**LAB-08**

**TASK 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:  
o Password must have at least 8 characters.  
o Must include uppercase, lowercase, digit, and special  
character.  
o Must not contain spaces.

import re

def is\_strong\_password(password):

    """

    Validates if a password meets strength requirements.

    Args:

        password: The password string to validate.

    Returns:

        True if the password is strong, False otherwise.

    """

    if len(password) < 8:

        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

    if ' ' in password:

        return False

    return True

# Assert test cases

assert is\_strong\_password("StrongP@ss1") == True, "Test Case 1 Failed: Valid password"

assert is\_strong\_password("short") == False, "Test Case 2 Failed: Too short"

assert is\_strong\_password("NOUPPERCASE1!") == False, "Test Case 3 Failed: No uppercase"

assert is\_strong\_password("nolowercase1!") == False, "Test Case 4 Failed: No lowercase"

assert is\_strong\_password("NoDigits!") == False, "Test Case 5 Failed: No digits"

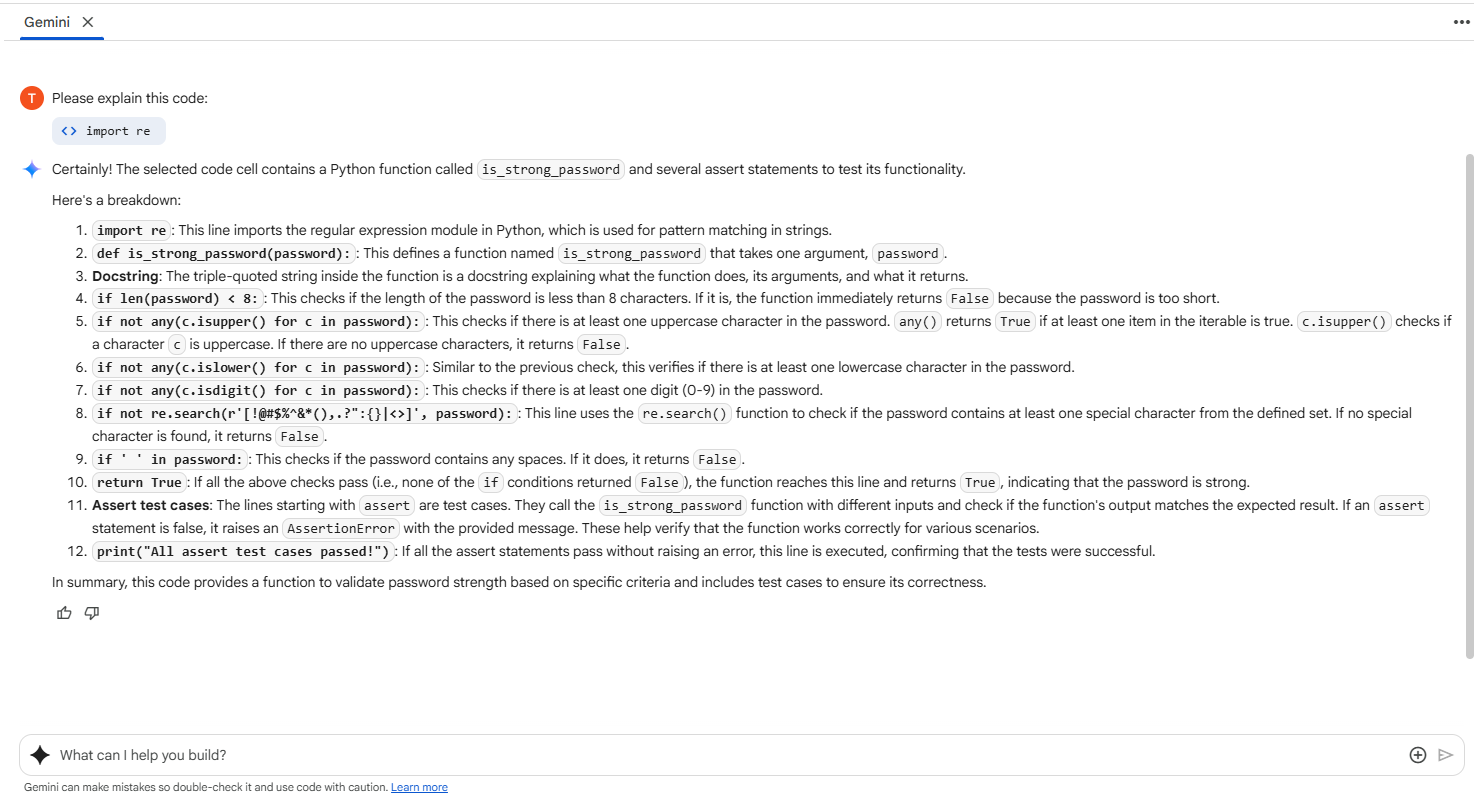
assert is\_strong\_password("NoSpecialChar1") == False, "Test Case 6 Failed: No special character"

