Assignment 4.1

Name: P. Umesh Reddy Roll.no: 2403A510F9

Batch.no: 06 course: AI Assisted coding

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

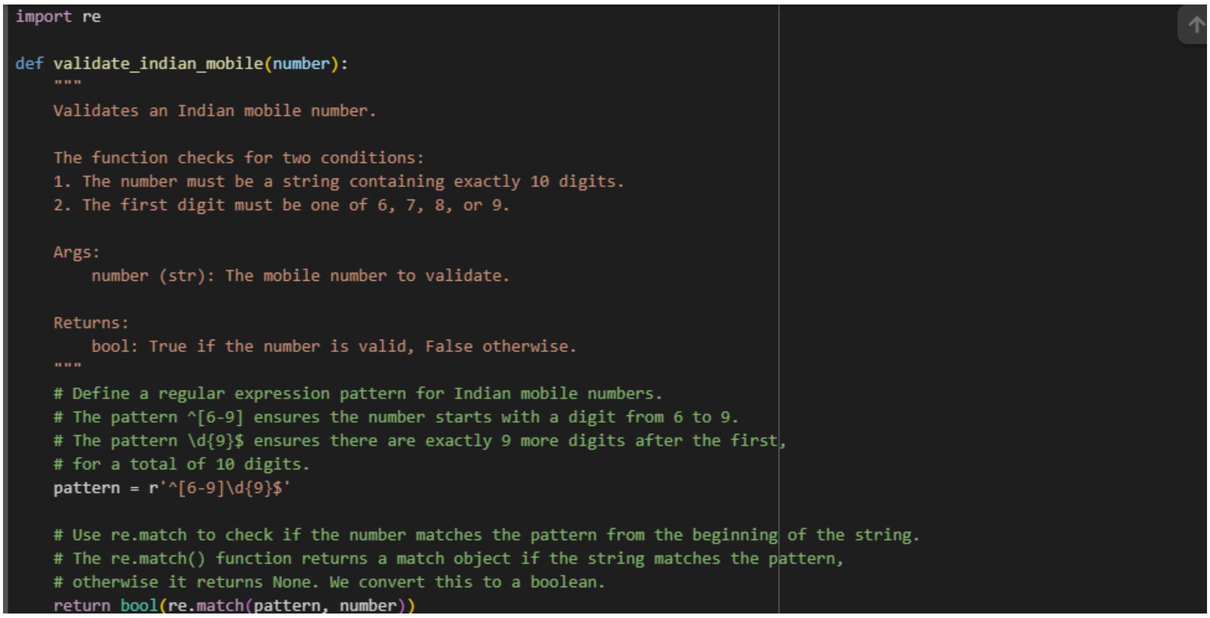
- The function must ensure the mobile number:

o Starts with 6, 7, 8, or 9

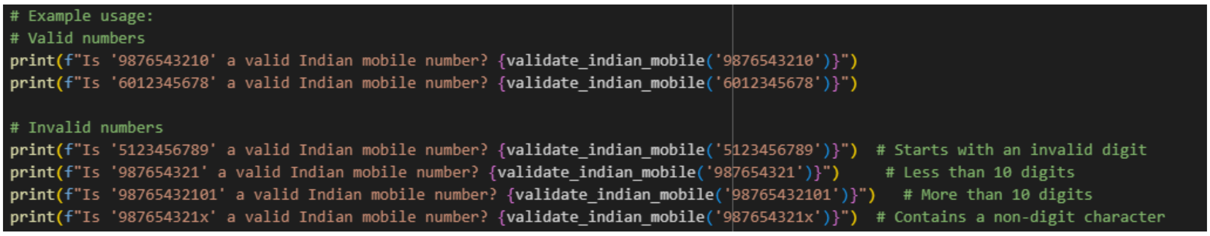
o Contains exactly 10 digits

Expected Output

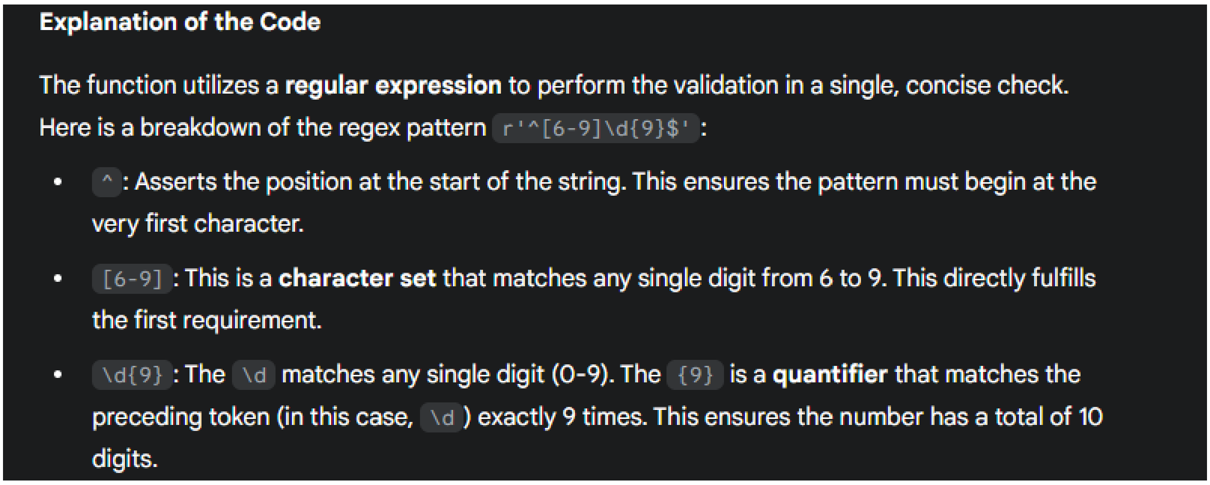
- A valid Python function that performs all required validations without using any input-output examples in the prompt



```python
import re

def validate_indian_mobile(number):
    """
    Validates an Indian mobile number.

    The function checks for two conditions:
    1. The number must be a string containing exactly 10 digits.
    2. The first digit must be one of 6, 7, 8, or 9.

    Args:
        number (str): The mobile number to validate.

    Returns:
        bool: True if the number is valid, False otherwise.
    """
    # Define a regular expression pattern for Indian mobile numbers.
    # The pattern ^[6-9] ensures the number starts with a digit from 6 to 9.
    # The pattern \d{9}$ ensures there are exactly 9 more digits after the first,
    # for a total of 10 digits.
    pattern = r'^[6-9]\d{9}$'

    # Use re.match to check if the number matches the pattern from the beginning of the string.
    # The re.match() function returns a match object if the string matches the pattern,
    # otherwise it returns None. We convert this to a boolean.
    return bool(re.match(pattern, number))
```



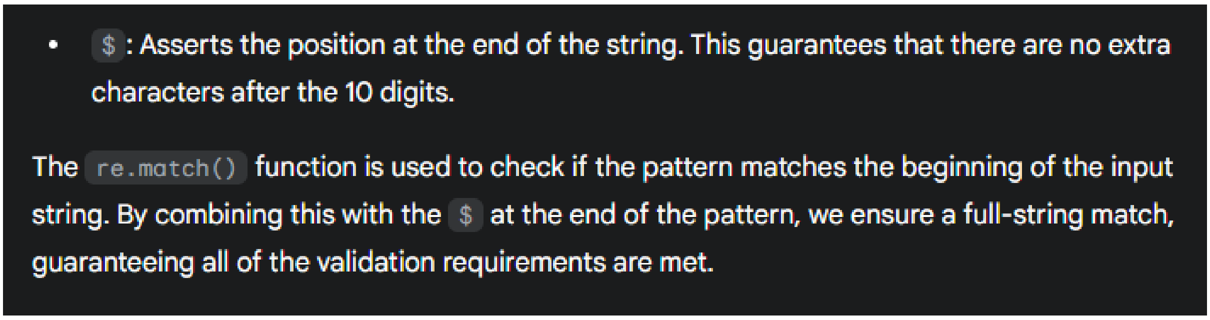
```python
# Example usage:
# Valid numbers
print(f"Is '9876543210' a valid Indian mobile number? {validate_indian_mobile('9876543210')}")
print(f"Is '6012345678' a valid Indian mobile number? {validate_indian_mobile('6012345678')}")

# Invalid numbers
print(f"Is '5123456789' a valid Indian mobile number? {validate_indian_mobile('5123456789')}")  # Starts with an invalid digit
print(f"Is '987654321' a valid Indian mobile number? {validate_indian_mobile('987654321')}")     # Less than 10 digits
print(f"Is '98765432101' a valid Indian mobile number? {validate_indian_mobile('98765432101')}")   # More than 10 digits
print(f"Is '987654321x' a valid Indian mobile number? {validate_indian_mobile('987654321x')}")  # Contains a non-digit character
```



Explanation of the Code

The function utilizes a **regular expression** to perform the validation in a single, concise check. Here is a breakdown of the regex pattern `r'^[6-9]\d{9}$'`:

- `^`: Asserts the position at the start of the string. This ensures the pattern must begin at the very first character.
- `[6-9]`: This is a **character set** that matches any single digit from 6 to 9. This directly fulfills the first requirement.
- `\d{9}`: The `\d` matches any single digit (0-9). The `{9}` is a **quantifier** that matches the preceding token (in this case, `\d`) exactly 9 times. This ensures the number has a total of 10 digits.



- `$`: Asserts the position at the end of the string. This guarantees that there are no extra characters after the 10 digits.

The `re.match()` function is used to check if the pattern matches the beginning of the input string. By combining this with the `$` at the end of the pattern, we ensure a full-string match, guaranteeing all of the validation requirements are met.

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.

