SR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

Course Title – AI Assisted Coding

Course Code : 24CS002PC215

Assignment Type : Lab

Academic Year : 2025–2026

Name : Soumith Sripathi

Enroll Number : 2403A51340

Batch Number : 24BTCAICSB14

Week – 4

Date – 13 August 2025

Advanced Prompt Engineering – Zero–shot, One–shot, and Few–shot Techniques

Task Description –1 :

Question 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.

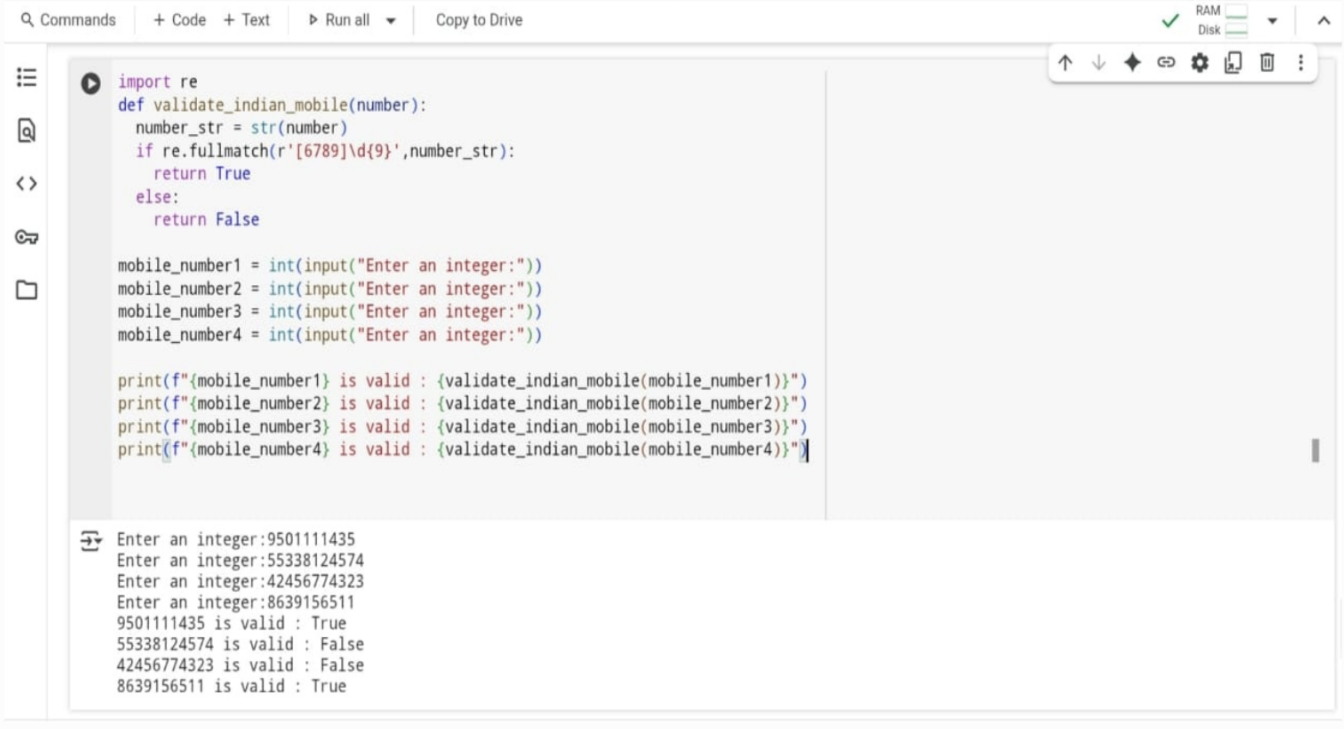
Requirements :

The function must ensure the mobile number:

Starts with 6, 7, 8, or 9

Contains exactly 10 digits

CODE & OUTPUT :



Explanation Of The Above Program :

~ The program runs an infinite loop with while True:.
~ It asks the user to enter an Indian mobile number or type 'quit'.
~ The input is stored in number_input.
~ If the user types 'quit' (case–insensitive), the loop ends with break.
~ Otherwise, the number is sent to validate_indian_mobile_number(number_input).
~ This function (defined elsewhere) checks if the input meets Indian mobile number rules.
~ Typical rules:

- Optional +91 or leading 0.
- Exactly 10 digits after removing prefix.
- First digit between 6 and 9.
- All characters are digits.

