**Lab 4: Advanced Prompt Engineering – Zero-shot, One-shot, and Few-shot Techniques**

**Assignment-4.1**

**Name: Bollam Sathvika**

**Htno:2403A51344**

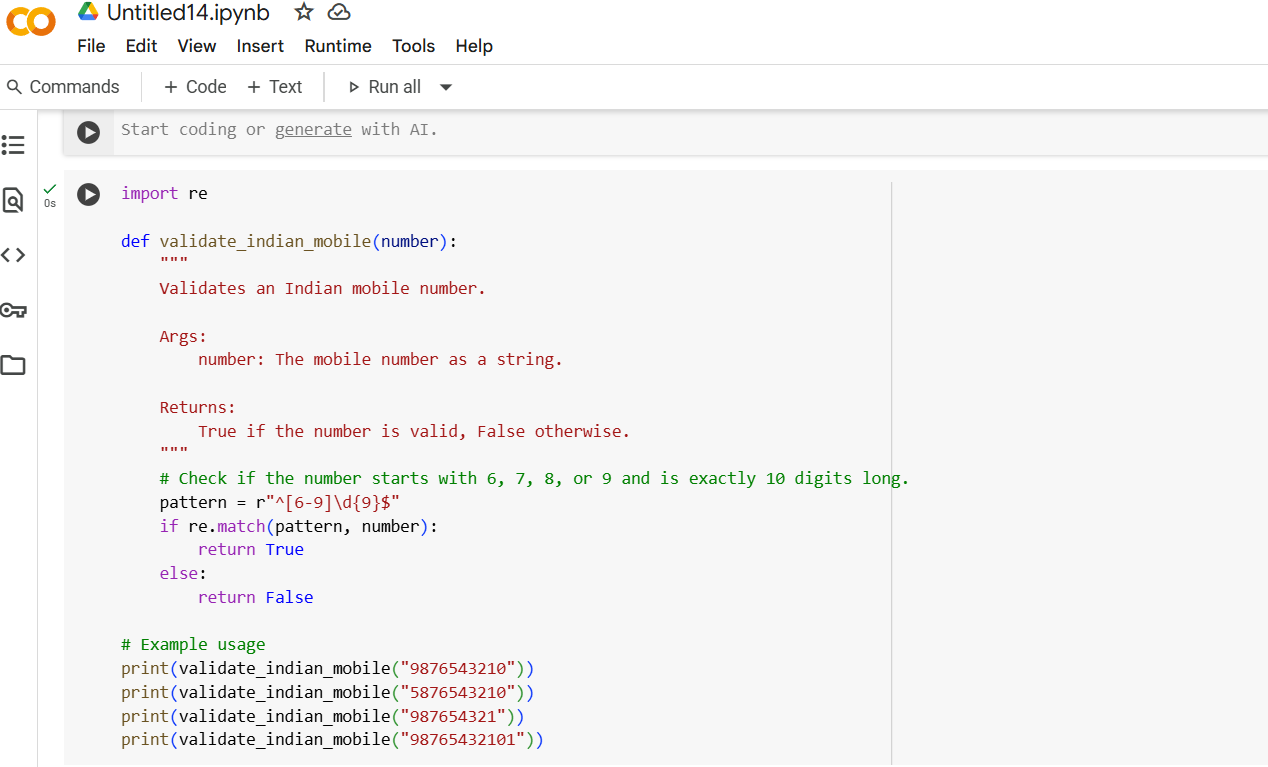
**Batchno:24BTCAICSB14**

**#TASK 1:**

**Prompt:**

Create a Python function that validates an Indian mobile number. The number should start with 6, 7, 8, or 9 and must be exactly 10 digits long. The function should return True if the number is valid and False otherwise.

**Code:**

****

**Output:**

****

**Explanation:**

1. **Importing the Regular Expression Module**  
   The code begins by importing Python’s built-in re module, which provides support for working with regular expressions—a powerful tool for pattern matching in strings.
2. **Defining the Validation Function**  
   A function named validate\_indian\_mobile is defined. This function takes a single argument called number, which is expected to be a string representing a mobile number.
3. **Adding a Docstring for Clarity**  
   Inside the function, a docstring is included to describe its purpose. It explains that the function validates an Indian mobile number, specifies that the input should be a string, and states that the function returns True if the number is valid and False otherwise.

