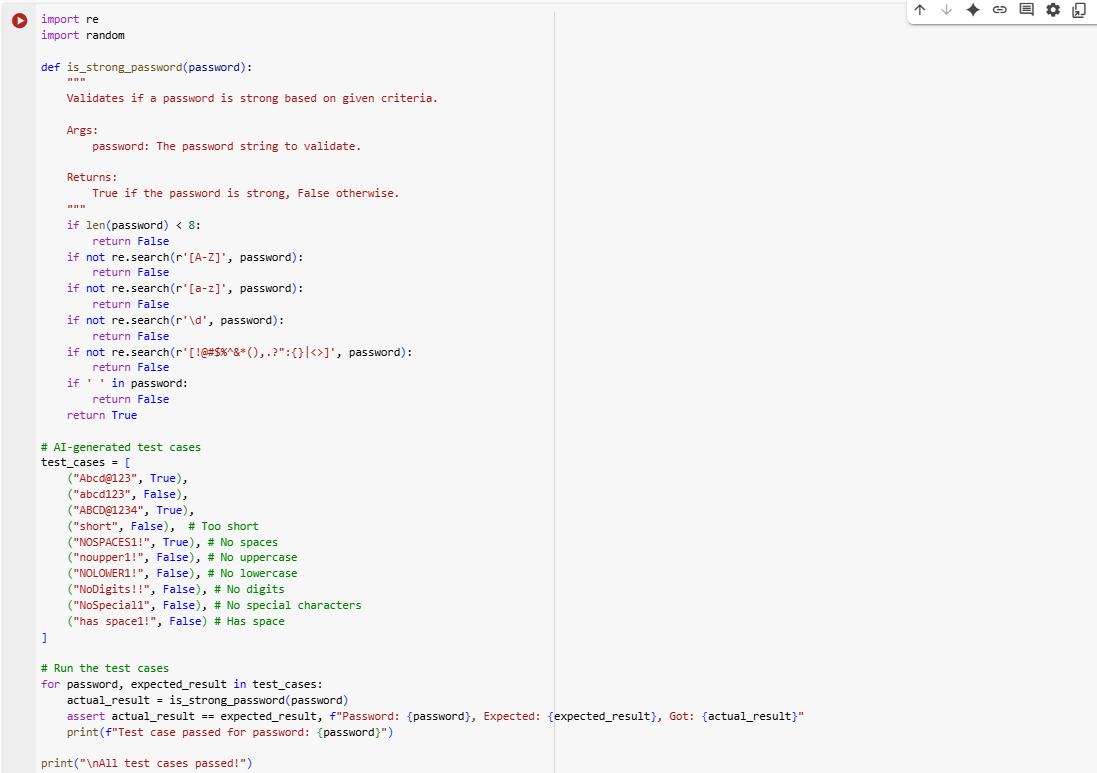
Assignment – 8.1

Task - 1:

Code:



Output:

