**ASSIGNMENT – 4.1**

**NAME : SAI CHARAN .P**

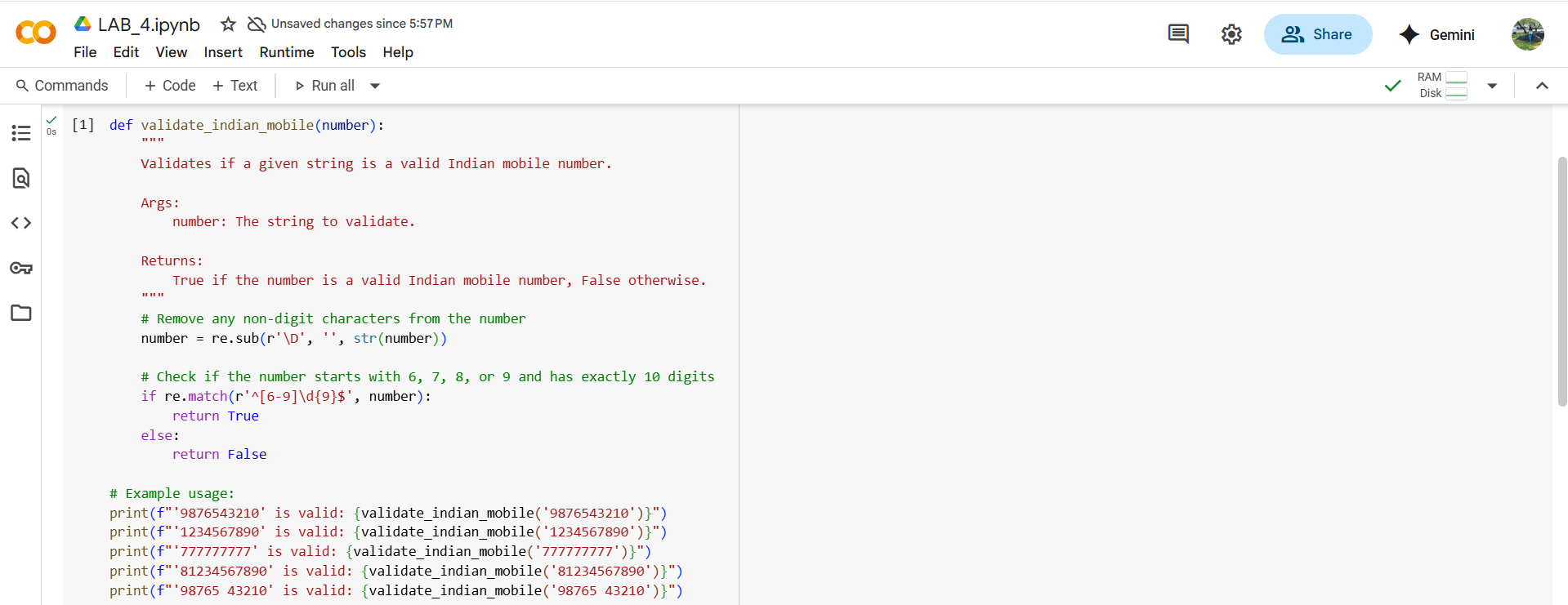
**ROLL:NO : 2403A52343**

**BATCH : AI 13**

**Prompt-1:** 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:**

**A white background with black text

AI-generated content may be incorrect.**

**Explanation:**

This code defines a Python function called validate\_indian\_mobile that checks if a given string is a valid Indian mobile number according to your requirements.

