LAB ASSIGNMENT – 4.1

NAME	**:**	M.Shiva
HALL.NO	**:**	2403A52377
BATCH.NO	**:**	AI 14
ASSIGNMENT	**:**	4.1

PROMPT 01 :

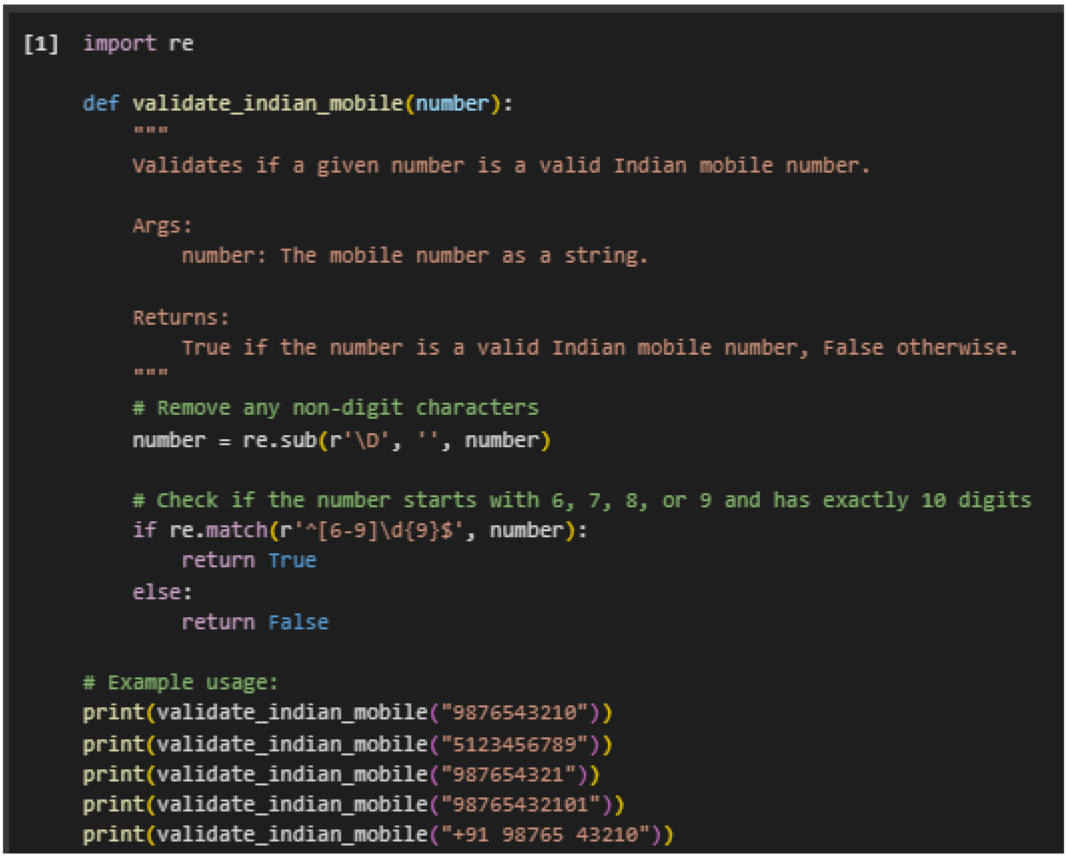
i need a python code function that validates an Indian mobile number.
Requirements

The function must ensure the mobile number:

Starts with 6, 7, 8, or 9

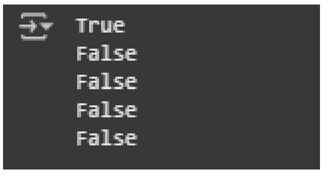
Contains exactly 10 digits

CODE :



```
[1]  import re

     def validate_indian_mobile(number):
         """
         Validates if a given number is a valid Indian mobile number.

         Args:
             number: The mobile number as a string.

         Returns:
             True if the number is a valid Indian mobile number, False otherwise.
         """
         # Remove any non-digit characters
         number = re.sub(r'\D', '', number)

         # Check if the number starts with 6, 7, 8, or 9 and has exactly 10 digits
         if re.match(r'^[6-9]\d{9}$', number):
             return True
         else:
             return False

     # Example usage:
     print(validate_indian_mobile("9876543210"))
     print(validate_indian_mobile("5123456789"))
     print(validate_indian_mobile("987654321"))
     print(validate_indian_mobile("98765432101"))
     print(validate_indian_mobile("+91 98765 43210"))
```

OUTPUT :



```
True
False
False
False
False
```

- ## 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.