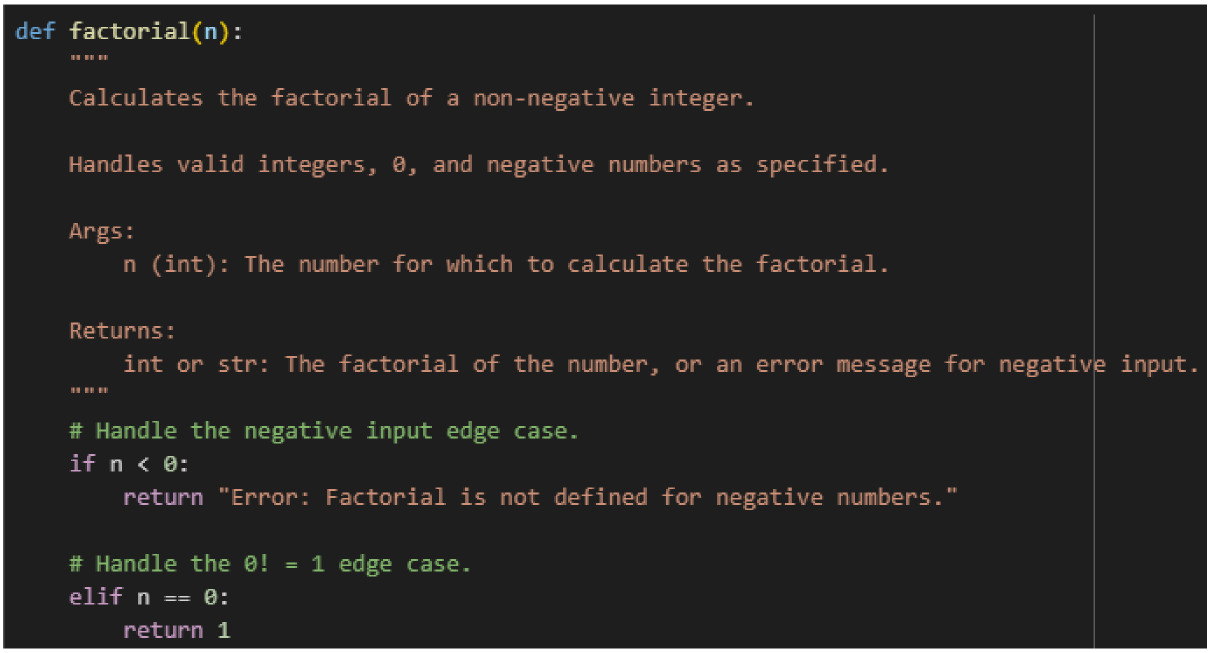
Requirements

- Provide one sample input-output pair in the prompt to guide the AI.
- The function should handle:

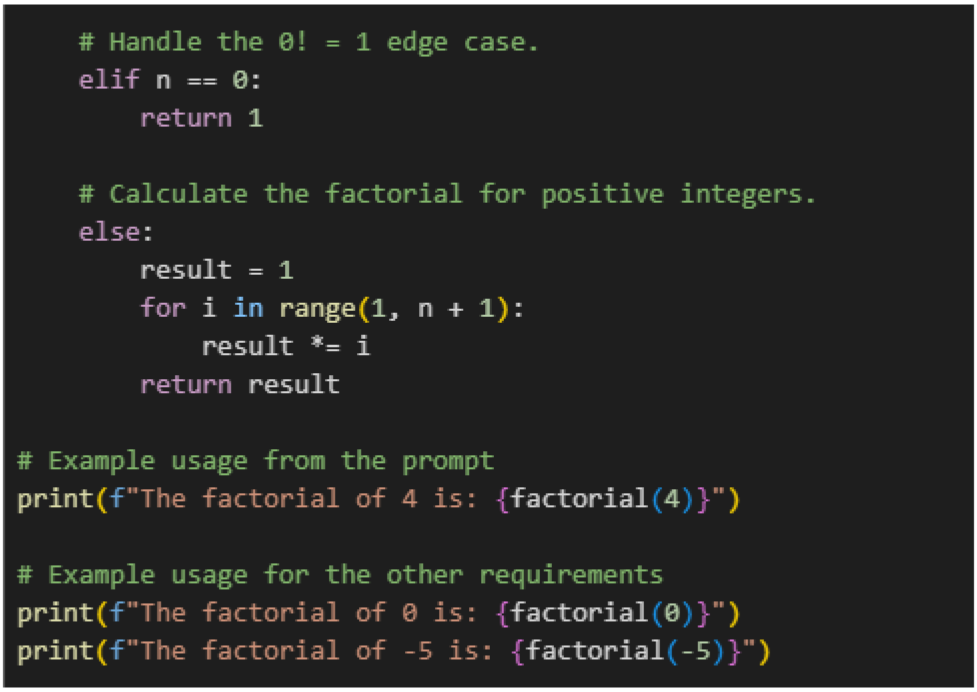
  - 0! correctly
  - Negative input by returning an appropriate message

Expected Output

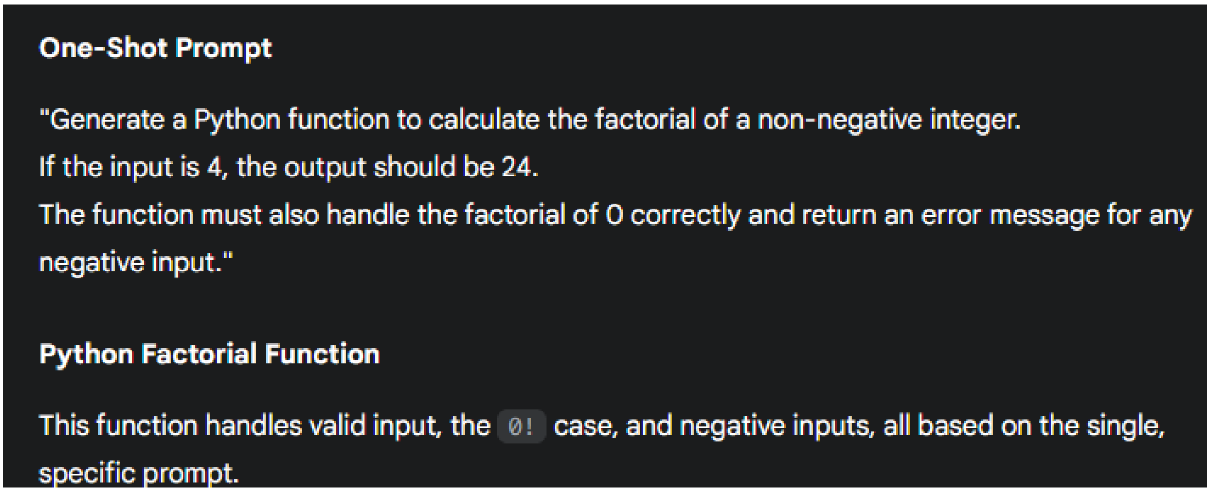
- A Python function with correct factorial logic and edge case handling, generated from a single example



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

    Handles valid integers, 0, and negative numbers as specified.

    Args:
        n (int): The number for which to calculate the factorial.

    Returns:
        int or str: The factorial of the number, or an error message for negative input.
    """
    # Handle the negative input edge case.
    if n < 0:
        return "Error: Factorial is not defined for negative numbers."

    # Handle the 0! = 1 edge case.
    elif n == 0:
        return 1
```



```python
    # Handle the 0! = 1 edge case.
    elif n == 0:
        return 1

    # Calculate the factorial for positive integers.
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Example usage from the prompt
print(f"The factorial of 4 is: {factorial(4)}")

