**ASSIGNMENT-8**

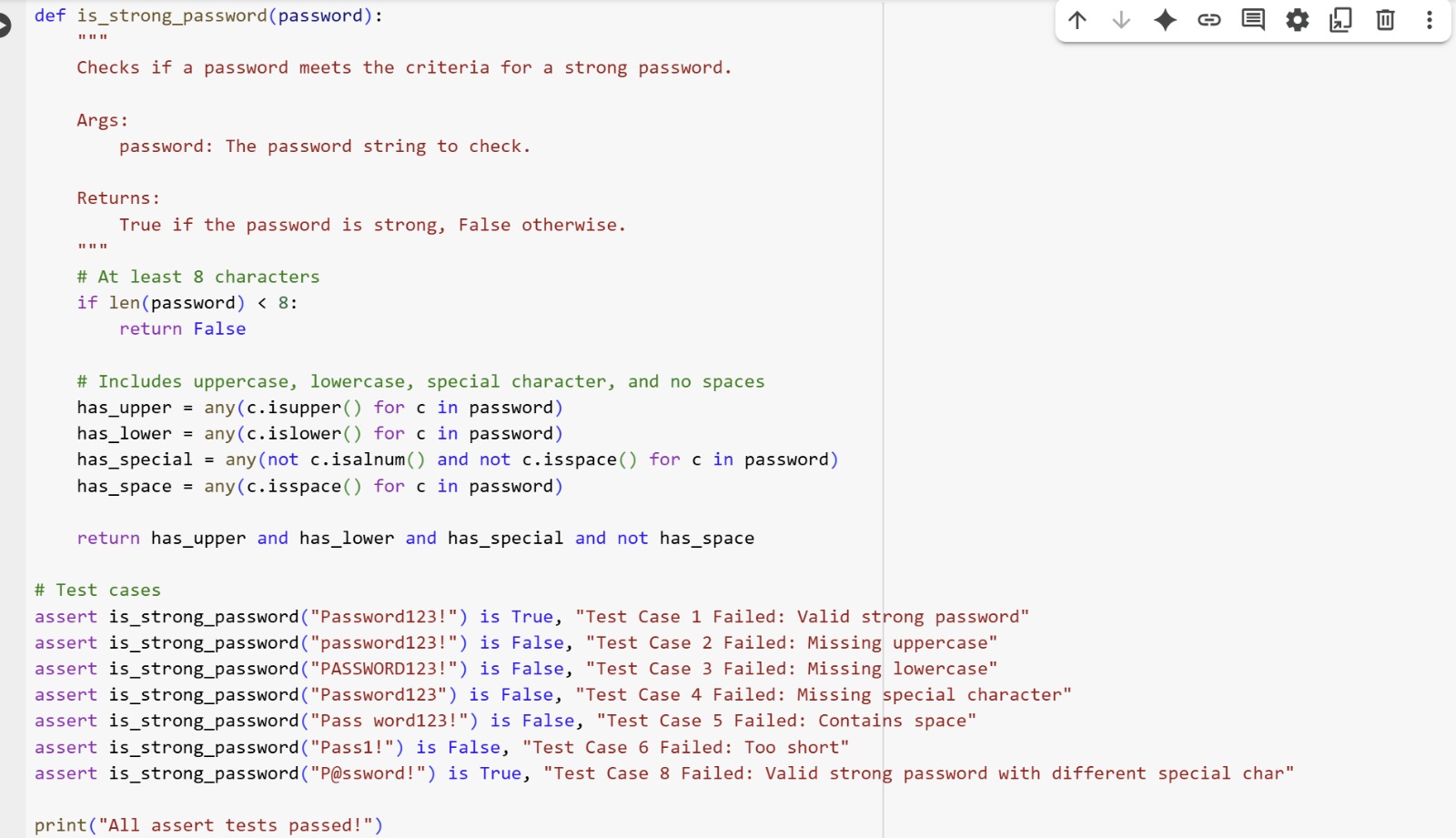
**H.NO:2403A52139**

* **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 implementthe validator function.

