

ASSIGNMENT 4.3

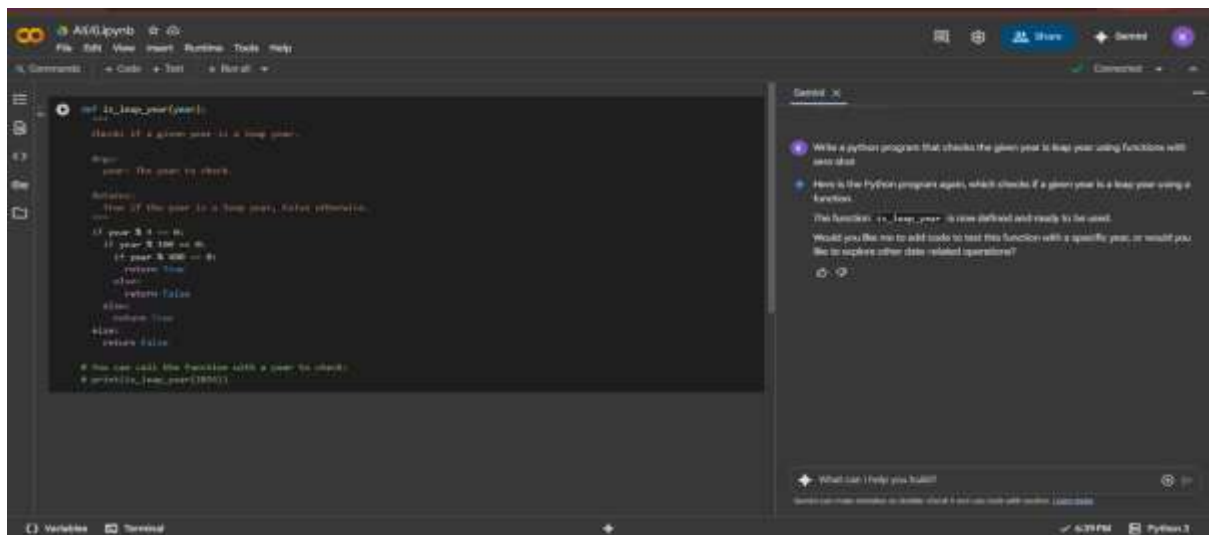
NAME: K. VINOD

2403A52132

Task 1:

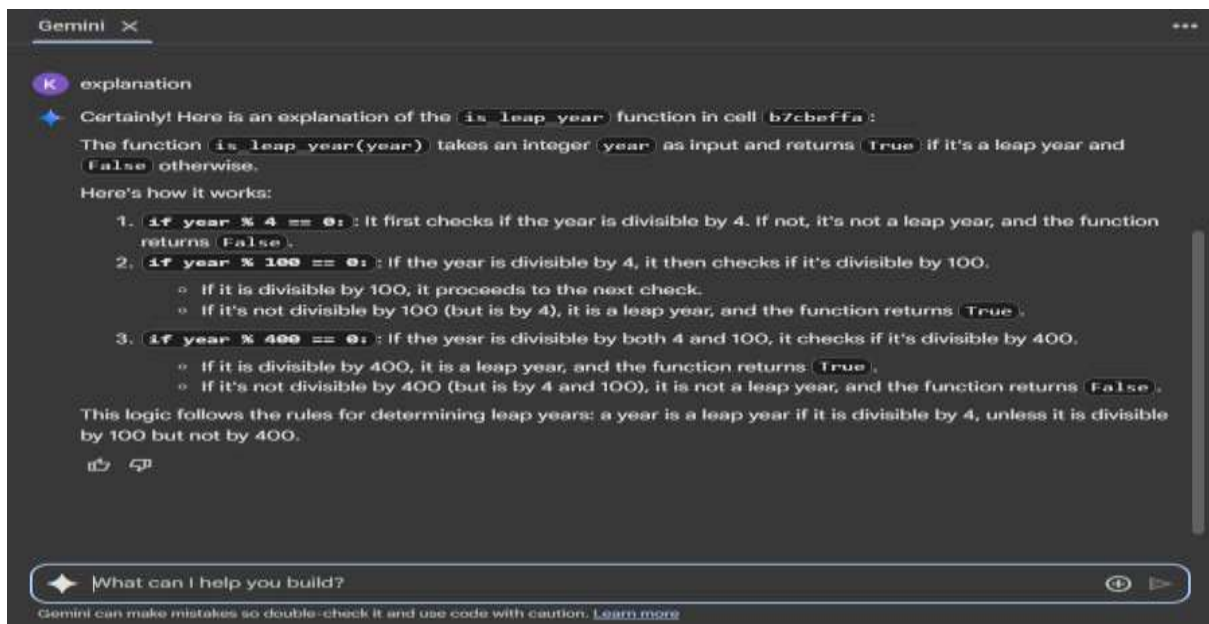
Write a python program that checks the given year is leap year using functions with zero shot.

