Assignment-8

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:

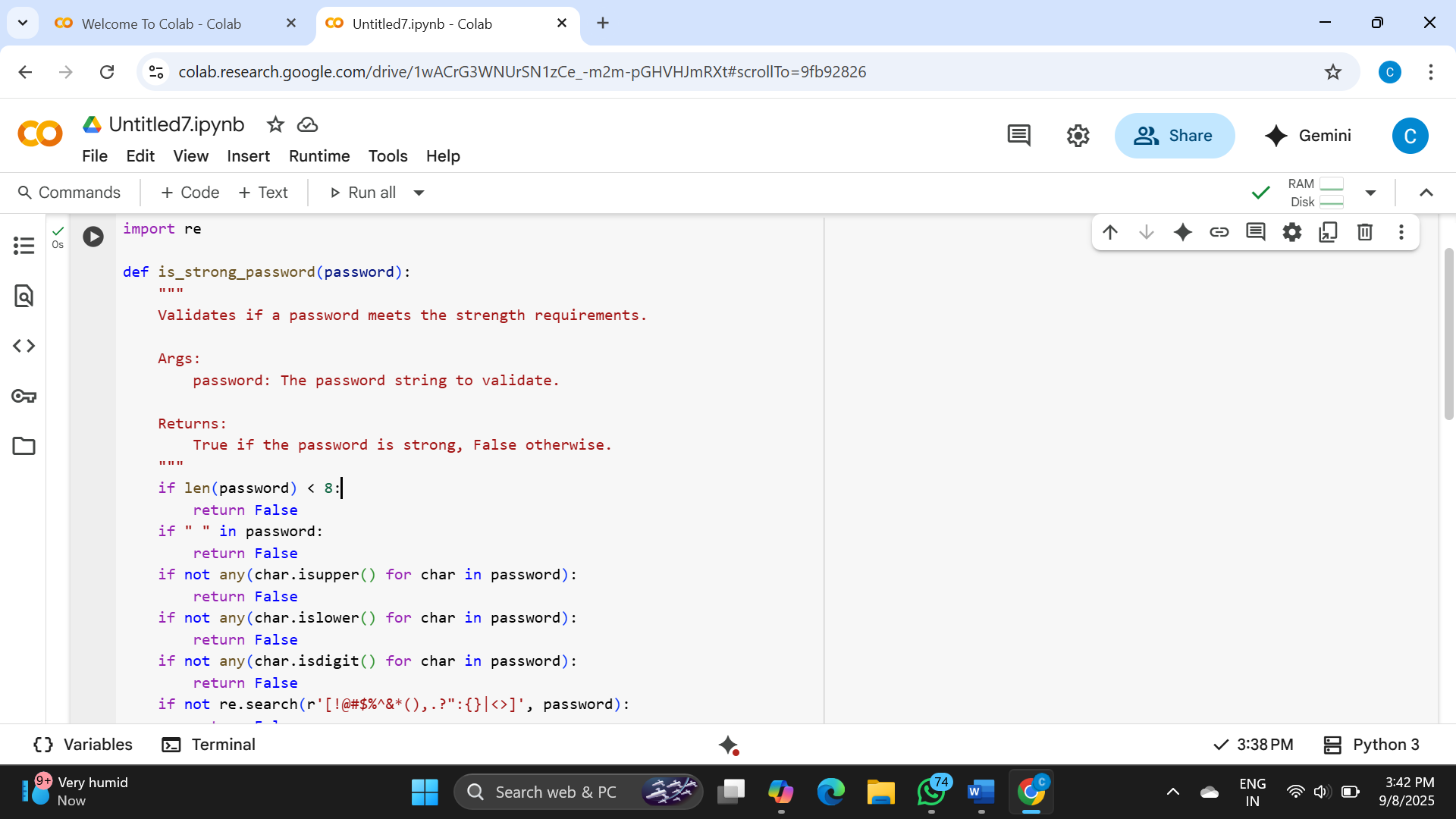
assert is\_strong\_password("Abcd@123") == True

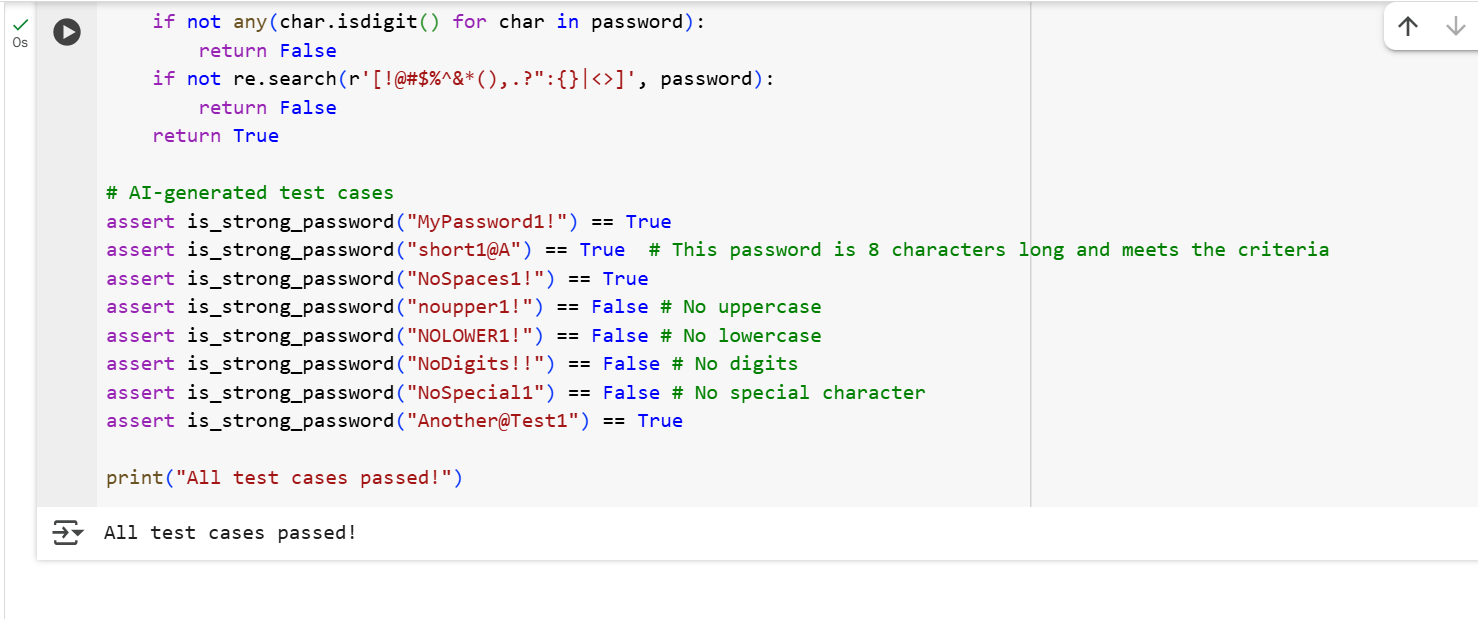
assert is\_strong\_password("abcd123") == False

assert is\_strong\_password("ABCD@1234") == True

Expected Output #1:

* Password validation logic passing all AI-generated test cases.





