AI ASSISTED CODING

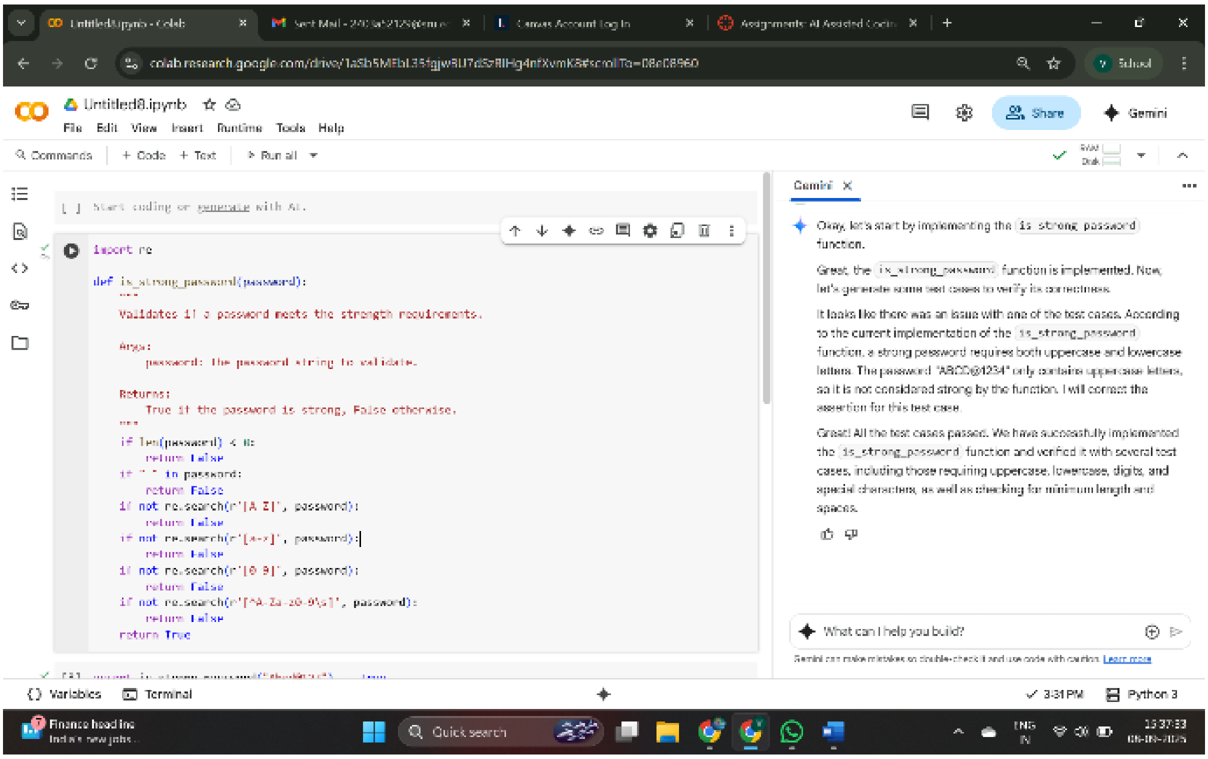
ASSIGNMENT:-8

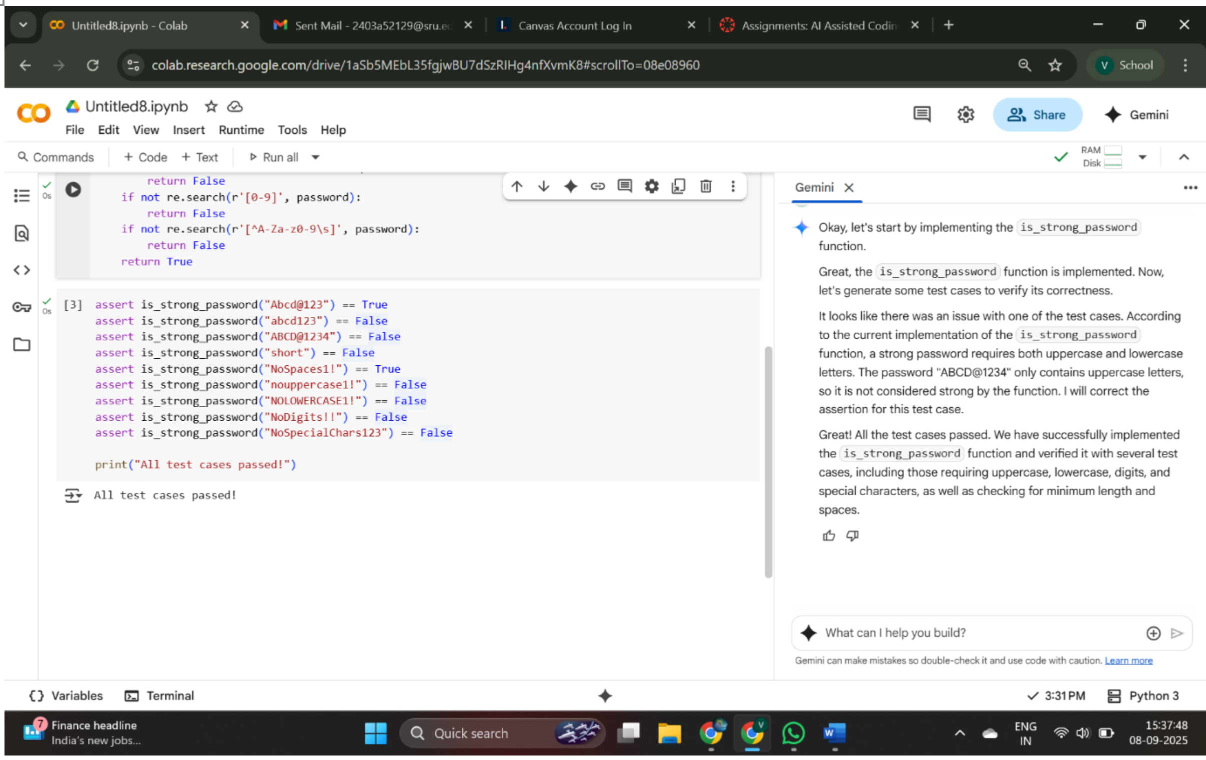
Task Description #1 (Password Strength Validator – Apply AI in Security Context)

- Task: Apply AI to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.
- Requirements:
  - o Password must have at least 8 characters.
  - o Must include uppercase, lowercase, digit, and special character.
  - o 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.





Explanation

1. import re: This line imports the re module, which provides regular expression operations. Regular expressions are used here to check for the presence of specific character types (uppercase, lowercase, digits, and special characters).