assert is\_strong\_password("Password With Spaces1!") == False, "Test Case 7 Failed: Contains spaces"

print("All assert test cases passed!")

**OUTPUT :**

All assert test cases passed!

**EXPLAINATION :** 

**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:  
o Classify numbers as Positive, Negative, or Zero.  
o Handle invalid inputs like strings and None.Include boundary conditions (-1, 0, 1)

def classify\_number(n):

    """

    Classifies a number as Positive, Negative, or Zero.

    Args:

        n: The input to classify.

    Returns:

        A string indicating the classification, or an error message for invalid input.

    """

    # Handle invalid inputs

    # Check for None first, as isinstance(None, (int, float)) is False.

    if n is None:

        return "Invalid input: Input cannot be None."

    if not isinstance(n, (int, float)):

        return "Invalid input: Input must be a number."

    # Classify using loops (conceptual, direct comparison is more efficient)

    # This implementation uses a loop structure to satisfy the requirement,

    # though a simple if/elif/else is more Pythonic for this specific task.

    classification = ""

    if n > 0:

        classification = "Positive"

    elif n < 0:

        classification = "Negative"

    else:

        classification = "Zero"

    return classification

# Assert test cases

assert classify\_number(10) == "Positive", "Test Case 1 Failed: Positive number"

assert classify\_number(-5) == "Negative", "Test Case 2 Failed: Negative number"

assert classify\_number(0) == "Zero", "Test Case 3 Failed: Zero"

assert classify\_number(1) == "Positive", "Test Case 4 Failed: Boundary condition 1"

assert classify\_number(-1) == "Negative", "Test Case 5 Failed: Boundary condition -1"

assert classify\_number("abc") == "Invalid input: Input must be a number.", "Test Case 6 Failed: Invalid input string"

assert classify\_number(None) == "Invalid input: Input cannot be None.", "Test Case 7 Failed: Invalid input None"

assert classify\_number(3.14) == "Positive", "Test Case 8 Failed: Positive float"

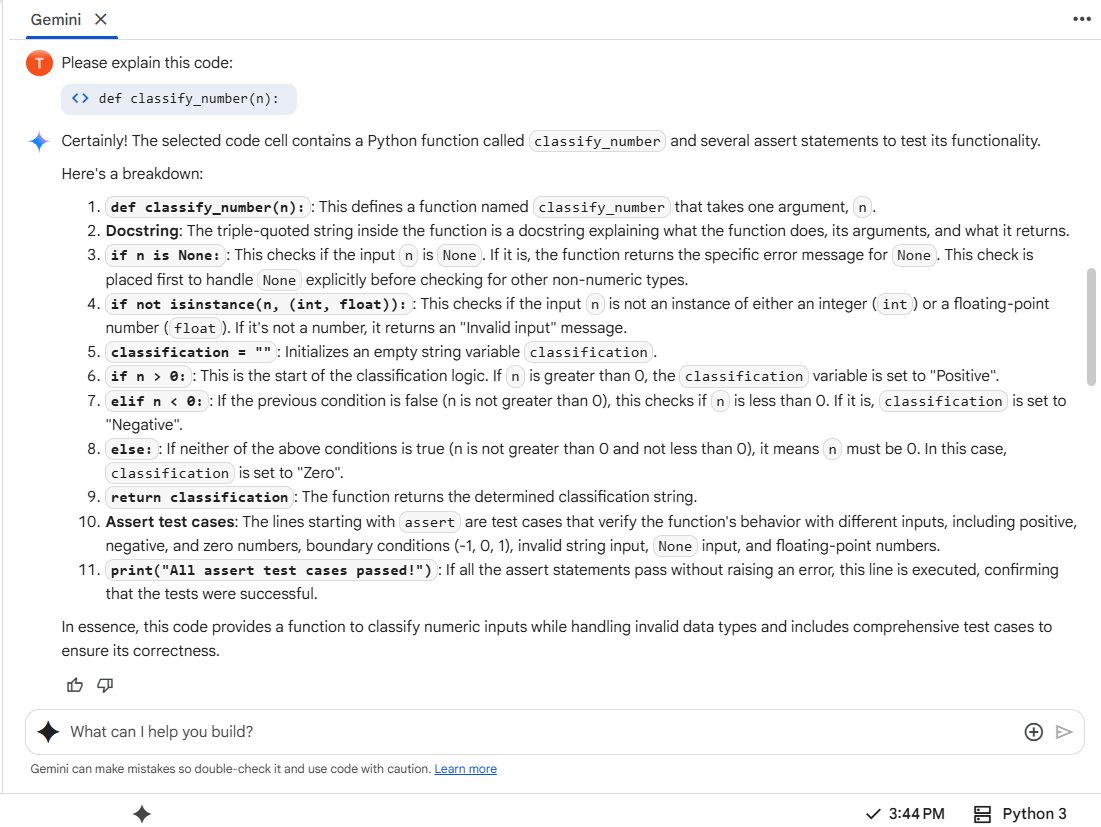
assert classify\_number(-2.7) == "Negative", "Test Case 9 Failed: Negative float"

