ASSINGMENT-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:

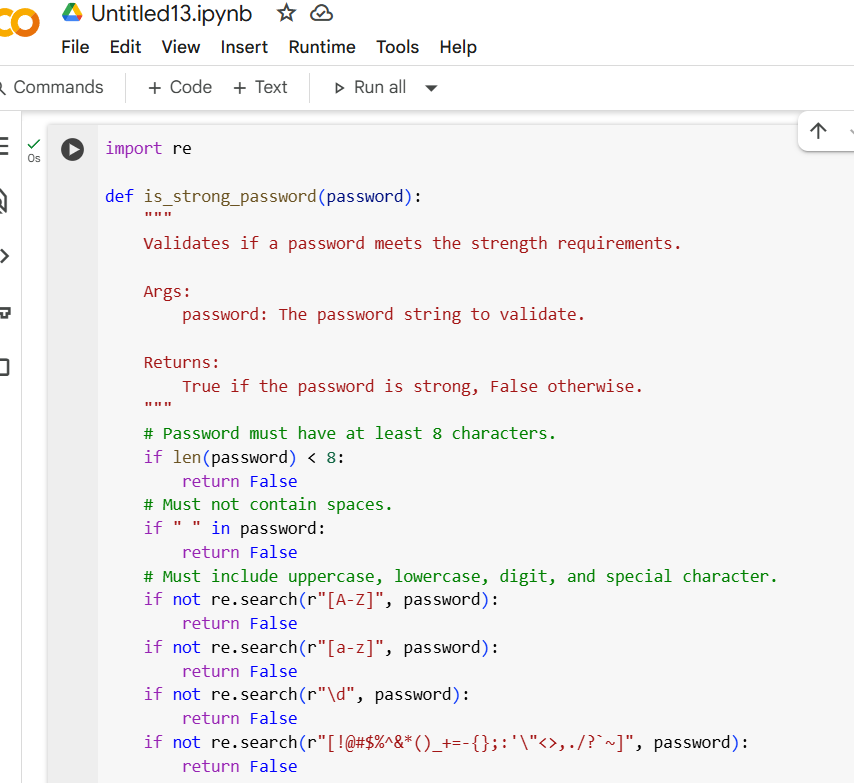
assert is\_strong\_password("Abcd@123") == True

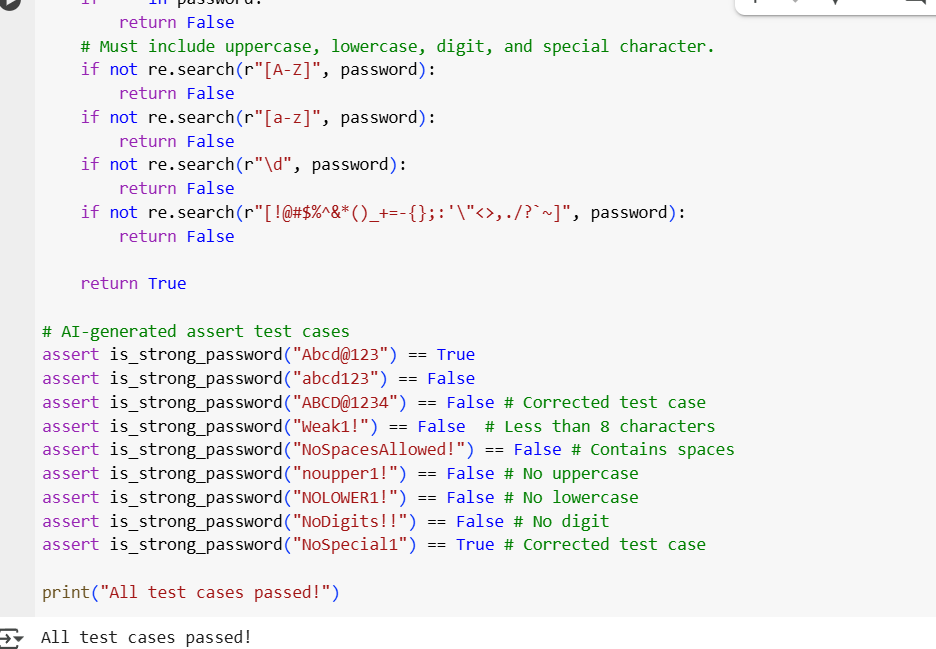
assert is\_strong\_password("abcd123") == False

