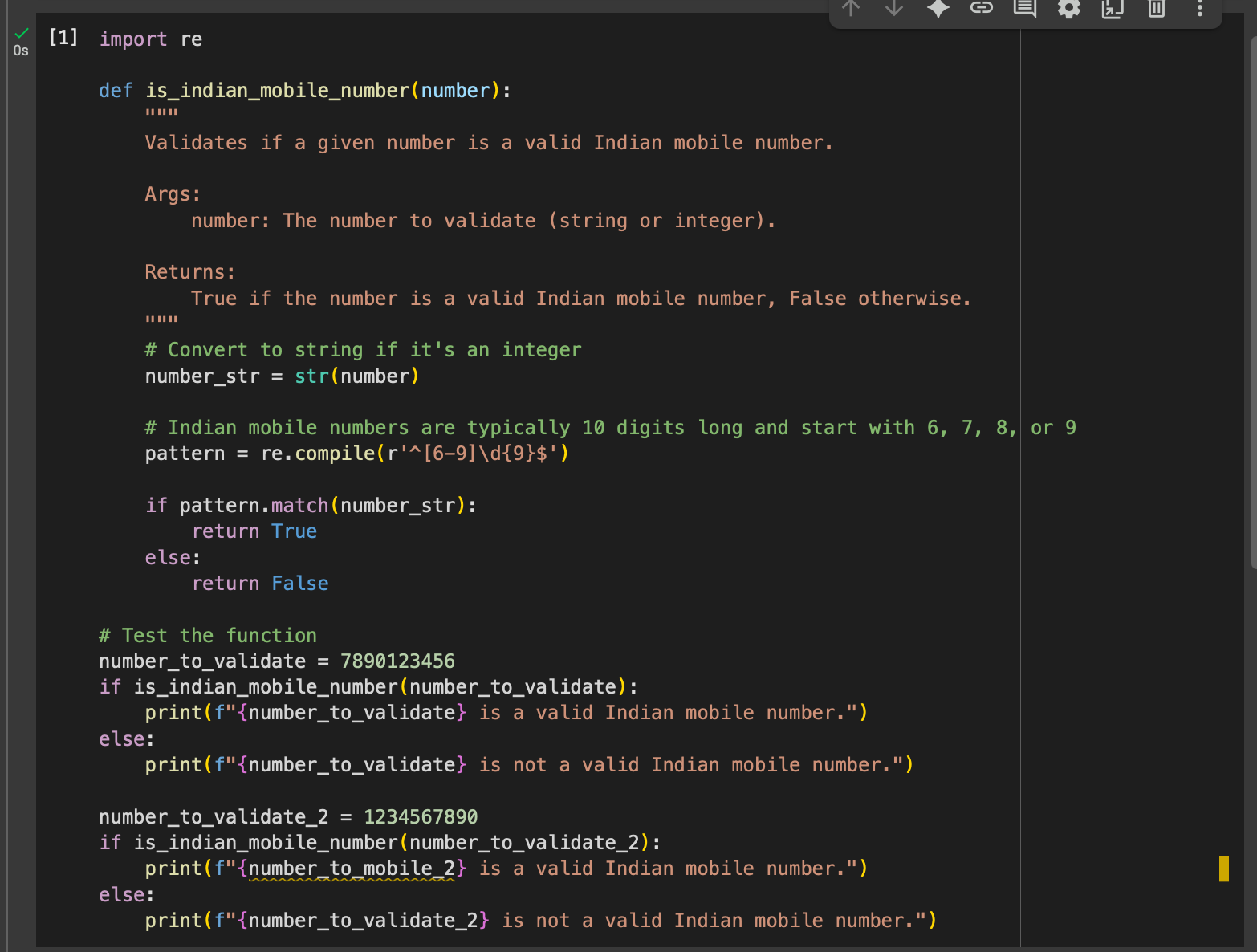
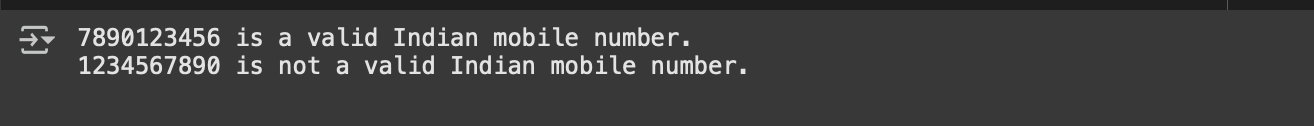
**ASSIGNMENT 4.1  
( 2403A52398 )**

**TASK#1**Promt :

generate a python function that validates is 7890123456 is an Indian mobile number or not

Code :



Output:  
****

Explanation:  
This code defines a Python function called validate\_indian\_mobile that checks if a given string is a valid Indian mobile number.