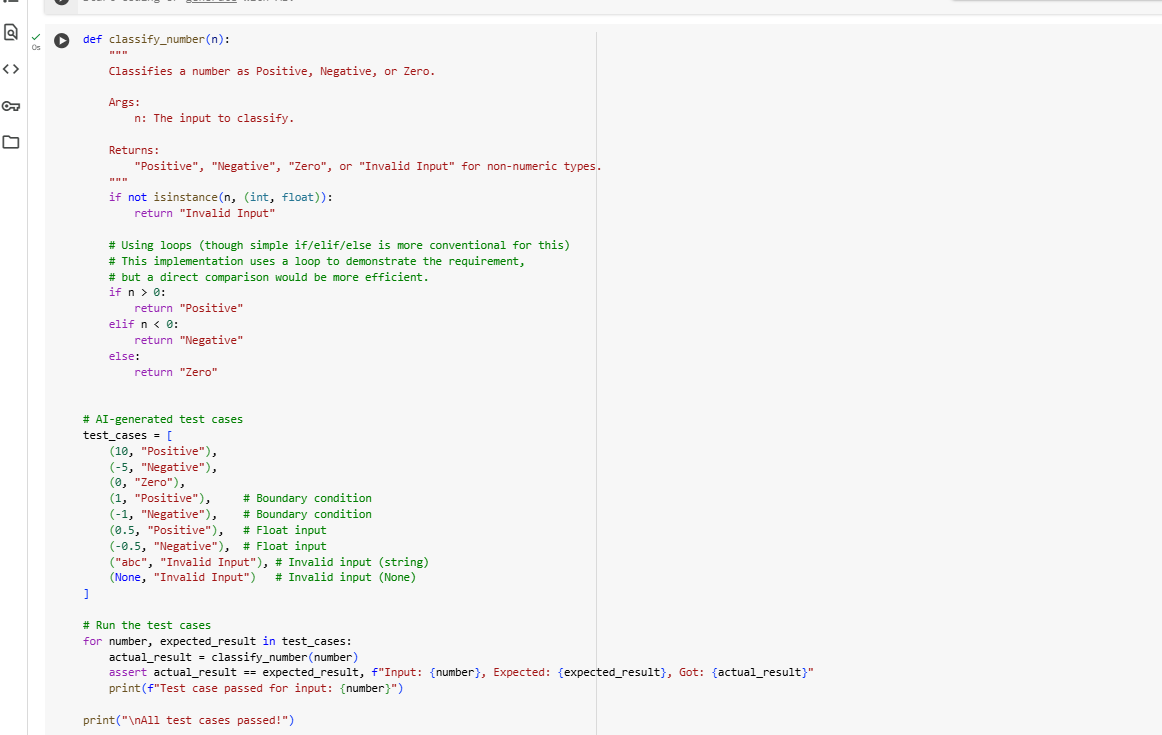


Explanation:

1. **Import necessary libraries**: The code starts by importing the re module for regular expressions (used for pattern matching in strings) and the random module (although random is imported, it's not actually used in this specific code snippet).
2. **Define the is\_strong\_password function**: This function takes a password string as input and checks if it meets the defined criteria for a strong password.
3. **Check password length**: Inside the function, the first check is if len(password) < 8:. If the password has fewer than 8 characters, the function immediately returns False.
4. **Check for uppercase letters**: if not re.search(r'[A-Z]', password): uses a regular expression to search for at least one uppercase letter ([A-Z]). If no uppercase letter is found, it returns False.
5. **Check for lowercase letters**: if not re.search(r'[a-z]', password): uses a regular expression to search for at least one lowercase letter ([a-z]). If no lowercase letter is found, it returns False.
6. **Check for digits**: if not re.search(r'\d', password): uses a regular expression to search for at least one digit (\d). If no digit is found, it returns False.
7. **Check for special characters**: if not re.search(r'[!@#$%^&\*(),.?":{}|<>]', password): uses a regular expression to search for at least one character from the specified set of special characters. If none are found, it returns False.
8. **Check for spaces**: if ' ' in password: checks if there is a space character in the password string. If a space is found, it returns False.
9. **Return True if all checks pass**: If the password passes all the above checks, the function reaches the end and returns True, indicating that the password is strong.
10. **Define test cases**: A list of tuples called test\_cases is created. Each tuple contains a password string and the expected boolean result (True for a strong password, False otherwise). These are used to test the is\_strong\_password function.
11. **Run the test cases**: The code then iterates through the test\_cases list.
12. **Call the function and assert the result**: Inside the loop, for each test case, the is\_strong\_password function is called with the password, and the result is stored in actual\_result. An assert statement is used to check if the actual\_result matches the expected\_result. If they don't match, an AssertionError is raised, indicating a failed test case.
13. **Print test case status**: If the assert statement passes, a message is printed indicating that the test case for that password passed.
14. **Print completion message**: After the loop finishes (meaning all test cases passed), a message "All test cases passed!" is printed.

