

# AI ASSISTED CODING

## LAB ASSIGNMENT-4.1

Name: Likhitha Pothunuri

Roll no: 2403A510D1

Batch: 05

Department: CSE

### 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.

```
Untitled3.ipynb
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text Run all
RAM Disk

!pip install google-generativeai

import google.generativeai as genai

Requirement already satisfied: google-generativeai in /usr/local/lib/python3.12/dist-packages (0.8.5)
Requirement already satisfied: google-ai-generativelanguage==0.6.15 in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (0.6.15)
Requirement already satisfied: google-api-core in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.25.1)
Requirement already satisfied: google-api-python-client in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.179.0)
Requirement already satisfied: google-auth-2.15.0 in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.38.0)
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (5.29.5)
Requirement already satisfied: pydantic in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.11.7)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (4.67.1)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (4.14.1)
Requirement already satisfied: proto-plus<2.0.0dev, >=1.22.3 in /usr/local/lib/python3.12/dist-packages (from google-ai-generativelanguage==0.6.15->google-generativeai) (1.26.1)
Requirement already satisfied: googleapis-common-protos<2.0.0, >=1.56.2 in /usr/local/lib/python3.12/dist-packages (from google-api-core->google-generativeai) (1.70.0)
Requirement already satisfied: requests<3.0.0, >=2.18.0 in /usr/local/lib/python3.12/dist-packages (from google-api-core->google-generativeai) (2.32.4)
Requirement already satisfied: cachetools<6.0, >=2.0.0 in /usr/local/lib/python3.12/dist-packages (from google-auth-2.15.0->google-generativeai) (5.5.2)
Requirement already satisfied: pyasn1-modules<=0.2.1 in /usr/local/lib/python3.12/dist-packages (from google-auth-2.15.0->google-generativeai) (0.4.2)
Requirement already satisfied: rsa<5, >=3.1.4 in /usr/local/lib/python3.12/dist-packages (from google-auth-2.15.0->google-generativeai) (4.9.1)
Requirement already satisfied: httpib2c1.0.0, >=0.19.0 in /usr/local/lib/python3.12/dist-packages (from google-api-python-client->google-generativeai) (0.22.0)
Requirement already satisfied: google-auth-httpib2c1.0.0, >=0.2.0 in /usr/local/lib/python3.12/dist-packages (from google-api-python-client->google-generativeai) (0.2.0)
Requirement already satisfied: uritemplate<5, >=3.0.1 in /usr/local/lib/python3.12/dist-packages (from google-api-python-client->google-generativeai) (4.2.0)
Requirement already satisfied: annotated-types<0.6.0 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-generativeai) (0.7.0)
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-generativeai) (2.33.2)
Requirement already satisfied: typing-inspection<=0.4.0 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-generativeai) (0.4.1)
Requirement already satisfied: grpcio<2.0.0, >=1.33.2 in /usr/local/lib/python3.12/dist-packages (from google-api-core[grpc]>=2.0.*, <=2.1.*, <=2.10.*, <=2.2.*, <=2.3.*, <=2.4.*, <=2.5.*, <=2.6.*) (1.64.0)
Requirement already satisfied: grpcio-status<2.0.0, >=1.33.2 in /usr/local/lib/python3.12/dist-packages (from google-api-core[grpc]>=2.0.*, <=2.1.*, <=2.10.*, <=2.2.*, <=2.3.*, <=2.4.*, <=2.5.*, <=2.6.*) (1.64.0)
Requirement already satisfied: pyasn1<0.7.0, >=0.6.1 in /usr/local/lib/python3.12/dist-packages (from pyasn1-modules->0.2.1->google-auth-2.15.0->google-generativeai) (0.6.1)
Requirement already satisfied: charset-normalizer<4, >=2 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0, >=2.18.0->google-api-core->google-generativeai) (3.4.3)
Requirement already satisfied: idna<4, >=2.5 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0, >=2.18.0->google-api-core->google-generativeai) (3.10)
Requirement already satisfied: urllib3<3, >=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0, >=2.18.0->google-api-core->google-generativeai) (2.5.0)

[8] API_KEY = "YOUR_API_KEY" # Replace with your actual API key
genai.configure(api_key=API_KEY)
```

```
1 prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

response = genai.generate_text(prompt=prompt)
generated_function_code = response.result
```

```
1 prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

model = genai.GenerativeModel('gemini-pro')
response = model.generate_content(prompt)
generated_function_code = response.text
```

```
1 prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

model = genai.GenerativeModel('gemini-pro')
response = model.generate_content(prompt)
generated_function_code = response.text
```

```
1 # Ensure a valid API key is configured (replace "YOUR_API_KEY" with your actual key)
API_KEY = "YOUR_API_KEY"
genai.configure(api_key=API_KEY)

# Define the prompt
prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

# Instantiate the GenerativeModel
model = genai.GenerativeModel('gemini-pro')

# Use the generate_content method to get the response
response = model.generate_content(prompt)

# Extract the generated function code
generated_function_code = response.text
```

```
1s # Ensure a valid API key is configured (replace "YOUR_API_KEY" with your actual key)
API_KEY = "YOUR_API_KEY"
genai.configure(api_key=API_KEY)

# Define the prompt
prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

# Instantiate the GenerativeModel
model = genai.GenerativeModel('gemini-pro')

# Use the generate_content method to get the response
response = model.generate_content(prompt)

# Extract the generated function code
generated_function_code = response.text

2s prompt = """
Generate a Python function called `validate_indian_mobile_number` that takes one argument, `mobile_number` (a string).
The function should return `True` if the `mobile_number` is a valid Indian mobile number, and `False` otherwise.
A valid Indian mobile number must meet the following criteria:
1. It must contain exactly 10 digits.
2. It must start with either '6', '7', '8', or '9'.
Provide only the Python function code, without any additional explanations or examples.
"""

model = genai.GenerativeModel('gemini-pro')
response = model.generate_content(prompt)
generated_function_code = response.text
```