**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!")

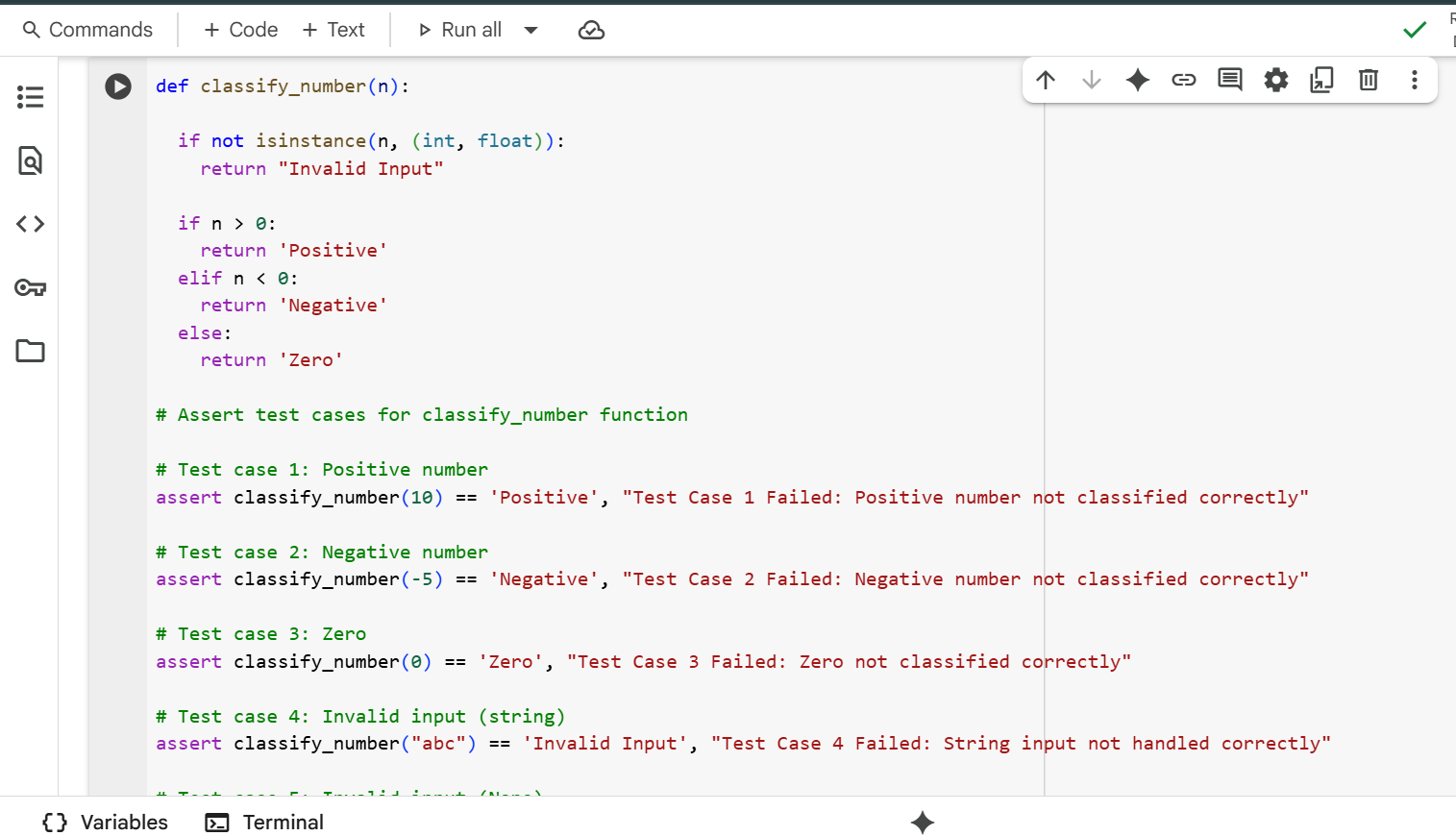
If none of the assertions fail, this line will print "All assert tests passed!" to the console**.**