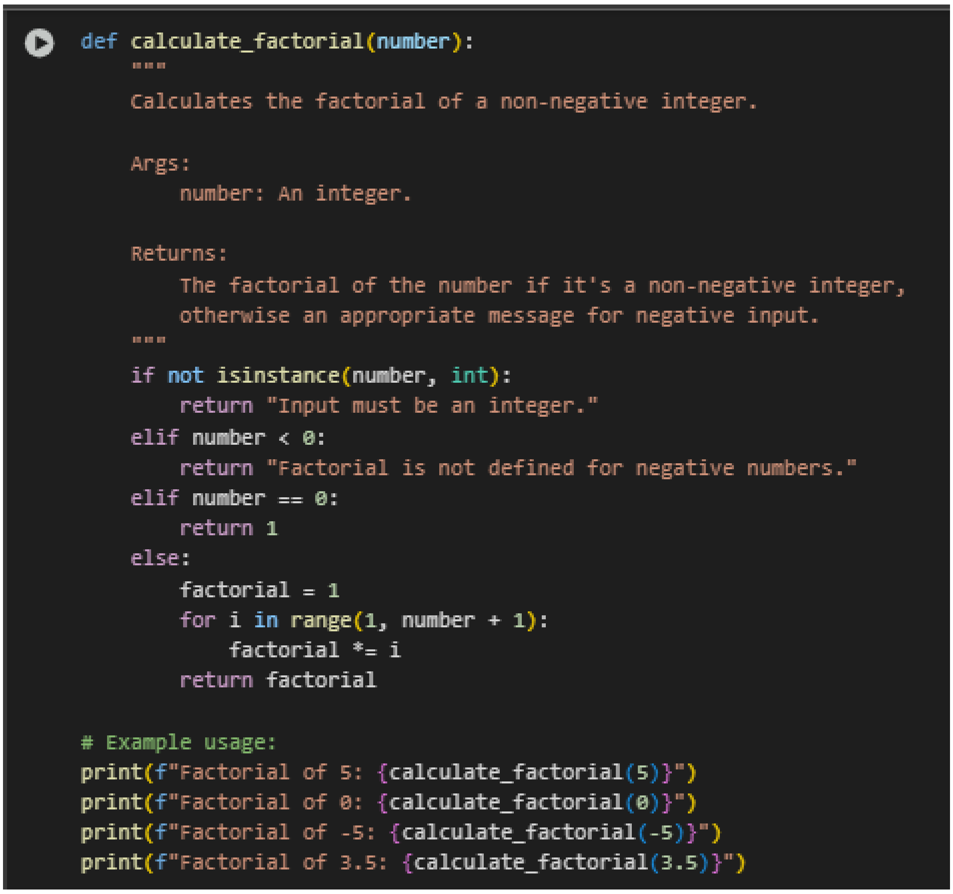
- **if re.match(r'^[6-9]\d{9}$', number):**: This is the core validation step.
    - re.match() attempts to match the regular expression pattern from the *beginning* of the cleaned number string.

  - ^[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 :

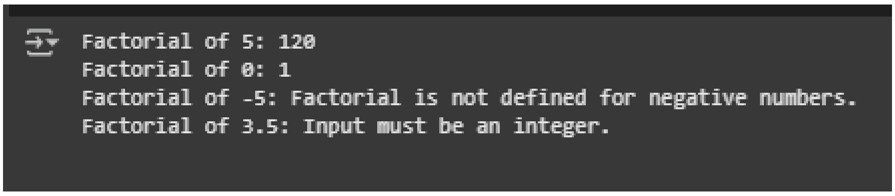
i need a python code function that calculates the factorial of a number. Requirements • Provide one sample input-output pair in the prompt to guide the AI. • The function should handle: o 0! correctly o Negative input by returning an appropriate message

CODE :



```python
def calculate_factorial(number):
    """
    Calculates the factorial of a non-negative integer.

    Args:
        number: An integer.

    Returns:
        The factorial of the number if it's a non-negative integer,
        otherwise an appropriate message for negative input.
    """
    if not isinstance(number, int):
        return "Input must be an integer."
    elif number < 0:
        return "Factorial is not defined for negative numbers."
    elif number == 0:
        return 1
    else:
        factorial = 1
        for i in range(1, number + 1):
            factorial *= i
        return factorial

# Example usage:
print(f"Factorial of 5: {calculate_factorial(5)}")
print(f"Factorial of 0: {calculate_factorial(0)}")
print(f"Factorial of -5: {calculate_factorial(-5)}")
print(f"Factorial of 3.5: {calculate_factorial(3.5)}")
```

OUTPUT :



```
Factorial of 5: 120
Factorial of 0: 1
Factorial of -5: Factorial is not defined for negative numbers.
Factorial of 3.5: Input must be an integer.
```

CODE EXPLANATION :