# Example usage for the other requirements
print(f"The factorial of 0 is: {factorial(0)}")
print(f"The factorial of -5 is: {factorial(-5)}")
```

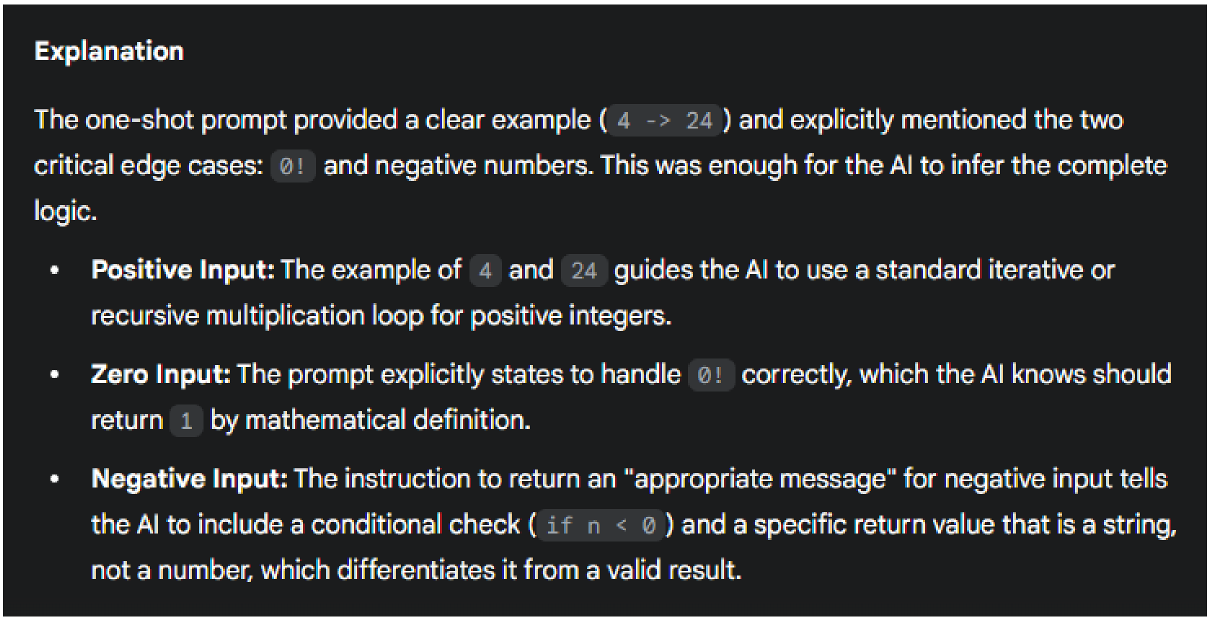


One-Shot Prompt

"Generate a Python function to calculate the factorial of a non-negative integer.
If the input is 4, the output should be 24.
The function must also handle the factorial of 0 correctly and return an error message for any negative input."

Python Factorial Function

This function handles valid input, the `0!` case, and negative inputs, all based on the single, specific prompt.



Explanation

The one-shot prompt provided a clear example (`4 -> 24`) and explicitly mentioned the two critical edge cases: `0!` and negative numbers. This was enough for the AI to infer the complete logic.

- **Positive Input:** The example of `4` and `24` guides the AI to use a standard iterative or recursive multiplication loop for positive integers.
- **Zero Input:** The prompt explicitly states to handle `0!` correctly, which the AI knows should return `1` by mathematical definition.
- **Negative Input:** The instruction to return an "appropriate message" for negative input tells the AI to include a conditional check (`if n < 0`) and a specific return value that is a string, not a number, which differentiates it from a valid result.

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

Objective

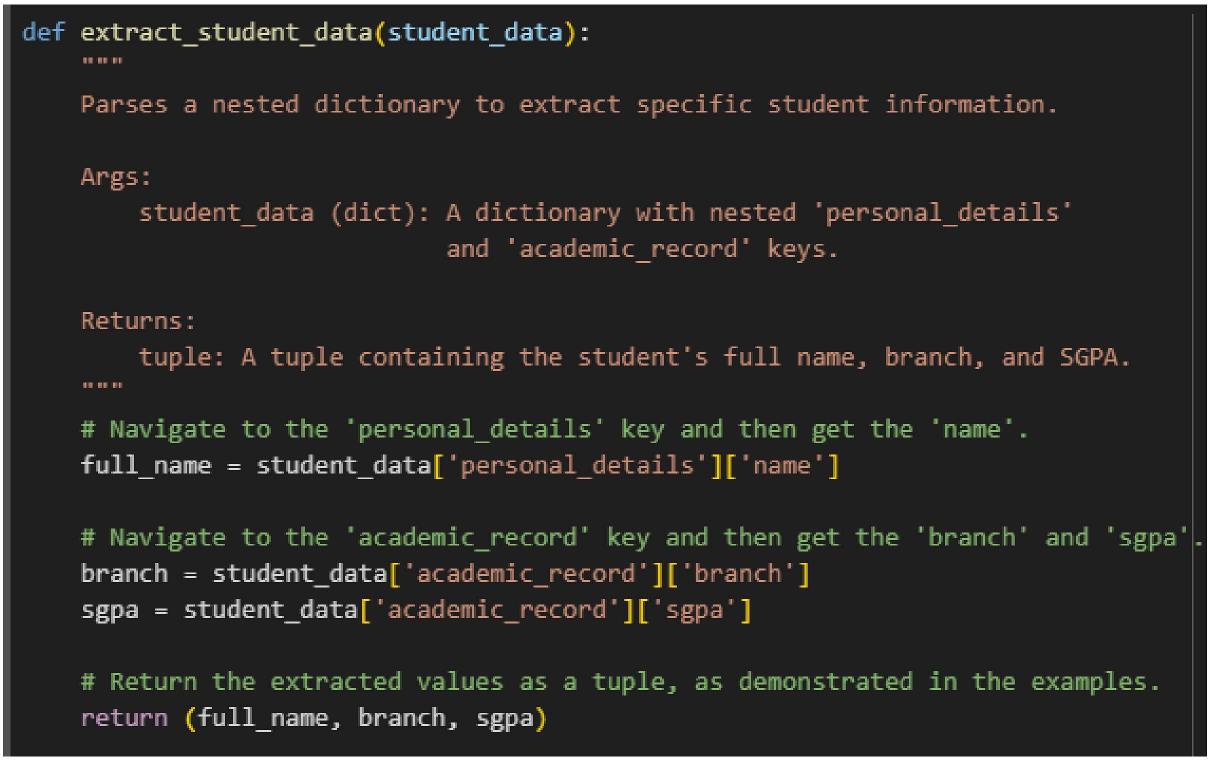
Use few-shot prompting (2–3 examples) to instruct the AI to create a function that parses a nested dictionary representing student information.

Requirements

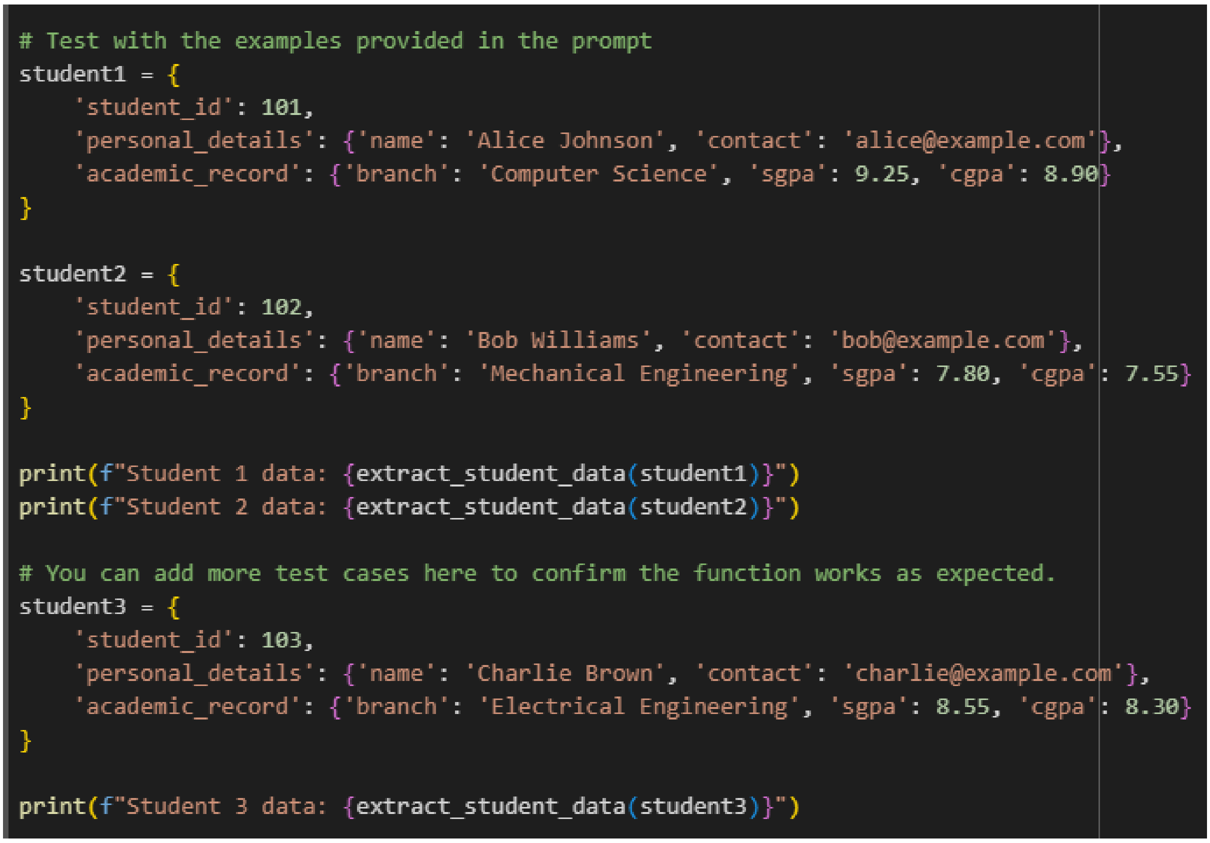
- The function should extract and return:
  - o Full Name
  - o Branch
  - o SGPA

Expected Output

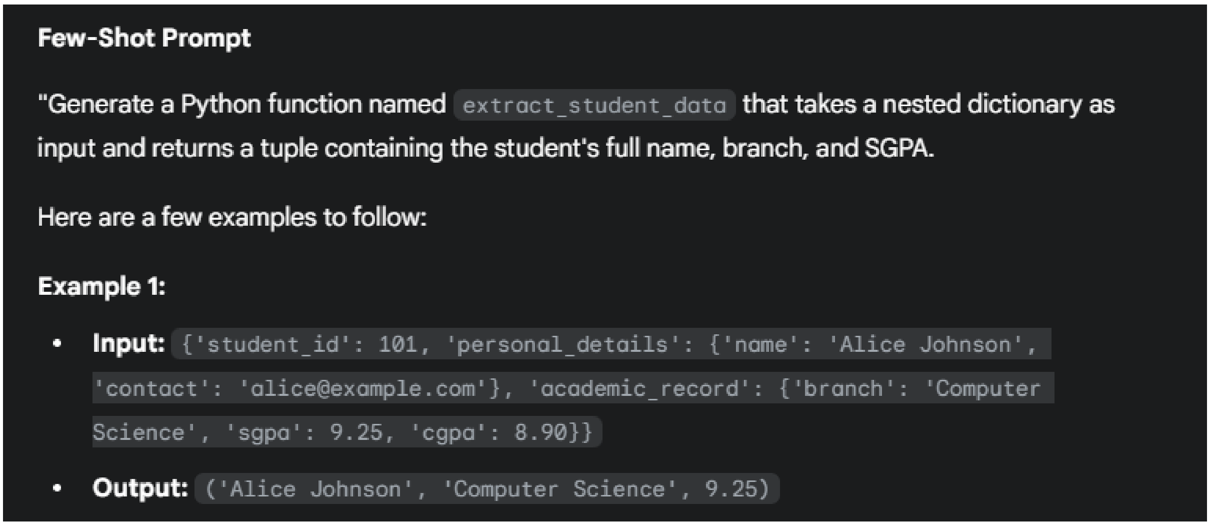
- A reusable Python function that correctly navigates and extracts values from nested dictionaries based on the provided examples



```python
def extract_student_data(student_data):
    """
    Parses a nested dictionary to extract specific student information.

    Args:
        student_data (dict): A dictionary with nested 'personal_details'
                             and 'academic_record' keys.