~ The function returns True if valid, otherwise False.
~ The result is printed in the format: '<number>' is valid: <True/False>.
~ Loop repeats until 'quit' is entered.

Task Description –2 :

Question 2 : Use one–shot prompting to generate a Python function that calculates the factorial of a number.

Requirements :

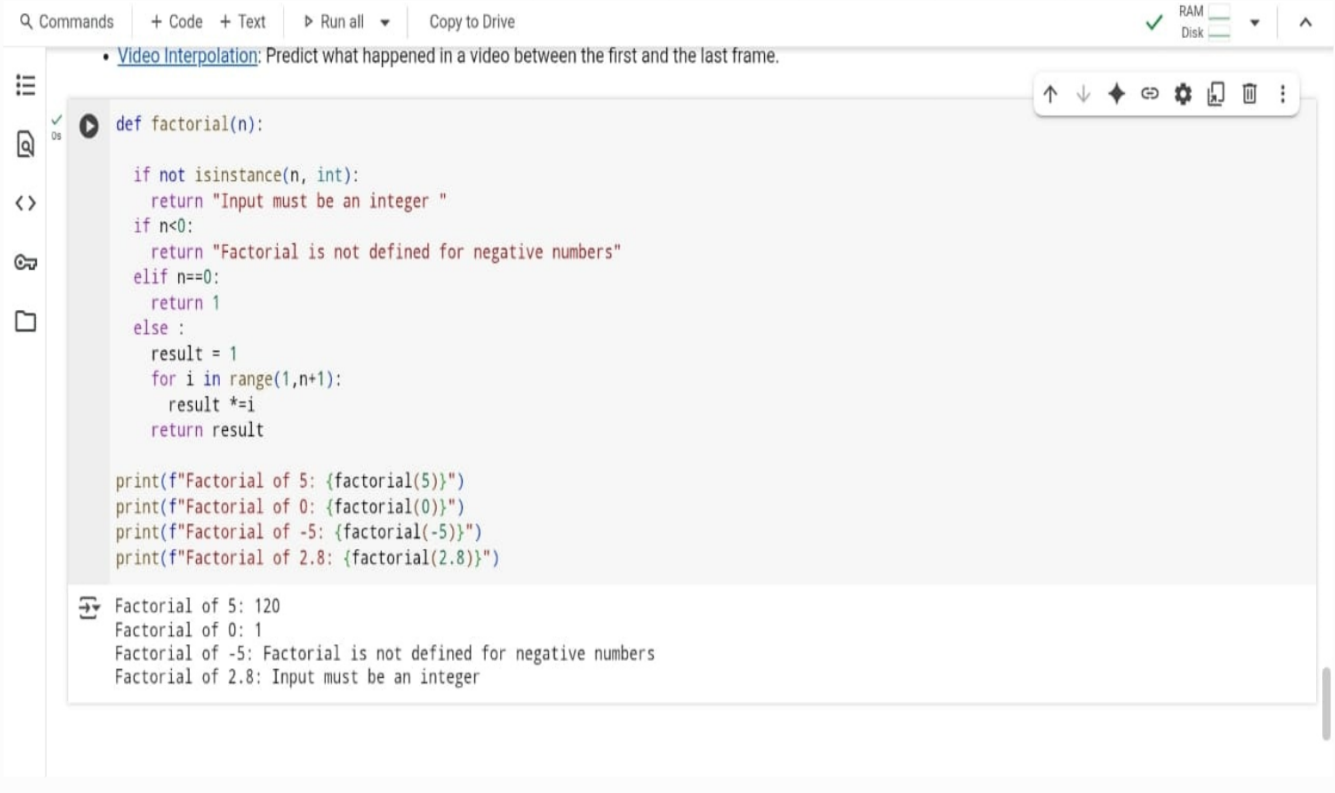
Provide one sample input–output pair in the prompt to guide the AI.

