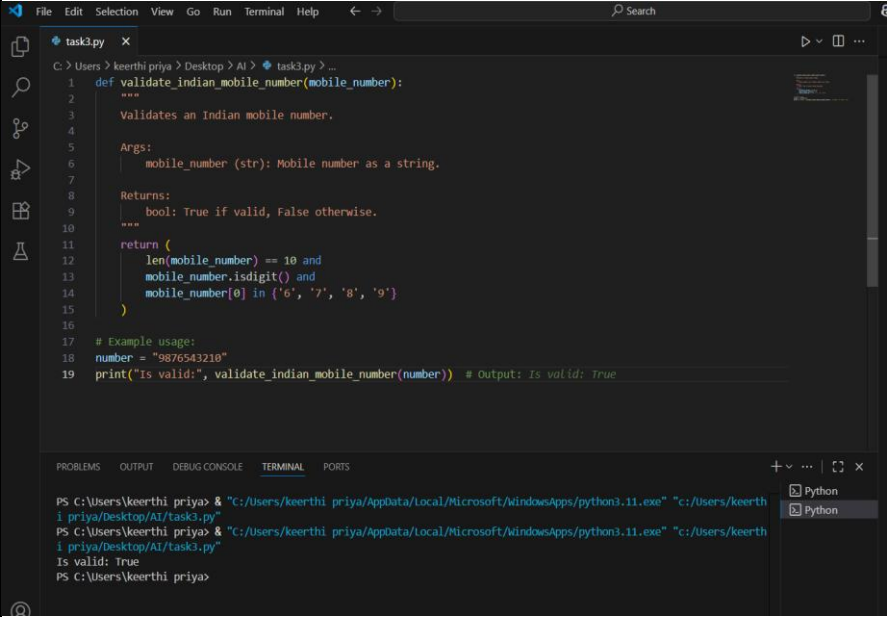


	<h2 style="text-align: center;">AI ASSISTED CODING</h2> <p>NAME: B.ARJUN ROLL NO: 2403A510A9 ASSIGNMENT: 4.1</p>	
	<p>Task #1 – Zero-Shot Prompting with Conditional Validation</p> <p>Objective Use zero-shot prompting to instruct an AI tool to generate a function that validates an Indian mobile number.</p> <p>Requirements</p> <ul style="list-style-type: none"> The function must ensure the mobile number: <ul style="list-style-type: none"> Starts with 6, 7, 8, or 9 Contains exactly 10 digits <p>Expected Output</p> <ul style="list-style-type: none"> A valid Python function that performs all required validations without using any input-output examples in the prompt. CODE WITH OUTPUT:  <p>The screenshot shows a VS Code editor with a file named 'task3.py'. The code defines a function 'validate_indian_mobile_number' that checks if a mobile number is 10 digits long, consists of digits, and starts with 6, 7, 8, or 9. Below the function, an example usage is shown: 'number = "9876543210"' and 'print("Is valid:", validate_indian_mobile_number(number))'. The terminal at the bottom shows the command to run the script and the output 'Is valid: True'.</p>	
	<p>Task #2 – One-Shot Prompting with Edge Case Handling</p> <p>Objective Use one-shot prompting to generate a Python function that calculates the factorial of a number.</p> <p>Requirements</p> <ul style="list-style-type: none"> Provide one sample input-output pair in the prompt to guide the AI. The function should handle: <ul style="list-style-type: none"> 0! correctly 	

- Negative input by returning an appropriate message
- Expected Output
- A Python function with correct factorial logic and edge case handling, generated from a single example.

