**Lab 4: Advanced Prompt Engineering – Zero-shot, One-shot, and Few-shot Techniques**

**Assignment-4.1:**

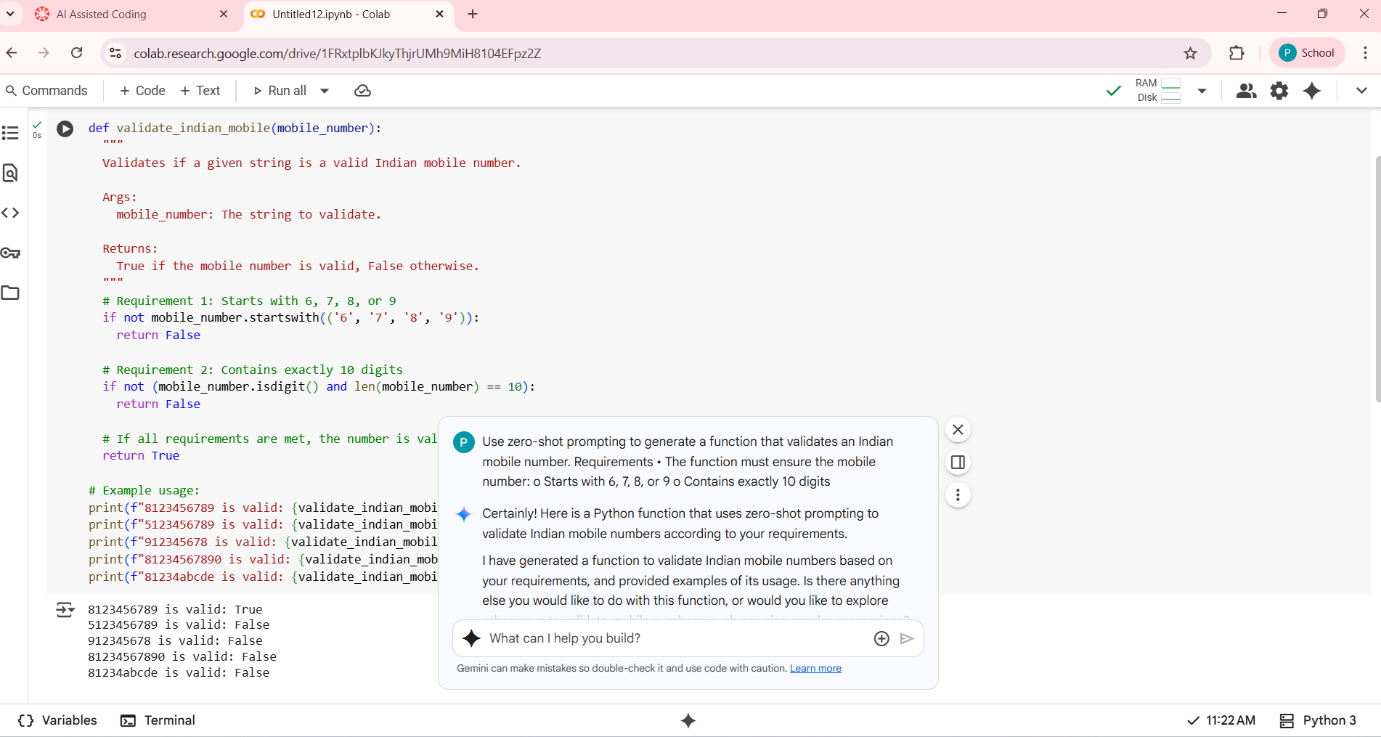
**HTNO:**2403a51331

**Name**:Pujari Navya

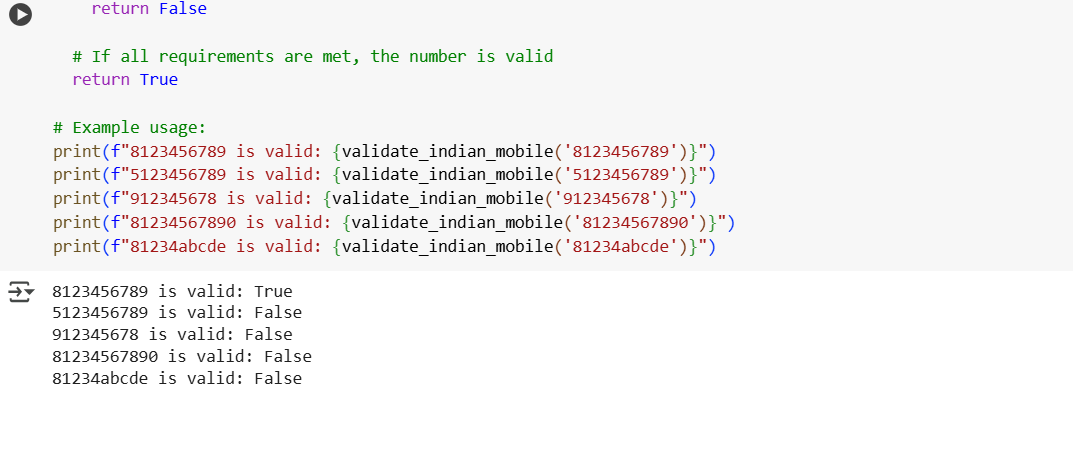
**Batch**:24BTCAICSB13

**Task-1:**

**Prompt**: Use zero-shot prompting to instruct an AI tool to generate a function that validates an Indian mobile number.