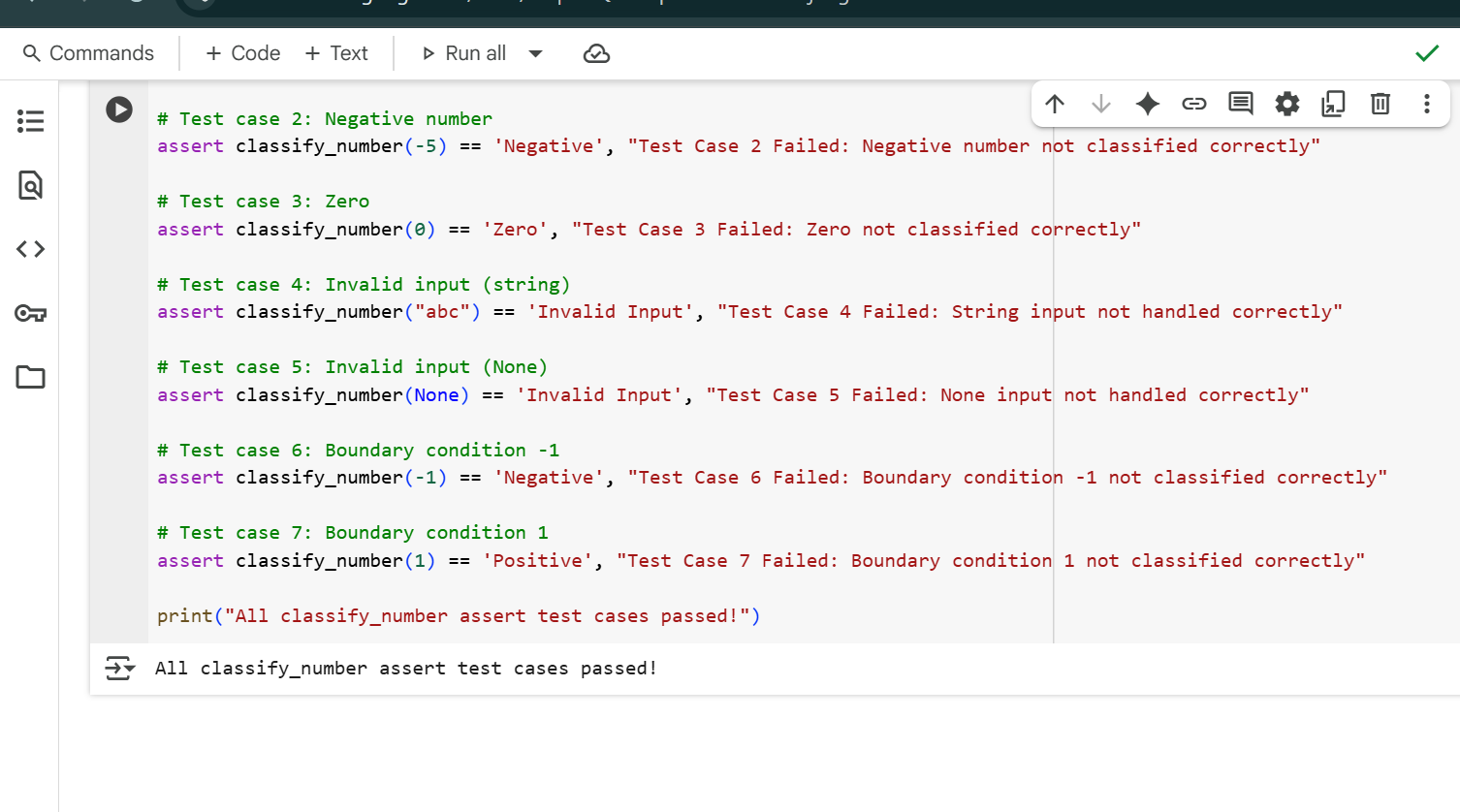
* **TASK-2**

(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).

**CODE AND OUTPUT:**

****

****

**EXPLANATION:**

This defines the function classify\_number that takes one argument n. The docstring explains the function's purpose, arguments, and return value.

  if not isinstance(n, (int, float)):  
    return "Invalid Input"

This line checks if the input n is not an integer or a float using isinstance(). If it's not a valid number type, the function immediately returns the string "Invalid Input".

  if n > 0:  
    return 'Positive'

If the input n is a valid number type and is greater than 0, the function returns the string 'Positive'.

  elif n < 0:  
    return 'Negative'

If the input n is a valid number type and is not greater than 0, this line checks if n is less than 0. If it is, the function returns the string 'Negative'.

  else:  
    return 'Zero'

If the input n is a valid number type and is neither greater than 0 nor less than 0, it must be 0. In this case, the function returns the string 'Zero'.

# Assert test cases for classify\_number function

This is a comment indicating the start of the assert test cases for the classify\_number function.

# Test case 1: Positive number  
assert classify\_number(10) == 'Positive', "Test Case 1 Failed: Positive number not classified correctly"

This assert statement tests the function with a positive number (10). It asserts that the function returns 'Positive'. If not, it raises an AssertionError with the specified message.

# Test case 2: Negative number  
assert classify\_number(-5) == 'Negative', "Test Case 2 Failed: Negative number not classified correctly"

This assert statement tests the function with a negative number (-5). It asserts that the function returns 'Negative'.

# Test case 3: Zero  
assert classify\_number(0) == 'Zero', "Test Case 3 Failed: Zero not classified correctly"

This assert statement tests the function with zero (0). It asserts that the function returns 'Zero'.

# Test case 4: Invalid input (string)  
assert classify\_number("abc") == 'Invalid Input', "Test Case 4 Failed: String input not handled correctly"

This assert statement tests the function with an invalid input (a string "abc"). It asserts that the function returns 'Invalid Input'.

# Test case 5: Invalid input (None)  
assert classify\_number(None) == 'Invalid Input', "Test Case 5 Failed: None input not handled correctly"

This assert statement tests the function with None as input. It asserts that the function returns 'Invalid Input'.

# Test case 6: Boundary condition -1  
assert classify\_number(-1) == 'Negative', "Test Case 6 Failed: Boundary condition -1 not classified correctly"

This assert statement tests the function with the boundary condition -1. It asserts that the function returns 'Negative'.

# Test case 7: Boundary condition 1  
assert classify\_number(1) == 'Positive', "Test Case 7 Failed: Boundary condition 1 not classified correctly"

This assert statement tests the function with the boundary condition 1. It asserts that the function returns 'Positive'.

print("All classify\_number assert test cases passed!")