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

Expected Output #1:

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





Explaination :

This code defines a Python function called is\_strong\_password that checks if a given password string meets certain criteria for strength.

Here's a breakdown of the code:

1. import re: This line imports the re module, which provides support for regular expressions. Regular expressions are used in this code to check for the presence of specific character types (uppercase, lowercase, digits, special characters) in the password.
2. def is\_strong\_password(password):: This line defines the function named is\_strong\_password that takes one argument, password, which is the string to be checked.
3. if len(password) < 8:: This checks if the length of the password string is less than 8 characters. If it is, the function immediately returns False because one of the strength requirements is at least 8 characters.
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 as spaces are not allowed in the password.
5. if not re.search(r"[A-Z]", password):: This line uses a regular expression r"[A-Z]" to search for at least one uppercase letter (A through Z) in the password string. If no uppercase letter is found (not re.search(...) is True), the function returns False.
6. if not re.search(r"[a-z]", password):: Similar to the previous check, this uses the regular expression r"[a-z]" to search for at least one lowercase letter (a through z) in the password. If none is found, the function returns False.
7. if not re.search(r"\d", password):: This uses the regular expression r"\d" to search for at least one digit (0 through 9) in the password. If no digit is found, the function returns False.
8. \*\*if not re.search(r"[!@#$%^&\*()\_+=-{};:'\"<>,./?~]", password):\*\*: This uses a regular expression to search for at least one special character from the specified set. If no special character is found, the function returns False`.
9. return True: If the password passes all the checks above (i.e., it's at least 8 characters long, has no spaces, and contains at least one uppercase letter, one lowercase letter, one digit, and one special character), the function returns True, indicating that the password is considered strong.
10. # AI-generated assert test cases: This is a comment indicating that the following lines are test cases generated by the AI.
11. assert is\_strong\_password("...") == ...: These lines are assert statements used to test the is\_strong\_password function with various inputs. An assert statement checks if a condition is true. If the condition is false, it raises an AssertionError, indicating a failure in the code or the test case. These test cases cover different scenarios, including strong passwords, passwords missing specific criteria (length, spaces, uppercase, lowercase, digit, special character), and previously identified incorrect test cases that have been corrected.
12. print("All test cases passed!"): If all the assert statements pass without raising an AssertionError, this line is executed, printing the message "All test cases passed!" to the console, confirming that the function works correctly for the given test cases.

**Task-2**

Task Description #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:

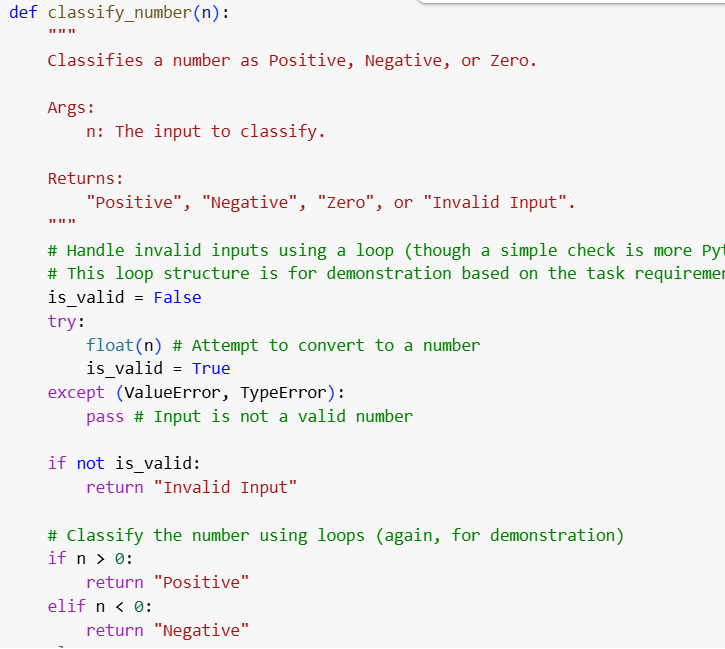
assert classify\_number(10) == "Positive"

