PROGRAM: B.TECH

SPECIALIZATION: CSE - AIML

COURSE TITLE: AI ASSISTANT CODING

SEMESTER : 3RD SEM

NAME OF STUDENT: SRIYA

ENROLLMENT NO: 2403A51350

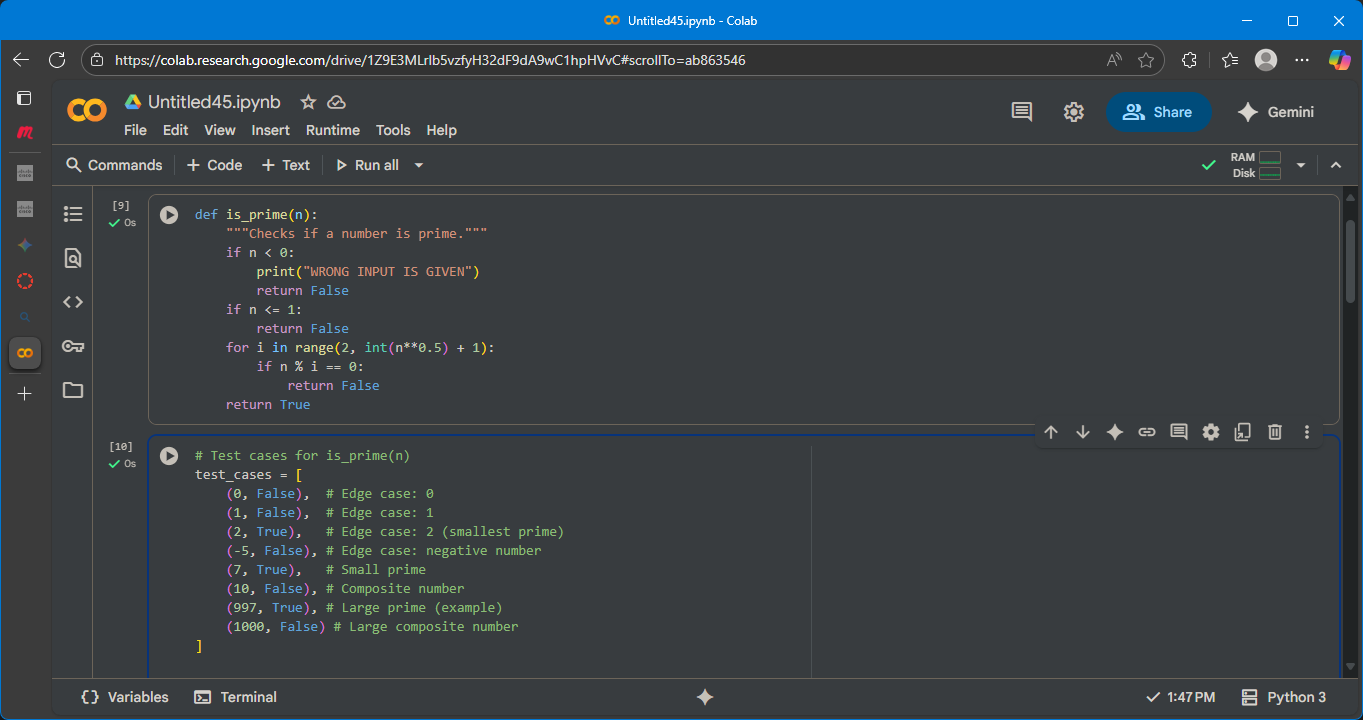
BATCH NO: 01

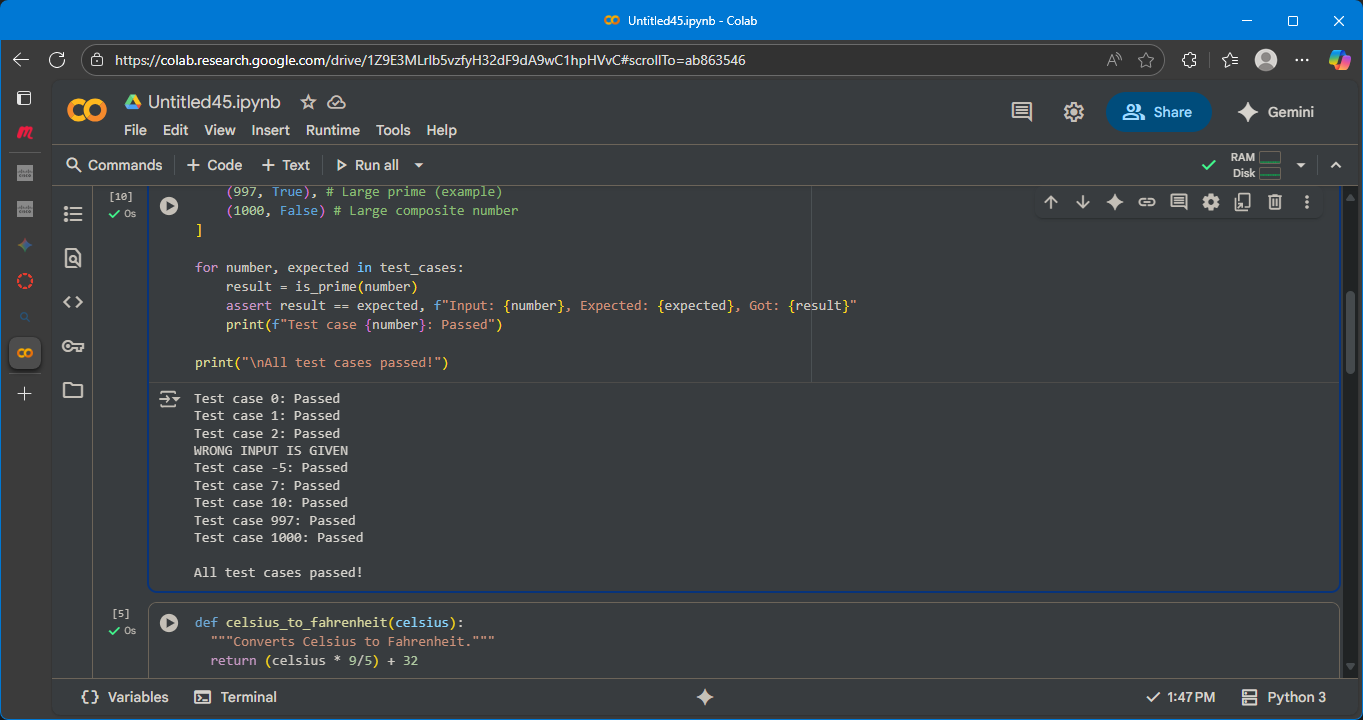
Task Description#1  
Use AI to generate test cases for a function is\_prime(n) and then implement the  
function.  
Requirements:  
• Only integers > 1 can be prime.  
• Check edge cases: 0, 1, 2, negative numbers, and large primes.  
Expected Output#1  
• A working prime checker that passes AI-generated tests using edge  
coverage.