Code:



```
def is_leap_year(year):  
    """  
    Checks if a given year is a leap year.  
    """  
    # Step 1: Check if the year is a leap year.  
    # Step 2: Check if the year is a leap year, false otherwise.  
    if year % 4 == 0:  
        if year % 100 == 0:  
            if year % 400 == 0:  
                return True  
            else:  
                return False  
        else:  
            return True  
    else:  
        return False  
  
# You can call the function with a year to check.  
# print(is_leap_year(2000))
```

Explanation:



explanation

Certainly! Here is an explanation of the `is_leap_year` function in cell `b7cbeffa`:

The function `is_leap_year(year)` takes an integer `year` as input and returns `True` if it's a leap year and `False` otherwise.

Here's how it works:

- `if year % 4 == 0`: It first checks if the year is divisible by 4. If not, it's not a leap year, and the function returns `False`.
- `if year % 100 == 0`: If the year is divisible by 4, it then checks if it's divisible by 100.
 - If it is divisible by 100, it proceeds to the next check.
 - If it's not divisible by 100 (but is by 4), it is a leap year, and the function returns `True`.
- `if year % 400 == 0`: If the year is divisible by both 4 and 100, it checks if it's divisible by 400.
 - If it is divisible by 400, it is a leap year, and the function returns `True`.
 - If it's not divisible by 400 (but is by 4 and 100), it is not a leap year, and the function returns `False`.

This logic follows the rules for determining leap years: a year is a leap year if it is divisible by 4, unless it is divisible by 100 but not by 400.

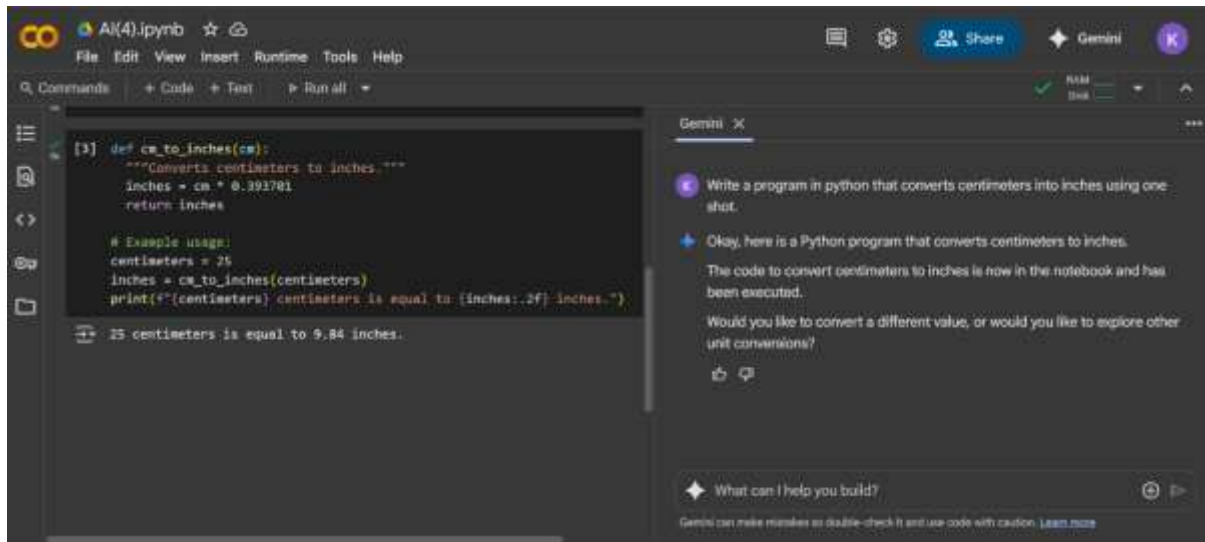
What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Task 2:

