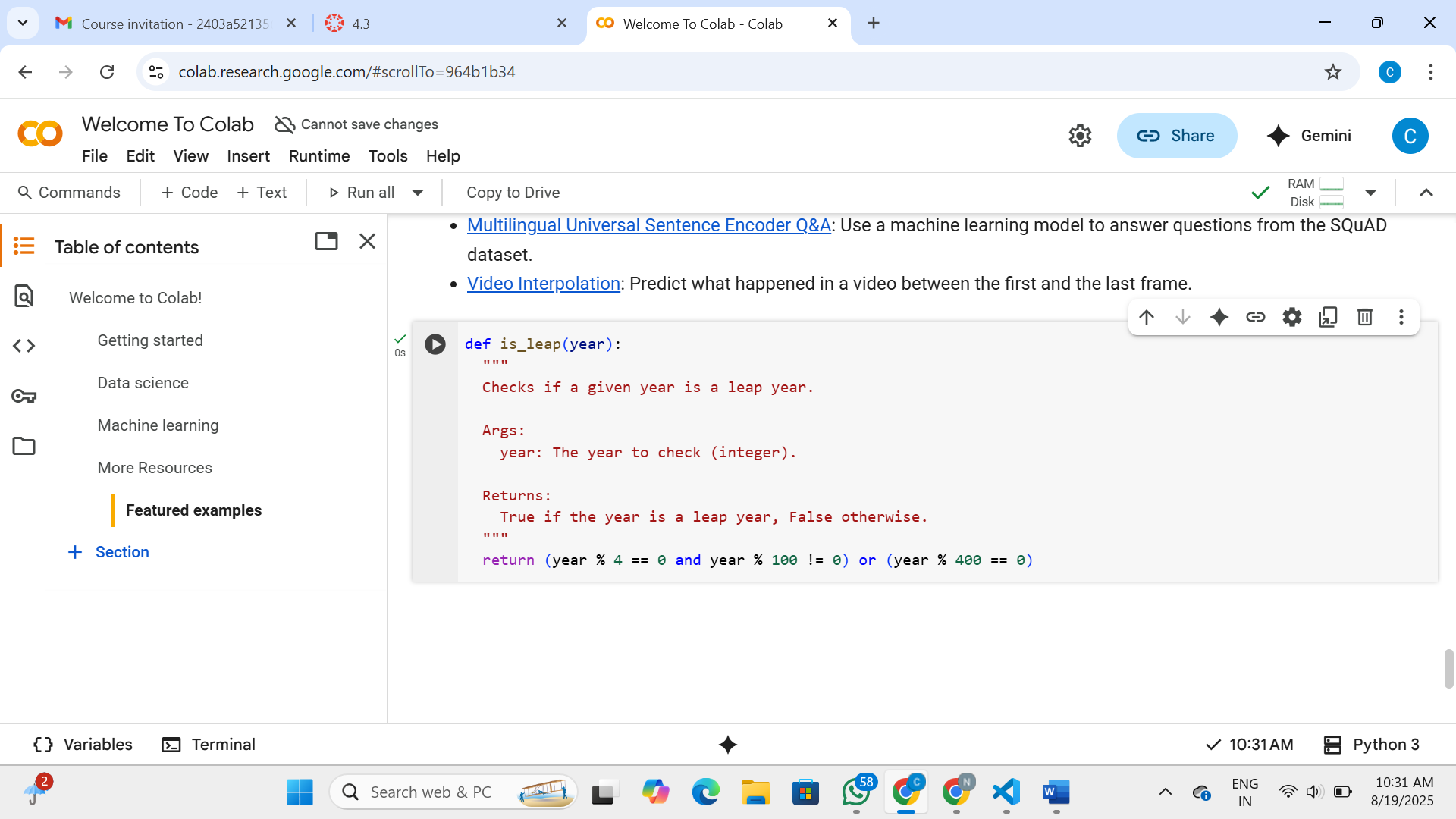
Assignment-4

Task-1

Zero write a function that checks whether a given year is a leap  
year.  
Expected Output#1  
● AI-generated function with no examples provided