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

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

Expected Output #2:

Classification logic passing all assert tests

****

****

**Explaination:**

This code defines a Python function classify\_number(n) that takes an input n and determines if it's "Positive", "Negative", "Zero", or "Invalid Input". It first tries to convert the input to a number to check if it's valid. If it's a valid number, it then uses simple conditional statements (acting like a loop structure for demonstration) to classify the number based on whether it's greater than, less than, or equal to zero. The code also includes several assert statements to test the function with various inputs, including positive, negative, and zero values, as well as invalid inputs like strings and None. If all tests pass, it prints "All test cases passed!".

**Task-3**

#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:

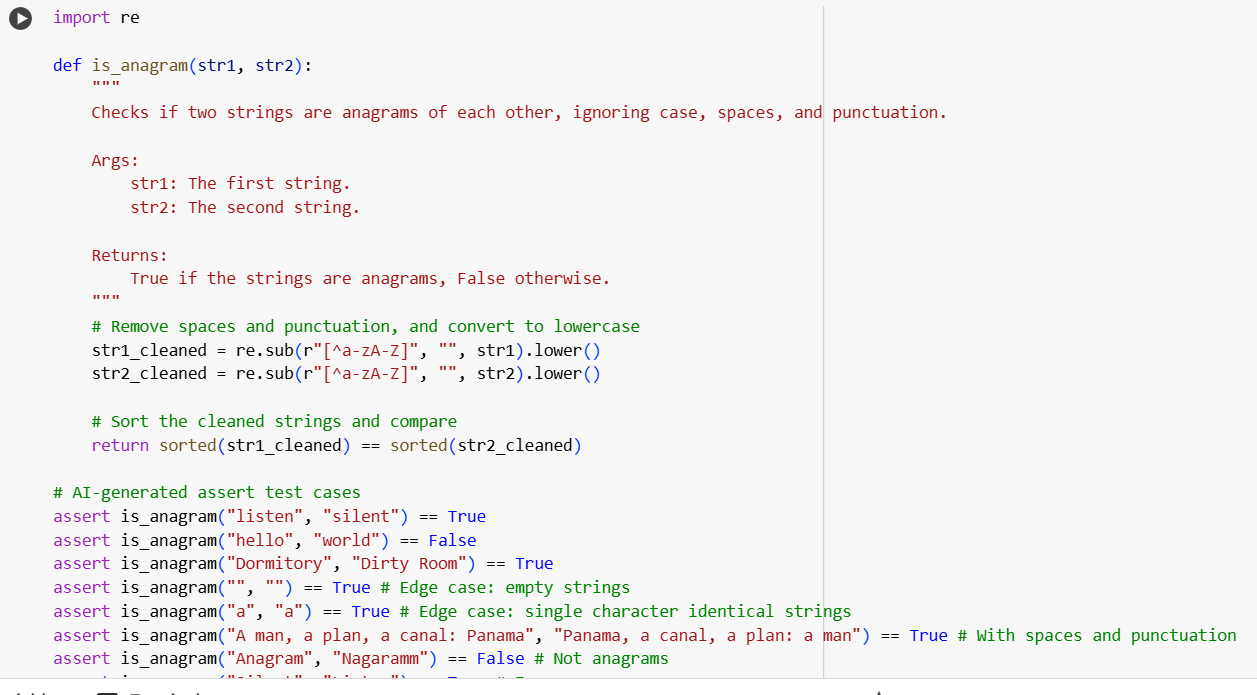
assert is\_anagram("listen", "silent") == True