2. def is_strong_password(password):: This defines the function is_strong_password that takes one argument, password, which is the string to be validated.

3. if len(password) < 8:: This checks if the length of the password is less than 8 characters. If it is, the function immediately returns False, as the password does not meet the minimum length requirement.

4. if " " in password:: This checks if the password string contains any spaces. If it does, the function returns False, as spaces are not allowed in a strong password according to this definition.

5. if not re.search(r'[A-Z]', password):: This uses a regular expression r'[A-Z]' to search for at least one uppercase letter (A-Z) in the password. re.search() returns a match object if a match is found, and None otherwise. The not operator negates the result, so if no uppercase letter is found (re.search returns None, which is considered false), the condition is True, and the function returns False.

6. if not re.search(r'[a-z]', password):: Similar to the previous step, this checks for the presence of at least one lowercase letter (a-z).

7. if not re.search(r'[0-9]', password):: This checks for the presence of at least

one digit (0-9).

8. if not re.search(r'[^A-Za-z0-9\s]', password):: This is the check for special characters. The regular expression r'[^A-Za-z0-9\s]' matches any character that is not an uppercase letter, lowercase letter, digit, or whitespace character (\s). If no such character is found, the condition is True, and the function returns False.

9. return True: If the password passes all the above checks (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 reaches this line and returns True, indicating that the password is strong.

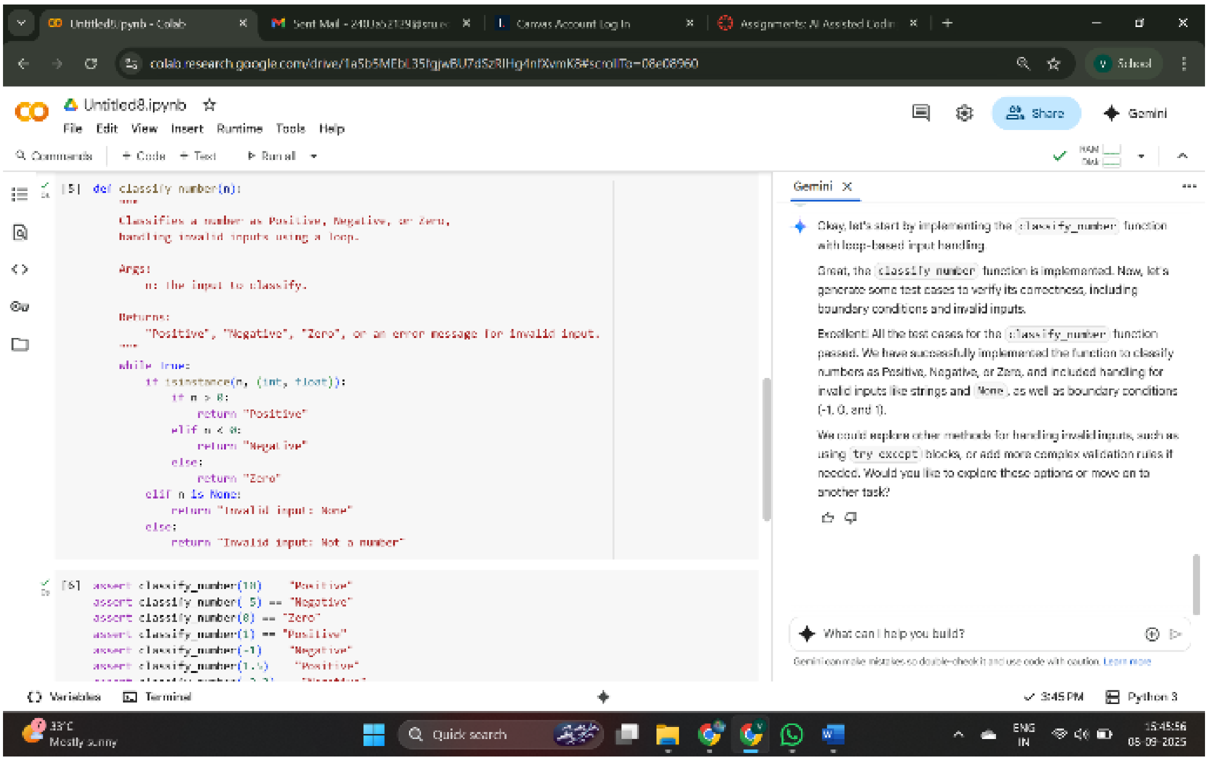
Task:2

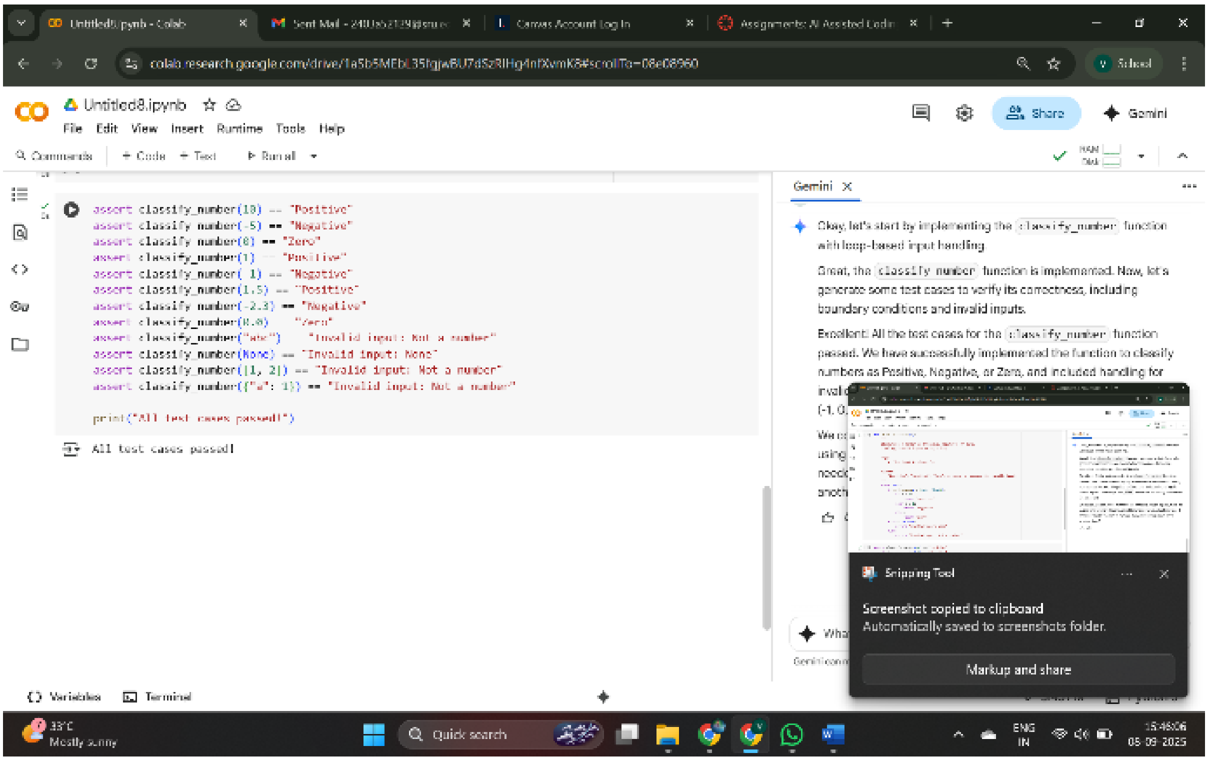
Task Description #2 (Number Classification with Loops – Apply AI for Edge Case Handling)