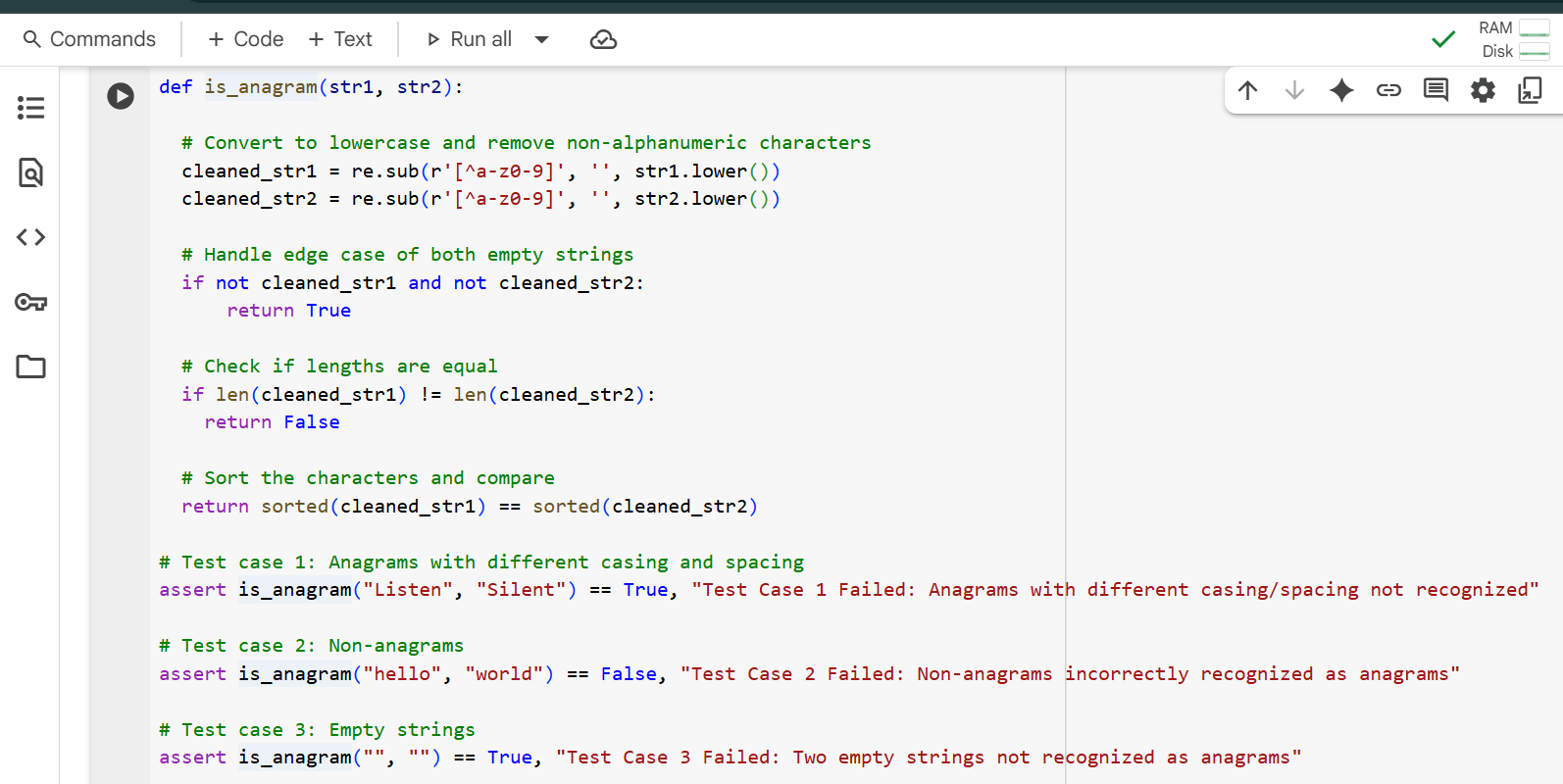
This line is executed if all the preceding assert statements pass. It prints a confirmation message to the console.

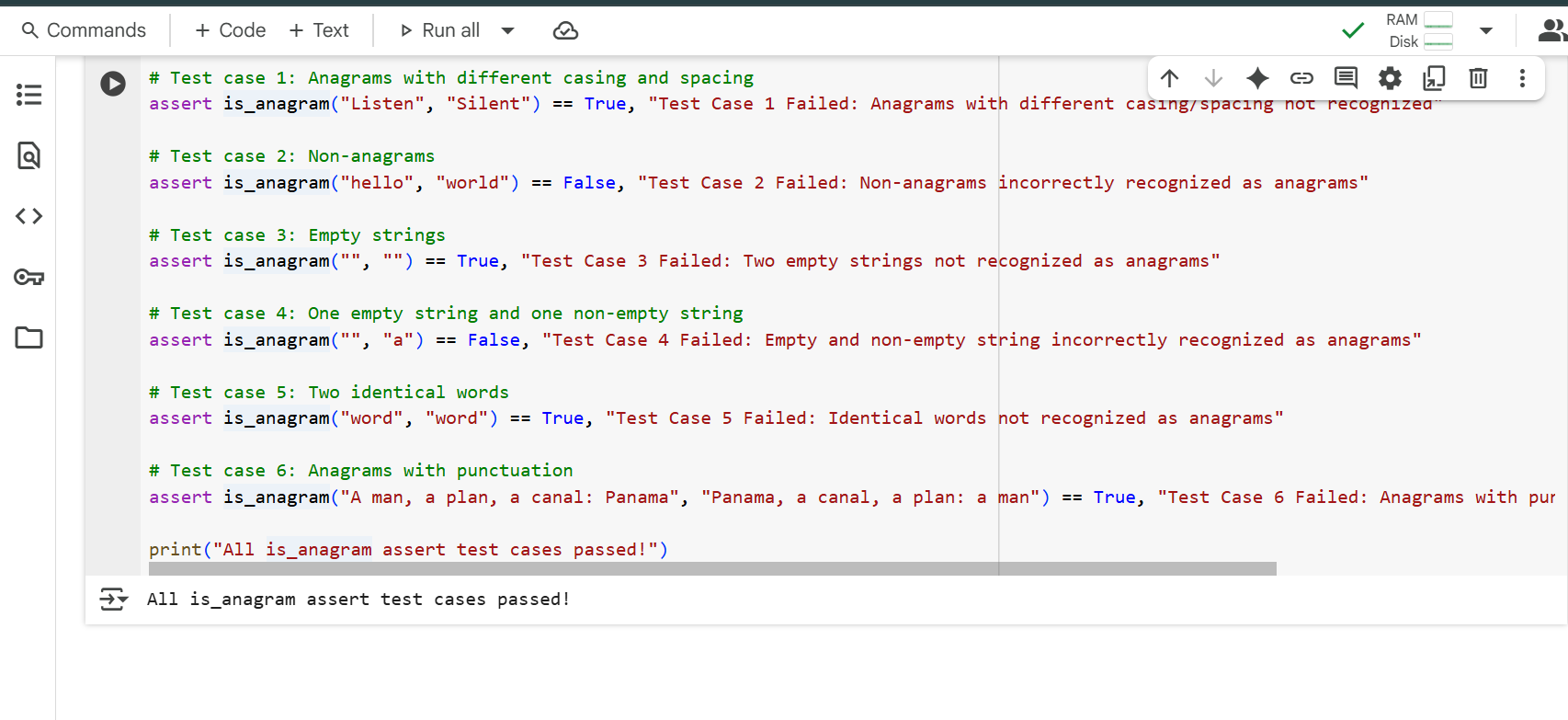
* **TASK-3**

(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).