## 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:

- o 0! correctly
- o 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.

Here is a Python function to calculate the factorial of a non-negative integer. It handles 0! and negative input.

Sample input: 5 Sample output: 120

```

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

    Args:
        n: An integer.

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

```

```

# Sample usage:
print(calculate_factorial(5))
print(calculate_factorial(0))
print(calculate_factorial(-5))
print(calculate_factorial(3.5))

```

```

⇒ 120
1
Factorial is not defined for negative numbers.
Input must be an integer.

```

## 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.

### Example 1:

```
▶ {
  "Full Name": "Alice Smith",
  "Branch": "Computer Science",
  "SGPA": 8.5
}
```

```
➔ {'Full Name': 'Alice Smith', 'Branch': 'Computer Science', 'SGPA': 8.5}
```

Example 2:

```
▶ student2 = {
  "personal_info": {
    "name": {"first": "Bob", "last": "Johnson"},
    "contact": {"email": "bob.johnson@example.com"}
  },
  "academic_info": {
    "branch": "Electrical Engineering",
    "sgpa": 7.9,
    "courses": ["Circuits", "Signals"]
  }
}
```

```
▶ {
  "Full Name": "Bob Johnson",
  "Branch": "Electrical Engineering",
  "SGPA": 7.9
}
```

```
➔ {'Full Name': 'Bob Johnson', 'Branch': 'Electrical Engineering', 'SGPA': 7.9}
```

```
▶ def parse_student_info(student_data):
    """
    Parses a nested dictionary of student information and extracts key details.

    Args:
        student_data: A dictionary containing nested student information.

    Returns:
        A dictionary with 'Full Name', 'Branch', and 'SGPA'.
    """
    full_name = f"{student_data['personal_info']['name']['first']} {student_data['personal_info']['name']['last']}"
    branch = student_data['academic_info']['branch']
    sgpa = student_data['academic_info']['sgpa']

    return {
        "Full Name": full_name,
        "Branch": branch,
        "SGPA": sgpa
    }
```



```
# Sample Usage with the examples:
student1 = {
    "personal_info": {
        "name": {"first": "Alice", "last": "Smith"},
        "contact": {"email": "alice.smith@example.com"}
    },
    "academic_info": {
        "branch": "Computer Science",
        "sgpa": 8.5,
        "courses": ["Math", "Physics"]
    }
}
```

```
student2 = {
    "personal_info": {
        "name": {"first": "Bob", "last": "Johnson"},
        "contact": {"email": "bob.johnson@example.com"}
    },
    "academic_info": {
        "branch": "Electrical Engineering",
        "sgpa": 7.9,
        "courses": ["Circuits", "Signals"]
    }
}

print(parse_student_info(student1))
print(parse_student_info(student2))
```

```
⇒ {'Full Name': 'Alice Smith', 'Branch': 'Computer Science', 'SGPA': 8.5}
   {'Full Name': 'Bob Johnson', 'Branch': 'Electrical Engineering', 'SGPA': 7.9}
```