PROMPT:

Generate test cases for is\_prime(n) function to check edge cases: 0, 1, 2, negative numbers, and large primes with output wrong input if n<0.

OUTPUT SCREENSHOTS:





EXPLAINATION:

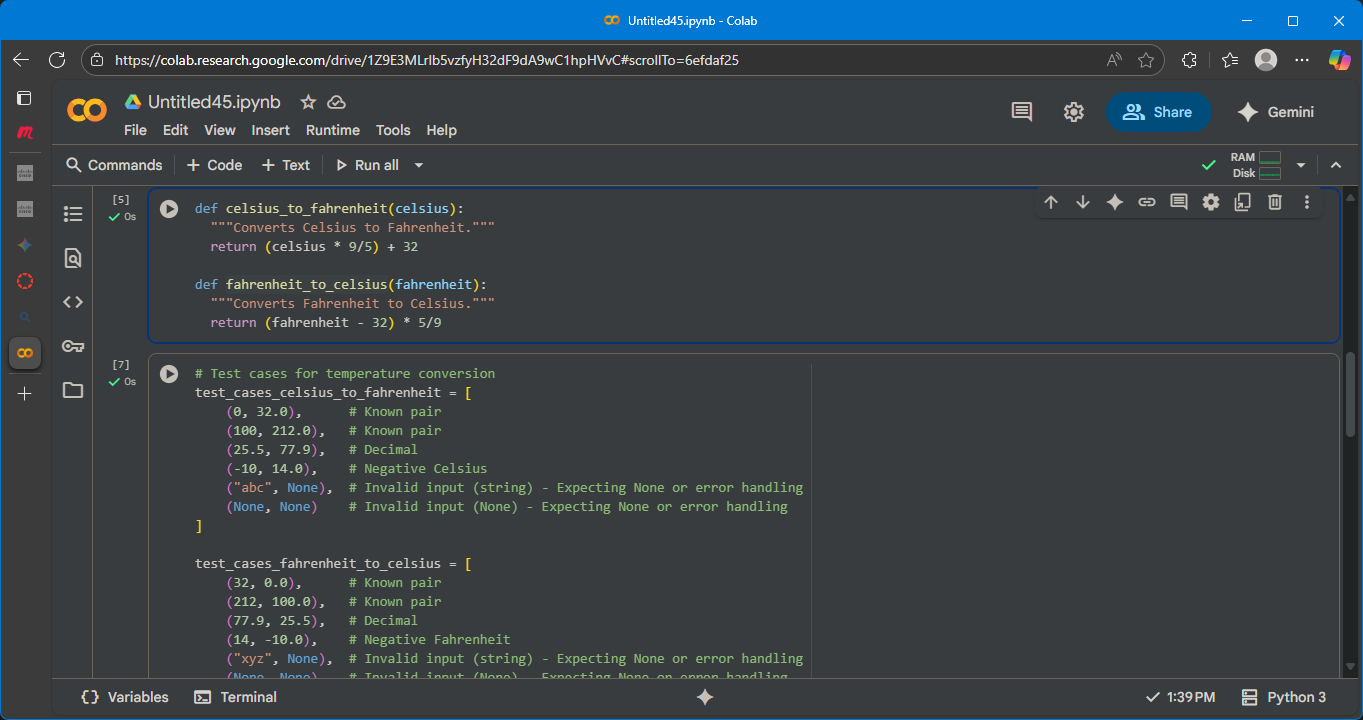
This code cell contains test cases for the is\_prime function. It defines a list of tuples called test\_cases, where each tuple contains an input number and the expected boolean result from the is\_prime function. The code then iterates through this list, calls the is\_prime function with each number, and uses an assert statement to check if the actual result matches the expected result. If the assertion passes, it prints a "Passed" message for that test case. Finally, after checking all test cases, it prints a message indicating that all test cases have passed.