The image displays two screenshots of a Visual Studio Code editor window. The top screenshot shows the source code for a Python function named `factorial(n)` in a file named `task3.py`. The code includes a docstring, type hints, and logic to handle negative, zero, and positive integer inputs. The bottom screenshot shows the same editor with sample input-output pairs added to the code and a terminal window at the bottom showing the execution results.

```

1  def factorial(n):
2      """
3      Calculates the factorial of a non-negative integer n.
4
5      Args:
6          n (int): The number to calculate factorial for.
7
8      Returns:
9          int or str: Factorial of n, or an error message for negative input.
10     """
11     if n < 0:
12         return "Factorial is not defined for negative numbers."
13     elif n == 0:
14         return 1
15     else:
16         result = 1
17         for i in range(1, n + 1):
18             result *= i
19         return result

```

```

20
21 # Sample input-output pair:
22 # Input: 5
23 # Output: 120
24 print(factorial(5)) # Output: 120
25 print(factorial(0)) # Output: 1
26 print(factorial(-3)) # Output: Factorial is not defined for negative

```

Terminal Output:

```

PS C:\Users\keerthi priya> & "C:/Users/keerthi priya/AppData/Local/Microsoft/windowsApps/python3.10.6/python.exe" I/task3.py
120
1
Factorial is not defined for negative numbers.
PS C:\Users\keerthi priya>

```

Task #3 – Few-Shot Prompting for Nested Dictionary Extraction

Objective

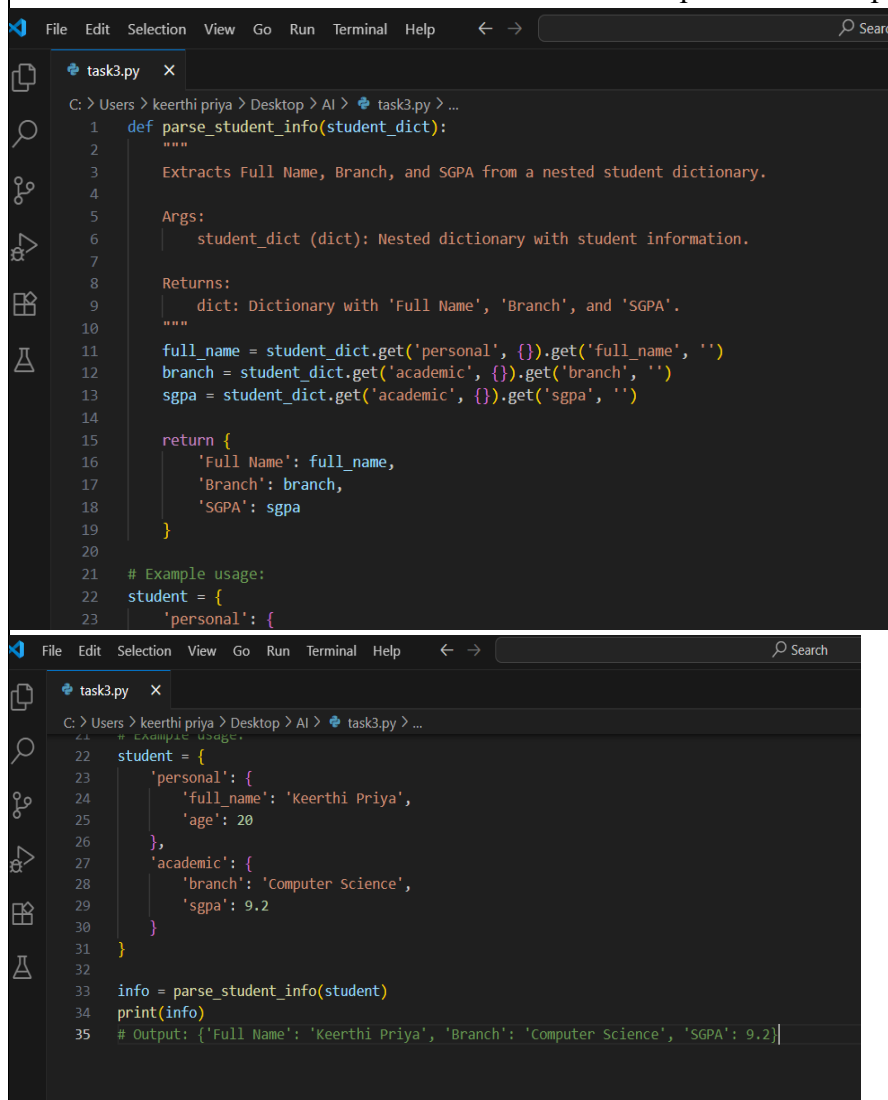
Use few-shot prompting (2–3 examples) to instruct the AI to create a function that parses a nested dictionary representing student information.