Write a program in python that converts centimeters into inches using one shot.

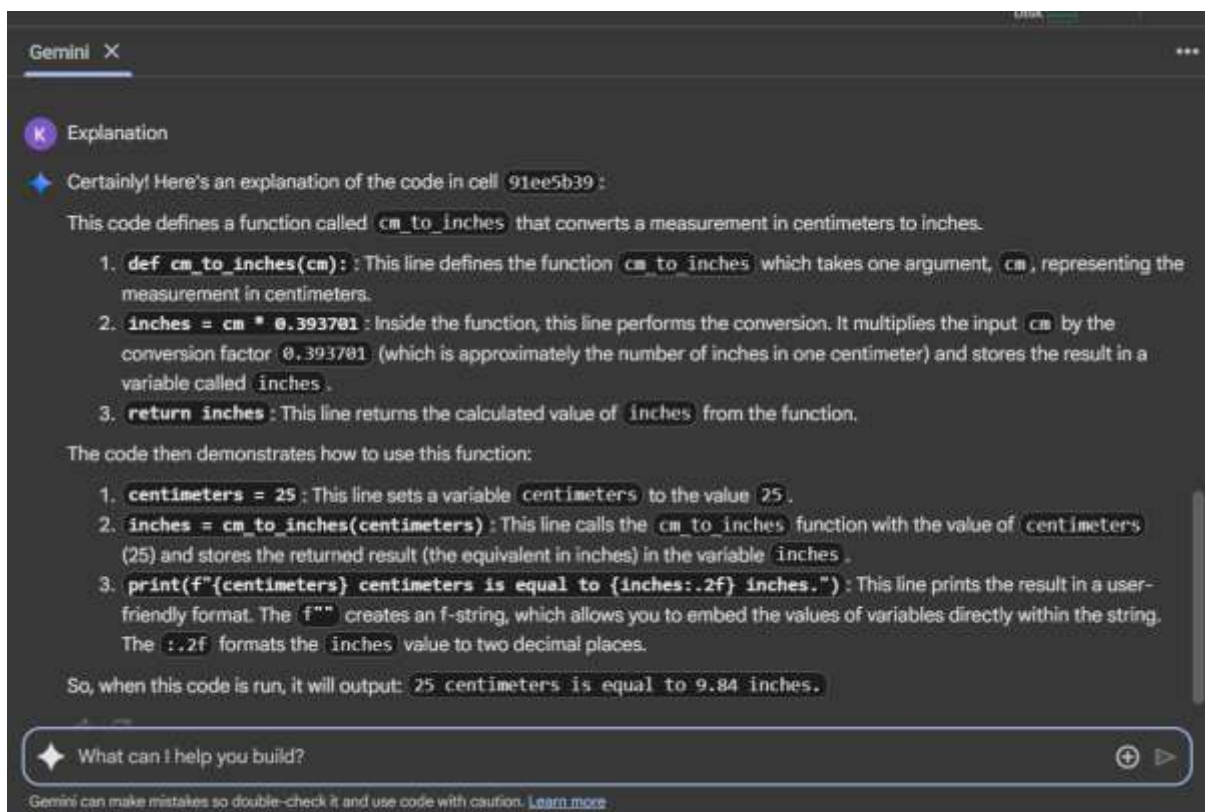
Code:



The screenshot shows a Jupyter Notebook titled 'AI(4).ipynb'. The code cell contains a function `cm_to_inches(cm)` that converts centimeters to inches. The function is defined with a docstring, a conversion factor, and an example usage. The output of the code cell shows the result of the function call: '25 centimeters is equal to 9.84 inches.' The right sidebar shows the Gemini chat interface with the user's prompt and the assistant's response.

```
[3] def cm_to_inches(cm):  
    """Converts centimeters to inches."""  
    inches = cm * 0.393701  
    return inches  
  
# Example usage:  
centimeters = 25  
inches = cm_to_inches(centimeters)  
print(f"{centimeters} centimeters is equal to {inches:.2f} inches.")  
  
25 centimeters is equal to 9.84 inches.
```

Explanation:



The screenshot shows the Gemini chat interface with the user's prompt and the assistant's response. The assistant explains the code in a structured manner, listing the steps and the output.

Explanation

Certainly! Here's an explanation of the code in cell 91ee5b39:

This code defines a function called `cm_to_inches` that converts a measurement in centimeters to inches.

- `def cm_to_inches(cm):`: This line defines the function `cm_to_inches` which takes one argument, `cm`, representing the measurement in centimeters.
- `inches = cm * 0.393701`: Inside the function, this line performs the conversion. It multiplies the input `cm` by the conversion factor `0.393701` (which is approximately the number of inches in one centimeter) and stores the result in a variable called `inches`.
- `return inches`: This line returns the calculated value of `inches` from the function.

The code then demonstrates how to use this function:

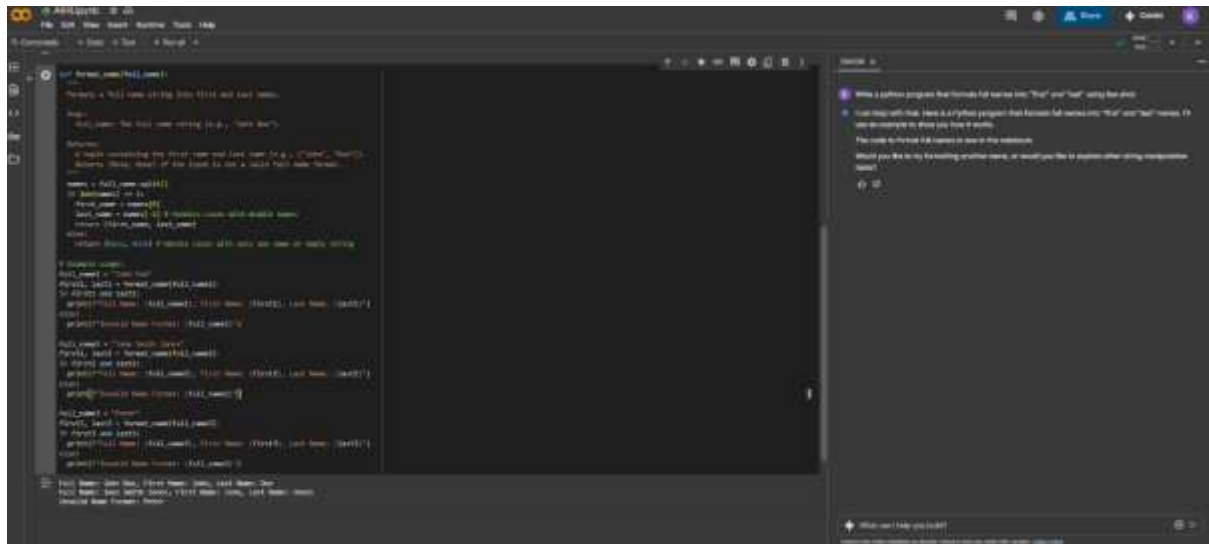
- `centimeters = 25`: This line sets a variable `centimeters` to the value `25`.
- `inches = cm_to_inches(centimeters)`: This line calls the `cm_to_inches` function with the value of `centimeters` (25) and stores the returned result (the equivalent in inches) in the variable `inches`.
- `print(f"{centimeters} centimeters is equal to {inches:.2f} inches.")`: This line prints the result in a user-friendly format. The `f""` creates an f-string, which allows you to embed the values of variables directly within the string. The `:.2f` formats the `inches` value to two decimal places.