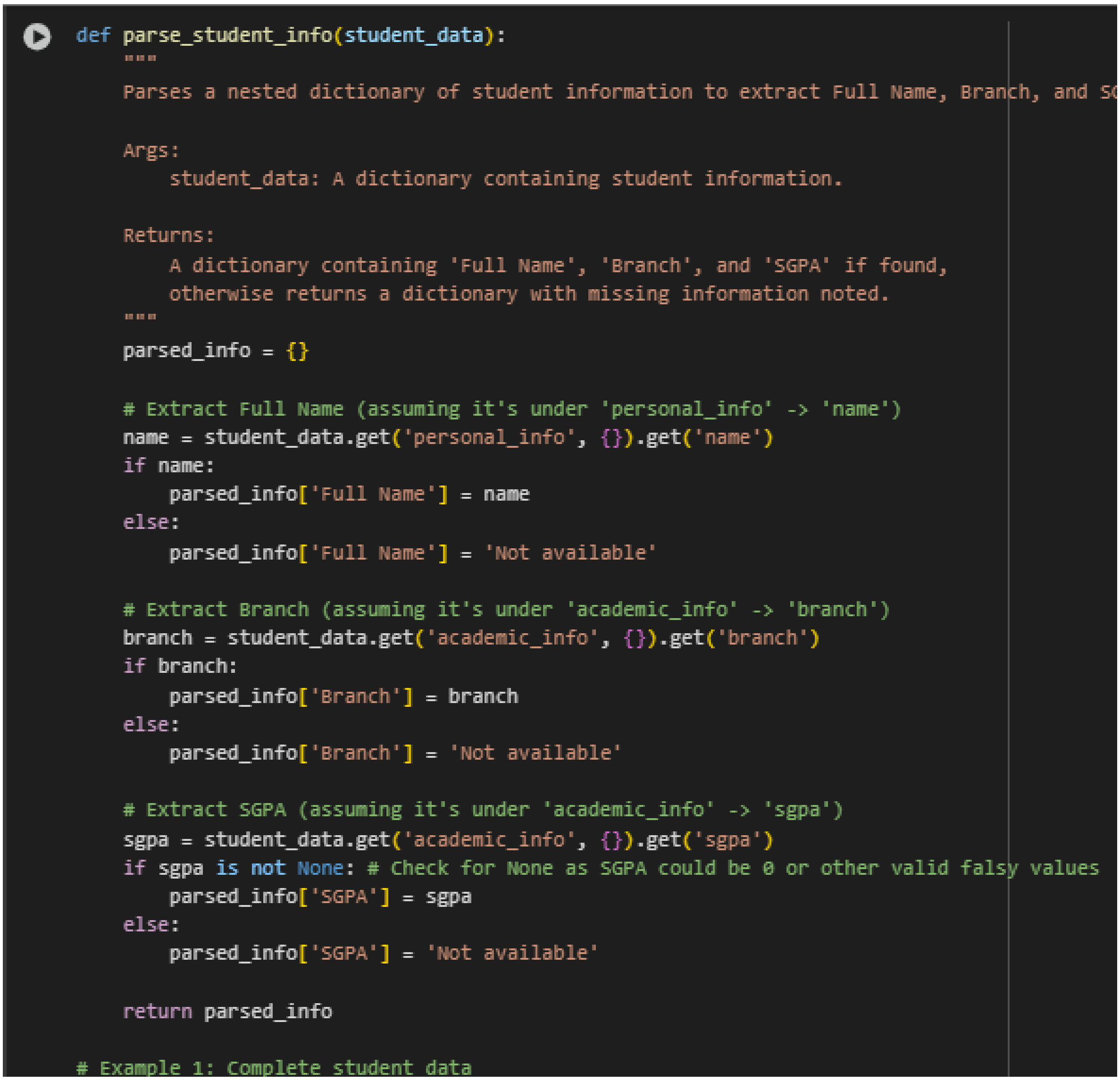
- def calculate_factorial(number):: This defines a function named calculate_factorial that takes one argument, number, which is the integer for which you want to calculate the factorial.
- if not isinstance(number, int):: This checks if the input number is not an integer. If it's not an integer, it returns 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.