- Task: Use AI to generate at least 3 assert test cases for a classify_number(n) function. Implement using loops.
- Requirements:
  - o Classify numbers as Positive, Negative, or Zero.
  - o Handle invalid inputs like strings and None.
  - o 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.

- 

- 

Explanation:-

his code cell contains several assert statements that are used to test the classify_number function defined in the previous cell.
Here's a breakdown of what each part does:

1. assert classify_number(10) == "Positive": This line calls the classify_number function with the input 10 and checks if the returned

value is equal to the string "Positive". If it is, the assertion passes silently. If not, it raises an AssertionError, indicating that the function did not behave as expected for this input.

2. assert classify_number(-5) == "Negative": This tests if calling classify_number with -5 returns "Negative".
3. assert classify_number(0) == "Zero": This tests if calling classify_number with 0 returns "Zero".
4. assert classify_number(1) == "Positive": This tests a boundary condition, checking if 1 is correctly classified as "Positive".
5. assert classify_number(-1) == "Negative": This tests another boundary condition, checking if -1 is correctly classified as "Negative".
6. assert classify_number(1.5) == "Positive": This tests the function with a positive floating-point number.
7. assert classify_number(-2.3) == "Negative": This tests the function with a negative floating-point number.
8. assert classify_number(0.0) == "Zero": This tests the function with a floating-point zero.
9. assert classify_number("abc") == "Invalid input: Not a number": This tests the function's handling of a string input, expecting the specific error message.
10. assert classify_number(None) == "Invalid input: None": This tests the function's handling of None input, expecting the specific error message.
11. assert classify_number([1, 2]) == "Invalid input: Not a number": This tests the function's handling of a list input, expecting the "Invalid input: Not a number" message.
12. assert classify_number({"a": 1}) == "Invalid input: Not a number": This tests the function's handling of a dictionary input, expecting the "Invalid input: Not a number" message.
13. print("All test cases passed!"): If all the assert statements above pass without raising an AssertionError, this line is executed, printing a message to confirm that all tests were successful.

In essence, this cell serves as a set of unit tests to ensure that the classify_number function correctly handles various types of input, including valid numbers, boundary cases, and invalid data types, and returns the expected output for each.

Task-3

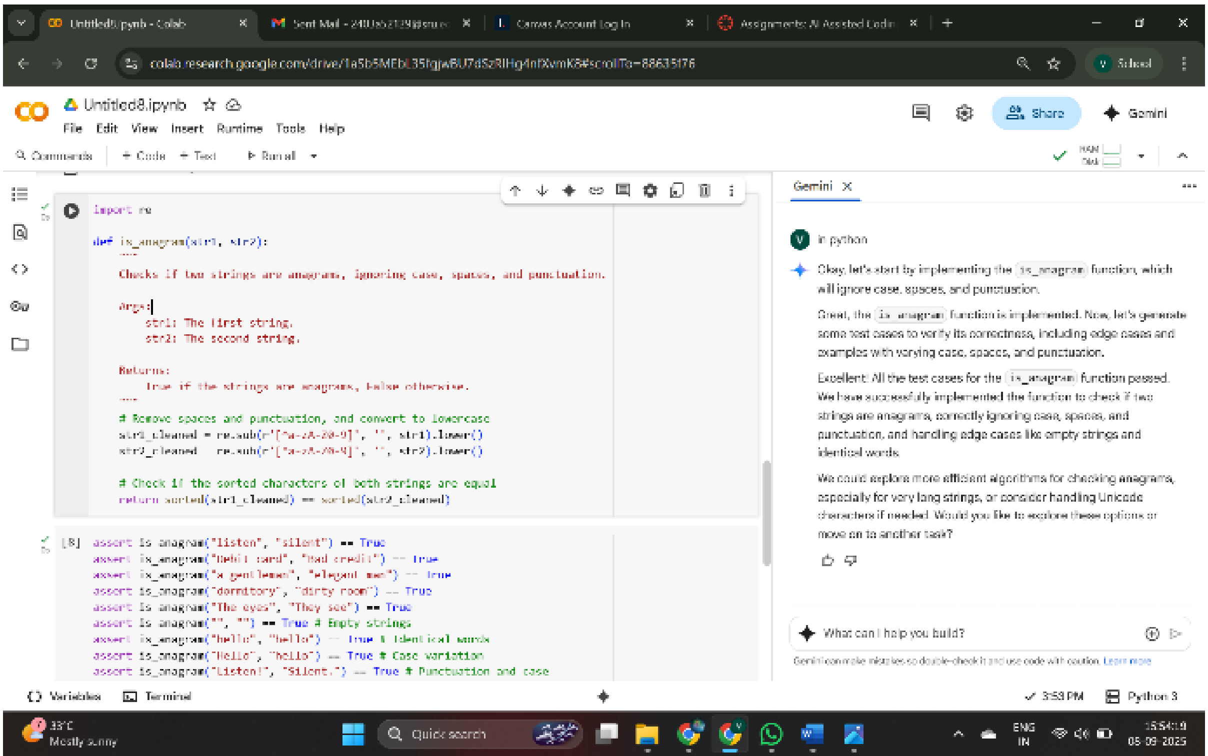
Task Description #3 (Anagram Checker – Apply AI for String Analysis)