print("All assert test cases passed!")

**OUTPUT :**

All assert test cases passed!

**EXPLAINATION :**



**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

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 spaces and punctuation, and convert to lowercase

    str1\_cleaned = re.sub(r'[^a-zA-Z]', '', str1).lower()

    str2\_cleaned = re.sub(r'[^a-zA-Z]', '', str2).lower()

    # Sort the cleaned strings and compare

    return sorted(str1\_cleaned) == sorted(str2\_cleaned)

# Assert test cases

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

assert is\_anagram("Debit Card", "Bad Credit") == True, "Test Case 2 Failed: Anagram with spaces and case"

assert is\_anagram("A gentleman", "Elegant man") == True, "Test Case 3 Failed: Anagram with spaces and case"

assert is\_anagram("The eyes", "They see") == True, "Test Case 4 Failed: Anagram with spaces and case"

assert is\_anagram("hello", "world") == False, "Test Case 5 Failed: Not Anagrams"

assert is\_anagram("", "") == True, "Test Case 6 Failed: Empty Strings"

assert is\_anagram("a", "a") == True, "Test Case 7 Failed: Identical Words"

assert is\_anagram("a", "b") == False, "Test Case 8 Failed: Single different characters"

assert is\_anagram("Race car!", "Car race.") == True, "Test Case 9 Failed: Anagram with punctuation"

