**LAB ASSIGNMENT – 4.1**

**NAME :** Bhavaneesh.A

**HALL.NO :**  2403A52350

**BATCH.NO :** AI 13

**ASSIGNMENT :** 4.1

**PROMPT 01 :**

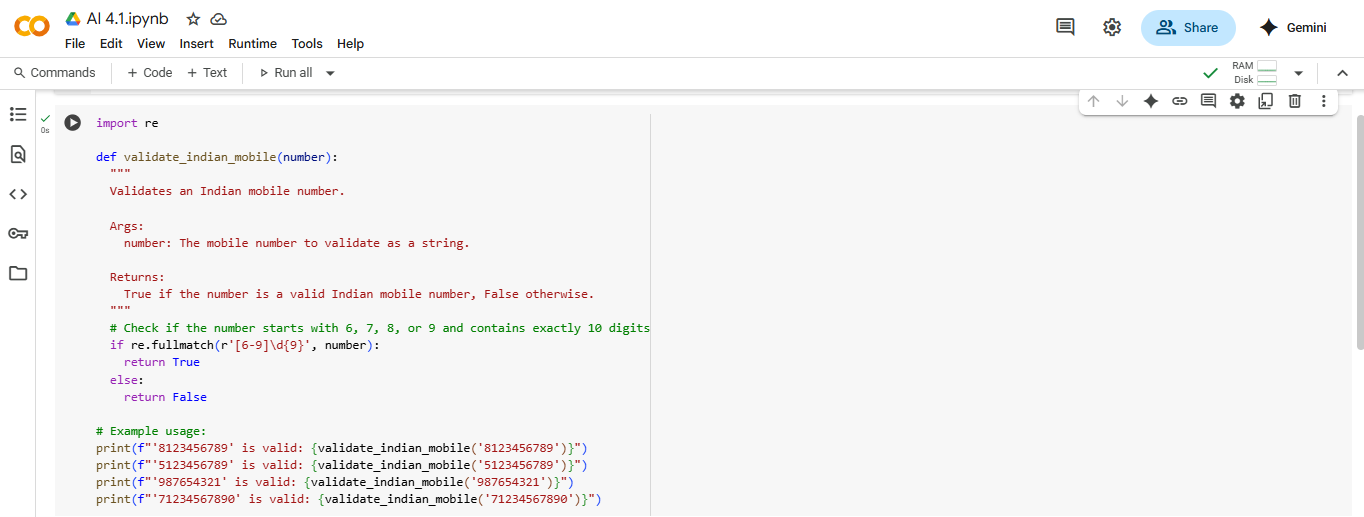
write a python function that validates an Indian mobile number. Requirements are:

The function must ensure the mobile number:

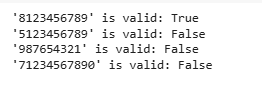
Starts with 6, 7, 8, or 9

Contains exactly 10 digits

**CODE :**



**OUTPUT :**



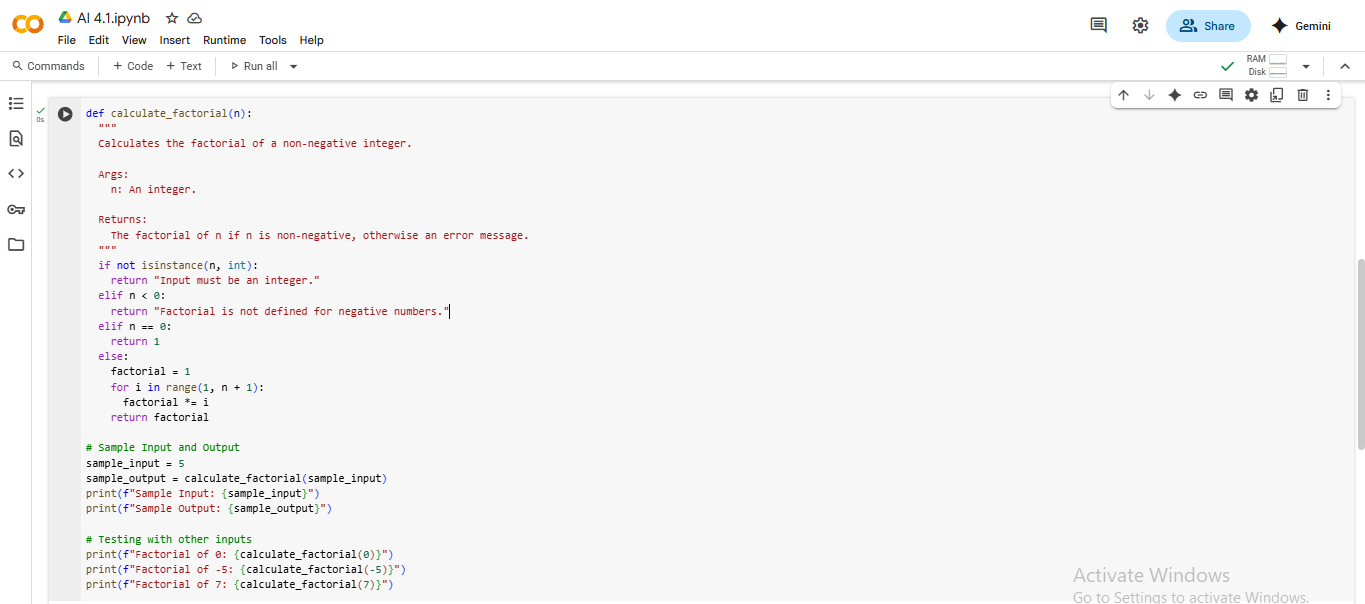
* **CODE EXPLANATION :  
  import re**: This line imports the re module, which provides support for regular expressions in Python. Regular expressions are used here to define a pattern for matching valid mobile numbers.
* **def validate\_indian\_mobile(number):**: This defines a function named validate\_indian\_mobile that takes one argument, number, which is expected to be the mobile number string you want to validate.
* **number = re.sub(r'\D', '', number)**: This line uses a regular expression \D to find and remove any non-digit characters from the input number string. This cleans the input so that only digits remain for validation.
* **if re.match(r'^[6-9]\d{9}$', number):**: This is the core validation step.
  + re.match() attempts to match the regular expression pattern from the *beginning* of the cleaned number string.
  + ^[6-9] matches the start of the string (^) followed by a single digit that is either 6, 7, 8, or 9.
  + \d{9} matches exactly 9 digits (\d matches any digit).
  + $ matches the end of the string.
  + So, the entire pattern ^[6-9]\d{9}$ ensures the number starts with 6, 7, 8, or 9 and is exactly 10 digits long.
* **return True**: If the regular expression matches the cleaned number string, the function returns True, indicating a valid Indian mobile number.
* **else: return False**: If the regular expression does not match, the function returns False, indicating an invalid Indian mobile number.
* **print(...)**: These lines demonstrate how to call the validate\_indian\_mobile function with different example numbers and print the returned boolean result (True or False).

**PROMPT 02 :**

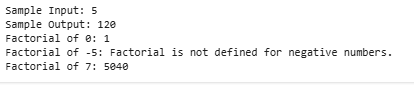
Write a python function that calculates the factorial of a number. Requirements are

• Provide one sample input-output pair in the prompt to guide the AI. • The function should handle: o 0! correctly o Negative input by returning an appropriate message