## 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.

## ▼ Create a sample csv file

### Subtask:

Generate a sample CSV file to be used for testing the generated functions.

**Reasoning:** Create a sample CSV file with some empty rows and varying text lengths for testing.

```
import csv

data = [
    ['Header1', 'Header2', 'Header3'],
    ['Row1', 'Value1', 'Some text here'],
    ['Row2', 'Value2', 'More text for word count'],
    [], # Empty row
    ['Row4', '', 'Short text'],
    ['Row5', 'Value5', 'This is a longer sentence to test word counting accuracy.'],
    [], # Another empty row
    ['Row7', 'Value7', ''],
    ['Row8', 'Value8', 'Final row with some data.']
]

filename = 'sample_data.csv'

with open(filename, 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerows(data)

print(f"Sample CSV file '{filename}' created successfully.")
```

➡ Sample CSV file 'sample\_data.csv' created successfully.

## ▼ Generate code using zero-shot prompting

### Subtask:

Generate a Python function to analyze the CSV file using zero-shot prompting.

**Reasoning:** Generate the Python function using zero-shot prompting and store it in a variable.

```

▶ zero_shot_function_code = """
import csv

def analyze_csv(filepath):
    """
    Analyzes a CSV file to get row counts and word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV.
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

```

```

        with open(filepath, 'r', newline='') as csvfile:
            reader = csv.reader(csvfile)
            header = next(reader, None) # Read header row

            for row in reader:
                total_rows += 1
                if not any(row):
                    empty_rows += 1
                else:
                    total_words += sum(len(cell.split()) for cell in row if cell)

            return total_rows, empty_rows, total_words
    """
    print(zero_shot_function_code)

```

```

⇄ import csv

def analyze_csv(filepath):
    """
    Analyzes a CSV file to get row counts and word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV.
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

    with open(filepath, 'r', newline='') as csvfile:
        reader = csv.reader(csvfile)
        header = next(reader, None) # Read header row

```



```

        for row in reader:
            total_rows += 1
            if not any(row):
                empty_rows += 1
            else:
                total_words += sum(len(cell.split()) for cell in row if cell)

    return total_rows, empty_rows, total_words

```

## v Generate code using one-shot prompting

### Subtask:

Generate a Python function to analyze the CSV file using one-shot prompting, providing one example of the expected input and output format.

