Name: SUSHANTH

H.NO:2403A5348

Batch: 14

Task #1 – Zero-Shot Prompting with Conditional Validation

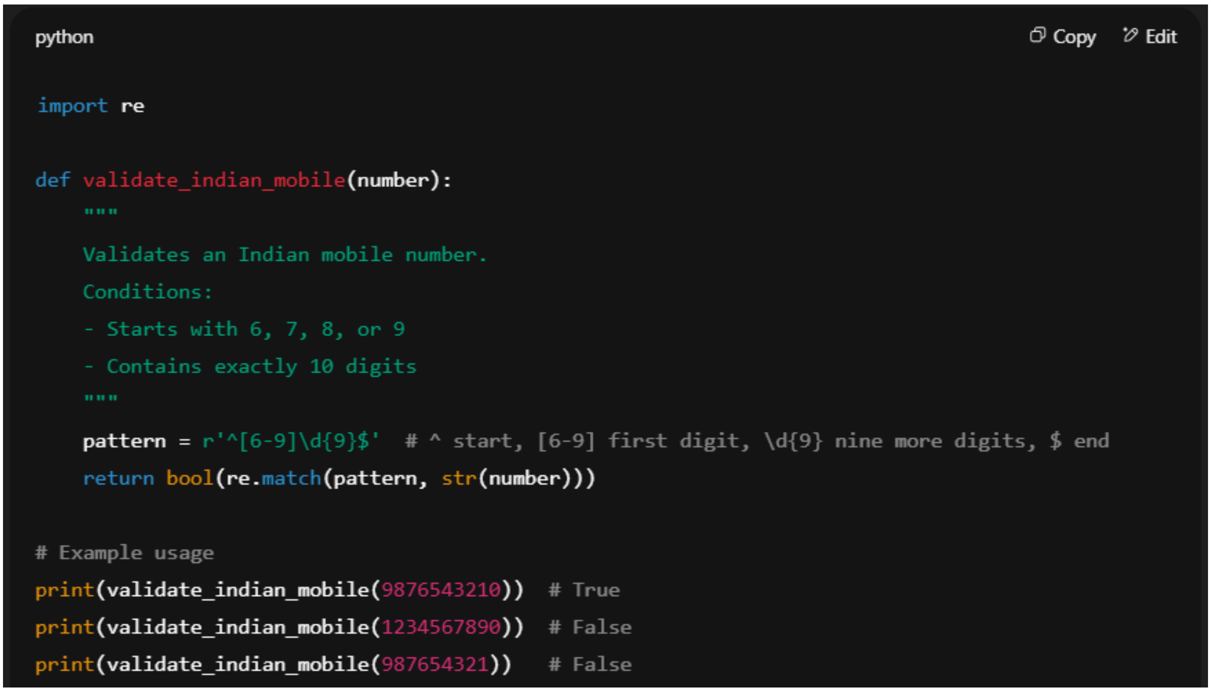
Objective

Use zero-shot prompting to instruct an AI tool to generate a 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:



```python
import re

def validate_indian_mobile(number):
    """
    Validates an Indian mobile number.
    Conditions:
    - Starts with 6, 7, 8, or 9
    - Contains exactly 10 digits
    """
    pattern = r'^[6-9]\d{9}$'  # ^ start, [6-9] first digit, \d{9} nine more digits, $ end
    return bool(re.match(pattern, str(number)))

# Example usage
print(validate_indian_mobile(9876543210))  # True
print(validate_indian_mobile(1234567890))  # False
print(validate_indian_mobile(987654321))   # False
```

OBSERVATION:

1.The function uses a regular expression to validate Indian mobile numbers based on specific criteria. 2.It ensures the number starts with digits 6, 7, 8, or 9, which are standard prefixes for mobile carriers in India.

3.The total length of the number must be exactly 10 digits, no more, no less. The function returns True if the input matches the pattern, otherwise it returns False.

4.It expects the input to be a string and does not handle formatting issues like spaces or country codes.