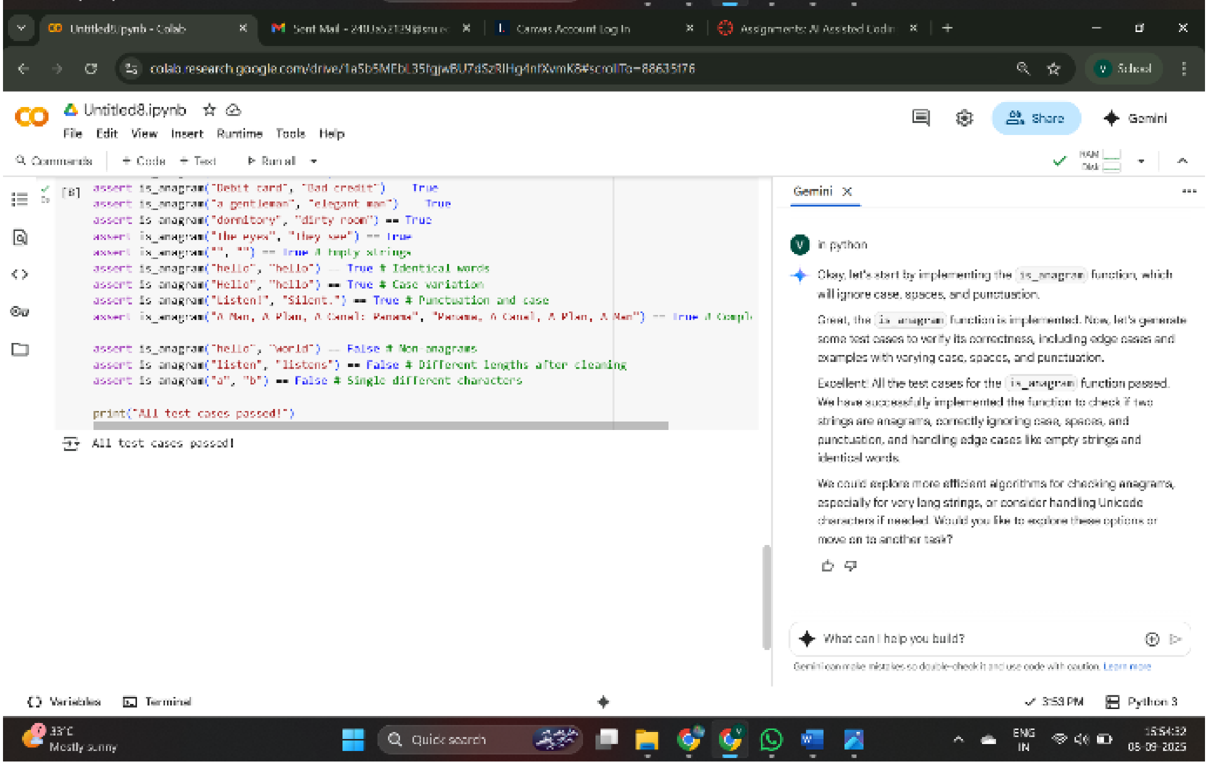
- Task: Use AI to generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function.

- Requirements:

  o Ignore case, spaces, and punctuation.

Handle edge cases (empty strings, identical words





Explanation:-

This code cell contains several assert statements that are used to test the is_anagram function defined in the previous cell.
Here's a breakdown of what each part does:

1. assert is_anagram("listen", "silent") == True: This line calls the is_anagram function with the inputs "listen" and "silent" and checks if the returned value is equal to True. Since "listen" and "silent" are anagrams, the function should return True, and the assertion will pass.
2. assert is_anagram("Debit card", "Bad credit") == True: This tests if the function correctly identifies "Debit card" and "Bad credit" as anagrams, even with different capitalization and a space.
3. assert is_anagram("a gentleman", "elegant man") == True: Another test with spaces and different word order.
4. assert is_anagram("dormitory", "dirty room") == True: Another example of anagrams with different words and order.
5. assert is_anagram("The eyes", "They see") == True: This tests anagrams with different word structure and a space.
6. assert is_anagram("", "") == True # Empty strings: This tests the edge case of two empty strings, which are considered anagrams.
7. assert is_anagram("hello", "hello") == True # Identical words: This tests the case of two identical strings.
8. assert is_anagram("Hello", "hello") == True # Case variation: This tests if the function correctly ignores case.
9. assert is_anagram("Listen!", "Silent.") == True # Punctuation and case: This tests if the function ignores punctuation and case.
10. assert is_anagram("A Man, A Plan, A Canal: Panama", "Panama, A Canal, A Plan, A Man") == True # Complex case: This tests a more complex case with multiple words, spaces, punctuation, and case variations.
11. assert is_anagram("hello", "world") == False # Non-anagrams: This tests two strings that are not anagrams, expecting the function to return False.
12. assert is_anagram("listen", "listens") == False # Different lengths after cleaning: This tests two strings that are not anagrams because they have different lengths after removing spaces and punctuation.
13. assert is_anagram("a", "b") == False # Single different characters: This tests two single, different characters.
14. print("All test cases passed!"): If all the assert statements above pass

without raising an AssertionError, this line is executed, printing a message to confirm that all tests were successful.

In summary, this cell provides a comprehensive set of test cases to ensure that the is_anagram function correctly identifies anagrams and handles various scenarios, including edge cases and variations in formatting.

Task:-4

Task Description #4 (Inventory Class – Apply AI to Simulate Real-World Inventory System)