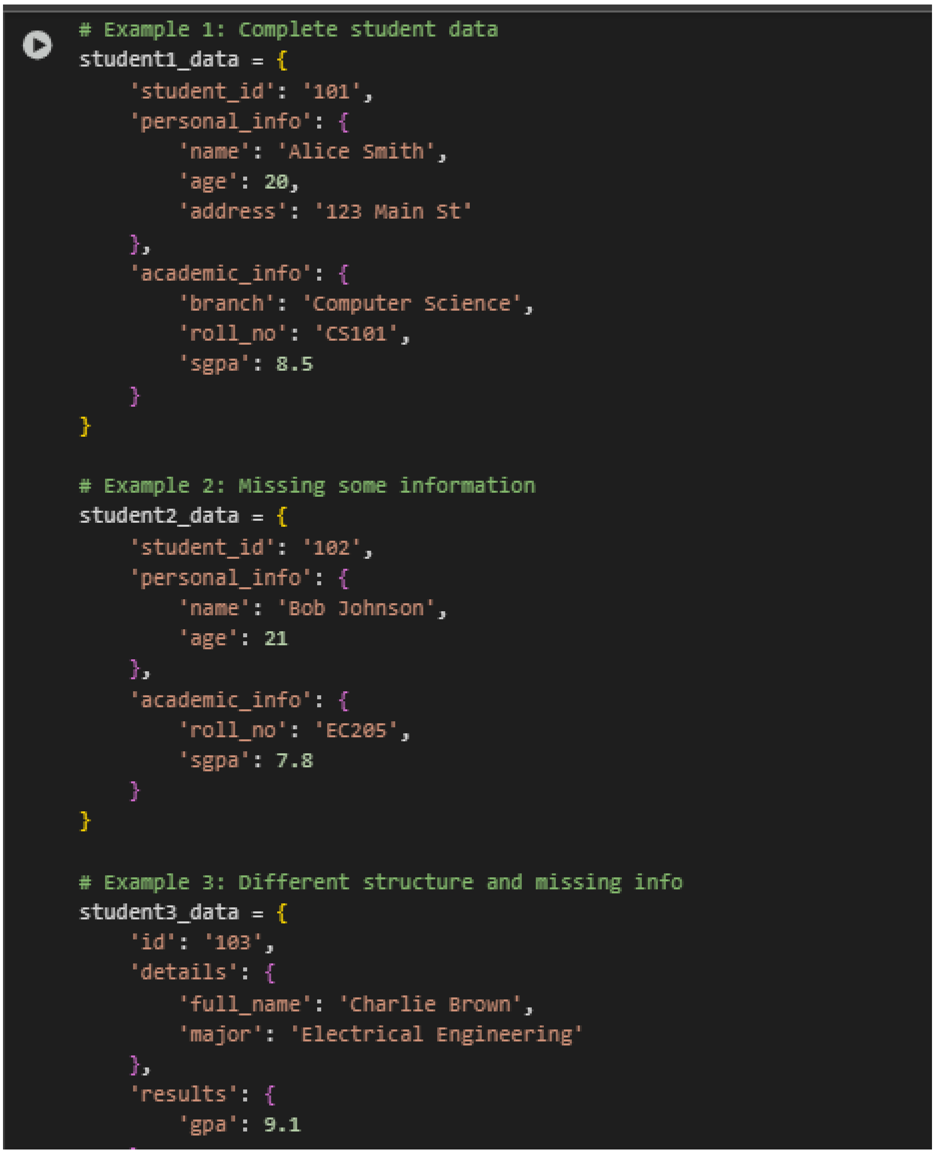
PROMPT 03 :

i need a python code with 2 to 3 examples function that parses a nested dictionary representing student information. Requirements • The function should extract and return: o Full Name o Branch o SGPA

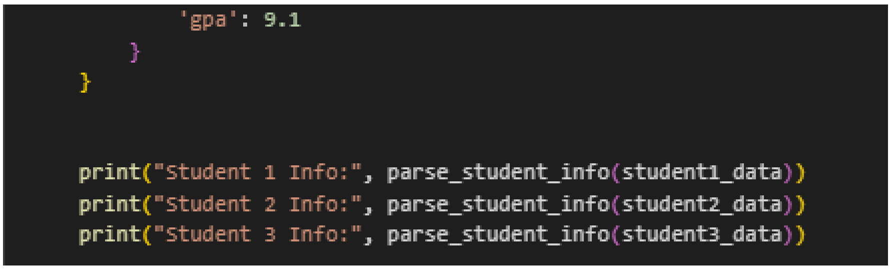
CODE :



```python
def parse_student_info(student_data):
    """
    Parses a nested dictionary of student information to extract Full Name, Branch, and S

    Args:
        student_data: A dictionary containing student information.

    Returns:
        A dictionary containing 'Full Name', 'Branch', and 'SGPA' if found,
        otherwise returns a dictionary with missing information noted.
    """
    parsed_info = {}

    # Extract Full Name (assuming it's under 'personal_info' -> 'name')
    name = student_data.get('personal_info', {}).get('name')
    if name:
        parsed_info['Full Name'] = name
    else:
        parsed_info['Full Name'] = 'Not available'

    # Extract Branch (assuming it's under 'academic_info' -> 'branch')
    branch = student_data.get('academic_info', {}).get('branch')
    if branch:
        parsed_info['Branch'] = branch
    else:
        parsed_info['Branch'] = 'Not available'

    # Extract SGPA (assuming it's under 'academic_info' -> 'sgpa')
    sgpa = student_data.get('academic_info', {}).get('sgpa')
    if sgpa is not None: # Check for None as SGPA could be 0 or other valid falsy values
        parsed_info['SGPA'] = sgpa
    else:
        parsed_info['SGPA'] = 'Not available'

    return parsed_info

# Example 1: Complete student data
```