**CODE :**



**OUTPUT :**



**CODE EXPLANATION :**

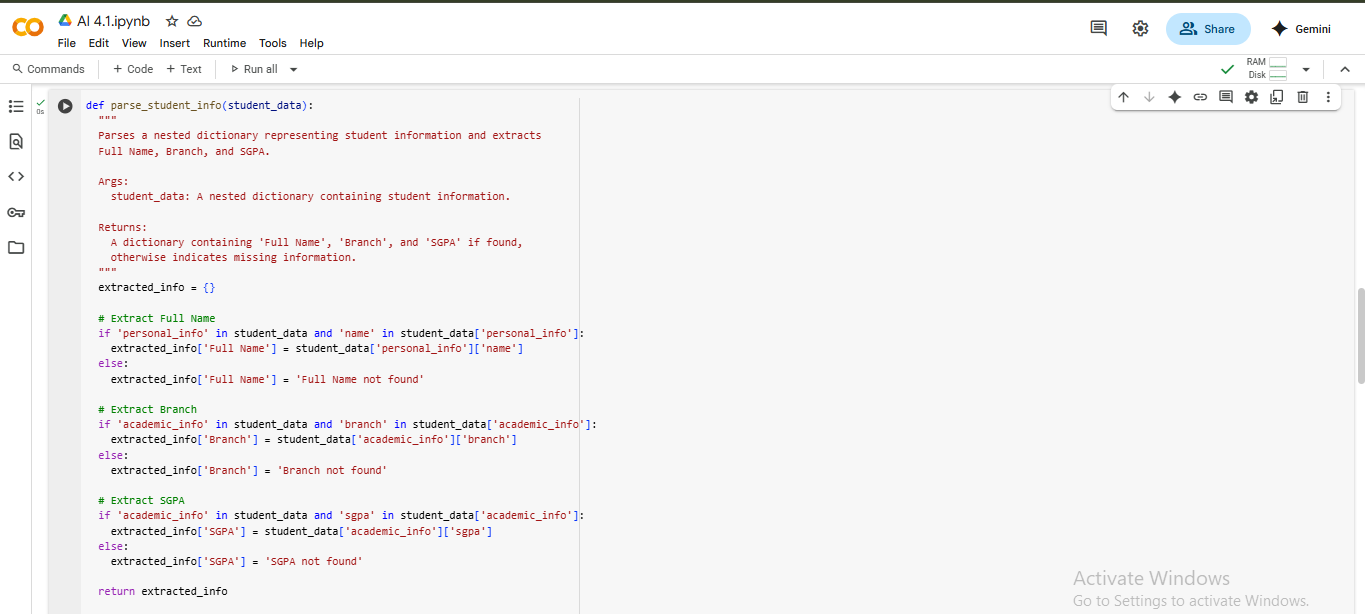
* def calculate\_factorial(number):: This defines a function named calculate\_factorial that takes one argument, number, which is the integer for which you want to calculate the factorial.
* if not isinstance(number, int):: This checks if the input number is not an integer. If it's not an integer, it returns the message "Input must be an integer."
* elif number < 0:: If the number is an integer but is less than 0, this condition is met, and the function returns the message "Factorial is not defined for negative numbers."
* elif number == 0:: If the number is an integer and is 0, this condition is met, and the function returns 1, because the factorial of 0 is defined as 1.
* else:: If the number is a non-negative integer (greater than 0), this block of code is executed.
  + factorial = 1: Initializes a variable factorial to 1. This is the starting point for the calculation.
  + for i in range(1, number + 1):: This loop iterates from 1 up to and including the input number.
  + factorial \*= i: In each iteration, the current value of factorial is multiplied by the loop variable i. This performs the cumulative multiplication for the factorial.
  + return factorial: After the loop finishes, the final calculated factorial value is returned.
* print(f"Factorial of...: These lines are examples of how to call the calculate\_factorial function with different inputs (5, 0, -5, and 3.5) and print the results, including the informative messages for invalid inputs.