**Creating the Regular Expression Pattern**  
The function then defines a regular expression pattern: r"^[6-9]\d{9}$". This pattern  is designed to match Indian mobile numbers that:

* + Start with a digit between 6 and 9 ([6-9])
  + Are followed by exactly nine more digits (\d{9})
  + Have no extra characters before or after (^ for start of string, $ for end of string)

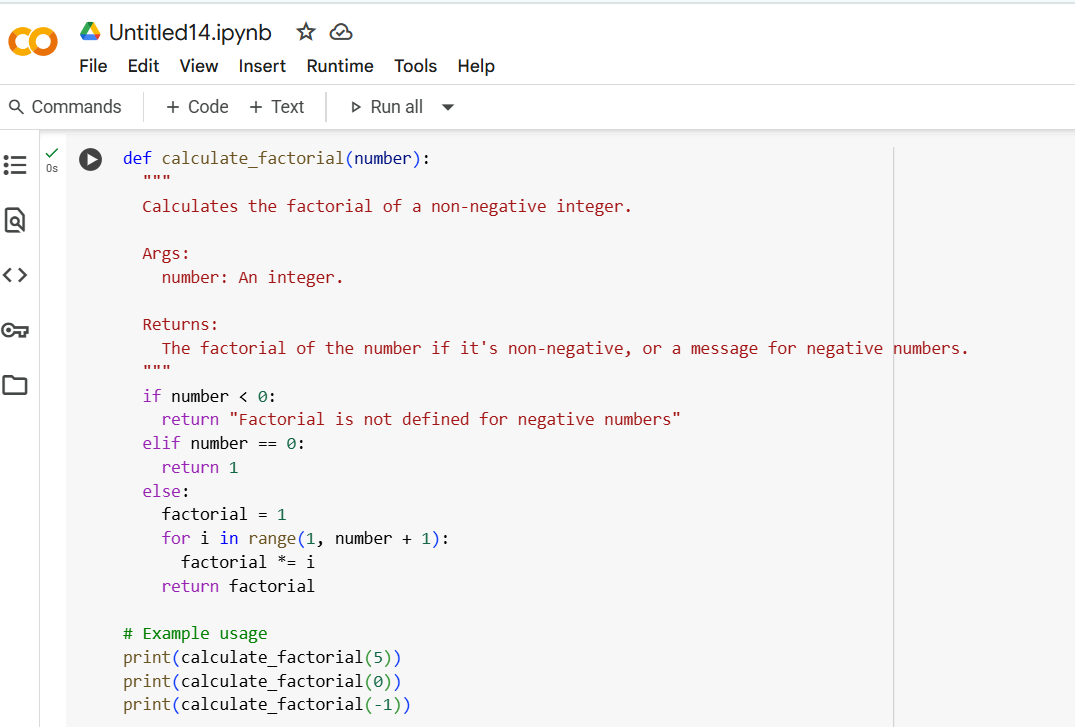
1. **Matching the Pattern Against the Input**  
   The function uses re.match() to compare the input number against the pattern. If the input matches the pattern, it means the number is valid according to Indian mobile number rules.
2. **Returning the Result**  
   If the match is successful, the function returns True, indicating a valid number. If not, it returns False, indicating the number is invalid.
3. **Testing the Function with Examples**  
   After defining the function, the code includes several print() statements to test it:

* "9876543210" returns True because it starts with 9 and has 10 digits.
* "5876543210" returns False because it starts with 5, which is not allowed.
* "987654321" returns False because it only has 9 digits.
* "98765432101" returns False because it has 11 digits.

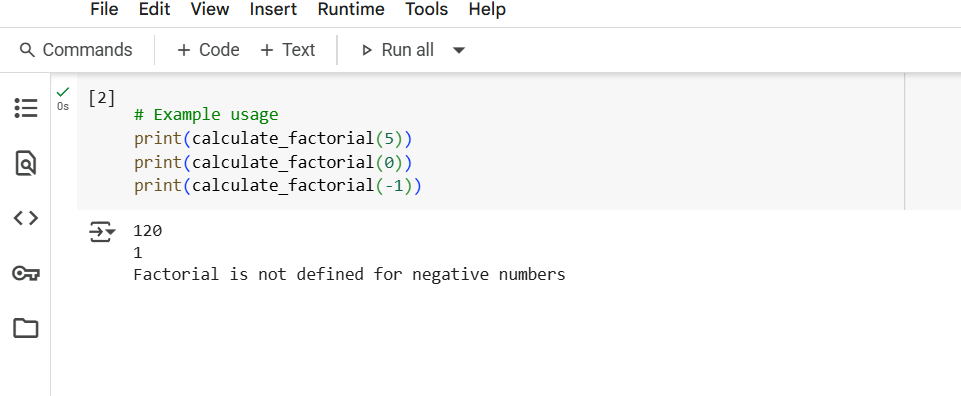
**#TASK 2:**

**Prompt:** Write a Python function that calculates the factorial of a number. The function should return 1 for an input of 0, and return a message like 'Factorial is not defined for negative numbers' for negative inputs. For example, if the input is 5, the output should be 120.

