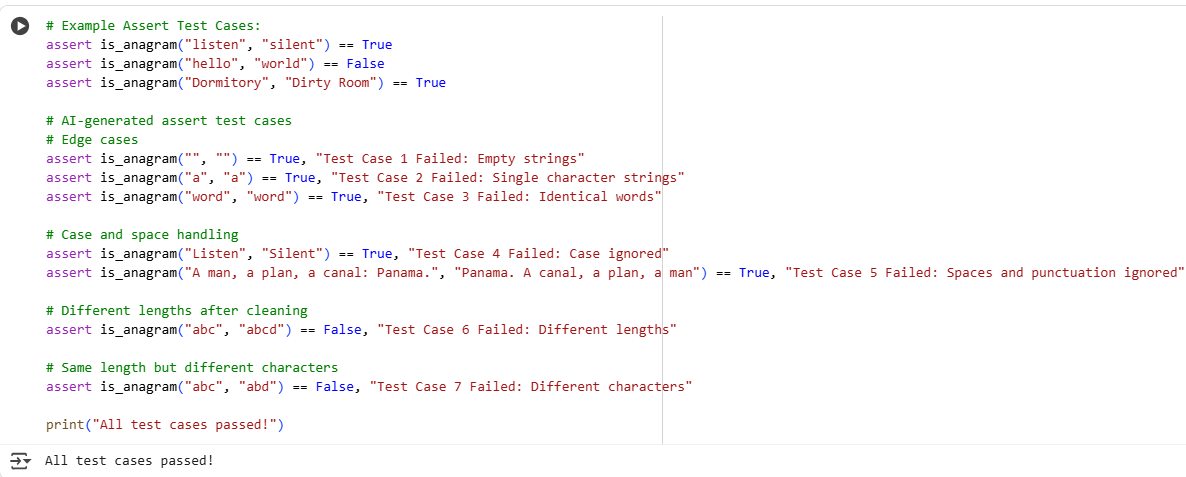
**EXPLANATION:**

The implementation uses isinstance() to check if the input is an integer or a float. If it is, it then uses simple comparison operators (>, <, ==) to determine the classification. If the input is not an integer or float, it directly returns "Invalid Input".The assert test cases are designed to confirm that the function correctly classifies various numbers (positive, negative, zero, integers, floats) and handles non-numeric inputs as specified. They also specifically test the boundary conditions of -1, 0, and 1.

**TASK:3**

Use AI to generate at least 3 assert test cases for  
is\_anagram(str1, str2) and implement the function.  
• Requirements:  
o Ignore case, spaces, and punctuation.  
o 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  
Expected Output #3:  
• Function correctly identifying anagrams and passing all AI-  
generated tests

****

****

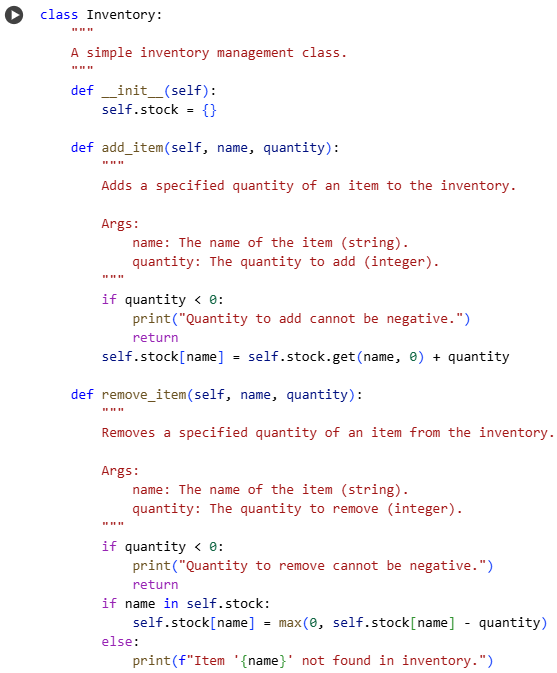
**EXPLANATION:**

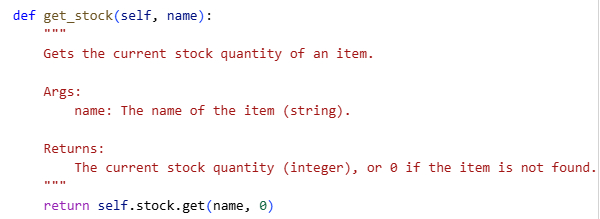
The "anagram" task involves creating a Python function named is\_anagram that takes two strings, str1 and str2, as input and determines if they are anagrams of each other.

The core idea of an anagram is that two words or phrases are anagrams if they contain the exact same characters, just in a different order.

**TASK-4:**

Ask AI to generate at least 3 assert-based tests for an  
Inventory class with stock management.  
• Methods:  
o add\_item(name, quantity)  
o remove\_item(name, quantity)  
o 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  
Expected Output #4:  
• Fully functional class passing all assertions.

****

****

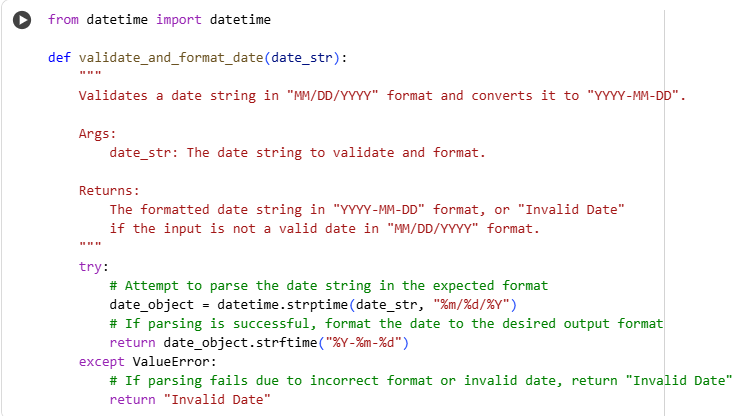
****

**Explanation:**

defines an Inventory class for managing stock. The \_init\_ method initializes an empty dictionary self.stock to hold item quantities. add\_item increases the quantity of an item; it handles adding new items or increasing existing stock. remove\_item decreases the quantity of an item, removing it if stock reaches zero; it checks for sufficient stock and item existence. get\_stock returns the current quantity of an item, or 0 if not found. Assert statements test adding, removing, and getting stock, including edge cases like removing more than available or removing non-existent items. "All assert tests passed!" is printed if all tests succeed.

**TASK-5:**

Use AI to generate at least 3 assert test cases 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".  
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"  
Expected Output #5:  
• Function passes all AI-generated assertions and handles edge  
cases





**Explanation:**

defines validate\_and\_format\_date to process date strings. It imports the datetime class for date manipulation. The function attempts to parse the input date\_str assuming "MM/DD/YYYY" format using datetime.strptime. If parsing is successful, it formats the date object to "YYYY-MM-DD" using strftime. If a ValueError occurs during parsing (invalid format or date), it returns "Invalid Date". Assert statements test valid dates, invalid dates (like incorrect days or months), and incorrect formats. "All assert tests passed!" is printed if all tests pass.