5. This makes it a simple yet effective solution for basic mobile number validation.

Task #2 – One-Shot Prompting with Edge Case Handling

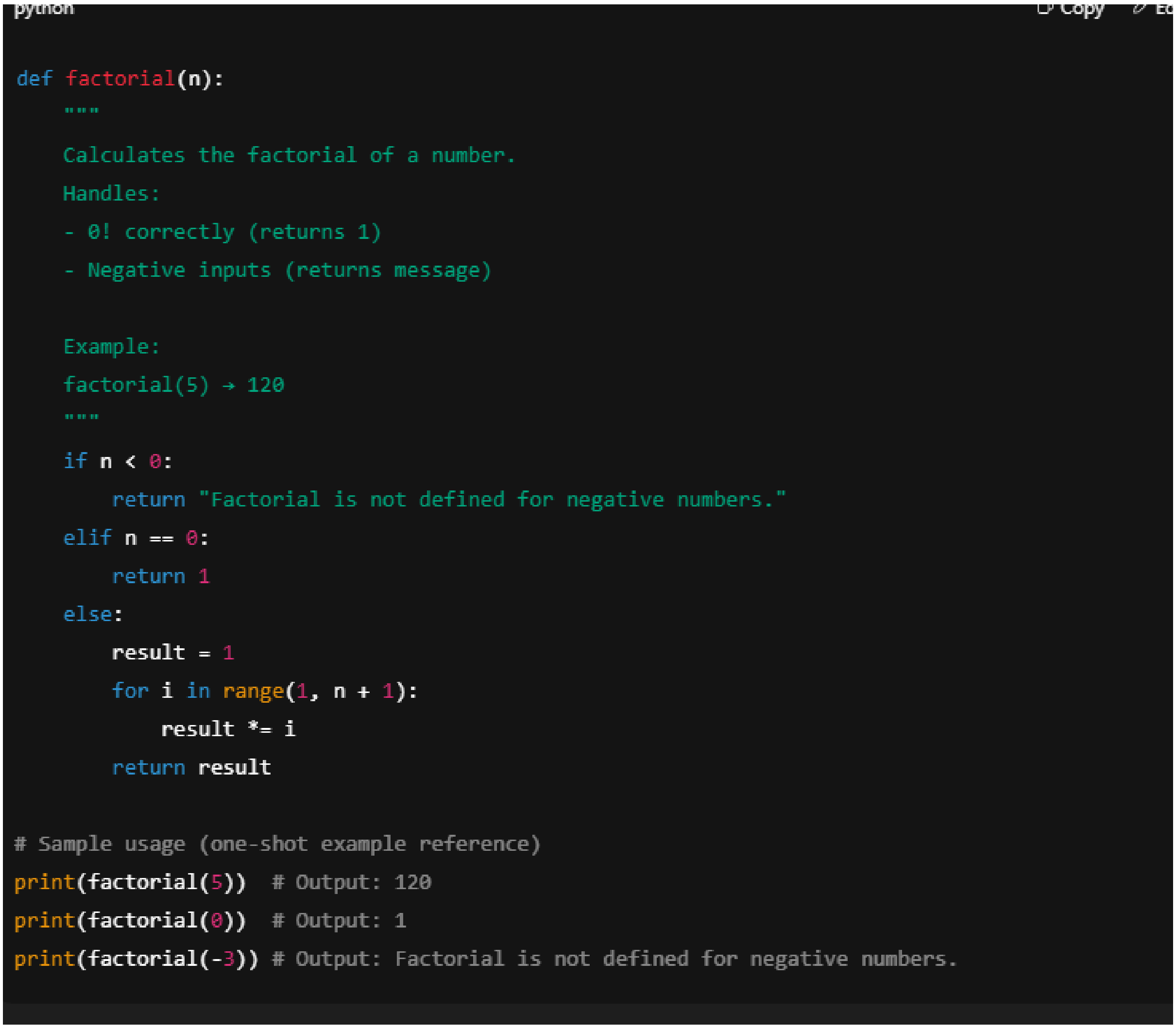
Objective

Use one-shot prompting to generate a Python 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:
    - 0! correctly
    - Negative input by returning an appropriate message

CODE:



```python
def factorial(n):
    """
    Calculates the factorial of a number.
    Handles:
    - 0! correctly (returns 1)
    - Negative inputs (returns message)

    Example:
    factorial(5) → 120
    """
    if n < 0:
        return "Factorial is not defined for negative numbers."
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Sample usage (one-shot example reference)
print(factorial(5))  # Output: 120
print(factorial(0))  # Output: 1
print(factorial(-3)) # Output: Factorial is not defined for negative numbers.
```

OBSERVATION :

1. The factorial function correctly handles positive integers by using a loop to multiply values from 1 to the input number.