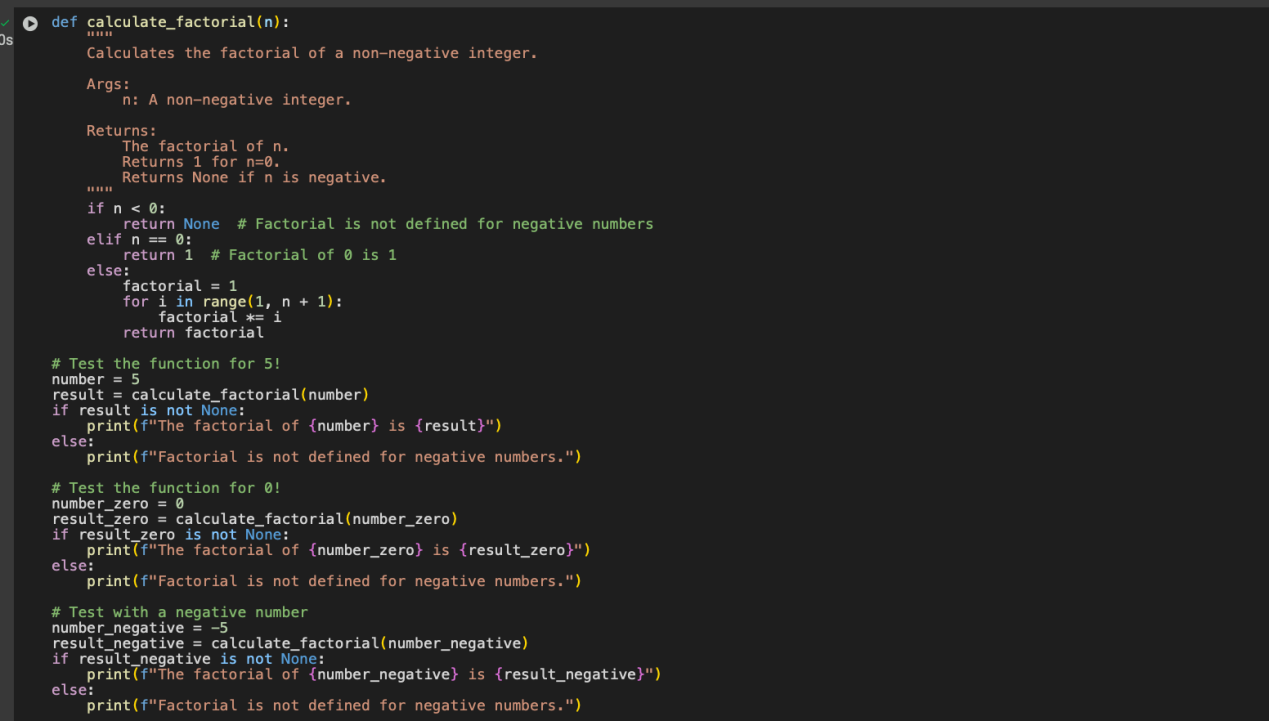
Here's a breakdown:

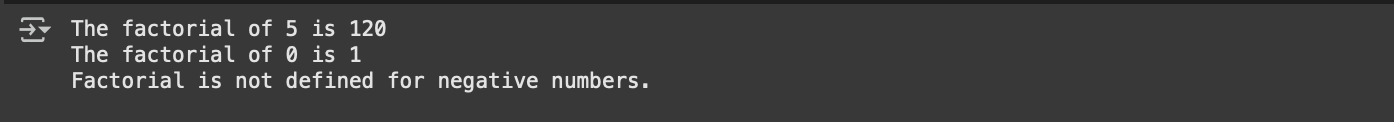
* import re: This line imports the regular expression module in Python, which is used for pattern matching in strings.
* def validate\_indian\_mobile(number):: This defines the function validate\_indian\_mobile that takes one argument, number.
* pattern = re.compile(r'^(?:(?:\+|0{0,2})91(\s\*[\-]\s\*)?|[0]?)?[6789]\d{9}$'): This is the core of the validation. It creates a regular expression pattern:
  + ^: Matches the beginning of the string.
  + (?:(?:\+|0{0,2})91(\s\*[\-]\s\*)?|[0]?)?: This part handles optional prefixes like +91, 091, 0091, or 0.
    - (?:\+|0{0,2})91: Matches +91 or 0, 00followed by 91.
    - (\s\*[\-]\s\*)?: Optionally matches any whitespace and a hyphen after the country code.
    - [0]?: Optionally matches a leading 0.
  + [6789]: Matches a digit that is 6, 7, 8, or 9 (Indian mobile numbers typically start with these digits).
  + \d{9}: Matches exactly 9 more digits (for a total of 10 digits after the initial digit).
  + $: Matches the end of the string.
* return bool(pattern.match(number)): This line attempts to match the pattern against the input number. pattern.match() returns a match object if the pattern is found at the beginning of the string, otherwise it returns None. bool() converts the result to a boolean: True if there's a match, False otherwise.

**TASK#2**

Promt :

generate a python function that generates factorial of 5 ,handles 0!

Code:  


Output :  


Explanation :  
This Python function calculates the factorial of a non-negative integer n.

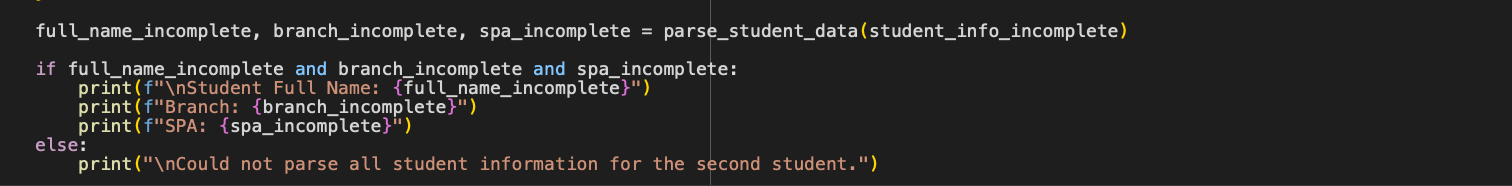
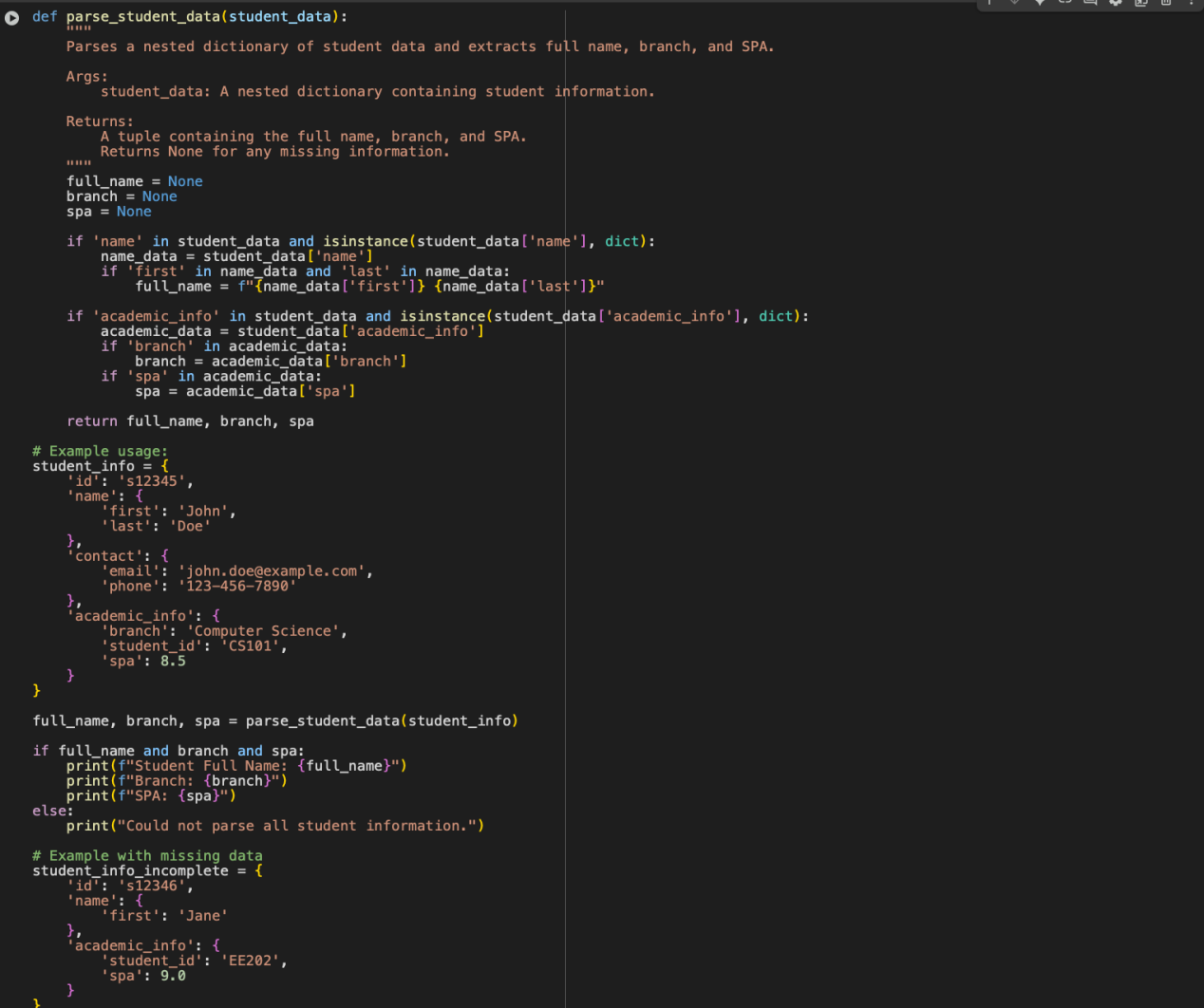
Here's a breakdown:

* def calculate\_factorial(n):: This line defines the function named calculate\_factorial which takes one argument, n.
* if n < 0:: This checks if the input number n is negative. The factorial is not defined for negative numbers.
  + return None: If n is negative, the function returns None to indicate that the factorial cannot be calculated.
* elif n == 0:: This checks if the input number n is zero. The factorial of 0 is defined as 1.
  + return 1: If n is 0, the function returns 1.
* else:: This block is executed if n is a positive integer.
  + factorial = 1: A variable factorial is initialized to 1. This will store the calculated factorial value.
  + for i in range(1, n + 1):: This loop iterates from 1 up to and including n.
    - factorial \*= i: In each iteration, the current value of factorial is multiplied by the loop variable i. This performs the calculation of n! (n \* (n-1) \* ... \* 2 \* 1).
  + return factorial: After the loop finishes, the function returns the calculated factorial.

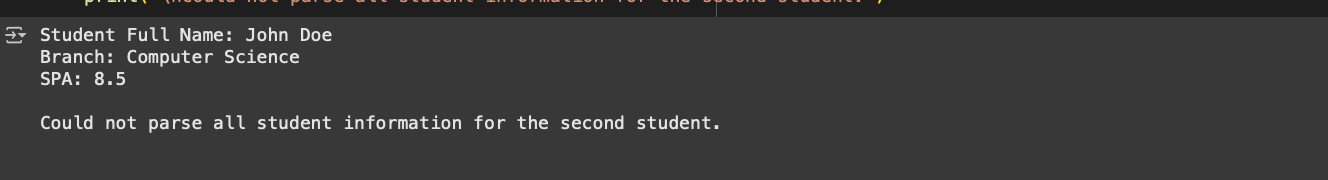
**TASK#3**Promt :

create a python function that parses a nested dictionary of a student and give student full name, branch , spa in output

Code:



Output :



Explanation:  
This Python function is designed to extract specific pieces of information (full name, branch, and SPA) from a nested dictionary representing a student's data.