Code Explanation:

1. **import re**: This line imports the regular expression module in Python, which is used later to check for special characters.
2. **def is\_strong\_password(password):**: This line 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 immediately returns False because the password is too short.
4. **if " " in password:**: This checks if there is a space character within the password string. If a space is found, the function returns False.
5. **if not any(char.isupper() for char in password):**: This line checks if the password contains at least one uppercase letter. any() is used with a generator expression that iterates through each character (char) in the password and checks if it's uppercase using char.isupper(). If no uppercase letters are found, the condition is True, and the function returns False.
6. **if not any(char.islower() for char in password):**: Similar to the previous line, this checks if the password contains at least one lowercase letter. If not, it returns False.
7. **if not any(char.isdigit() for char in password):**: This checks if the password contains at least one digit. If not, it returns False.
8. **if not re.search(r'[!@#$%^&\*(),.?":{}|<>]', password):**: This line uses a regular expression to check for the presence of at least one special character from the defined set. re.search() attempts to find a match for the pattern in the password string. If no match is found, the condition is True, and the function returns False.
9. **return True**: If the password passes all the previous checks (meaning it meets all the strength requirements), the function returns True.
10. **# AI-generated test cases**: This is a comment indicating that the following lines are test cases.
11. **assert is\_strong\_password(...) == ...**: These lines are assert statements that test the is\_strong\_password function with different inputs and expected outputs. If the function's output does not match the expected output for any of these tests, an AssertionError will be raised, indicating a problem with the function.
12. **print("All test cases passed!")**: This line is executed only if all the assert statements pass without raising an error, confirming that the function works correctly for these test cases.

Task-2

(Number Classification with Loops – Apply AI for Edge Case Handling)

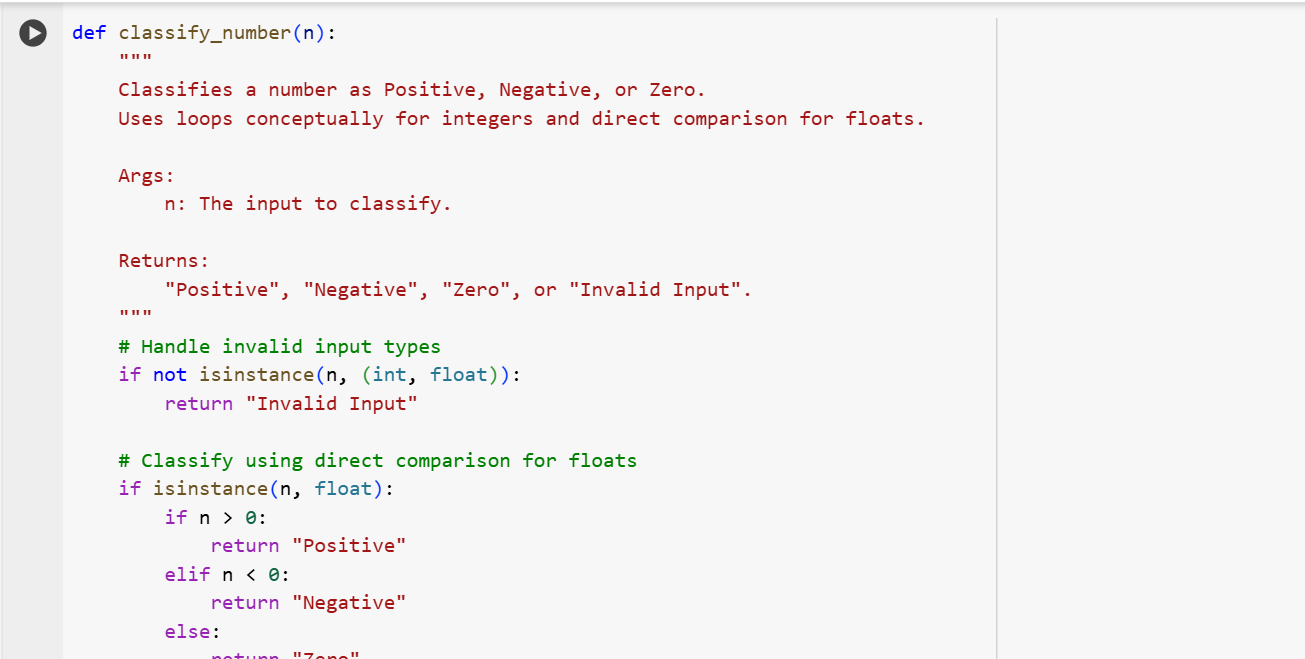
* Task: Use AI to generate at least 3 assert test cases for a classify\_number(n) function. Implement using loops.
* Requirements:
  + Classify numbers as Positive, Negative, or Zero.
  + Handle invalid inputs like strings and None.
  + Include boundary conditions (-1, 0, 1).