Requirements

- The function should extract and return:
 - Full Name
 - Branch
 - SGPA

Expected Output

- A reusable Python function that correctly navigates and extracts values from nested dictionaries based on the provided examples.



```
task3.py
C:\Users\keerthi priya > Desktop > AI > task3.py > ...
1  def parse_student_info(student_dict):
2      """
3      Extracts Full Name, Branch, and SGPA from a nested student dictionary.
4
5      Args:
6          student_dict (dict): Nested dictionary with student information.
7
8      Returns:
9          dict: Dictionary with 'Full Name', 'Branch', and 'SGPA'.
10     """
11     full_name = student_dict.get('personal', {}).get('full_name', '')
12     branch = student_dict.get('academic', {}).get('branch', '')
13     sgpa = student_dict.get('academic', {}).get('sgpa', '')
14
15     return {
16         'Full Name': full_name,
17         'Branch': branch,
18         'SGPA': sgpa
19     }
20
21 # Example usage:
22 student = {
23     'personal': {
24         'full_name': 'Keerthi Priya',
25         'age': 20
26     },
27     'academic': {
28         'branch': 'Computer Science',
29         'sgpa': 9.2
30     }
31 }
32
33 info = parse_student_info(student)
34 print(info)
35 # Output: {'Full Name': 'Keerthi Priya', 'Branch': 'Computer Science', 'SGPA': 9.2}
```

Task #4 – Comparing Prompting Styles for File Analysis

Objective

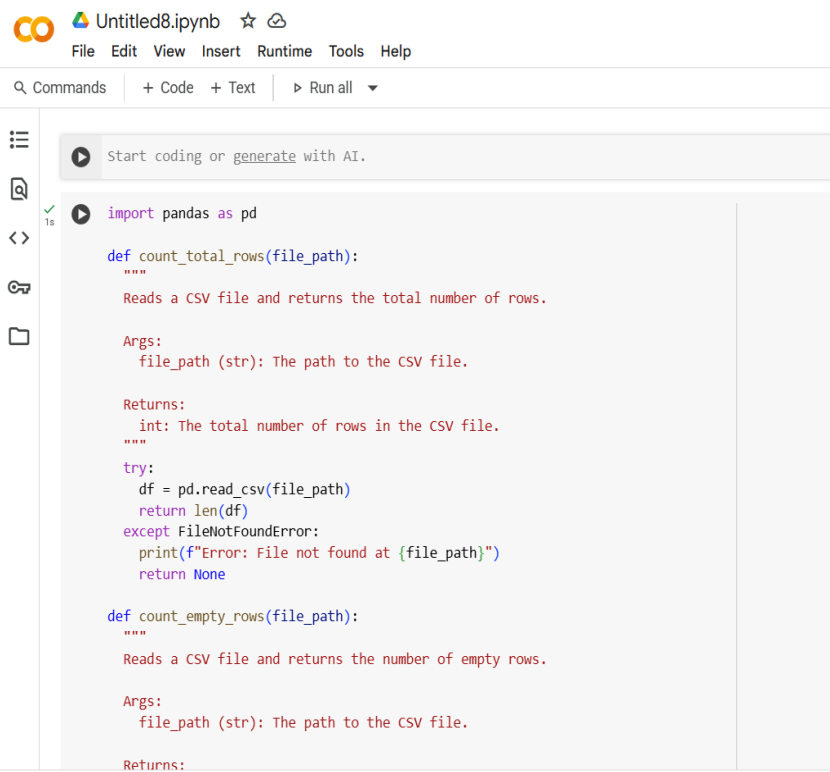
Experiment with zero-shot, one-shot, and few-shot prompting to generate functions for CSV file analysis.

Requirements

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

Expected Output

- Working Python functions for each prompting style, with a brief reflection comparing their accuracy, clarity, and efficiency.