**CODE AND OUTPUT:**

****

**EXPLANATION:**

import re

This line imports the re module, which is Python's regular expression library. It's used here to easily remove unwanted characters from the input 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.  
  """

This defines the function is\_anagram that takes two string arguments, str1 and str2. The docstring explains the function's purpose, arguments, and what it returns.

  # Convert to lowercase and remove non-alphanumeric characters  
  cleaned\_str1 = re.sub(r'[^a-z0-9]', '', str1.lower())  
  cleaned\_str2 = re.sub(r'[^a-z0-9]', '', str2.lower())

These two lines clean the input strings.

* str1.lower() and str2.lower() convert both strings to lowercase. This ensures that the comparison is case-insensitive.
* re.sub(r'[^a-z0-9]', '', ...) uses a regular expression to remove characters that are NOT lowercase letters (a-z) or digits (0-9). [^a-z0-9] matches any character that is not in the specified set. The matched characters are replaced with an empty string (''), effectively removing them. The result is stored in cleaned\_str1 and cleaned\_str2.

  # Handle edge case of both empty strings  
  if not cleaned\_str1 and not cleaned\_str2:  
      return True

This checks for an edge case where both cleaned strings are empty. If both are empty, they are considered anagrams (of each other), and the function returns True.

  # Check if lengths are equal  
  if len(cleaned\_str1) != len(cleaned\_str2):  
    return False

If the cleaned strings are not both empty, this line checks if their lengths are different. If the lengths are not equal, the strings cannot be anagrams, so the function returns False.

  # Sort the characters and compare  
  return sorted(cleaned\_str1) == sorted(cleaned\_str2)

This is the core logic for checking anagrams.

* sorted(cleaned\_str1) and sorted(cleaned\_str2) convert each cleaned string into a list of its characters and then sorts those lists alphabetically.
* The == operator compares the two sorted lists. If the lists of sorted characters are identical, it means the original cleaned strings contained the same characters with the same frequencies, so they are anagrams. The function returns True in this case, and False otherwise.

# Test case 1: Anagrams with different casing and spacing  
assert is\_anagram("Listen", "Silent") == True, "Test Case 1 Failed: Anagrams with different casing/spacing not recognized"

This is the first assert statement. It calls is\_anagram with "Listen" and "Silent" and asserts that the result is True. If it's not, it raises an AssertionError with a descriptive message.

The following assert statements (Test case 2 through Test case 6) follow the same pattern, testing the is\_anagram function with different inputs, including non-anagrams, empty strings, identical words, and strings with punctuation, and asserting the expected boolean result.

print("All is\_anagram assert test cases passed!")

This line is executed only if all the preceding assert statements pass without raising an error. It prints a success message to the console.

* **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)

**CODE:**

class Inventory:

  """

  A class to manage inventory items and their quantities.

  """

  def \_\_init\_\_(self):

    """

    Initializes an empty inventory dictionary.

    """

    self.items = {}

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

    """

    Adds a specified quantity of an item to the inventory.

    Args:

      item\_name: The name of the item.

      quantity: The quantity to add (must be positive).

    """

    if quantity <= 0:

      print("Quantity to add must be positive.")

      return

    if item\_name in self.items:

      self.items[item\_name] += quantity

    else:

      self.items[item\_name] = quantity

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

    """

    Removes a specified quantity of an item from the inventory.

    Args:

      item\_name: The name of the item.

      quantity: The quantity to remove (must be positive).

    """

    if quantity <= 0:

      print("Quantity to remove must be positive.")

      return

    if item\_name not in self.items:

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

      return

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

      print(f"Warning: Removing {quantity} of '{item\_name}', but only {self.items[item\_name]} in stock. Setting stock to 0.")

      self.items[item\_name] = 0

    else:

      self.items[item\_name] -= quantity

  def get\_stock(self, item\_name):

    """

    Returns the current stock of an item.

    Args:

      item\_name: The name of the item.

    Returns:

      The current stock of the item, or 0 if the item is not in inventory.

    """

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

# Create an instance of the Inventory class

inventory = Inventory()

# Add items

inventory.add\_item("apple", 10)

inventory.add\_item("banana", 5)

inventory.add\_item("apple", 5) # Add more apples

# Assert initial stock levels

assert inventory.get\_stock("apple") == 15, "Test Case 1 Failed: Adding items not working correctly"

assert inventory.get\_stock("banana") == 5, "Test Case 2 Failed: Adding items not working correctly"

# Remove items

inventory.remove\_item("apple", 3)

# Assert updated stock level after removal

assert inventory.get\_stock("apple") == 12, "Test Case 3 Failed: Removing items not working correctly"

# Attempt to remove more than available stock

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

# Assert stock is set to 0 after removing more than available

assert inventory.get\_stock("banana") == 0, "Test Case 4 Failed: Removing more than stock not handled correctly"

# Assert stock of a non-existent item is 0

assert inventory.get\_stock("orange") == 0, "Test Case 5 Failed: Getting stock of non-existent item not handled correctly"

print("All Inventory assert test cases passed!")