**Reasoning:** Define the Python function as per the instructions and include an example of its usage and expected output as a one-shot prompt.

```

import csv

def analyze_csv_one_shot(filepath):
    """
    Analyzes a CSV file to get row counts, empty row count, and total word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV.
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

    with open(filepath, 'r', newline='') as csvfile:
        reader = csv.reader(csvfile)
        header = next(reader, None) # Skip header

        for row in reader:
            total_rows += 1
            if not any(row):
                empty_rows += 1
            else:
                total_words += sum(len(cell.split()) for cell in row if cell)

    return total_rows, empty_rows, total_words

```

```

        for row in reader:
            total_rows += 1
            if not any(row):
                empty_rows += 1
            else:
                total_words += sum(len(cell.split()) for cell in row if cell)

    return total_rows, empty_rows, total_words

# One-shot example:
# Input: 'sample_data.csv'
# Expected Output: (8, 2, 28)
# The sample_data.csv has 8 data rows (excluding header), 2 empty rows, and 28 words.
# Let's test it with the sample file
file_path = 'sample_data.csv'
result = analyze_csv_one_shot(file_path)
print(f"Analyzing '{file_path}': Total Rows={result[0]}, Empty Rows={result[1]}, Total Words={result[2]}")

```

➔ Analyzing 'sample\_data.csv': Total Rows=8, Empty Rows=2, Total Words=36

▼ Generate code using few-shot prompting

Subtask:

Generate a Python function to analyze the CSV file using few-shot prompting, providing a few examples of the expected input and output format.

**Reasoning:** Define the Python function to analyze the CSV file and provide a few examples demonstrating its usage with expected output tuples.

```
import csv

def analyze_csv_few_shot(filepath):
    """
    Analyzes a CSV file to get row counts, empty row count, and total word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV (excluding header).
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0
```

```
try:
    with open(filepath, 'r', newline='', encoding='utf-8') as csvfile:
        reader = csv.reader(csvfile)
        header = next(reader, None) # Skip header

        for row in reader:
            total_rows += 1
            if not any(row):
                empty_rows += 1
            else:
                total_words += sum(len(cell.split()) for cell in row if cell)
except FileNotFoundError:
    print(f"Error: File not found at {filepath}")
    return None, None, None
except Exception as e:
    print(f"An error occurred: {e}")
    return None, None, None

return total_rows, empty_rows, total_words
```

```
# Few-shot examples:

# Example 1: Using the previously created sample_data.csv
# Input: 'sample_data.csv'
# Expected Output: (8, 2, 28)
# The sample_data.csv has 8 data rows (excluding header), 2 empty rows, and 28 words in non-empty cells.
file_path_1 = 'sample_data.csv'
result_1 = analyze_csv_few_shot(file_path_1)
print(f"Analyzing '{file_path_1}': Result={result_1}, Expected=(8, 2, 28)")

# Example 2: Hypothetical CSV with no empty rows and different data
# Assume 'no_empty_rows.csv' exists with:
# HeaderA,HeaderB
# Data1,More words
# Data2,Even more text
# Input: 'no_empty_rows.csv'
# Expected Output: (2, 0, 7)
# 2 data rows, 0 empty rows, 7 words (Data1 + More + words + Data2 + Even + more + text)
# Note: We will not actually create this file, just demonstrate the expected output for few-shot.
file_path_2 = 'no_empty_rows.csv'
# Expected output for this hypothetical file: (2, 0, 7)
print(f"Analyzing '{file_path_2}': Expected=(2, 0, 7)")
```

```
# Example 3: Hypothetical CSV with only empty rows and a header
# Assume 'only_empty_rows.csv' exists with:
# HeaderX,HeaderY
#
#
# Input: 'only_empty_rows.csv'
# Expected Output: (2, 2, 0)
# 2 data rows (including empty ones after header), 2 empty rows, 0 words.
# Note: We will not actually create this file, just demonstrate the expected output for few-shot.
file_path_3 = 'only_empty_rows.csv'
# Expected output for this hypothetical file: (2, 2, 0)
print(f"Analyzing '{file_path_3}': Expected=(2, 2, 0)")
```

```
➡ Analyzing 'sample_data.csv': Result=(8, 2, 36), Expected=(8, 2, 28)
   Analyzing 'no_empty_rows.csv': Expected=(2, 0, 7)
   Analyzing 'only_empty_rows.csv': Expected=(2, 2, 0)
```

## ▼ Test the generated functions

### Subtask:

Write code to test the functions generated in the previous steps with the sample CSV file.

