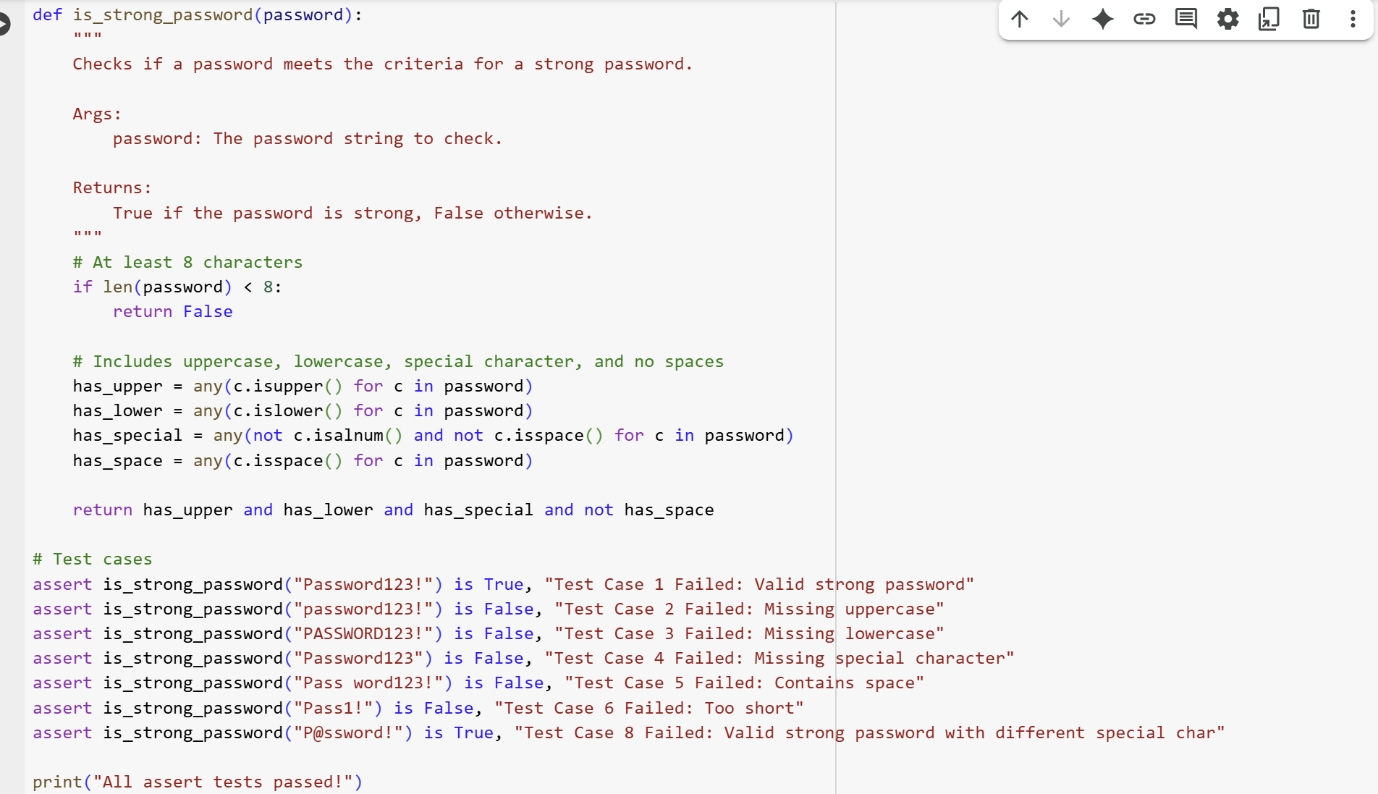
**HTNO: 2403A52134**

**Assignment-8**

**Task-1:**

Code:

Output:



Explanation:

def is\_strong\_password(password):

This line defines a function named is\_strong\_password that takes one argument, password.

    """  
    Checks if a password meets the criteria for a strong password.  
  
    Args:  
        password: The password string to check.  
  
    Returns:  
        True if the password is strong, False otherwise.  
    """

This is a docstring that explains what the function does, its arguments, and what it returns.

    # At least 8 characters  
    if len(password) < 8:  
        return False

This block checks if the length of the password is less than 8 characters. If it is, the function immediately returns False, as the password is not long enough.