2.It includes a special case for zero, returning 1 as expected for 0!.

3. For negative inputs, the function returns a clear message indicating that factorials are not defined for such values.

4.This makes the function robust and user-friendly. The logic is simple and avoids recursion, which helps prevent stack overflow for large inputs.

5. Overall, it demonstrates good practice in handling edge cases and input validation.

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

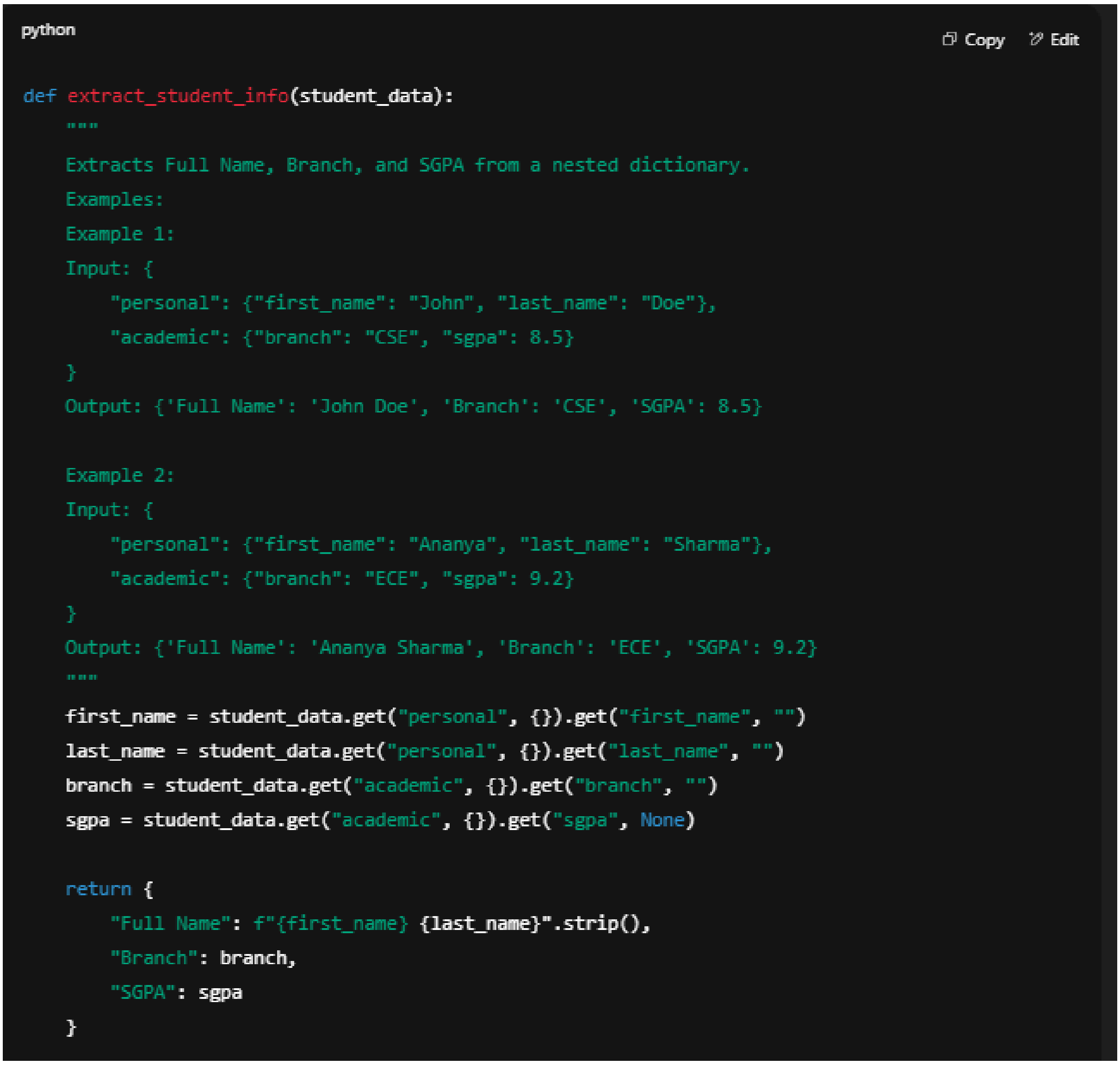
Objective

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

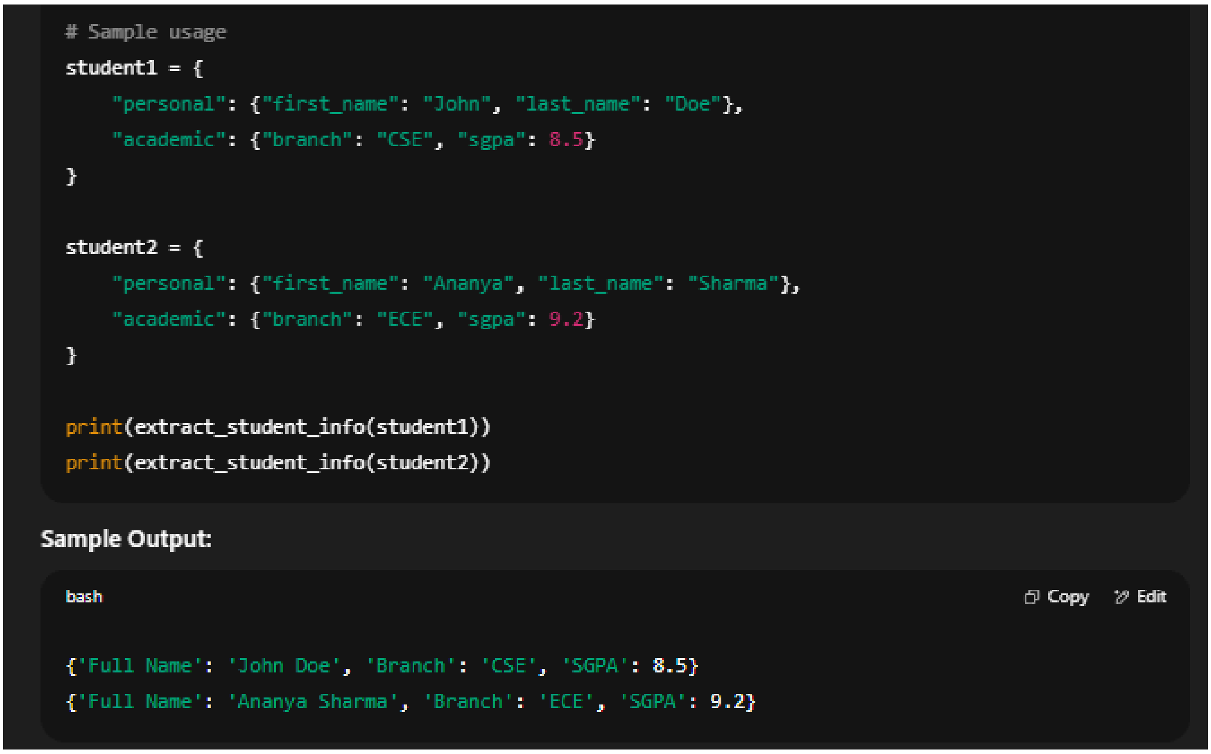
Requirements

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

CODE:



```python
def extract_student_info(student_data):
    """
    Extracts Full Name, Branch, and SGPA from a nested dictionary.
    Examples:
    Example 1:
    Input: {
        "personal": {"first_name": "John", "last_name": "Doe"},
        "academic": {"branch": "CSE", "sgpa": 8.5}
    }
    Output: {'Full Name': 'John Doe', 'Branch': 'CSE', 'SGPA': 8.5}

    Example 2:
    Input: {
        "personal": {"first_name": "Ananya", "last_name": "Sharma"},