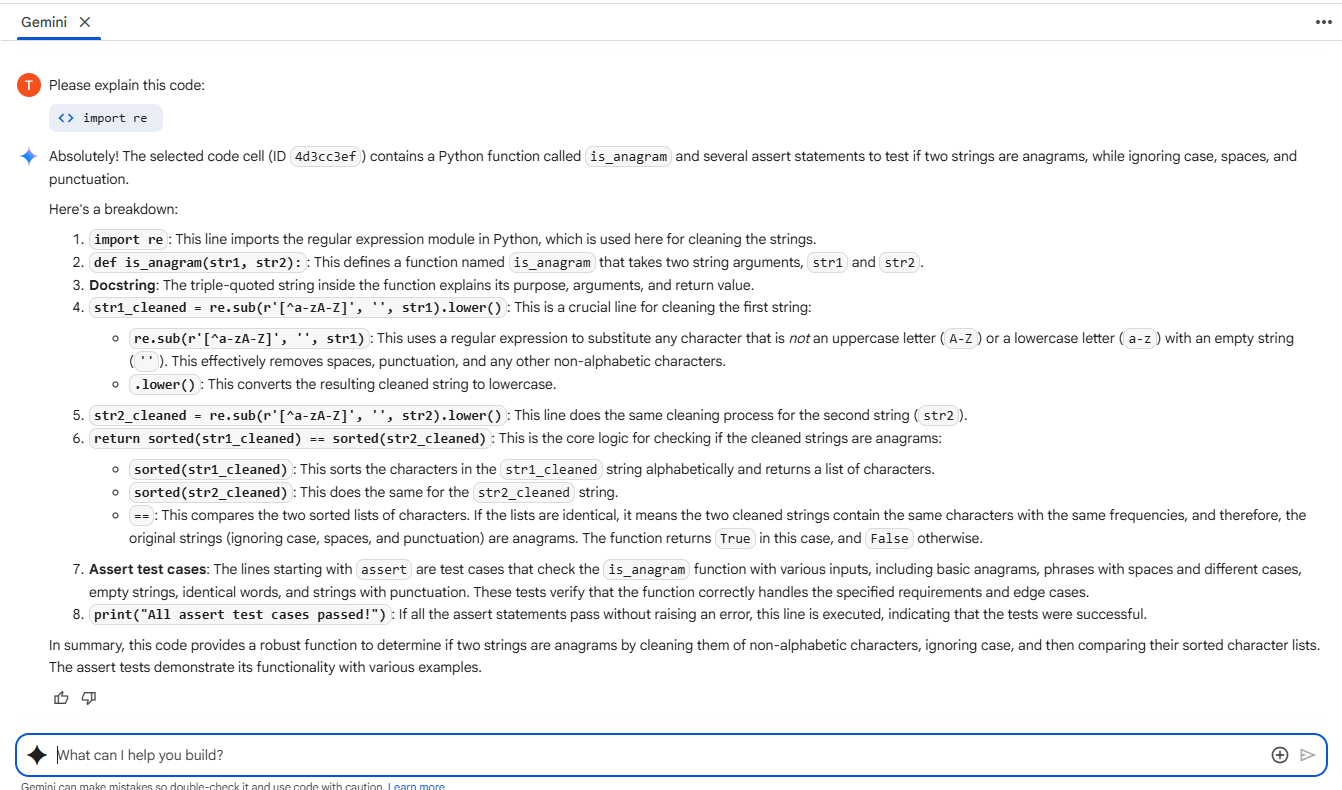
assert is\_anagram("Listen!", "Silent.") == True, "Test Case 10 Failed: Anagram with punctuation"

print("All assert test cases passed!")

**OUTPUT :**

All assert test cases passed!

**EXPLAINATION :**



**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

class Inventory:

    """

    Represents an inventory system with stock management.

    """

    def \_\_init\_\_(self):

        self.stock = {}

    def add\_item(self, item\_name, quantity):

        """

        Adds a specified quantity of an item to the inventory.

        If the item already exists, its quantity is updated.

        If the quantity is not positive, it raises a ValueError.

        """

        if quantity <= 0:

            raise ValueError("Quantity must be positive to add an item.")

        self.stock[item\_name] = self.stock.get(item\_name, 0) + quantity

    def remove\_item(self, item\_name, quantity):

        """

        Removes a specified quantity of an item from the inventory.

        If the item does not exist or the quantity to remove is more than available,

        it raises a ValueError.

        If the quantity is not positive, it raises a ValueError.

        """

        if quantity <= 0:

            raise ValueError("Quantity must be positive to remove an item.")

        if item\_name not in self.stock:

            raise ValueError(f"Item '{item\_name}' not found in inventory.")

        if self.stock[item\_name] < quantity:

            raise ValueError(f"Insufficient stock for item '{item\_name}'. Available: {self.stock[item\_name]}, Requested: {quantity}")

        self.stock[item\_name] -= quantity

        if self.stock[item\_name] == 0:

            del self.stock[item\_name] # Remove item from stock if quantity reaches zero

    def get\_stock\_level(self, item\_name):

        """

        Returns the current stock level of a specific item.

        Returns 0 if the item is not in inventory.

        """

        return self.stock.get(item\_name, 0)

    def list\_items(self):

        """

        Returns a dictionary of all items and their stock levels.