**Output:**

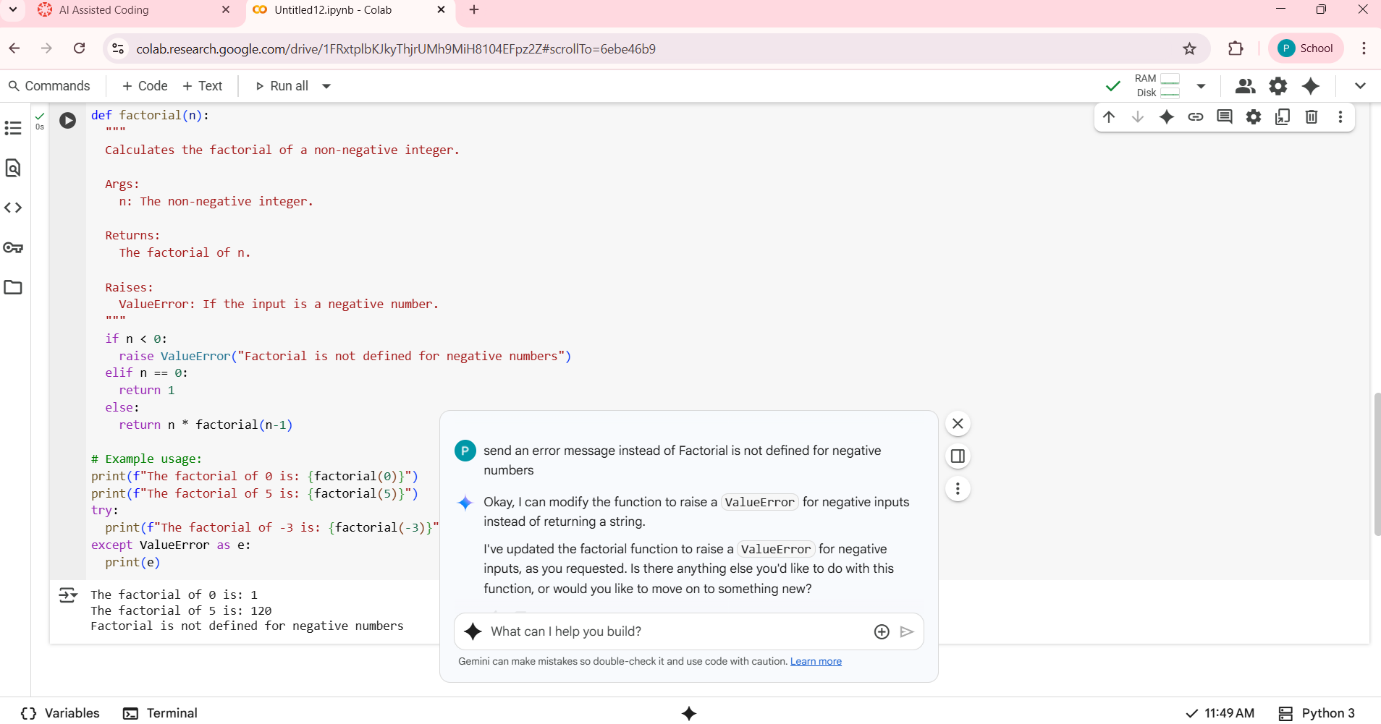


**Explaination:**

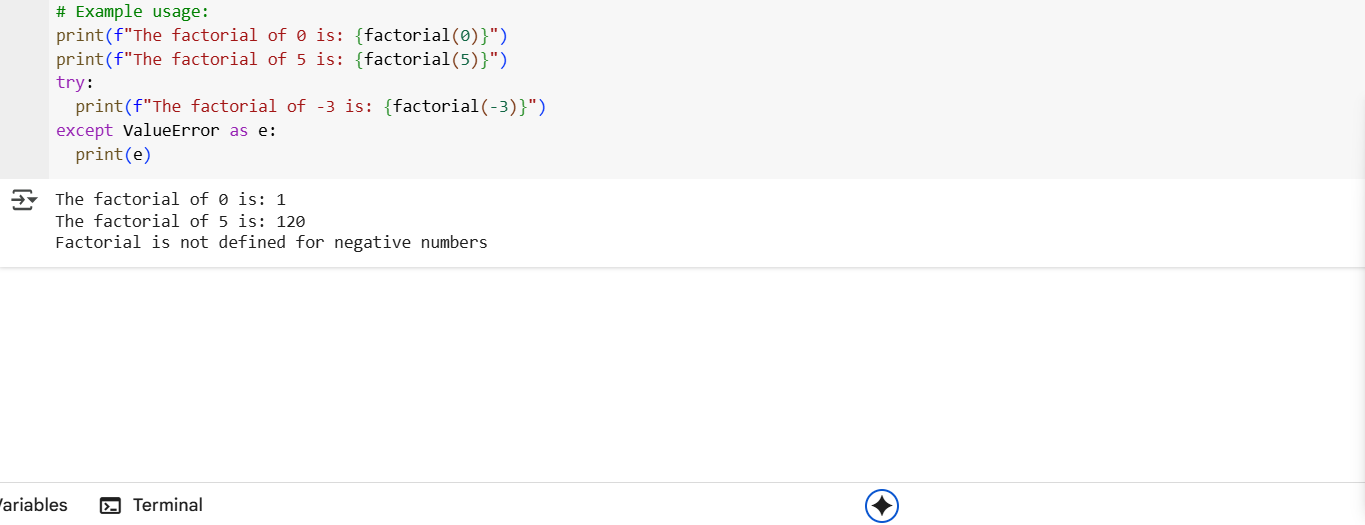
1. **import re**: This line imports the regular expression module in Python, which is used for pattern matching in strings.
2. **def validate\_indian\_mobile\_number(number**):: This line defines the function named validate\_indian\_mobile\_number that accepts one argument, number.
3. **Docstring:** The triple-quoted string is a docstring that explains what the function does, its arguments, and what it returns. It also specifies the criteria for a valid Indian mobile number.
4. **pattern = r"^((\+91|0)?)[789]\d{9}$":** This is the core of the validation. It defines a regular expression pattern:
   * ^: Matches the beginning of the string.
   * (\+91|0)?: This part matches an optional country code. \+91 matches the literal "+91", | means "or", and 0 matches the digit 0. The ? makes the entire group optional (meaning the number can start with "+91", "0", or neither).
   * [789]: Matches a single digit that is either 7, 8, or 9. Indian mobile numbers typically start with these digits.
   * \d{9}: Matches exactly 9 digits (\d matches any digit, and {9} specifies the count).
   * $: Matches the end of the string.
5. **if re.match(pattern, number**):: This line uses the re.match() function to check if the pattern matches the beginning of the number string. If a match is found, it means the number follows the specified format.
6. **return True**: If re.match() finds a match, the function returns True, indicating a valid Indian mobile number.
7. **else:** return False: If re.match() does not find a match at the beginning of the string, the function returns False, indicating an invalid number.

**Task-2:**

**Prompt:** Use one-shot prompting to generate a Python function that calculates the factorial of a number.

****

**Output:**

****

**Explaination:**

1. **def calculate\_factorial(n**):: This line defines a function named calculate\_factorial that takes one argument, n.
2. **"""Docstring""":** This is a docstring that explains what the function does, its arguments, and what it returns**.**
3. **if n == 0::** This is the base case for the recursion (although this implementation uses iteration). The factorial of 0 is defined as 1.
4. **return 1:** If n is 0, the function returns 1**.**
5. **else**:: This block is executed if n is not 0**.**
6. **result = 1**: Initializes a variable result to 1. This variable will store the calculated factorial.
7. **for i in range(1, n + 1)::** This loop iterates from 1 up to and including n.
8. **result \*= i:** In each iteration, the result is multiplied by the current value of i. This performs the multiplication for the factorial calculation (e.g., 1 \* 2 \* 3 \* ... \* n).
9. **return result**: After the loopfinishes, the function returns the final calculated result, which is the factorialof n.