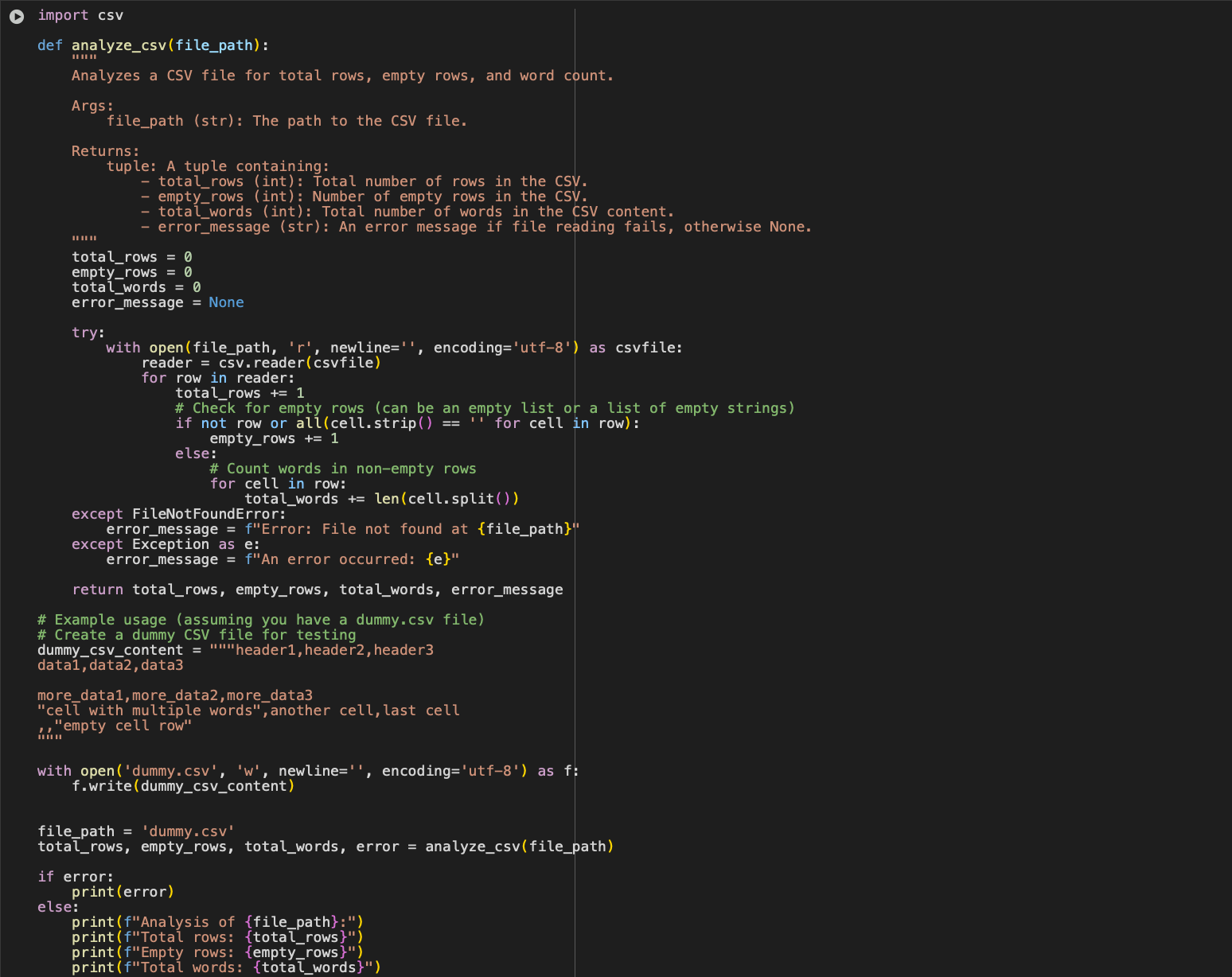
Here's a step-by-step explanation:

* def parse\_student\_data(student\_data):: This line defines the function parse\_student\_data that takes one argument, student\_data, which is expected to be the nested dictionary containing the student's information.
* full\_name = None**,** branch = None**,** spa = None: These lines initialize three variables to None. These variables will store the extracted full name, branch, and SPA. They are initialized to None so that if any of the information is not found in the dictionary, the function will return None for that specific piece of data.
* if 'name' in student\_data and isinstance(student\_data['name'], dict):: This checks if there is a key named 'name' in the student\_data dictionary and if the value associated with 'name' is itself a dictionary. This is important because the name is expected to be in a nested dictionary.
  + name\_data = student\_data['name']: If the conditions in the ifstatement are true, this line assigns the nested 'name' dictionary to the name\_data variable.
  + if 'first' in name\_data and 'last' in name\_data:: This checks if the name\_data dictionary contains both the keys 'first'and 'last'.
  + full\_name = f"{name\_data['first']} {name\_data['last']}": If both 'first' and 'last' names are present, this line constructs the full name by combining the values associated with these keys, separated by a space, and assigns it to the full\_name variable.
* if 'academic\_info' in student\_data and isinstance(student\_data['academic\_info'], dict):: This checks if there is a key named 'academic\_info' in the student\_data dictionary and if the value associated with 'academic\_info' is also a dictionary. This is where the branch and SPA are expected to be found.
  + academic\_data = student\_data['academic\_info']: If the conditions in the if statement are true, this line assigns the nested 'academic\_info' dictionary to the academic\_data variable.
  + if 'branch' in academic\_data:: This checks if the academic\_data dictionary contains the key 'branch'.
  + branch = academic\_data['branch']: If the 'branch' key is present, its value is assigned to the branch variable.
  + if 'spa' in academic\_data:: This checks if the academic\_data dictionary contains the key 'spa'.
  + spa = academic\_data['spa']: If the 'spa' key is present, its value is assigned to the spa variable.
* return full\_name, branch, spa: Finally, the function returns a tuple containing the values of full\_name, branch, and spa. If any of these pieces of information were not found in the input dictionary, the corresponding value in the returned tuple will be None.

**TASK#4**

Promt :generate a python function for css analysis , which reads a .csv file returns total no.of rows . counts no.of empty rows , counts no.of words

**Code:**

****

**Output :**

**  
  
Explanation:  
This Python function reads a CSV (Comma Separated Values) file and performs some basic analysis: counting the total number of rows, identifying and counting empty rows, and counting the total number of words across all cells in the file.**

**Here's a breakdown of the code:**

**import csv: This line imports the csv module, which is part of Python's standard library and provides functionality for working with CSV files.**

**def analyze\_csv(file\_path):: This defines the function analyze\_csv that takes one argument, file\_path, which is the path to the CSV file you want to analyze.**

**total\_rows = 0, empty\_rows = 0, total\_words = 0, error\_message = None: These lines initialize variables to store the results of the analysis.**

**total\_rows will store the count of all rows.**

**empty\_rows will store the count of rows that are empty or contain only empty strings.**

**total\_words will store the total count of words in the file.**

**error\_message will be used to store any error message if something goes wrong during file processing. It's initialized to None.**

**try...except block: This block is used for error handling. It attempts to open and read the file, and if an error occurs (like the file not being found), it catches the exception and sets the error\_message.**

**with open(file\_path, 'r', newline='', encoding='utf-8') as csvfile:: This opens the file specified by file\_path for reading ('r').**

**newline='' is important when working with CSV files to prevent extra blank rows.**

**encoding='utf-8' specifies the character encoding for reading the file.**

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

**reader = csv.reader(csvfile): This creates a csv.readerobject, which is an iterator that reads rows from the CSV file.**

**for row in reader:: This loop iterates through each row in the CSV file. Each row is a list of strings, where each string represents a cell in that row.**

**total\_rows += 1: For every row read, the total\_rowscounter is incremented.**

**if not row or all(cell.strip() == '' for cell in row):: This checks if the current row is considered empty.**

**not row: Checks if the row is an empty list [].**

**all(cell.strip() == '' for cell in row): This checks if all the cells in the row, after removing leading/trailing whitespace (cell.strip()), are empty strings. This handles cases where a row might contain commas but no actual data.**

**empty\_rows += 1: If the row is determined to be empty, the empty\_rows counter is incremented.**

**else:: This block is executed for rows that are not empty.**

**for cell in row:: This inner loop iterates through each cell (string) in the current row.**

**total\_words += len(cell.split()): For each cell, it splits the string by whitespace (cell.split()) to get a list of words and then adds the number of words (len(...)) to the total\_words counter.**