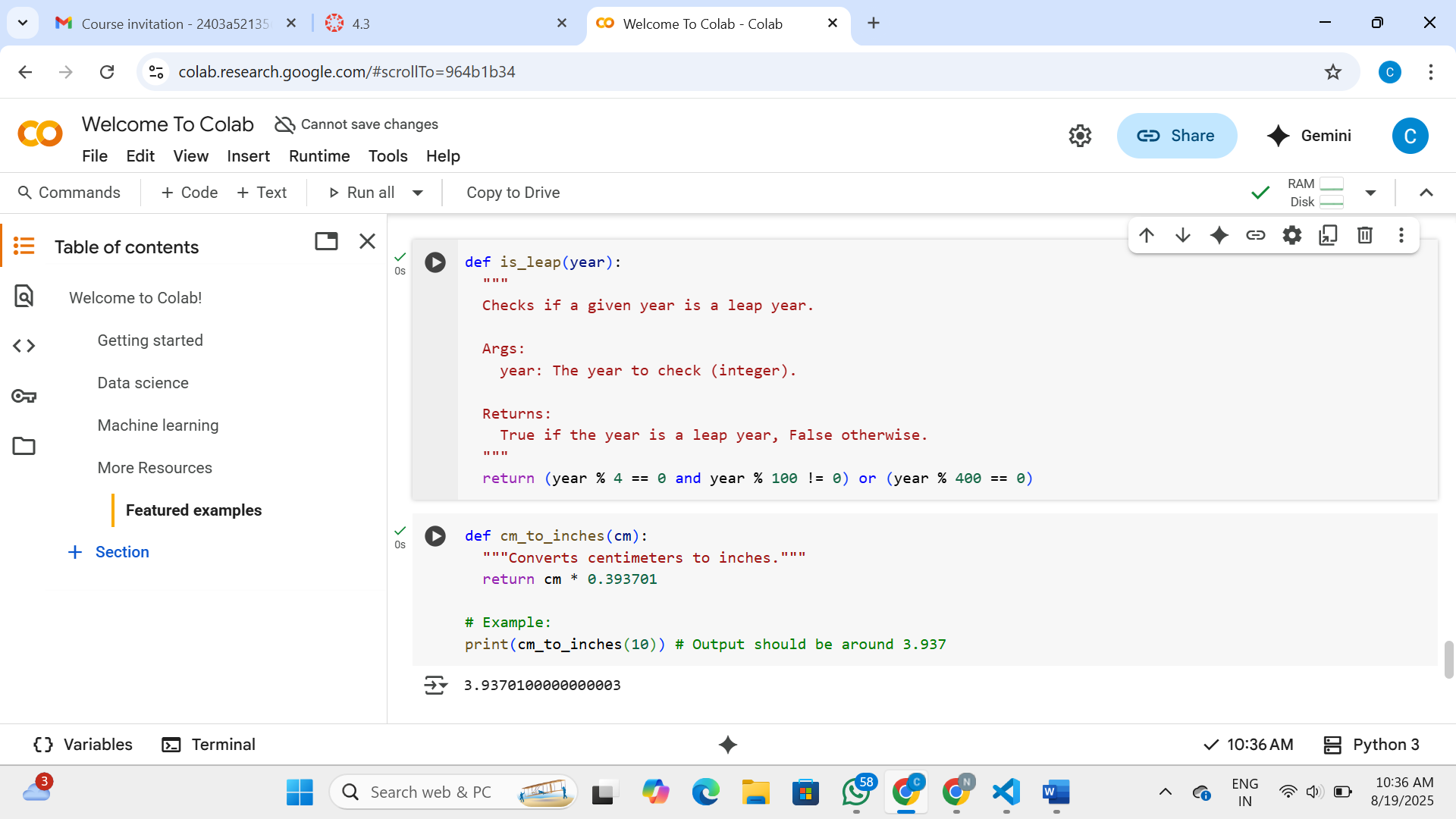
Explanation:

* **def is\_leap(year):**: This line defines the function is\_leap and specifies that it accepts one parameter, year.
* **""" Checks if a given year is a leap year. ... """**: This is a docstring, which explains what the function does, its arguments, and what it returns.
* **return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)**: This is the core of the function. It checks the conditions for a leap year:
  + year % 4 == 0: The year must be divisible by 4.
  + year % 100 != 0: If the year is divisible by 4, it must *not* be divisible by 100, unless...
  + year % 400 == 0: ...it *is* divisible by 400.

The and and or operators combine these conditions according to the rules for determining a leap year. The function returns True if the year meets these conditions (is a leap year) and False otherwise.