The screenshot shows a Jupyter Notebook titled 'Untitled8.ipynb'. The interface includes a top bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help' menus. Below the menu bar is a search bar and a 'Run all' button. The notebook content area displays the following Python code:

```
import pandas as pd

def count_total_rows(file_path):
    """
    Reads a CSV file and returns the total number of rows.

    Args:
        file_path (str): The path to the CSV file.

    Returns:
        int: The total number of rows in the CSV file.
    """
    try:
        df = pd.read_csv(file_path)
        return len(df)
    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
        return None

def count_empty_rows(file_path):
    """
    Reads a CSV file and returns the number of empty rows.

    Args:
        file_path (str): The path to the CSV file.

    Returns:
```

The screenshot shows a Jupyter Notebook titled 'Untitled8.ipynb'. The code is as follows:

```

[1] empty_rows = df.isnull().all(axis=1).sum()
    return empty_rows
except FileNotFoundError:
    print(f"Error: File not found at {file_path}")
    return None

def count_total_words(file_path):
    """
    Reads a CSV file and returns the total number of words across all cells.

    Args:
        file_path (str): The path to the CSV file.

    Returns:
        int: The total number of words in the CSV file.
    """
    try:
        df = pd.read_csv(file_path)
        # Fill NaN values with empty strings to avoid errors in word count
        df = df.fillna('')
        total_words = df.apply(lambda row: ' '.join(row.values.astype(str)), axis=1).str.split().str.len().sum()
        return total_words
    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
        return None

You can use these functions by providing the path to your CSV file:

# Replace 'your_file.csv' with the actual path to your CSV file
file_path = 'your_file.csv'

return None

```

Below the code, there is a section titled 'You can use these functions by providing the path to your CSV file:' followed by a code block that demonstrates how to use the functions:

```

# Replace 'your_file.csv' with the actual path to your CSV file
file_path = 'your_file.csv'

total_rows = count_total_rows(file_path)
empty_rows = count_empty_rows(file_path)
total_words = count_total_words(file_path)

if total_rows is not None:
    print(f"Total number of rows: {total_rows}")
if empty_rows is not None:
    print(f"Number of empty rows: {empty_rows}")
if total_words is not None:
    print(f"Total number of words: {total_words}")

```

Task #5 – Few-Shot Prompting for Text Processing and Word Frequency

Objective

Use few-shot prompting (with at least 3 examples) to generate a Python 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

Expected Output

- A functional Python script that performs text cleaning,

tokenization, and returns the most common word using only the examples provided in the prompt

```
task3.py
C: > Users > keerthi priya > Desktop > AI > task3.py > ...
1 import string
2 from collections import Counter
3
4 def most_frequent_word(paragraph):
5     """
6     Processes text and returns the most frequently used word.
7
8     Args:
9         paragraph (str): Input paragraph.
10
11     Returns:
12         str: Most frequently used word.
13     """
14     # Convert to lowercase
15     text = paragraph.lower()
16     # Remove punctuation
17     text = text.translate(str.maketrans('', '', string.punctuation))
18     # Split into words
19     words = text.split()
20     # Count word frequency
21     word_counts = Counter(words)
22     # Get the most common word
23     if word_counts:
```

```
task3.py
C: > Users > keerthi priya > Desktop > AI > task3.py > ...
4 def most_frequent_word(paragraph):
22     # Get the most common word
23     if word_counts:
24         return word_counts.most_common(1)[0][0]
25     else:
26         return ""
27
28 # Example usage:
29 para = "Hello world! Hello everyone. Welcome to the world of Python."
30 print(most_frequent_word(para)) #
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\keerthi priya> & "C:/Users/keerthi priya/AppData/Local/Microsoft/Windows/
I/task3.py"
hello
PS C:\Users\keerthi priya>
```

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and

output and if required, screenshots

Evaluation Criteria:

Criteria	Max Marks
Zero Shot (Task #1)	0.5
One Shot (Task#2)	0.5
Few Shot (Task#3, Task#4 & Task #5)	1.5
Total	2.5 Marks