**Code:**

****

**Output:**

****

**Explanation:**

1. **Defining the Factorial Function**  
   The code defines a function named calculate\_factorial, which is designed to compute the factorial of a non-negative integer. Factorial is a mathematical operation where you multiply all whole numbers from 1 up to the given number (e.g., 5! = 5×4×3×2×1 = 120).
2. **Adding a Docstring for Clarity**  
   Inside the function, a docstring explains its purpose. It states that the function calculates the factorial of a non-negative integer, takes one argument called number, and returns either the factorial result or a message if the input is invalid.
3. **Handling Negative Inputs**  
   The first condition checks if the input number is less than zero. If it is, the function returns a message: "Factorial is not defined for negative numbers". This is because factorials are only defined for non-negative integers.
4. **Handling Zero as a Special Case**  
   The next condition checks if the number is zero. If so, the function returns 1, since by definition, the factorial of zero is always 1.
5. **5.Calculating the Factorial for Positive Numbers**  
   If the number is positive, the function initializes a variable called factorial to 1. It then uses a for loop to multiply factorial by every integer from 1 up to the input number. This loop builds the factorial step by step.
6. **Returning the Result**  
   After the loop finishes, the function returns the final value of factorial, which is the computed result.
7. **Testing the Function with Examples**  
   The code includes three example calls to the function:

* calculate\_factorial(5) returns 120, since 5! = 5×4×3×2×1.
* calculate\_factorial(0) returns 1, as expected.
* calculate\_factorial(-1) returns the message "Factorial is not defined for negative numbers" because the input is invalid.

**#TASK 3:**

**Prompt:**

Create a Python function that extracts the full name, branch, and SGPA from a nested dictionary containing student information. For example:

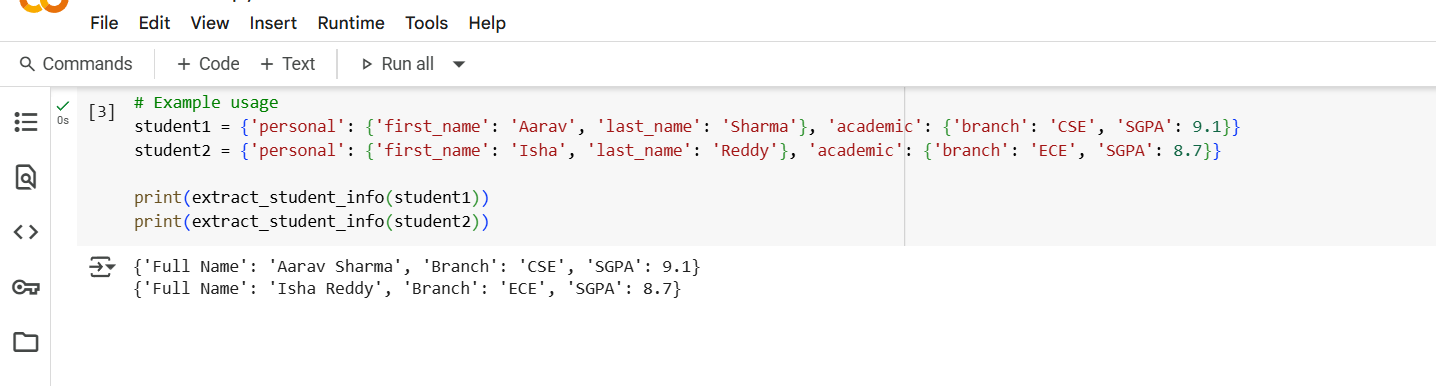
* If the input is  
  {'personal': {'first\_name': 'Aarav', 'last\_name': 'Sharma'}, 'academic': {'branch': 'CSE', 'SGPA': 9.1}}  
  the output should be  
  {'Full Name': 'Aarav Sharma', 'Branch': 'CSE', 'SGPA': 9.1}
* If the input is  
  {'personal': {'first\_name': 'Isha', 'last\_name': 'Reddy'}, 'academic': {'branch': 'ECE', 'SGPA': 8.7}}  
  the output should be  
  {'Full Name': 'Isha Reddy', 'Branch': 'ECE', 'SGPA': 8.7}

Write a reusable function that works for any student dictionary structured like these examples.