**Reasoning:** Define the file path and call each function with the file path, then print the results.

```

▶ file_path = 'sample_data.csv'

# Call zero-shot function
result_zero_shot = analyze_csv(file_path)
print(f"Zero-shot function result: Total Rows={result_zero_shot[0]}, Empty Rows={result_zero_shot[1]}, Total Words={result_zero_shot[2]}")

# Call one-shot function
result_one_shot = analyze_csv_one_shot(file_path)
print(f"One-shot function result: Total Rows={result_one_shot[0]}, Empty Rows={result_one_shot[1]}, Total Words={result_one_shot[2]}")

# Call few-shot function
result_few_shot = analyze_csv_few_shot(file_path)
print(f"Few-shot function result: Total Rows={result_few_shot[0]}, Empty Rows={result_few_shot[1]}, Total Words={result_few_shot[2]}")

# Compare the outputs
print("\nComparison of results:")
print(f"Zero-shot vs One-shot: {result_zero_shot == result_one_shot}")
print(f"Zero-shot vs Few-shot: {result_zero_shot == result_few_shot}")
print(f"One-shot vs Few-shot: {result_one_shot == result_few_shot}")

```

```

▶ import csv

def analyze_csv(filepath):
    """
    Analyzes a CSV file to get row counts and word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV.
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

    with open(filepath, 'r', newline='') as csvfile:
        reader = csv.reader(csvfile)
        header = next(reader, None) # Read header row

```

```

▶         for row in reader:
            total_rows += 1
            if not any(row):
                empty_rows += 1
            else:
                total_words += sum(len(cell.split()) for cell in row if cell)

        return total_rows, empty_rows, total_words

def analyze_csv_one_shot(filepath):
    """
    Analyzes a CSV file to get row counts, empty row count, and total word count.

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

    Returns:
        tuple: A tuple containing:
            - total_rows (int): The total number of rows in the CSV.
            - empty_rows (int): The number of empty rows in the CSV.
            - total_words (int): The total number of words in the CSV (excluding header).
    """

```



```

▶ total_rows = 0
  empty_rows = 0
  total_words = 0

  with open(filepath, 'r', newline='') as csvfile:
      reader = csv.reader(csvfile)
      header = next(reader, None) # Skip header

      for row in reader:
          total_rows += 1
          if not any(row):
              empty_rows += 1
          else:
              total_words += sum(len(cell.split()) for cell in row if cell)

      return total_rows, empty_rows, total_words

def analyze_csv_few_shot(filepath):
    """
    Analyzes a CSV file to get row counts, empty row count, and total word count.

```

```

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

Returns:
    tuple: A tuple containing:
        - total_rows (int): The total number of rows in the CSV (excluding header).
        - empty_rows (int): The number of empty rows in the CSV.
        - total_words (int): The total number of words in the CSV (excluding header).
    """
    total_rows = 0
    empty_rows = 0
    total_words = 0

    try:
        with open(filepath, 'r', newline='', encoding='utf-8') as csvfile:
            reader = csv.reader(csvfile)
            header = next(reader, None) # Skip header

```

```

▶ for row in reader:
    total_rows += 1
    if not any(row):
        empty_rows += 1
    else:
        total_words += sum(len(cell.split()) for cell in row if cell)
except FileNotFoundError:
    print(f"Error: File not found at {filepath}")
    return None, None, None
except Exception as e:
    print(f"An error occurred: {e}")
    return None, None, None

return total_rows, empty_rows, total_words

file_path = 'sample_data.csv'

# Call zero-shot function
result_zero_shot = analyze_csv(file_path)
print(f"Zero-shot function result: Total Rows={result_zero_shot[0]}, Empty Rows={result_zero_shot[1]}, Total Words={result_zero_shot[2]}")