Here's a breakdown:

1. **import re**: This line imports the re module, which provides support for regular expressions. Regular expressions are used for pattern matching in strings.
2. **def validate\_indian\_mobile(number):**: This defines the function named validate\_indian\_mobile that takes one argument, number.
3. **number = re.sub(r'\D', '', str(number))**: This line cleans the input number.
   * str(number) ensures the input is treated as a string.
   * re.sub(r'\D', '', ...) uses a regular expression to replace all non-digit characters (\D) in the string with an empty string (''). This removes spaces, dashes, or any other non-numeric characters that might be present in the input.
4. **if re.match(r'^[6-9]\d{9}$', number):**: This is the core validation step.
   * re.match() attempts to match a pattern at the beginning of the string.
   * r'^[6-9]\d{9}$' is the regular expression pattern:
     + ^: Matches the beginning of the string.
     + [6-9]: Matches a single digit that is 6, 7, 8, or 9.
     + \d{9}: Matches exactly nine digits (\d matches any digit, {9} means exactly 9 times).
     + $: Matches the end of the string.
   * So, this pattern checks if the cleaned number starts with a digit from 6 to 9 and is followed by exactly nine more digits, making a total of 10 digits.
5. **return True**: If the re.match() finds a match, the function returns True, indicating that the number is valid.
6. **else: return False**: If the re.match() does not find a match, the function returns False, indicating that the number is not valid.

The lines after the function definition are examples of how to use the function and print the results to the console.

More detailed explanation, especially of the regular expression used in the validate\_indian\_mobile function. Let's break down the pattern ^[6-9]\d{9}$:

Regular expressions are powerful tools for pattern matching in strings. Each character and symbol in the pattern has a specific meaning:

* **^**: This is an **anchor** that matches the **beginning of the string**. It asserts that the pattern must start at the very first character of the input string. Without ^, the pattern could potentially match a substring within a longer string.
* **[6-9]**: This is a **character class**. It matches **any single character** that is within the specified range. In this case, it matches any single digit from 6 to 9. This is how we enforce the requirement that the mobile number must start with one of these digits.
* **\d**: This is a **special sequence** that matches **any digit character** (equivalent to [0-9]).
* **{9}**: This is a **quantifier**. It specifies that the preceding element (\d in this case) must occur **exactly 9 times**. So, \d{9} matches exactly nine digits.
* **$**: This is another **anchor** that matches the **end of the string**. It asserts that the pattern must end at the very last character of the input string. Without $, the pattern could match the beginning of a string even if there were extra characters after the 10 digits.

Putting it all together, the regular expression ^[6-9]\d{9}$ means:

"Match a string that **starts** (^) with a single digit between 6 and 9 ([6-9]), is immediately followed by exactly nine digits (\d{9}), and then **ends** ($)."

This precisely enforces your requirements: the number must start with 6, 7, 8, or 9, and it must contain exactly 10 digits in total.

The re.match() function specifically tries to match the pattern at the *beginning* of the string. Since our pattern includes both ^ and $, re.match() will only return a match if the *entire* string conforms to the pattern from start to finish.

**Prompt-2:** 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:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Output:**

**A white background with black text

AI-generated content may be incorrect.**

**Explanation:**

This function is designed to compute the factorial of a non-negative integer. It also includes checks to handle invalid inputs.

Here's a breakdown of the code:

1. **def calculate\_factorial(n):**: This line defines the function named calculate\_factorial that takes one argument, n.
2. **""" ... """**: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns). It's good practice to include docstrings for documentation.
3. **if not isinstance(n, int):**: This is the first check. isinstance(n, int) checks if the input n is an integer. If it's *not* an integer (e.g., a float, string, etc.), the condition not isinstance(n, int) is true, and the function proceeds to the next line.
4. **return "Input must be an integer."**: If the input n is not an integer, this line returns the specified message and the function stops executing.
5. **if n < 0:**: This is the second check, executed only if the input is an integer. It checks if the integer n is less than 0 (i.e., a negative number).
6. **return "Factorial is not defined for negative numbers."**: If n is a negative integer, this line returns the message indicating that the factorial is not defined for negative numbers, and the function stops.
7. **elif n == 0:**: This is the third check, executed only if the input is a non-negative integer. elif stands for "else if". It checks if n is equal to 0.
8. **return 1**: If n is 0, this line returns 1, which is the correct factorial of 0.
9. **else:**: This block is executed if none of the previous conditions were met, meaning n is a positive integer.
10. **factorial = 1**: Inside the else block, a variable factorial is initialized to 1. This will store the calculated factorial.
11. **for i in range(1, n + 1):**: This is a for loop that iterates from 1 up to and including n. The range(1, n + 1) function generates a sequence of numbers starting from 1 and ending at n.
12. **factorial \*= i**: In each iteration of the loop, the current value of factorial is multiplied by the loop variable i. This is how the factorial is calculated (e.g., for n=5, it calculates 1\*1, then 1\*2, then 2\*3, then 6\*4, then 24\*5).
13. **return factorial**: After the loop finishes, the final calculated value of factorial is returned.