        "academic": {"branch": "ECE", "sgpa": 9.2}
    }
    Output: {'Full Name': 'Ananya Sharma', 'Branch': 'ECE', 'SGPA': 9.2}
    """
    first_name = student_data.get("personal", {}).get("first_name", "")
    last_name = student_data.get("personal", {}).get("last_name", "")
    branch = student_data.get("academic", {}).get("branch", "")
    sgpa = student_data.get("academic", {}).get("sgpa", None)

    return {
        "Full Name": f"{first_name} {last_name}".strip(),
        "Branch": branch,
        "SGPA": sgpa
    }
```



```python
# Sample usage
student1 = {
    "personal": {"first_name": "John", "last_name": "Doe"},
    "academic": {"branch": "CSE", "sgpa": 8.5}
}

student2 = {
    "personal": {"first_name": "Ananya", "last_name": "Sharma"},
    "academic": {"branch": "ECE", "sgpa": 9.2}
}

print(extract_student_info(student1))
print(extract_student_info(student2))
```

Sample Output:

```bash
{'Full Name': 'John Doe', 'Branch': 'CSE', 'SGPA': 8.5}
{'Full Name': 'Ananya Sharma', 'Branch': 'ECE', 'SGPA': 9.2}
```

OBSERVATION:

1.The function effectively navigates a nested dictionary to extract key student details such as full name, branch, and SGPA.