Task-2

One-shot: Give one input-output example to guide AI in writing a function that  
converts centimeters to inches.  
Expected Output#2  
● Function with correct conversion logic



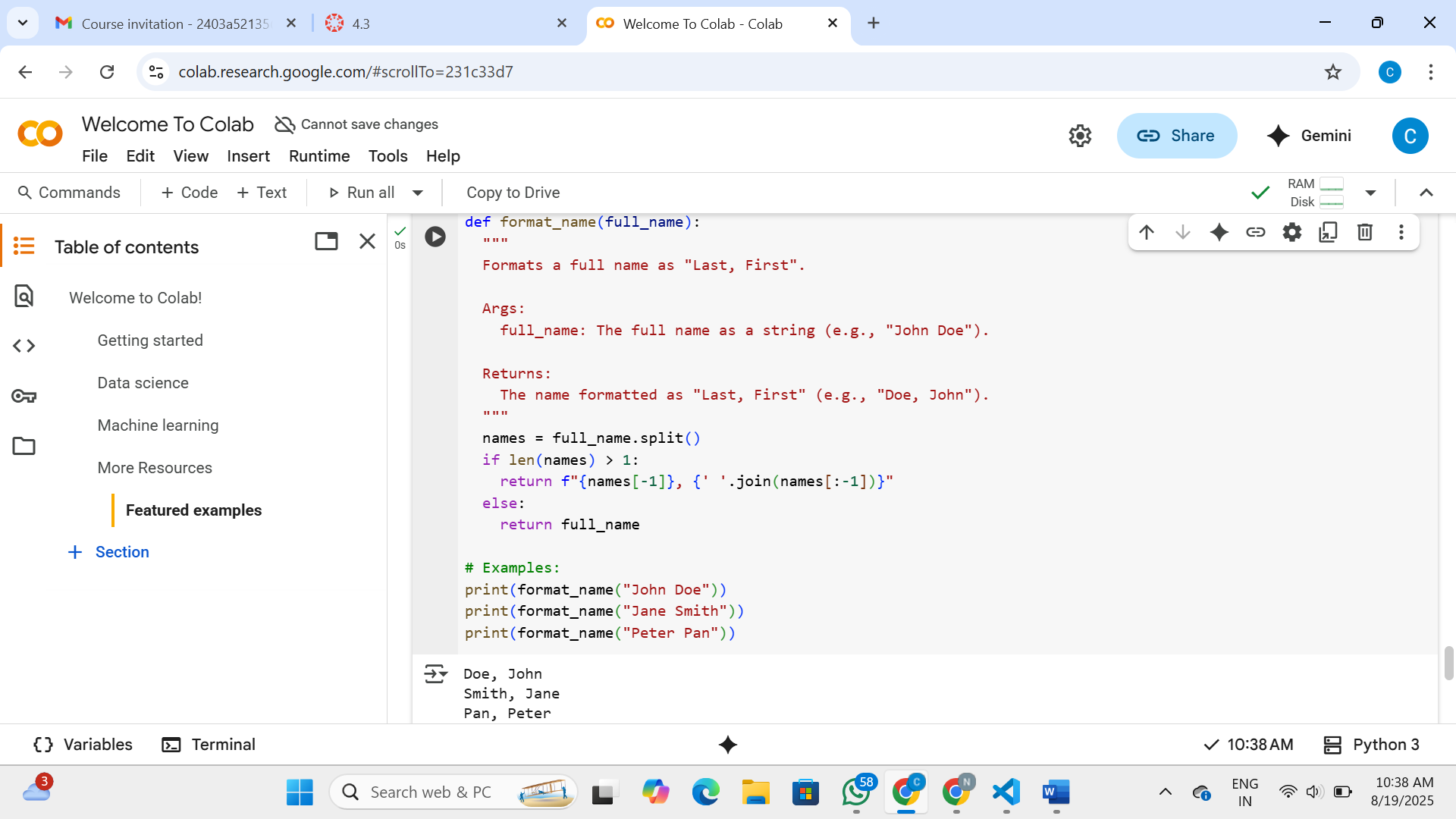
Explanation:

* **def cm\_to\_inches(cm):**: This line defines a function named cm\_to\_inches that takes one argument, cm, which represents the value in centimeters you want to convert.
* **"""Converts centimeters to inches."""**: This is a docstring explaining the purpose of the function.
* **return cm \* 0.393701**: This is the core of the function. It takes the input value cm and multiplies it by the conversion factor 0.393701. This factor is the number of inches in one centimeter. The result of this multiplication is then returned by the function.
* **# Example:**: This is a comment indicating that the following line is an example of how to use the function.
* **print(cm\_to\_inches(10))**: This line calls the cm\_to\_inches function with the value 10 (representing 10 centimeters) and then prints the result to the console. The comment # Output should be around 3.937 indicates the expected output of this example.