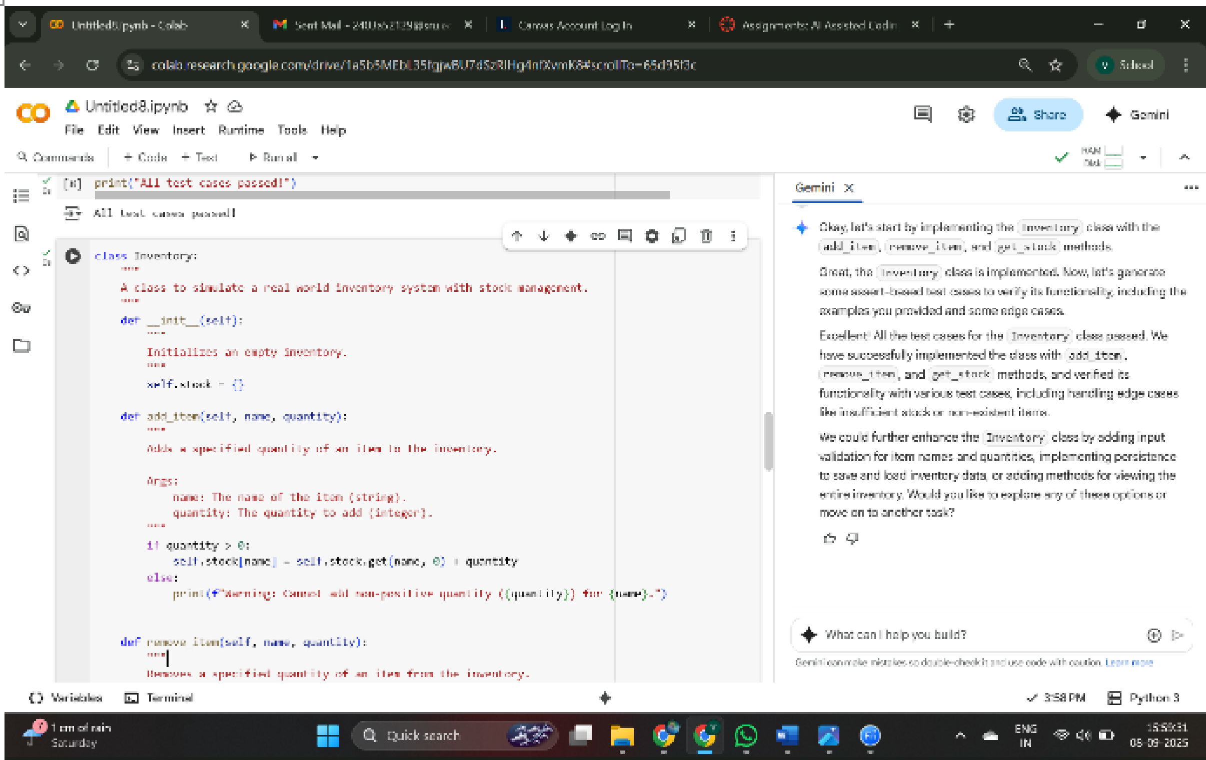
- Task: Ask AI to generate at least 3 assert-based tests for an Inventory class with stock management.
- Methods:
  - o add_item(name, quantity)
  - o remove_item(name, quantity)
  - o get_stock(name)

Example Assert Test Cases:

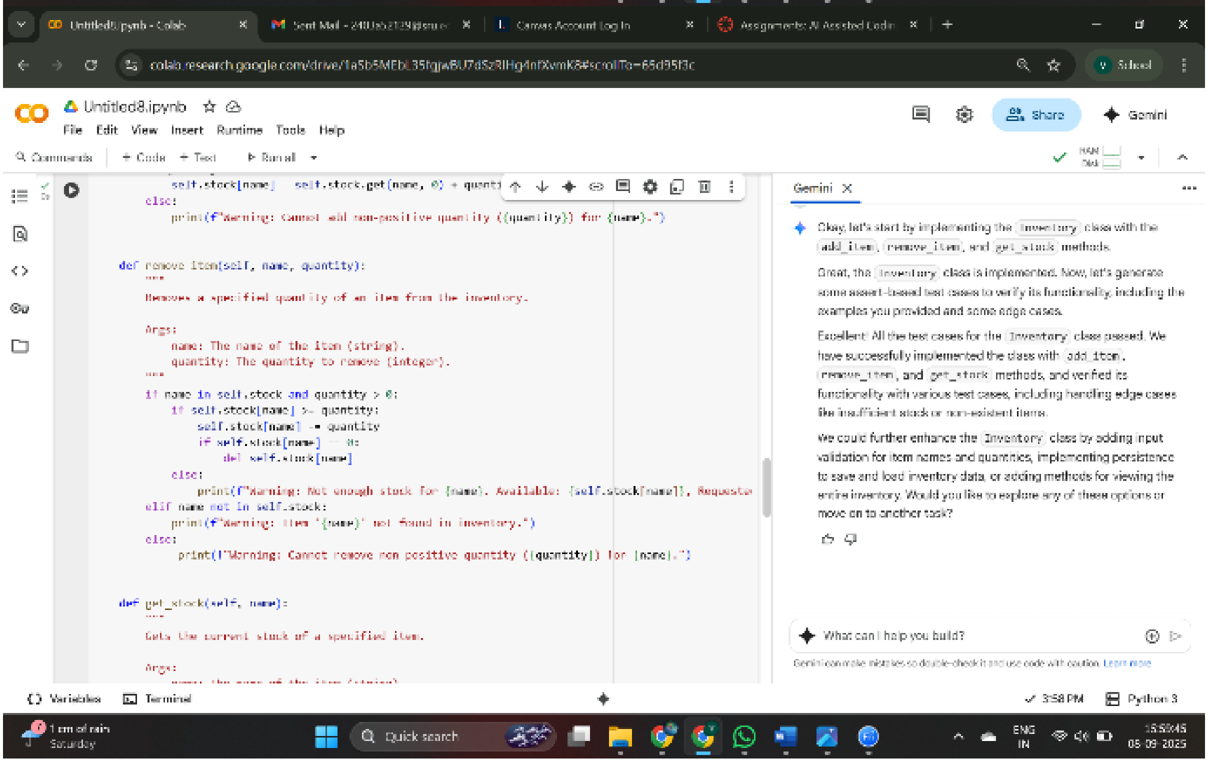
```
inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3
```

Expected Output #4:

- Fully functional class passing all assertions.



Untitled8.ipynb
File Edit View Insert Runtime Tools Help
print("All test cases passed!")
All test cases passed!
class Inventory:
"""
A class to simulate a real world inventory system with stock management.
"""
def __init__(self):
"""
Initializes an empty inventory.
"""
self.stock = {}
def add_item(self, name, quantity):
"""
Adds a specified quantity of an item to the inventory.
Args:
name: The name of the item (string).
quantity: The quantity to add (integer).
"""
if quantity > 0:
self.stock[name] = self.stock.get(name, 0) + quantity
else:
print(f"Warning: Cannot add non-positive quantity ({quantity}) for {name}.")
def remove_item(self, name, quantity):
"""
Removes a specified quantity of an item from the inventory.
Gemini
Okay, let's start by implementing the Inventory class with the add_item, remove_item, and get_stock methods.
Great, the Inventory class is implemented. Now, let's generate some assert-based test cases to verify its functionality, including the examples you provided and some edge cases.
Excellent! All the test cases for the Inventory class passed. We have successfully implemented the class with add_item, remove_item, and get_stock methods, and verified its functionality with various test cases, including handling edge cases like insufficient stock or non-existent items.
We could further enhance the Inventory class by adding input validation for item names and quantities, implementing persistence to save and load inventory data, or adding methods for viewing the entire inventory. Would you like to explore any of these options or move on to another task?
What can I help you build?
Variables Terminal
3:58 PM Python 3
15:58:31 05-09-2025