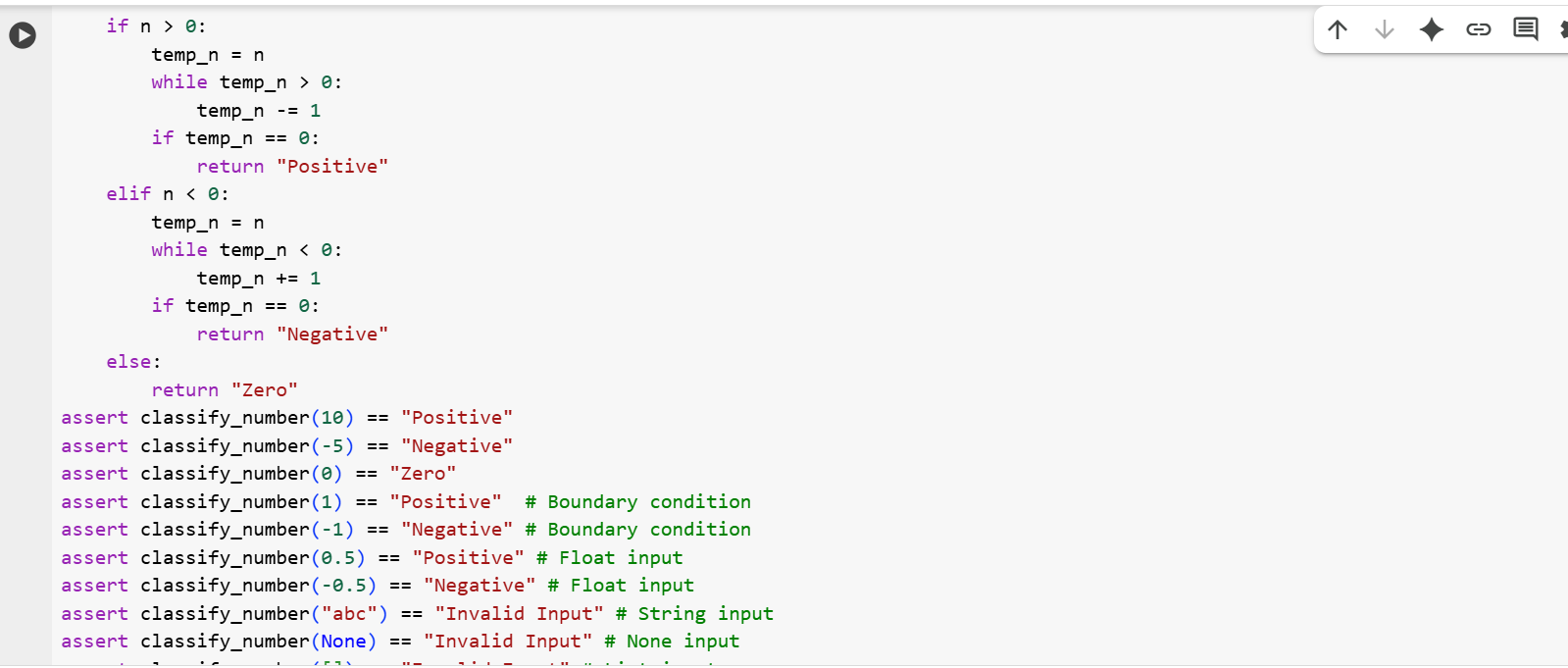
Example Assert Test Cases:

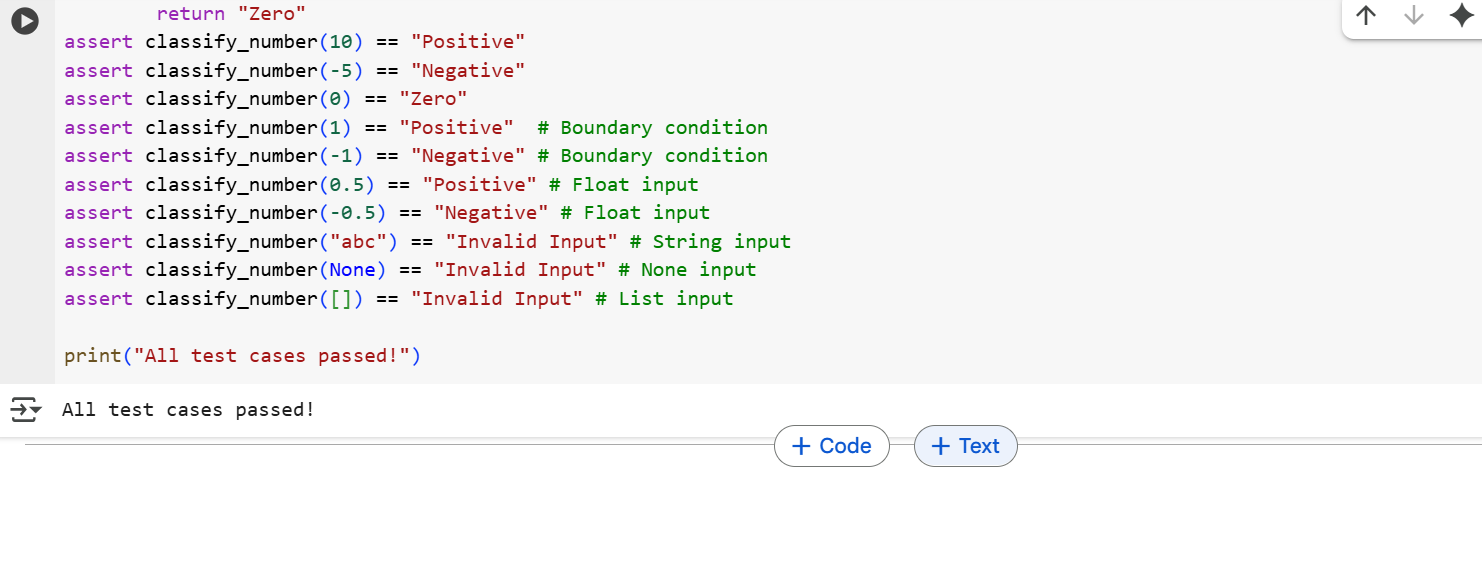
assert classify\_number(10) == "Positive"

assert classify\_number(-5) == "Negative"

assert classify\_number(0) == "Zero"







Code Explanation:

1. **def classify\_number(n):**: This defines the function classify\_number that takes one argument, n, which is the input we want to classify.
2. **if not isinstance(n, (int, float)):**: This is the first check the function performs. It uses isinstance() to see if the input n is *not* an integer (int) or a floating-point number (float). If the input is of any other type (like a string, a list, None, etc.), it immediately returns the string "Invalid Input". This handles the invalid input requirement.
3. **if isinstance(n, float):**: This block specifically handles floating-point numbers. Since the loop-based approach is not suitable for floats, direct comparison is used here.
   * **if n > 0:**: Checks if the float is greater than zero. If true, it returns "Positive".
   * **elif n < 0:**: If the float is not greater than zero, this checks if it's less than zero. If true, it returns "Negative".
   * **else:**: If the float is neither greater than nor less than zero, it must be zero. In this case, it returns "Zero".
4. **if n > 0:**: This block handles positive integers using a conceptual loop.
   * **temp\_n = n**: A temporary variable temp\_n is created and initialized with the value of n.
   * **while temp\_n > 0:**: This is the conceptual loop. It continues as long as temp\_n is greater than 0.
   * **temp\_n -= 1**: In each iteration of the loop, 1 is subtracted from temp\_n. The idea is that if an integer is positive, repeatedly subtracting 1 will eventually lead to 0.
   * **if temp\_n == 0:**: After the loop finishes, this checks if temp\_n has become exactly 0. If it has, it means the original number was a positive integer, and the function returns "Positive".
5. **elif n < 0:**: This block handles negative integers using a conceptual loop.
   * **temp\_n = n**: A temporary variable temp\_n is created and initialized with the value of n.
   * **while temp\_n < 0:**: This loop continues as long as temp\_n is less than 0.
   * **temp\_n += 1**: In each iteration, 1 is added to temp\_n. The idea is that if an integer is negative, repeatedly adding 1 will eventually lead to 0.
   * **if temp\_n == 0:**: After the loop, this checks if temp\_n has become exactly 0. If it has, it means the original number was a negative integer, and the function returns "Negative".
6. **else:**: If the input n is not an invalid type, not a float, not greater than 0, and not less than 0, it must be the integer 0. In this case, the function returns "Zero".
7. **# AI-generated test cases**: This comment indicates the start of the test cases.
8. **assert classify\_number(...) == ...**: These lines are assert statements that verify the function's behavior with various inputs, including positive, negative, and zero integers and floats, as well as invalid inputs. They check if the function returns the expected classification for each input.
9. **print("All test cases passed!")**: This line is executed only if all the assert statements are successful, confirming that the function works correctly for the provided test cases.

