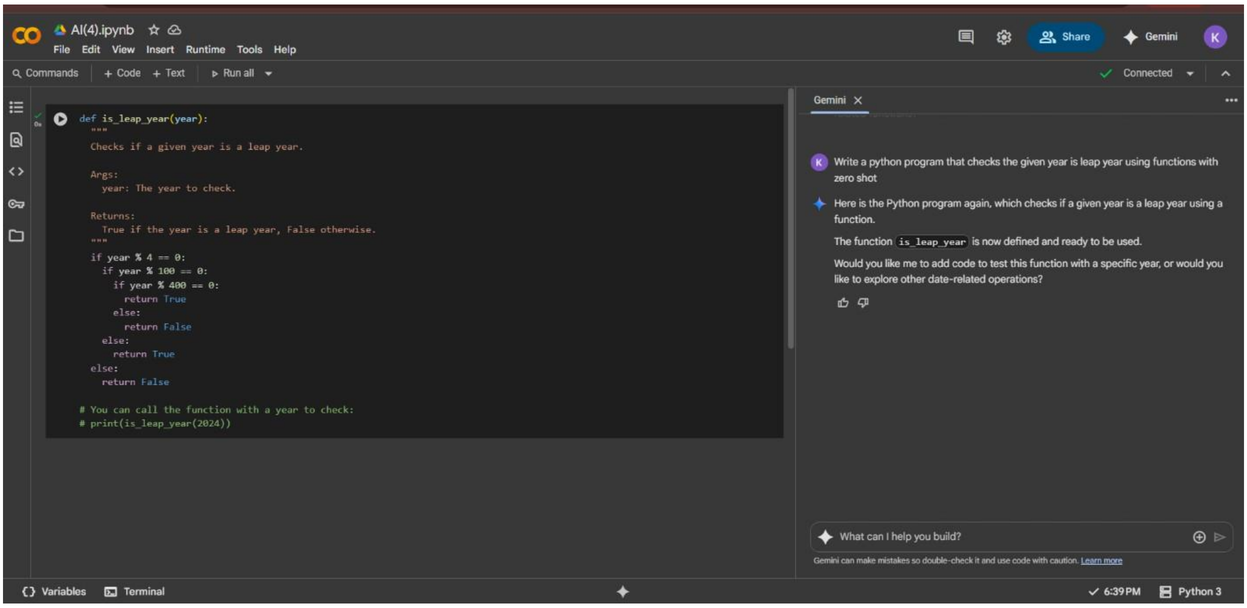
. ASSIGNMENT_4.3 .

NAME: K.SRI SAI HARSHA KISHORE
2403A52133

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.

  Args:
    year: The year to check.