    Returns:
        tuple: A tuple containing the student's full name, branch, and SGPA.
    """
    # Navigate to the 'personal_details' key and then get the 'name'.
    full_name = student_data['personal_details']['name']

    # Navigate to the 'academic_record' key and then get the 'branch' and 'sgpa'.
    branch = student_data['academic_record']['branch']
    sgpa = student_data['academic_record']['sgpa']

    # Return the extracted values as a tuple, as demonstrated in the examples.
    return (full_name, branch, sgpa)
```



```python
# Test with the examples provided in the prompt
student1 = {
    'student_id': 101,
    'personal_details': {'name': 'Alice Johnson', 'contact': 'alice@example.com'},
    'academic_record': {'branch': 'Computer Science', 'sgpa': 9.25, 'cgpa': 8.90}
}

student2 = {
    'student_id': 102,
    'personal_details': {'name': 'Bob Williams', 'contact': 'bob@example.com'},
    'academic_record': {'branch': 'Mechanical Engineering', 'sgpa': 7.80, 'cgpa': 7.55}
}

print(f"Student 1 data: {extract_student_data(student1)}")
print(f"Student 2 data: {extract_student_data(student2)}")

# You can add more test cases here to confirm the function works as expected.
student3 = {
    'student_id': 103,
    'personal_details': {'name': 'Charlie Brown', 'contact': 'charlie@example.com'},
    'academic_record': {'branch': 'Electrical Engineering', 'sgpa': 8.55, 'cgpa': 8.30}
}