The lines after the function definition are examples of how to call the function with different inputs (a positive integer, zero, a negative integer, and a non-integer) and print the results to the console, demonstrating how the function handles each case.

**Prompt-3:** 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.

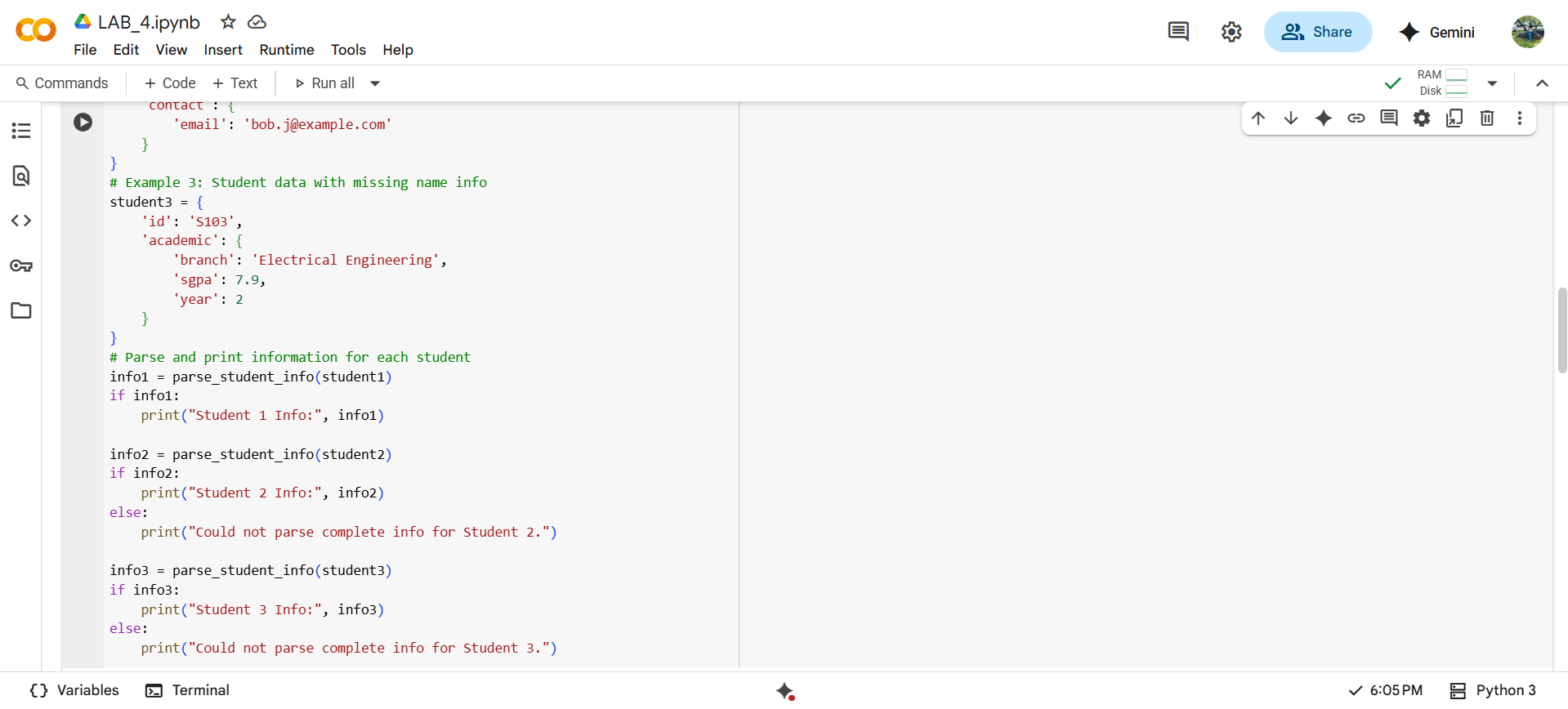
**Code:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