  Returns:
    True 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(2024))
```

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

Here is the Python program again, which checks if a given year is a leap year using a function.

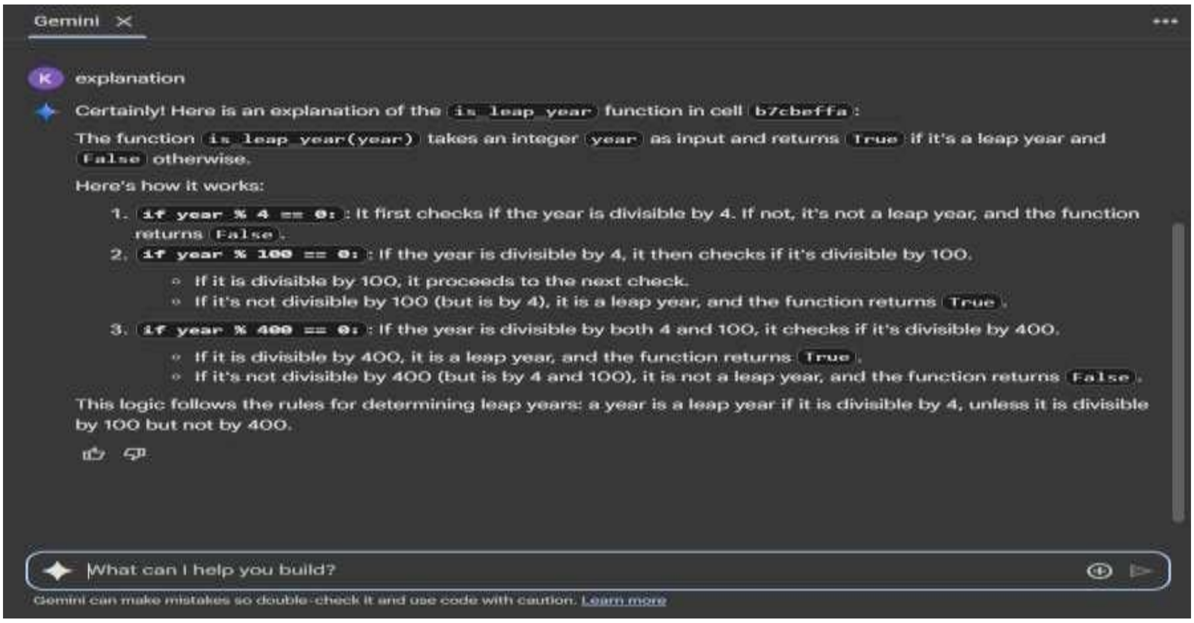
The function `is_leap_year` is now defined and ready to be used.

Would you like me to add code to test this function with a specific year, or would you like to explore other date-related operations?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

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:

1. `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`.
2. `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`.
3. `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.

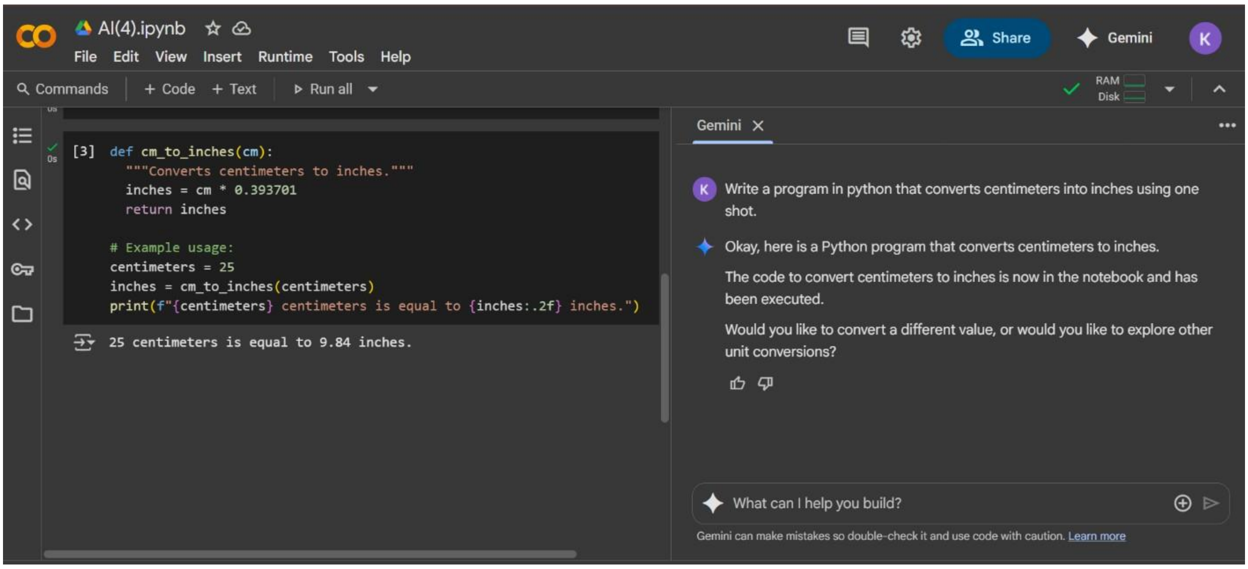
What can I help you build?

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:



```
[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.
```

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

Okay, here is a Python program that converts centimeters to inches.

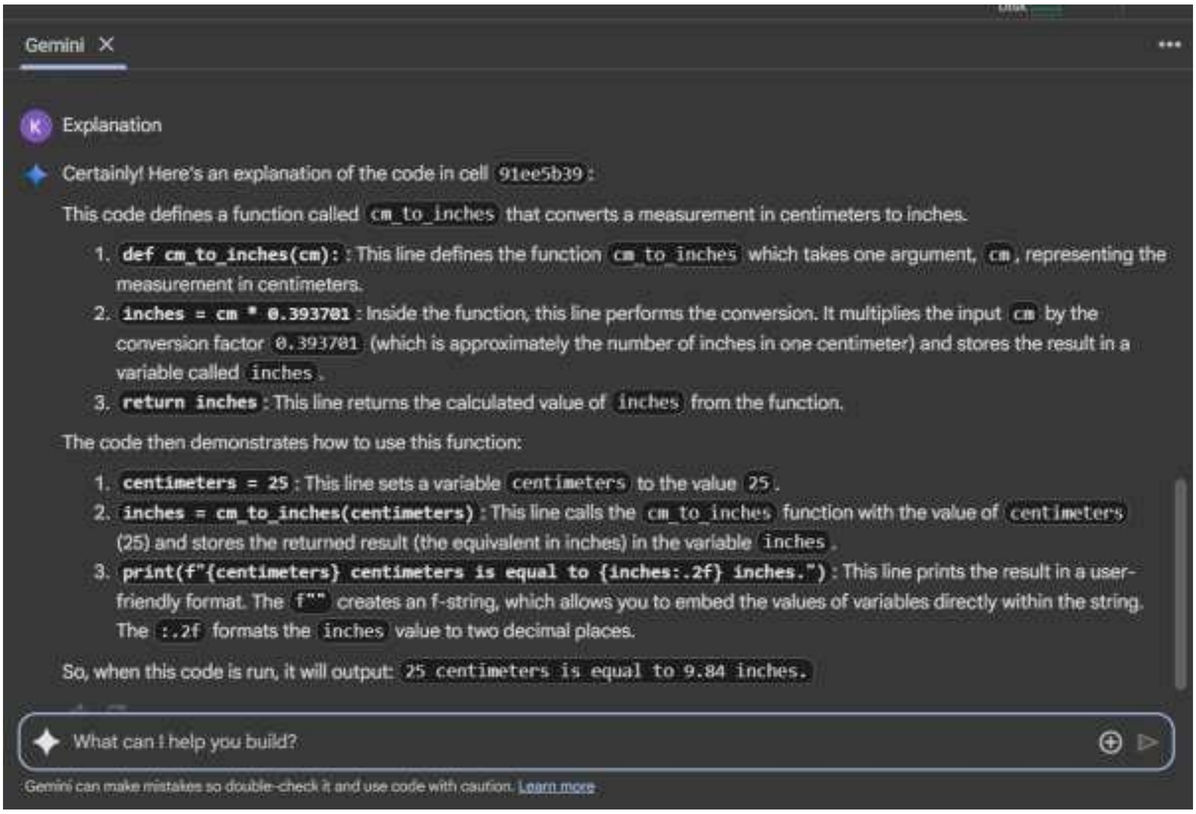
The code to convert centimeters to inches is now in the notebook and has been executed.

Would you like to convert a different value, or would you like to explore other unit conversions?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



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.

1. `def cm_to_inches(cm):`: This line defines the function `cm_to_inches` which takes one argument, `cm`, representing the measurement in centimeters.