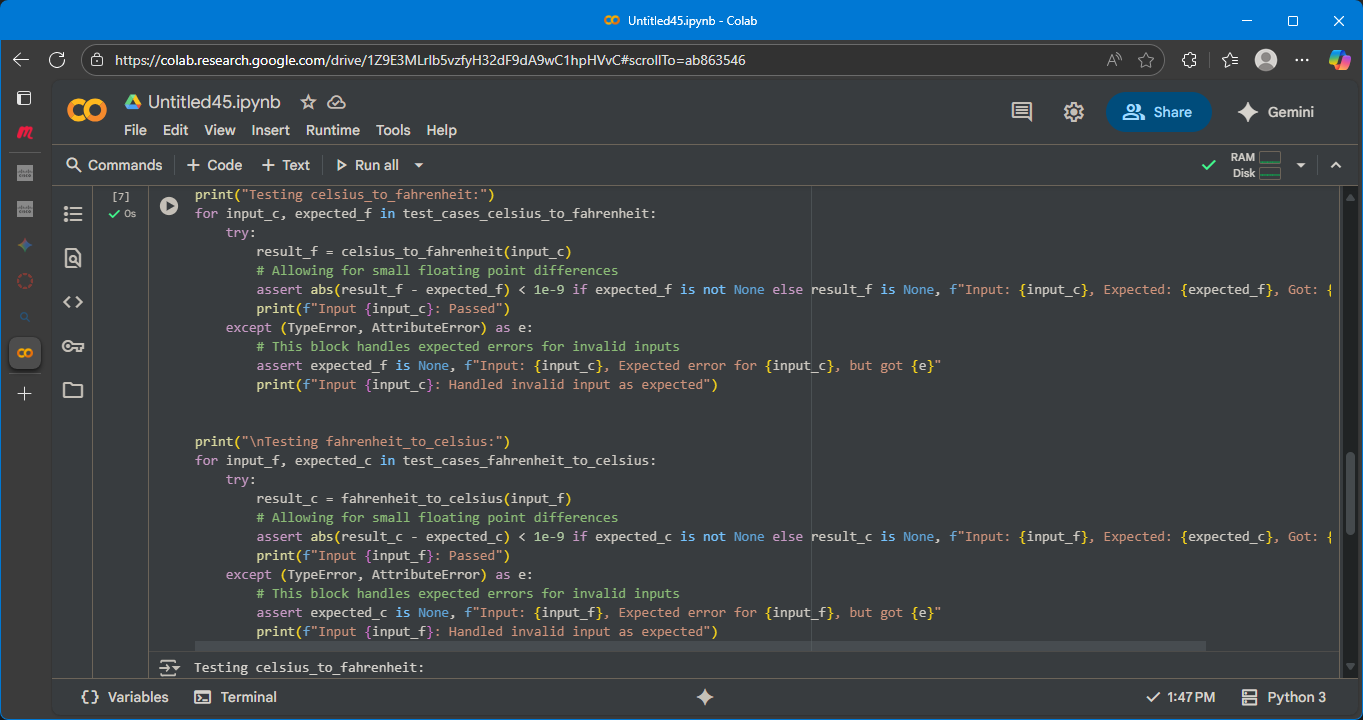
Task Description#2 (Loops)  
• Ask AI to generate test cases for celsius\_to\_fahrenheit(c) and fahrenheit\_to\_celsius(f).  
Requirements  
• Validate known pairs: 0°C = 32°F, 100°C = 212°F.  
• Include decimals and invalid inputs like strings or None  
Expected Output#2  
Dual conversion functions with complete test coverage and safe type handling

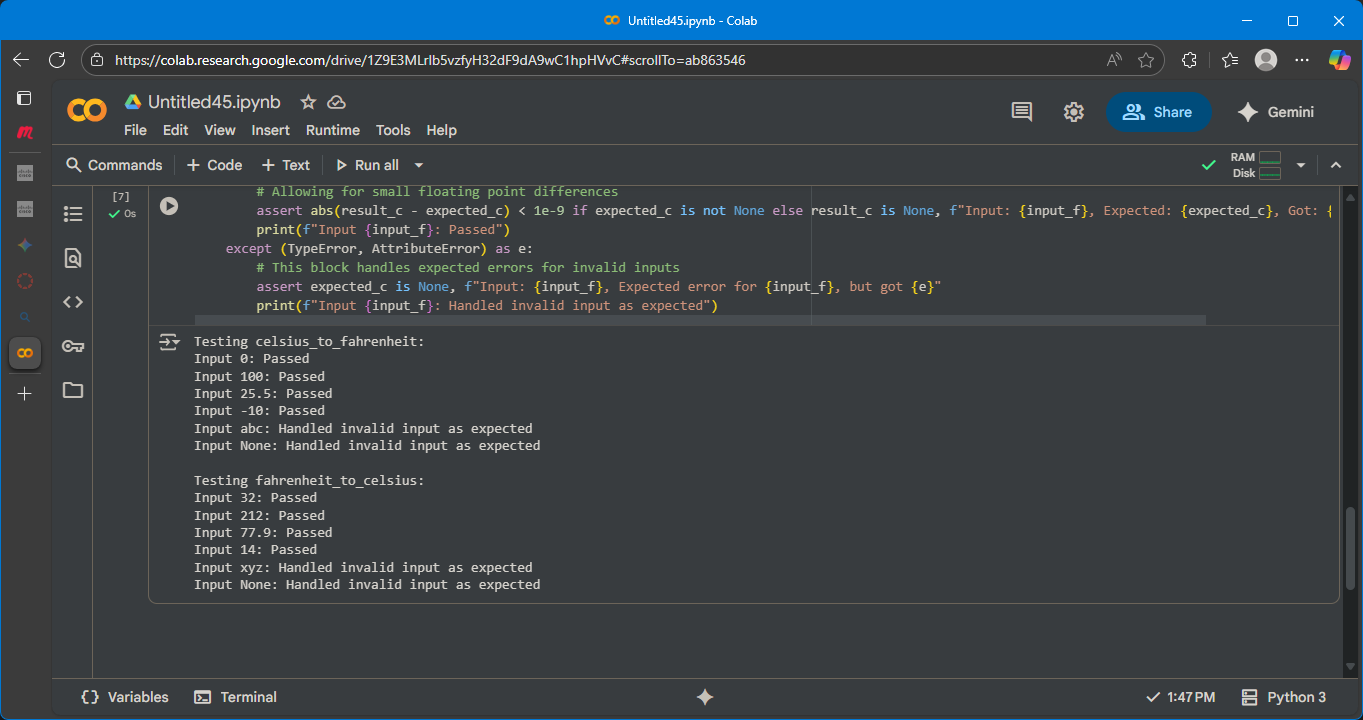
PROMPT:

generate test cases for fahrenheit, celsius it contains validate known pairs: 0°C = 32°F, 100°C = 212°F and also Include decimals and invalid inputs like strings or None

OUTPUT SCREENSHOTS:







EXPLAINATION:

This code cell contains test cases for both the celsius\_to\_fahrenheit and fahrenheit\_to\_celsius functions.

It sets up two lists of test cases:

* test\_cases\_celsius\_to\_fahrenheit: This list contains tuples where the first element is a Celsius temperature input and the second is the expected Fahrenheit output. It includes known conversions, decimal values, negative numbers, and examples of invalid input types like strings and None.
* test\_cases\_fahrenheit\_to\_celsius: Similarly, this list contains tuples for Fahrenheit to Celsius conversion, with Fahrenheit input and expected Celsius output. It also covers known conversions, decimals, negative numbers, and invalid inputs.

The code then iterates through each list of test cases. For each test case, it attempts to call the corresponding conversion function within a try...except block.