The function should handle:

~ 0! correctly

~ Negative input by returning an appropriate message

CODE & OUTPUT :



Explanation Of The Above Program :

~ calculate_factorial(n) takes an integer n and explains its purpose in a docstring.

~ If n < 0, it immediately returns "Factorial is not defined for negative numbers".

~ If n == 0, it returns
1 (by definition, 0! = 1).
~ Otherwise it sets an accumulator: factorial = 1.

~ It loops i from 1 to n inclusive:
for i in range(1, n+1).

~ Each iteration multiplies the accumulator:
factorial *= i.

~ After the loop, it returns the final factorial value.
~ calculate_factorial(5) → 1×2×3×4×5 = 120 (printed).
~ calculate_factorial(0) → 1;
calculate_factorial(−5) → the error message.
~ calculate_factorial(3) → 1×2×3 = 6, showing the loop works for small n.

Task Description –3 :

Question 3 : 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 & OUTPUT :

```
def parse_student_info(student_data):
  """
  Parses a nested dictionary containing student information.

  # Few-shot prompts:
  # Input: {'student': {'personal_info': {'name': 'Alice Smith'}, 'academic_info': {'branch': 'Computer Science', 'sgpa': 8.5}}}
  # Output: {'Full Name': 'Alice Smith', 'Branch': 'Computer Science', 'SGPA': 8.5}
  # Input: {'student': {'personal_info': {'name': 'Bob Johnson'}, 'academic_info': {'branch': 'Electrical Engineering', 'sgpa': 7.9}}}
  # Output: {'Full Name': 'Bob Johnson', 'Branch': 'Electrical Engineering', 'SGPA': 7.9}
  # Input: {'student': {'personal_info': {'name': 'Charlie Brown'}, 'academic_info': {'branch': 'Mechanical Engineering', 'sgpa': 9.1}}}
  # Output: {'Full Name': 'Charlie Brown', 'Branch': 'Mechanical Engineering', 'SGPA': 9.1}

  Args:
    student_data: A nested dictionary with student information.

  Returns:
    A dictionary containing the extracted Full Name, Branch, and SGPA.
  """
  full_name = student_data.get('student', {}).get('personal_info', {}).get('name')
  branch = student_data.get('student', {}).get('academic_info', {}).get('branch')
  sgpa = student_data.get('student', {}).get('academic_info', {}).get('sgpa')

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

# Example usage:
student1_data = {'student': {'personal_info': {'name': 'Alice Smith'}, 'academic_info': {'branch': 'Computer Science', 'sgpa': 8.5}}}
student2_data = {'student': {'personal_info': {'name': 'Bob Johnson'}, 'academic_info': {'branch': 'Electrical Engineering', 'sgpa': 7.9}}}
student3_data = {'student': {'personal_info': {'name': 'Charlie Brown'}, 'academic_info': {'branch': 'Mechanical Engineering', 'sgpa': 9.1}}}


print(parse_student_info(student1_data))
print(parse_student_info(student2_data))
print(parse_student_info(student3_data))
```

```
{'Full Name': 'Alice Smith', 'Branch': 'Computer Science', 'SGPA': 8.5}
{'Full Name': 'Bob Johnson', 'Branch': 'Electrical Engineering', 'SGPA': 7.9}
{'Full Name': 'Charlie Brown', 'Branch': 'Mechanical Engineering', 'SGPA': 9.1}
```

Explanation Of The Above Program :

~ The function parse_student_info(student_data) extracts key details from a nested student dictionary.
~ It expects the data under the top key 'student'.
~ From 'personal_info', it gets the student's 'name' and stores it in full_name.
~ From 'academic_info', it gets the 'branch'.
~ Also from 'academic_info', it gets the 'sgpa'.
~ The .get() method with {} as a default avoids key errors if data is missing.
~ It returns a new dictionary with 'Full Name', 'Branch', and 'SGPA'.
~ Example dictionaries (student1_data, student2_data, student3_data) are defined with nested personal and academic info.
~ The function is called for each student's data.
~ The printed output shows extracted information in a clean format.