****

**Output:**

**A black text on a white background

AI-generated content may be incorrect.**

**Explanation:**

This function is designed to extract specific pieces of information (Full Name, Branch, and SGPA) from a nested dictionary that represents a student's data. It also includes logic to handle cases where some of the expected information might be missing.

Here's a breakdown of the code:

1. **def parse\_student\_info(student\_data):**: This line defines the function named parse\_student\_info that takes one argument, student\_data, which is expected to be a dictionary containing the student's information.
2. **""" ... """**: This is a docstring, explaining the purpose of the function, its arguments, and what it returns.
3. **extracted\_info = {}**: An empty dictionary called extracted\_info is initialized. This dictionary will store the extracted Full Name, Branch, and SGPA.
4. **name\_info = student\_data.get('name', {})**: This line attempts to get the value associated with the key 'name' from the student\_data dictionary.
   * .get('name', {}) is a safe way to access dictionary keys. If the key 'name' exists, it returns its value (which is expected to be another dictionary containing name details). If the key 'name' does *not* exist in student\_data, it returns the default value specified, which is an empty dictionary {}. This prevents a KeyError if the 'name' key is missing.
5. **extracted\_info['Full Name'] = f"{name\_info.get('first', '')} {name\_info.get('last', '')}".strip()**: This line constructs the 'Full Name'.
   * name\_info.get('first', '') attempts to get the value for the key 'first' from the name\_info dictionary. If 'first' is missing, it defaults to an empty string ''.
   * name\_info.get('last', '') does the same for the 'last' key.
   * f"{...} {...}" is an f-string, used to format the first and last names into a single string with a space in between.
   * .strip() removes any leading or trailing whitespace from the resulting string (useful if either first or last name was an empty string).
   * The resulting full name string is stored in the extracted\_info dictionary with the key 'Full Name'.
6. **academic\_info = student\_data.get('academic', {})**: Similar to the name extraction, this line safely gets the dictionary associated with the key 'academic' from student\_data, defaulting to an empty dictionary if the key is missing.
7. **extracted\_info['Branch'] = academic\_info.get('branch', 'N/A')**: This line attempts to get the value for the key 'branch' from the academic\_info dictionary. If 'branch' is missing, it defaults to the string 'N/A'. The value is stored in extracted\_info with the key 'Branch'.
8. **extracted\_info['SGPA'] = academic\_info.get('sgpa', 'N/A')**: This line attempts to get the value for the key 'sgpa' from the academic\_info dictionary. If 'sgpa' is missing, it defaults to the string 'N/A'. The value is stored in extracted\_info with the key 'SGPA'.
9. **if not extracted\_info['Full Name'] or extracted\_info['Branch'] == 'N/A' or extracted\_info['SGPA'] == 'N/A':**: This is a check to see if the essential information (Full Name, Branch, and SGPA) was successfully extracted.
   * not extracted\_info['Full Name'] is true if the 'Full Name' is an empty string (meaning both 'first' and 'last' names were missing).
   * extracted\_info['Branch'] == 'N/A' is true if the 'Branch' was not found and defaulted to 'N/A'.
   * extracted\_info['SGPA'] == 'N/A' is true if the 'SGPA' was not found and defaulted to 'N/A'.
   * The or operator means the condition is true if *any* of these essential pieces of information are missing.
10. **print("Warning: Could not extract all required information.")**: If the if condition in step 9 is true, this line prints a warning message.
11. **return None**: If essential information was missing, the function returns None. You could modify this to return the extracted\_info dictionary even if partial data was found, depending on your needs.
12. **return extracted\_info**: If all essential information was successfully extracted (the if condition in step 9 is false), the function returns the extracted\_info dictionary containing the parsed data.