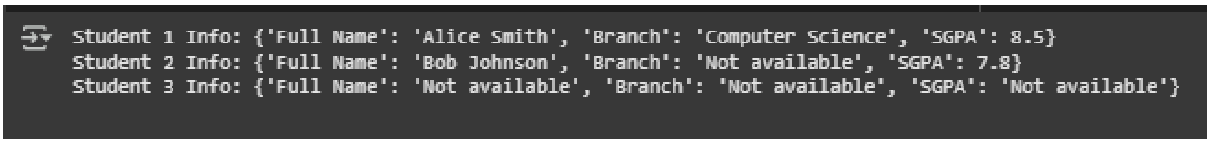


```
# Example 1: Complete student data
student1_data = {
    'student_id': '101',
    'personal_info': {
        'name': 'Alice Smith',
        'age': 20,
        'address': '123 Main St'
    },
    'academic_info': {
        'branch': 'Computer Science',
        'roll_no': 'CS101',
        'sgpa': 8.5
    }
}

# Example 2: Missing some information
student2_data = {
    'student_id': '102',
    'personal_info': {
        'name': 'Bob Johnson',
        'age': 21
    },
    'academic_info': {
        'roll_no': 'EC205',
        'sgpa': 7.8
    }
}

# Example 3: Different structure and missing info
student3_data = {
    'id': '103',
    'details': {
        'full_name': 'Charlie Brown',
        'major': 'Electrical Engineering'
    },
    'results': {
        'gpa': 9.1
```



```
        'gpa': 9.1
    }
}


print("Student 1 Info:", parse_student_info(student1_data))
print("Student 2 Info:", parse_student_info(student2_data))
print("Student 3 Info:", parse_student_info(student3_data))
```

OUT PUT :



```
Student 1 Info: {'Full Name': 'Alice Smith', 'Branch': 'Computer Science', 'SGPA': 8.5}
Student 2 Info: {'Full Name': 'Bob Johnson', 'Branch': 'Not available', 'SGPA': 7.8}
Student 3 Info: {'Full Name': 'Not available', 'Branch': 'Not available', 'SGPA': 'Not available'}
```

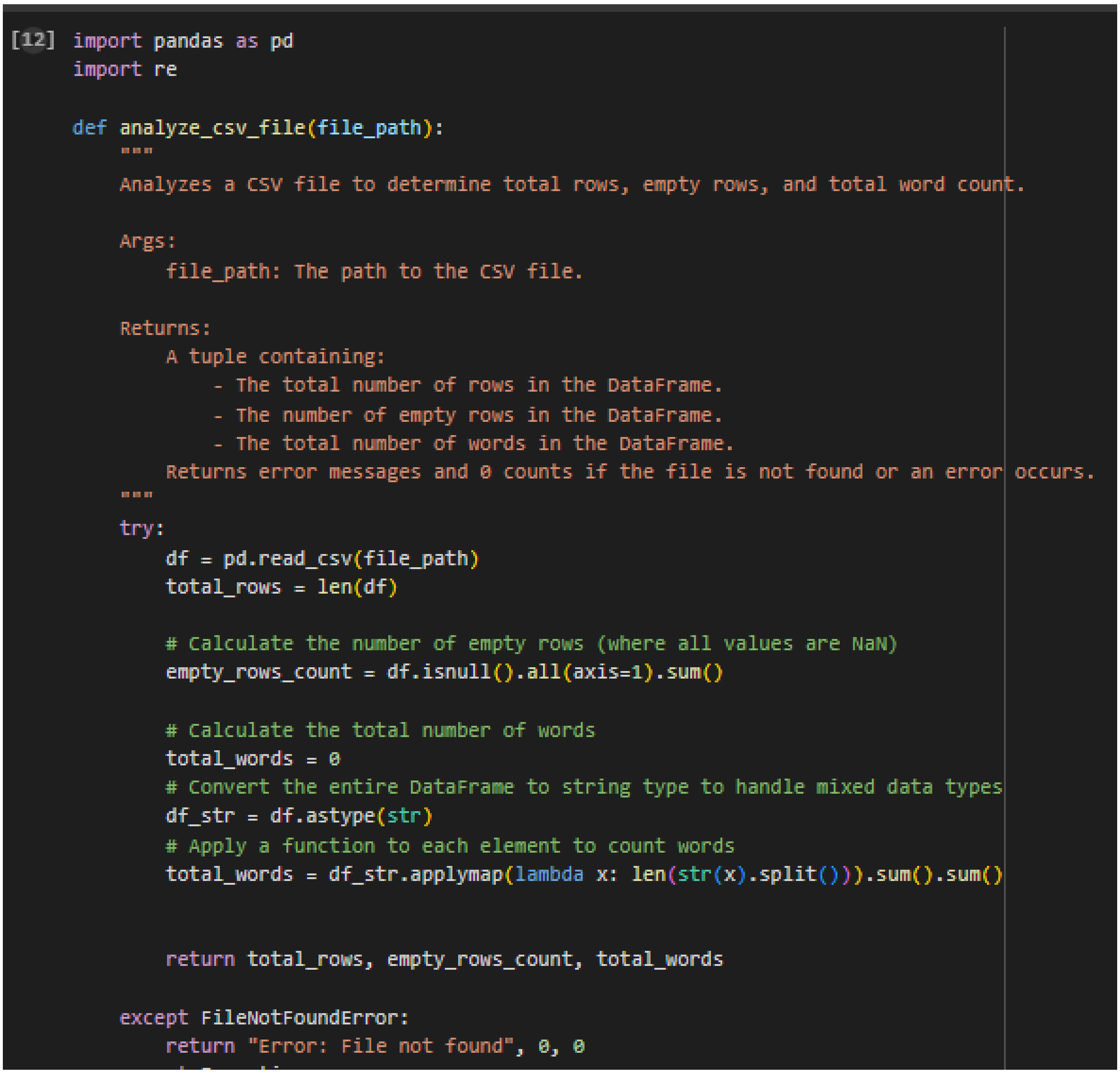
CODE EXPLANATION :