**EXPLANATION:**

class Inventory:  
  """  
  A class to manage inventory items and their quantities.  
  """

This defines a Python class named Inventory. The docstring explains that this class is used to manage inventory items and their quantities.

  def \_\_init\_\_(self):  
    """  
    Initializes an empty inventory dictionary.  
    """  
    self.items = {}

This is the constructor method (\_\_init\_\_) of the Inventory class. It's called when you create a new Inventory object.

* self: Refers to the instance of the class being created.
* self.items = {}: This line initializes an empty dictionary called items as an attribute of the Inventory instance. This dictionary will store the inventory, where keys will be item names (strings) and values will be their quantities (numbers).

  def add\_item(self, item\_name, quantity):  
    """  
    Adds a specified quantity of an item to the inventory.  
  
    Args:  
      item\_name: The name of the item.  
      quantity: The quantity to add (must be positive).  
    """  
    if quantity <= 0:  
      print("Quantity to add must be positive.")  
      return

This defines the add\_item method, which takes the item name and quantity to add as arguments.

* It first checks if the quantity is less than or equal to 0. If it is, it prints a message and returns, as you cannot add a non-positive quantity.

    if item\_name in self.items:  
      self.items[item\_name] += quantity  
    else:  
      self.items[item\_name] = quantity

* If the quantity is positive, this checks if the item\_name already exists as a key in the self.items dictionary.
* If the item exists, the quantity is added to the current stock (self.items[item\_name] += quantity).
* If the item does not exist, it's added to the dictionary with the given quantity as its initial stock (self.items[item\_name] = quantity).

  def remove\_item(self, item\_name, quantity):  
    """  
    Removes a specified quantity of an item from the inventory.  
  
    Args:  
      item\_name: The name of the item.  
      quantity: The quantity to remove (must be positive).  
    """

  if quantity <= 0:  
      print("Quantity to remove must be positive.")  
      return

This defines the remove\_item method, which takes the item name and quantity to remove as arguments.

* It first checks if the quantity is less than or equal to 0. If it is, it prints a message and returns, as you cannot remove a non-positive quantity.

    if item\_name not in self.items:  
      print(f"Item '{item\_name}' not found in inventory.")  
      return

* If the quantity is positive, this checks if the item\_name exists in the self.items dictionary. If the item is not found, it prints a message and returns.

    if self.items[item\_name] < quantity:  
      print(f"Warning: Removing {quantity} of '{item\_name}', but only {self.items[item\_name]} in stock. Setting stock to 0.")  
      self.items[item\_name] = 0  
    else:  
      self.items[item\_name] -= quantity

* If the item is found, this checks if the current stock (self.items[item\_name]) is less than the quantity to remove.
* If the stock is less than the quantity to remove, it prints a warning message indicating that the removal quantity exceeds the stock and sets the stock of that item to 0.
* If the stock is sufficient, the quantity is subtracted from the current stock (self.items[item\_name] -= quantity).

  def get\_stock(self, item\_name):  
    """  
    Returns the current stock of an item.  
  
    Args:  
      item\_name: The name of the item.  
  
    Returns:  
      The current stock of the item, or 0 if the item is not in inventory.  
    """  
    return self.items.get(item\_name, 0)

This defines the get\_stock method, which takes the item name as an argument.

* self.items.get(item\_name, 0) uses the dictionary's get() method. This method returns the value associated with item\_name if the key exists in the dictionary. If the key does not exist, it returns the default value specified (in this case, 0). This conveniently handles cases where you check for the stock of an item that hasn't been added yet.

# Create an instance of the Inventory class  
inventory = Inventory()

This line creates an instance (an object) of the Inventory class and assigns it to the variable inventory. This calls the \_\_init\_\_ method, initializing an empty inventory.

# Add items  
inventory.add\_item("apple", 10)  
inventory.add\_item("banana", 5)  
inventory.add\_item("apple", 5) # Add more apples

These lines demonstrate adding items to the inventory using the add\_item method. "apple" is added twice to show how the quantity is updated.

# Assert initial stock levels  
assert inventory.get\_stock("apple") == 15, "Test Case 1 Failed: Adding items not working correctly"  
assert inventory.get\_stock("banana") == 5, "Test Case 2 Failed: Adding items not working correctly"

These are assert statements that test the initial stock levels after adding items. They verify that the get\_stock method returns the expected quantities.

# Remove items  
inventory.remove\_item("apple", 3)

This line demonstrates removing 3 apples from the inventory using the remove\_item method.

# Assert updated stock level after removal  
assert inventory.get\_stock("apple") == 12, "Test Case 3 Failed: Removing items not working correctly"

This assert statement verifies that the stock of apples is correctly updated after removal.

# Attempt to remove more than available stock  
inventory.remove\_item("banana", 10)

This line attempts to remove 10 bananas, even though there are only 5 in stock. This will trigger the warning message within the remove\_item method.

# Assert stock is set to 0 after removing more than available  
assert inventory.get\_stock("banana") == 0, "Test Case 4 Failed: Removing more than stock not handled correctly"

This assert statement verifies that when attempting to remove more than the available stock, the stock is correctly set to 0.

# Assert stock of a non-existent item is 0  
assert inventory.get\_stock("orange") == 0, "Test Case 5 Failed: Getting stock of non-existent item not handled correctly"

This assert statement tests the get\_stock method for an item that has not been added to the inventory ("orange"). It asserts that the method correctly returns 0.

print("All Inventory assert test cases passed!")