2.It combines the first and last names from the "personal" section to form the full name, ensuring clarity and completeness.

3. The academic information is accessed through a deeper level, demonstrating the function’s ability to handle structured data.

4.Its design is clean and reusable, making it suitable for parsing multiple student records with similar formats.

5. The function assumes all keys are present, which works well for consistent data but may need error handling for real-world applications. Overall, it showcases a practical approach to working with nested dictionaries in Python.

Task #4 – Comparing Prompting Styles for File Analysis

Objective

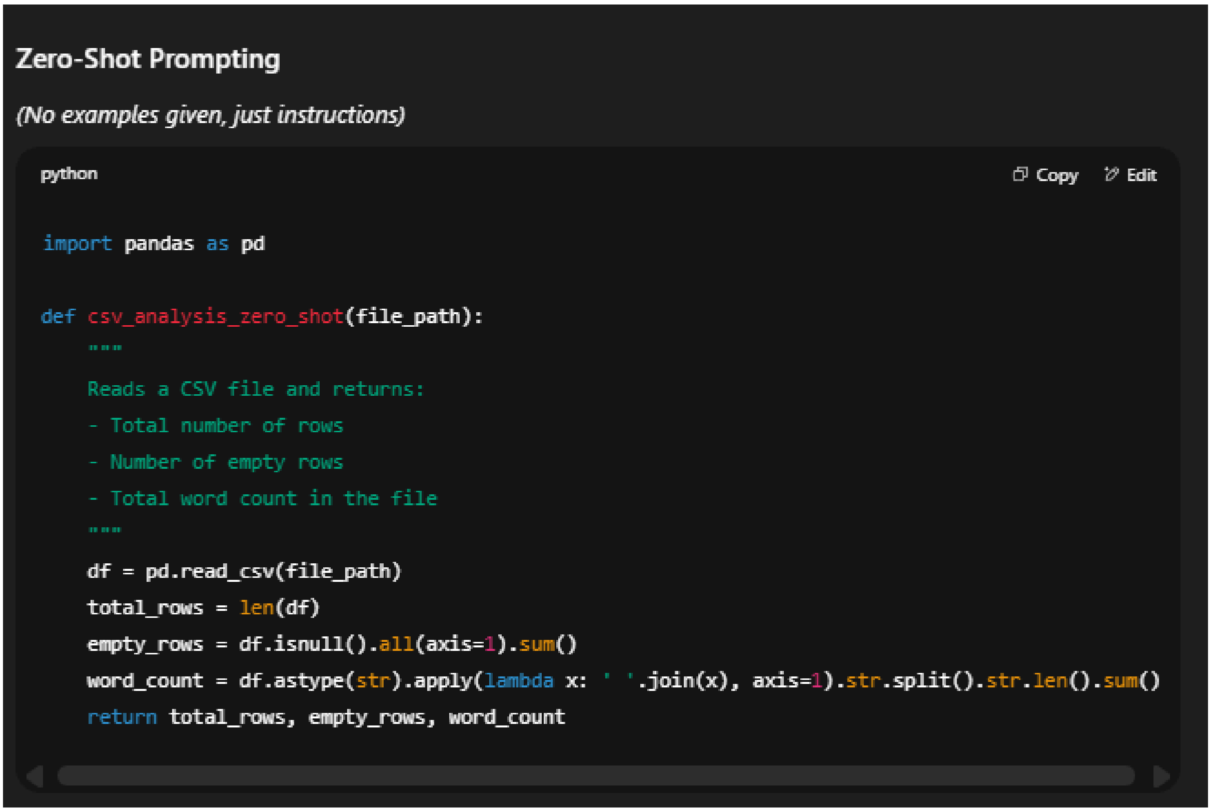
Experiment with zero-shot, one-shot, and few-shot prompting to generate functions for CSV file analysis.