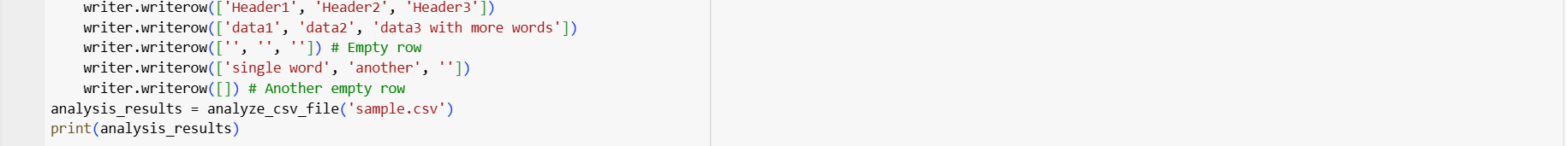
The code after the function definition provides three example student\_data dictionaries (one with complete data, one missing academic info, and one missing name info) and demonstrates how to call the parse\_student\_info function for each and print the results, showing how the function handles both complete and incomplete data.

**Prompt-4:** 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:**



**Explanation:**

This function is designed to read a CSV file and provide some basic statistics about its content: the total number of rows, the count of completely empty rows, and the total number of words across all cells.

Here's a breakdown of the code:

1. **import csv**: This line imports the csv module, which provides functionality for working with CSV files.
2. **def analyze\_csv\_file(file\_path):**: This defines the function named analyze\_csv\_file that takes one argument, file\_path, which is the path to the CSV file you want to analyze.
3. **""" ... """**: This is a docstring, explaining what the function does, its arguments, and what it returns.
4. **total\_rows = 0, empty\_rows = 0, total\_words = 0**: These lines initialize three variables to zero. These variables will store the counts as the function reads through the file.
5. **try:**: This starts a try block, which is used for error handling. Code within the try block is executed, and if an error occurs, the code within the corresponding except block is executed.
6. **with open(file\_path, 'r', newline='', encoding='utf-8') as csvfile:**: This line opens the CSV file for reading.
   * open(file\_path, 'r', ...) opens the file at the given file\_path in read mode ('r').
   * newline='' is important for working with CSV files to prevent extra blank rows from being inserted when reading on some operating systems.
   * encoding='utf-8' specifies the character encoding to use when reading the file. UTF-8 is a common and widely compatible encoding.
   * as csvfile: assigns the opened file object to the variable csvfile. The with statement ensures that the file is automatically closed even if errors occur.
7. **reader = csv.reader(csvfile)**: This line creates a csv.reader object. This object iterates over lines in the csvfile, parsing each line into a list of strings (the columns in that row).
8. **for row in reader:**: This loop iterates through each row (which is a list of strings) read from the CSV file by the csv.reader.
9. **total\_rows += 1**: Inside the loop, for each row read, the total\_rows counter is incremented by 1.
10. **if not any(cell.strip() for cell in row):**: This line checks if the current row is empty.
    * for cell in row: This part iterates through each cell (each string element) in the current row list.
    * cell.strip(): This removes any leading or trailing whitespace from the content of a cell.
    * any(...): This built-in Python function returns True if *any* item in the iterable (in this case, the result of cell.strip() for cell in row) is true. A non-empty string is considered true in a boolean context.
    * not any(...): So, not any(...) will be True only if *all* cells in the row are empty strings after stripping whitespace. This effectively identifies completely empty rows.
11. **empty\_rows += 1**: If the if condition in step 10 is true (the row is empty), the empty\_rows counter is incremented.
12. **for cell in row:**: This starts a nested loop that iterates through each cell (string) within the current row.
13. **total\_words += len(cell.split())**: This line counts the words in each cell and adds them to total\_words.
    * cell.split(): This splits the string in the cell into a list of words, using whitespace as the default delimiter.
    * len(...): This gets the number of elements in the list of words.
    * total\_words += ...: This adds the word count for the current cell to the running total.
14. **return { ... }**: After the loop finishes processing all rows, this line returns a dictionary containing the final counts for total\_rows, empty\_rows, and total\_words.
15. **except FileNotFoundError:**: This block is executed if a FileNotFoundError occurs during the execution of the try block (specifically, when trying to open the file). It returns a dictionary with an error message.
16. **except Exception as e:**: This is a general exception handler that catches any other type of error (Exception) that might occur in the try block. It returns a dictionary with a general error message including the specific exception e.