**Code:**

****

**Output:**

****

**Explanation:**

1. **Defining the Function**  
   The code defines a function called extract\_student\_info, which is designed to pull specific details—namely the full name, branch, and SGPA—from a nested dictionary that contains student information.
2. **Adding a Docstring for Clarity**  
   Inside the function, a docstring explains its purpose. It states that the function extracts the full name, branch, and SGPA from a dictionary structured with two main keys: 'personal' and 'academic'. It also clarifies that the function returns a simplified dictionary containing just those three pieces of information.
3. **Extracting the Full Name**  
   The function first constructs the student’s full name by accessing the 'first\_name' and 'last\_name' fields inside the 'personal' section of the dictionary. It uses an f-string to combine them into a single string called full\_name.
4. **Extracting the Branch**  
   Next, it retrieves the student’s branch by accessing the 'branch' field inside the 'academic' section of the dictionary. This value is stored in the variable branch.
5. **Extracting the SGPA**  
   Similarly, the function pulls the student’s SGPA from the 'SGPA' field within the 'academic' section and stores it in the variable sgpa.
6. **Returning the Extracted Information**  
   After gathering all three pieces of data, the function returns a new dictionary containing the keys 'full\_name', 'branch', and 'SGPA', with their corresponding values.
7. **Testing the Function with Examples**  
   The code then creates two sample student dictionaries: one for a student named Aarav Sharma in the CSE branch with an SGPA of 9.1, and another for Isha Reddy in the ECE branch with an SGPA of 8.7. It calls the extract\_student\_info function on each dictionary and prints the results, showing how the function simplifies the nested data into a clean summary.

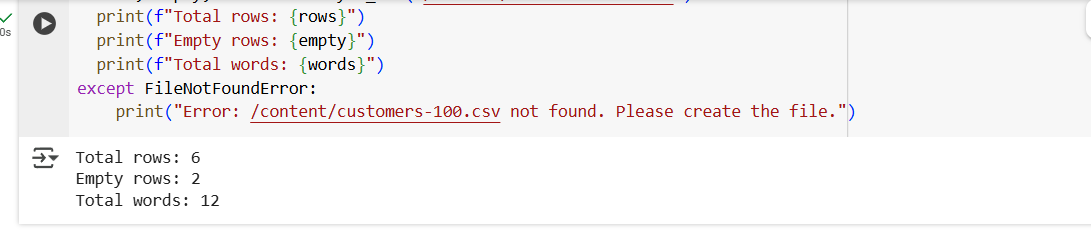
**#Task 4:**

**Zero shot:  
prompt:** Write a Python function that reads a .csv file and returns the total number of rows, the number of empty rows, and the total number of words across all cells in the file.