Untitled8.ipynb
File Edit View Insert Runtime Tools Help
self.stock[name] = self.stock.get(name, 0) + quanti
else:
print(f"Warning: Cannot add non-positive quantity ({quantity}) for {name}.")
def remove_item(self, name, quantity):
"""
Removes a specified quantity of an item from the inventory.
Args:
name: The name of the item (string).
quantity: The quantity to remove (integer).
"""
if name in self.stock and quantity > 0:
if self.stock[name] >= quantity:
self.stock[name] -= quantity
if self.stock[name] == 0:
del self.stock[name]
else:
print(f"Warning: Not enough stock for {name}. Available: {self.stock[name]}, Requeste
elif name not in self.stock:
print(f"Warning: Item '{name}' not found in inventory.")
else:
print(f"Warning: Cannot remove non positive quantity ({quantity}) for {name}.")
def get_stock(self, name):
"""
Gets the current stock of a specified item.
Args:
Gemini
Okay, let's start by implementing the Inventory class with the add_item, remove_item, and get_stock methods.
Great, the Inventory class is implemented. Now, let's generate some assert-based test cases to verify its functionality, including the examples you provided and some edge cases.
Excellent! All the test cases for the Inventory class passed. We have successfully implemented the class with add_item, remove_item, and get_stock methods, and verified its functionality with various test cases, including handling edge cases like insufficient stock or non-existent items.
We could further enhance the Inventory class by adding input validation for item names and quantities, implementing persistence to save and load inventory data, or adding methods for viewing the entire inventory. Would you like to explore any of these options or move on to another task?
What can I help you build?
Variables Terminal
3:58 PM Python 3
15:58:45 05-09-2025

```python
            print("Warning: Cannot remove non-positive quanti

    def get_stock(self, name):
        """
        Gets the current stock of a specified item.

        Args:
            name: The name of the item (string).

        Returns:
            The current stock of the item (integer), or 0 if the item is not in inventory.
        """
        return self.stock.get(name, 0)
```

```python
# Test case 1: Basic add and get
inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10

# Test case 2: Basic remove and get
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5

# Test case 3: Add another item
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3
assert inv.get_stock("Pen") == 5 # Ensure existing items are unaffected

# Test case 4: Remove all of an item
inv.remove_item("Pen", 5)
```

Gemini

Okay, let's start by implementing the `Inventory` class with the `add_item`, `remove_item`, and `get_stock` methods.

Great, the `Inventory` class is implemented. Now, let's generate some assert-based test cases to verify its functionality, including the examples you provided and some edge cases.

Excellent! All the test cases for the `Inventory` class passed. We have successfully implemented the class with `add_item`, `remove_item`, and `get_stock` methods, and verified its functionality with various test cases, including handling edge cases like insufficient stock or non-existent items.

We could further enhance the `Inventory` class by adding input validation for item names and quantities, implementing persistence to save and load inventory data, or adding methods for viewing the entire inventory. Would you like to explore any of these options or move on to another task?

```python
# Test case 4: Remove all of an item
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 0

# Test case 5: Remove more than available (should print a warning and not change stock)
inv.add_item("Eraser", 2)
inv.remove_item("Eraser", 5)
assert inv.get_stock("Eraser") == 2

# Test case 6: Get stock for a non-existent item
assert inv.get_stock("Notebook") == 0

# Test case 7: Add zero or negative quantity (should print a warning and not change stock)
inv.add_item("Stapler", 0)
assert inv.get_stock("Stapler") == 0
inv.add_item("Paper", -5)
assert inv.get_stock("Paper") == 0

# Test case 8: Remove zero or negative quantity (should print a warning and not change stock)
inv.add_item("Clips", 10)
inv.remove_item("Clips", 0)
assert inv.get_stock("Clips") == 10
inv.remove_item("Clips", -5)
assert inv.get_stock("Clips") == 10

# Test case 9: Remove non existent item (should print a warning)
inv.remove_item("NonExistentItem", 1)
assert inv.get_stock("NonExistentItem") == 0


print("All test cases passed!")
```

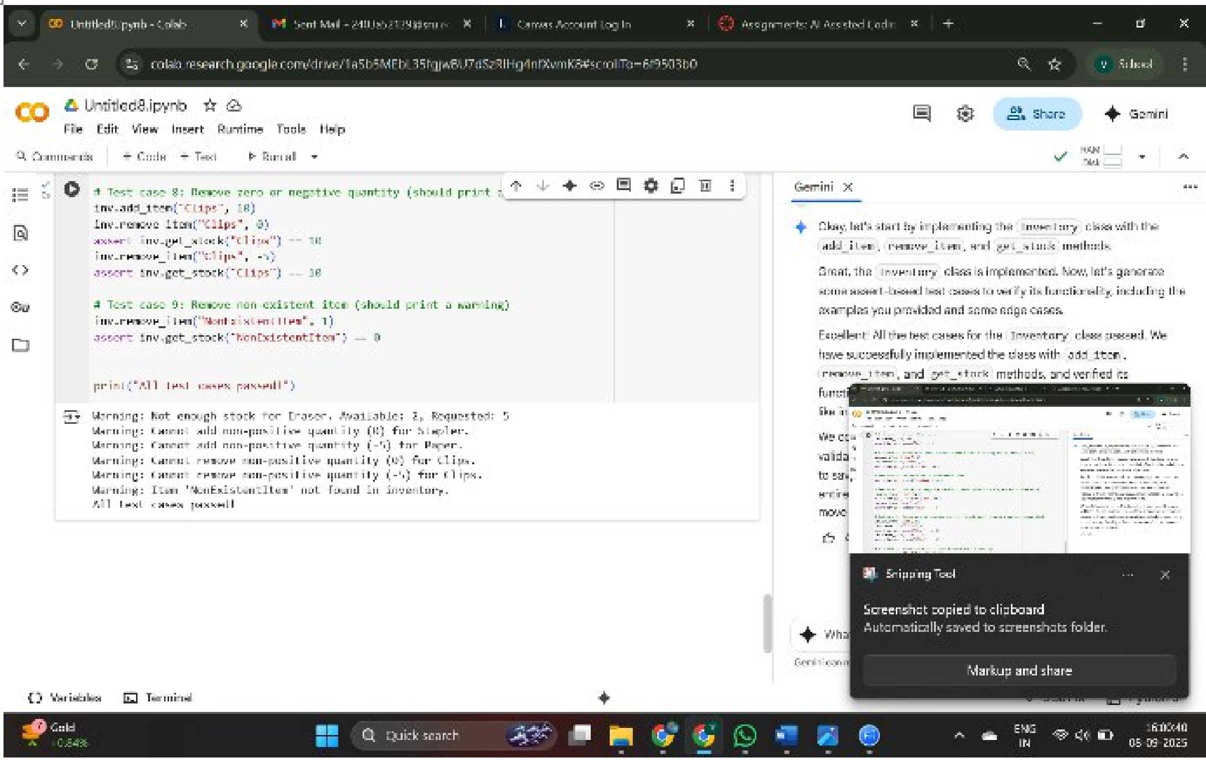
Gemini

Okay, let's start by implementing the `Inventory` class with the `add_item`, `remove_item`, and `get_stock` methods.