    # Includes uppercase, lowercase, special character, and no spaces  
    has\_upper = any(c.isupper() for c in password)

This line checks if the password contains at least one uppercase letter. any() returns True if any character c in the password is uppercase (c.isupper()).

    has\_lower = any(c.islower() for c in password)

This line checks if the password contains at least one lowercase letter using c.islower().

    has\_special = any(not c.isalnum() and not c.isspace() for c in password)

This line checks if the password contains at least one special character. It iterates through each character c and checks if it is not alphanumeric (not c.isalnum()) and not a space (not c.isspace()).

    has\_space = any(c.isspace() for c in password)

This line checks if the password contains any space characters using c.isspace().

    return has\_upper and has\_lower and has\_special and not has\_space

This line returns the final result. The password is considered strong only if it has an uppercase letter (has\_upper), a lowercase letter (has\_lower), a special character (has\_special), and no spaces (not has\_space).

# Test cases  
assert is\_strong\_password("Password123!") is True, "Test Case 1 Failed: Valid strong password"

This is an assertion that checks if calling is\_strong\_password with "Password123!" returns True. If it doesn't, it will raise an AssertionError with the specified message. This is a valid strong password according to the criteria.

assert is\_strong\_password("password123!") is False, "Test Case 2 Failed: Missing uppercase"

This assertion checks if a password missing an uppercase letter returns False.

assert is\_strong\_password("PASSWORD123!") is False, "Test Case 3 Failed: Missing lowercase"

This assertion checks if a password missing a lowercase letter returns False.

assert is\_strong\_password("Password123") is False, "Test Case 4 Failed: Missing special character"

This assertion checks if a password missing a special character returns False.

assert is\_strong\_password("Pass word123!") is False, "Test Case 5 Failed: Contains space"

This assertion checks if a password containing a space returns False.

assert is\_strong\_password("Pass1!") is False, "Test Case 6 Failed: Too short"

This assertion checks if a password less than 8 characters returns False.

assert is\_strong\_password("P@ssword!") is True, "Test Case 8 Failed: Valid strong password with different special char"

This assertion checks if a valid strong password with a different special character returns True.

print("All assert tests passed!")

If none of the assertions fail, this line will print "All assert tests passed!" to the console.

**Task-2:**

Code:



Output:



Explanation:

**classify\_number(n) function:**

* This function takes one argument n.
* It first checks if the input n is not an integer or a float using isinstance(n, (int, float)). If it's neither, it returns the string "Invalid input".
* If the input is a number, it then checks if n is greater than 0. If true, it returns "Positive".
* If n is not greater than 0, it checks if n is less than 0. If true, it returns "Negative".
* If n is neither greater than 0 nor less than 0 (meaning it's 0), it returns "Zero".

**Test cases:**

* A list of tuples called test\_cases is created. Each tuple contains an input value and the expected output string for that input. This list includes examples of positive, negative, and zero numbers, as well as invalid inputs like strings, None, and a list. It also includes boundary conditions (-1, 0, 1).
* The code then iterates through each input\_val and expected\_output pair in the test\_cases list.
* Inside the loop, it calls the classify\_number function with the input\_val to get the actual\_output.
* An assert statement is used to check if the actual\_output matches the expected\_output.
* If the assertion fails (meaning the actual output does not match the expected output), an AssertionError is raised, and the message included in the assert statement (e.g., "assert classify\_number(10) == "Positive"") is displayed, indicating which test case failed.
* If all the assert statements pass without raising an error, the code will print "All assert tests passed!".

This structure is a common way to write unit tests for functions in Python, using assert statements within a loop to efficiently test multiple inputs.

**Task-3:**

Code & Output:



Explanation:

import re

This line imports the re module, which provides regular expression operations. This module is used to remove punctuation and spaces from the strings.

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.  
    """  
    # Normalize strings: convert to lowercase, remove spaces and punctuation  
    str1\_normalized = re.sub(r'[^a-z0-9]', '', str1.lower())  
    str2\_normalized = re.sub(r'[^a-z0-9]', '', str2.lower())  
  
    # Check if sorted characters are the same  
    return sorted(str1\_normalized) == sorted(str2\_normalized)

This block defines the is\_anagram function:

* It takes two string arguments, str1 and str2.
* The docstring explains the function's purpose, arguments, and return value.
* Inside the function, str1.lower() and str2.lower() convert both strings to lowercase to ensure case-insensitivity.
* re.sub(r'[^a-z0-9]', '', ...) uses a regular expression to remove any character that is NOT a lowercase letter (a-z) or a digit (0-9). This effectively removes spaces and punctuation. The result is stored in str1\_normalized and str2\_normalized.
* Finally, sorted(str1\_normalized) == sorted(str2\_normalized) compares the sorted lists of characters of the normalized strings. If the sorted lists are the same, the strings contain the same characters with the same frequencies, meaning they are anagrams, and the function returns True. Otherwise, it returns False.