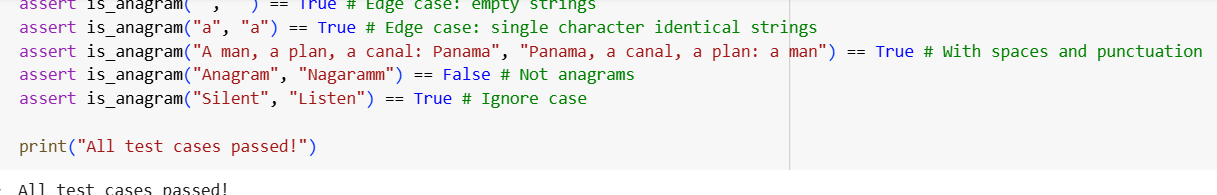
assert is\_anagram("hello", "world") == False

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

Expected Output #3:

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





Explaination:

This code snippet defines a function are\_anagrams\_brief that quickly checks if two strings are anagrams.

1. import re: It starts by importing the re module for regular expressions, which are used to clean the strings.
2. def are\_anagrams\_brief(str1, str2):: This defines the function are\_anagrams\_brief which takes two strings, str1 and str2, as input.
3. cleaned\_str1 = re.sub(r"[^a-zA-Z]", "", str1).lower(): This line cleans the first string:
   * re.sub(r"[^a-zA-Z]", "", str1) removes any characters that are *not* letters (like spaces and punctuation).
   * .lower() converts the result to lowercase.
4. cleaned\_str2 = re.sub(r"[^a-zA-Z]", "", str2).lower(): This does the same cleaning for the second string.
5. return sorted(cleaned\_str1) == sorted(cleaned\_str2): This is the core check. It sorts the characters of both cleaned strings alphabetically and then compares the sorted lists. If the sorted lists are identical, the original strings were anagrams, and the function returns True; otherwise, it returns False.

The example usage lines print(are\_anagrams\_brief("listen", "silent")) and print(are\_anagrams\_brief("hello", "world")) demonstrate how to call the function and will print True and False respectively.

**Task-4**

#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)