Task-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:

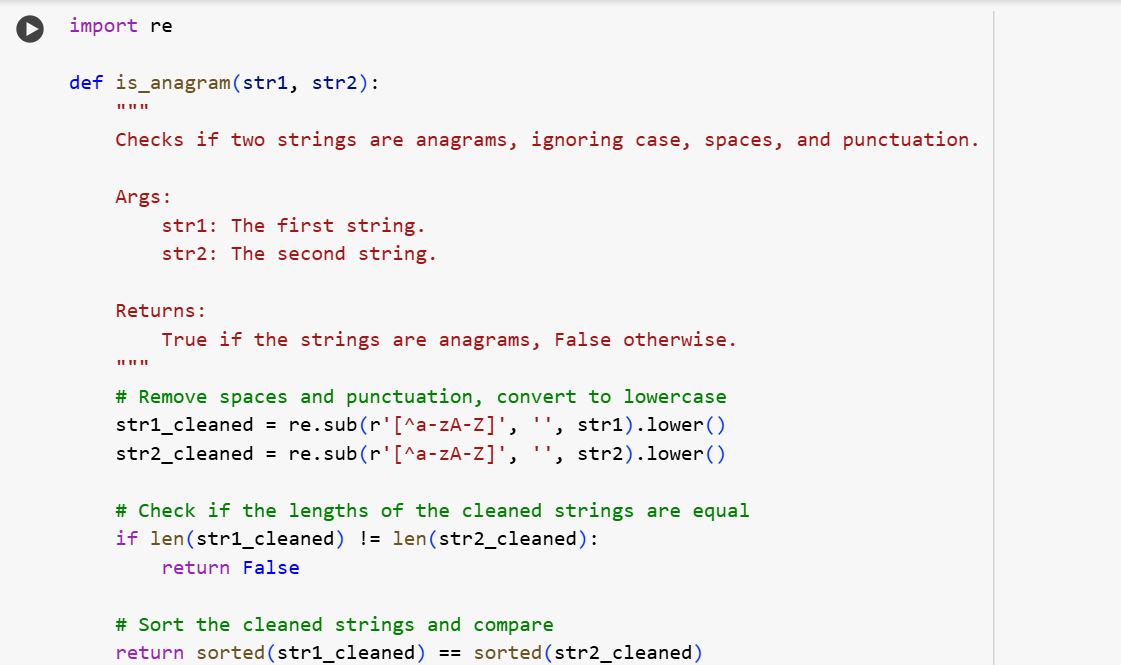
assert is\_anagram("listen", "silent") == True

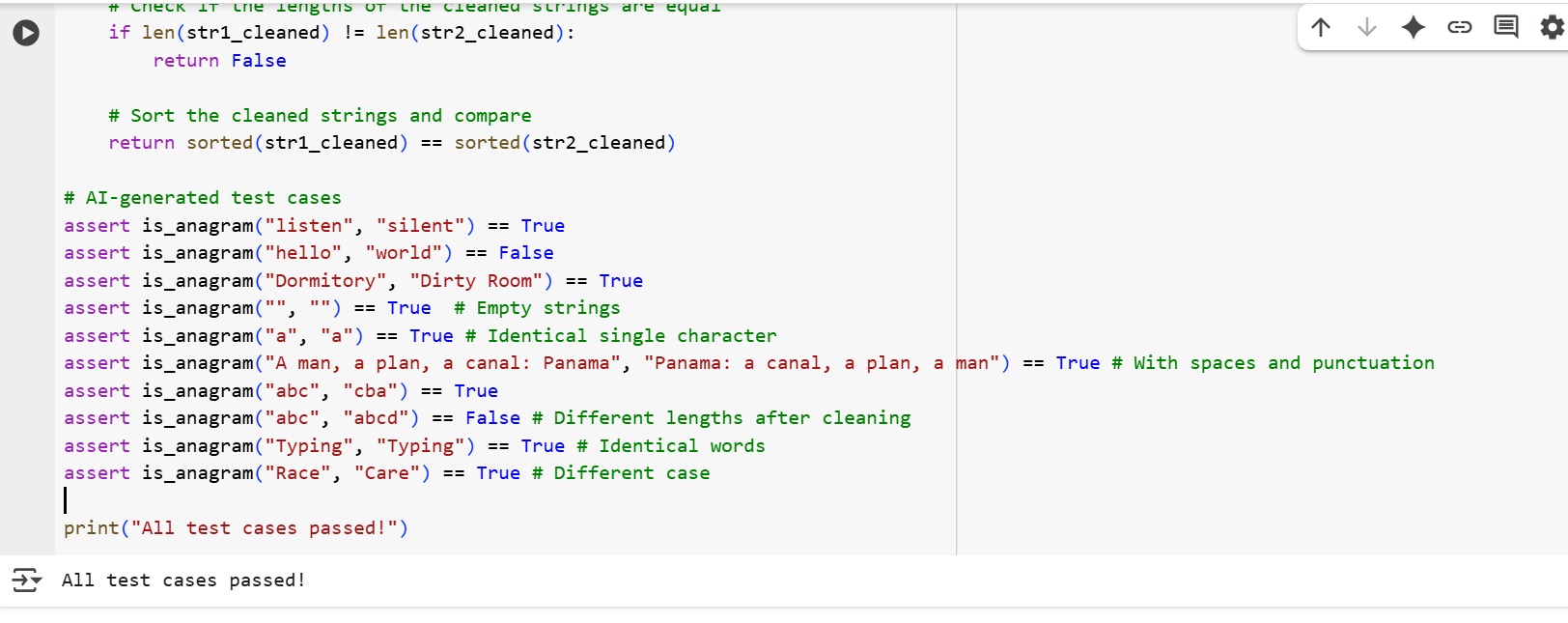
assert is\_anagram("hello", "world") == False

assert is\_anagram("Dormitory", "Dirty Room") == True

Expected Output #3:

* Function correctly identifying anagrams and passing all AI-generated tests.