# Test cases  
assert is\_anagram("listen", "silent") is True, "Test Case 1 Failed: Basic Anagram"

This is an assert statement that checks if "listen" and "silent" are correctly identified as anagrams (they are).

assert is\_anagram("hello", "world") is False, "Test Case 2 Failed: Not Anagrams"

This assert statement checks if "hello" and "world" are correctly identified as not being anagrams.

assert is\_anagram("Dormitory", "Dirty Room") is True, "Test Case 3 Failed: Anagram with spaces and different case"

This assert statement tests the function's ability to handle different cases and spaces by checking if "Dormitory" and "Dirty Room" are identified as anagrams.

assert is\_anagram("", "") is True, "Test Case 4 Failed: Empty Strings"

This assert statement checks the edge case of two empty strings, which are considered anagrams.

assert is\_anagram("a", "a") is True, "Test Case 5 Failed: Identical words"

This assert statement checks the edge case of two identical single-character strings.

assert is\_anagram("A man, a plan, a canal: Panama", "Panama, a canal: a plan, a man") is True, "Test Case 6 Failed: Complex Anagram"

This assert statement tests a more complex anagram with various punctuation and spaces.

assert is\_anagram("abc", "abcd") is False, "Test Case 7 Failed: Different length after normalization"

This assert statement checks if strings that have different lengths after normalization (meaning they cannot be anagrams) are correctly identified.

print("All assert tests passed!")

If all the assert statements above pass without raising an AssertionError, this line will be executed, indicating that all the tests were successful.

This cell provides a clear example of how to define a function with specific requirements and then use assert statements to test different scenarios and edge cases to ensure the function works correctly.

**Task-4:**

Code:

class Inventory:

    """

    A simple inventory class with stock management.

    """

    def \_\_init\_\_(self):

        """Initializes an empty inventory."""

        self.stock = {}

    def add\_item(self, name, quantity):

        """Adds the specified quantity of an item to the inventory."""

        if not isinstance(name, str) or not isinstance(quantity, int) or quantity <= 0:

            # print("Invalid input: Item name must be a string, quantity must be a positive integer.") # Removed print

            return

        if name in self.stock:

            self.stock[name] += quantity

        else:

            self.stock[name] = quantity

        # print(f"Added {quantity} of {name}. Current stock: {self.stock[name]}") # Removed print

    def remove\_item(self, name, quantity):

        """Removes the specified quantity of an item from the inventory."""

        if not isinstance(name, str) or not isinstance(quantity, int) or quantity <= 0:

            # print("Invalid input: Item name must be a string, quantity must be a positive integer.") # Removed print

            return

        if name in self.stock:

            if self.stock[name] >= quantity:

                self.stock[name] -= quantity

                # print(f"Removed {quantity} of {name}. Remaining stock: {self.stock[name]}") # Removed print

                if self.stock[name] == 0:

                    del self.stock[name] # Remove item from stock if quantity is zero

                    # print(f"{name} is now out of stock and removed from inventory.") # Removed print

            else:

                # print(f"Not enough stock of {name}. Available: {self.stock[name]}") # Removed print

                pass # Added pass to maintain structure after removing print

        else:

            # print(f"{name} is not in the inventory.") # Removed print

            pass # Added pass to maintain structure after removing print

    def get\_stock(self, name):

        """Returns the current stock quantity of an item."""

        return self.stock.get(name, 0) # Return 0 if item not in stock

# Test cases

inv = Inventory()

# Test adding items

inv.add\_item("Pen", 10)

assert inv.get\_stock("Pen") == 10, "Test Case 1 Failed: Add initial item"

inv.add\_item("Pen", 5)

assert inv.get\_stock("Pen") == 15, "Test Case 2 Failed: Add more of existing item"

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

assert inv.get\_stock("Book") == 3, "Test Case 3 Failed: Add new item"

# Test removing items

inv.remove\_item("Pen", 5)

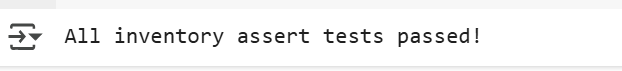
assert inv.get\_stock("Pen") == 10, "Test Case 4 Failed: Remove some quantity"

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

assert inv.get\_stock("Book") == 0, "Test Case 5 Failed: Remove all quantity"

print("All inventory assert tests passed!")