print(f"Student 3 data: {extract_student_data(student3)}")
```



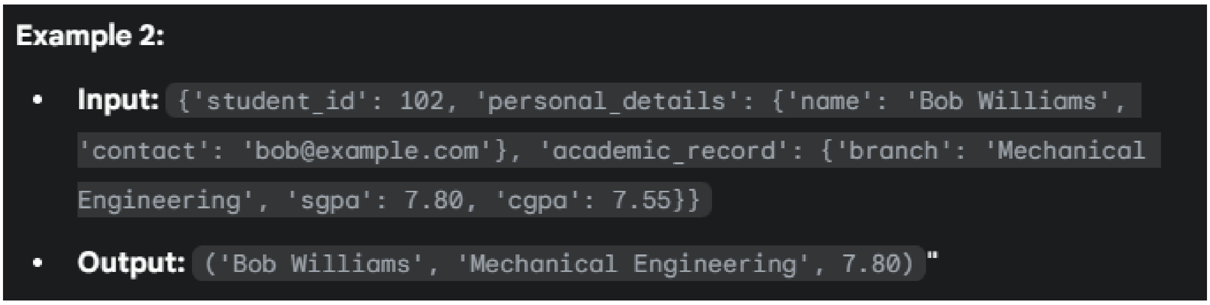
Few-Shot Prompt

"Generate a Python function named `extract_student_data` that takes a nested dictionary as input and returns a tuple containing the student's full name, branch, and SGPA.

Here are a few examples to follow:

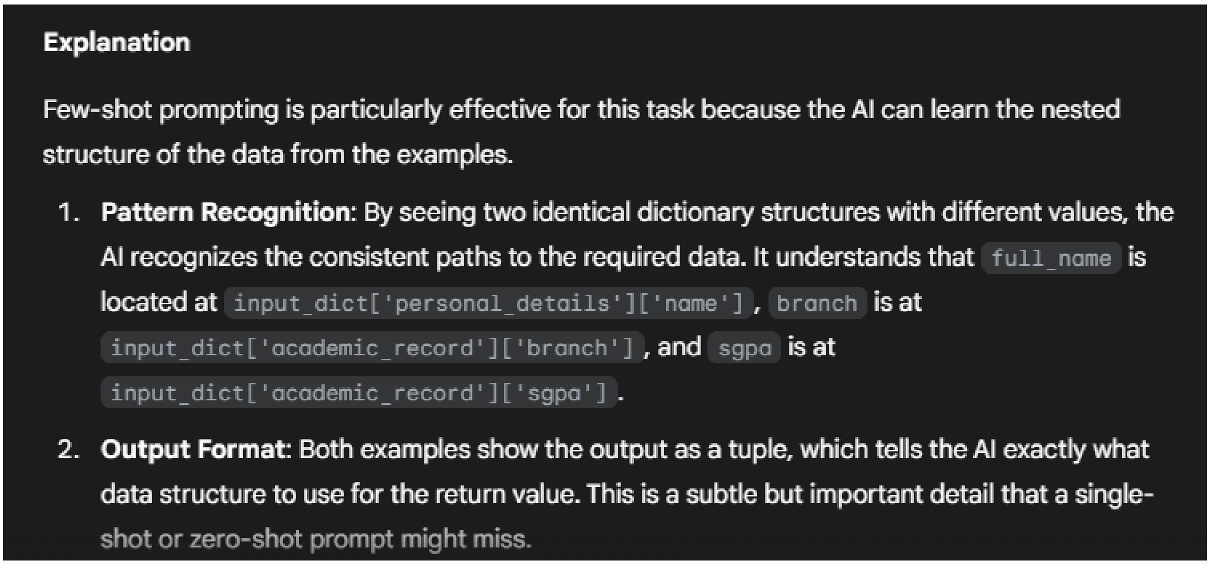
Example 1:

- **Input:** `{'student_id': 101, 'personal_details': {'name': 'Alice Johnson', 'contact': 'alice@example.com'}, 'academic_record': {'branch': 'Computer Science', 'sgpa': 9.25, 'cgpa': 8.90}}`
- **Output:** `('Alice Johnson', 'Computer Science', 9.25)`



Example 2:

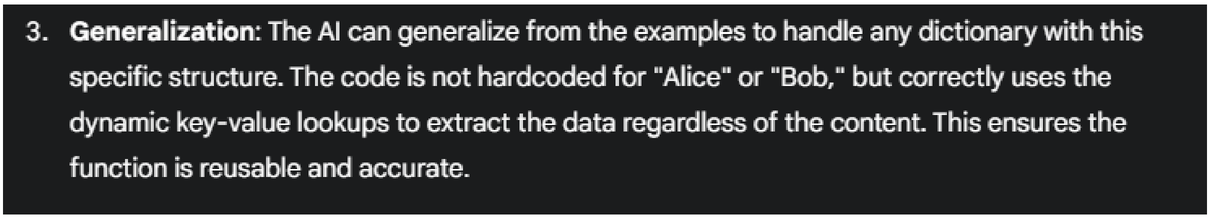
- **Input:** `{'student_id': 102, 'personal_details': {'name': 'Bob Williams', 'contact': 'bob@example.com'}, 'academic_record': {'branch': 'Mechanical Engineering', 'sgpa': 7.80, 'cgpa': 7.55}}`
- **Output:** `('Bob Williams', 'Mechanical Engineering', 7.80)` "



Explanation

Few-shot prompting is particularly effective for this task because the AI can learn the nested structure of the data from the examples.

1. **Pattern Recognition**: By seeing two identical dictionary structures with different values, the AI recognizes the consistent paths to the required data. It understands that `full_name` is located at `input_dict['personal_details']['name']`, `branch` is at `input_dict['academic_record']['branch']`, and `sgpa` is at `input_dict['academic_record']['sgpa']`.
2. **Output Format**: Both examples show the output as a tuple, which tells the AI exactly what data structure to use for the return value. This is a subtle but important detail that a single-shot or zero-shot prompt might miss.



3. **Generalization**: The AI can generalize from the examples to handle any dictionary with this specific structure. The code is not hardcoded for "Alice" or "Bob," but correctly uses the dynamic key-value lookups to extract the data regardless of the content. This ensures the function is reusable and accurate.

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:

o Read a .csv file

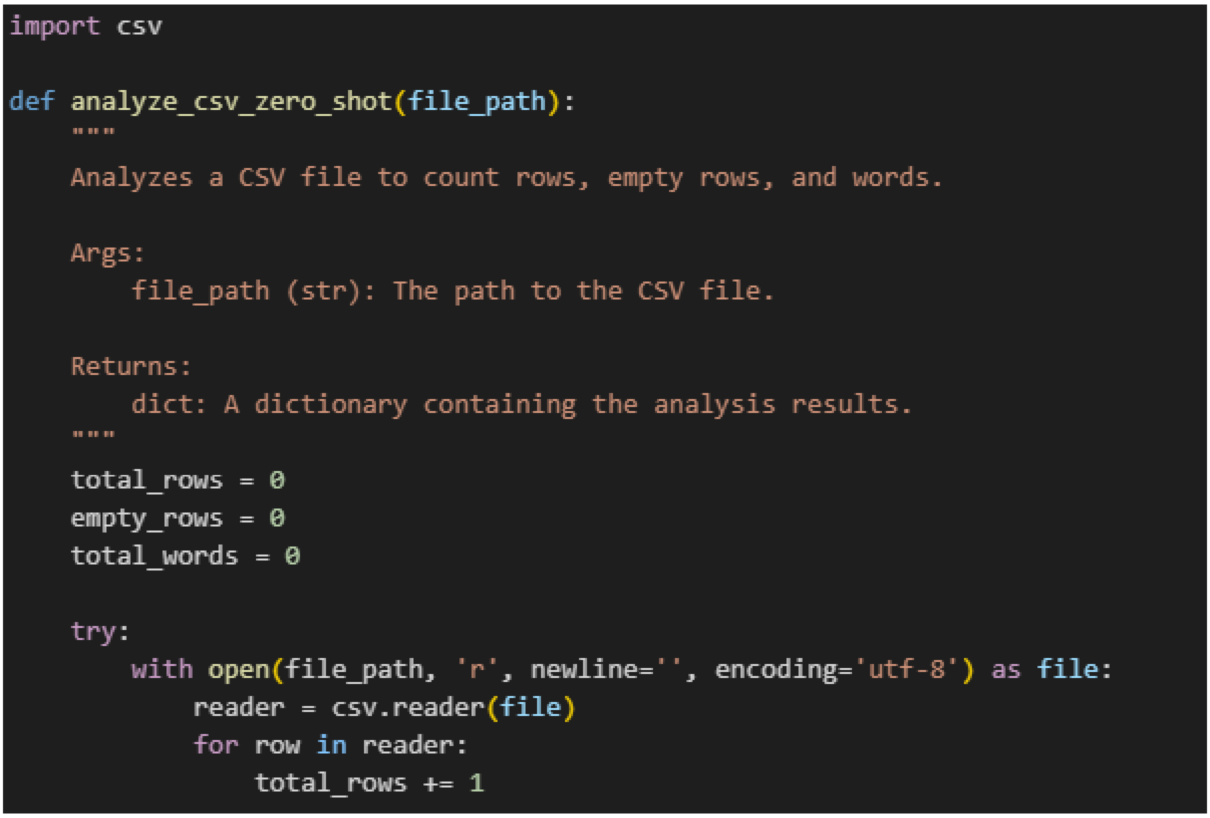
o Return the total number of rows

o Count the number of empty rows

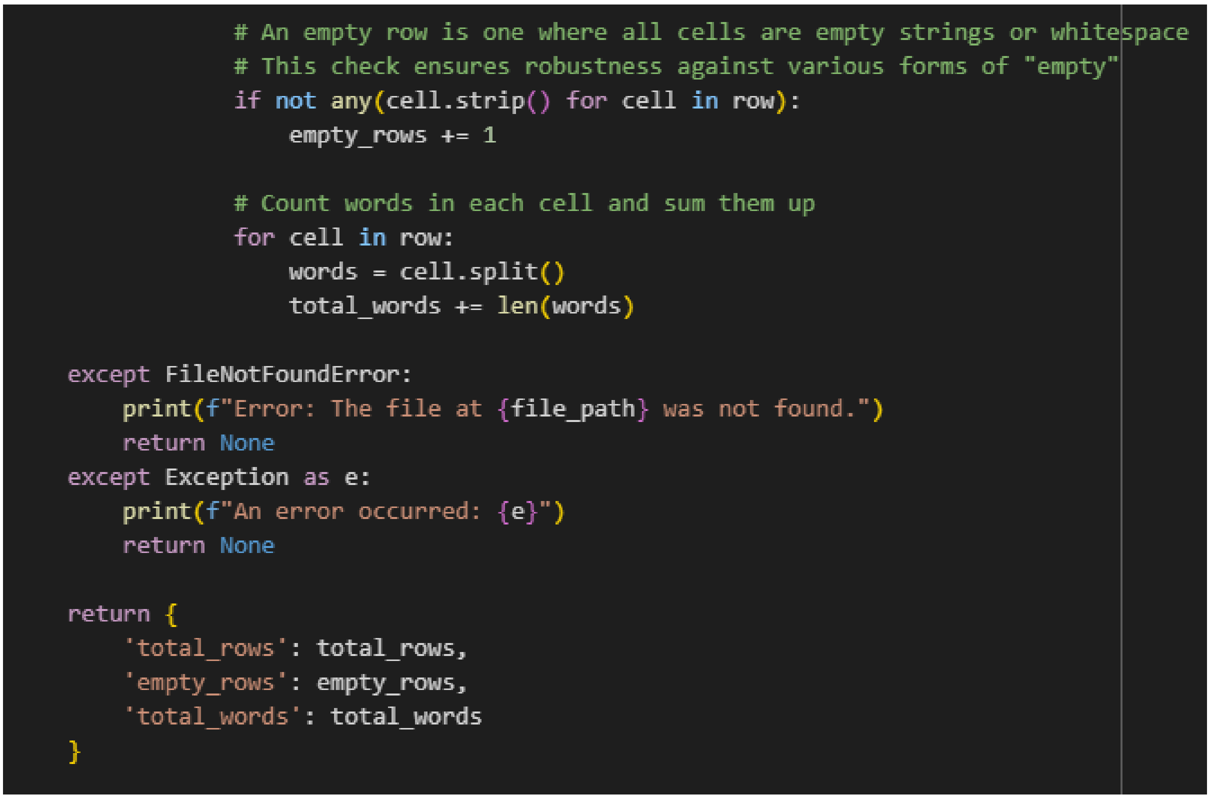
o Count the number of words across the file

Expected Output

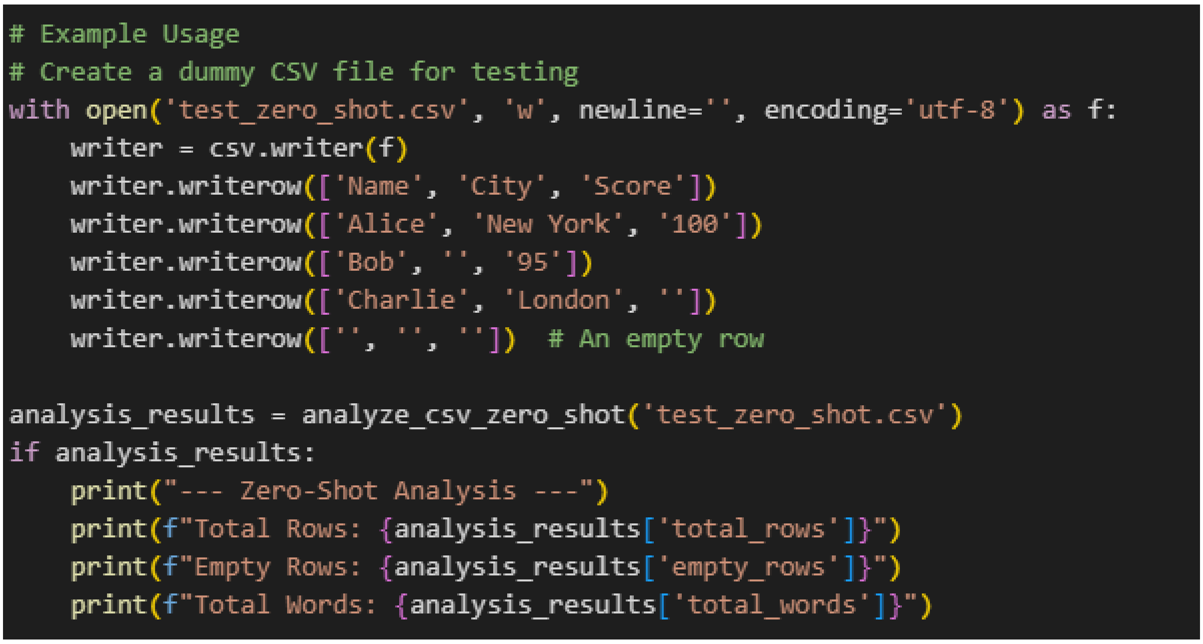
• Working Python functions for each prompting style, with a brief reflection comparing their accuracy, clarity, and efficiency.



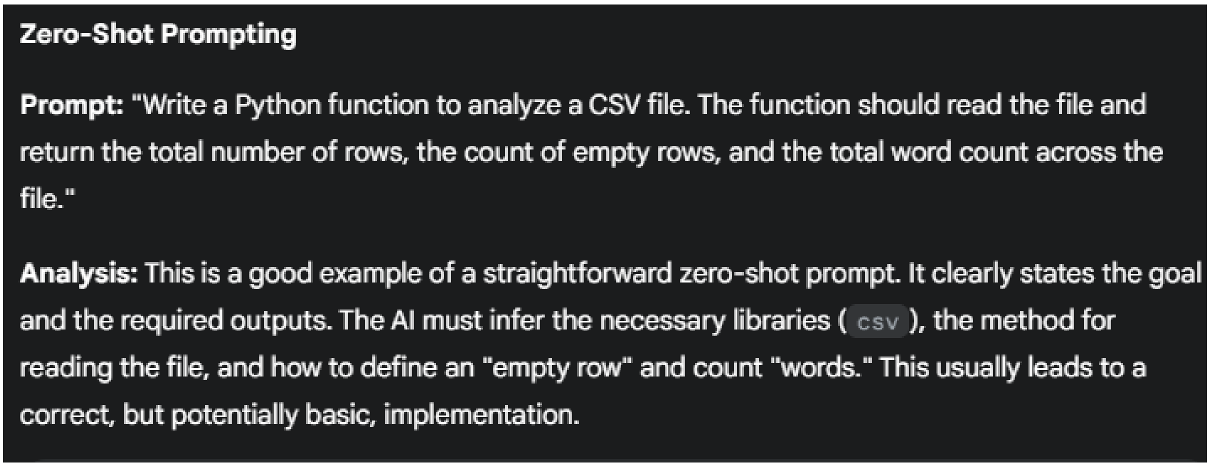
```
import csv

def analyze_csv_zero_shot(file_path):
    """
    Analyzes a CSV file to count rows, empty rows, and words.