**except FileNotFoundError:: This catches the specific error that occurs if the file specified by file\_path does not exist.**

**error\_message = f"Error: File not found at {file\_path}": If a FileNotFoundError occurs, a descriptive error message is stored in error\_message.**

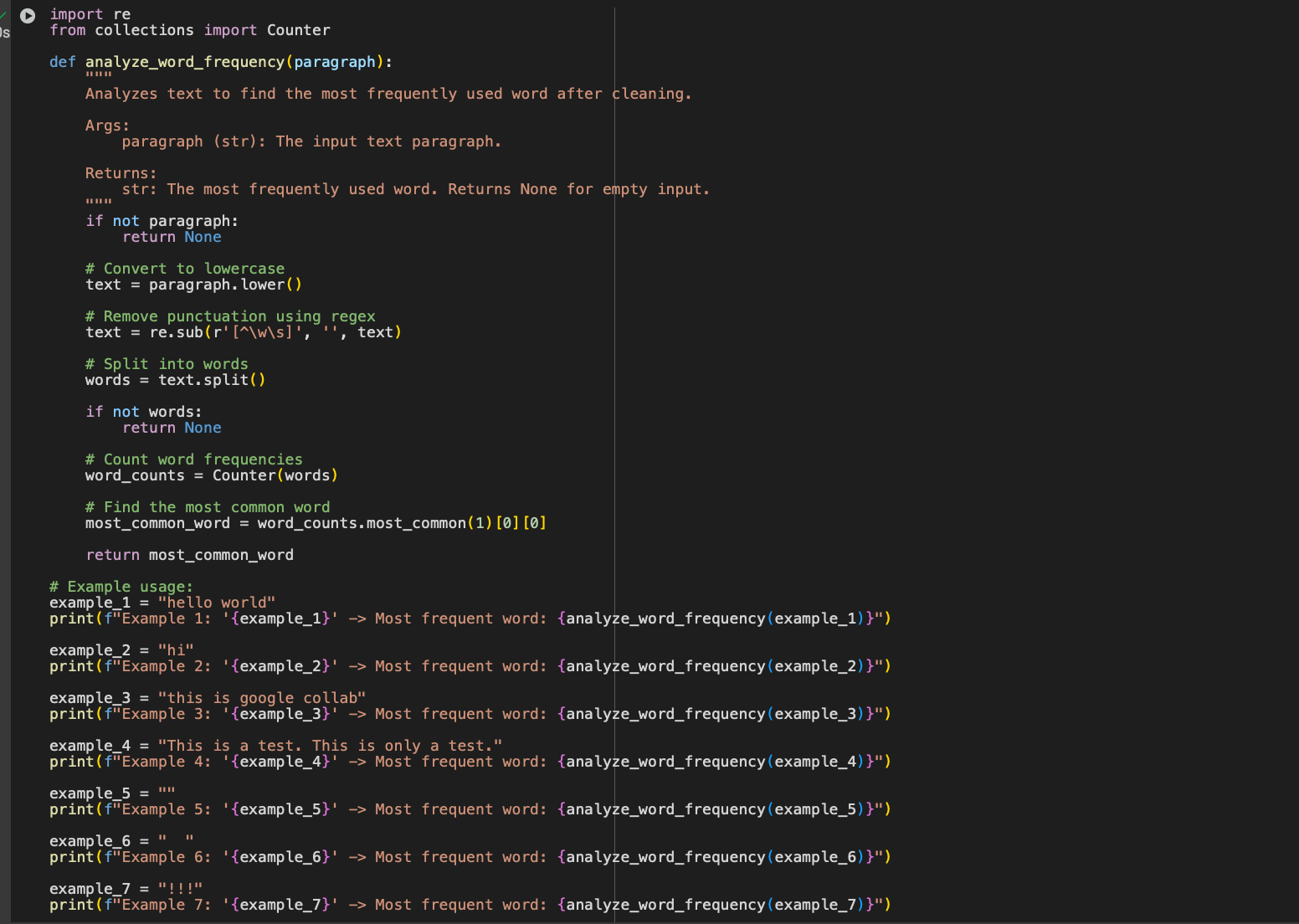
**except Exception as e:: This catches any other potential errors that might occur during file processing.**