**Code:**

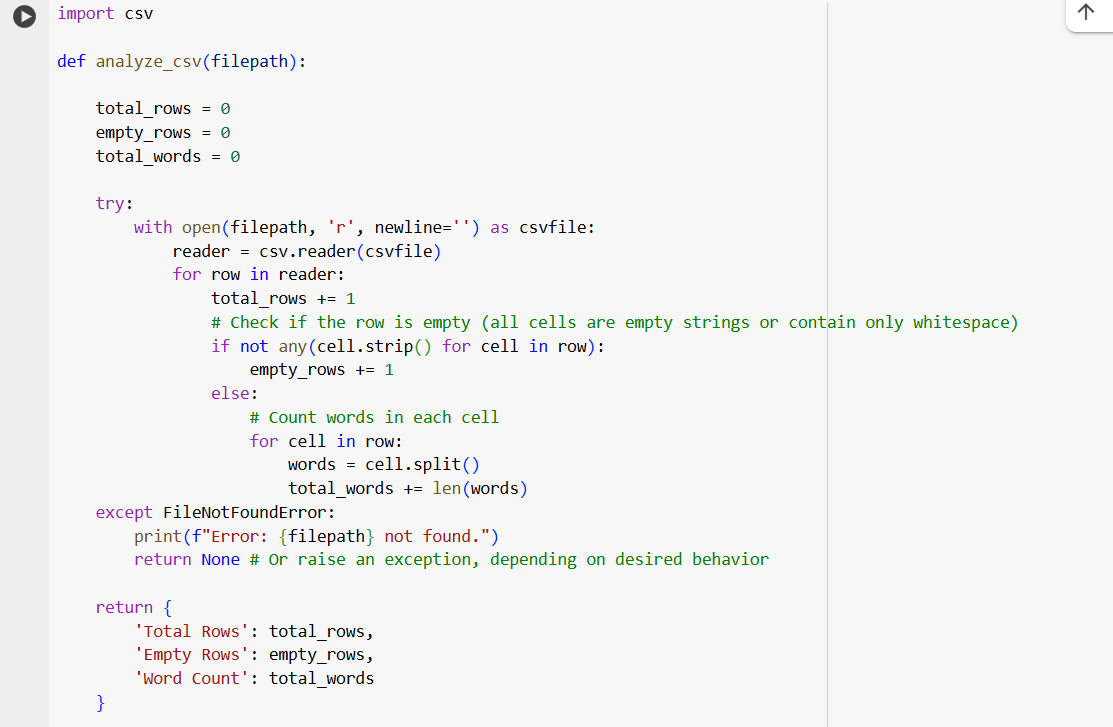
****

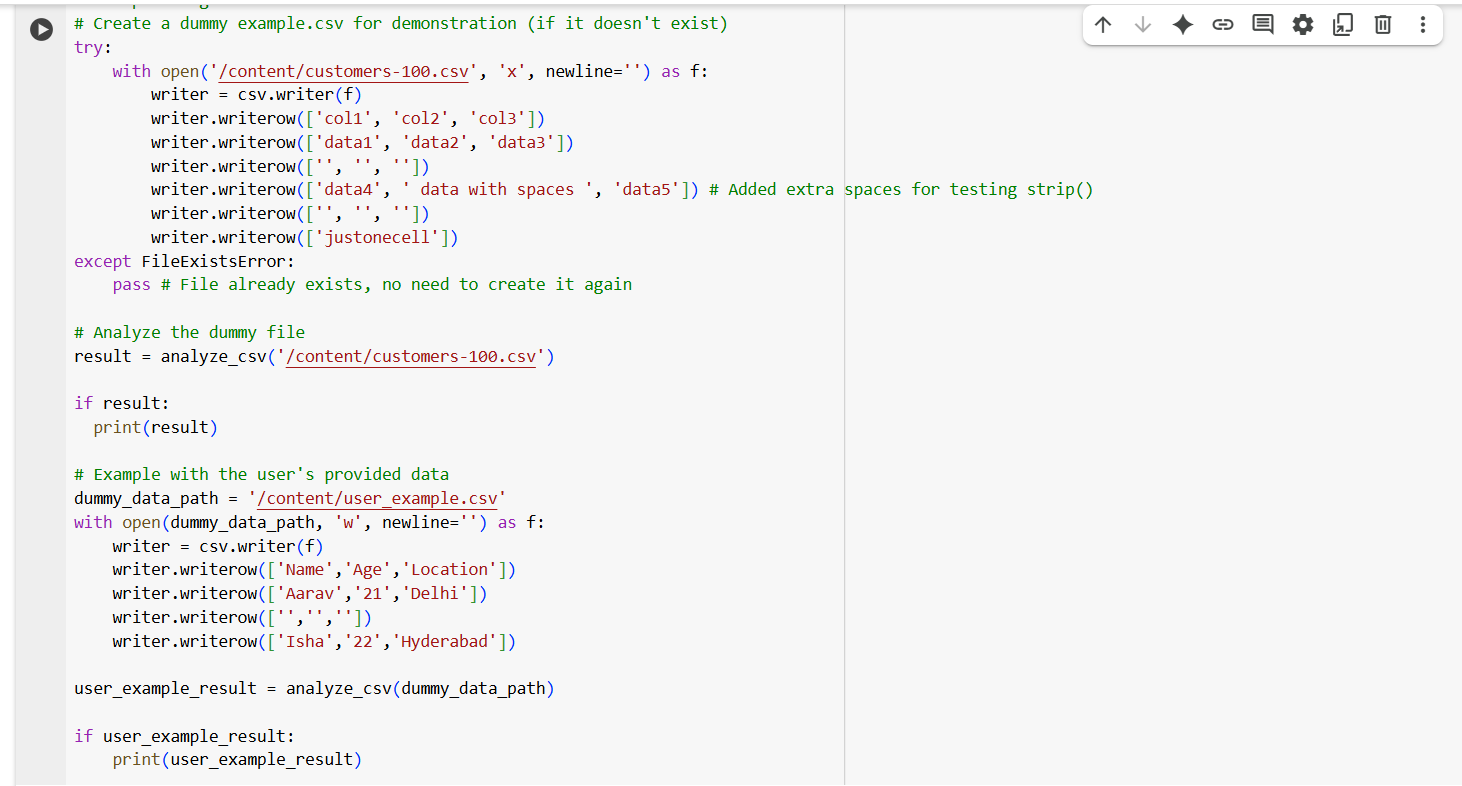
**Output:**

****

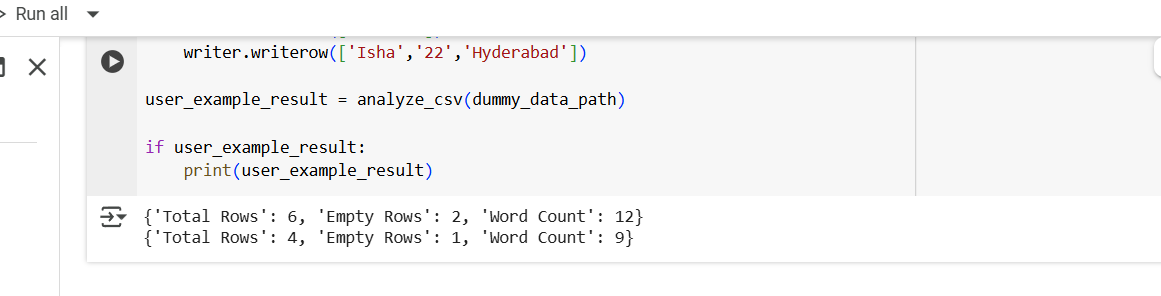
**One shot:  
prompt:**

Write a Python function that reads a .csv file and returns the total number of rows, the number of empty rows, and the total number of words across all cells. For example, if the file contains:Name, Age, Location  
Aarav,21, Delhi  
,,  
Isha, 22, Hyderabad  
then the function should return:  