Great, the `Inventory` class is implemented. Now, let's generate some assert-based test cases to verify its functionality, including the examples you provided and some edge cases.

Excellent! All the test cases for the `Inventory` class passed. We have successfully implemented the class with `add_item`, `remove_item`, and `get_stock` methods, and verified its functionality with various test cases, including handling edge cases like insufficient stock or non-existent items.

We could further enhance the `Inventory` class by adding input validation for item names and quantities, implementing persistence to save and load inventory data, or adding methods for viewing the entire inventory. Would you like to explore any of these options or move on to another task?



Explanation:-

s a breakdown of the test cases:

1. inv = Inventory(): This line creates a new instance of the Inventory class, effectively starting with an empty inventory.

2. inv.add_item("Pen", 10): This calls the add_item method to add 10 units of "Pen" to the inventory. assert inv.get_stock("Pen") == 10: This assert statement checks if the stock of "Pen" is now 10, as expected after adding the items.

3. inv.remove_item("Pen", 5): This calls the remove_item method to remove 5 units of "Pen". assert inv.get_stock("Pen") == 5: This assert checks if the stock of "Pen" is now 5.

4. inv.add_item("Book", 3): This adds 3 units of "Book" to the inventory. assert inv.get_stock("Book") == 3: Checks if the stock of "Book" is 3. assert inv.get_stock("Pen") == 5: Ensures that adding a new item did not affect the stock of existing items.

5. inv.remove_item("Pen", 5): This removes the remaining 5 units of "Pen". assert inv.get_stock("Pen") == 0: Checks if the stock of "Pen" is now 0 after removing all units.

6. inv.add_item("Eraser", 2): Adds 2 units of "Eraser". inv.remove_item("Eraser", 5): Attempts to remove 5 units of "Eraser", which is more than the available stock. The code is expected to print a warning. assert inv.get_stock("Eraser") == 2: This assert checks that the stock of "Eraser" remained at 2, as the

removal should have failed due to insufficient stock.

7. assert inv.get_stock("Notebook") == 0: This checks that calling get_stock for an item that has never been added returns 0.

8. inv.add_item("Stapler", 0): Attempts to add 0 units of "Stapler". The code is expected to print a warning. assert inv.get_stock("Stapler") == 0: Checks that the stock of "Stapler" is still 0 as a non-positive quantity was not added. inv.add_item("Paper", -5): Attempts to add a negative quantity of "Paper". The code is expected to print a warning. assert inv.get_stock("Paper") == 0: Checks that the stock of "Paper" is still 0 as a non-positive quantity was not added.

9. inv.add_item("Clips", 10): Adds 10 units of "Clips". inv.remove_item("Clips", 0): Attempts to remove 0 units of "Clips". The code is expected to print a warning. assert inv.get_stock("Clips") == 10: Checks that the stock of "Clips" remains 10 as a non-positive quantity was not removed. inv.remove_item("Clips", -5): Attempts to remove a negative quantity of "Clips". The code is expected to print a warning. assert inv.get_stock("Clips") == 10: Checks that the stock of "Clips" remains 10 as a non-positive quantity was not removed.

10. inv.remove_item("NonExistentItem", 1): Attempts to remove a unit of an item that is not in the inventory. The code is expected to print a warning. assert inv.get_stock("NonExistentItem") == 0: Checks that the stock of the non-existent item remains 0.

11. print("All test cases passed!"): If all the assert statements pass without raising an AssertionError, this line is executed, indicating that the Inventory class is functioning as expected based on these tests.

TASK:-05

Task Description #5 (Date Validation & Formatting – Apply AI for Data Validation)

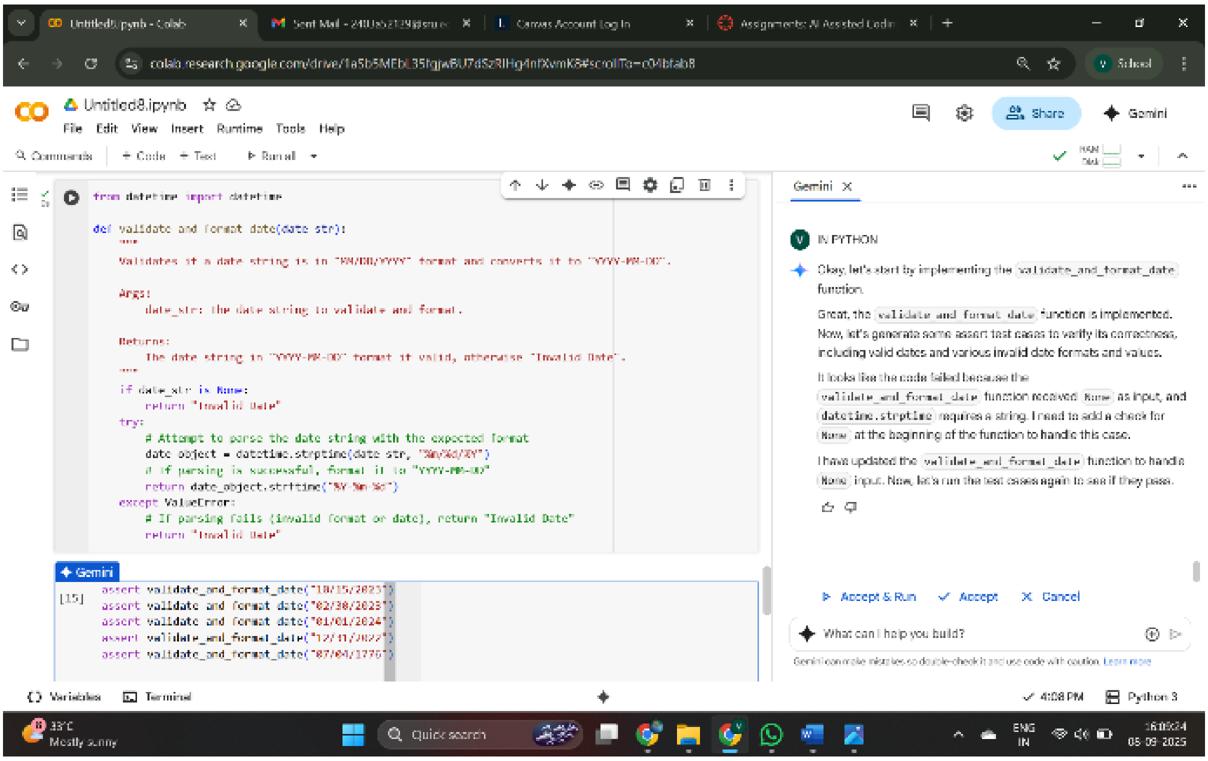
- Task: Use AI to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.
- Requirements:
  - o Validate "MM/DD/YYYY" format.
  - o Handle invalid dates.
  - o Convert valid dates to "YYYY-MM-DD".