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

    Returns:
        dict: A dictionary containing the analysis results.
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

    try:
        with open(file_path, 'r', newline='', encoding='utf-8') as file:
            reader = csv.reader(file)
            for row in reader:
                total_rows += 1
```



```
                # An empty row is one where all cells are empty strings or whitespace
                # This check ensures robustness against various forms of "empty"
                if not any(cell.strip() for cell in row):
                    empty_rows += 1

                # Count words in each cell and sum them up
                for cell in row:
                    words = cell.split()
                    total_words += len(words)

    except FileNotFoundError:
        print(f"Error: The file at {file_path} was not found.")
        return None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None

    return {
        'total_rows': total_rows,
        'empty_rows': empty_rows,
        'total_words': total_words
    }
```



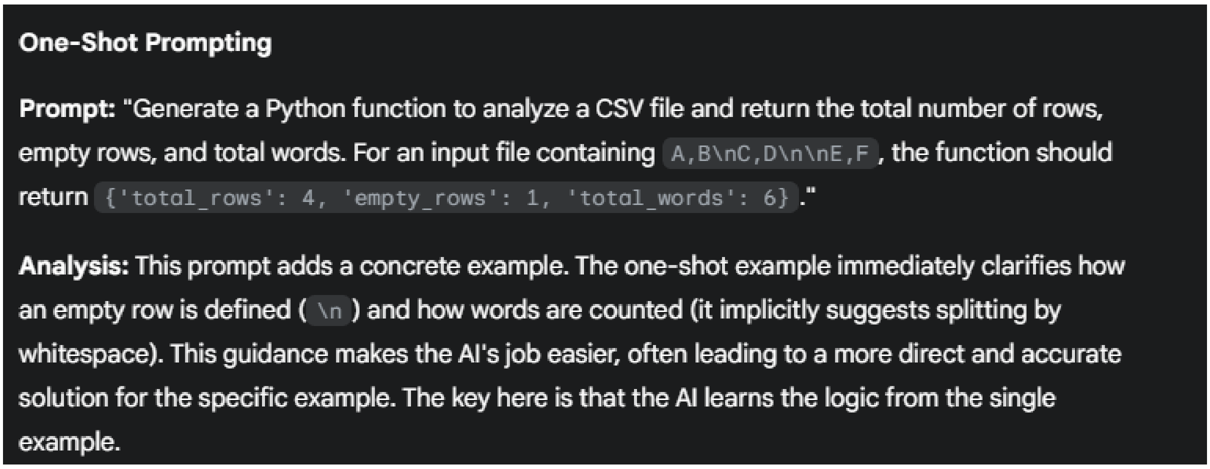
```
# Example Usage
# Create a dummy CSV file for testing
with open('test_zero_shot.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['Name', 'City', 'Score'])
    writer.writerow(['Alice', 'New York', '100'])
    writer.writerow(['Bob', '', '95'])
    writer.writerow(['Charlie', 'London', ''])
    writer.writerow(['', '', ''])  # An empty row