2. `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`.
3. `return inches`: This line returns the calculated value of `inches` from the function.

The code then demonstrates how to use this function:

1. `centimeters = 25`: This line sets a variable `centimeters` to the value `25`.
2. `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`.
3. `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.`

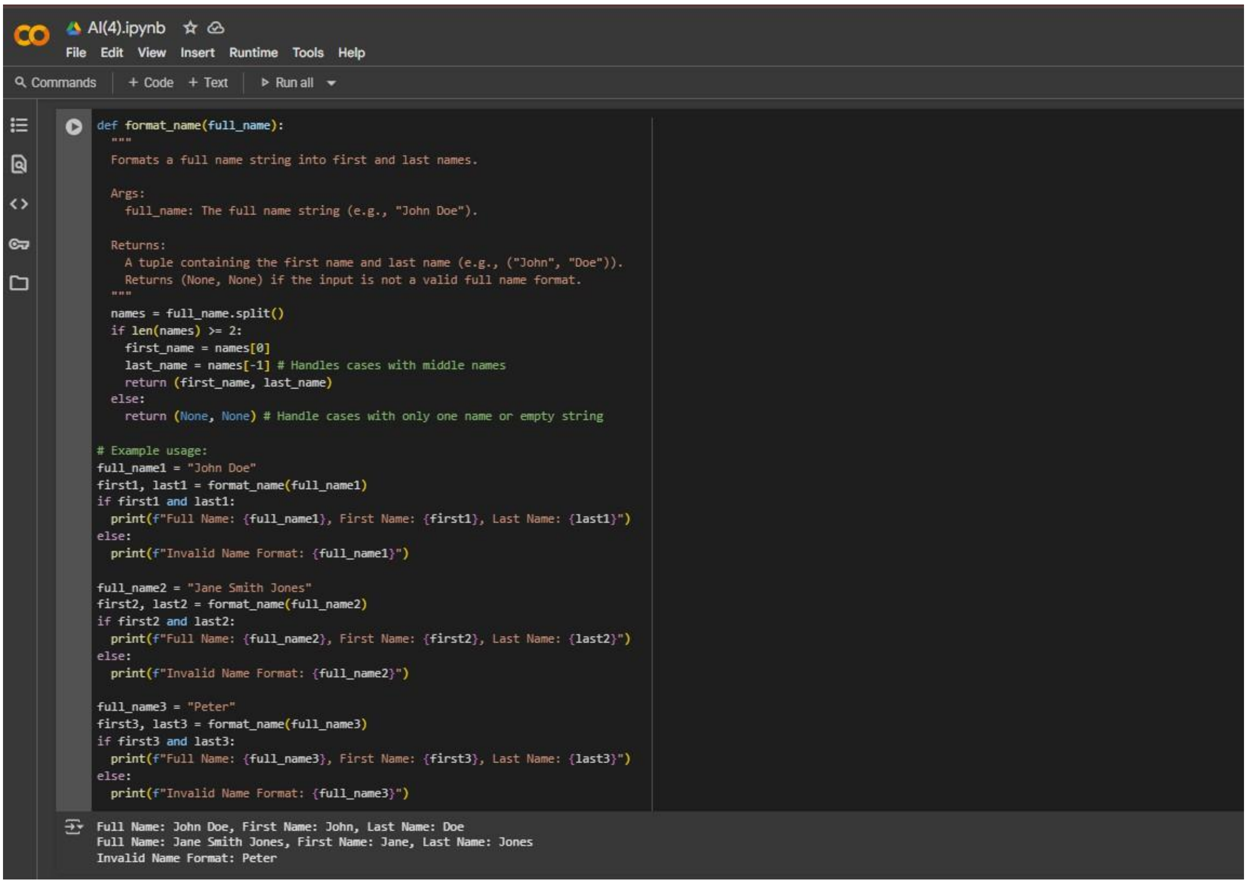
What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 3:

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

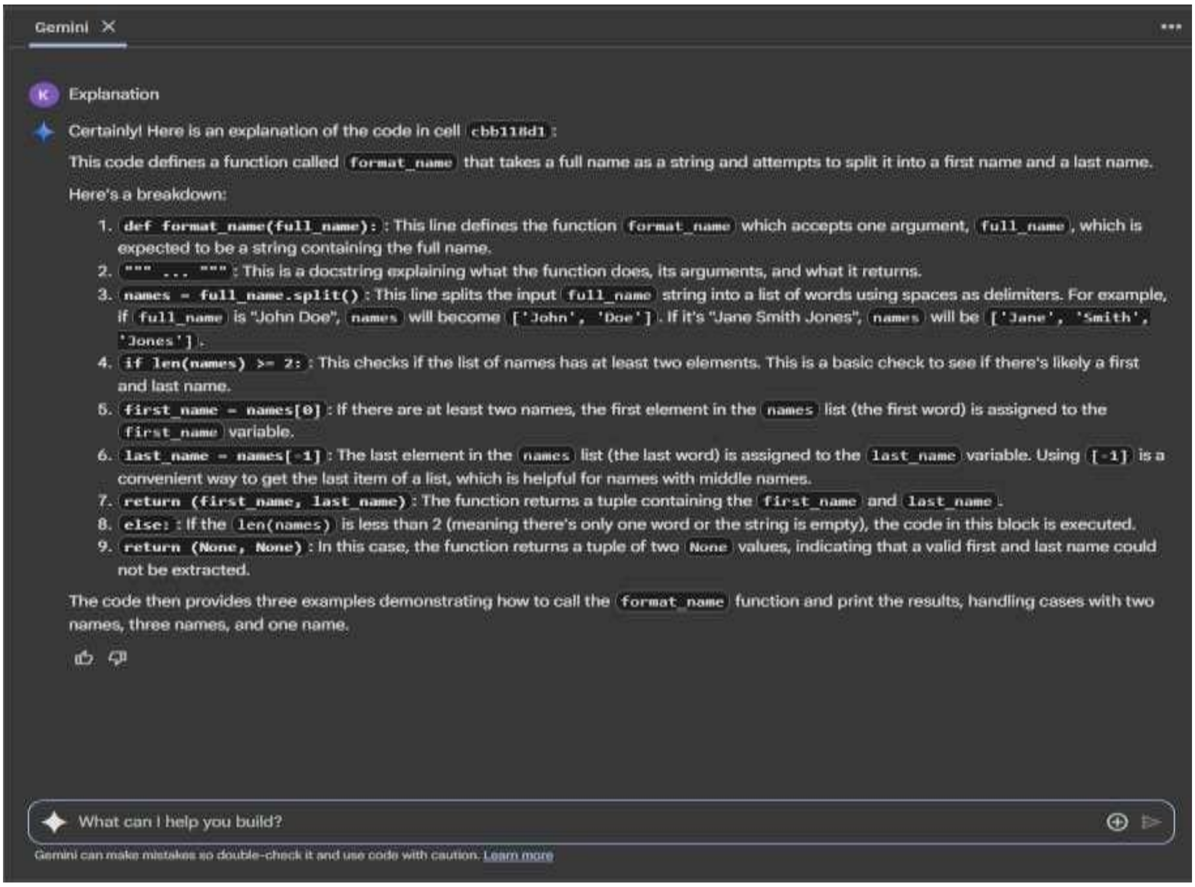
Code:


AI(4).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text ▷ Run all