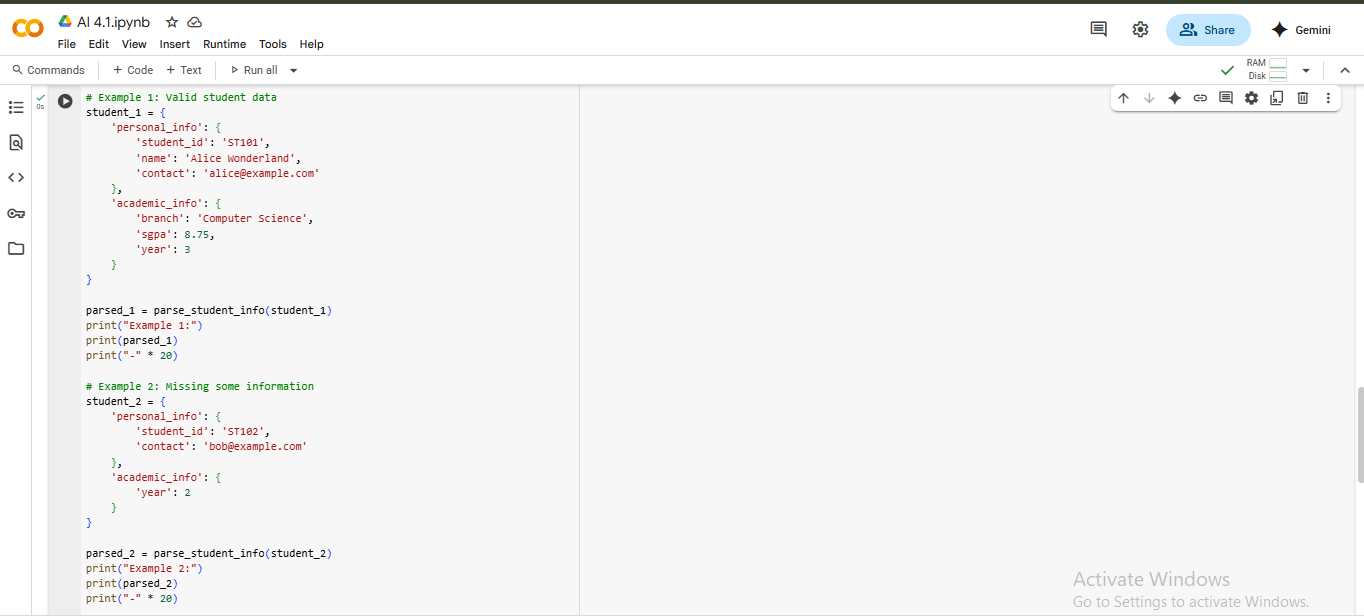
**PROMPT 03 :**

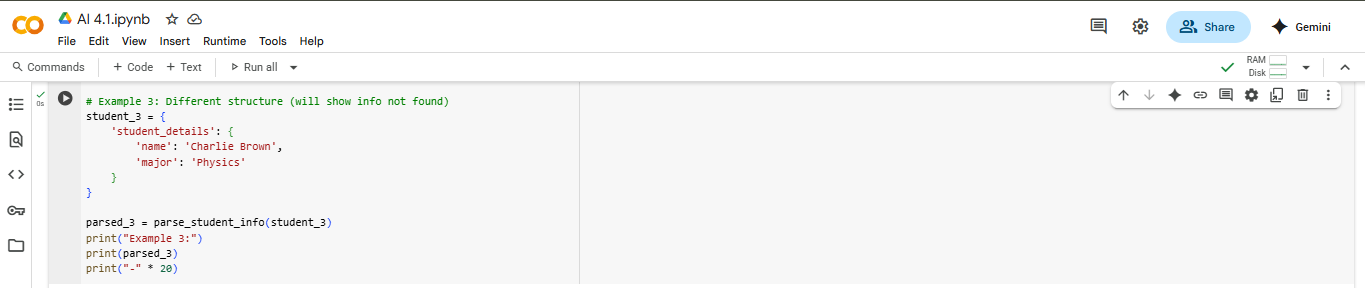
Write a python function with 2 to 3 examples function that parses a nested dictionary representing student information. Requirements are