Output:



Explanation:

class Inventory:  
    """  
    A simple inventory class with stock management.  
    """  
    def \_\_init\_\_(self):  
        """Initializes an empty inventory."""  
        self.stock = {}

This block defines the Inventory class and its constructor (\_\_init\_\_).

* class Inventory: declares the beginning of the class definition.
* The docstring explains the class's purpose.
* def \_\_init\_\_(self): is the constructor method. It's called when a new Inventory object is created.
* self.stock = {} initializes an empty dictionary called stock as an instance variable. This dictionary will store the items and their quantities (e.g., {"Pen": 10, "Book": 3}).

    def add\_item(self, name, quantity):  
        """Adds the specified quantity of an item to the inventory."""  
        if not isinstance(name, str) or not isinstance(quantity, int) or quantity <= 0:  
            # print("Invalid input: Item name must be a string, quantity must be a positive integer.") # Removed print  
            return  
  
        if name in self.stock:  
            self.stock[name] += quantity  
        else:  
            self.stock[name] = quantity  
        # print(f"Added {quantity} of {name}. Current stock: {self.stock[name]}") # Removed print

This method add\_item is used to add items to the inventory:

* It takes the item name (string) and quantity (integer) as arguments.
* It includes input validation to ensure name is a string, quantity is an integer, and quantity is positive. If validation fails, it returns without doing anything.
* It checks if the name is already a key in the self.stock dictionary.
* If the item exists, the quantity is added to the current stock.
* If the item does not exist, it's added to the self.stock dictionary with the given quantity.
* The original print statements were commented out to make the code shorter as requested in a previous turn.

    def remove\_item(self, name, quantity):  
        """Removes the specified quantity of an item from the inventory."""  
        if not isinstance(name, str) or not isinstance(quantity, int) or quantity <= 0:  
            # print("Invalid input: Item name must be a string, quantity must be a positive integer.") # Removed print  
            return  
  
        if name in self.stock:  
            if self.stock[name] >= quantity:  
                self.stock[name] -= quantity  
                # print(f"Removed {quantity} of {name}. Remaining stock: {self.stock[name]}") # Removed print  
                if self.stock[name] == 0:  
                    del self.stock[name] # Remove item from stock if quantity is zero  
                    # print(f"{name} is now out of stock and removed from inventory.") # Removed print  
            else:  
                # print(f"Not enough stock of {name}. Available: {self.stock[name]}") # Removed print  
                pass # Added pass to maintain structure after removing print  
        else:  
            # print(f"{name} is not in the inventory.") # Removed print  
            pass # Added pass to maintain structure after removing print

This method remove\_item is used to remove items from the inventory:

* It takes the item name (string) and quantity (integer) to remove.
* It includes input validation similar to add\_item.
* It checks if the name is in the self.stock.
* If the item exists and the quantity to remove is less than or equal to the available stock, the quantity is subtracted from the stock.
* If the stock becomes zero after removal, the item is removed from the self.stock dictionary using del.
* If the item exists but there's not enough stock, a message is printed (commented out).
* If the item is not in the inventory, a message is printed (commented out).
* The original print statements were commented out to make the code shorter as requested in a previous turn, and pass statements were added to maintain the code structure.

    def get\_stock(self, name):  
        """Returns the current stock quantity of an item."""  
        return self.stock.get(name, 0) # Return 0 if item not in stock

This method get\_stock returns the current quantity of an item:

* It takes the item name as an argument.
* self.stock.get(name, 0) attempts to get the value associated with the name key from the self.stock dictionary. If the name is not found, it returns the default value 0.
* **Test cases**: The lines following this comment are test cases that demonstrate how to use the Inventory class and verify its functionality using assert statements. Each assert checks if the result of a method call is equal to the expected output. If an assert fails, it will raise an AssertionError with the provided message.

.

**Task-5:**

Code & Output:



Explanation:

from datetime import datetime

This line imports the datetime class from the datetime module. This class is necessary for working with dates and times, specifically for parsing and formatting date strings.

def validate\_and\_format\_date(date\_str):

This line defines a function named validate\_and\_format\_date that takes one argument, date\_str. This argument is expected to be the date string that needs to be validated and formatted.