Task – 2:

Code:



Output:

A screenshot of a computer

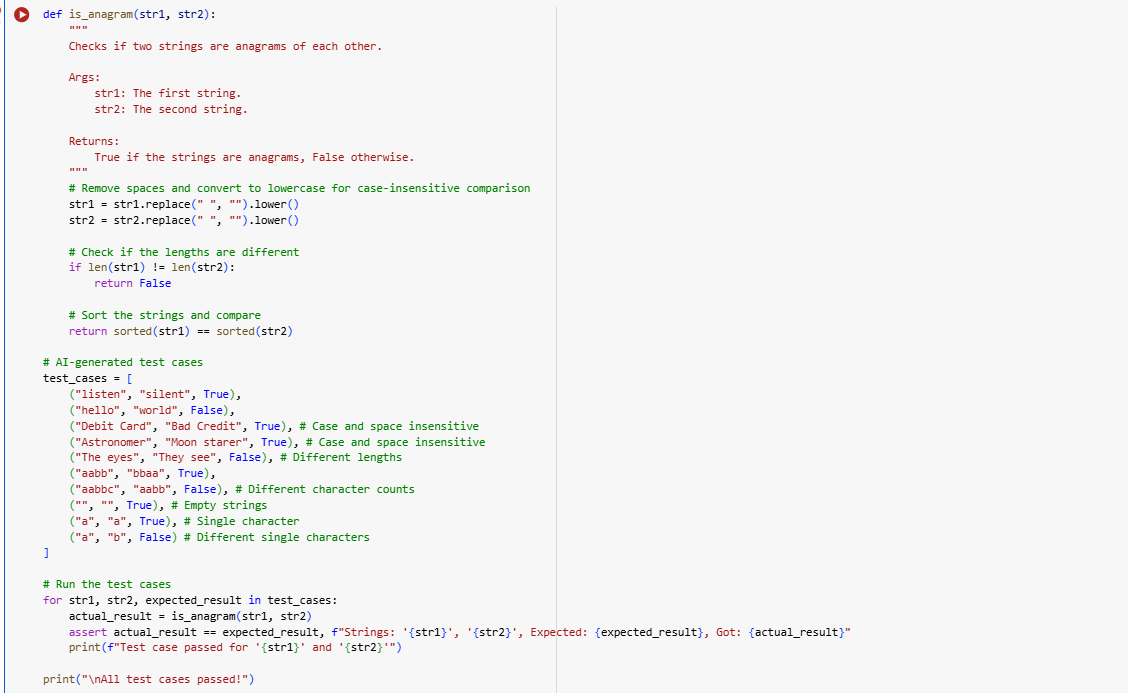
AI-generated content may be incorrect.

Explanation:

1. **Define the classify\_number function**: This function takes one argument, n, which is the input number (or other data type) to be classified.
2. **Check for invalid input**: The first if statement, if not isinstance(n, (int, float)):, checks if the input n is neither an integer (int) nor a floating-point number (float). If it's not a number, the function immediately returns the string "Invalid Input".
3. **Check if the number is positive**: The if n > 0: condition checks if the number n is greater than zero. If it is, the function returns the string "Positive".
4. **Check if the number is negative**: If the number is not greater than zero, the code moves to the elif n < 0: condition, which checks if the number n is less than zero. If it is, the function returns the string "Negative".
5. **Handle zero**: If the number is neither greater than nor less than zero, it must be zero. The else: block handles this case and returns the string "Zero".
6. **Define test cases**: A list of tuples called test\_cases is created. Each tuple contains an input value (number) and the expected\_result (the string that classify\_number should return for that input). These test cases cover various scenarios, including positive, negative, and zero numbers, boundary conditions (1 and -1), floating-point numbers, and invalid inputs (a string and None).
7. **Iterate through test cases**: The for loop iterates through each test\_cases tuple.
8. **Call the function and store the actual result**: Inside the loop, actual\_result = classify\_number(number) calls the classify\_number function with the current number from the test case and stores the returned value in the actual\_result variable.
9. **Assert the result**: assert actual\_result == expected\_result, f"Input: {number}, Expected: {expected\_result}, Got: {actual\_result}" checks if the actual\_result is equal to the expected\_result for the current test case. If they are not equal, an AssertionError is raised, indicating a test failure. The error message includes the input, expected result, and the actual result to help in debugging.
10. **Print success message for each test case**: If the assert statement passes without raising an error, the line print(f"Test case passed for input: {number}") is executed, indicating that the current test case was successful.
11. **Print final completion message**: After the loop has finished checking all the test cases, print("\nAll test cases passed!") is executed, confirming that all the provided test cases passed the assertions.