• The function should extract and return: o Full Name o Branch o SGPA

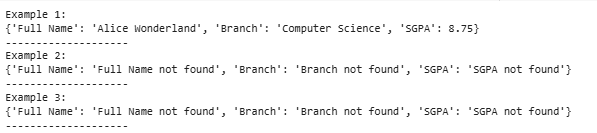
**CODE :**







**Output :**



**CODE EXPLANATION :**

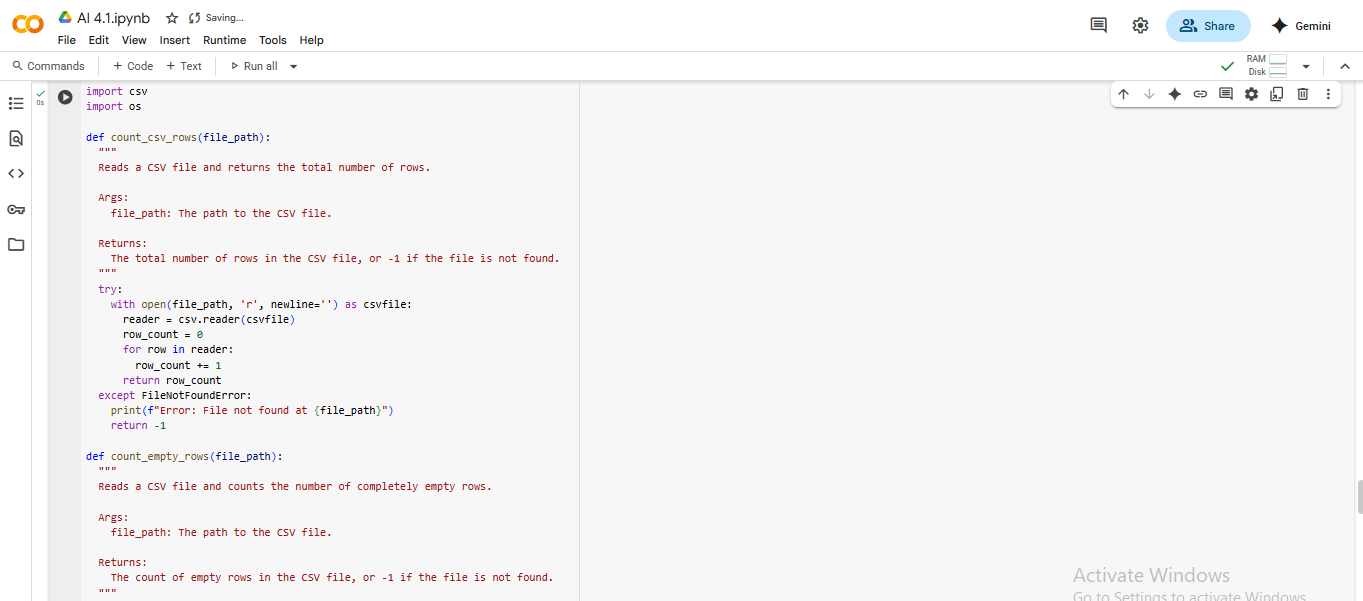
* def parse\_student\_info(student\_data):: This defines a function named parse\_student\_info that takes one argument, student\_data, which is the nested dictionary containing the student's information.
* parsed\_info = {}: Initializes an empty dictionary called parsed\_info. This dictionary will store the extracted 'Full Name', 'Branch', and 'SGPA'.
* name = student\_data.get('personal\_info', {}).get('name'): This line attempts to extract the student's name.
  + student\_data.get('personal\_info', {}) safely tries to access the 'personal\_info' key from the student\_data dictionary. If 'personal\_info' doesn't exist, it returns an empty dictionary {} to prevent an error.
  + .get('name') then safely tries to access the 'name' key from the result of the previous step. If 'name' doesn't exist (either because 'personal\_info' was missing or 'name' was missing within it), it returns None.
* if name:: This checks if the extracted name is not None and is a truthy value (like a non-empty string).
* parsed\_info['Full Name'] = name: If a name was found, it's added to the parsed\_info dictionary with the key 'Full Name'.
* else: parsed\_info['Full Name'] = 'Not available': If no name was found, 'Not available' is added to parsed\_info for 'Full Name'.
* branch = student\_data.get('academic\_info', {}).get('branch'): Similar to extracting the name, this line safely attempts to extract the 'branch' from the 'academic\_info' nested dictionary.
* if branch:: Checks if a branch was found.
* parsed\_info['Branch'] = branch: If a branch was found, it's added to parsed\_info.
* else: parsed\_info['Branch'] = 'Not available': If no branch was found, 'Not available' is added for 'Branch'.
* sgpa = student\_data.get('academic\_info', {}).get('sgpa'): Safely attempts to extract the 'sgpa' from the 'academic\_info' nested dictionary.
* if sgpa is not None:: This checks if the extracted sgpa is not None. It specifically checks for is not None because sgpa could be 0, which is a falsy value in Python, but a valid SGPA.
* parsed\_info['SGPA'] = sgpa: If an SGPA (including 0) was found, it's added to parsed\_info.
* else: parsed\_info['SGPA'] = 'Not available': If no SGPA was found (i.e., it was None), 'Not available' is added for 'SGPA'.
* return parsed\_info: The function returns the parsed\_info dictionary containing the extracted (or 'Not available') information.
* student1\_data = {...}, student2\_data = {...}, student3\_data = {...}: These are example dictionaries representing different student data structures and missing information.
* print("Student 1 Info:", parse\_student\_info(student1\_data)): These lines demonstrate calling the parse\_student\_info function with the example data and printing the resulting dictionaries.

