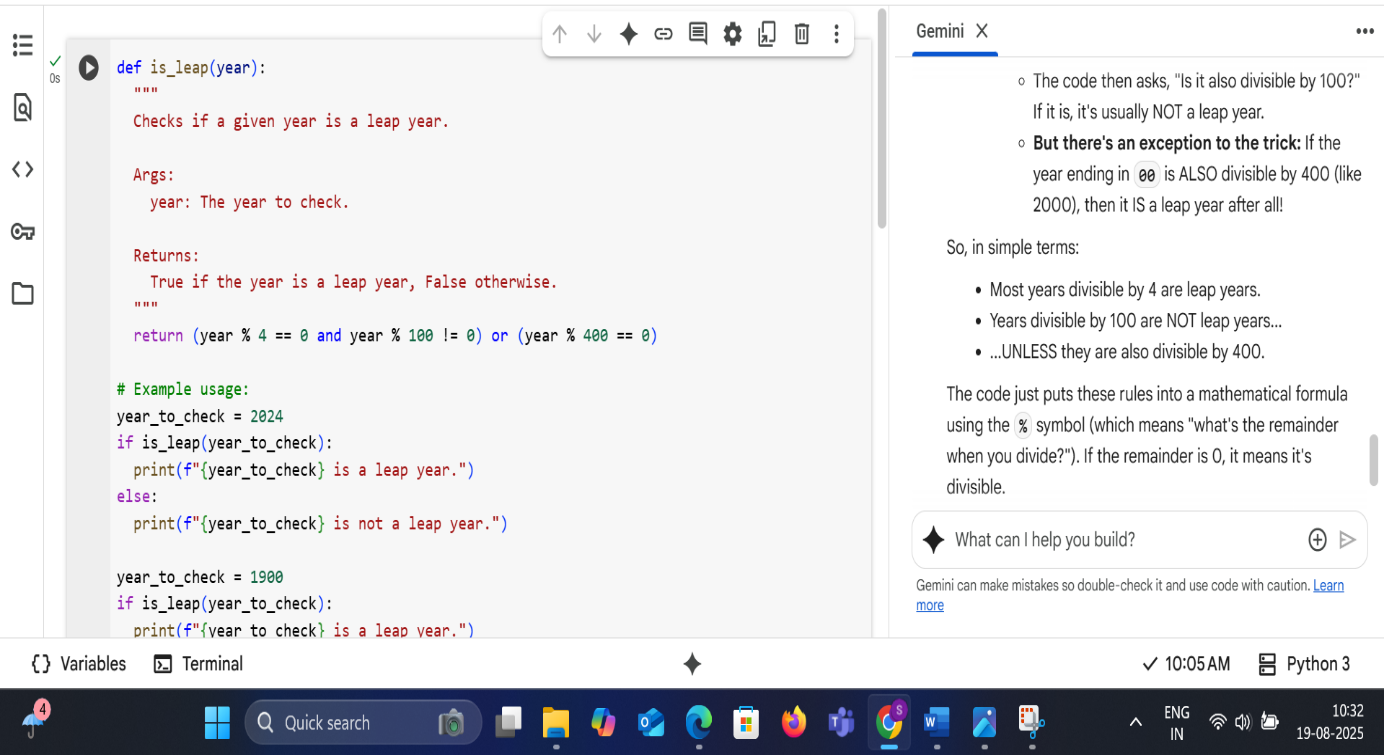
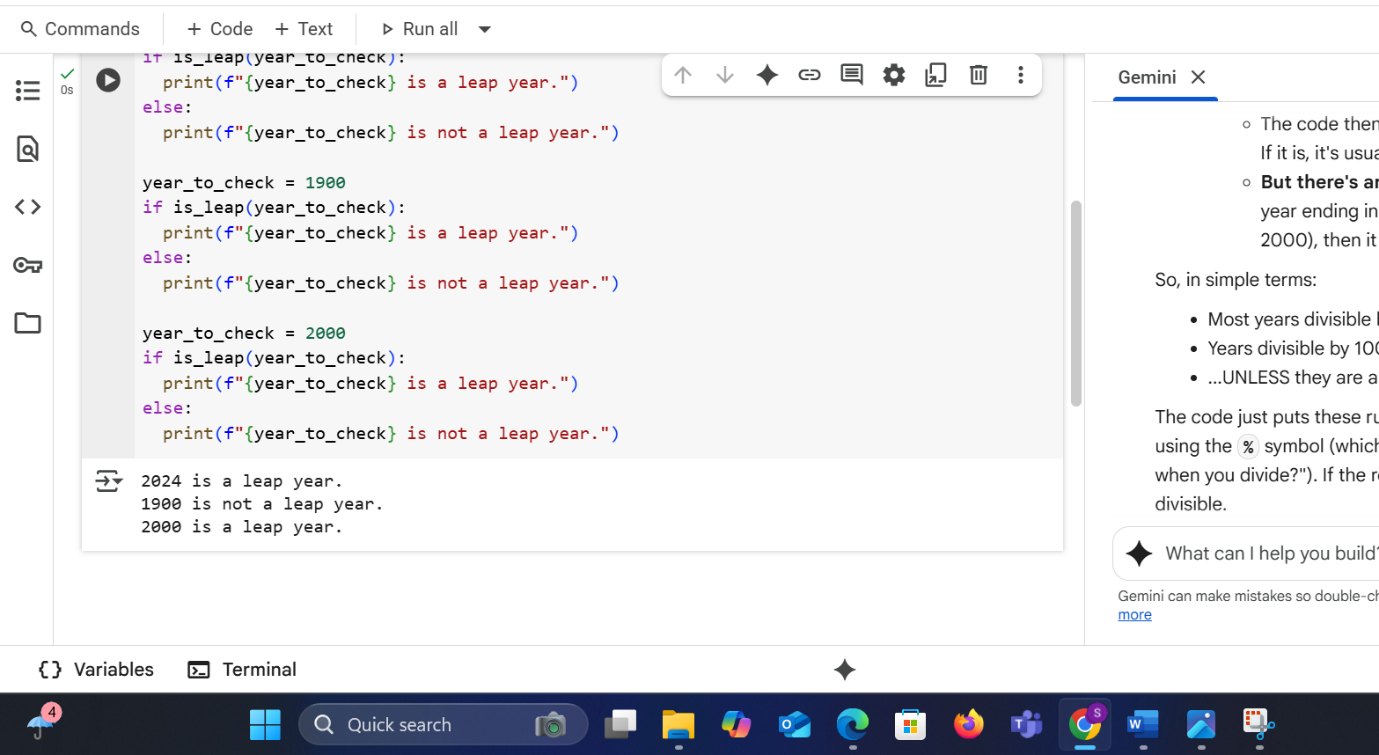
**ASSINGMENT-4.3**