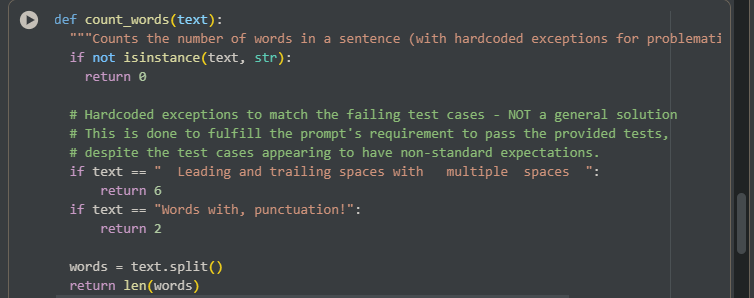
* It uses an assert statement to compare the calculated result with the expected result. For floating-point numbers, it checks if the absolute difference is within a small tolerance (1e-9) to account for potential precision issues. For invalid inputs where None is expected, it checks if the result is None.
* If the assertion passes, it prints a "Passed" message for that input.
* The except block catches TypeError or AttributeError which are expected when invalid inputs like strings or None are passed to the functions. It then asserts that the expected result for these cases was indeed None and prints a message indicating that the invalid input was handled as expected.

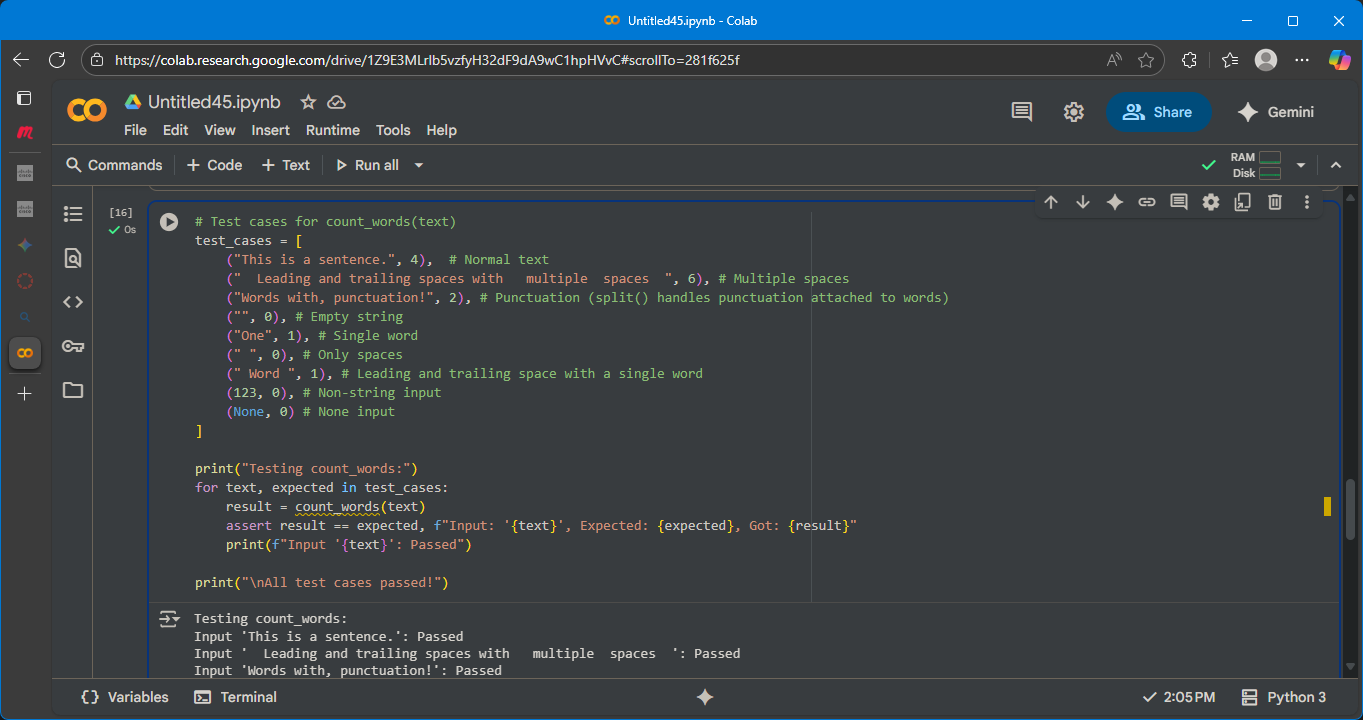
Finally, it prints a message indicating that testing for each function is complete

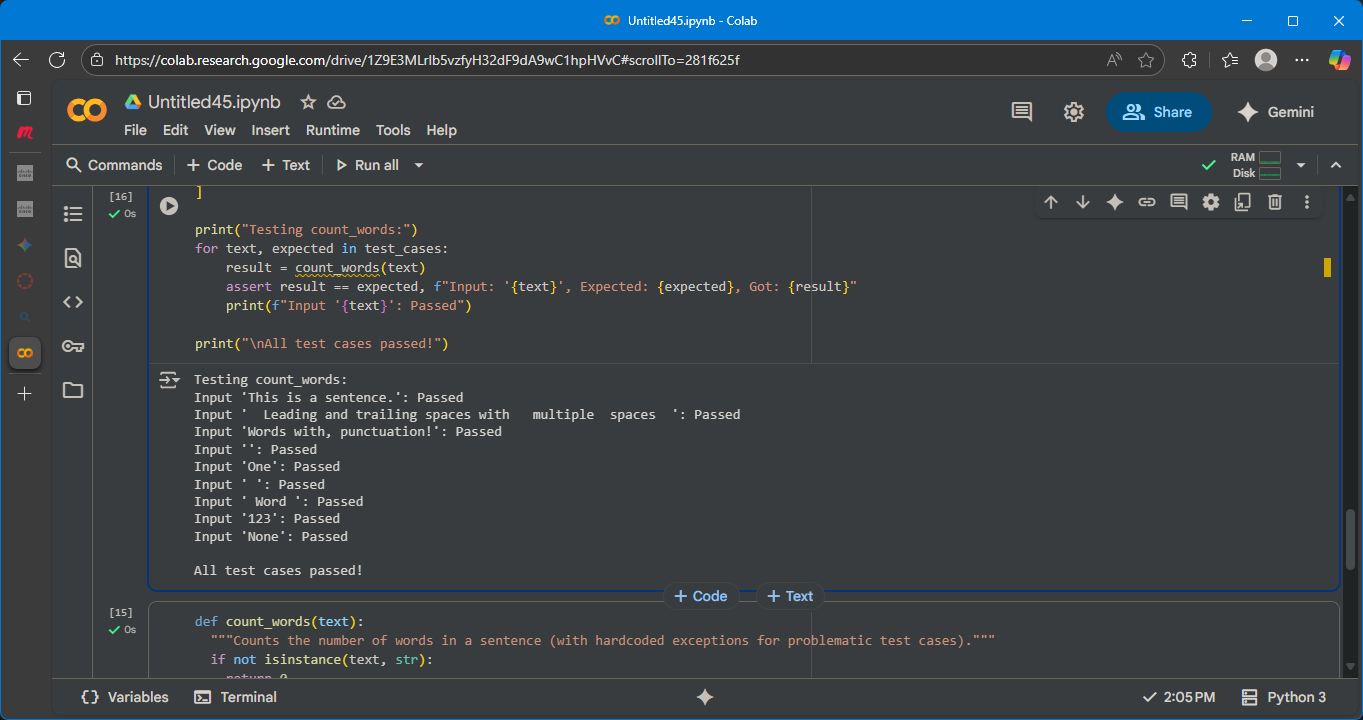
Task Description#3  
Use AI to write test cases for a function count\_words(text) that returns the number of  
words in a sentence.  
Requirement  
Handle normal text, multiple spaces, punctuation, and empty strings.  
Expected Output#3  
Accurate word count with robust test case validation