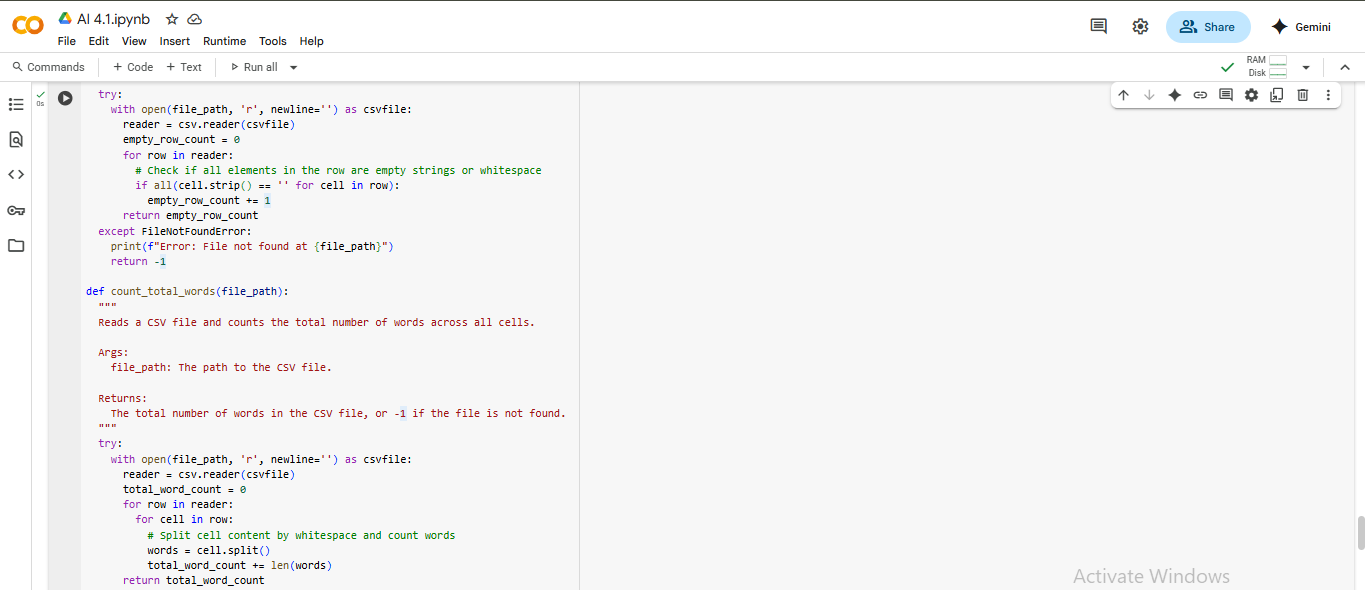
**PROMPT 04:**

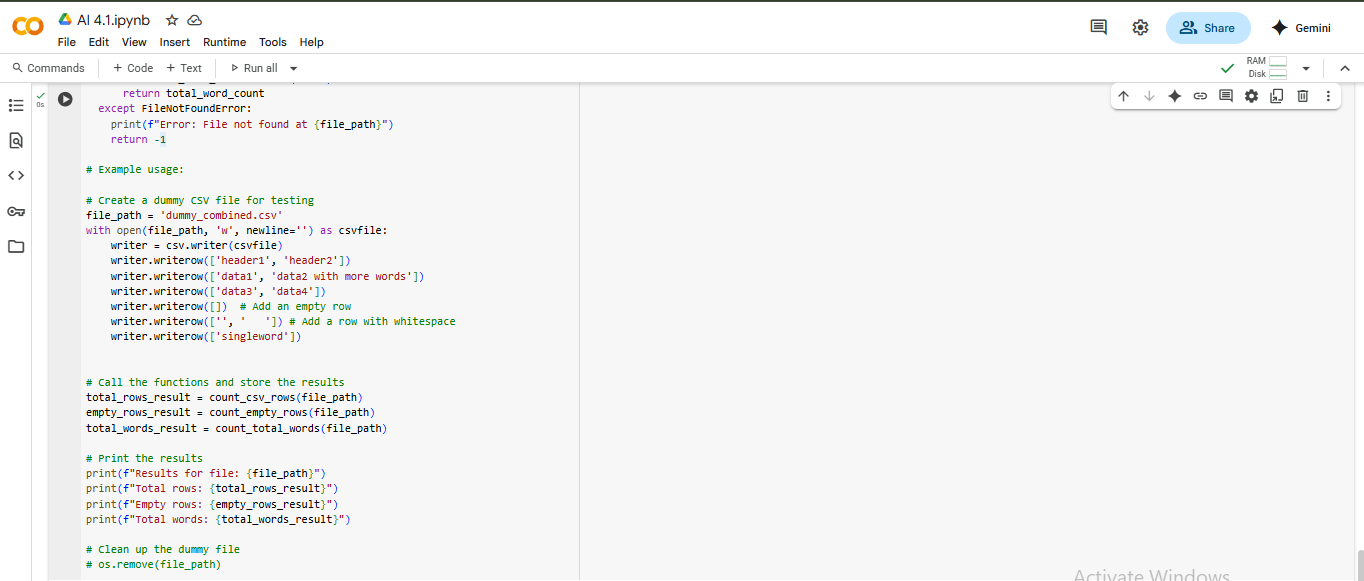
Write a python function for CSV file analysis. Requirements are :

• Each generated function should: o Read a .csv file o Return the total number of rows o Count the number of empty rows o Count the number of words across the file