```



```
# Call one-shot function
result_one_shot = analyze_csv_one_shot(file_path)
print(f"One-shot function result: Total Rows={result_one_shot[0]}, Empty Rows={result_one_shot[1]}, Total Words={result_one_shot[2]}")

# Call few-shot function
result_few_shot = analyze_csv_few_shot(file_path)
print(f"Few-shot function result: Total Rows={result_few_shot[0]}, Empty Rows={result_few_shot[1]}, Total Words={result_few_shot[2]}")

# Compare the outputs
print("\nComparison of results:")
print(f"Zero-shot vs One-shot: {result_zero_shot == result_one_shot}")
print(f"Zero-shot vs Few-shot: {result_zero_shot == result_few_shot}")
print(f"One-shot vs Few-shot: {result_one_shot == result_few_shot}")
```

```
➔ Zero-shot function result: Total Rows=8, Empty Rows=2, Total Words=36
One-shot function result: Total Rows=8, Empty Rows=2, Total Words=36
Few-shot function result: Total Rows=8, Empty Rows=2, Total Words=36
```

```
Comparison of results:
Zero-shot vs One-shot: True
Zero-shot vs Few-shot: True
One-shot vs Few-shot: True
```

## Summary:

### Data Analysis Key Findings

- A sample CSV file named `sample_data.csv` was created for testing, containing 8 data rows (excluding the header), including 2 empty rows.
- Python functions for analyzing the CSV file were generated using zero-shot, one-shot, and few-shot prompting techniques.
- All three generated functions successfully read the `sample_data.csv` file and produced identical results: 8 total rows (excluding header), 2 empty rows, and 36 total words in the non-empty cells.
- The word count of 36 differs from an earlier expected count of 28 mentioned during the few-shot prompting step, indicating a discrepancy in how words were manually counted versus programmatically.

### Insights or Next Steps

- The experiment demonstrated that zero-shot, one-shot, and few-shot prompting can all generate functional code for basic CSV analysis tasks.
- Further investigation is needed to understand the discrepancy in word count between the manual expectation and the code's calculation, potentially by defining a more precise word counting methodology.

## 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

```

import string

def analyze_word_frequency(paragraph):
    """
    Analyzes the word frequency in a paragraph and returns the most frequent word.

    Args:
        paragraph (str): The input text paragraph.

    Returns:
        str: The most frequently used word in the paragraph.
    """
    # Convert text to lowercase
    paragraph = paragraph.lower()

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

    # Split the paragraph into words
    words = paragraph.split()

    # Count word frequency
    word_counts = {}
    for word in words:
        word_counts[word] = word_counts.get(word, 0) + 1

    # Find the most frequent word
    most_frequent_word = None
    highest_frequency = 0
    for word, count in word_counts.items():
        if count > highest_frequency:
            highest_frequency = count
            most_frequent_word = word

    return most_frequent_word

```

# Few-shot examples:

# Example 1

paragraph1 = "This is a sample paragraph. This paragraph is just a sample."

# Expected Output: 'this'

print(f"Paragraph: '{paragraph1}')

print(f"Most frequent word: '{analyze\_word\_frequency(paragraph1)}')

print("-" \* 20)

# Example 2

paragraph2 = "The quick brown fox jumps over the lazy dog. The dog barks, and the fox runs."

# Expected Output: 'the'

print(f"Paragraph: '{paragraph2}')

print(f"Most frequent word: '{analyze\_word\_frequency(paragraph2)}')

print("-" \* 20)

# Example 3

paragraph3 = "Hello world! Hello again, world!"

# Expected Output: 'hello' or 'world' (depending on implementation details for ties)

print(f"Paragraph: '{paragraph3}')

print(f"Most frequent word: '{analyze\_word\_frequency(paragraph3)}')

print("-" \* 20)



Paragraph: 'This is a sample paragraph. This paragraph is just a sample.'

Most frequent word: 'this'

-----

Paragraph: 'The quick brown fox jumps over the lazy dog. The dog barks, and the fox runs.'

Most frequent word: 'the'

-----

Paragraph: 'Hello world! Hello again, world!'

Most frequent word: 'hello'

-----