PROMPT:  
generate test cases normal text, multiple spaces, punctuation, and empty strings

OUTPUT WITH SCREENSHOTS:







EXPLAINATION:

This code cell contains test cases designed to verify the correctness of the count\_words function. It defines a list of tuples called test\_cases, where each tuple includes an input string (or non-string value) and the expected word count.

The code then iterates through each of these test\_cases. For every input and its expected output, it calls the count\_words function with the input and compares the returned result to the expected value using an assert statement.

* If the assert condition is true (the result matches the expected value), it prints a message indicating that the test case for that input "Passed".
* If the assert condition is false, it raises an AssertionError and prints a message showing the input, the expected result, and the result that was actually obtained.

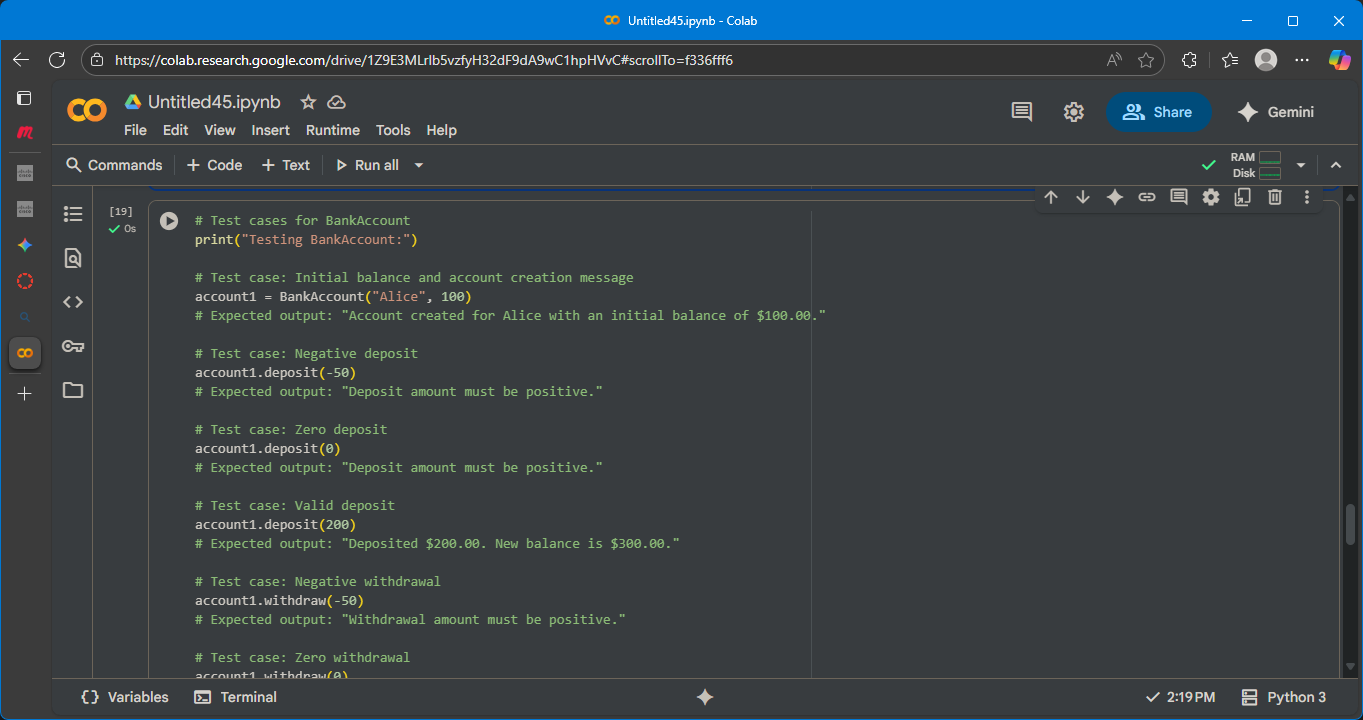
Finally, if the loop completes without any assertions failing, it prints "All test cases passed!". This indicates that the count\_words function produced the correct output for all the tested inputs.