```python
def format_name(full_name):
  """
  Formats a full name string into first and last names.

  Args:
    full_name: The full name string (e.g., "John Doe").

  Returns:
    A tuple containing the first name and last name (e.g., ("John", "Doe")).
    Returns (None, None) if the input is not a valid full name format.
  """
  names = full_name.split()
  if len(names) >= 2:
    first_name = names[0]
    last_name = names[-1] # Handles cases with middle names
    return (first_name, last_name)
  else:
    return (None, None) # Handle cases with only one name or empty string

# Example usage:
full_name1 = "John Doe"
first1, last1 = format_name(full_name1)
if first1 and last1:
  print(f"Full Name: {full_name1}, First Name: {first1}, Last Name: {last1}")
else:
  print(f"Invalid Name Format: {full_name1}")

full_name2 = "Jane Smith Jones"
first2, last2 = format_name(full_name2)
if first2 and last2:
  print(f"Full Name: {full_name2}, First Name: {first2}, Last Name: {last2}")
else:
  print(f"Invalid Name Format: {full_name2}")

full_name3 = "Peter"
first3, last3 = format_name(full_name3)
if first3 and last3:
  print(f"Full Name: {full_name3}, First Name: {first3}, Last Name: {last3}")
else:
  print(f"Invalid Name Format: {full_name3}")
```

```
Full Name: John Doe, First Name: John, Last Name: Doe
Full Name: Jane Smith Jones, First Name: Jane, Last Name: Jones
Invalid Name Format: Peter
```

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:

1. `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.
2. `""" ... """` : This is a docstring explaining what the function does, its arguments, and what it returns.
3. `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']`.
4. `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.
5. `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.
6. `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.
7. `return (first_name, last_name)` : The function returns a tuple containing the `first_name` and `last_name`.
8. `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.
9. `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.

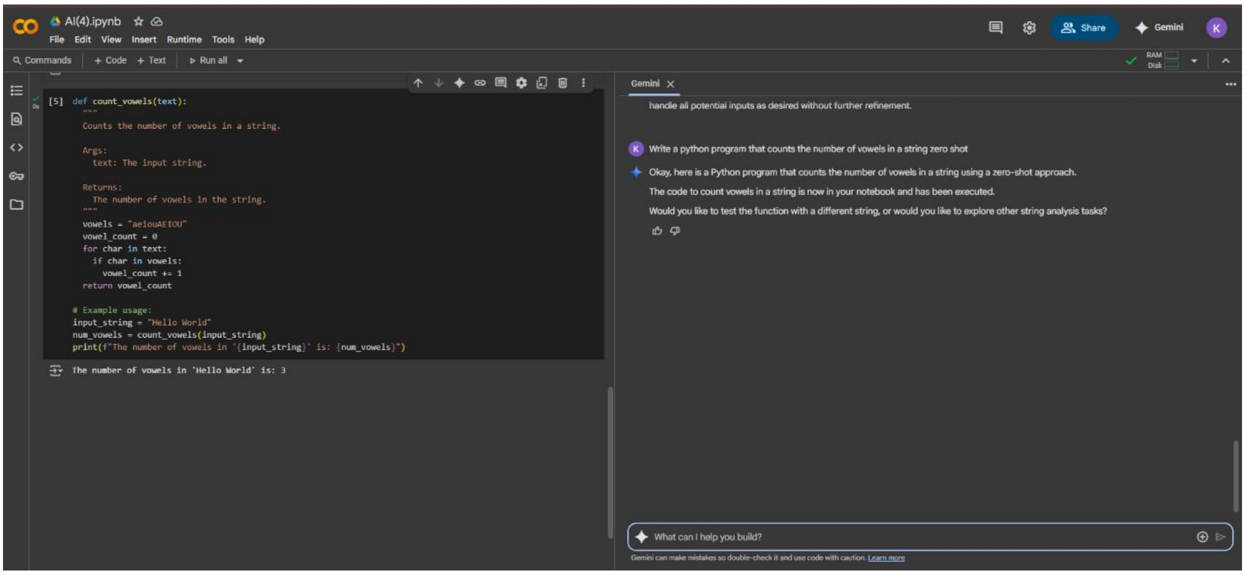
The code then provides three examples demonstrating how to call the `format_name` function and print the results, handling cases with two names, three names, and one name.

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:


AI(4).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Gemini
RAM
Disk
Gemini

```
[5] def count_vowels(text):
      """
      Counts the number of vowels in a string.

      Args:
        text: The input string.