{'Total Rows': 4, 'Empty Rows': 1, 'Word Count': 6}.

**Code:  
**

****

**Output:**

****

**Few shot:**

**Prompt:**

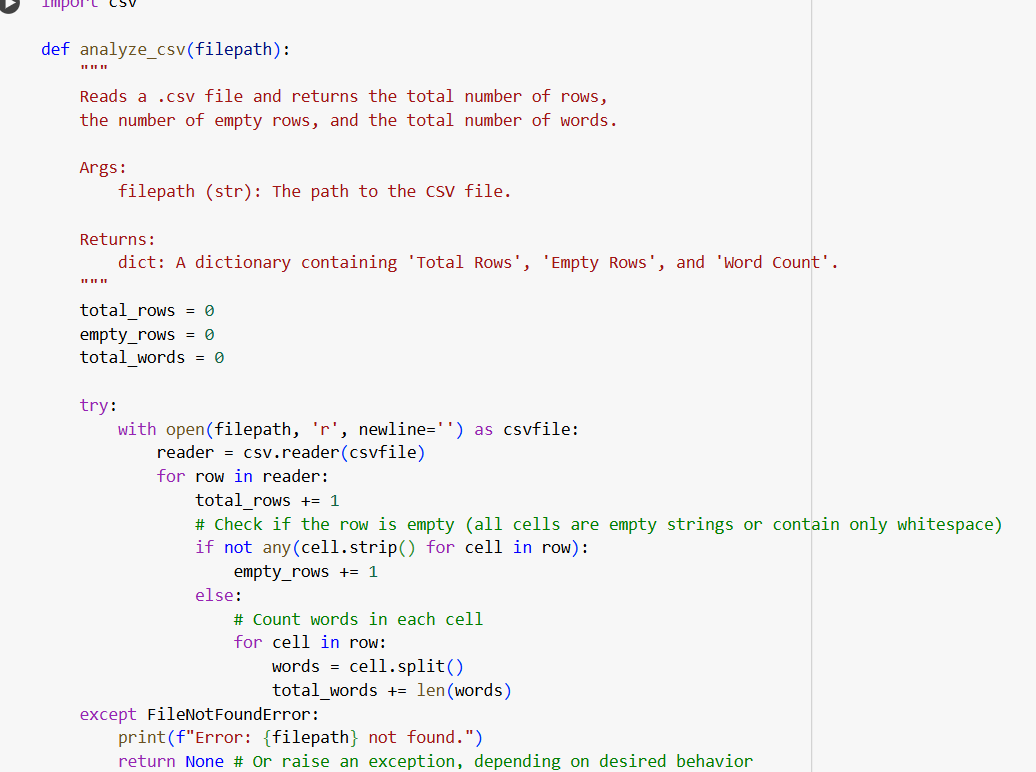
Write a Python function that reads a .csv file and returns the total number of rows, the number of empty rows, and the total number of words across all cells.

For example:  
If the file contains:  
Name,Age,Location  
Aarav,21,Delhi  
,,  
Isha,22,Hyderabad  
the function should return:  
{'Total Rows': 4, 'Empty Rows': 1, 'Word Count': 6}