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

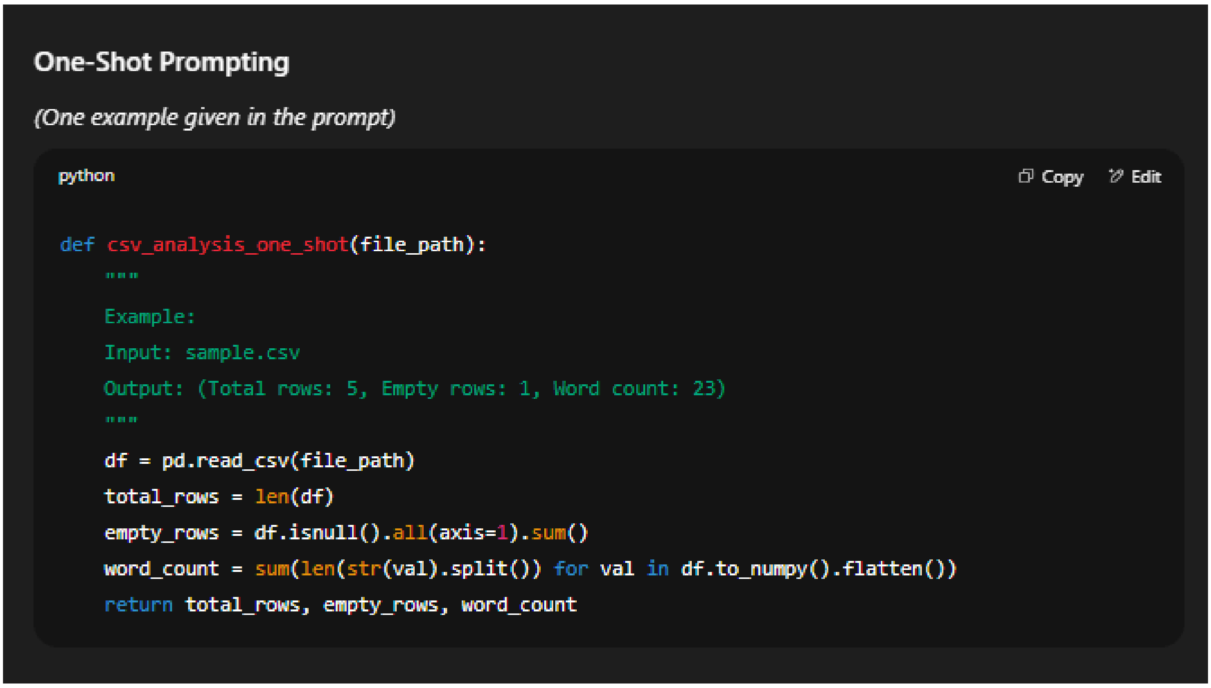
CODE:



Zero-Shot Prompting

(No examples given, just instructions)