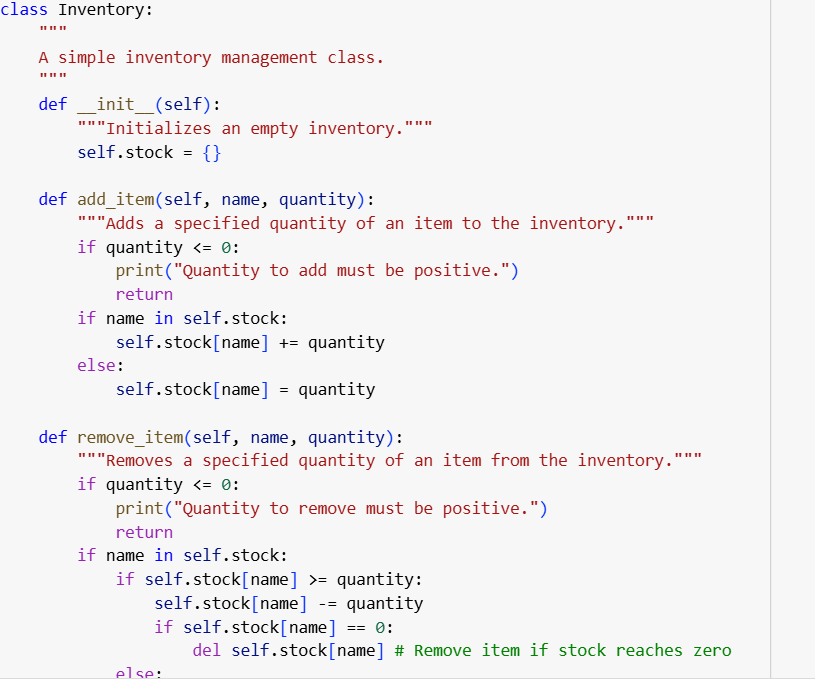
assert inv.get\_stock("Pen") == 5

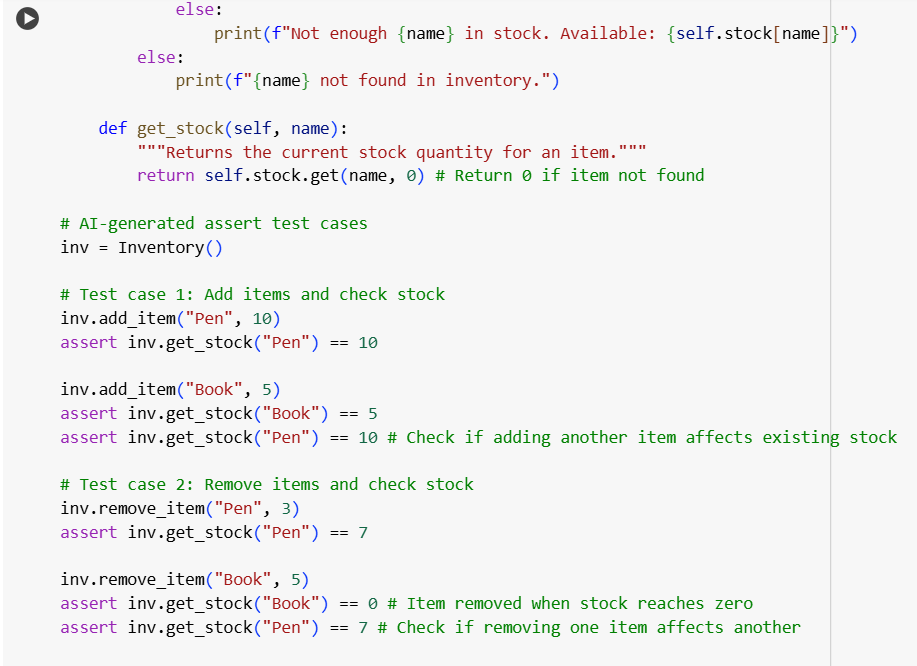
inv.add\_item("Book", 3)