So, when this code is run, it will output: `25 centimeters is equal to 9.84 inches.`

Task 3:

Write a python program that formats full names into “first” and “last” using few shot.

Code:



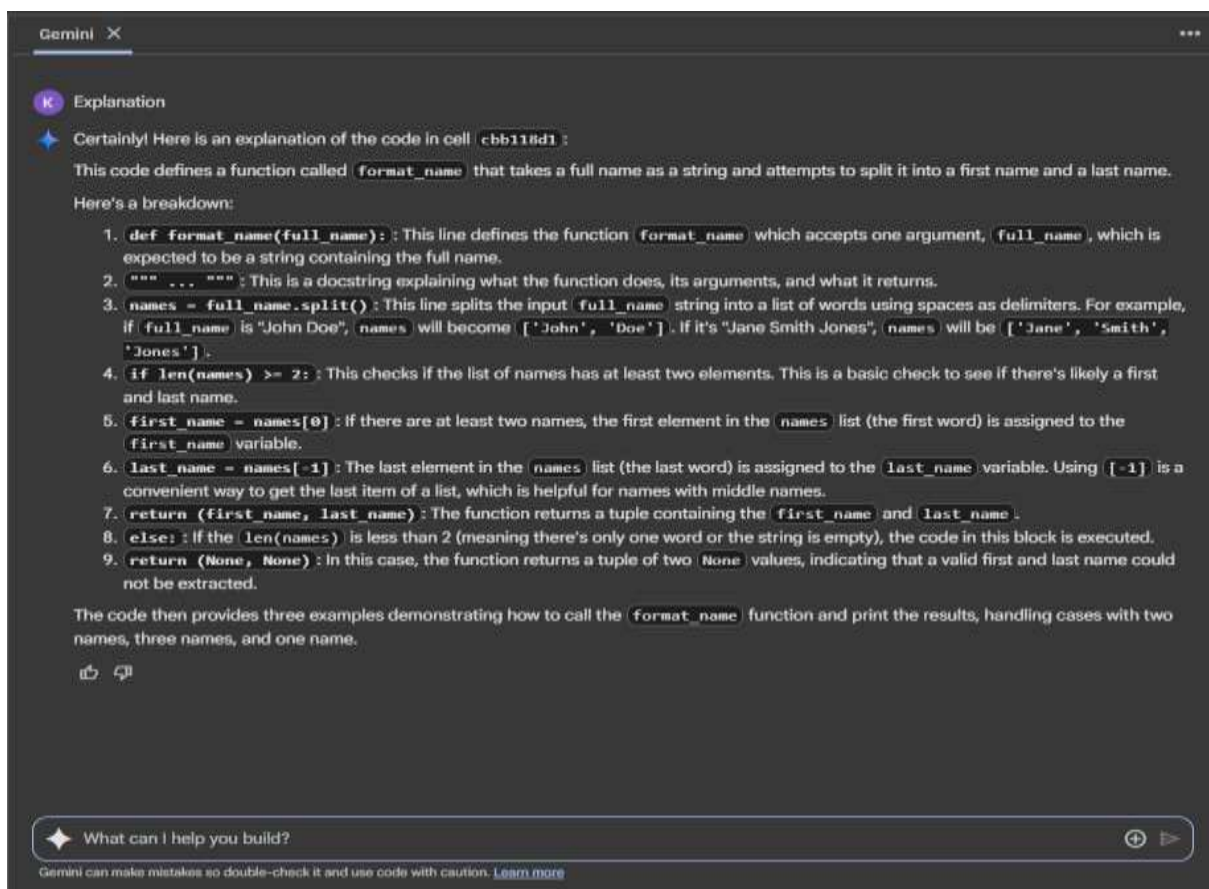
```
def format_name(full_name):
    """
    Takes a full name string (e.g., "John Doe")
    and returns a tuple containing the first name and last name (e.g., ("John", "Doe")).
    If the name is empty or contains only one word, it returns (None, None).
    """
    names = full_name.split()
    if len(names) >= 2:
        first_name = names[0]
        last_name = names[-1]
        return (first_name, last_name)
    else:
        return (None, None)

# Examples
first_name, last_name = format_name("John Doe")
print(f"First Name: {first_name}, Last Name: {last_name}")

first_name, last_name = format_name("Jane Smith Jones")
print(f"First Name: {first_name}, Last Name: {last_name}")

first_name, last_name = format_name("")
print(f"First Name: {first_name}, Last Name: {last_name}")
```