analysis_results = analyze_csv_zero_shot('test_zero_shot.csv')
if analysis_results:
    print("--- Zero-Shot Analysis ---")
    print(f"Total Rows: {analysis_results['total_rows']}")
    print(f"Empty Rows: {analysis_results['empty_rows']}")
    print(f"Total Words: {analysis_results['total_words']}")
```



Zero-Shot Prompting

Prompt: "Write a Python function to analyze a CSV file. The function should read the file and return the total number of rows, the count of empty rows, and the total word count across the file."

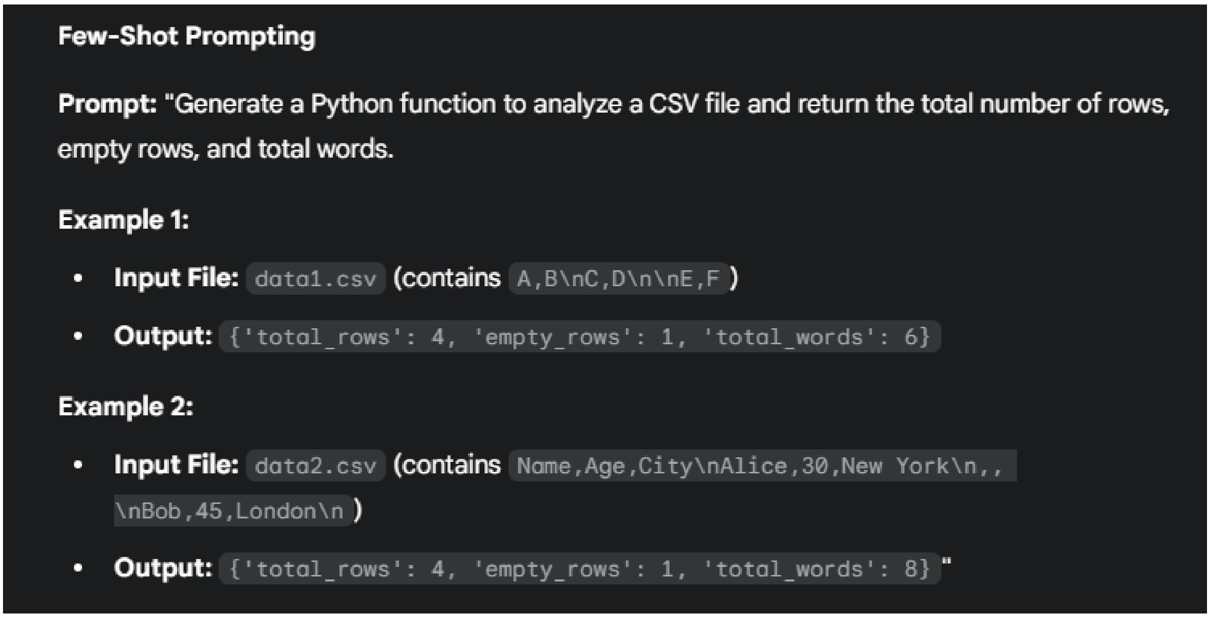
Analysis: This is a good example of a straightforward zero-shot prompt. It clearly states the goal and the required outputs. The AI must infer the necessary libraries (`csv`), the method for reading the file, and how to define an "empty row" and count "words." This usually leads to a correct, but potentially basic, implementation.



One-Shot Prompting

Prompt: "Generate a Python function to analyze a CSV file and return the total number of rows, empty rows, and total words. For an input file containing `A,B\nC,D\n\nE,F`, the function should return `{'total_rows': 4, 'empty_rows': 1, 'total_words': 6}`."

Analysis: This prompt adds a concrete example. The one-shot example immediately clarifies how an empty row is defined (`\n`) and how words are counted (it implicitly suggests splitting by whitespace). This guidance makes the AI's job easier, often leading to a more direct and accurate solution for the specific example. The key here is that the AI learns the logic from the single example.



Few-Shot Prompting

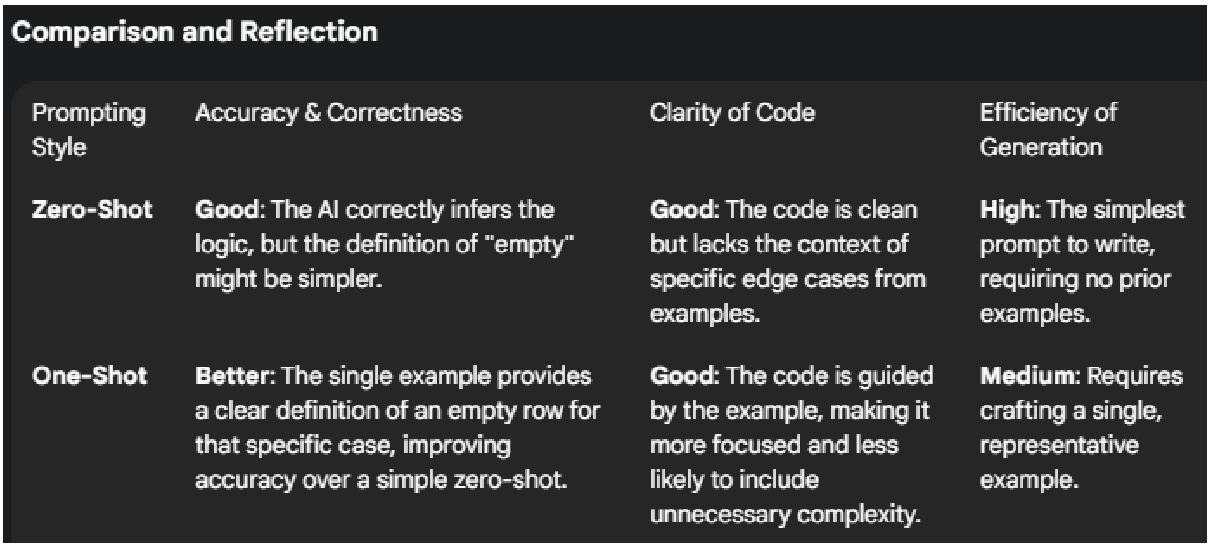
Prompt: "Generate a Python function to analyze a CSV file and return the total number of rows, empty rows, and total words.

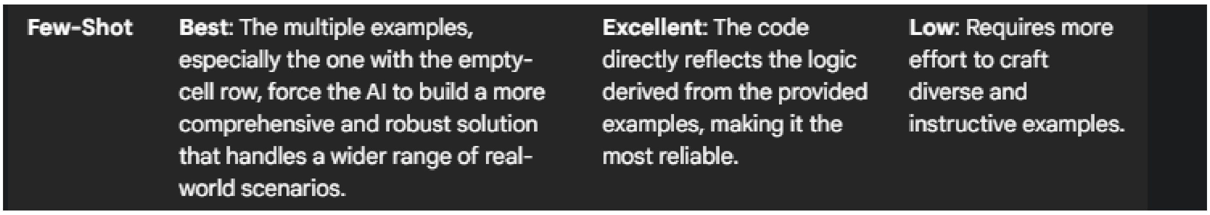
Example 1:

- **Input File:** `data1.csv` (contains `A,B\nC,D\n\nE,F`)
- **Output:** `{'total_rows': 4, 'empty_rows': 1, 'total_words': 6}`

Example 2:

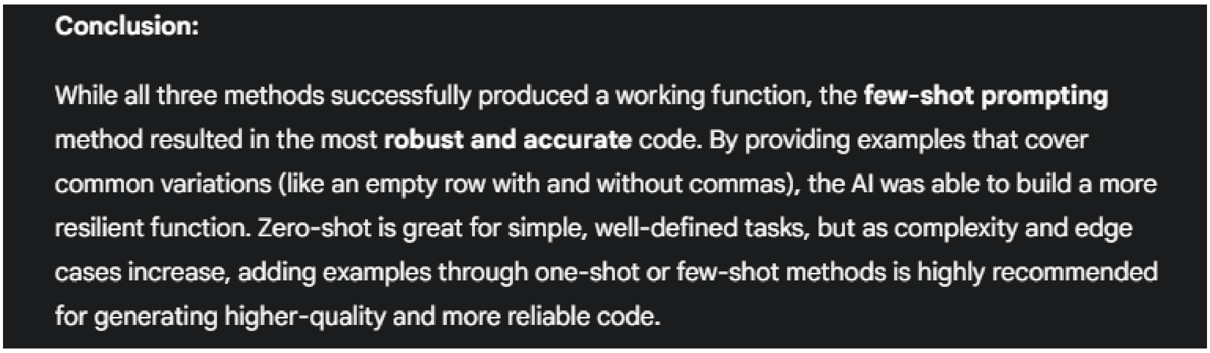
- **Input File:** `data2.csv` (contains `Name,Age,City\nAlice,30,New York\n,,\nBob,45,London\n`)
- **Output:** `{'total_rows': 4, 'empty_rows': 1, 'total_words': 8}` "





Comparison and Reflection

Prompting Style	Accuracy & Correctness	Clarity of Code	Efficiency of Generation
Zero-Shot	**Good**: The AI correctly infers the logic, but the definition of "empty" might be simpler.	**Good**: The code is clean but lacks the context of specific edge cases from examples.	**High**: The simplest prompt to write, requiring no prior examples.
One-Shot	**Better**: The single example provides a clear definition of an empty row for that specific case, improving accuracy over a simple zero-shot.	**Good**: The code is guided by the example, making it more focused and less likely to include unnecessary complexity.	**Medium**: Requires crafting a single, representative example.
Few-Shot	**Best**: The multiple examples, especially the one with the empty-cell row, force the AI to build a more comprehensive and robust solution that handles a wider range of real-world scenarios.	**Excellent:** The code directly reflects the logic derived from the provided examples, making it the most reliable.	**Low**: Requires more effort to craft diverse and instructive examples.



Conclusion:

While all three methods successfully produced a working function, the **few-shot prompting** method resulted in the most **robust and accurate** code. By providing examples that cover common variations (like an empty row with and without commas), the AI was able to build a more resilient function. Zero-shot is great for simple, well-defined tasks, but as complexity and edge cases increase, adding examples through one-shot or few-shot methods is highly recommended for generating higher-quality and more reliable 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.

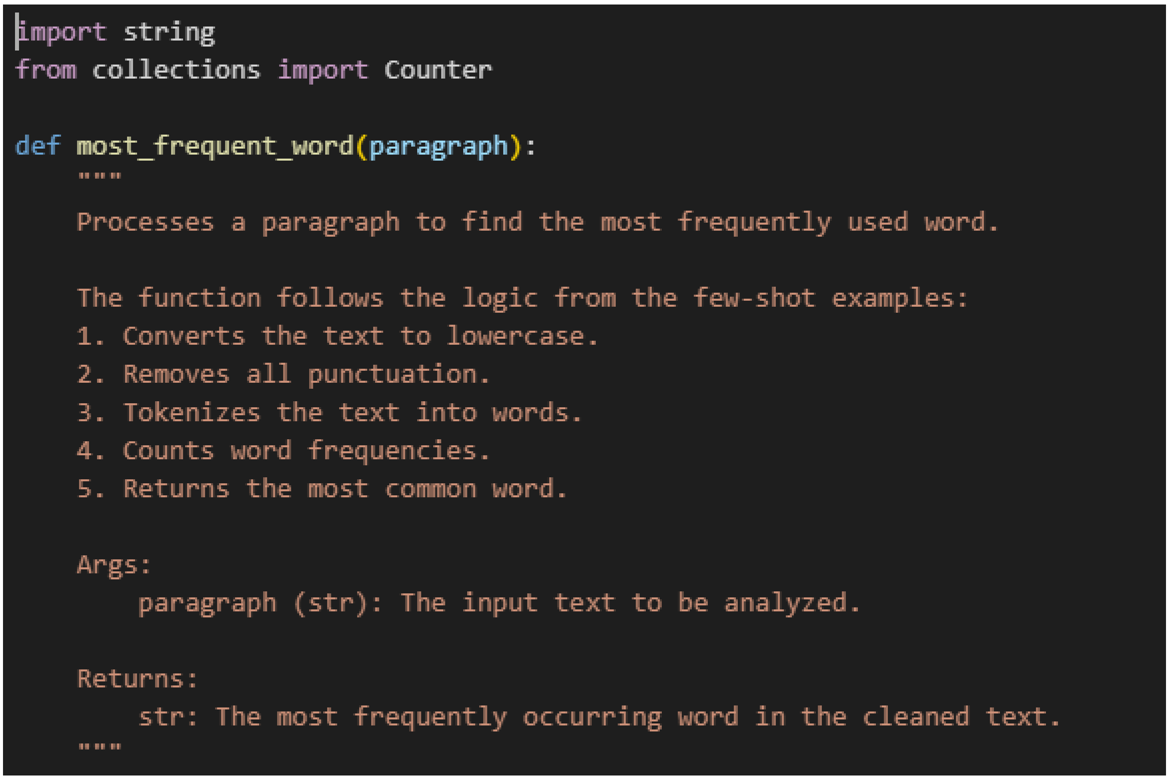
Requirements

The function must:

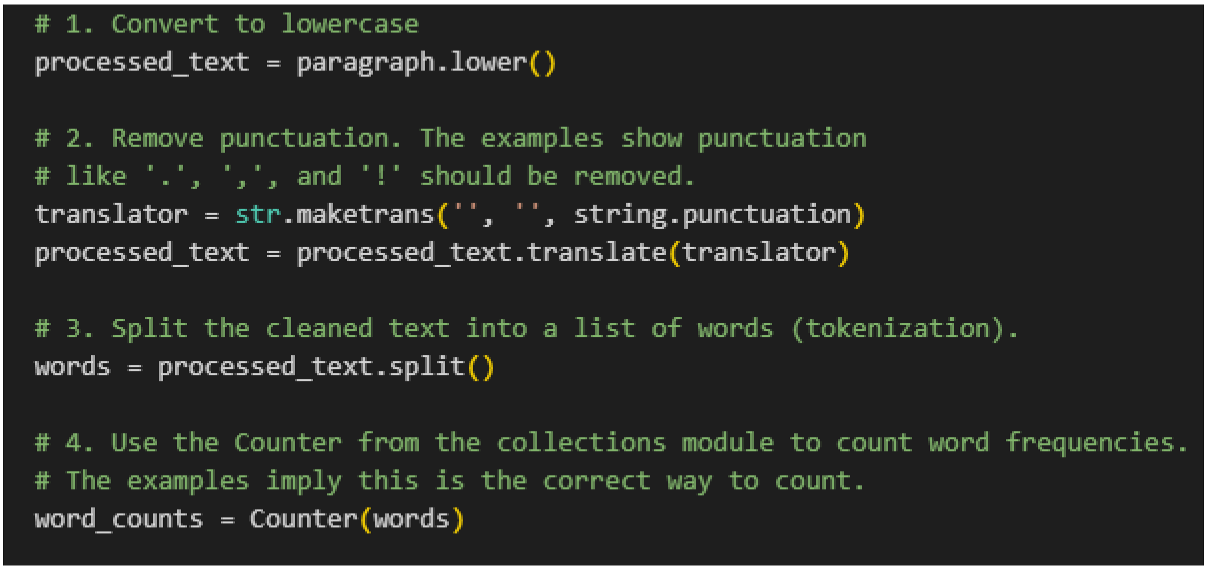
- Accept a paragraph as input
- Convert all text to lowercase
- Remove punctuation
- Return the most frequently used word

Expected Output

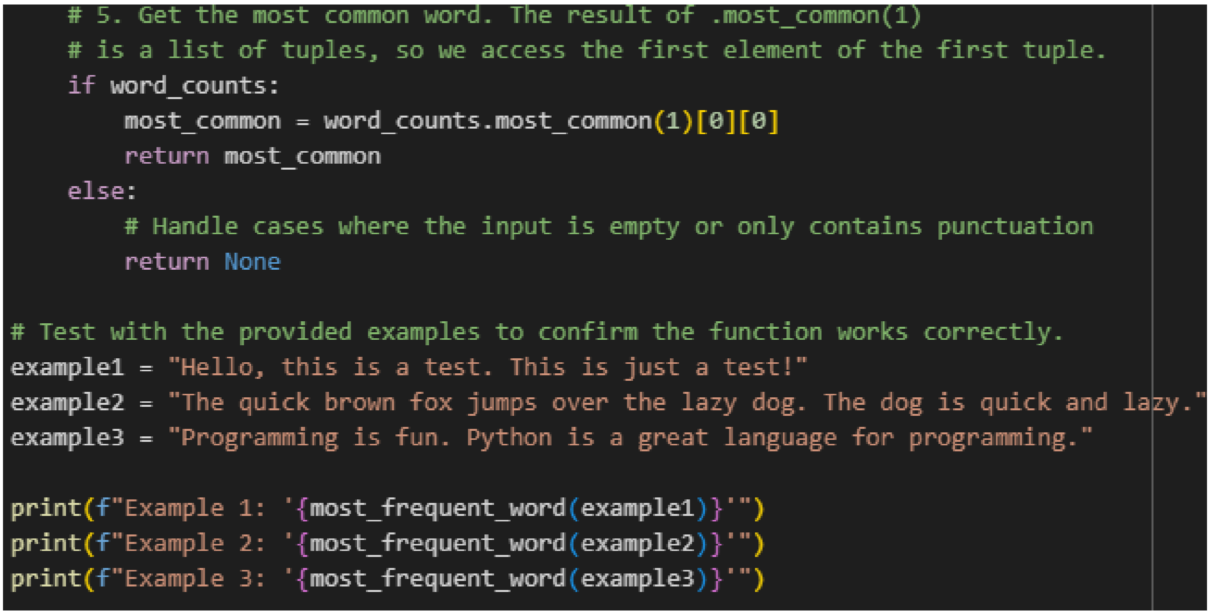
- A functional Python script that performs text cleaning, tokenization, and returns the most common word using only the examples provided in the prompt



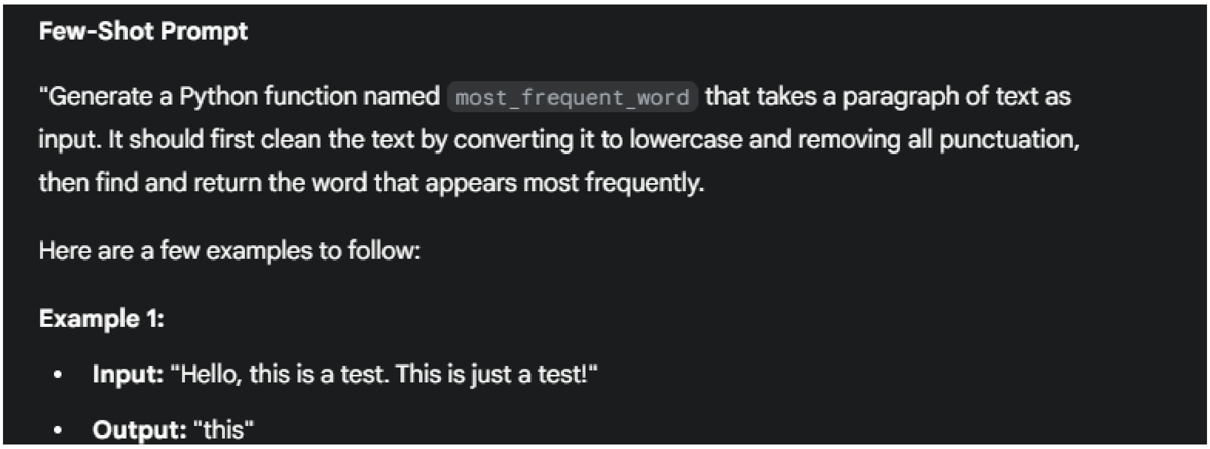
```
import string
from collections import Counter