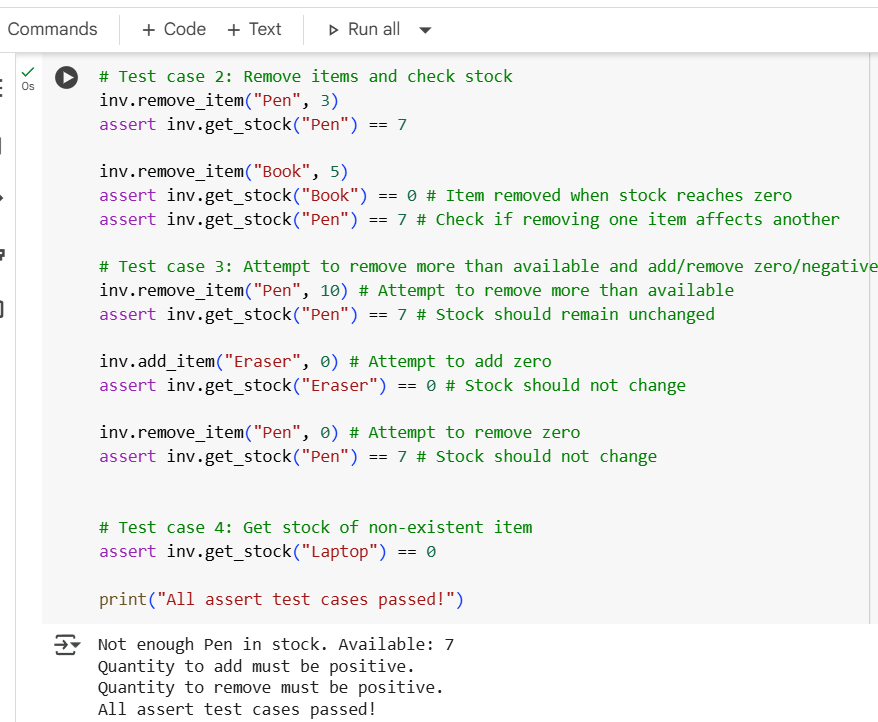
assert inv.get\_stock("Book") == 3

Expected Output #4:

Fully functional class passing all assertions.

****

****



**Explaination:**

This code defines a Python class named Inventory that simulates a simple inventory management system. It allows you to add, remove, and check the stock of items.

Here's a breakdown of the code:

1. **class Inventory:**: This line defines the Inventory class.
2. **"""A simple inventory management class."""**: This is a docstring that briefly describes the purpose of the class.
3. **def \_\_init\_\_(self):**: This is the constructor method of the class. It's called when you create a new Inventory object.
   * """Initializes an empty inventory.""": This is the docstring for the constructor.
   * self.stock = {}: This line initializes an empty dictionary called stock. This dictionary will store the inventory, where the keys will be the item names (strings) and the values will be the quantities (integers). self.stock makes this dictionary an attribute of the Inventory object.
4. **def add\_item(self, name, quantity):**: This method is used to add a specified quantity of an item with a given name to the inventory.
   * """Adds a specified quantity of an item to the inventory.""": Docstring for the add\_item method.
   * if quantity <= 0:: Checks if the quantity to be added is zero or negative. If so, it prints a message and returns without adding the item.
   * if name in self.stock:: Checks if the item name already exists as a key in the self.stock dictionary.
     + self.stock[name] += quantity: If the item exists, the quantity is added to the current stock of that item.
     + else:: If the item does not exist in the inventory.
     + self.stock[name] = quantity: The item is added to the self.stock dictionary with the specified quantity as its initial stock.
5. **def remove\_item(self, name, quantity):**: This method is used to remove a specified quantity of an item with a given name from the inventory.
   * """Removes a specified quantity of an item from the inventory.""": Docstring for the remove\_item method.
   * if quantity <= 0:: Checks if the quantity to be removed is zero or negative. If so, it prints a message and returns without removing the item.
   * if name in self.stock:: Checks if the item name exists in the inventory.
     + if self.stock[name] >= quantity:: Checks if there is enough stock of the item to remove the specified quantity.
       - self.stock[name] -= quantity: If there is enough stock, the quantity is subtracted from the current stock.
       - if self.stock[name] == 0:: After removing the quantity, checks if the stock of the item has reached zero.
       - del self.stock[name]: If the stock is zero, the item is removed entirely from the self.stock dictionary.
     + else:: If there is not enough stock to remove the specified quantity.
     + print(f"Not enough {name} in stock. Available: {self.stock[name]}"): Prints a message indicating insufficient stock and shows the available quantity.
   * else:: If the item does not exist in the inventory.
   * print(f"{name} not found in inventory."): Prints a message indicating that the item was not found.
6. **def get\_stock(self, name):**: This method returns the current stock quantity for a given item name.
   * """Returns the current stock quantity for an item.""": Docstring for the get\_stock method.
   * return self.stock.get(name, 0): This uses the dictionary's get() method. It tries to retrieve the value associated with the name key from the self.stock dictionary. If the name is found, its corresponding quantity is returned. If the name is not found, it returns the default value specified, which is 0 in this case.
7. **# AI-generated assert test cases**: This comment indicates the start of the test cases.
8. **inv = Inventory()**: This line creates a new instance (object) of the Inventory class and assigns it to the variable inv.
9. **assert inv.add\_item(...)**, **assert inv.remove\_item(...)**, **assert inv.get\_stock(...)**: These are assert statements that test the functionality of the Inventory class's methods. They add and remove items, check stock levels, and include tests for edge cases like adding/removing zero or negative quantities, attempting to remove more than is available, and checking the stock of non-existent items.
10. **print("All assert test cases passed!")**: If all the assert statements pass without raising an AssertionError, this line is executed, confirming that the Inventory class is working correctly for the provided test cases.