Task – 3:

Code:



Output:

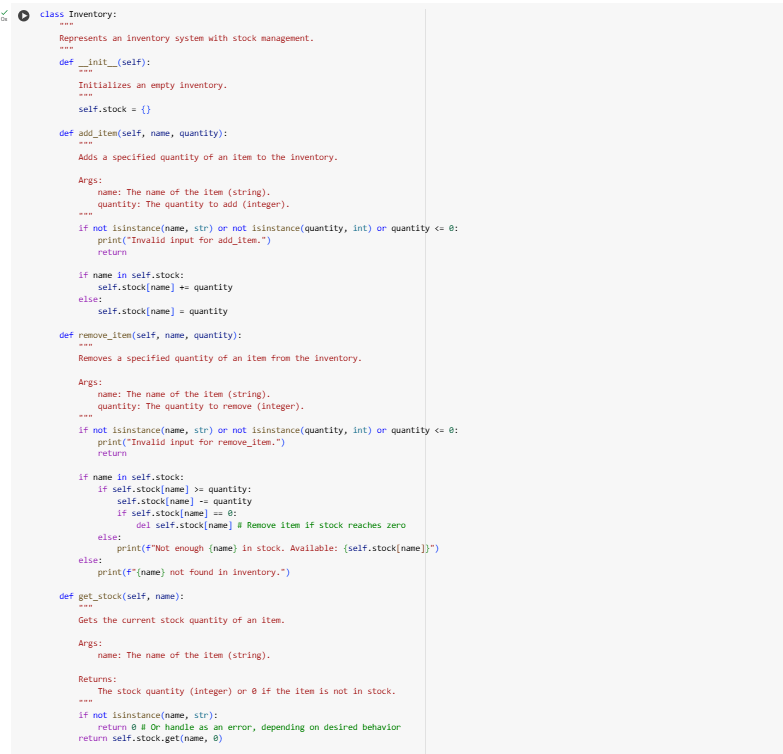


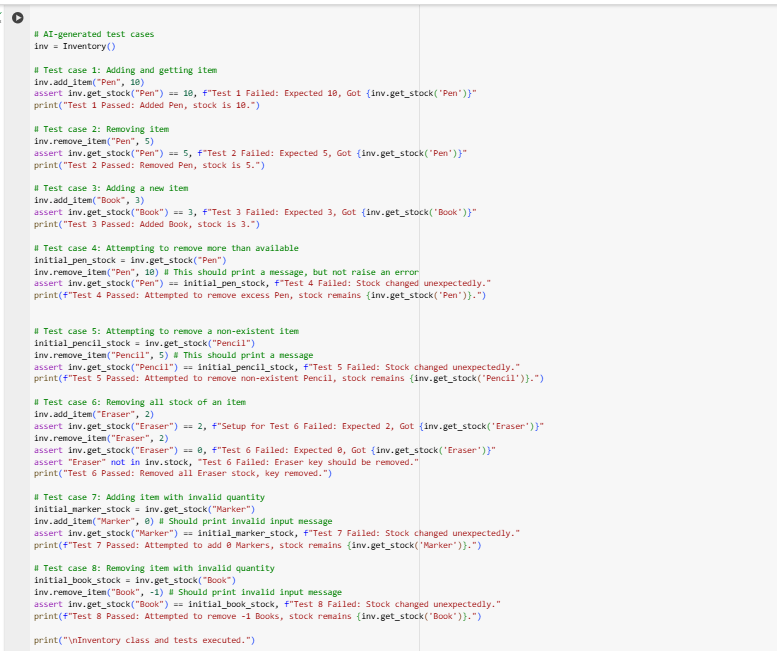
Explanation:

1. **Import Counter**: The line from collections import Counter imports the Counter class from the collections module. The Counter class is a specialized dictionary subclass for counting hashable objects.
2. **Define the is\_anagram function**: This defines a function named is\_anagram that takes two arguments, str1 and str2, which are the two strings you want to check if they are anagrams of each other.
3. **Normalize the strings**:
   * str1 = str1.replace(" ", "").lower(): This line creates a new string str1 by removing all spaces from the original str1 using .replace(" ", "") and then converting the entire string to lowercase using .lower().
   * str2 = str2.replace(" ", "").lower(): This does the same normalization for the second string, str2. This ensures that the comparison is case-insensitive and ignores spaces.
4. **Check if lengths are different**: if len(str1) != len(str2): return False checks if the lengths of the normalized strings are different. If they are, the strings cannot be anagrams, so the function immediately returns False.
5. **Compare character counts using Counter**: return Counter(str1) == Counter(str2) is the core of the anagram check.
   * Counter(str1) creates a Counter object for the normalized str1. This object is like a dictionary where the keys are the characters in the string and the values are the counts of each character.
   * Counter(str2) does the same for the normalized str2.
   * == compares the two Counter objects. Two Counter objects are equal if they contain the same keys (characters) with the same values (counts). If the character counts match exactly for both strings, the function returns True (indicating they are anagrams), otherwise it returns False.
6. **Define test cases**: The test\_cases list contains several tuples. Each tuple represents a test case with the first string, the second string, and the expected\_result (True if they are anagrams, False otherwise). These test cases cover various scenarios, including actual anagrams, non-anagrams, strings with spaces and different cases, empty strings, and strings with different character counts.
7. **Run the test cases**: The for loop iterates through each tuple in the test\_cases list.
8. **Call the function and assert the result**: Inside the loop:
   * actual\_result = is\_anagram(str1, str2) calls the is\_anagram function with the current pair of strings and stores the returned boolean value.
   * assert actual\_result == expected\_result, ... checks if the actual\_result matches the expected\_result for that test case. If they don't match, an AssertionError is raised, indicating a test failure. The message in the assert provides details about the strings, expected result, and the actual result.
9. **Print test case status**: print(f"Test case passed for '{str1}' and '{str2}'") is executed if the assert statement does not raise an error, confirming that the test case passed.
10. **Print final success message**: print("\nAll test cases passed!") is printed after the loop finishes, indicating that all the test cases in the list passed successfully.

Task – 4:

Code:





Output:

A screenshot of a computer

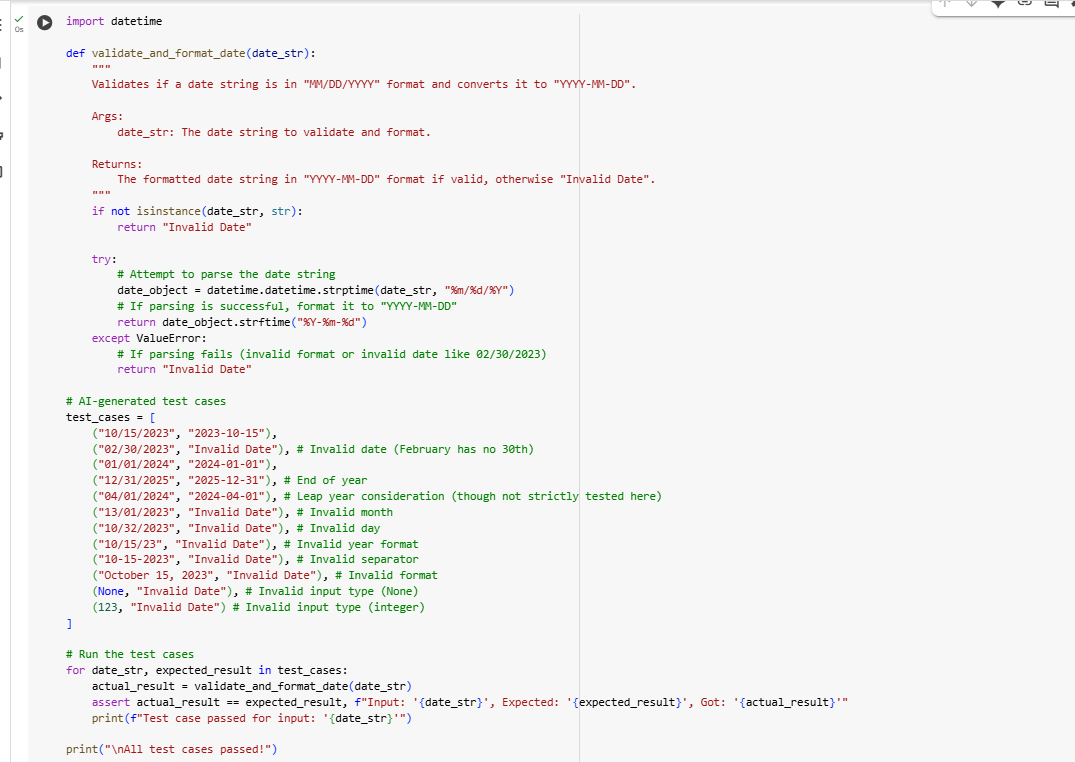
AI-generated content may be incorrect.

Explanation:

1. **Define the Inventory Class**: The code starts by defining a class named Inventory. This class is designed to represent an inventory system.
2. **Initialize the Inventory (\_\_init\_\_ method)**:
   * The \_\_init\_\_(self) method is the constructor of the class. It's called when you create a new Inventory object.
   * self.stock = {} initializes an empty dictionary named stock as an attribute of the Inventory object. This dictionary will store the items in the inventory, with item names as keys and their quantities as values.
3. **Define the add\_item method**:
   * This method, add\_item(self, name, quantity), is used to add items to the inventory or increase the quantity of existing items.
   * self refers to the instance of the Inventory class.
   * name is the name of the item (a string).
   * quantity is the amount to add (an integer).
   * It includes input validation to check if name is a string, quantity is an integer, and quantity is positive. If the input is invalid, it prints an error message and returns.
   * If the name is already a key in the self.stock dictionary, it increases the existing quantity: self.stock[name] += quantity.
   * If the name is not in the dictionary, it adds the item with the given quantity: self.stock[name] = quantity.
4. **Define the remove\_item method**:
   * This method, remove\_item(self, name, quantity), is used to remove items from the inventory or decrease their quantity.
   * name is the name of the item.
   * quantity is the amount to remove.
   * It also includes input validation similar to add\_item.
   * It checks if the name is in the self.stock dictionary.
   * If the item exists, it checks if there is enough stock (self.stock[name] >= quantity).
   * If there is enough stock, it decreases the quantity: self.stock[name] -= quantity.
   * If the stock becomes zero after removal, it removes the item's key from the dictionary using del self.stock[name].
   * If there is not enough stock, it prints a message indicating the available stock.
   * If the item is not found in the inventory, it prints a "not found" message.
5. **Define the get\_stock method**:
   * This method, get\_stock(self, name), is used to get the current quantity of a specific item.
   * name is the name of the item.
   * It checks if the name is a string. If not, it returns 0.
   * return self.stock.get(name, 0) uses the dictionary's get() method. It attempts to retrieve the value associated with name. If name is found, it returns the quantity. If name is not found, it returns the default value specified, which is 0 in this case.
6. **Create an Inventory object**: inv = Inventory() creates an instance of the Inventory class and assigns it to the variable inv.
7. **Run AI-generated test cases**: The code then proceeds to run a series of test cases using assert statements to verify the functionality of the Inventory methods.
   * Each test case calls one or more methods (add\_item, remove\_item, get\_stock).
   * An assert statement checks if the actual result of a method call matches the expected result. If the assertion fails, it means there's an issue with the code, and an AssertionError is raised with a descriptive message.
   * print() statements are used after each successful assertion to indicate which test case passed.
   * These test cases cover various scenarios, including adding items, removing items, getting stock, attempting to remove more than available, attempting to remove non-existent items, removing all stock of an item, and handling invalid input types or quantities.
8. **Print completion message**: Finally, print("\nInventory class and tests executed.") is printed after all the test cases have been run (assuming no assertions failed), indicating the completion of the script.