- 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.

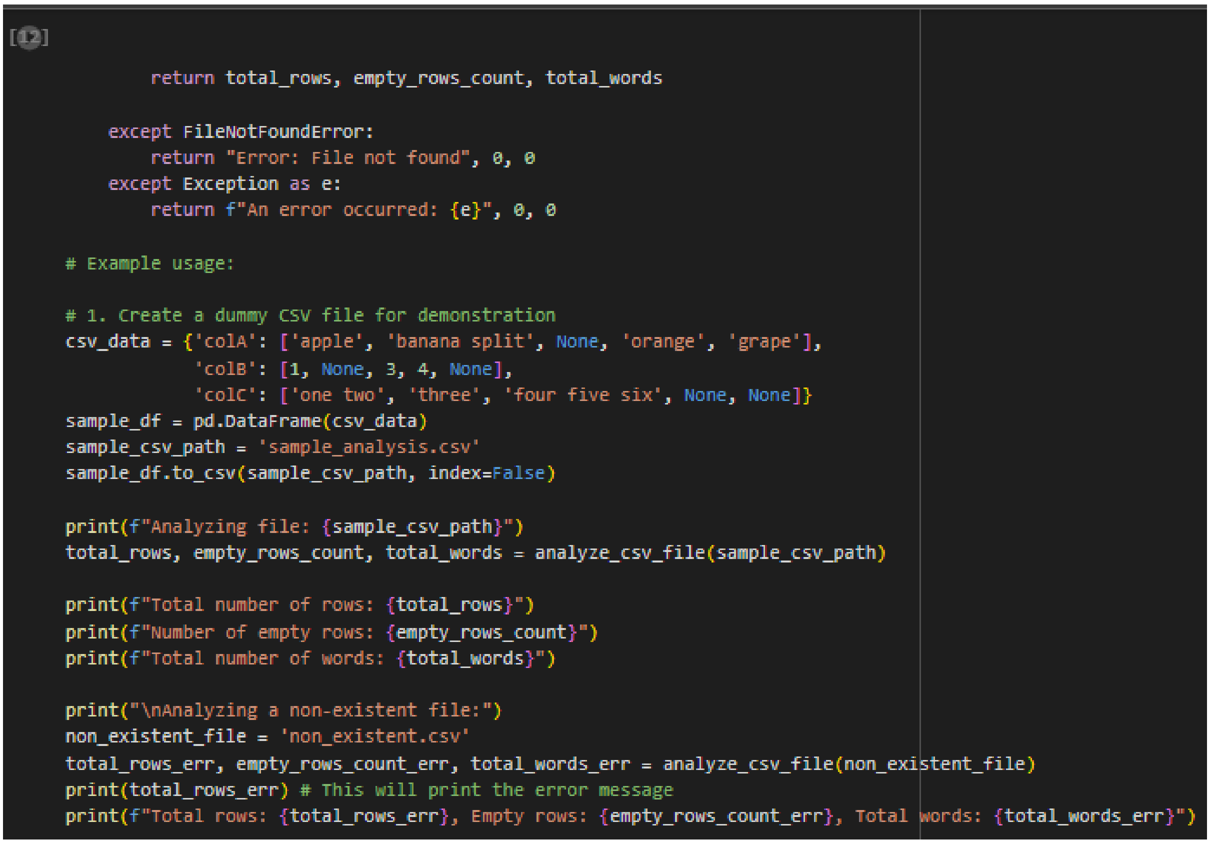
PROMPT 04:

i need a pyhon code functions for CSV file analysis. Requirements • 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 :