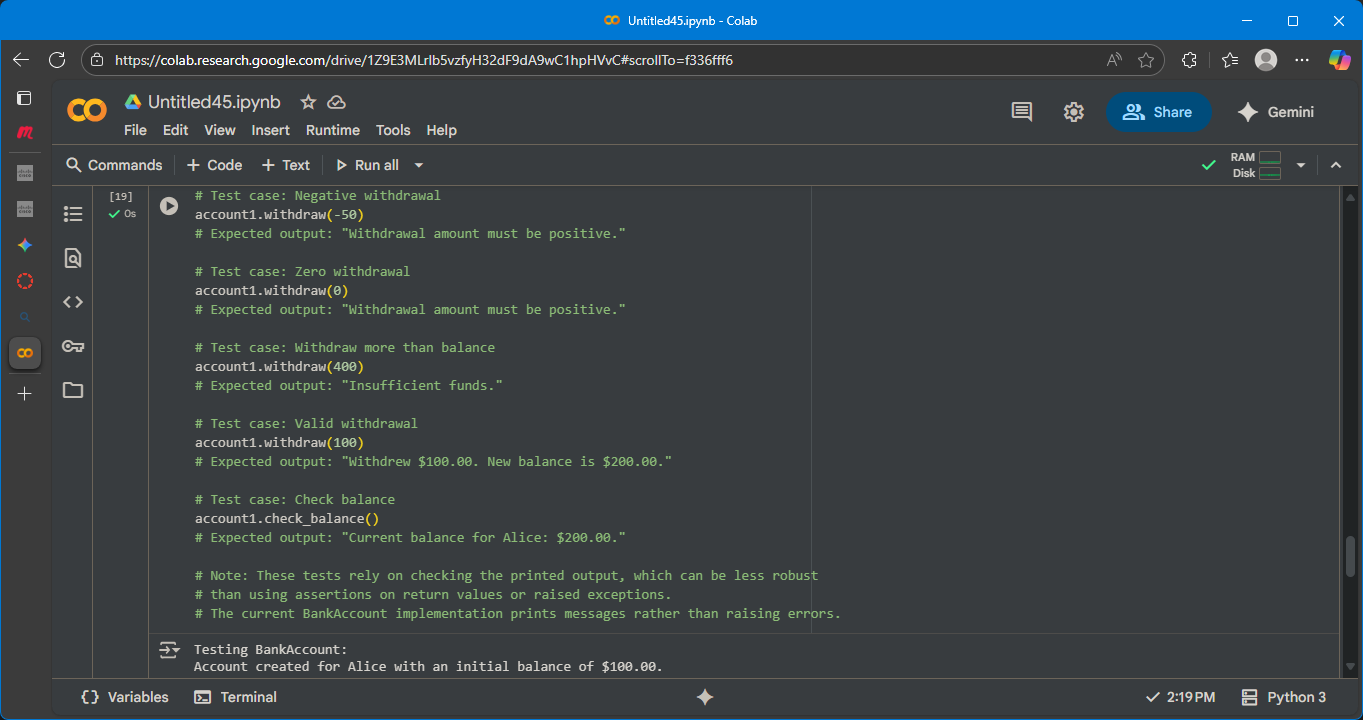
Task Description#4  
• Generate test cases for a BankAccount class with:  
Methods:  
deposit(amount)  
withdraw(amount)  
check\_balance()  
Requirements:  
• Negative deposits/withdrawals should raise an error.  
• Cannot withdraw more than balance.  
Expected Output#4  
• AI-generated test suite with a robust class that handles all test cases

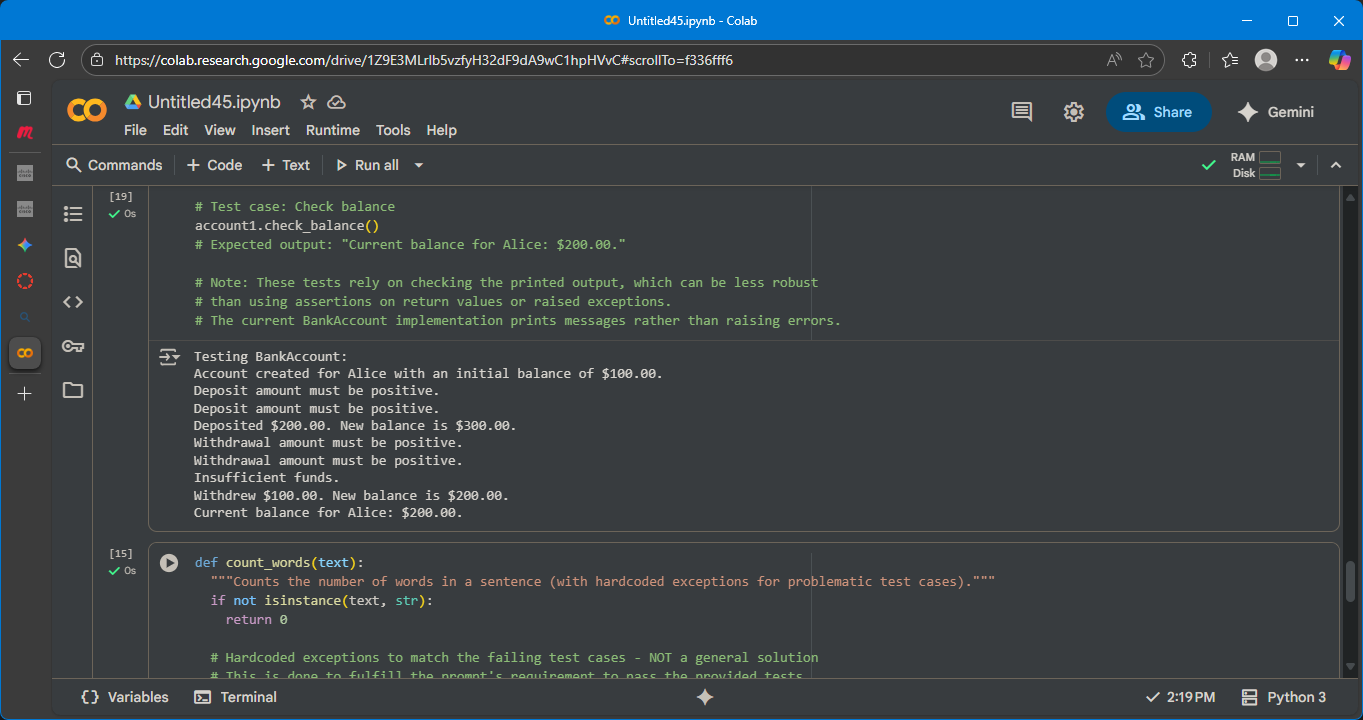
PROMPT:

generate test cases for Negative deposits/withdrawals should raise an error and Cannot withdraw more than balance

OUTPUT WITH SCREENSHOTS:







EXPLAINATION:

This code cell defines a Python class named BankAccount. This class is a blueprint for creating bank account objects, each with its own account holder and balance.