Task – 5:

Code:



Output:

A screenshot of a computer program

AI-generated content may be incorrect.

Explanation:

1. **Import the datetime module**: The line import datetime imports Python's built-in datetime module, which provides classes for working with dates and times.
2. **Define the validate\_and\_format\_date function**: This defines a function named validate\_and\_format\_date that takes one argument, date\_str, which is the input string you want to validate and format as a date.
3. **Check if the input is a string**: The first if statement, if not isinstance(date\_str, str):, checks if the input date\_str is actually a string. If it's not a string (e.g., it's None or a number), the function immediately returns the string "Invalid Date".
4. **Use a try-except block for date parsing**: The code uses a try...except ValueError: block to handle potential errors that might occur when trying to interpret the input string as a date.
5. **Attempt to parse the date string**: Inside the try block, date\_object = datetime.datetime.strptime(date\_str, "%m/%d/%Y") attempts to parse the date\_str.
   * datetime.datetime.strptime() is a function that parses a string into a datetime object.
   * date\_str is the input string to be parsed.
   * "%m/%d/%YYYY" is the format code that tells strptime to expect the date string in "Month/Day/Year" format (e.g., "10/15/2023"). If the input string does not match this exact format, a ValueError will be raised.
6. **Format the date if parsing is successful**: If the strptime call in the try block is successful (meaning the date\_str was in the correct "MM/DD/YYYY" format and represented a valid date), the code proceeds to return date\_object.strftime("%Y-%m-%d").
   * date\_object is the datetime object created from the parsed string.
   * .strftime("%Y-%m-%d") is a method of the datetime object that formats the date into a string according to the specified format code. "%Y-%m-%d" formats it as "Year-Month-Day" (e.g., "2023-10-15").
7. **Handle invalid date formats or values**: If the strptime call in the try block fails for any reason (e.g., the format is wrong like "10-15-2023", or the date itself is invalid like "02/30/2023"), a ValueError is raised.
   * The except ValueError: block catches this specific error.
   * Inside the except block, return "Invalid Date" returns the string "Invalid Date", indicating that the input string could not be validated and formatted as a date.
8. **Define AI-generated test cases**: The test\_cases list contains several tuples. Each tuple represents a test case with an input date\_str and the expected\_result string. These test cases cover valid dates, invalid dates (due to incorrect day, month, or format), and invalid input types (None and an integer) to check the function's robustness.
9. **Run the test cases**: The for loop iterates through each tuple in the test\_cases list.
10. **Call the function and assert the result**: Inside the loop:
    * actual\_result = validate\_and\_format\_date(date\_str) calls the validate\_and\_format\_date function with the current input string and stores the returned value.
    * assert actual\_result == expected\_result, ... checks if the actual\_result matches the expected\_result for that test case. If they don't match, an AssertionError is raised with a message detailing the input, expected result, and actual result.
11. **Print test case status**: print(f"Test case passed for input: '{date\_str}'") is executed if the assert statement passes, confirming the test case was successful.
12. **Print final success message**: print("\nAll test cases passed!") is printed after the loop finishes, confirming that all the provided test cases passed without any assertions failing.

\*\*\*\*\*\*