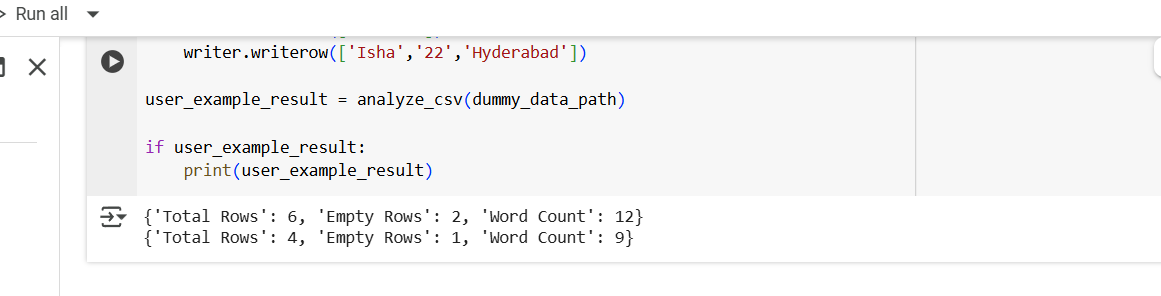
Another example:  
If the file contains:  
Product,Price,Stock  
Laptop,50000,10  
Phone,30000,  
,,  
Tablet,25000,5  
the function should return:  
{'Total Rows': 5, 'Empty Rows': 1, 'Word Count': 9}

Write a reusable function that works for any CSV file structured like these examples.

**Code:**

****

**Output:**

****

**Explanation:**

1. **Importing the CSV Module**  
   The code starts by importing Python’s built-in csv module, which is used to read and process CSV (Comma-Separated Values) files—a common format for storing structured data like spreadsheets.
2. **Defining the Analysis Function**  
   A function named analyze\_csv is defined, which takes a single argument called filepath. This argument represents the location of the CSV file to be analyzed.
3. **Initializing Counters**  
   Inside the function, three variables are initialized: total\_rows to count all rows in the file, empty\_rows to count rows that contain no meaningful data, and total\_words to count all words across the file.
4. **Opening the CSV File**  
   The function opens the file using a with statement, which ensures the file is properly closed after reading. It uses open(filepath, 'r', newline='') to read the file and creates a CSV reader object to iterate through its rows.
5. **Processing Each Row**  
   The function enters a loop that goes through each row in the CSV file. For every row encountered, it increments the total\_rows counter.
6. **Identifying Empty Rows**  
   It checks whether a row is empty by using not any(cell.strip() for cell in row). This condition ensures that all cells in the row are either blank or contain only whitespace. If the row is empty, empty\_rows is incremented.
7. **Counting Words in Non-Empty Rows**  
   If the row contains data, the function loops through each cell in the row. It splits the cell’s content into words using cell.split() and adds the number of words to the total\_words counter.
8. **Returning the Results**  
   After processing all rows, the function returns a tuple containing three values: the total number of rows, the number of empty rows, and the total word count.
9. **Executing the Function and Handling Errors**  
   Outside the function, the script calls analyze\_csv with the path '/content/customers-100.csv'. It stores the returned values in rows, empty, and words, and prints them. If the file is not found, the script catches the FileNotFoundError and prints an error message asking the user to create the file.

**#Task 5:**

**Prompt:**

Write a Python function that takes a paragraph of text as input, converts all text to lowercase, removes punctuation, and returns the most frequently used word.