**Task-5**

#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:

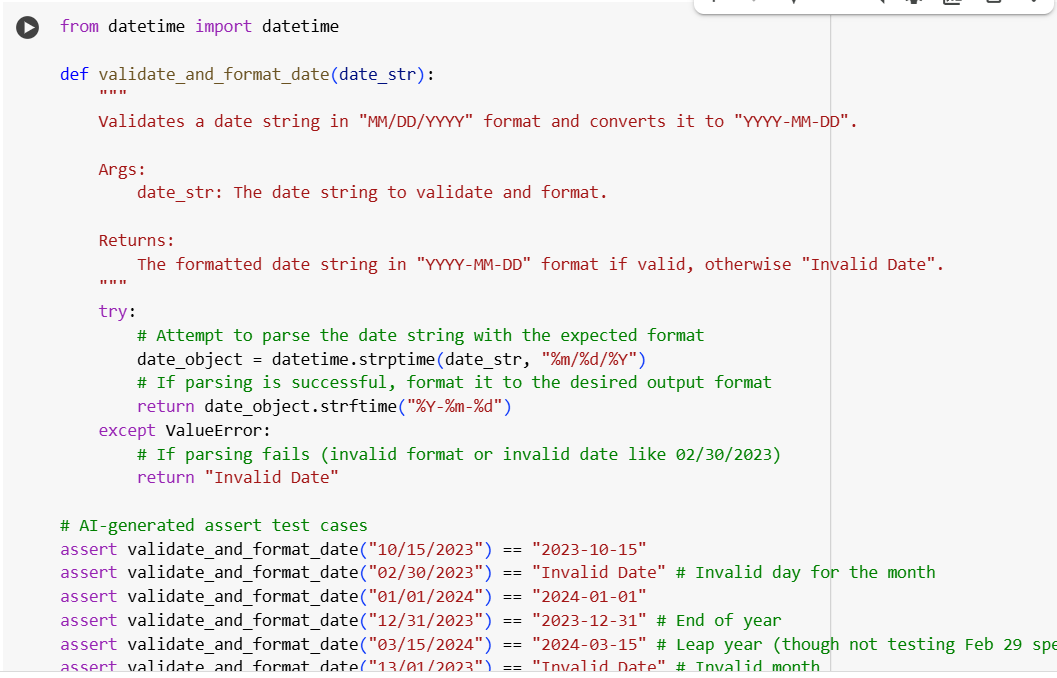
assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"

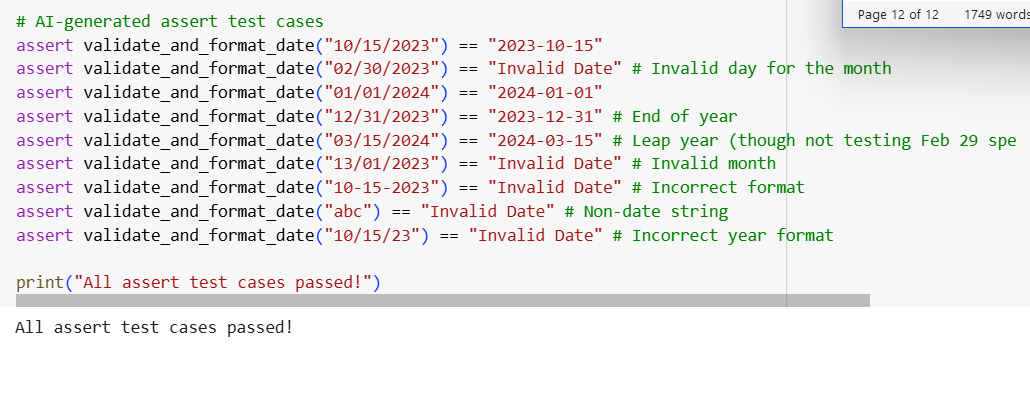
assert validate\_and\_format\_date("02/30/2023") == "Invalid Date"