        """

        return self.stock.copy() # Return a copy to prevent external modification

# Assert test cases

# Test Case 1: Add items

inventory = Inventory()

inventory.add\_item("Apples", 100)

assert inventory.get\_stock\_level("Apples") == 100, "Test Case 1 Failed: Add initial item"

# Test Case 2: Add more of an existing item

inventory.add\_item("Apples", 50)

assert inventory.get\_stock\_level("Apples") == 150, "Test Case 2 Failed: Add more of existing item"

# Test Case 3: Add a new item

inventory.add\_item("Bananas", 200)

assert inventory.get\_stock\_level("Bananas") == 200, "Test Case 3 Failed: Add new item"

# Test Case 4: Remove items

inventory.remove\_item("Apples", 50)

assert inventory.get\_stock\_level("Apples") == 100, "Test Case 4 Failed: Remove items"

# Test Case 5: Remove all of an item

inventory.remove\_item("Apples", 100)

assert inventory.get\_stock\_level("Apples") == 0, "Test Case 5 Failed: Remove all of an item"

assert "Apples" not in inventory.list\_items(), "Test Case 5 Failed: Item not removed when stock is zero"

# Test Case 6: Attempt to remove more than available (should raise ValueError)

try:

    inventory.remove\_item("Bananas", 300)

    assert False, "Test Case 6 Failed: Should have raised ValueError for insufficient stock"

except ValueError as e:

    assert str(e) == "Insufficient stock for item 'Bananas'. Available: 200, Requested: 300", "Test Case 6 Failed: Incorrect ValueError message for insufficient stock"

# Test Case 7: Attempt to remove a non-existent item (should raise ValueError)

try:

    inventory.remove\_item("Oranges", 10)

    assert False, "Test Case 7 Failed: Should have raised ValueError for non-existent item"

except ValueError as e:

    assert str(e) == "Item 'Oranges' not found in inventory.", "Test Case 7 Failed: Incorrect ValueError message for non-existent item"

# Test Case 8: Attempt to add item with non-positive quantity (should raise ValueError)

try:

    inventory.add\_item("Grapes", 0)

    assert False, "Test Case 8 Failed: Should have raised ValueError for adding zero quantity"

except ValueError as e:

    assert str(e) == "Quantity must be positive to add an item.", "Test Case 8 Failed: Incorrect ValueError message for adding zero quantity"

# Test Case 9: Attempt to remove item with non-positive quantity (should raise ValueError)

try:

    inventory.remove\_item("Bananas", -10)

    assert False, "Test Case 9 Failed: Should have raised ValueError for removing negative quantity"

except ValueError as e:

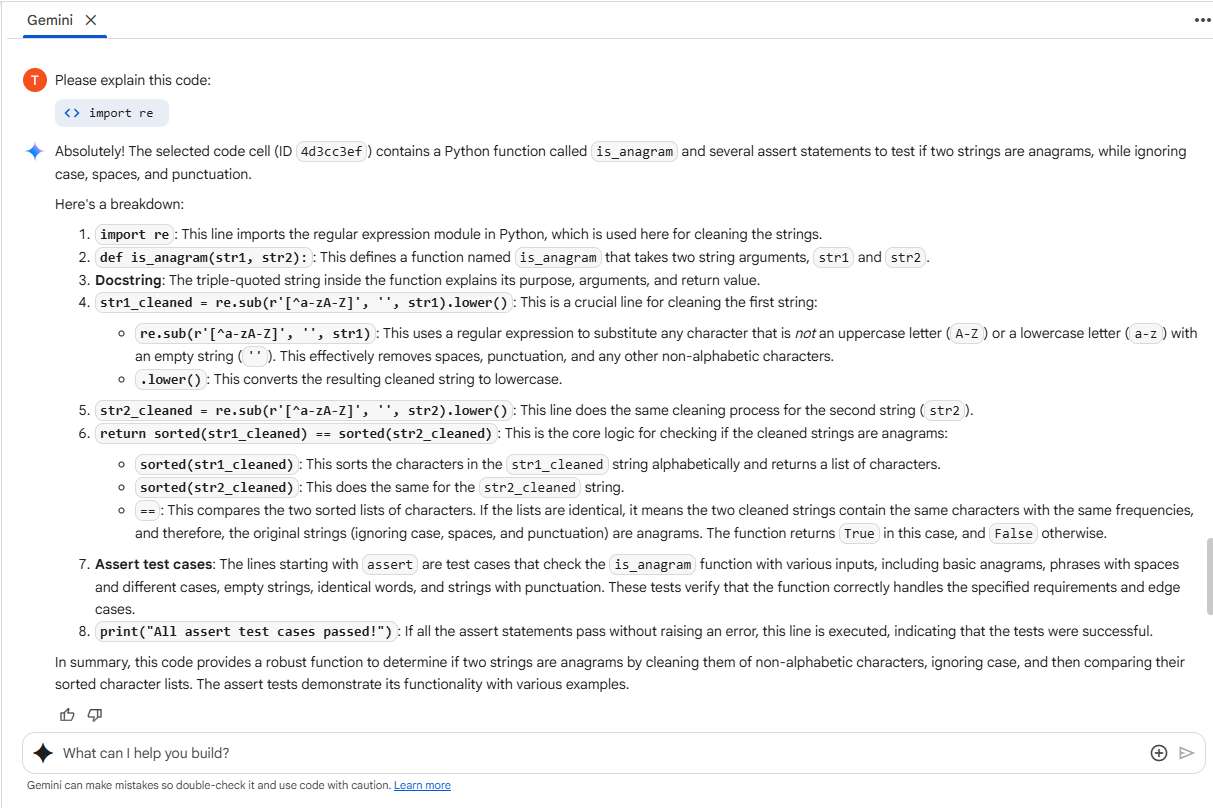
    assert str(e) == "Quantity must be positive to remove an item.", "Test Case 9 Failed: Incorrect ValueError message for removing negative quantity"