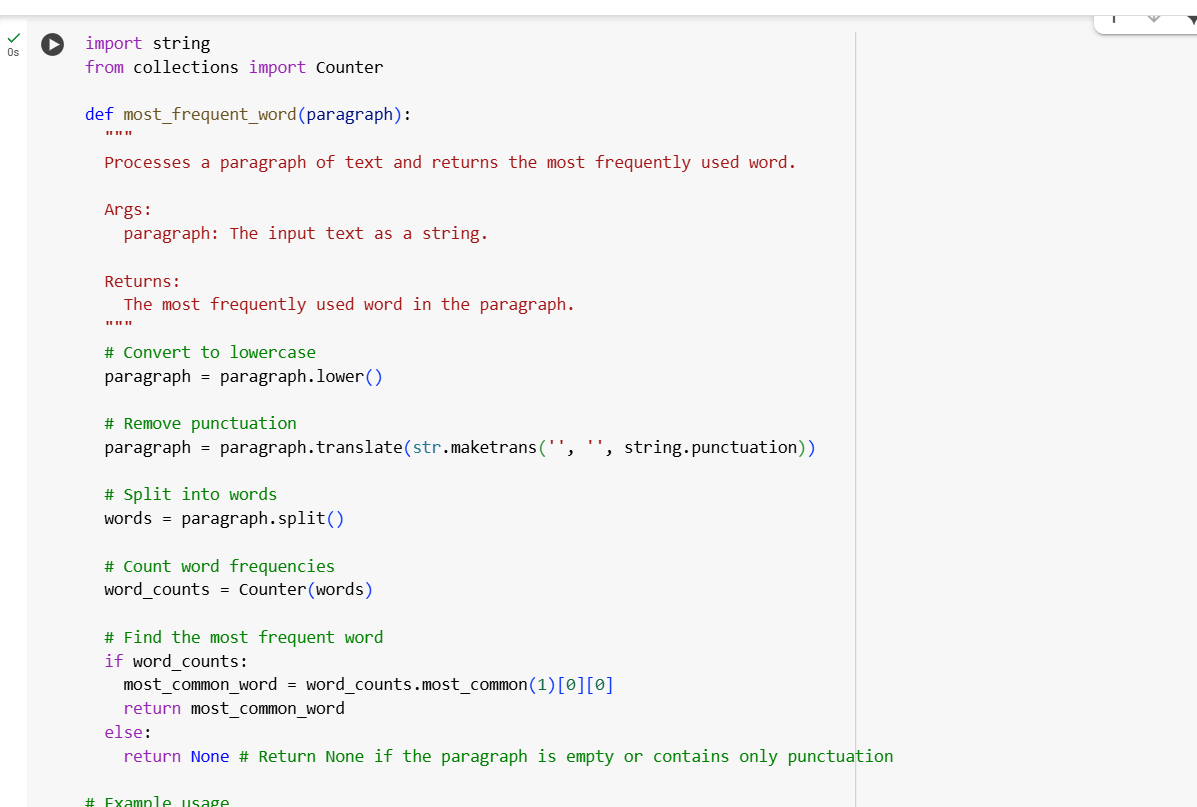
Here are some examples:\*\*

**Example 1:**  
Input: "Python is great. Python is easy to learn!"  
Output: "python"

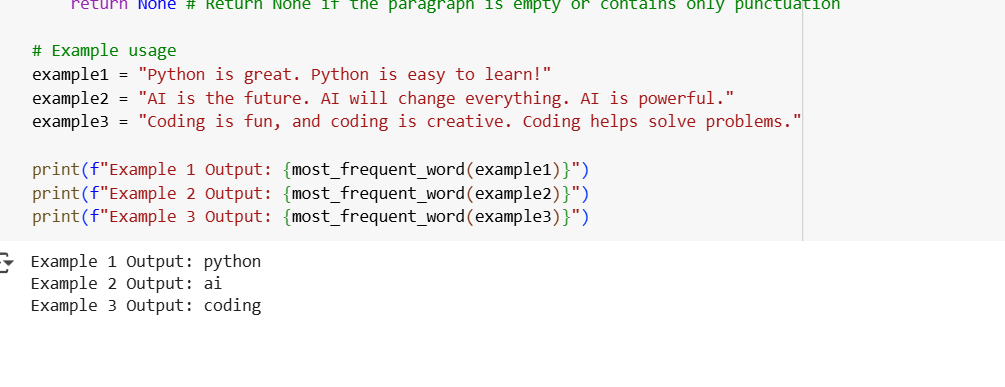
**Example 2:**  
Input: "AI is the future. AI will change everything. AI is powerful."  
Output: "ai"

**Example 3:**  
Input: "Coding is fun, and coding is creative. Coding helps solve problems."  
Output: "coding"

**Code:**

****

**Output:**

****

**Explanation:**

1. **Importing Required Modules**  
   The code begins by importing two modules: string and Counter from the collections module. The string module provides access to common string constants like punctuation marks, while Counter is a specialized dictionary used to count occurrences of items—in this case, words.
2. **Defining the Function**  
   A function named most\_frequent\_word is defined. It takes a single argument called paragraph, which is expected to be a string of text.
3. **Adding a Docstring for Clarity**  
   Inside the function, a docstring explains its purpose. It states that the function processes a paragraph of text and returns the most frequently used word. It also documents the input and output clearly.
4. **Converting Text to Lowercase**  
   The function first converts the entire paragraph to lowercase using paragraph.lower(). This ensures that words like “The” and “the” are treated as the same word during frequency analysis.
5. **Removing Punctuation**  
   Next, the function removes all punctuation from the paragraph using paragraph.translate(str.maketrans('', '', string.punctuation)). This step helps prevent punctuation marks from being attached to words, which could distort the word count.
6. **Splitting the Text into Words**  
   The cleaned paragraph is then split into individual words using the split() method. This creates a list of words that can be analyzed.
7. **Counting Word Frequencies**  
   The function uses Counter(words) to count how many times each word appears in the list. This creates a dictionary-like object where each word is a key and its frequency is the value.
8. **Finding the Most Frequent Word**  
   The function checks if the word\_counts object is not empty. If it contains words, it uses word\_counts.most\_common(1)[0][0] to retrieve the word with the highest frequency. This is then returned as the result.
9. **Handling Empty Input**  
   If the paragraph is empty or contains only punctuation, the function returns None to indicate that no valid word was found.