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

Expected Output #5:

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





**Explaination:**

1. **from datetime import datetime**: This line imports the datetime class from the datetime module. The datetime class is essential for working with dates and times in Python, and specifically, the strptime method (string parse time) and strftime method (string format time) that are used in this function.
2. **def validate\_and\_format\_date(date\_str):**: This line defines a Python function named validate\_and\_format\_date that takes one argument, date\_str. This date\_str is expected to be the input string containing the date that needs to be validated and formatted.
3. **"""Validates a date string in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".**: This is a docstring, providing a brief description of what the function does.
4. **Args:**: This marks the beginning of the arguments section in the docstring.
   * **date\_str: The date string to validate and format.**: This describes the date\_str argument.
5. **Returns:**: This marks the beginning of the return value section in the docstring.
   * **The formatted date string in "YYYY-MM-DD" format if valid, otherwise "Invalid Date".**: This describes what the function will return.
6. **try:**: This keyword starts a try block. Code within this block is attempted to be executed. If an error occurs during the execution of the code in the try block, the program will jump to the corresponding except block. This is used here to handle potential errors that might occur when trying to parse the date string.
7. **date\_object = datetime.strptime(date\_str, "%m/%d/%Y")**: This is the core line for validation and initial parsing.
   * datetime.strptime(): This is a class method of the datetime class that attempts to parse a string (date\_str) into a datetime object.
   * date\_str: This is the input string containing the date.
   * "%m/%d/%Y": This is the format code string that tells strptime exactly how the input date\_str is expected to be structured. %m represents the month as a zero-padded decimal number (e.g., 01, 12), %d represents the day of the month as a zero-padded decimal number (e.g., 01, 31), and %Y represents the year with century (e.g., 2023). If the date\_str does not exactly match this format, or if the date itself is invalid (like February 30th), strptime will raise a ValueError.
   * date\_object = ...: If strptime successfully parses the string, it returns a datetime object representing that date, which is then stored in the variable date\_object.
8. **return date\_object.strftime("%Y-%m-%d")**: This line is executed only if the strptime call in the try block was successful (meaning the date string was valid and in the correct input format).
   * date\_object.strftime(): This is a method of the datetime object (date\_object) that formats the date and time represented by the object into a string according to a specified format code.
   * "%Y-%m-%d": This is the format code string that tells strftime how to format the output string. %Y is the year with century, %m is the month, and %d is the day, separated by hyphens. The function returns this newly formatted string.
9. **except ValueError:**: This keyword starts an except block. This block is executed if a ValueError occurs in the preceding try block. A ValueError will be raised by datetime.strptime() if the input date\_str does not match the specified format ("%m/%d/%YYYY") or if the date itself is invalid (e.g., a non-existent date like 02/30/2023).
10. **return "Invalid Date"**: If a ValueError is caught, this line is executed, and the function returns the string "Invalid Date", indicating that the input date string was not valid according to the requirements.
11. **# AI-generated assert test cases**: This is a comment indicating that the following lines are test cases generated by the AI.
12. **assert validate\_and\_format\_date("...") == "..."**: These lines are assert statements used to test the validate\_and\_format\_date function with various input strings. They check if the function returns the expected formatted date for valid inputs and "Invalid Date" for invalid inputs, covering different scenarios and edge cases like invalid months, days, formats, and non-date strings.
13. **print("All assert test cases passed!")**: If all the assert statements pass without raising an AssertionError, this line is executed, printing the message "All assert test cases passed!" to the console, confirming that the function works correctly for the given test cases.