* **TASK-1**
* Zero-shot: Prompt AI to write a function that checks whether a given year is a leap year





**EXPLANATION:**

Imagine a leap year is like a special bonus day added to the calendar every so often. Here's how the code decides if a year gets that bonus day:

1. **Check if it's divisible by 4:** The code first asks, "Can you divide this year evenly by 4?" If yes, it might be a leap year.
2. **But there's a trick for years ending in 00:** If the year ends in 00 (like 1900 or 2000), there's an extra rule.
   * The code then asks, "Is it also divisible by 100?" If it is, it's usually NOT a leap year.
   * **But there's an exception to the trick:** If the year ending in 00 is ALSO divisible by 400 (like 2000), then it IS a leap year after all!

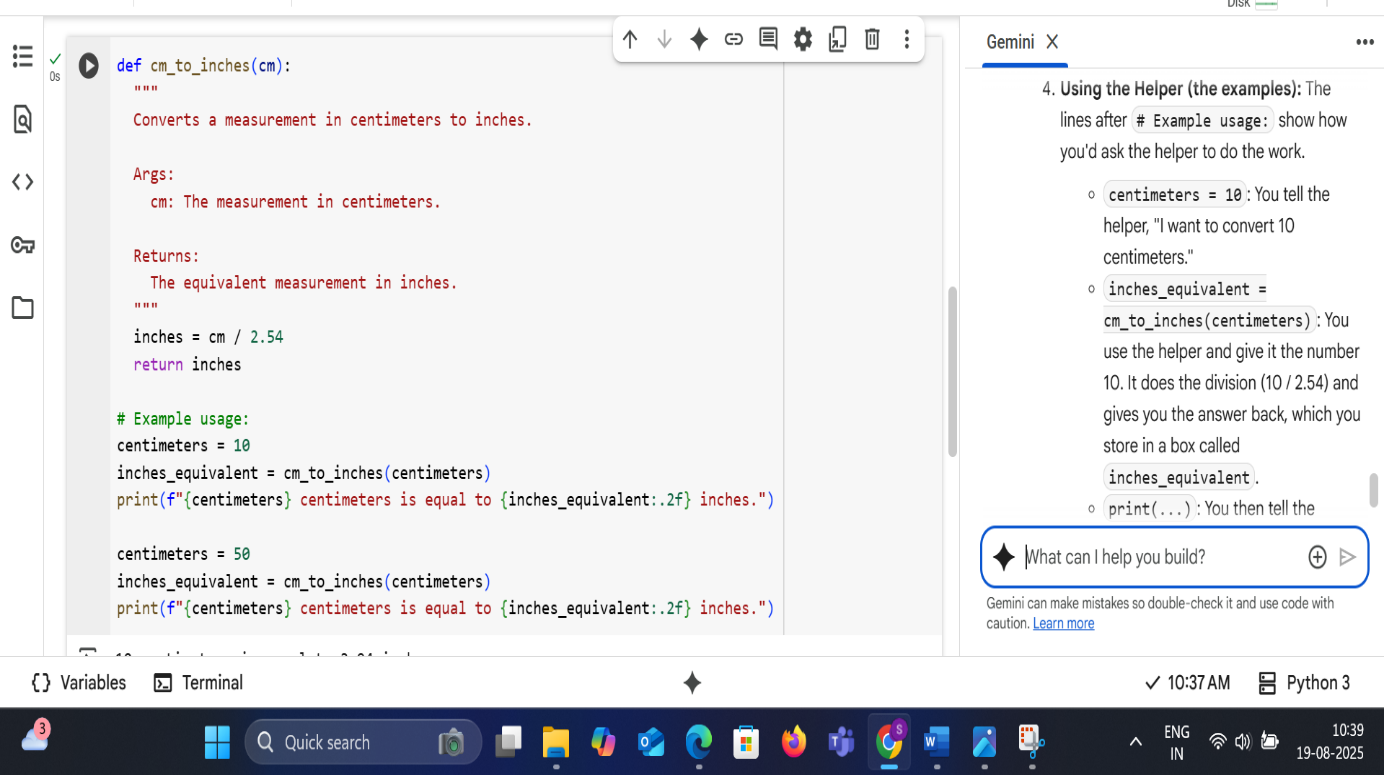
So, in simple terms:

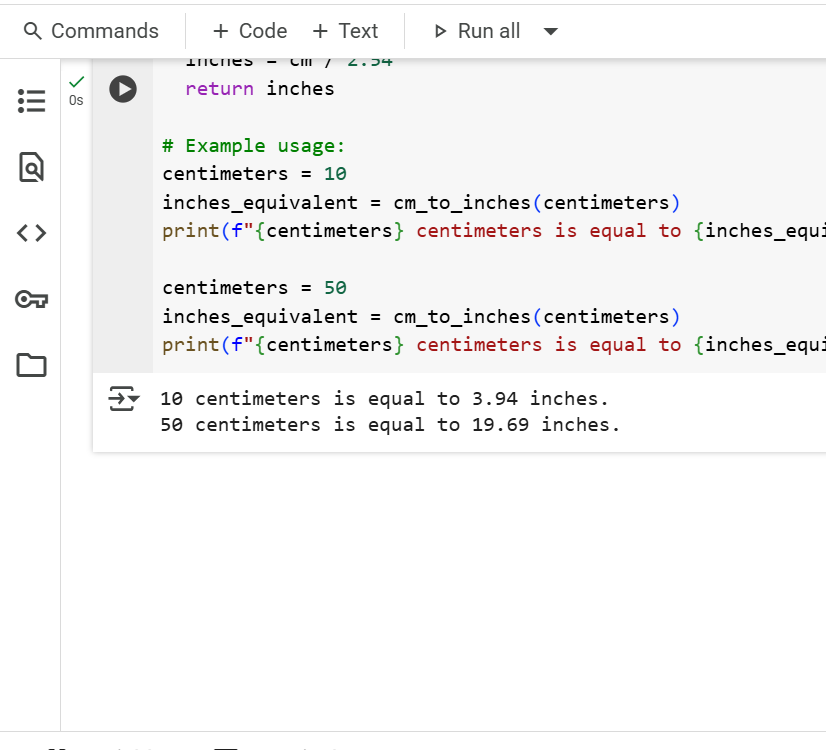
* Most years divisible by 4 are leap years.
* Years divisible by 100 are NOT leap years...
* ...UNLESS they are also divisible by 400.

The code just puts these rules into a mathematical formula using the % symbol (which means "what's the remainder when you divide?"). If the remainder is 0, it means it's divisible.

The examples at the end just show how the code applies these rules to 2024, 1900, and 2000 and prints the result.

* **TASK-2**
* One-shot: Give one input-output example to guide AI in writing a function that  
  converts centimeters to inches.





**EXPLANATION:**

Imagine you have a ruler that measures in centimeters, and you want to know how long something is using an inch ruler instead.