Explanation:



Explanation

Certainly! Here is an explanation of the code in cell `cbb118d1`:

This code defines a function called `format_name` that takes a full name as a string and attempts to split it into a first name and a last name. Here's a breakdown:

- `def format_name(full_name):`: This line defines the function `format_name` which accepts one argument, `full_name`, which is expected to be a string containing the full name.
- `""" ... """`: This is a docstring explaining what the function does, its arguments, and what it returns.
- `names = full_name.split()`: This line splits the input `full_name` string into a list of words using spaces as delimiters. For example, if `full_name` is "John Doe", `names` will become `['John', 'Doe']`. If it's "Jane Smith Jones", `names` will be `['Jane', 'Smith', 'Jones']`.
- `if len(names) >= 2:`: This checks if the list of names has at least two elements. This is a basic check to see if there's likely a first and last name.
- `first_name = names[0]`: If there are at least two names, the first element in the `names` list (the first word) is assigned to the `first_name` variable.
- `last_name = names[-1]`: The last element in the `names` list (the last word) is assigned to the `last_name` variable. Using `[-1]` is a convenient way to get the last item of a list, which is helpful for names with middle names.
- `return (first_name, last_name)`: The function returns a tuple containing the `first_name` and `last_name`.
- `else:`: If the `len(names)` is less than 2 (meaning there's only one word or the string is empty), the code in this block is executed.
- `return (None, None)`: In this case, the function returns a tuple of two `None` values, indicating that a valid first and last name could not be extracted.

The code then provides three examples demonstrating how to call the `format_name` function and print the results, handling cases with two names, three names, and one name.

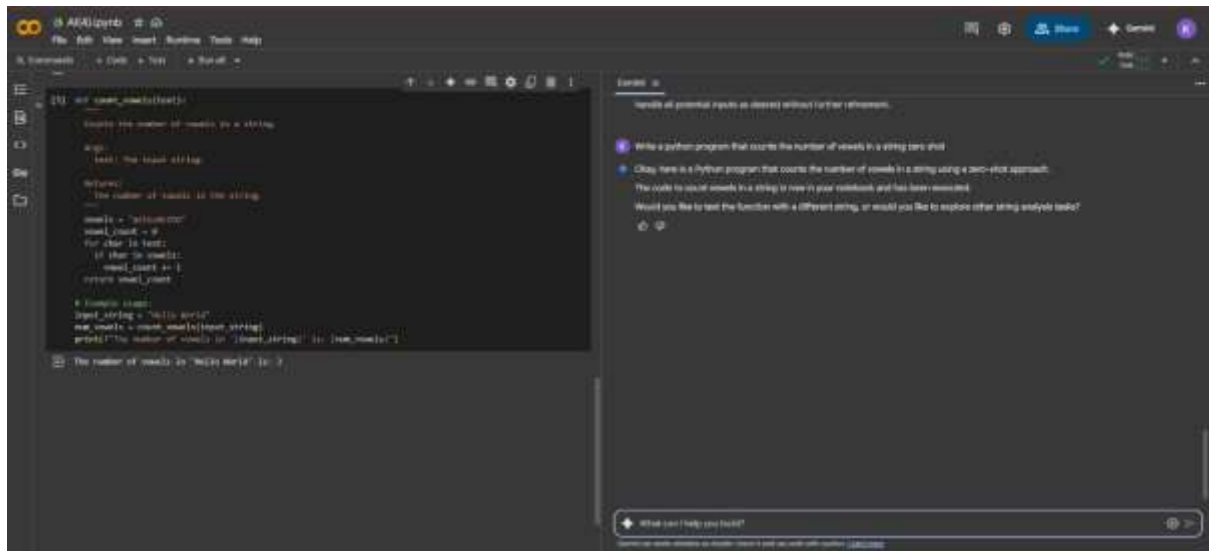
What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