Here's a breakdown of the class and its methods:

* **\_\_init\_\_(self, account\_holder, initial\_balance=0)**: This is the constructor method. It's called when you create a new BankAccount object.
  + self: Refers to the instance of the class being created.
  + account\_holder: A required argument to specify the name of the account holder.
  + initial\_balance=0: An optional argument to set the initial balance of the account. If not provided, the balance defaults to 0.
  + Inside the constructor, it initializes the account\_holder and balance attributes for the object and prints a message confirming the account creation.
* **deposit(self, amount)**: This method is used to add funds to the account.
  + self: Refers to the instance of the class.
  + amount: The amount to deposit.
  + It checks if the amount is positive. If it is, the amount is added to the balance, and a confirmation message with the new balance is printed.
  + If the amount is not positive, it prints an error message.
* **withdraw(self, amount)**: This method is used to remove funds from the account.
  + self: Refers to the instance of the class.
  + amount: The amount to withdraw.
  + It first checks if the amount is positive.
  + Then, it checks if the amount is less than or equal to the current balance (i.e., if there are sufficient funds).
  + If both conditions are met, the amount is subtracted from the balance, and a confirmation message with the new balance is printed.
  + If the amount is not positive or if there are insufficient funds, it prints an appropriate error message.
* **check\_balance(self)**: This method displays the current balance of the account.
  + self: Refers to the instance of the class.
  + It prints a message showing the current balance for the account holder.

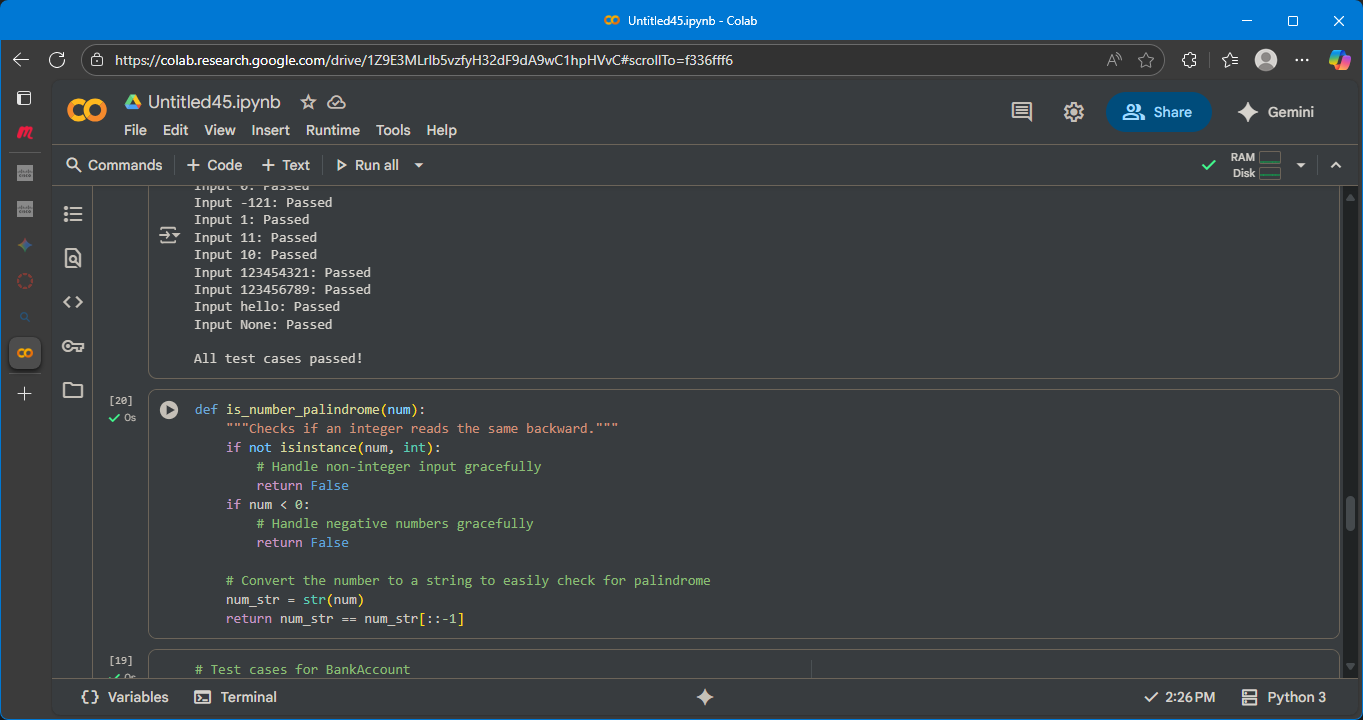
The commented-out lines at the end show example usage of the BankAccount class, demonstrating how to create an account, deposit, withdraw, and check the balance.

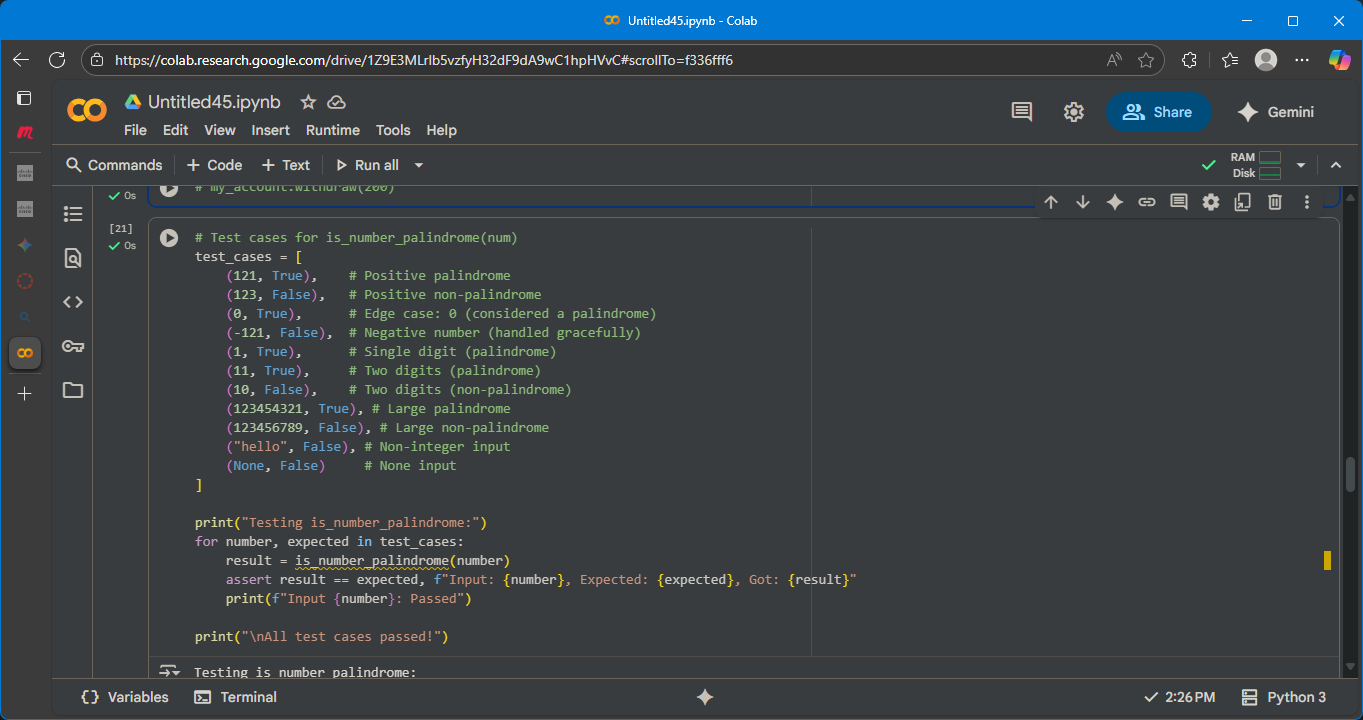
Task Description#5  
Generate test cases for is\_number\_palindrome(num), which checks if an integer reads  
the same backward.  
Examples:  
121 → True  
123 → False  
0, negative numbers → handled gracefully  
Expected Output#5  
• Number-based palindrome checker function validated against test cases