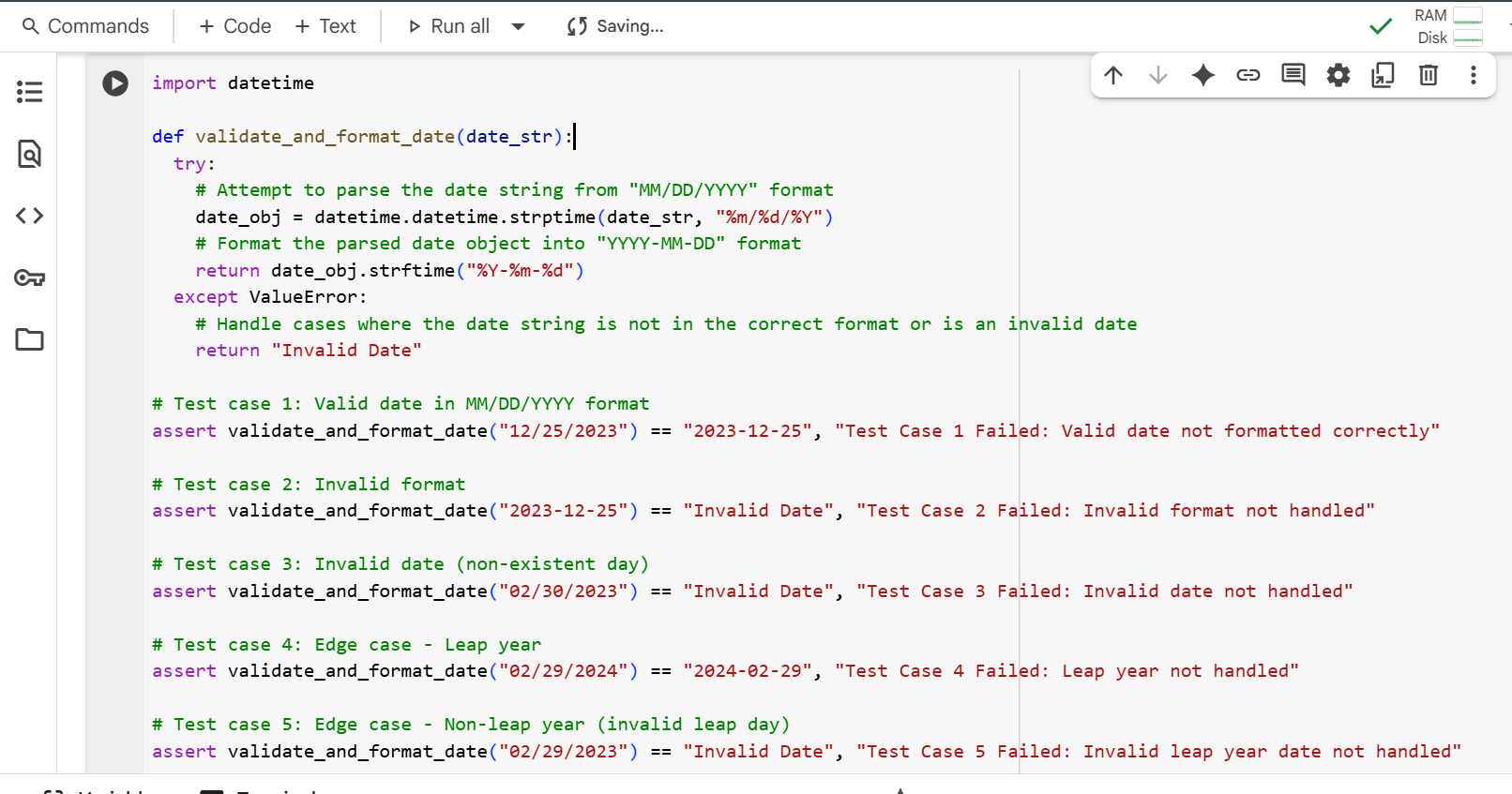
This line is executed only if all the preceding assert statements pass. It prints a success message to the console**.**