      Returns:
        The number of vowels in the string.
      """
      vowels = "aeiouAEIOU"
      vowel_count = 0
      for char in text:
        if char in vowels:
          vowel_count += 1
      return vowel_count

    # Example usage:
    input_string = "Hello World"
    num_vowels = count_vowels(input_string)
    print(f"The number of vowels in '{input_string}' is: {num_vowels}")
```

```
The number of vowels in 'Hello World' is: 3
```

handle all potential inputs as desired without further refinement.

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

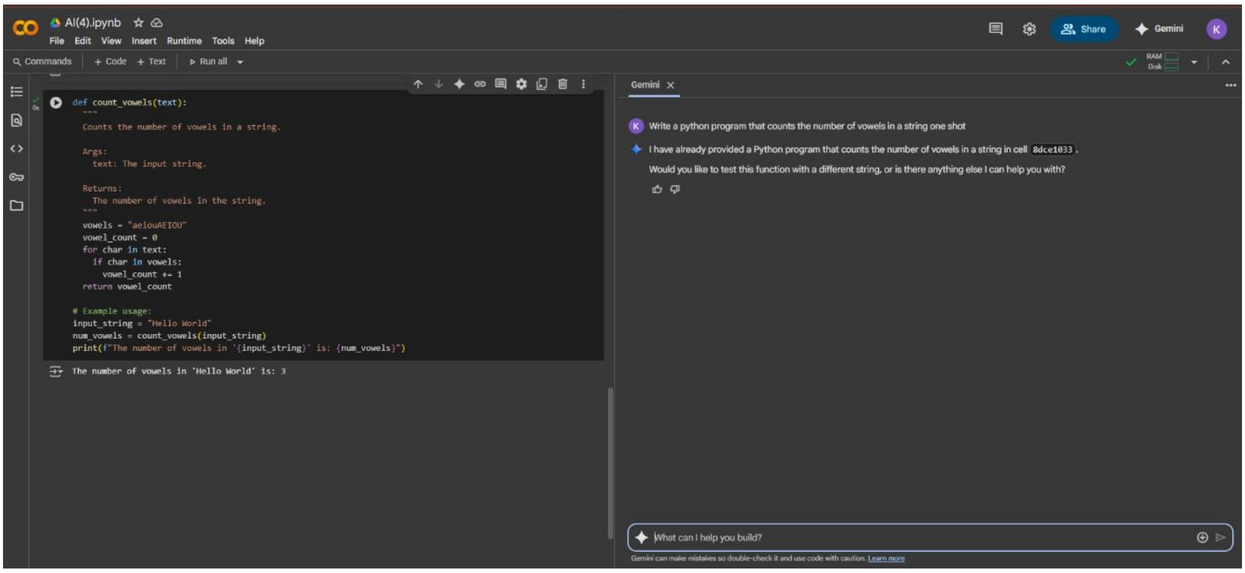
Okay, here is a Python program that counts the number of vowels in a string using a zero-shot approach.

The code to count vowels in a string is now in your notebook and has been executed.

Would you like to test the function with a different string, or would you like to explore other string analysis tasks?

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

One shot code:


AI(4).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Gemini
RAM
Disk
Gemini

```
def count_vowels(text):
  """
  Counts the number of vowels in a string.

  Args:
    text: The input string.

  Returns:
    The number of vowels in the string.
  """
  vowels = "aeiouAEIOU"
  vowel_count = 0
  for char in text:
    if char in vowels:
      vowel_count += 1
  return vowel_count

# Example usage:
input_string = "Hello World"
num_vowels = count_vowels(input_string)
print(f"The number of vowels in '{input_string}' is: {num_vowels}")
```