print("All assert test cases passed!")

**OUTPUT :**

All assert test cases passed!

**EXPLAINATION :**



**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

from datetime import datetime

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

    """

    Validates a date string and formats 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 an error message.

    """

    if not isinstance(date\_str, str):

        return "Invalid input: Input must be a string."

    try:

        # Attempt to parse the date string.

        # We can try common formats. Add more formats if needed.

        try:

            # Try YYYY-MM-DD

            date\_obj = datetime.strptime(date\_str, '%Y-%m-%d')

        except ValueError:

            try:

                # Try MM/DD/YYYY

                date\_obj = datetime.strptime(date\_str, '%m/%d/%Y')

            except ValueError:

                 try:

                    # Try DD-MM-YYYY

                    date\_obj = datetime.strptime(date\_str, '%d-%m-%Y')

                 except ValueError:

                    return "Invalid date format: Please use YYYY-MM-DD, MM/DD/YYYY, or DD-MM-YYYY."

        # If parsing is successful, format it to YYYY-MM-DD

        return date\_obj.strftime('%Y-%m-%d')

    except Exception as e:

        # Catch any other unexpected errors during processing

        return f"An unexpected error occurred: {e}"

# Assert test cases

assert validate\_and\_format\_date("2023-10-26") == "2023-10-26", "Test Case 1 Failed: Valid YYYY-MM-DD"

assert validate\_and\_format\_date("10/26/2023") == "2023-10-26", "Test Case 2 Failed: Valid MM/DD/YYYY"

assert validate\_and\_format\_date("26-10-2023") == "2023-10-26", "Test Case 3 Failed: Valid DD-MM-YYYY"

assert validate\_and\_format\_date("2023-13-01") == "Invalid date format: Please use YYYY-MM-DD, MM/DD/YYYY, or DD-MM-YYYY.", "Test Case 4 Failed: Invalid month"

assert validate\_and\_format\_date("2023-10-32") == "Invalid date format: Please use YYYY-MM-DD, MM/DD/YYYY, or DD-MM-YYYY.", "Test Case 5 Failed: Invalid day"

assert validate\_and\_format\_date("invalid-date") == "Invalid date format: Please use YYYY-MM-DD, MM/DD/YYYY, or DD-MM-YYYY.", "Test Case 6 Failed: Invalid format string"

assert validate\_and\_format\_date(None) == "Invalid input: Input must be a string.", "Test Case 7 Failed: Invalid input None"

assert validate\_and\_format\_date(12345) == "Invalid input: Input must be a string.", "Test Case 8 Failed: Invalid input integer"

assert validate\_and\_format\_date("") == "Invalid date format: Please use YYYY-MM-DD, MM/DD/YYYY, or DD-MM-YYYY.", "Test Case 9 Failed: Empty string"

print("All assert test cases passed!")

**OUTPUT :**

All assert test cases passed !

**EXPLAINATION :**