Code Explanation:

1. **import re**: This line imports the regular expression module, which is used for cleaning the input strings.
2. **def is\_anagram(str1, str2):**: This defines the function is\_anagram that takes two arguments, str1 and str2, which are the strings to be checked.
3. **str1\_cleaned = re.sub(r'[^a-zA-Z]', '', str1).lower()**: This is a key step for cleaning the first string (str1).
   * re.sub(r'[^a-zA-Z]', '', str1): This uses the re.sub() function to substitute (replace) characters in str1. The pattern [^a-zA-Z] matches any character that is *not* an uppercase letter (A-Z) or a lowercase letter (a-z). These matched characters are replaced with an empty string (''), effectively removing them. This gets rid of spaces, punctuation, and any other non-alphabetic characters.
   * .lower(): After removing non-alphabetic characters, .lower() converts the entire resulting string to lowercase. This ensures that the comparison is case-insensitive.
4. **str2\_cleaned = re.sub(r'[^a-zA-Z]', '', str2).lower()**: This line does the exact same cleaning process for the second string (str2).
5. **if len(str1\_cleaned) != len(str2\_cleaned):**: This checks if the lengths of the two cleaned strings are different. If they are, the strings cannot be anagrams (because anagrams must use the same letters the same number of times), so the function immediately returns False.
6. **return sorted(str1\_cleaned) == sorted(str2\_cleaned)**: If the lengths are the same, the function proceeds to this line.
   * sorted(str1\_cleaned): This sorts the characters of the cleaned str1 alphabetically and returns them as a list of characters.
   * sorted(str2\_cleaned): This does the same for the cleaned str2.
   * ... == ...: This compares the two sorted lists of characters. If the sorted lists are identical, it means the two original strings (ignoring case, spaces, and punctuation) contained the exact same characters with the same frequencies, and thus they are anagrams. The function returns True. Otherwise, it returns False.
7. **# AI-generated test cases**: This comment indicates the start of the test cases.
8. **assert is\_anagram(...) == ...**: These lines are assert statements that test the is\_anagram function with various pairs of strings, including known anagrams, non-anagrams, empty strings, identical words, and strings with different cases, spaces, and punctuation. They check if the function returns the expected True or False for each pair.
9. **print("All test cases passed!")**: This line is executed only if all the assert statements pass without raising an error, confirming that the function works correctly for these test cases.

Task-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)

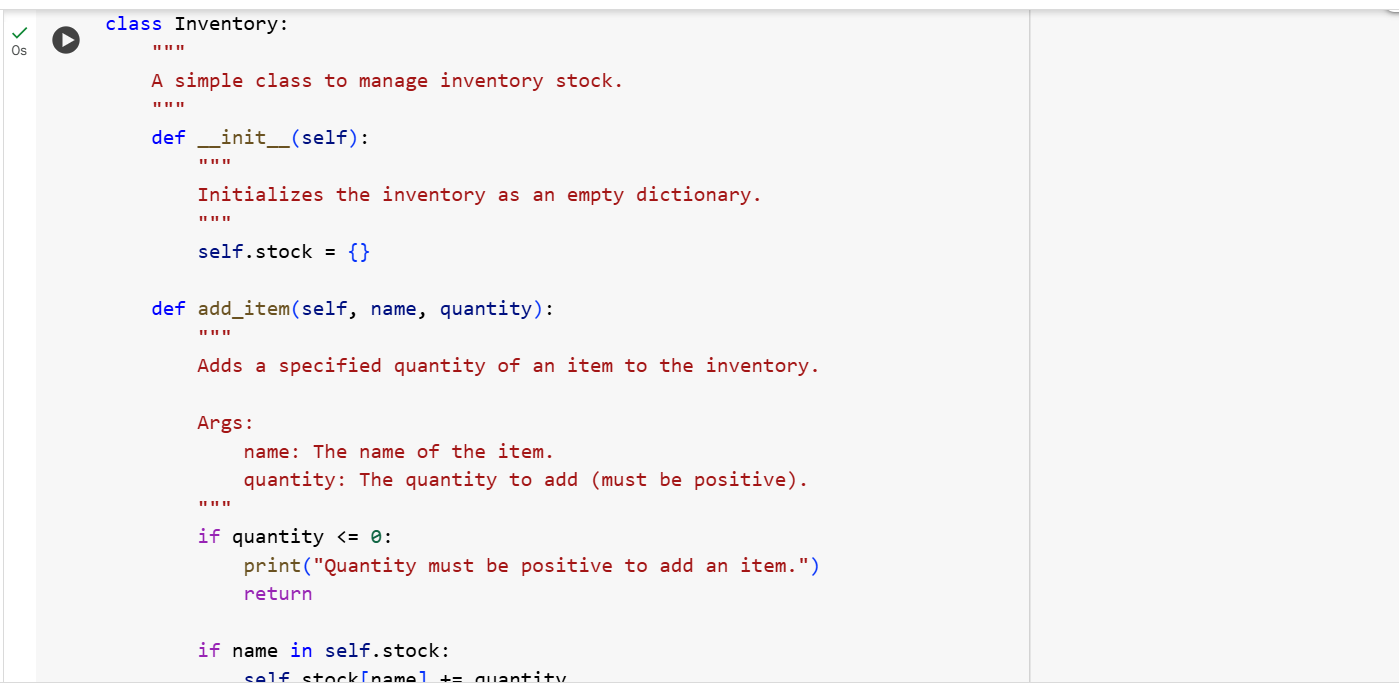
assert inv.get\_stock("Pen") == 5

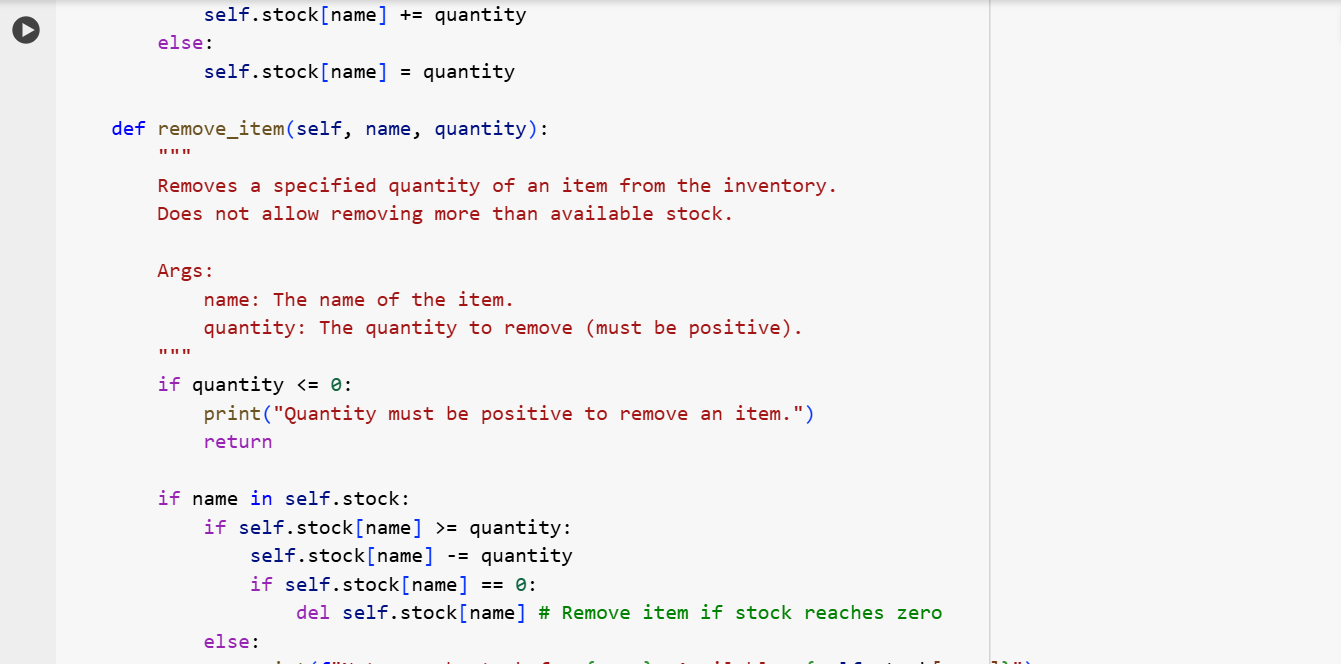
inv.add\_item("Book", 3)