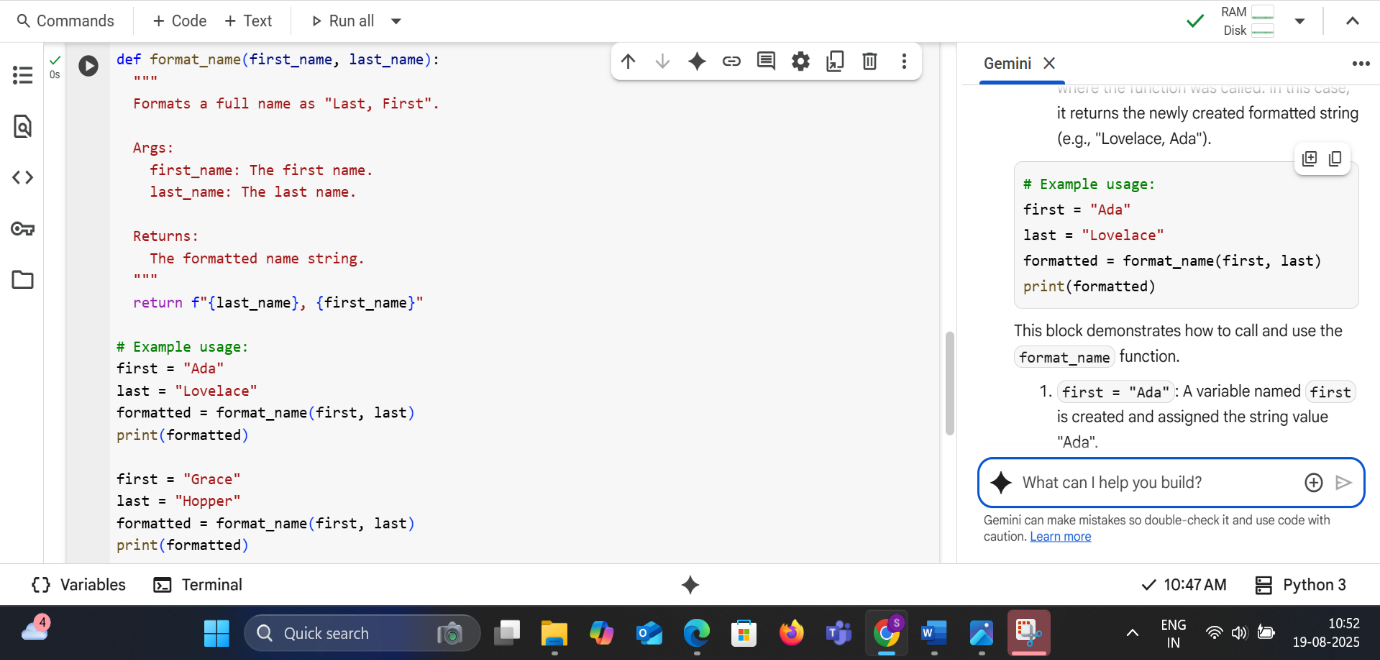
This code is like a little helper that does that conversion for you.

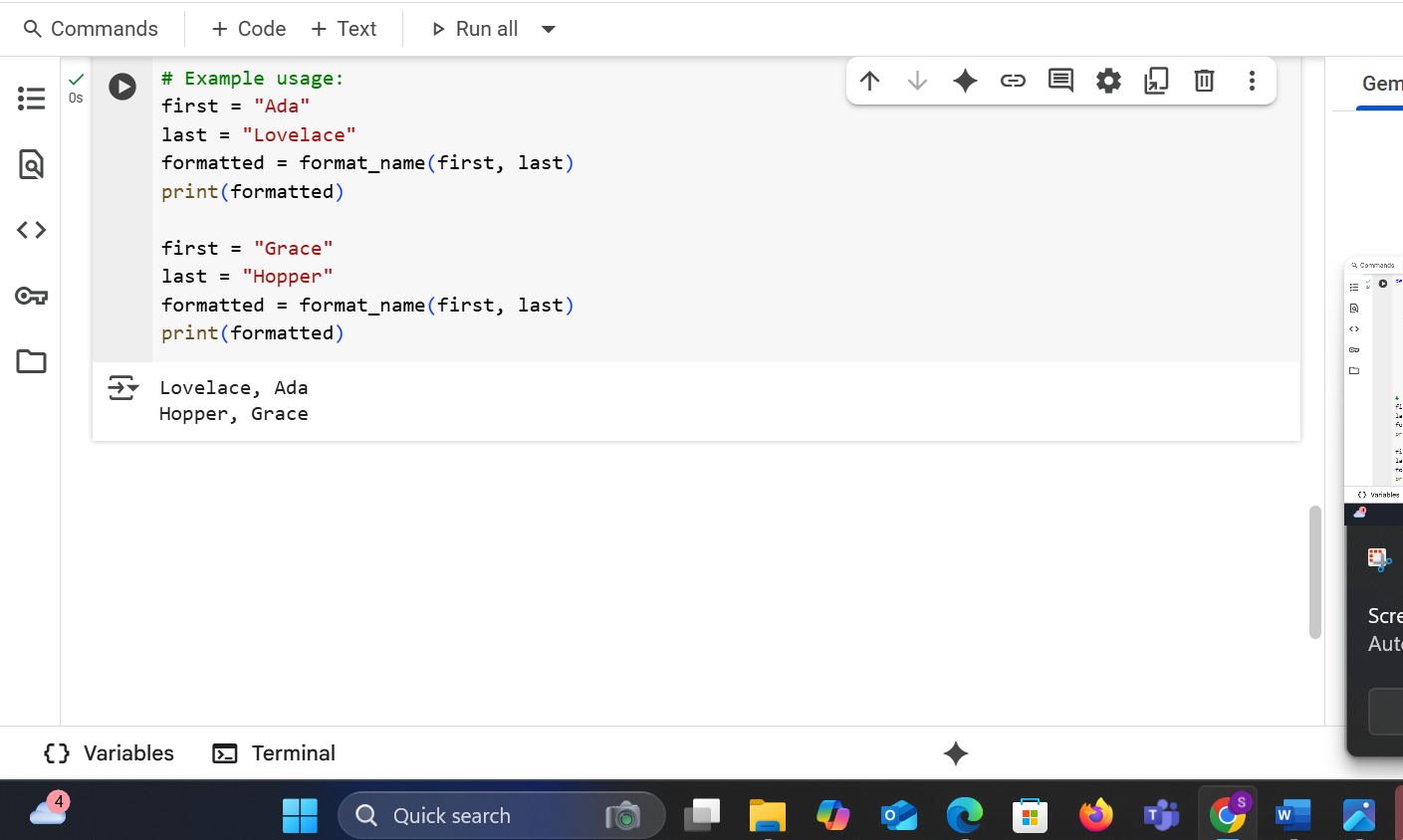
1. **The Helper (the function):** The first part def cm\_to\_inches(cm): is like saying, "Okay, I'm creating a helper called 'cm\_to\_inches', and you need to give it a number for the centimeters (cm) you want to convert."
2. **The Magic Formula:** Inside the helper, the line inches = cm / 2.54 is the key. It says, "To get the number of inches, take the centimeter number you were given and divide it by 2.54." (Because 1 inch is the same as 2.54 centimeters). The answer is then saved as inches.
3. **Giving You the Answer:** The line return inches is the helper saying, "Okay, I did the math, and here is the number of inches I calculated."
4. **Using the Helper (the examples):** The lines after # Example usage: show how you'd ask the helper to do the work.
   * centimeters = 10: You tell the helper, "I want to convert 10 centimeters."
   * inches\_equivalent = cm\_to\_inches(centimeters): You use the helper and give it the number 10. It does the division (10 / 2.54) and gives you the answer back, which you store in a box called inches\_equivalent.
   * print(...): You then tell the computer to show you the answer in a nice sentence.