```
The number of vowels in 'Hello World' is: 3
```

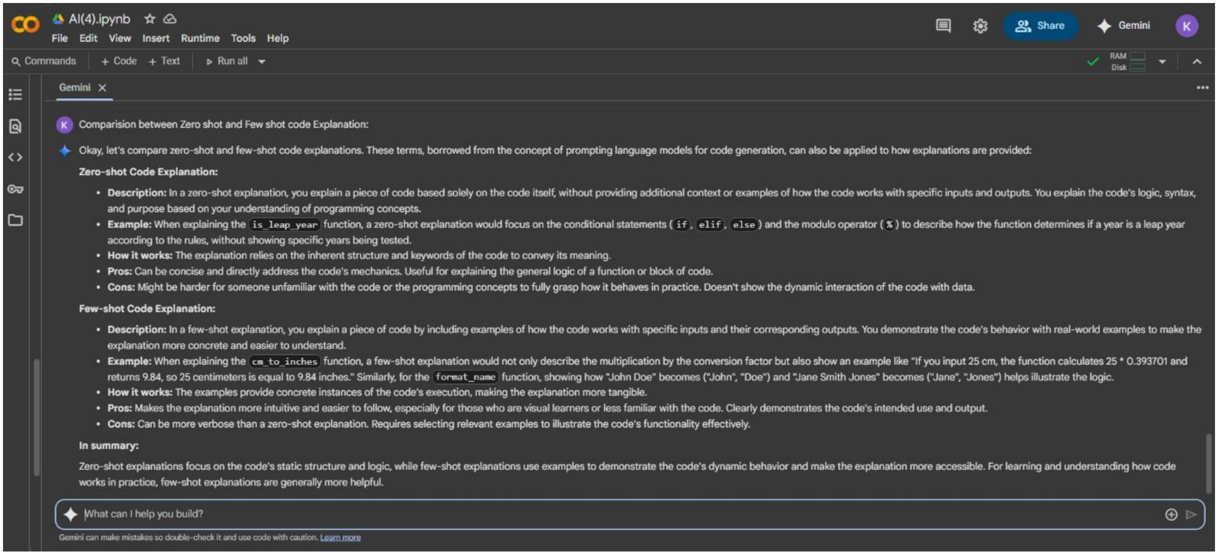
Write a python program that counts the number of vowels in a string one shot

I have already provided a Python program that counts the number of vowels in a string in cell `8dce1033`.

Would you like to test this function with a different string, or is there anything else I can help you with?

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:


AI(4).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Gemini
RAM
Disk
Gemini

Comparision between Zero shot and Few shot code Explanation:

Okay, let's compare zero-shot and few-shot code explanations. These terms, borrowed from the concept of prompting language models for code generation, can also be applied to how explanations are provided:

Zero-shot Code Explanation:

- **Description:** In a zero-shot explanation, you explain a piece of code based solely on the code itself, without providing additional context or examples of how the code works with specific inputs and outputs. You explain the code's logic, syntax, and purpose based on your understanding of programming concepts.
- **Example:** When explaining the `is_leap_year` function, a zero-shot explanation would focus on the conditional statements (`if`, `elif`, `else`) and the modulo operator (`%`) to describe how the function determines if a year is a leap year according to the rules, without showing specific years being tested.
- **How it works:** The explanation relies on the inherent structure and keywords of the code to convey its meaning.
- **Pros:** Can be concise and directly address the code's mechanics. Useful for explaining the general logic of a function or block of code.
- **Cons:** Might be harder for someone unfamiliar with the code or the programming concepts to fully grasp how it behaves in practice. Doesn't show the dynamic interaction of the code with data.

Few-shot Code Explanation:

- **Description:** In a few-shot explanation, you explain a piece of code by including examples of how the code works with specific inputs and their corresponding outputs. You demonstrate the code's behavior with real-world examples to make the explanation more concrete and easier to understand.
- **Example:** When explaining the `cm_to_inches` function, a few-shot explanation would not only describe the multiplication by the conversion factor but also show an example like "If you input 25 cm, the function calculates 25 * 0.393701 and returns 9.84, so 25 centimeters is equal to 9.84 inches." Similarly, for the `format_name` function, showing how "John Doe" becomes ("John", "Doe") and "Jane Smith Jones" becomes ("Jane", "Jones") helps illustrate the logic.
- **How it works:** The examples provide concrete instances of the code's execution, making the explanation more tangible.
- **Pros:** Makes the explanation more intuitive and easier to follow, especially for those who are visual learners or less familiar with the code. Clearly demonstrates the code's intended use and output.
- **Cons:** Can be more verbose than a zero-shot explanation. Requires selecting relevant examples to illustrate the code's functionality effectively.

In summary:

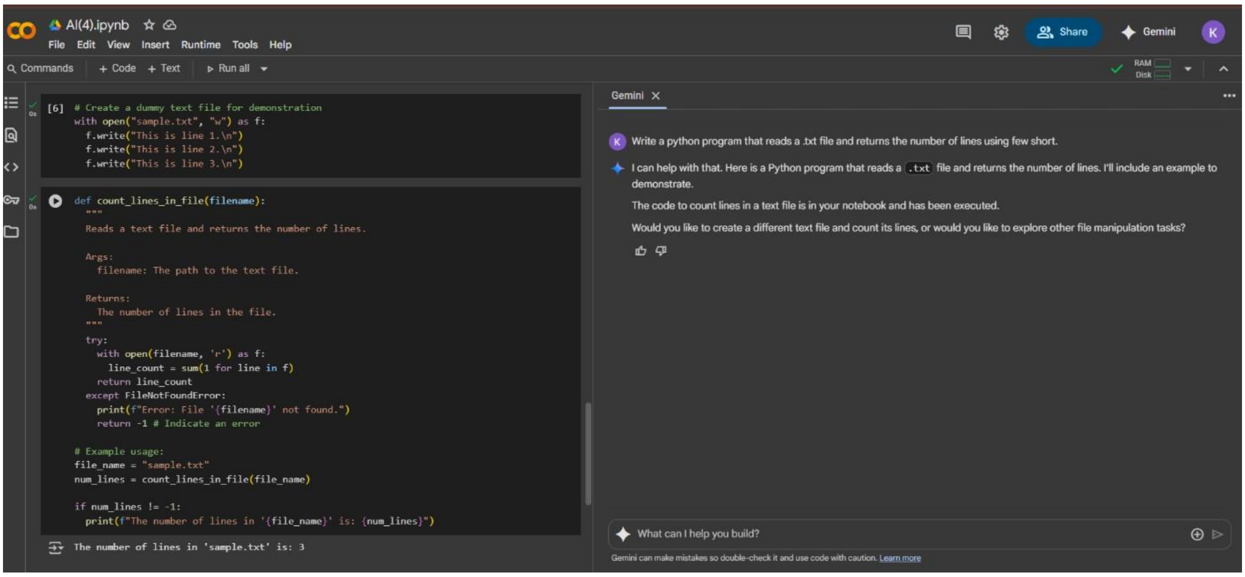
Zero-shot explanations focus on the code's static structure and logic, while few-shot explanations use examples to demonstrate the code's dynamic behavior and make the explanation more accessible. For learning and understanding how code works in practice, few-shot explanations are generally more helpful.

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 5:

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

Code:



```
[6] # Create a dummy text 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
```

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

I can help with that. Here is a Python program that reads a `.txt` file and returns the number of lines. I'll include an example to demonstrate.

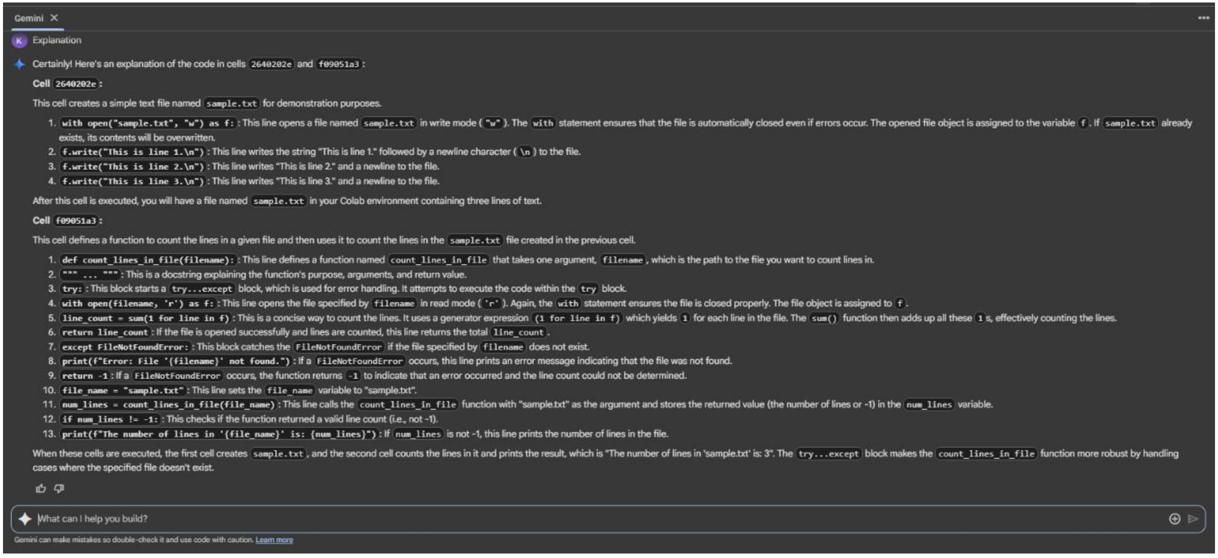
The code to count lines in a text file is in your notebook and has been executed.

Would you like to create a different text file and count its lines, or would you like to explore other file manipulation tasks?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



Explanation

Certainly! Here's an explanation of the code in cells `2640202e` and `f09051a3`:

Cell `2640202e`:

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 errors occur. 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 Colab environment containing three lines of text.

Cell `f09051a3`:

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.

1. `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.
2. `""" ... """` : This is a docstring explaining the function's purpose, arguments, and return value.
3. `try:` : This block starts a `try...except` block, which is used for error handling. It attempts to execute the code within the `try` block.
4. `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`.
5. `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 `1` s, effectively counting the lines.
6. `return line_count` : If the file is opened successfully and lines are counted, this line returns the total `line_count`.
7. `except FileNotFoundError:` : This block catches the `FileNotFoundError` if the file specified by `filename` does not exist.
8. `print(f"Error: File '{filename}' not found.")` : If a `FileNotFoundError` occurs, this line prints an error message indicating that the file was not found.
9. `return -1` : If a `FileNotFoundError` occurs, the function returns `-1` to indicate that an error occurred and the line count could not be determined.
10. `file_name = "sample.txt"` : This line sets the `file_name` variable to "sample.txt".
11. `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.
12. `if num_lines != -1:` : This checks if the function returned a valid line count (i.e., not -1).
13. `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.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more