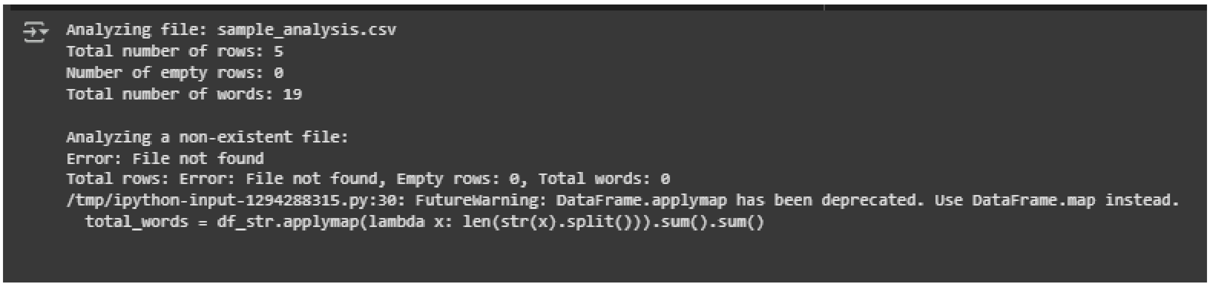


```
import pandas as pd
import re

def analyze_csv_file(file_path):
    """
    Analyzes a CSV file to determine total rows, empty rows, and total word count.

    Args:
        file_path: The path to the CSV file.

    Returns:
        A tuple containing:
            - The total number of rows in the DataFrame.
            - The number of empty rows in the DataFrame.
            - The total number of words in the DataFrame.
        Returns error messages and 0 counts if the file is not found or an error occurs.
    """
    try:
        df = pd.read_csv(file_path)
        total_rows = len(df)

        # Calculate the number of empty rows (where all values are NaN)
        empty_rows_count = df.isnull().all(axis=1).sum()

        # Calculate the total number of words
        total_words = 0
        # Convert the entire DataFrame to string type to handle mixed data types
        df_str = df.astype(str)
        # Apply a function to each element to count words
        total_words = df_str.applymap(lambda x: len(str(x).split())).sum().sum()


        return total_rows, empty_rows_count, total_words

    except FileNotFoundError:
        return "Error: File not found", 0, 0
```



```
        return total_rows, empty_rows_count, total_words

    except FileNotFoundError:
        return "Error: File not found", 0, 0
    except Exception as e:
        return f"An error occurred: {e}", 0, 0

# Example usage:

# 1. Create a dummy CSV file for demonstration
csv_data = {'colA': ['apple', 'banana split', None, 'orange', 'grape'],
            'colB': [1, None, 3, 4, None],
            'colC': ['one two', 'three', 'four five six', None, None]}
sample_df = pd.DataFrame(csv_data)
sample_csv_path = 'sample_analysis.csv'
sample_df.to_csv(sample_csv_path, index=False)

print(f"Analyzing file: {sample_csv_path}")
total_rows, empty_rows_count, total_words = analyze_csv_file(sample_csv_path)

print(f"Total number of rows: {total_rows}")
print(f"Number of empty rows: {empty_rows_count}")
print(f"Total number of words: {total_words}")

print("\nAnalyzing a non-existent file:")
non_existent_file = 'non_existent.csv'
total_rows_err, empty_rows_count_err, total_words_err = analyze_csv_file(non_existent_file)
print(total_rows_err) # This will print the error message
print(f"Total rows: {total_rows_err}, Empty rows: {empty_rows_count_err}, Total words: {total_words_err}")
```

OUTPUT :



```
Analyzing file: sample_analysis.csv
Total number of rows: 5
Number of empty rows: 0
Total number of words: 19

Analyzing a non-existent file:
Error: File not found
Total rows: Error: File not found, Empty rows: 0, Total words: 0
/tmp/ipython-input-1294288315.py:30: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  total_words = df_str.applymap(lambda x: len(str(x).split())).sum().sum()
```