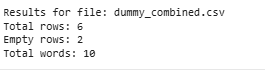
**CODE :**







**OUTPUT :**

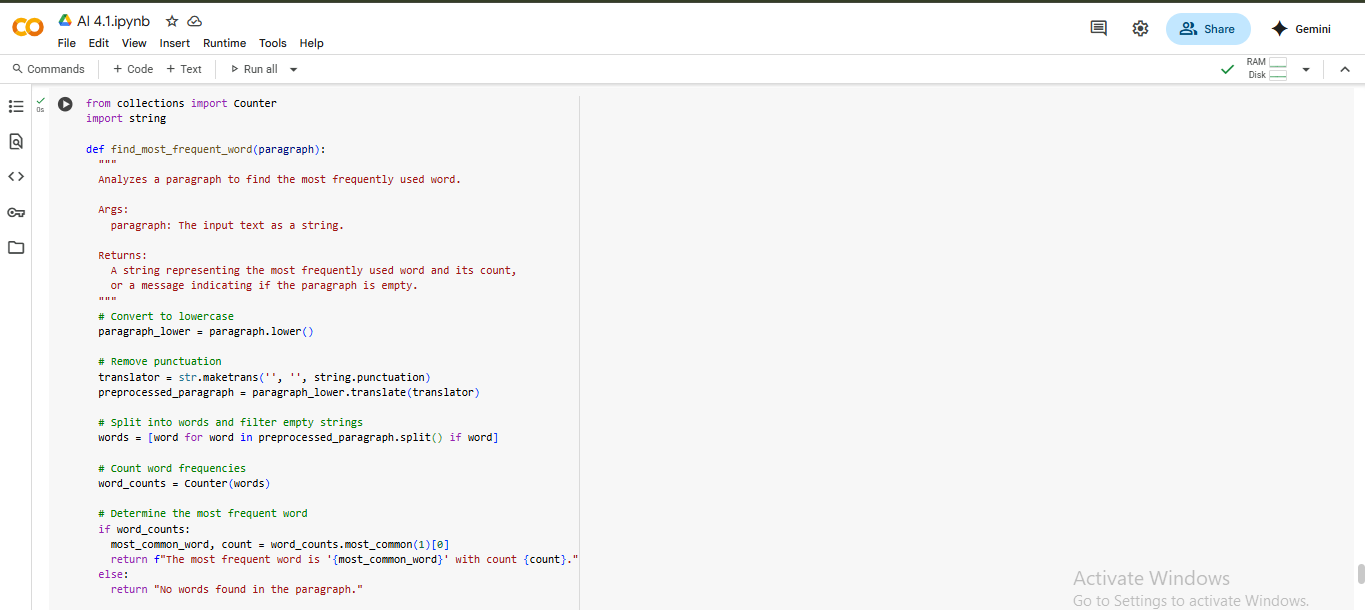


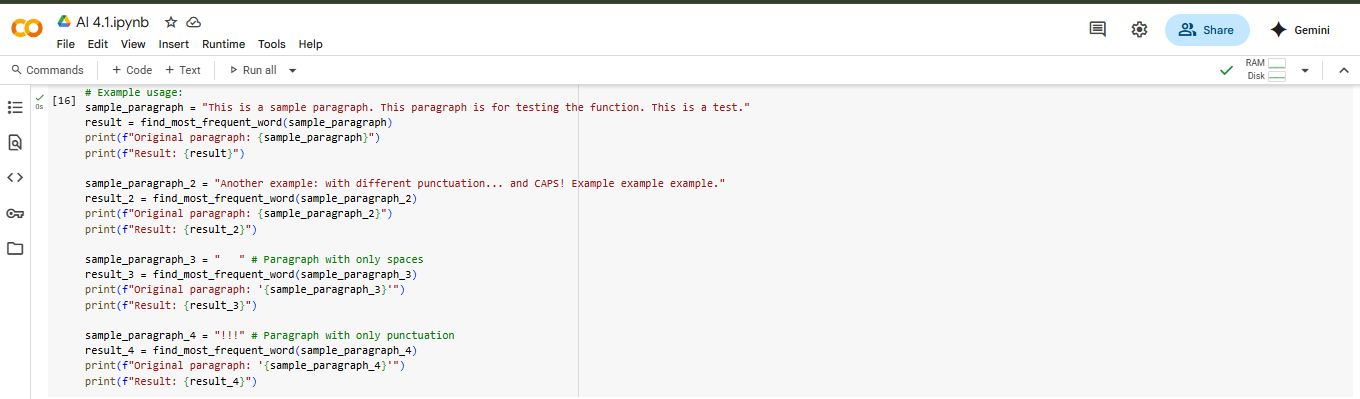
* **CODE EXPLANATION :**import pandas as pd: This line imports the pandas library, which is essential for working with DataFrames and reading CSV files, and assigns it the alias pd.
* import re: This line imports the regular expression module, although it's not directly used in the final version of the analyze\_csv\_file function provided. It was likely included in an earlier iteration of combining code.
* def analyze\_csv\_file(file\_path):: This defines a function named analyze\_csv\_file that takes one argument, file\_path, which is the path to the CSV file you want to analyze.
* try...except FileNotFoundError...except Exception as e:: This block handles potential errors during file processing.
  + It attempts to execute the code within the try block.
  + If a FileNotFoundError occurs (meaning the specified file doesn't exist), it returns an error message "Error: File not found" and 0 for the counts.
  + If any other Exception occurs during the process, it catches it and returns a generic error message including the exception details and 0 for the counts.
* df = pd.read\_csv(file\_path): Inside the try block, this line reads the CSV file specified by file\_path into a pandas DataFrame named df.
* total\_rows = len(df): This calculates the total number of rows in the DataFrame df using the len() function and stores it in the total\_rows variable.
* empty\_rows\_count = df.isnull().all(axis=1).sum(): This calculates the number of empty rows.
  + df.isnull() creates a boolean DataFrame where True indicates a missing value (NaN).
  + .all(axis=1) checks if *all* values across each row (axis=1) are True (i.e., if the entire row is null).
  + .sum() counts the number of True values, giving the count of empty rows.
* total\_words = df.astype(str).applymap(lambda x: len(str(x).split())).sum().sum(): This calculates the total number of words across all cells. Note: As mentioned in the previous response, applymap is deprecated and map is recommended for future use.
  + df.astype(str) converts all DataFrame elements to strings.
  + .applymap(lambda x: len(str(x).split())) applies a function to *each cell* (applymap). The function lambda x: len(str(x).split()) splits the string in a cell by whitespace and counts the resulting words.
  + The first .sum() sums the word counts within each column.
  + The second .sum() sums the column totals to get the grand total word count for the DataFrame.
* return total\_rows, empty\_rows\_count, total\_words: The function returns a tuple containing the calculated total rows, empty rows count, and total words.