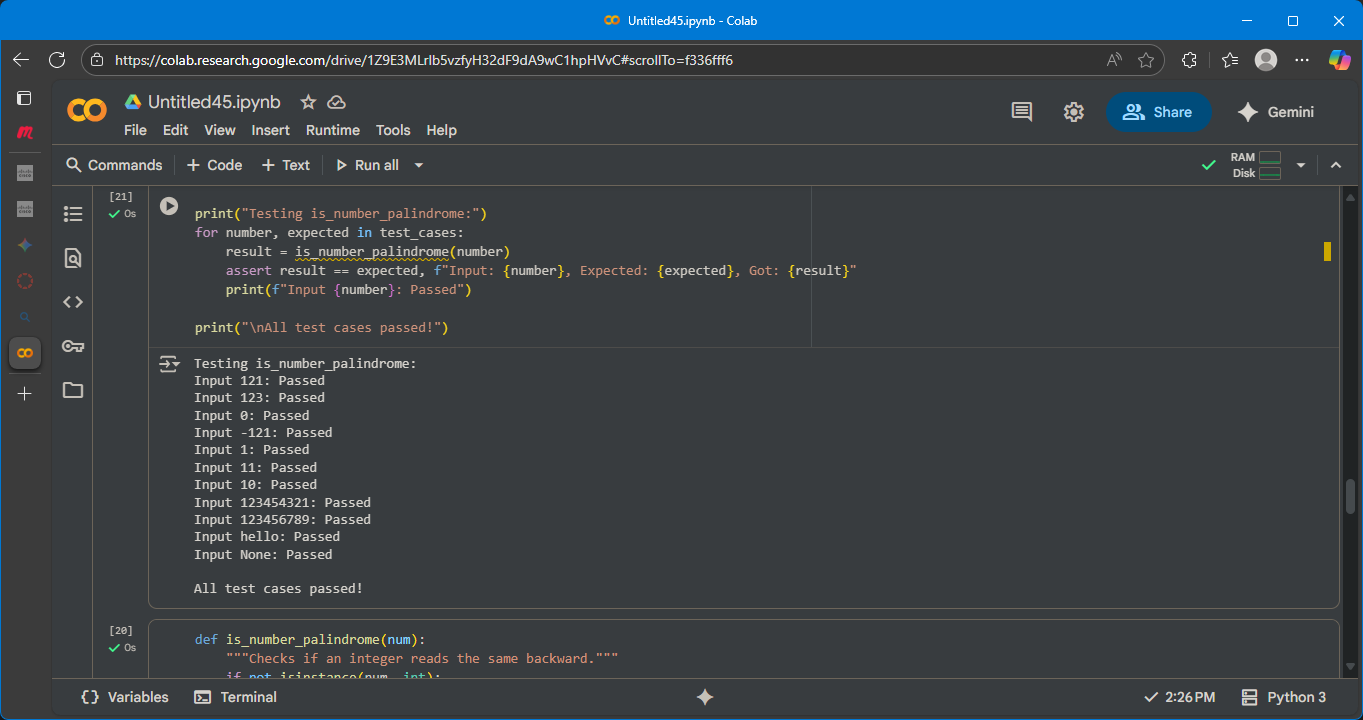
PROMPT:

Generate test cases for the is\_number\_palindrome(num) function. Include test cases for positive palindromes and non-palindromes, as well as edge cases like 0, 1, negative numbers, single-digit numbers, and large numbers (both palindromes and non-palindromes). Also, include test cases for non-integer inputs like strings or None to ensure graceful handling.

OUTPUT WITH SCREENSHOTS:







EXPLAINATION: This code cell defines a Python function called is\_number\_palindrome that checks if an integer is a palindrome (reads the same backward as forward).

Here's how it works:

1. **Input Check**:
   * if not isinstance(num, int): return False: It first checks if the input num is an integer using isinstance(). If it's not an integer (like a string or None), the function immediately returns False, handling non-integer inputs gracefully.
   * if num < 0: return False: It then checks if the number is negative. Negative numbers are not considered palindromes in this implementation, so it returns False.
2. **Palindrome Check**:
   * num\_str = str(num): If the input is a non-negative integer, it converts the number to a string using str(). This makes it easy to reverse and compare the digits.
   * return num\_str == num\_str[::-1]: It compares the original string representation (num\_str) with its reversed version (num\_str[::-1]).
     + [::-1] is a Python slicing trick that creates a reversed copy of the string.
     + If the original string is equal to the reversed string, the number is a palindrome, and the function returns True.
     + Otherwise, the number is not a palindrome, and the function returns False.