In essence, the function takes a measurement in centimeters and gives you the equivalent measurement in inches.

Task-3

Few-shot: Provide 2–3 examples to generate a function that formats full names as  
“Last, First”.  
Expected Output#3  
● Well-structured function respecting the examples



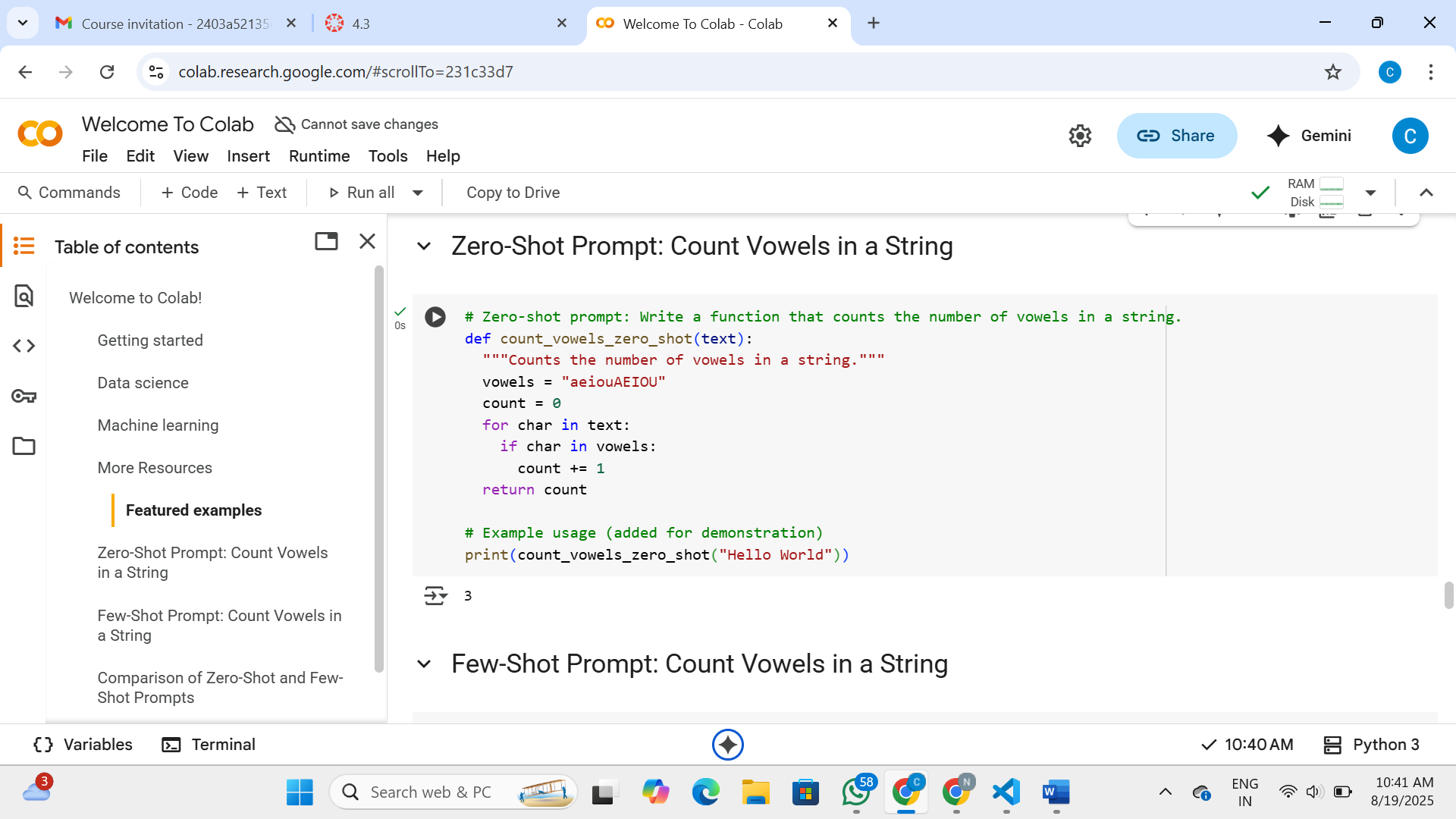
Explanation:

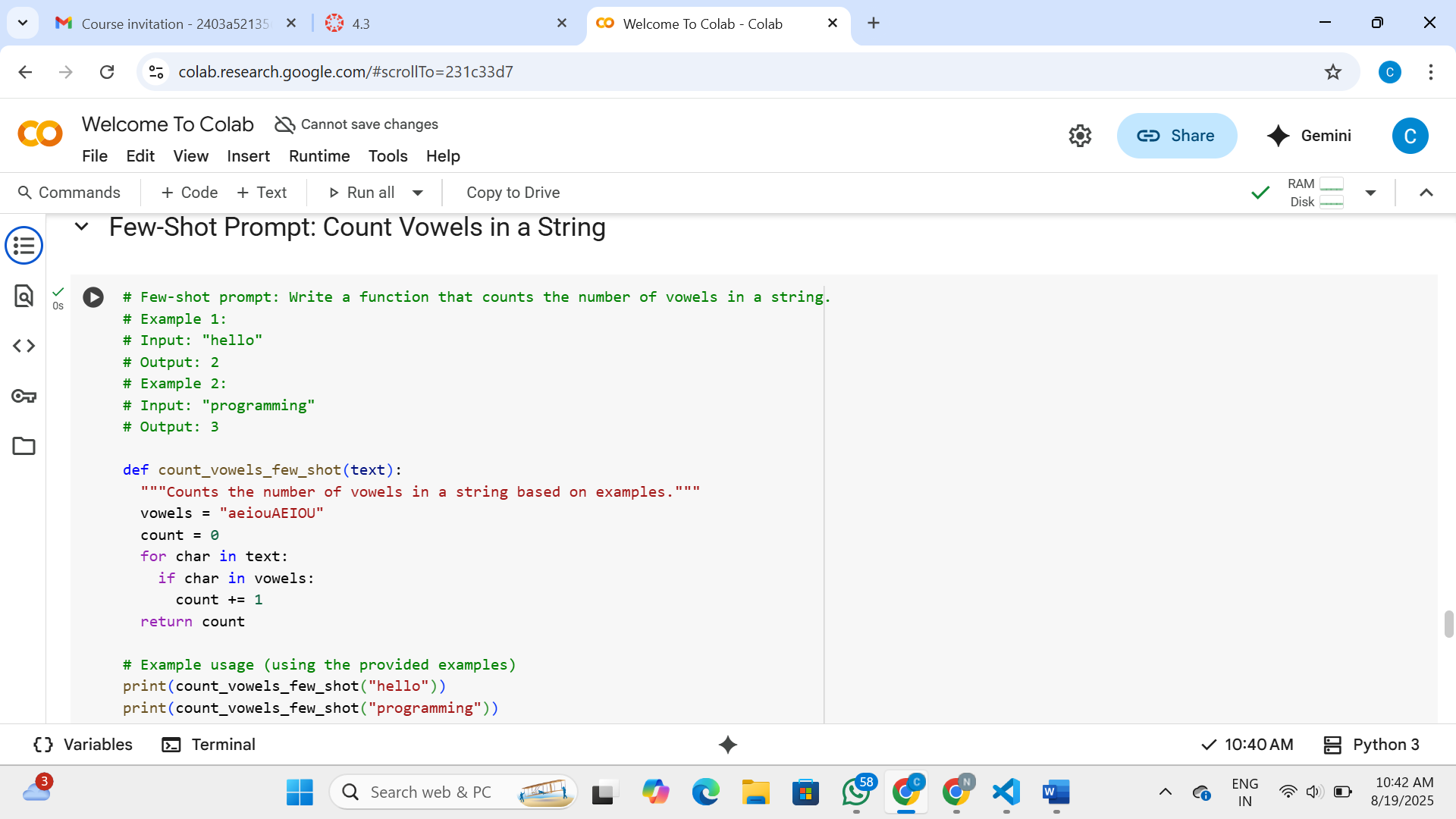
* **def format\_name(full\_name):**: This line defines the function format\_name and indicates that it takes one argument, full\_name, which is expected to be a string containing the full name.
* **""" Formats a full name as "Last, First". ... """**: This is the docstring, explaining what the function does, its arguments, and what it returns.
* **names = full\_name.split()**: This line splits the full\_name string into a list of individual names using spaces as the delimiter. For example, "John Doe" would become ['John', 'Doe'].
* **if len(names) > 1:**: This checks if there is more than one name in the list (i.e., if it's not just a single word).
* **return f"{names[-1]}, {' '.join(names[:-1])}"**: If there's more than one name, this line formats the output string:
  + names[-1] gets the last element of the list (the last name).
  + names[:-1] gets all elements except the last one (the first and any middle names).
  + ' '.join(names[:-1]) joins the elements of the list (except the last one) back into a single string with spaces in between.
  + The f"" is an f-string, which allows you to embed variables directly into the string. It puts the last name first, followed by a comma and a space, and then the rest of the names.
* **else:**: This handles the case where the input full\_name is only one word.
* **return full\_name**: If there's only one name, the function returns the original input string without any changes.
* **# Examples:**: This is a comment indicating the start of example usage.
* **print(format\_name("John Doe"))**, **print(format\_name("Jane Smith"))**, **print(format\_name("Peter Pan"))**: These lines demonstrate how to call the format\_name function with different inputs and print the results.