Task Description –4 :

Question 4 : 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 :

```python
import csv

def analyze_csv_zeroshot(file_path):
    total_rows = 0
    empty_rows = 0
    word_count = 0

    with open(file_path, "r", newline='') as file:
        reader = csv.reader(file)
        for row in reader:
            total_rows += 1
            if not any(cell.strip() for cell in row):
                empty_rows += 1
            else:
                for cell in row:
                    word_count += len(cell.split())

    return total_rows, empty_rows, word_count
```

One–Shot :

```python
import csv

def analyze_csv_oneshot(file_path):
    total_rows = 0
    empty_rows = 0
    word_count = 0

    with open(file_path, "r", newline='') as file:
        reader = csv.reader(file)
        for row in reader:
            total_rows += 1
            if all(cell.strip() == "" for cell in row):
                empty_rows += 1
            else:
                word_count += sum(len(cell.split()) for cell in row)

    return total_rows, empty_rows, word_count
```

Few–Shot :

```python
import csv

def analyze_csv_fewshot(file_path):
    total_rows = 0
    empty_rows = 0
    word_count = 0

    with open(file_path, "r", newline='') as file:
        reader = csv.reader(file)
        for row in reader:
            total_rows += 1
            if all(cell.strip() == "" for cell in row):
                empty_rows += 1
            else:
                # Handle multiple spaces and quoted text correctly
                for cell in row:
                    cell = cell.strip()
                    if cell:
                        word_count += len(cell.split())

    return total_rows, empty_rows, word_count
```

OUTPUT :

makefile Copy Edit

```
Zero-shot: (4, 1, 6)
One-shot: (4, 1, 6)
Few-shot: (4, 1, 6)
```

Explanation Of The Above Program :

~ analyze_csv(file_path) opens the CSV and sets total_rows, empty_rows_count, total_word_count to 0.

~ It creates a csv.reader to iterate over rows.

~ For each row it increments total_rows by 1.

~ if not any(row): checks if all cells are empty strings — if so it increments empty_rows_count.

~ It then iterates each cell in the row.

~ For each cell it does cell.split() and adds len(...) to total_word_count.

~ That counts whitespace–separated words across every cell.

~ After all rows are processed it returns (total_rows, empty_rows_count, total_word_count).

~ Example result (4, 1, 9) means 4 rows total, 1 empty row, 9 words in the file.

The function requires a valid CSV file path to run.

Task Description –5 :

Question 5 : 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 & OUTPUT :

```
import string

def most_frequent_word(paragraph):
  """
  Analyzes a paragraph to find the most frequently occurring word.

  Args:
    paragraph: The input string paragraph.

  Returns:
    The most frequent word in the paragraph.
    Returns any one of the tied words if there is a tie.
  """
  # Convert to lowercase
  paragraph = paragraph.lower()

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

  # Split into words
  words = paragraph.split()

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

  # Find the most frequent word
  if not word_counts:
    return None # Handle empty paragraph case

  words = paragraph.split()

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

  # Find the most frequent word
  if not word_counts:
    return None # Handle empty paragraph case

  most_frequent = max(word_counts, key=word_counts.get)
  return most_frequent

# Example Usage (based on the prompt examples):
print(most_frequent_word("This is a sample paragraph. This paragraph is a sample."))
print(most_frequent_word("Hello world! Hello, world?"))
print(most_frequent_word("A quick brown fox jumps over the lazy dog. The quick brown fox is quick."))
```

```
this
hello
quick
```

Explanation Of The Above Program :

~ most_frequent_word(paragraph) — function to find the word that appears most often in a string.
~ Convert the paragraph to lowercase (paragraph.lower()) so matching is case–insensitive.

~ Remove punctuation using paragraph.translate(str.maketrans('', '', string.punctuation)).
~ Split the cleaned string into words with words = paragraph.split().
~ Initialize an empty dict word_counts = {}.
~ Loop over words and update counts: word_counts[word] = word_counts.get(word, 0) + 1.
~ If word_counts is empty (no words), return None to handle the empty input case.
~ Find the most frequent word with max(word_counts, key=word_counts.get).
~ Return that word as the result (e.g., "this" for "This is a sample paragraph. This paragraph is a sample.")