**Task-3:**

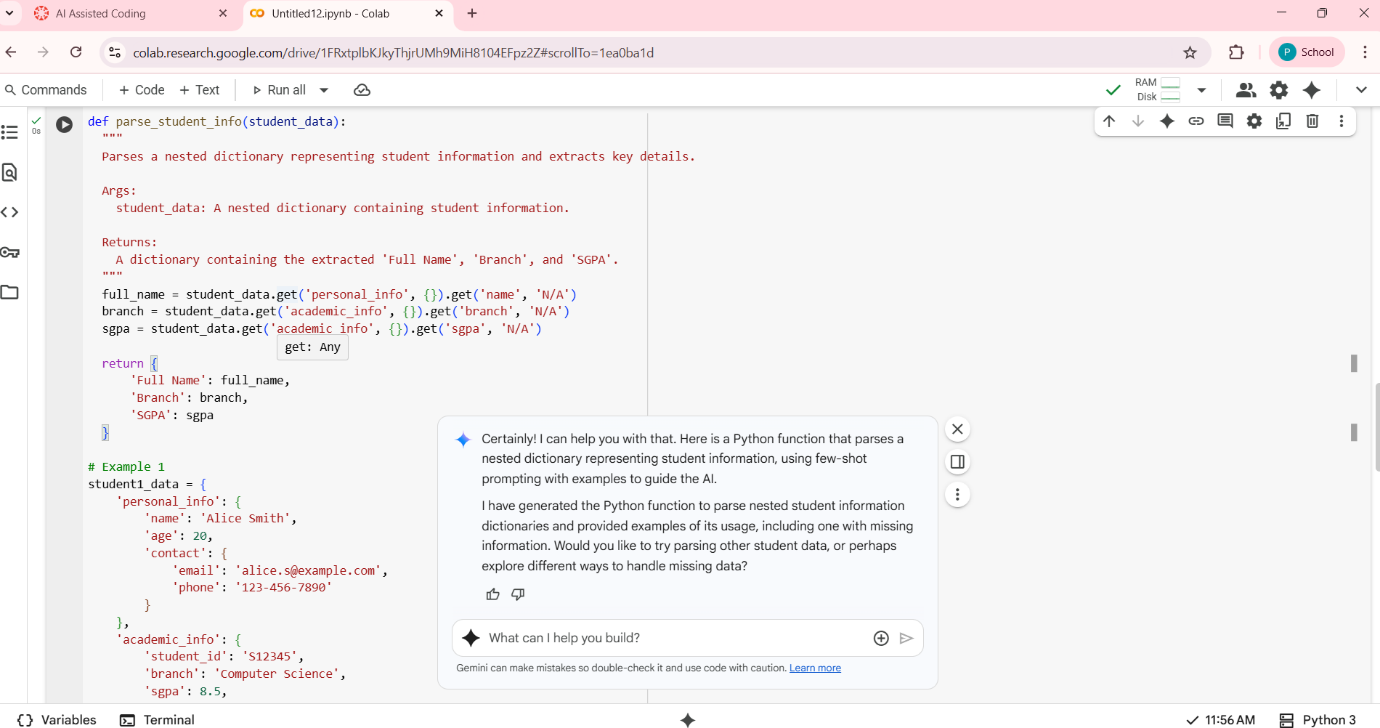
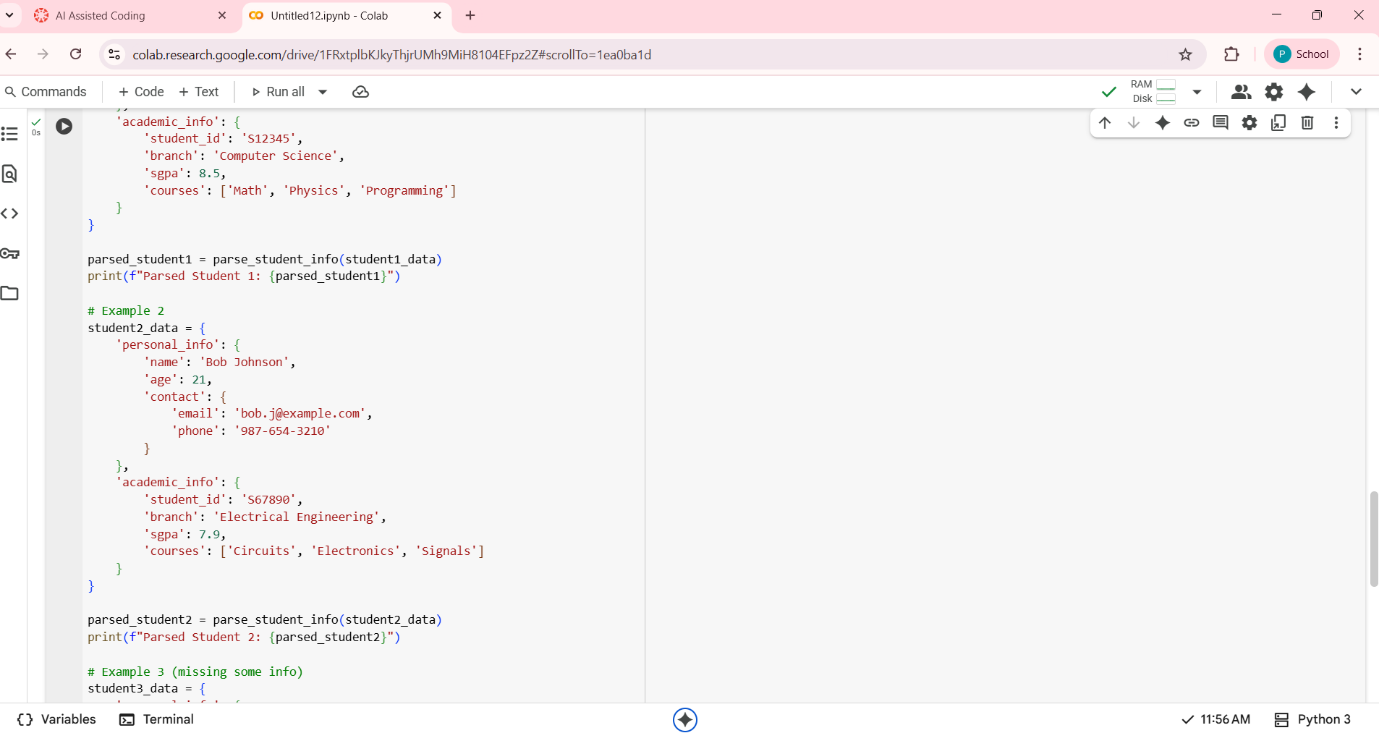
**Prompt:**