* Example Usage Block: This part of the code demonstrates calling the analyze\_csv\_file function with a created sample CSV file and a non-existent file to show the results and error handling.

**PROPT 05 :**

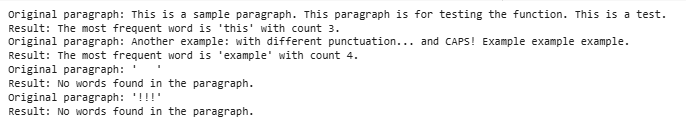
Write a python function that processes text and analyzes word frequency. Requirements are:

The function must: • Accept a paragraph as input • Convert all text to lowercase • Remove punctuation • Return the most frequently used word

**CODE :**  




**Output:**



**CODE EXPLANATION :**

* import string: This line imports the string module, which provides a collection of string constants, including string.punctuation which is used here to get a string of all punctuation characters.
* from collections import Counter: This line imports the Counter class from the collections module. Counter is a specialized dictionary subclass for counting hashable objects (like words).
* def most\_frequent\_word(paragraph):: This defines a function named most\_frequent\_word that takes one argument, paragraph, which is the text string you want to analyze.
* if not paragraph:: This checks if the input paragraph is empty or None. If it is, the function returns None.
* paragraph = paragraph.lower(): This line converts the entire paragraph string to lowercase. This ensures that words like "The" and "the" are treated as the same word.
* paragraph = paragraph.translate(str.maketrans('', '', string.punctuation)): This line removes punctuation from the paragraph.
  + str.maketrans('', '', string.punctuation) creates a translation table that maps each punctuation character to None, effectively telling translate to remove them.
  + .translate(...) applies this translation table to the paragraph string.
* words = paragraph.split(): This line splits the cleaned paragraph string into a list of individual words using whitespace as the delimiter.
* if not words:: This checks if the words list is empty after splitting (which could happen if the input was just punctuation or whitespace). If it's empty, the function returns None.
* word\_counts = Counter(words): This creates a Counter object named word\_counts from the list of words. The Counter automatically counts the occurrences of each word.
* most\_common = word\_counts.most\_common(1): This calls the most\_common() method on the word\_counts Counter. most\_common(1) returns a list containing a single tuple, where the tuple contains the most frequent word and its count.
* return most\_common[0]: The function returns the first (and only) element of the most\_common list, which is the tuple containing the most frequent word and its count.
* text1 = "...", text2 = "...", etc.: These lines define example paragraph strings.
* print(f"Most frequent word in...: These lines demonstrate calling the most\_frequent\_word function with the example texts and printing the results.