In summary, the function takes a full name, splits it into parts, and then reconstructs it with the last name appearing first, separated by a comma

Task-4

Compare zero-shot and few-shot prompts for writing a function that counts the  
number of vowels in a string.  
Expected Output#4  
● Functional output and comparative reflection





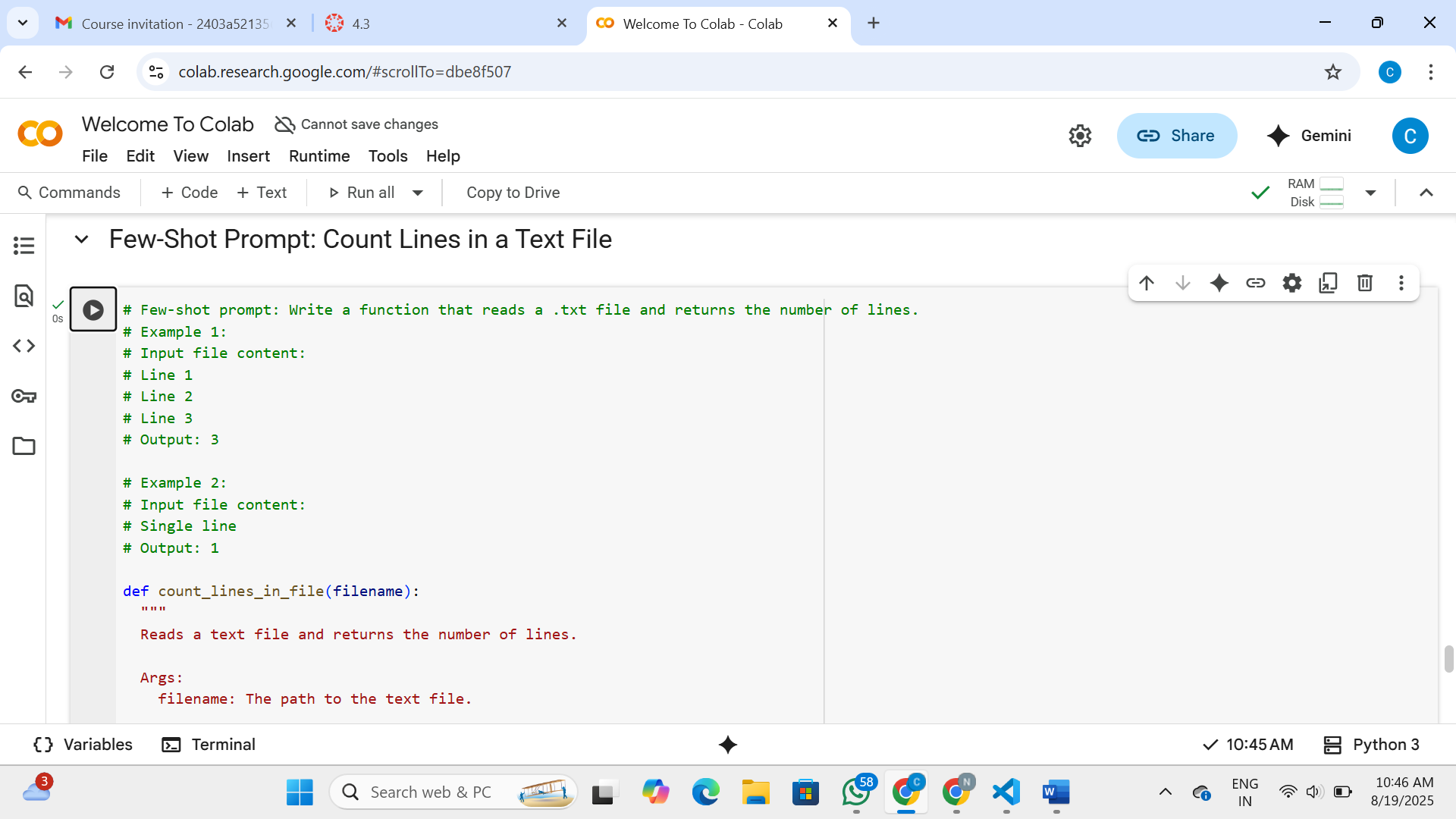
Explanation:

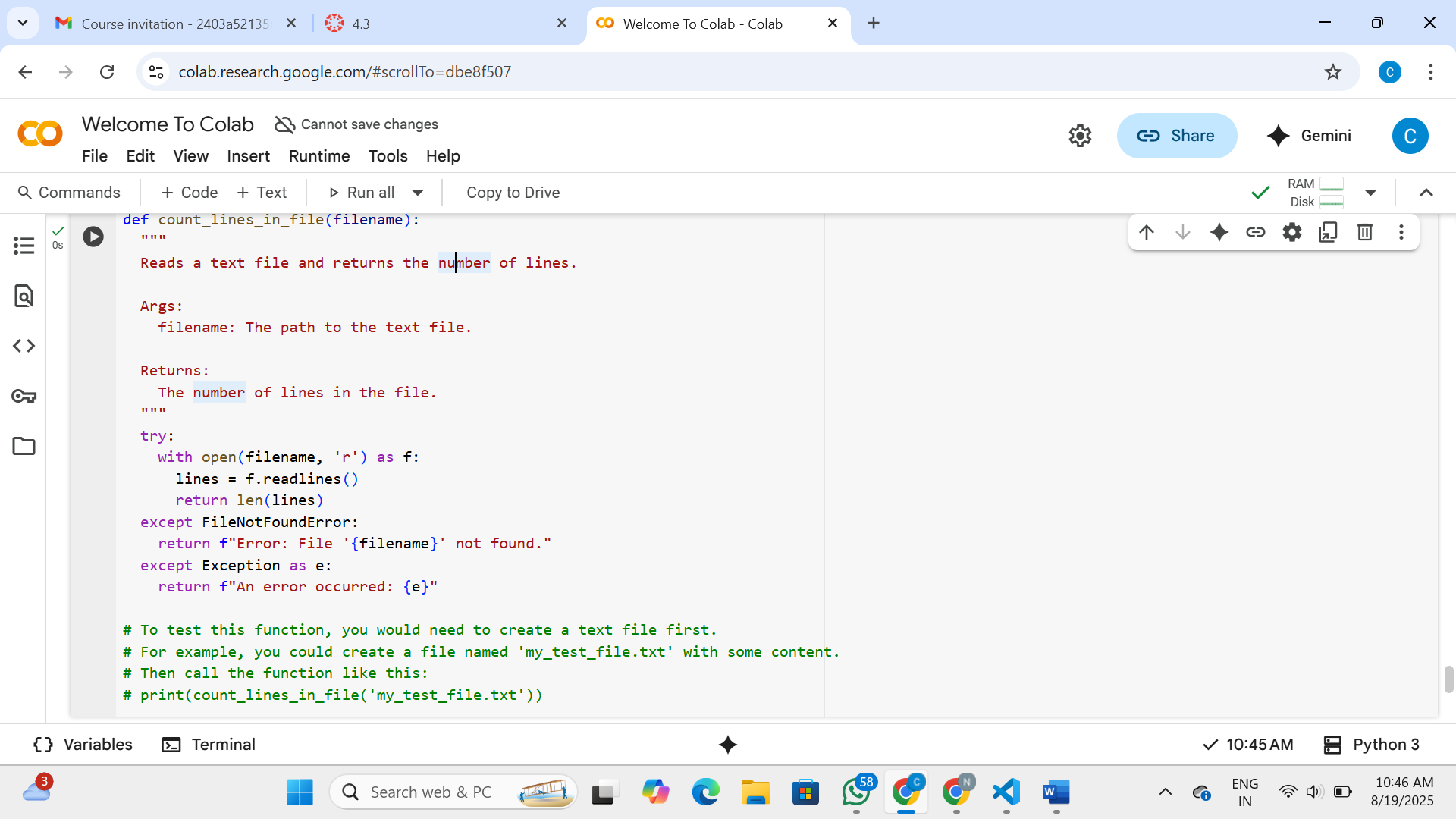
Both zero-shot and few-shot prompting were used to generate a Python function that counts the number of vowels in a string.  
  