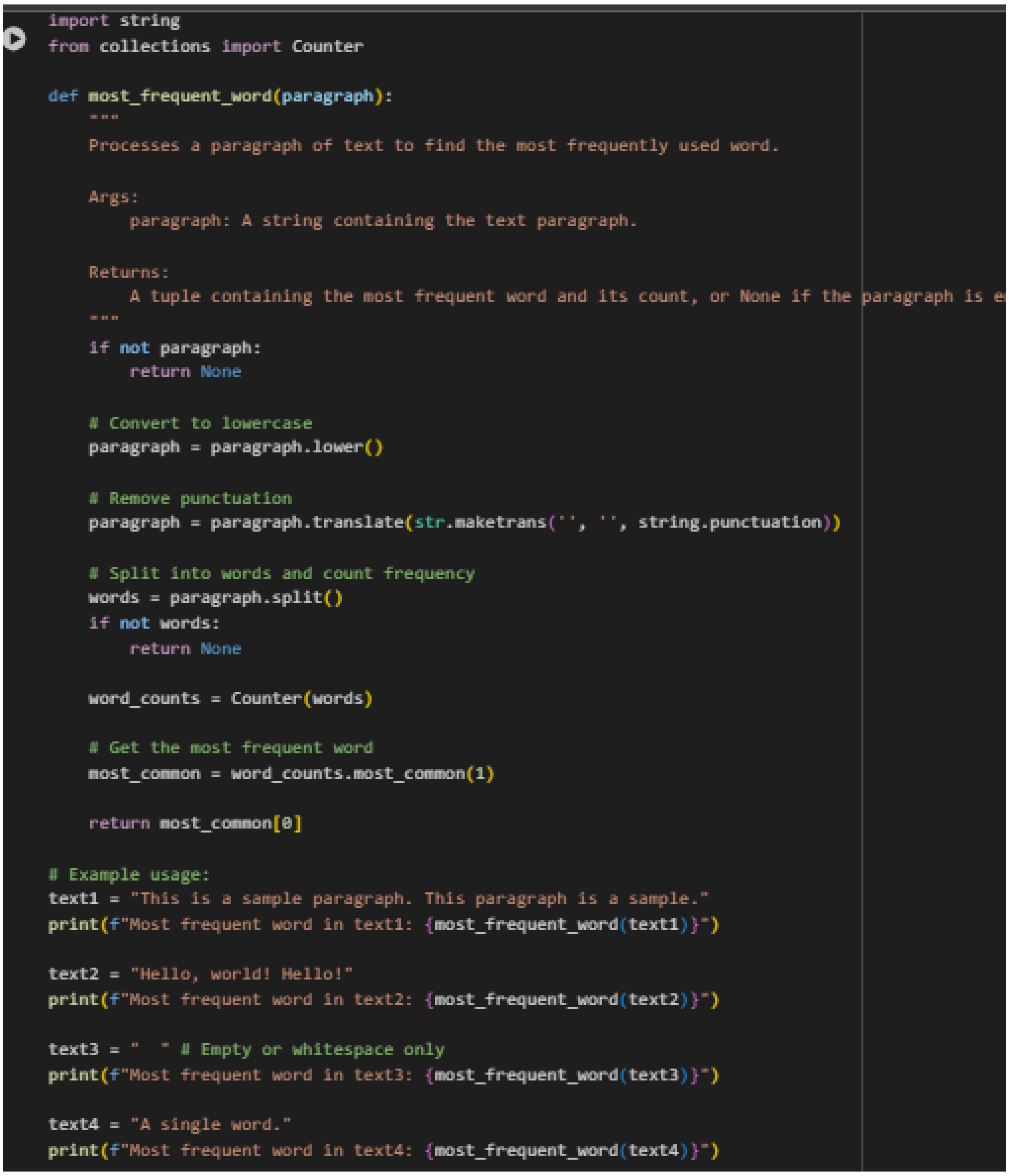
- **CODE EXPLANATION :**
  import pandas as pd: This line imports the pandas library, which is essential for working with DataFrames and reading CSV files, and assigns it the alias pd.
- import re: This line imports the regular expression module, although it's not directly used in the final version of 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 :

i need a python code function that processes text and analyzes word frequency. Requirements The function must: • Accept a paragraph as input • Convert all text to lowercase • Remove punctuation • Return the most frequently used word

CODE :



```python
import string
from collections import Counter

def most_frequent_word(paragraph):
    """
    Processes a paragraph of text to find the most frequently used word.

    Args:
        paragraph: A string containing the text paragraph.

    Returns:
        A tuple containing the most frequent word and its count, or None if the paragraph is e
    """
    if not paragraph:
        return None

    # Convert to lowercase
    paragraph = paragraph.lower()

    # Remove punctuation
    paragraph = paragraph.translate(str.maketrans('', '', string.punctuation))

    # Split into words and count frequency
    words = paragraph.split()
    if not words:
        return None

    word_counts = Counter(words)

    # Get the most frequent word
    most_common = word_counts.most_common(1)

    return most_common[0]

# Example usage:
text1 = "This is a sample paragraph. This paragraph is a sample."
print(f"Most frequent word in text1: {most_frequent_word(text1)}")

text2 = "Hello, world! Hello!"
print(f"Most frequent word in text2: {most_frequent_word(text2)}")

text3 = "   " # Empty or whitespace only
print(f"Most frequent word in text3: {most_frequent_word(text3)}")

text4 = "A single word."
print(f"Most frequent word in text4: {most_frequent_word(text4)}")
```

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.