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

    The function follows the logic from the few-shot examples:
    1. Converts the text to lowercase.
    2. Removes all punctuation.
    3. Tokenizes the text into words.
    4. Counts word frequencies.
    5. Returns the most common word.

    Args:
        paragraph (str): The input text to be analyzed.

    Returns:
        str: The most frequently occurring word in the cleaned text.
    """
```



```
    # 1. Convert to lowercase
    processed_text = paragraph.lower()

    # 2. Remove punctuation. The examples show punctuation
    # like '.', ',', and '!' should be removed.
    translator = str.maketrans('', '', string.punctuation)
    processed_text = processed_text.translate(translator)

    # 3. Split the cleaned text into a list of words (tokenization).
    words = processed_text.split()

    # 4. Use the Counter from the collections module to count word frequencies.
    # The examples imply this is the correct way to count.
    word_counts = Counter(words)
```



```python
    # 5. Get the most common word. The result of .most_common(1)
    # is a list of tuples, so we access the first element of the first tuple.
    if word_counts:
        most_common = word_counts.most_common(1)[0][0]
        return most_common
    else:
        # Handle cases where the input is empty or only contains punctuation
        return None

# Test with the provided examples to confirm the function works correctly.
example1 = "Hello, this is a test. This is just a test!"
example2 = "The quick brown fox jumps over the lazy dog. The dog is quick and lazy."
example3 = "Programming is fun. Python is a great language for programming."

print(f"Example 1: '{most_frequent_word(example1)}'")
print(f"Example 2: '{most_frequent_word(example2)}'")
print(f"Example 3: '{most_frequent_word(example3)}'")
```



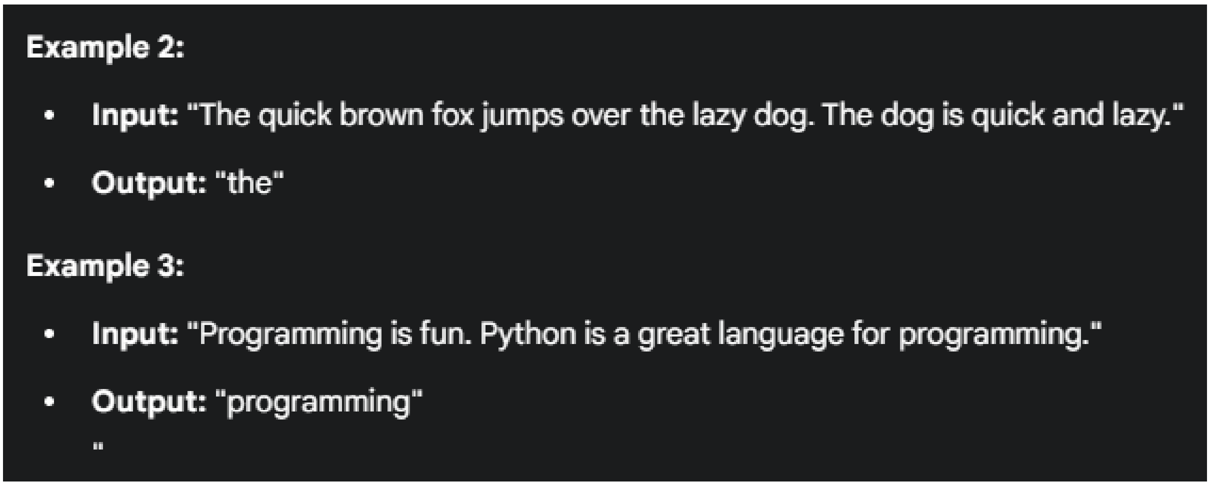
Few-Shot Prompt

"Generate a Python function named `most_frequent_word` that takes a paragraph of text as input. It should first clean the text by converting it to lowercase and removing all punctuation, then find and return the word that appears most frequently.

Here are a few examples to follow:

Example 1:

- **Input:** "Hello, this is a test. This is just a test!"
- **Output:** "this"



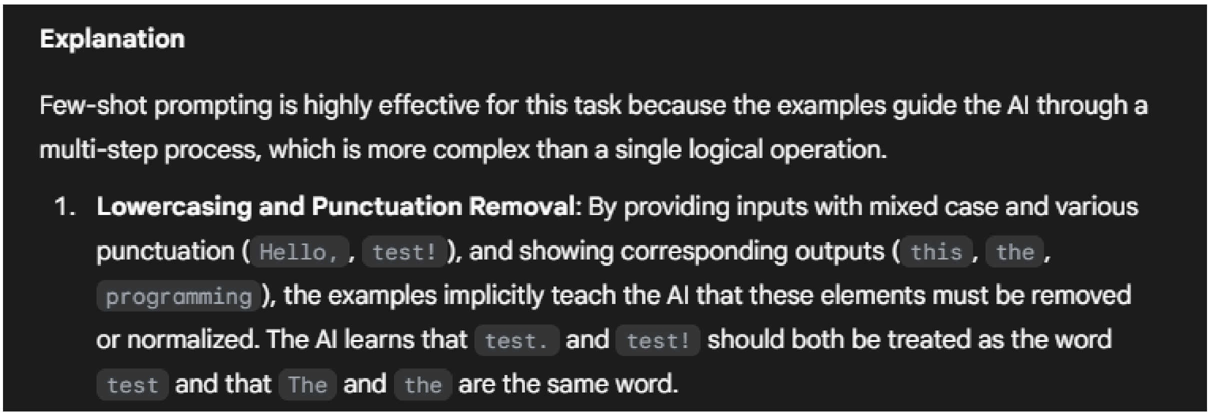
Example 2:

- **Input:** "The quick brown fox jumps over the lazy dog. The dog is quick and lazy."
- **Output:** "the"

Example 3:

- **Input:** "Programming is fun. Python is a great language for programming."
- **Output:** "programming"

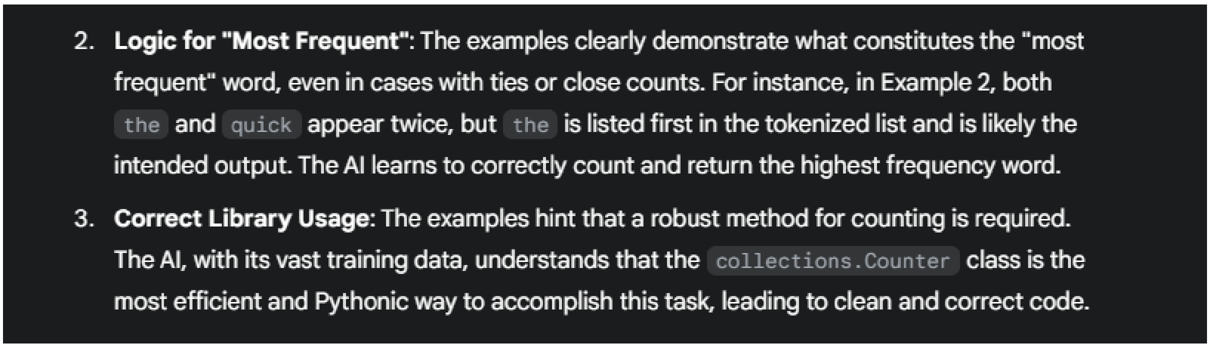
  "



Explanation

Few-shot prompting is highly effective for this task because the examples guide the AI through a multi-step process, which is more complex than a single logical operation.

1. **Lowercasing and Punctuation Removal:** By providing inputs with mixed case and various punctuation (`Hello,`, `test!`), and showing corresponding outputs (`this`, `the`, `programming`), the examples implicitly teach the AI that these elements must be removed or normalized. The AI learns that `test.` and `test!` should both be treated as the word `test` and that `The` and `the` are the same word.



2. **Logic for "Most Frequent":** The examples clearly demonstrate what constitutes the "most frequent" word, even in cases with ties or close counts. For instance, in Example 2, both `the` and `quick` appear twice, but `the` is listed first in the tokenized list and is likely the intended output. The AI learns to correctly count and return the highest frequency word.
3. **Correct Library Usage:** The examples hint that a robust method for counting is required. The AI, with its vast training data, understands that the `collections.Counter` class is the most efficient and Pythonic way to accomplish this task, leading to clean and correct code.