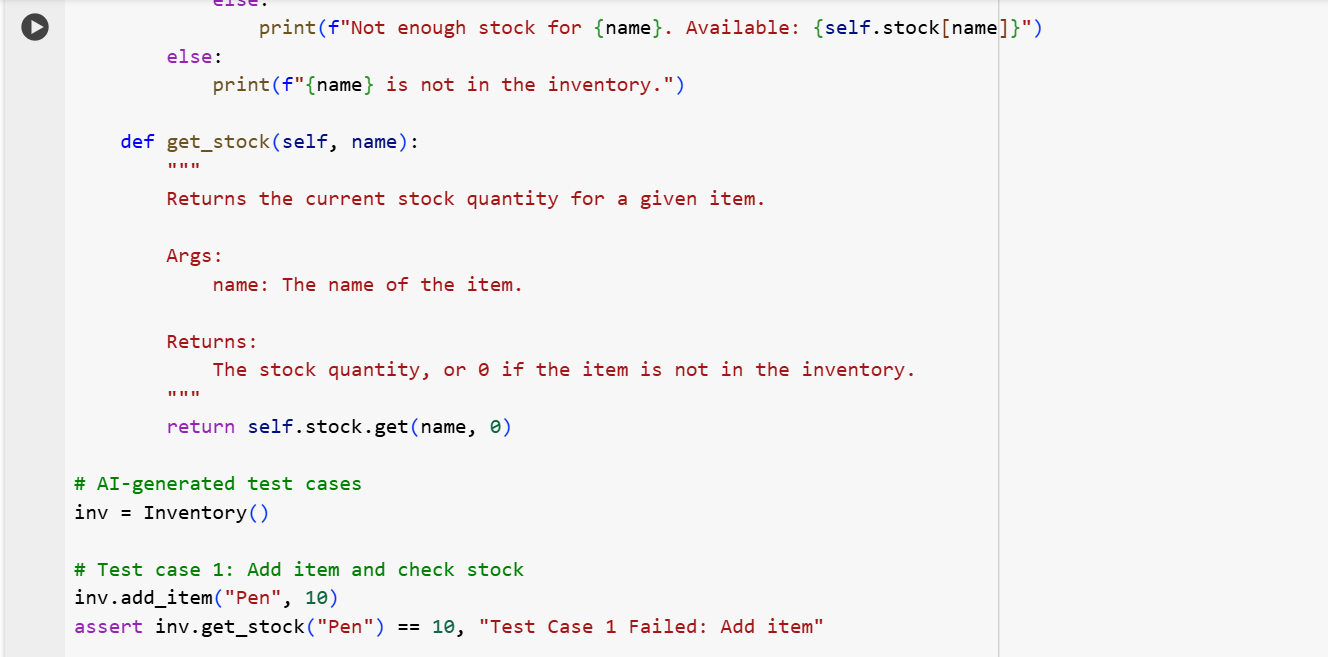
assert inv.get\_stock("Book") == 3

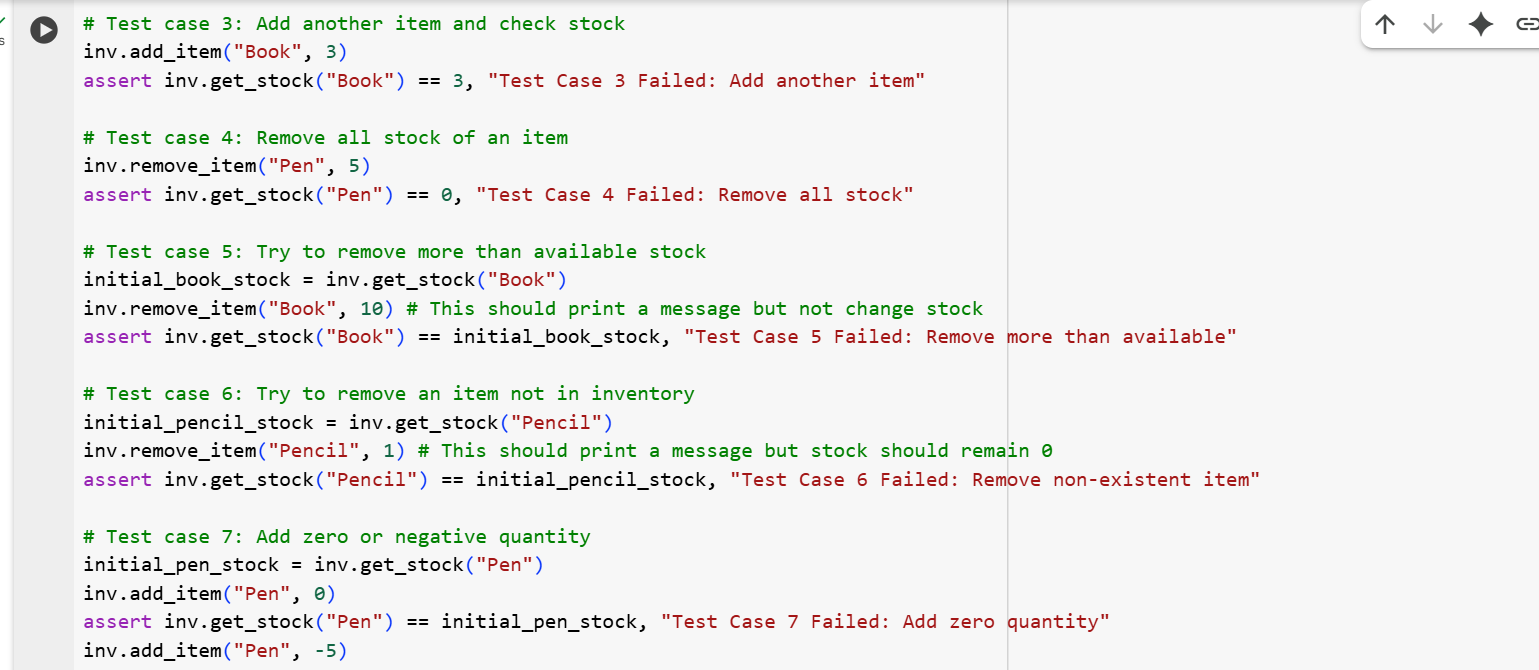
Expected Output #4:

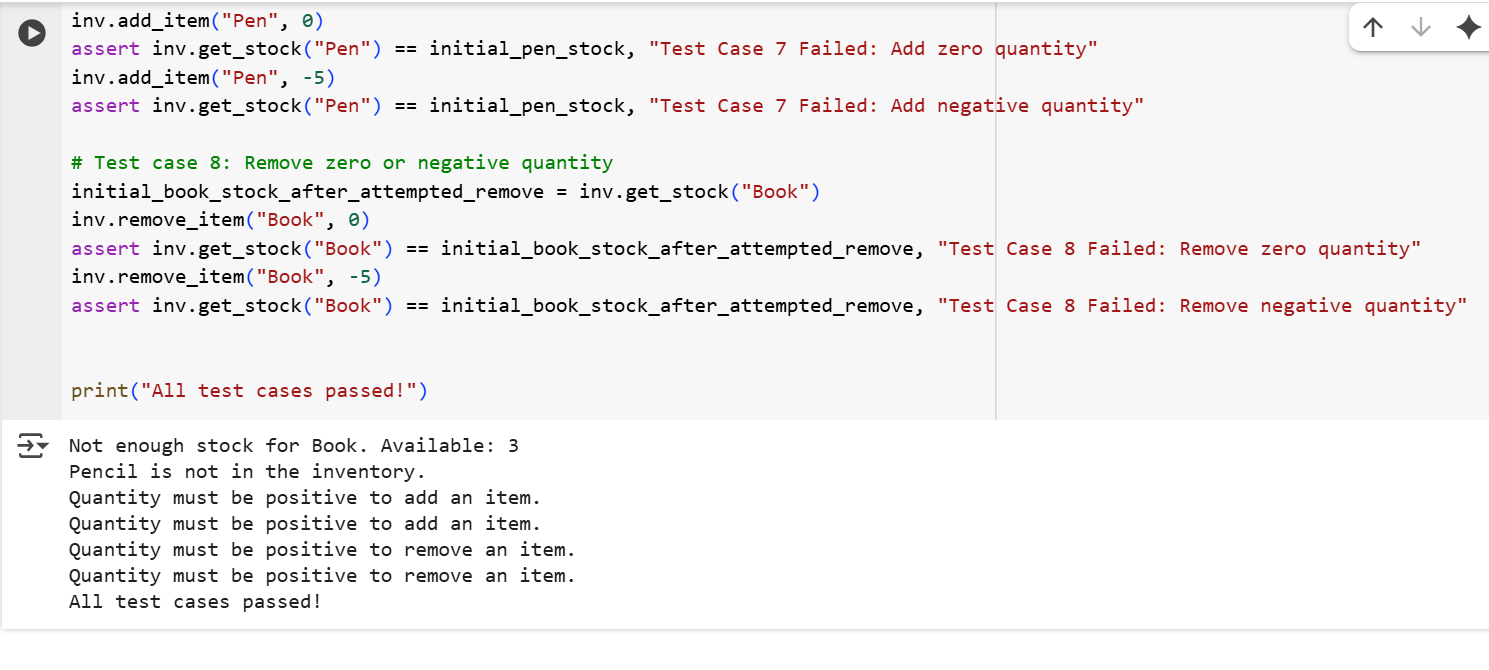
* Fully functional class passing all assertions.











Code Explanation:

This code creates a blueprint for an "Inventory" system, like managing items in a store or warehouse.

* **class Inventory:**: This is like saying "We are creating a new type of thing called an Inventory."
* **def \_\_init\_\_(self):**: This is what happens when you create a *new* Inventory. It sets up an empty list (actually a dictionary called stock) to keep track of all the items and how many of each you have.
* **def add\_item(self, name, quantity):**: This is a command you can give to your Inventory. You tell it the name of the item (like "Pen") and the quantity (how many you want to add, like 10). It adds that many to your stock. It also checks if you're trying to add a sensible number (more than zero).
* **def remove\_item(self, name, quantity):**: This is another command. You tell it the name and quantity to take away. It subtracts that many from your stock. It's smart enough not to let you take away more than you have! If you remove the last one, it takes the item off the list. It also checks if you're trying to remove a sensible number (more than zero).
* **def get\_stock(self, name):**: This command asks the Inventory how many of a specific item (name) you currently have. It tells you the number. If the item isn't in your stock at all, it tells you 0.

**The lines after the class are tests:**

* **inv = Inventory()**: This creates an actual Inventory using the blueprint.
* **inv.add\_item("Pen", 10)**: This tells the Inventory to add 10 Pens.
* **assert inv.get\_stock("Pen") == 10**: This is a check. It says, "Make sure that if I ask the Inventory how many Pens it has, the answer is 10." If it's not 10, something is wrong!

The other assert lines are similar checks to make sure adding, removing, and getting stock work correctly in different situations, including trying to remove too many items or items that aren't there.

Task-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:

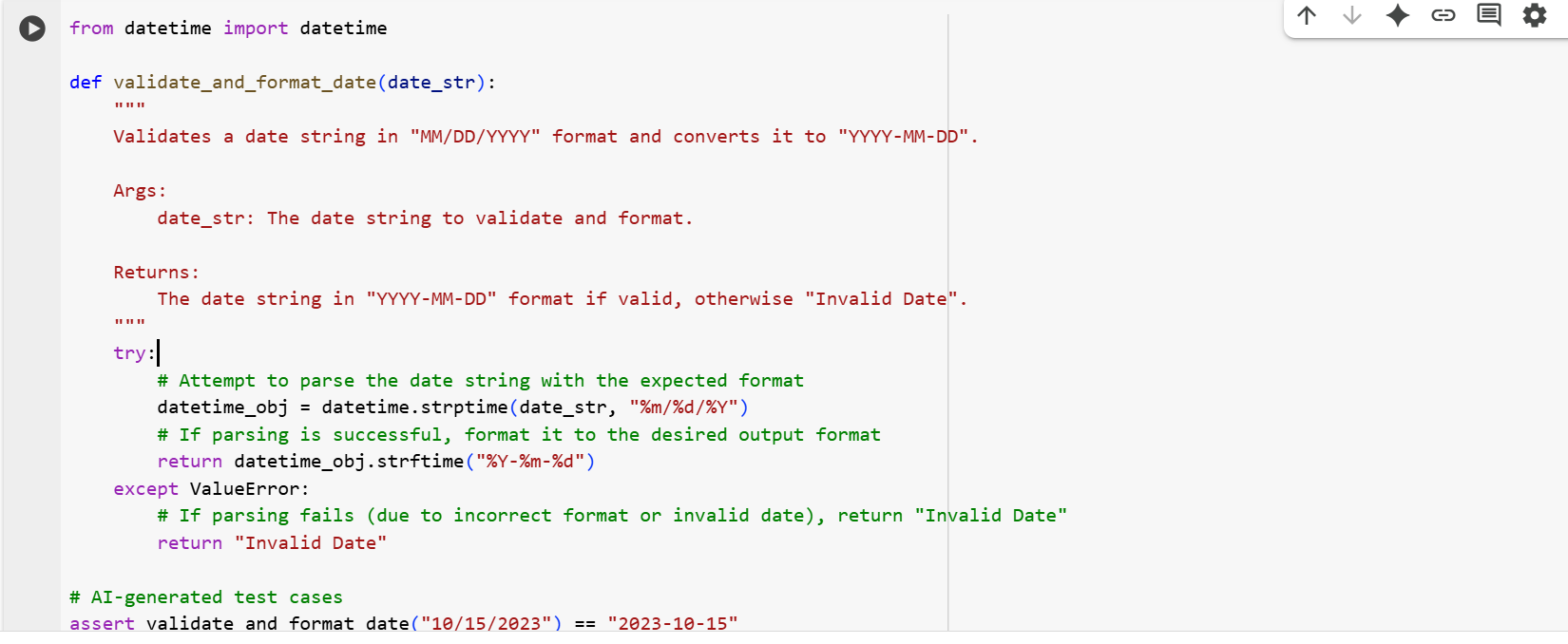
assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"

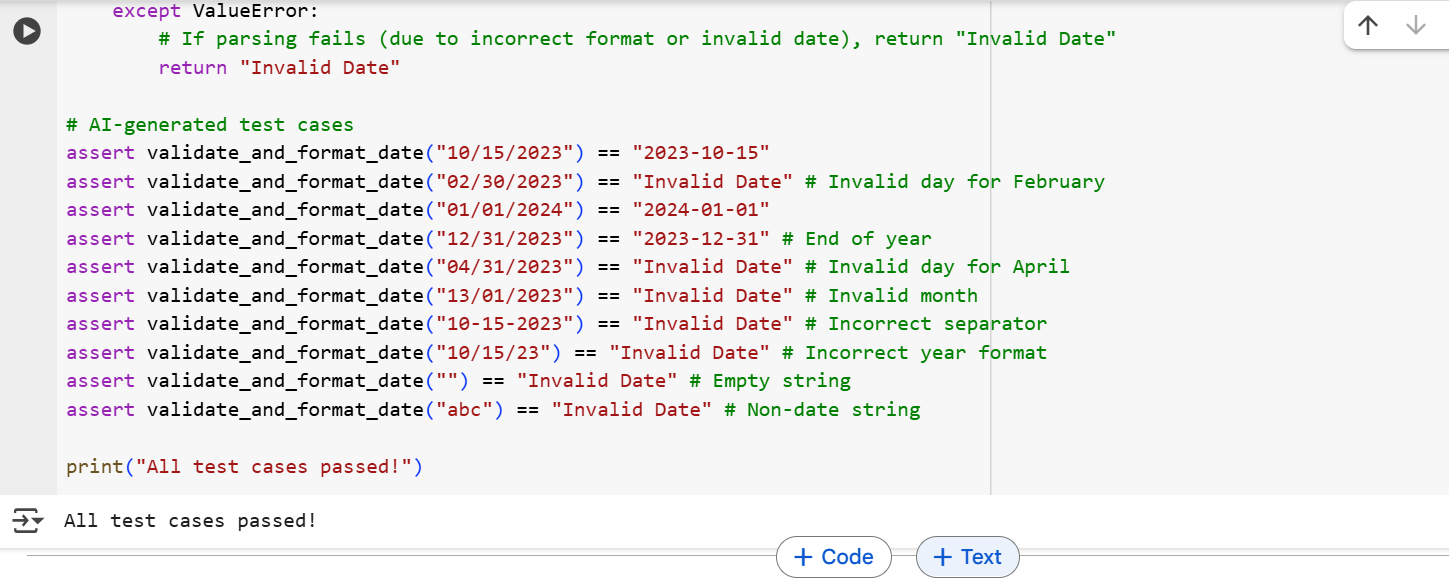
assert validate\_and\_format\_date("02/30/2023") == "Invalid Date"

assert validate\_and\_format\_date("01/01/2024") == "2024-01-01"

Expected Output #5:

* Function passes all AI-generated assertions and handles edge cases.





Code Explanation:

1. **from datetime import datetime**: This line just brings in some tools from Python that help work with dates and times.
2. **def validate\_and\_format\_date(date\_str):**: This is like creating a little machine or tool called validate\_and\_format\_date. You give it a piece of text that you think is a date (date\_str).
3. **try:**: This is like saying, "Okay, I'm going to try to do something that *might* go wrong."
4. **datetime\_obj = datetime.strptime(date\_str, "%m/%d/%Y")**: This is the main part inside the "try." It tells the tool to look at the date\_str you gave it and try to understand it as a date that looks *exactly* like "Month/Day/Year" (that's what "%m/%d/%Y" means). If it can understand it and it's a real date (like not February 30th), it turns it into a special date object (datetime\_obj).
5. **return datetime\_obj.strftime("%Y-%m-%d")**: If the step above worked without any problems, this line takes that special date object and turns it back into a text string, but this time in the "Year-Month-Day" format (that's what "%Y-%m-%d" means).
6. **except ValueError:**: This is what happens if the "try" part failed. If the date\_str wasn't in the "MM/DD/YYYY" format, or if it was in the right format but represented an impossible date (like "13/01/2023" or "02/30/2023"), Python will raise a ValueError. This except block catches that error.
7. **return "Invalid Date"**: If the ValueError happens, the function returns the text "Invalid Date" to tell you that the input wasn't a valid date in the expected format.

**The lines after the function are tests:**

* **assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"**: This is a check. It says, "Make sure that if I give the function '10/15/2023', it correctly gives back '2023-10-15'."
* **assert validate\_and\_format\_date("02/30/2023") == "Invalid Date"**: This check makes sure that an invalid date (February 30th) is correctly identified as "Invalid Date".

There are more assert lines to test different valid and invalid date inputs.

So, in simple terms, the code tries to read a date string in a specific format. If it's a valid date, it rewrites it in a different format; otherwise, it tells you it's an "Invalid Date".