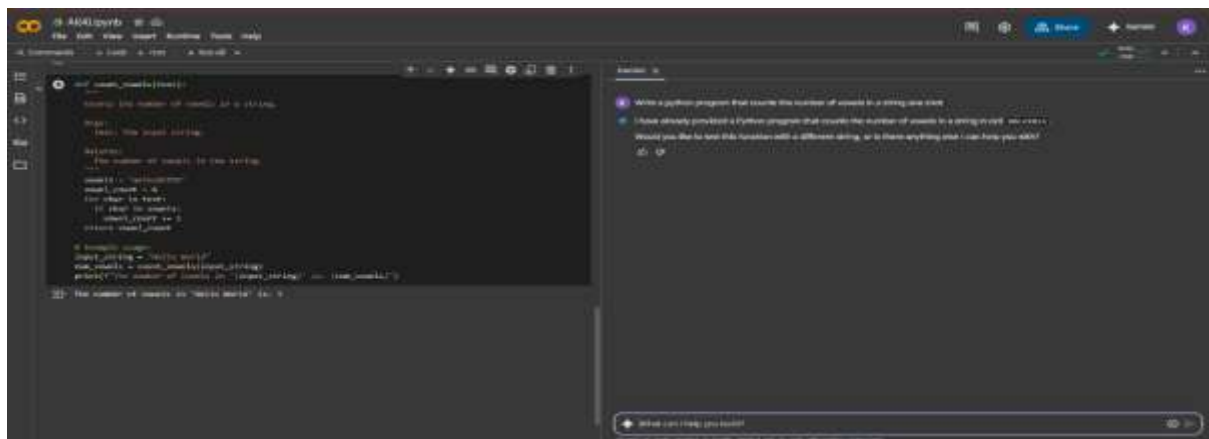
Task 4:

Write a python program that counts the number of vowels in a string.

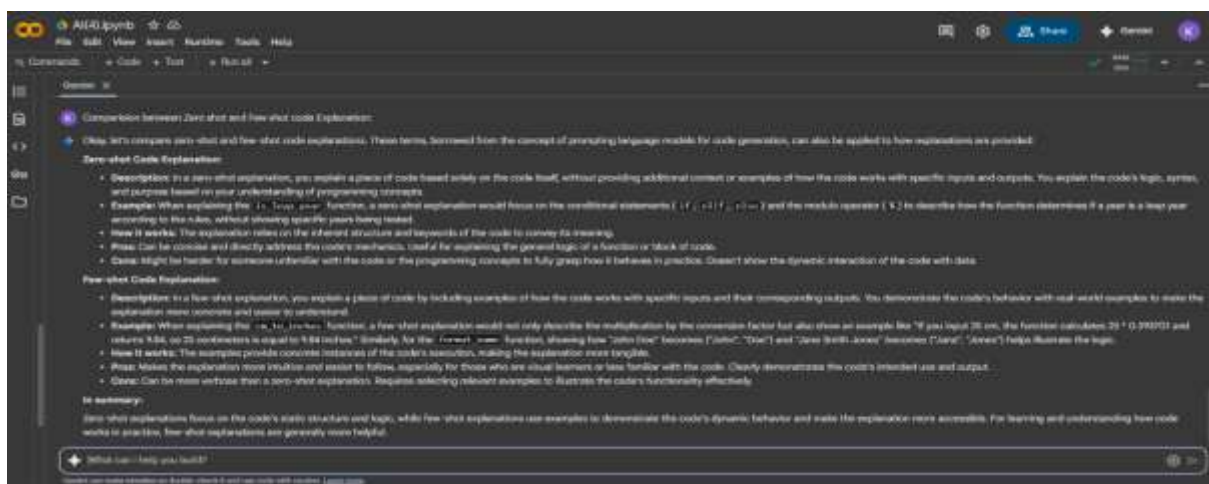
Zero shot code:



One shot code:



Explanation:



Task 5:

Write a python program that reads a .txt file and returns the number of lines using few short.