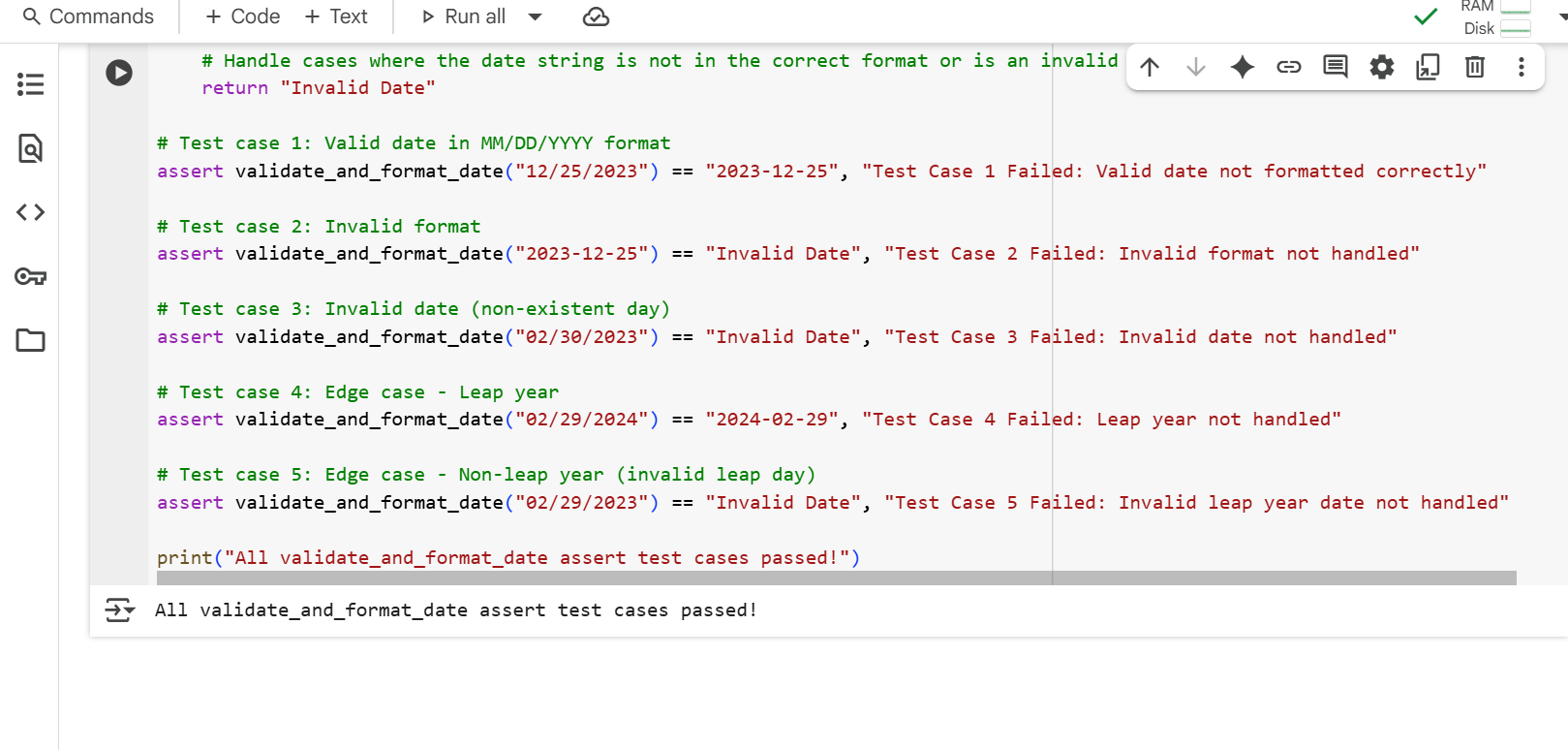
* **TASK-5**

(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".

**CODE AND OUTPUT**

****

**EXPLANATION:**

import datetime

This line imports the datetime module, which provides classes for working with dates and times. We'll use this module for parsing and formatting date strings.

def validate\_and\_format\_date(date\_str):  
  """  
  Validates and formats a date string from "MM/DD/YYYY" to "YYYY-MM-DD".  
  
  Args:  
    date\_str: The date string in "MM/DD/YYYY" format.  
  
  Returns:  
    The date string in "YYYY-MM-DD" format if valid, otherwise "Invalid Date".  
  """

This defines the function validate\_and\_format\_date that takes one argument, date\_str, which is expected to be a string representing a date. The docstring explains the function's purpose, arguments, and what it returns.

  try:

This line starts a try block. Code within this block will be attempted, and if an error (specifically a ValueError in this case) occurs, the code within the corresponding except block will be executed. This is used here to gracefully handle invalid date formats or non-existent dates.

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

Inside the try block, this line attempts to parse the input date\_str into a datetime object.

* datetime.datetime.strptime() is a class method that parses a string according to a specified format.
* date\_str: The input string to be parsed.
* "%m/%d/%Y": This is the format code that tells strptime to expect the date string in "Month/Day/Year" format (e.g., "12/25/2023"). If date\_str does not match this format or represents an invalid date (like "02/30/2023"), a ValueError will be raised.

    # Format the parsed date object into "YYYY-MM-DD" format  
    return date\_obj.strftime("%Y-%m-%d")

If the strptime call is successful (meaning the date string was valid and in the correct format), this line is executed.

* date\_obj.strftime("%Y-%m-%d") is a method of the datetime object that formats the date into a string according to a specified format code.
* "%Y-%m-%d": This format code tells strftime to format the date as "Year-Month-Day" (e.g., "2023-12-25"). The formatted date string is then returned by the function.

  except ValueError:  
    # Handle cases where the date string is not in the correct format or is an invalid date  
    return "Invalid Date"

This is the except block that is executed if a ValueError occurs within the try block (which happens when strptime fails to parse the date string).

* return "Invalid Date": If a ValueError is caught, the function returns the string "Invalid Date" to indicate that the input date string was not valid.

# Test case 1: Valid date in MM/DD/YYYY format  
assert validate\_and\_format\_date("12/25/2023") == "2023-12-25", "Test Case 1 Failed: Valid date not formatted correctly"

This is the first assert statement. It calls the validate\_and\_format\_date function with a valid date string "12/25/2023" and asserts that the function returns the correctly formatted string "2023-12-25". If the result is not as expected, an AssertionError is raised with the provided message.

The following assert statements (Test case 2 through Test case 5) test the function with different inputs, including invalid formats, non-existent dates, and leap year cases, and assert the expected return values ("Invalid Date" or the correctly formatted date).

print("All validate\_and\_format\_date assert test cases passed!")

This line is executed only if all the preceding assert statements pass without raising an AssertionError. It prints a success message to the console.