    """  
    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 formatted date string in "YYYY-MM-DD" format if valid,  
        otherwise "Invalid Date".  
    """

This is a docstring that explains the purpose of the function, describes its argument (date\_str), and explains what the function returns.

    if not isinstance(date\_str, str):  
        return "Invalid Date"

This is a crucial line for error handling. It checks if the input date\_str is actually a string using the isinstance() function. If the input is not a string (for example, if it's None, an integer, or a list), the function immediately returns the string "Invalid Date" because it cannot process non-string inputs as dates.

    try:

This line starts a try block. This block contains code that might potentially raise an exception (an error). In this case, the code that attempts to parse the date string might fail if the string is not in the expected format or represents an invalid date.

        # Attempt to parse the date string with the expected format  
        date\_obj = datetime.strptime(date\_str, "%m/%d/%Y")

Inside the try block, this line attempts to parse the date\_str into a datetime object. datetime.strptime() is used for this purpose. The second argument, "%m/%d/%Y", is a format code that tells strptime to expect the date string to be in the format "Month/Day/Year" (e.g., "10/15/2023"). If date\_str does not match this format or represents an invalid date (like "02/30/2023"), a ValueError exception will be raised.

        # If parsing is successful, format the date to "YYYY-MM-DD"  
        return date\_obj.strftime("%Y-%m-%d")

If the strptime() call in the previous line is successful (meaning the date string was valid and in the correct format), this line is executed. date\_obj.strftime("%Y-%m-%d") formats the parsed datetime object into a new string with the format "Year-Month-Day" (e.g., "2023-10-15"). This formatted string is then returned by the function.

    except ValueError:

This line starts an except block that specifically catches a ValueError. This block is executed if a ValueError occurs within the preceding try block (which happens if strptime() fails).

        # If parsing fails due to incorrect format or invalid date, return "Invalid Date"  
        return "Invalid Date"

If a ValueError is caught, this line is executed, and the function returns the string "Invalid Date", indicating that the input string could not be successfully parsed as a valid date in the expected format.

# Test cases  
assert validate\_and\_format\_date("10/15/2023") == "2023-10-15", "Test Case 1 Failed: Valid date"

This is an assert statement that calls the validate\_and\_format\_date function with a valid date string "10/15/2023" and checks if the returned value is equal to the expected formatted string "2023-10-15". If it's not, an AssertionError is raised with the provided message.

assert validate\_and\_format\_date("02/30/2023") == "Invalid Date", "Test Case 2 Failed: Invalid day"

This assert statement tests an invalid date (February 30th) and checks if the function correctly returns "Invalid Date".

assert validate\_and\_format\_date("01/01/2024") == "2024-01-01", "Test Case 3 Failed: Valid date, leap year consideration"

This assert statement checks a valid date, including a date in a leap year, and verifies the correct formatting.

assert validate\_and\_format\_date("13/10/2023") == "Invalid Date", "Test Case 4 Failed: Invalid month"

This assert statement tests an invalid month (13) and checks if the function returns "Invalid Date".

assert validate\_and\_format\_date("10-15-2023") == "Invalid Date", "Test Case 5 Failed: Incorrect format"

This assert statement tests a date string in an incorrect format (using hyphens instead of slashes) and checks for the "Invalid Date" return.

assert validate\_and\_format\_date("10/15/23") == "Invalid Date", "Test Case 6 Failed: Incorrect year format"

This assert statement tests a date string with a two-digit year, which doesn't match the expected four-digit year format, and checks for "Invalid Date".

assert validate\_and\_format\_date("abc") == "Invalid Date", "Test Case 7 Failed: Non-date string"

This assert statement tests a non-date string input and verifies that the function returns "Invalid Date".

assert validate\_and\_format\_date(None) == "Invalid Date", "Test Case 8 Failed: None input"

This assert statement tests the case where the input is None and checks if the function correctly returns "Invalid Date" due to the isinstance check at the beginning.

print("All assert tests passed!")

If all the preceding assert statements pass without raising an AssertionError, this line is executed, indicating that all the test cases for the validate\_and\_format\_date function were successful.

This code effectively demonstrates how to validate and format date strings with error handling for invalid formats and values, along with a comprehensive set of assert tests to ensure its reliability.