The same thing happens for the 50 centimeters example.

So, the code is just a simple tool to take a measurement in centimeters and tell you what it is in inches.

* **Task-3**
* Few-shot: Provide 2–3 examples to generate a function that formats full names as:“Last, First”.





**EXPLANATION:**

format\_name function code:

def format\_name(first\_name, last\_name):

This line defines a function named format\_name that accepts two arguments: first\_name and last\_name.

  """  
  Formats a full name as "Last, First".  
  
  Args:  
    first\_name: The first name.  
    last\_name: The last name.  
  
  Returns:  
    The formatted name string.  
  """

This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns). It's a good practice to include these to make your code understandable.

  return f"{last\_name}, {first\_name}"

This line is the core of the function. It uses an f-string (formatted string literal) to create the output string. It takes the value of the last\_name variable, adds a comma and a space, and then adds the value of the first\_name variable. The resulting string is then returned by the function.

# Example usage:

This is a comment indicating the start of example code demonstrating how to use the function.

first = "Ada"  
last = "Lovelace"

These lines assign the string "Ada" to the variable first and the string "Lovelace" to the variable last.

formatted = format\_name(first, last)

This line calls the format\_name function with the values of first and last as arguments. The returned formatted name string is stored in the variable formatted.

print(formatted)

This line prints the value stored in the formatted variable to the console.

first = "Grace"  
last = "Hopper"

These lines assign new string values, "Grace" and "Hopper", to the first and last variables, respectively.