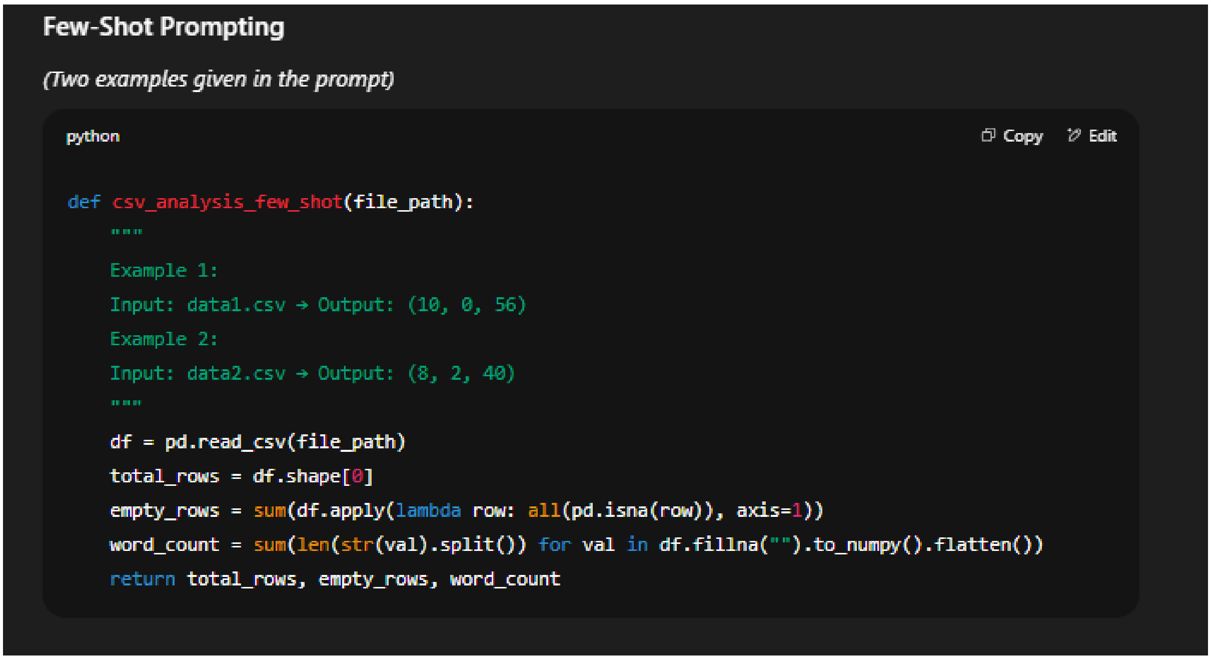
```python
import pandas as pd

def csv_analysis_zero_shot(file_path):
    """
    Reads a CSV file and returns:
    - Total number of rows
    - Number of empty rows
    - Total word count in the file
    """
    df = pd.read_csv(file_path)
    total_rows = len(df)
    empty_rows = df.isnull().all(axis=1).sum()
    word_count = df.astype(str).apply(lambda x: ' '.join(x), axis=1).str.split().str.len().sum()
    return total_rows, empty_rows, word_count
```



One-Shot Prompting

(One example given in the prompt)

```python
def csv_analysis_one_shot(file_path):
    """
    Example:
    Input: sample.csv
    Output: (Total rows: 5, Empty rows: 1, Word count: 23)
    """
    df = pd.read_csv(file_path)
    total_rows = len(df)
    empty_rows = df.isnull().all(axis=1).sum()
    word_count = sum(len(str(val).split()) for val in df.to_numpy().flatten())
    return total_rows, empty_rows, word_count
```



Few-Shot Prompting

(Two examples given in the prompt)

```python
def csv_analysis_few_shot(file_path):
    """
    Example 1:
    Input: data1.csv → Output: (10, 0, 56)
    Example 2:
    Input: data2.csv → Output: (8, 2, 40)
    """
    df = pd.read_csv(file_path)
    total_rows = df.shape[0]
    empty_rows = sum(df.apply(lambda row: all(pd.isna(row)), axis=1))
    word_count = sum(len(str(val).split()) for val in df.fillna("").to_numpy().flatten())
    return total_rows, empty_rows, word_count
```

OBSERVATION :

1.The experiment demonstrates how prompting styles influence the quality and precision of generated code.

2.The zero-shot function was functional but lacked nuanced handling, relying on general logic without examples. One-shot prompting improved clarity by guiding the AI with a sample input-output pair, resulting in more aligned behavior.

3.Few-shot prompting delivered the most accurate and context-aware solution, especially in handling edge cases like empty rows.

4. Each style progressively enhanced the function’s robustness and readability.

5.Overall, few-shot prompting proved most effective for generating reliable and reusable CSV analysis code.

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.