**\*\*Zero-Shot Prompt:\*\***  
- **\*\*Prompt:\*\*** "Write a function that counts the number of vowels in a string."  
- **\*\*Result:\*\*** The AI generated a correct and functional Python function (`count\_vowels\_zero\_shot`) to count vowels. No examples were provided in the prompt, so the AI relied solely on its training data to understand the request and generate the code.  
  
**\*\*Few-Shot Prompt:\*\***  
- **\*\*Prompt:\*\*** "Write a function that counts the number of vowels in a string." (followed by two input-output examples).  
- **\*\*Result:\*\*** The AI also generated a correct and functional Python function (`count\_vowels\_few\_shot`). In this case, the provided examples helped guide the AI in understanding the desired behavior and format of the function, although for this relatively simple task, the core logic remained the same as the zero-shot case.  
  
**\*\*Comparative Reflection:\*\***  
  
For a straightforward task like counting vowels, both zero-shot and few-shot prompting can produce accurate results. However, few-shot prompting can be particularly useful for:  
  
\*   **\*\*Complex or ambiguous tasks:\*\*** When the request is more complex or could have multiple interpretations, providing examples helps to clarify the desired outcome and guide the AI towards the correct solution.  
\*   **\*\*Specific output formats:\*\*** If you need the output in a very specific format, examples can demonstrate that format to the AI.  
\*   **\*\*Learning new patterns:\*\*** Few-shot prompting can help the AI learn new patterns or apply existing knowledge in novel ways by showing it how to handle specific cases.  
  
In this specific case of counting vowels, the few-shot examples primarily served to reinforce the understanding of the task rather than fundamentally change the generated code. For more intricate programming tasks, few-shot prompting can significantly improve the relevance and accuracy of the AI's response.

This markdown cell summarizes the process and results of using both zero-shot and few-shot prompting for the vowel counting task. It explains what each approach involved and discusses the comparative effectiveness, highlighting when few-shot prompting can be particularly beneficial, especially for more complex tasks or when a specific output format is required.

Task-5

Use few-shot prompting to generate a function that reads a .txt file and returns the  
number of lines.  
Expected Output#5  
● Working file-processing function with AI-guided logic





Explanation:

This code defines a function called count\_lines\_in\_file that is designed to read a text file and return the total number of lines it contains.

* **def count\_lines\_in\_file(filename):**: This line defines the function count\_lines\_in\_file and indicates that it takes one argument, filename, which is expected to be a string representing the path to the text file.
* **""" Reads a text file and returns the number of lines. ... """**: This is the docstring, explaining the function's purpose, arguments, and return value.
* **try:**: This block starts a try-except block, which is used for error handling. It attempts to execute the code within the try block, and if an error occurs, it will look for a matching except block to handle it.
* **with open(filename, 'r') as f:**: This line opens the file specified by filename in read mode ('r'). The with statement ensures that the file is automatically closed even if errors occur. The opened file object is assigned to the variable f.
* **lines = f.readlines()**: This line reads all the lines from the opened file f and stores them as a list of strings in the variable lines. Each string in the list represents a line from the file, including the newline character at the end.
* **return len(lines)**: This line calculates the number of elements in the lines list using the len() function. Since each element in the list is a line from the file, the length of the list is the number of lines. This value is then returned by the function.
* **except FileNotFoundError:**: This block catches the FileNotFoundError exception, which occurs if the specified file does not exist.
* **return f"Error: File '{filename}' not found."**: If a FileNotFoundError occurs, this line returns an informative error message indicating that the file was not found.
* **except Exception as e:**: This is a general exception handler that catches any other type of exception that might occur during file processing.
* **return f"An error occurred: {e}"**: If any other exception occurs, this line returns a generic error message including the specific error details.
* **# To test this function, ...**: This is a comment explaining how to test the function by creating a test file and calling the function with its name.

In summary, the function attempts to open and read a file, count the lines, and return the count. It also includes error handling for cases where the file is not found or other issues occur during file processing.