Few-Shot Prompting for Nested Dictionary Extraction

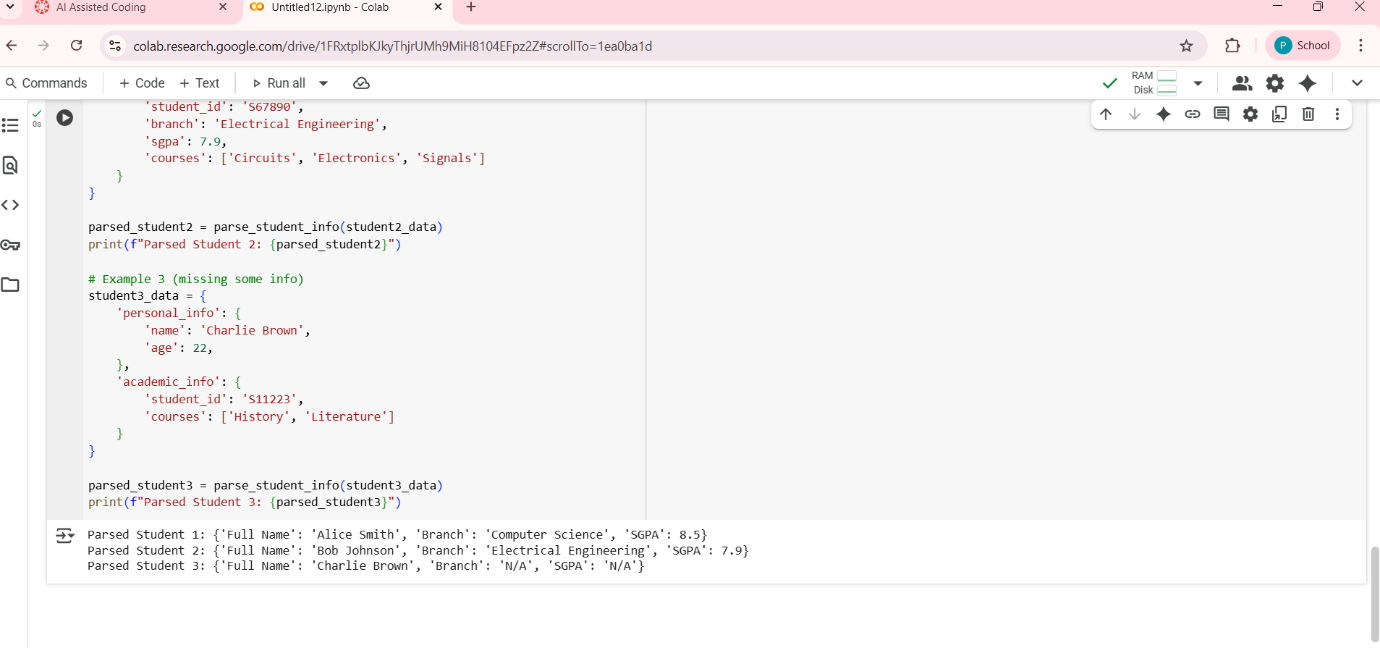
Objective

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
* 
* 

**Output:**

****

**Explaination:**

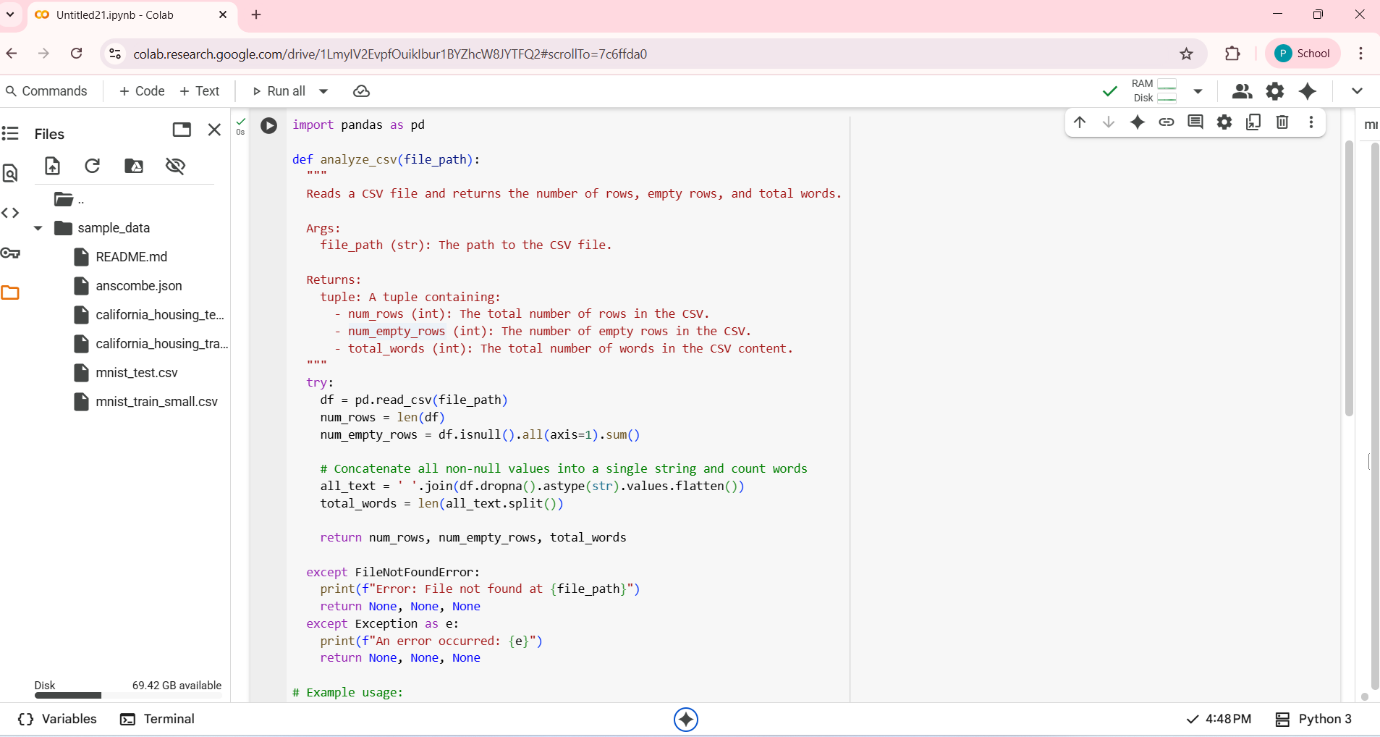
1. **def parse\_student\_info(student\_data)::** This line defines a function named parse\_student\_info that takes one argument, student\_data, which is expected to be a dictionary.
2. **"""Docstring**""": This is a docstring explaining the function's purpose, arguments, and return value.
3. **full\_name = student\_data.get('personal\_info', {}).get('name'):**
   * student\_data.get('personal\_info', {}): This safely attempts to access the key 'personal\_info' in the student\_data dictionary. If 'personal\_info' is not found, it returns an empty dictionary {} instead of raising a KeyError.
   * .get('name'): This then attempts to access the key 'name' within the dictionary returned by the first get(). If 'name' is not found (either because 'personal\_info' was missing or 'name' was missing within 'personal\_info'), it returns None. This chain of .get() calls provides a robust way to handle missing keys at different levels of the nested dictionary.
4. **branch =** student\_data.get('academic\_info', {}).get('branch'): Similar to the full\_name extraction, this line safely attempts to get the 'branch' from the nested 'academic\_info' dictionary.
5. **sgpa = student\_data.get('academic\_info', {}).get('sgpa'):** This line safely attempts to get the 'sgpa' from the nested 'academic\_info' dictionary.
6. **if full\_name is not None and branch is not None and sgpa is not None::** This conditional statement checks if all three required pieces of information (full\_name, branch, and sgpa) were successfully extracted (i.e., they are not None).
7. **return { ... }:** If all information is found, the function returns a new dictionary containing the extracted 'Full Name', 'Branch', and 'SGPA' with descriptive keys**.**
8. **else::** This block is executed if any of the required information is missing**.**
9. **return None:** If any of the required information is missing, the function returns None to indicate that the parsing was not fully successful.

**Task-4:**

**Prompt:**

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
* 

**Output:**

****

**Explaination:**

1. **Read the code carefully, line by line**: Pay attention to keywords, variable names, and the overall structure.
2. **Identify the purpose of the code:**What problem is this code trying to solve? What is its overall goal?
3. **Break down the code into smaller parts:**Functions, loops, conditional statements, and classes are often good places to start. Understand what each part does individually.
4. **Understand the variables**: What data do they hold? How are they used and modified throughout the code?
5. **Trace the flow of execution:**How does the program move from one part of the code to another? What conditions affect the execution path?
6. **Look for comments and documentation:**These can provide valuable insights into the developer's intentions and the code's functionality**.**
7. **If possible, run the code and experiment:**Change some values and see how the output changes. This can help you understand how the code works in practice.
8. **Use online resources:**If you encounter unfamiliar syntax or concepts, search for explanations and examples online**.**