Code:

```

# Create a .txt file for demonstration.
with open('sample.txt', 'w') as f:
    f.write("This is line 1.\n")
    f.write("This is line 2.\n")
    f.write("This is line 3.\n")

def count_lines_in_file(filename):
    """
    Reads a text file and returns the number of lines.

    Args:
        filename: The path to the text file.

    Returns:
        The number of lines in the file.

    """
    try:
        with open(filename, 'r') as f:
            line_count = sum(1 for line in f)
            return line_count
    except FileNotFoundError:
        print(f"Error: file '{filename}' not found.")
        return -1 # Indicate an error

# Example usage:
file_name = 'sample.txt'
num_lines = count_lines_in_file(file_name)

if num_lines != -1:
    print(f"The number of lines in '{file_name}' is: {num_lines}")

The number of lines in 'sample.txt' is: 3

```

Explanation:

```

# Explanation

# Create sample.txt as explained of the code in calls, variables and comments)

Call count_lines_in_file():
This cell creates a simple text file named 'sample.txt' for demonstration purposes.

1. with open('sample.txt', 'w') as f: This line opens a file named 'sample.txt' in write mode ('w'). The with statement ensures that the file is automatically closed even if an error occurs. The opened file object is assigned to the variable f. If 'sample.txt' already exists, its contents will be overwritten.
2. f.write("This is line 1.\n") This line writes the string "This is line 1" followed by a newline character (\n) to the file.
3. f.write("This is line 2.\n") This line writes "This is line 2" and a newline to the file.
4. f.write("This is line 3.\n") This line writes "This is line 3" and a newline to the file.

After this cell is executed, you will have a file named 'sample.txt' in your Jupyter environment containing three lines of text.

Call count_lines_in_file():
This cell defines a function to count the lines in a given file and then uses it to count the lines in the 'sample.txt' file created in the previous cell.

5. def count_lines_in_file(filename): This line defines a function named count_lines_in_file that takes one argument, filename, which is the path to the file you want to count lines in.
6. """ This is a docstring explaining the function's purpose, arguments, and return value.
7. try: This block starts a try...except block, which is used for error handling. It attempts to execute the code within the try block.
8. with open(filename, 'r') as f: This line opens the file specified by filename in read mode ('r'). Again, the with statement ensures the file is closed properly. The file object is assigned to f.
9. line_count = sum(1 for line in f): This is a concise way to count the lines. It uses a generator expression (1 for line in f) which yields 1 for each line in the file. The sum() function then adds up all these 1s, effectively counting the lines.
10. return line_count: If the file is opened successfully and lines are counted, this line returns the total line_count.
11. except FileNotFoundError: This block catches the FileNotFoundError if the file specified by filename does not exist.
12. print(f"Error: file '{filename}' not found.") If a FileNotFoundError occurs, this line prints an error message indicating that the file was not found.
13. return -1: If a FileNotFoundError occurs, the function returns -1 to indicate that an error occurred and the line count could not be determined.
14. file_name = 'sample.txt': This line sets the file_name variable to 'sample.txt'.
15. num_lines = count_lines_in_file(file_name): This line calls the count_lines_in_file function with 'sample.txt' as the argument and stores the returned value (the number of lines or -1) in the num_lines variable.
16. if num_lines != -1: This checks if the function returned a valid line count (i.e., not -1).
17. print(f"The number of lines in '{file_name}' is: {num_lines}") If num_lines is not -1, this line prints the number of lines in the file.

When these cells are executed, the first cell creates 'sample.txt' and the second cell counts the lines in it and prints the result, which is "The number of lines in 'sample.txt' is: 3". The try...except block makes the count_lines_in_file function more robust by handling cases where the specified file doesn't exist.

```