The code after the function definition first creates a simple sample.csv file with some data and empty rows for demonstration purposes, and then calls the analyze\_csv\_file function with the path to this sample file, printing the returned analysis results.

**Prompt-5:** 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:**

A screenshot of a computer

AI-generated content may be incorrect.

**Output:**

**A close-up of a text

AI-generated content may be incorrect.**

**Explanation:**

This function is designed to take a paragraph of text, process it, and find the word that appears most frequently.

Here's a breakdown of the code:

1. **import string**: This line imports the string module, which contains a collection of useful string constants, including string.punctuation.
2. **def analyze\_word\_frequency(paragraph):**: This defines the function named analyze\_word\_frequency that takes one argument, paragraph, which is the input text.
3. **""" ... """**: This is a docstring, explaining what the function does, its arguments, and what it returns.
4. **lowercase\_paragraph = paragraph.lower()**: This line converts the entire input paragraph to lowercase using the .lower() string method. This ensures that words like "The" and "the" are treated as the same word for frequency counting.
5. **translator = str.maketrans('', '', string.punctuation)**: This line creates a translation table that will be used to remove punctuation.
   * str.maketrans(x, y, z) is a class method that creates a translation table.
   * The first two arguments (x and y) are used for character mapping (replacing characters), but here they are empty strings, meaning no characters will be replaced.
   * The third argument (z) is a string of characters to be deleted from the original string. string.punctuation provides a string containing all common punctuation characters (like !, ", #, $, %, &, ', (, ), \*, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \, ], ^, \_, `, {, |, }, ~).
6. **cleaned\_text = lowercase\_paragraph.translate(translator)**: This line uses the translate() string method to apply the translator table created in the previous step to the lowercase\_paragraph. This effectively removes all punctuation characters from the text.
7. **words = cleaned\_text.split()**: This line splits the cleaned\_text into a list of individual words. The .split() method, when called without arguments, splits the string by whitespace (spaces, tabs, newlines), returning a list of the resulting substrings (words).
8. **word\_counts = {}**: An empty dictionary called word\_counts is initialized. This dictionary will store each unique word as a key and its frequency (how many times it appears) as the corresponding value.
9. **for word in words:**: This loop iterates through each word in the words list created in step 7.
10. **if word in word\_counts:**: Inside the loop, this checks if the current word is already a key in the word\_counts dictionary.
11. **word\_counts[word] += 1**: If the word is already in the dictionary, its current count is incremented by 1.
12. **else: word\_counts[word] = 1**: If the word is not yet in the dictionary, it means this is the first time we've encountered this word. So, it's added to the word\_counts dictionary as a new key with an initial value of 1.
13. **if word\_counts:**: After the loop finishes counting all words, this checks if the word\_counts dictionary is not empty. An empty dictionary evaluates to False in a boolean context.
14. **most\_frequent\_word = max(word\_counts, key=word\_counts.get)**: If the word\_counts dictionary is not empty, this line finds the word with the highest frequency.
    * max(word\_counts, ...) finds the maximum element in the word\_counts dictionary (which are the keys, i.e., the words).
    * key=word\_counts.get specifies that the comparison for finding the maximum should be based on the *values* of the dictionary (the word counts), not the keys themselves. word\_counts.get is a method that returns the value for a given key.
15. **return most\_frequent\_word**: The word with the highest frequency is returned.
16. **else: return None**: If the word\_counts dictionary is empty (meaning the input paragraph had no words after cleaning, perhaps it was just punctuation or empty), the function returns None.

The code in cell provides examples of how to use this function with different paragraphs and prints the results.