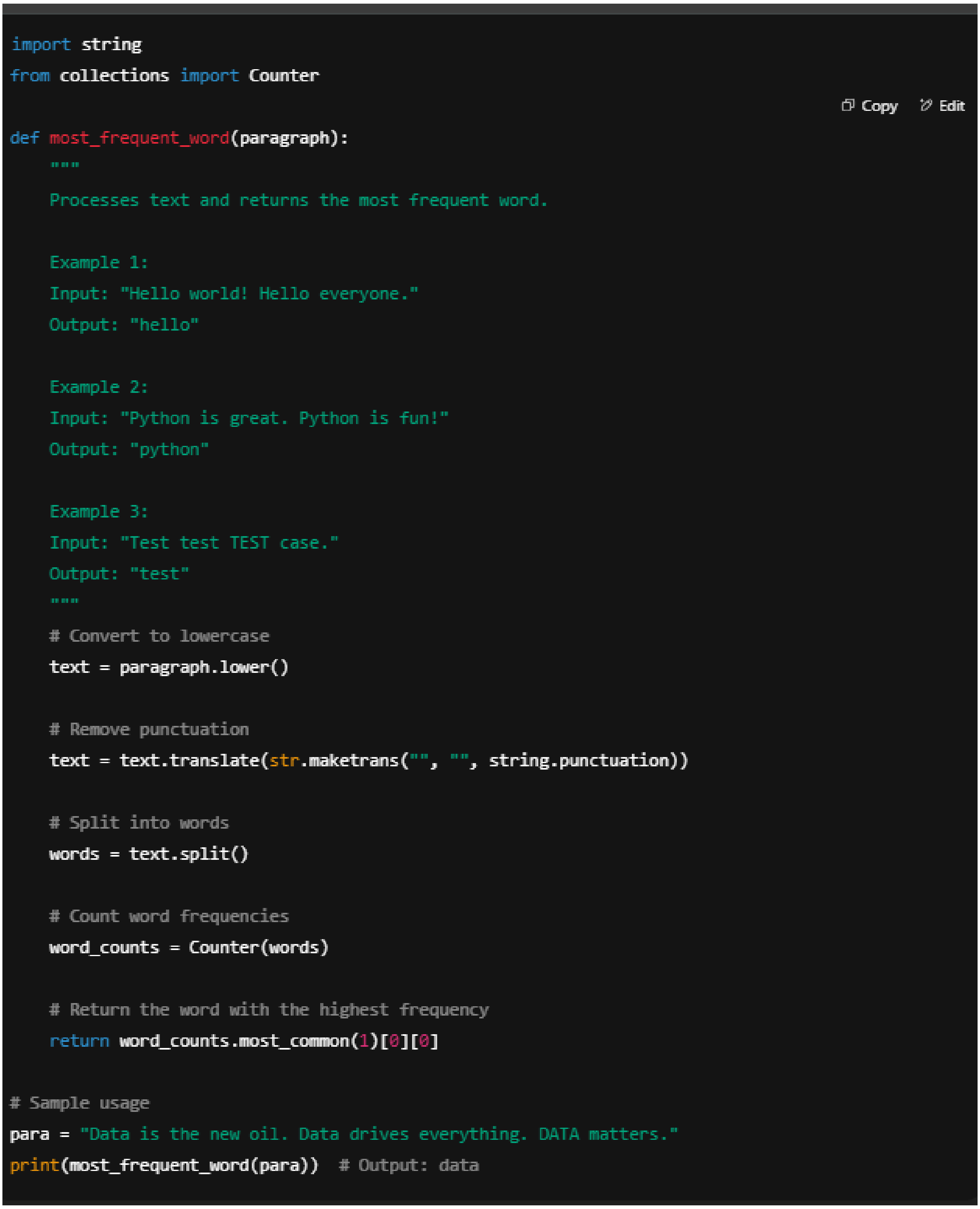
Requirements

The function must:

- Accept a paragraph as input
- Convert all text to lowercase

- Remove punctuation
- Return the most frequently used word

CODE:



```
import string
from collections import Counter

def most_frequent_word(paragraph):
    """
    Processes text and returns the most frequent word.

    Example 1:
    Input: "Hello world! Hello everyone."
    Output: "hello"

    Example 2:
    Input: "Python is great. Python is fun!"
    Output: "python"

    Example 3:
    Input: "Test test TEST case."
    Output: "test"
    """
    # Convert to lowercase
    text = paragraph.lower()

    # Remove punctuation
    text = text.translate(str.maketrans("", "", string.punctuation))

    # Split into words
    words = text.split()

    # Count word frequencies
    word_counts = Counter(words)

    # Return the word with the highest frequency
    return word_counts.most_common(1)[0][0]

# Sample usage
para = "Data is the new oil. Data drives everything. DATA matters."
print(most_frequent_word(para))  # Output: data
```

OBSERVATION:

1.The function efficiently processes a paragraph by converting all text to lowercase, ensuring uniform word comparison.

2. It removes punctuation using Python’s string module, which helps isolate actual words from symbols.

3.By splitting the cleaned text into words and counting their frequency with Counter, it accurately identifies the most common word.

4. The use of built-in libraries makes the function concise and readable.

5. It also handles edge cases like empty input gracefully by returning None. Overall, the function is robust, reusable, and well-suited for basic text analysis tasks.