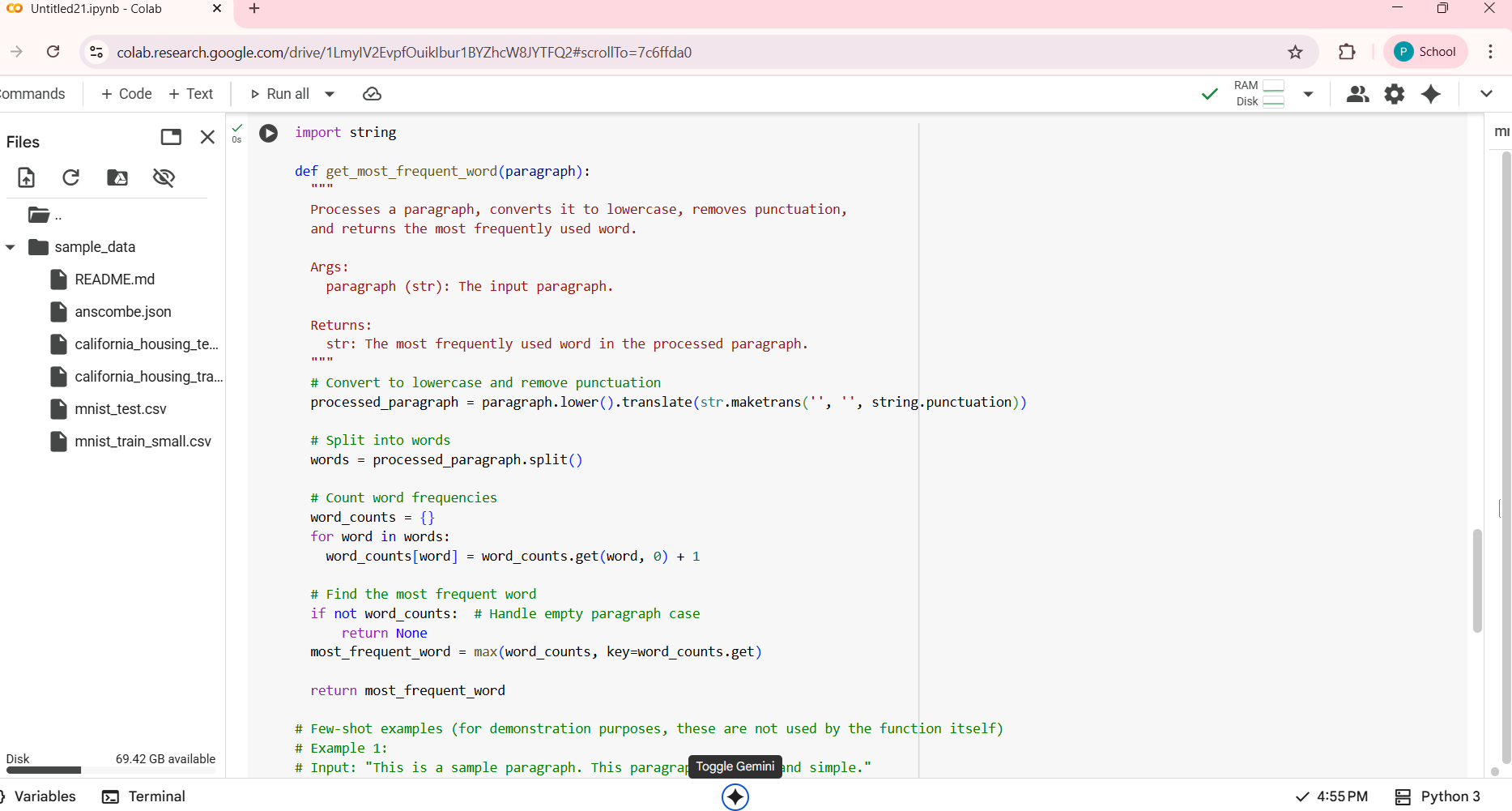
**Task-5:**

**Prompt:**

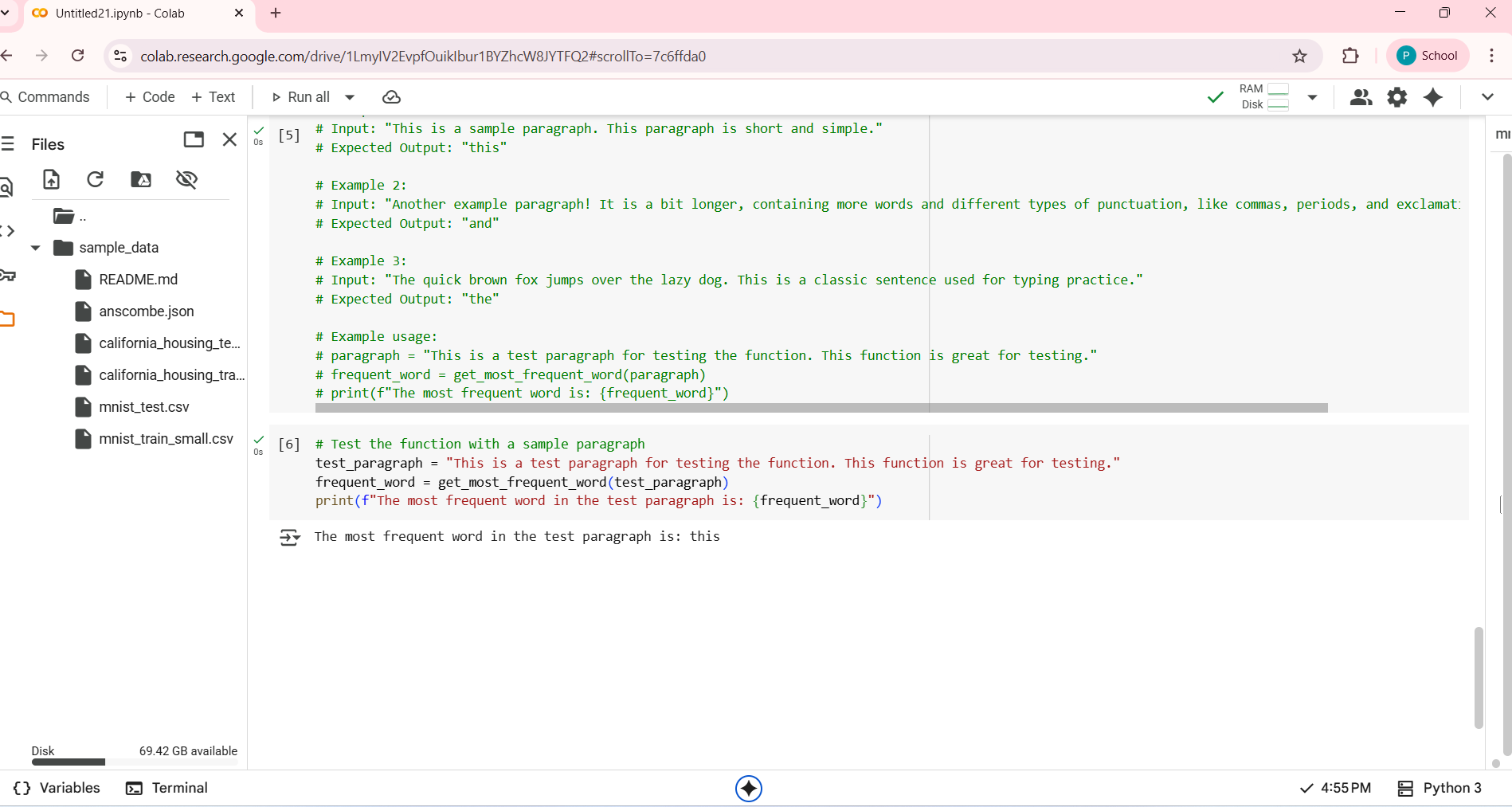
Use few-shot prompting (with at least 3 examples) to generate a Python function that processes text and analyzes word frequency.

Requirements

The function must:

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

**Output:**

****

**Explaination:**

1. **import string:** Imports the string module, which contains a collection of useful string constants, including string.punctuation**.**
2. **from collections import Counter:** Imports the Counter class from the collections module. Counter is a specialized dictionary subclass for counting hashable objects.
3. **def analyze\_word\_frequency(paragraph)::** Defines the function analyze\_word\_frequency that takes one argument, paragraph.
4. **if not paragraph::** Checks if the input paragraph is empty. If it is, the function returns None**.**
5. **paragraph = paragraph.lower():** Converts the entire paragraph string to lowercase using the .lower() method. This ensures that words like "This" and "this" are treated as the same word**.**
6. **translator = str.maketrans('', '', string.punctuation):** Creates a translation table. str.maketrans(x, y, z) creates a mapping where characters in x are mapped to characters in y, and characters in z are mapped to None (effectively removed). In this case, it maps all characters in string.punctuation to None.
7. **paragraph = paragraph.translate(translator):** Applies the created translation table to the paragraph string, removing all punctuation characters.
8. **words = paragraph.split():** Splits the processed paragraph string into a list of words using whitespace as the delimiter.
9. **word\_counts = Counter(words):** Creates a Counter object named word\_counts from the list of words. This object automatically counts the occurrences of each unique word.
10. **if word\_counts**:: Checks if the word\_counts Counter is not empty (meaning there were words in the paragraph after processing).
11. **most\_common\_word = word\_counts.most\_common(1)[0][0]:**
    * **word\_counts.most\_common(1):** Returns a list of the 1 most common elements and their counts from the Counter object. The result is a list of tuples, like [('word', count)].
    * [0]: Accesses the first tuple in the list (which is the most common word and its count).
    * [0]: Accesses the first element of the tuple (which is the word itself).
12. **return most\_common\_word:** Returns the extracted most frequently used word.
13. **else: return None:** If word\_counts is empty (e.g., the input was only punctuation), the function returns None.