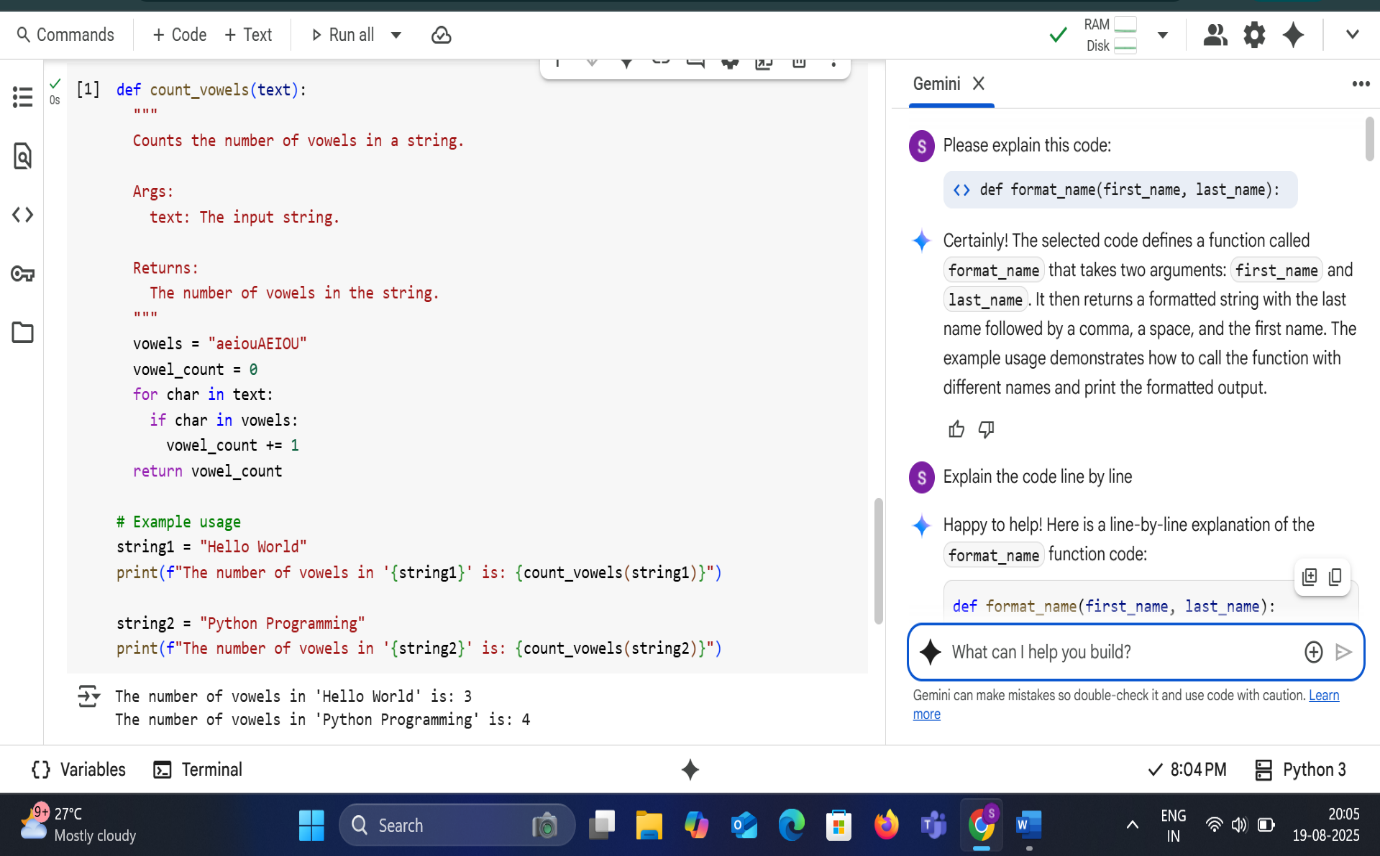
formatted = format\_name(first, last)

This line calls the format\_name function again with the new values of first and last. The returned formatted name string is again stored in the formatted variable, overwriting the previous value.

print(formatted)

This line prints the new value stored in the formatted variable to the console

* **Task-4**
* Compare zero-shot and few-shot prompts for writing a function that counts the number of vowels in a string



**EXPLANATION:**

**Zero-Shot Prompting** :In a zero-shot scenario, you would provide the AI model with only the instruction or description of the task, without any examples.

* **Prompt Example:** "Write a Python function that counts the number of vowels in a string."
* **How it works:** The AI relies solely on its pre-training and understanding of the task description to generate the code. It leverages the patterns and knowledge it has learned from a massive dataset of text and code.
* **Likely Outcome (for this specific task):** For a relatively straightforward task like counting vowels, a capable AI model is likely to generate a correct or nearly correct function in a zero-shot setting. The task is well-defined and common in programming.

**Few-Shot Prompting:**

In a few-shot scenario, you would provide the AI model with the instruction or description of the task along with a few examples of input-output pairs or examples of how the desired code should look.

* **Prompt Example:**

Write a Python function that counts the number of vowels in a string.

Example 1:

Input: "Hello"

Output: 2 # (e, o)

Example 2:

Input: "Python"

Output: 1 # (o)

Example 3:

Input: "AEIOU" Output: 5 # (A, E, I, O, U)

Write the function here:

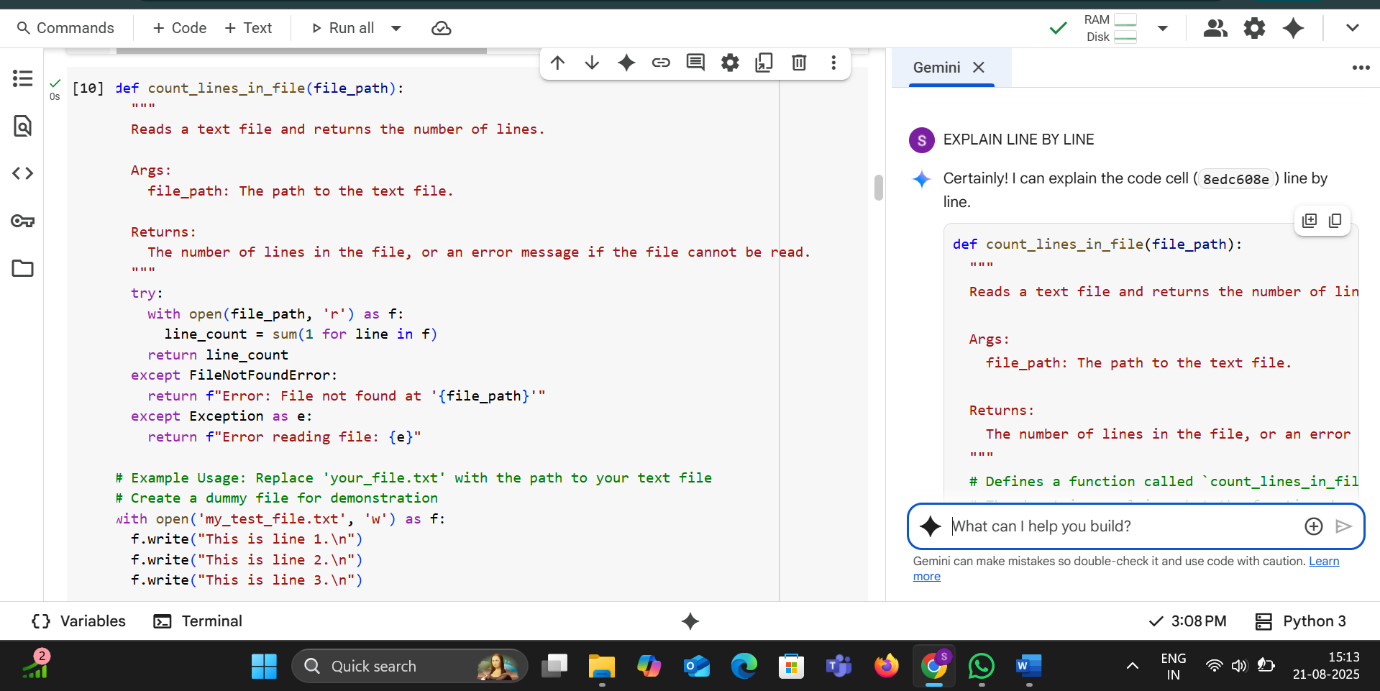
* **How it works:** The AI uses the provided examples as a guide to understand the desired behavior and format of the output code. The examples help the model to "prime" itself for the specific task and can be particularly useful for more complex or nuanced tasks.
* **Likely Outcome (for this specific task):** Providing examples for counting vowels would likely reinforce the correct logic for the AI and might help it avoid minor errors or edge cases (like handling uppercase letters, which our function does by including them in the vowels string). It could lead to a more robust or precisely formatted function, depending on the examples provided.