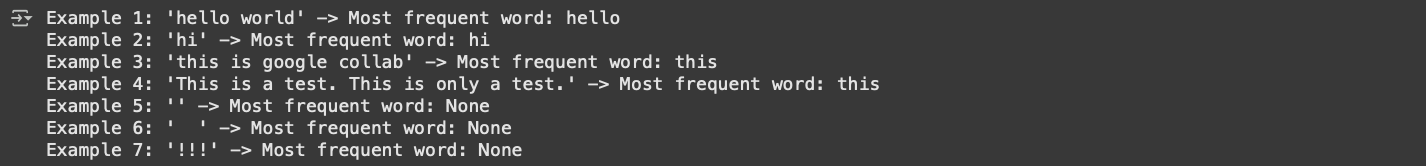
**error\_message = f"An error occurred: {e}": A generic error message including the exception details (e) is stored in error\_message.**

**return total\_rows, empty\_rows, total\_words, error\_message: The function returns a tuple containing the calculated values and any error message.**

**TASK#5**Promt:  
enerate a Python function that processes text and analyzes word frequency. The function must Accept a paragraph as input, Convert all text to lowercase,Remove punctuation,Return the most frequently used word example-1 : hello world example-2 : hi example-3 : this is google collab

Code:

****

**Output:  
**

**Explanation:**This Python function is designed to find the most frequently occurring word in a given block of text (a paragraph), after cleaning the text by converting it to lowercase and removing punctuation.

Here's a step-by-step explanation:

* import re: This line imports the re module, which is Python's regular expression library. It's used here for removing punctuation.
* from collections import Counter: This line imports the Counter class from the collections module. Counter is a specialized dictionary subclass for counting hashable objects. It's very useful for counting the occurrences of words.
* def analyze\_word\_frequency(paragraph):: This defines the function analyze\_word\_frequency that takes one argument, paragraph, which is the input text string.
* if not paragraph:: This checks if the input paragraph is empty (an empty string).
  + return None: If the paragraph is empty, there are no words to analyze, so the function returns None.
* text = paragraph.lower(): This line converts the entire input paragraph to lowercase. This is done to ensure that words like "The" and "the" are treated as the same word for frequency counting.
* text = re.sub(r'[^\w\s]', '', text): This line uses a regular expression to remove punctuation from the text.
  + re.sub() is used for substitution.
  + r'[^\w\s]': This is the regular expression pattern.
    - r'': Denotes a raw string, which is often used for regular expressions to avoid issues with backslashes.
    - []: Defines a character set.
    - ^: Inside a character set, ^ means "not".
    - \w: Matches any word character (alphanumeric characters plus underscore: [a-zA-Z0-9\_]).
    - \s: Matches any whitespace character (spaces, tabs, newlines, etc.).
    - So, [^\w\s] matches any character that is NOT a word character and NOT a whitespace character (i.e., it matches punctuation and other symbols).
  + '': This is the replacement string. It's an empty string, meaning the matched punctuation characters are replaced with nothing (effectively removed).
  + text: This is the string on which the substitution is performed.
  + The result of the substitution is assigned back to the text variable.
* words = text.split(): This line splits the cleaned text string into a list of individual words using whitespace as the delimiter.
* if not words:: This checks if the resulting words list is empty after splitting (which could happen if the input was only punctuation or whitespace).
  + return None: If there are no words, the function returns None.
* word\_counts = Counter(words): This is where the Counter class comes in handy. It takes the words list and automatically creates a dictionary-like object where keys are the words and values are their frequencies (counts).
* most\_common\_word = word\_counts.most\_common(1)[0][0]: This line finds the most frequently used word.
  + word\_counts.most\_common(1): This method of the Counter object returns a list of the N most common elements and their counts. We pass 1 to get only the single most common word. The result is a list of tuples, like [('the', 5)].
  + [0]: We access the first element of the list (which is the tuple for the most common word). So, we get ('the', 5).
  + [0]: We access the first element of the tuple (which is the word itself). So, we get 'the'.
  + This word is then assigned to the most\_common\_word variable.
* return most\_common\_word: The function returns the most\_common\_word.