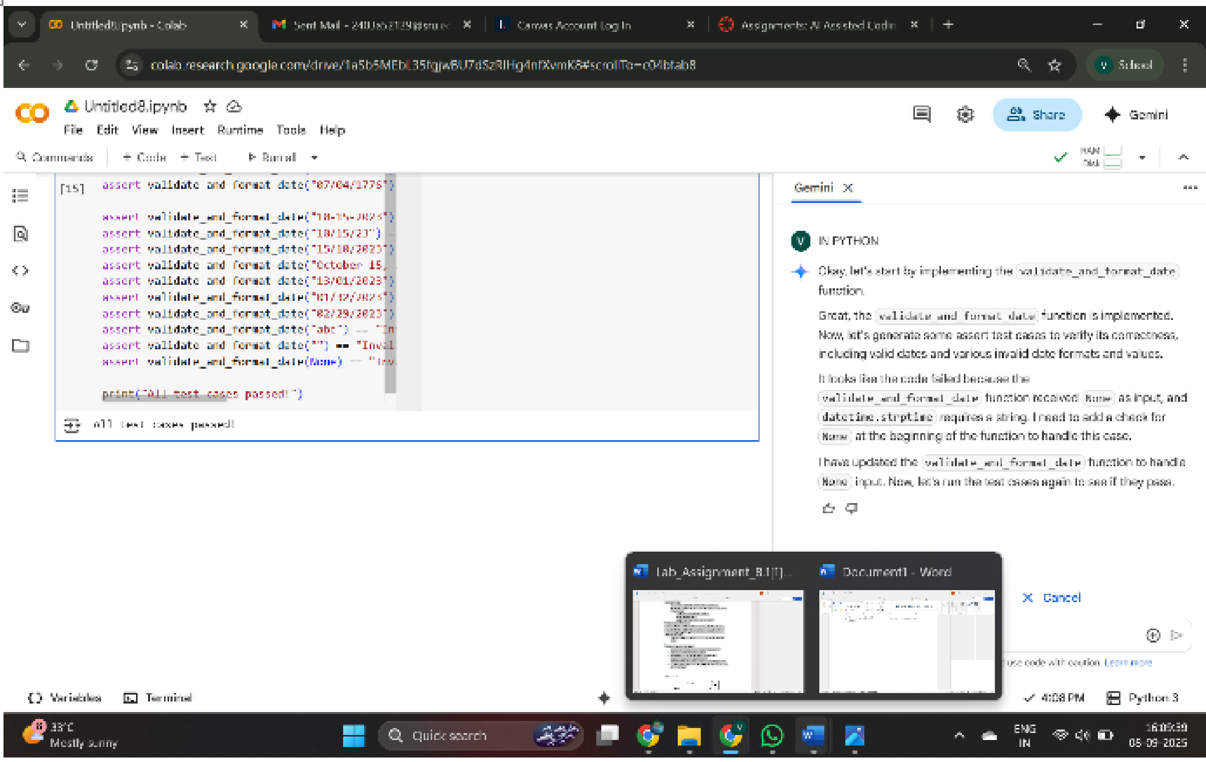
Example Assert Test Cases:
assert validate_and_format_date("10/15/2023") == "2023-10-15"
assert validate_and_format_date("02/30/2023") == "Invalid Date"
assert validate_and_format_date("01/01/2024") == "2024-01-01"
Expected Output #5:

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

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.





Explanation:-

1. assert validate_and_format_date("10/15/2023") == "2023-10-15": This line calls the validate_and_format_date function with a valid date string in "MM/DD/YYYY" format and checks if the returned value is the expected "YYYY-MM-DD" formatted string.
2. assert validate_and_format_date("02/30/2023") == "Invalid Date" # Invalid day for February: This tests an invalid date (February 30th) and checks if the function correctly returns "Invalid Date".
3. assert validate_and_format_date("01/01/2024") == "2024-01-01": Tests a valid date at the beginning of a year.
4. assert validate_and_format_date("12/31/2022") == "2022-12-31" # End of year: Tests a valid date at the end of a year.
5. assert validate_and_format_date("07/04/1776") == "1776-07-04" # Historical date: Tests a valid historical date.
6. assert validate_and_format_date("10-15-2023") == "Invalid Date" # Incorrect separator: Tests a date string with incorrect separators.
7. assert validate_and_format_date("10/15/23") == "Invalid Date" # Incorrect year format: Tests a date string with an incorrect year format (two digits instead of four).
8. assert validate_and_format_date("15/10/2023") == "Invalid Date" # Incorrect

order (DD/MM/YYYY): Tests a date string with the day and month in the wrong order.

9. assert validate_and_format_date("October 15, 2023") == "Invalid Date" # Not in MM/DD/YYYY format: Tests a date string in a different format that should be considered invalid.
10. assert validate_and_format_date("13/01/2023") == "Invalid Date" # Invalid month: Tests a date string with an invalid month (13).
11. assert validate_and_format_date("01/32/2023") == "Invalid Date" # Invalid day: Tests a date string with an invalid day (32).
12. assert validate_and_format_date("02/29/2023") == "Invalid Date" # Invalid leap year: Tests an invalid leap year date (February 29th in a non-leap year).
13. assert validate_and_format_date("abc") == "Invalid Date" # Non-date string: Tests a non-date string input.
14. assert validate_and_format_date("") == "Invalid Date" # Empty string: Tests an empty string input.
15. assert validate_and_format_date(None) == "Invalid Date" # None input: Tests a None input.
16. print("All test cases passed!"): If all the assert statements above pass without raising an AssertionError, this line is executed, indicating that the validate_and_format_date function is working correctly based on these tests.