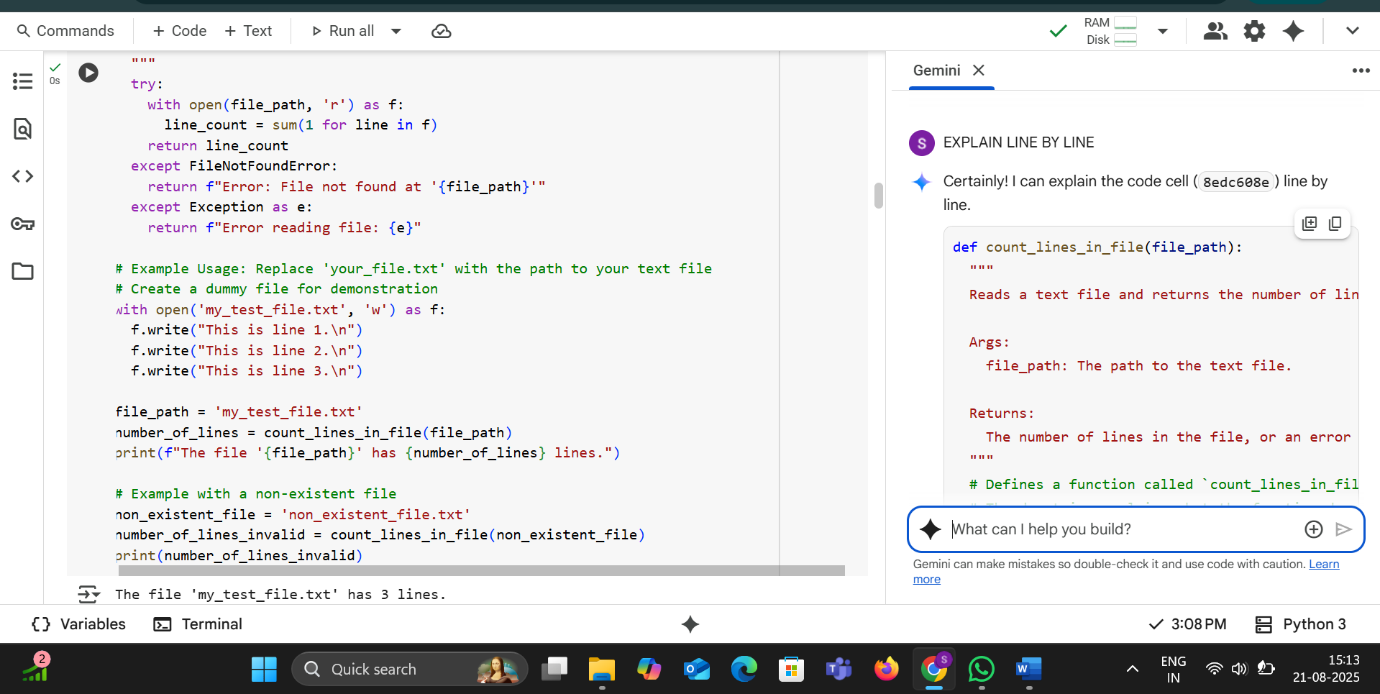
**Comparison:**

* **Simplicity:** Zero-shot is simpler as it requires less input from the user.
* **Performance:** For simple, common tasks, zero-shot can perform well. For more complex, ambiguous, or novel tasks, few-shot prompting generally leads to better results because the examples provide crucial context and reduce the ambiguity of the request.
* **Efficiency:** Zero-shot is generally faster as the prompt is shorter.
* **Control:** Few-shot prompting gives you more control over the style and specific requirements of the output by demonstrating them in the examples.

In the case of writing a function to count vowels, both zero-shot and few-shot prompting would likely yield a correct solution with a capable model. However, for more complex or domain-specific coding tasks, few-shot prompting would be the preferred approach to guide the AI towards the desired outcome.

* **Task-5**
* Use few-shot prompting to generate a function that reads a .txt file and returns the number of lines





* **LINE BY LINE EXPLANATION:**

def count\_lines\_in\_file(file\_path):

"""

Reads a text file and returns the number of lines.

Args:

file\_path: The path to the text file.

Returns:

The number of lines in the file, or an error message if the file cannot be read.

"""

# Defines a function called `count\_lines\_in\_file` that takes one argument, `file\_path`.

# The docstring explains what the function does, its arguments, and what it returns.

try:

# Starts a try block to handle potential errors during file operations.

with open(file\_path, 'r') as f:

# Opens the file specified by `file\_path` in read mode ('r').

# The `with` statement ensures the file is automatically closed even if errors occur.

# The opened file object is assigned to the variable `f`.

line\_count = sum(1 for line in f)

# This is a generator expression `(1 for line in f)` that iterates through each line in the file `f`.

# For each line, it yields the value `1`.

# The `sum()` function then sums up all the `1`s generated, effectively counting the number of lines.

return line\_count

# If the file is read successfully, the function returns the calculated `line\_count`.

except FileNotFoundError:

# This block is executed if a `FileNotFoundError` occurs (i.e., the file specified by `file\_path` does not exist).

return f"Error: File not found at '{file\_path}'"

# Returns an informative error message indicating that the file was not found.

except Exception as e:

# This block is executed for any other type of exception that might occur during file reading.

# The exception object is assigned to the variable `e`.

return f"Error reading file: {e}"

# Returns a generic error message including the specific exception that occurred.

# Example Usage: Replace 'your\_file.txt' with the path to your text file

# This is a comment explaining the following example usage section.

# Create a dummy file for demonstration

# This is a comment indicating the creation of a temporary file for testing.

with open('my\_test\_file.txt', 'w') as f:

# Opens a file named 'my\_test\_file.txt' in write mode ('w').

# If the file exists, its contents are truncated. If it doesn't exist, it's created.

# The `with` statement ensures the file is closed.

f.write("This is line 1.\n")

# Writes the string "This is line 1." followed by a newline character to the file.

f.write("This is line 2.\n")

# Writes the string "This is line 2." followed by a newline character to the file.

f.write("This is line 3.\n")

# Writes the string "This is line 3." followed by a newline character to the file.

file\_path = 'my\_test\_file.txt'

# Assigns the string 'my\_test\_file.txt' to the variable `file\_path`.

number\_of\_lines = count\_lines\_in\_file(file\_path)

# Calls the `count\_lines\_in\_file` function with the `file\_path` and stores the returned value in `number\_of\_lines`.

print(f"The file '{file\_path}' has {number\_of\_lines} lines.")

# Prints a formatted string to the console showing the number of lines in the created file.

# Example with a non-existent file

# This is a comment indicating the following example uses a file that does not exist.

non\_existent\_file = 'non\_existent\_file.txt'

# Assigns the string 'non\_existent\_file.txt' to the variable `non\_existent\_file`.

number\_of\_lines\_invalid = count\_lines\_in\_file(non\_existent\_file)

# Calls the `count\_lines\_in\_file` function with a non-existent file path and stores the returned error message.

print(number\_of\_lines\_invalid)

# Prints the error message returned by the function to the console.