

AI Assistant Coding

Assignment-2.5

Name: Ashwitha Ravula

HT.NO.: 2303A51340

Batch:20

Lab 2: Exploring Additional AI Coding Tools beyond Copilot - Gemini (Colab) and Cursor AI

Task 1: Refactoring Odd/Even Logic (List Version)

Scenario

You are improving legacy code that calculates the sum of odd and even numbers in a list.

Prompt Used (Gemini / Cursor AI):

"Write a Python program to calculate the sum of odd and even numbers in a list, then refactor it to improve readability and efficiency."

Original Code (Legacy Version)

```
numbers = [10, 15, 22, 33, 40]
odd_sum = 0
even_sum = 0
for i in range(len(numbers)):
    if numbers[i] % 2 == 0:
        even_sum = even_sum + numbers[i]
    else:
        odd_sum = odd_sum + numbers[i]
```

```
print("Odd Sum:", odd_sum)
```

```
print("Even Sum:", even_sum)
```

Refactored Code (AI-Improved Version)

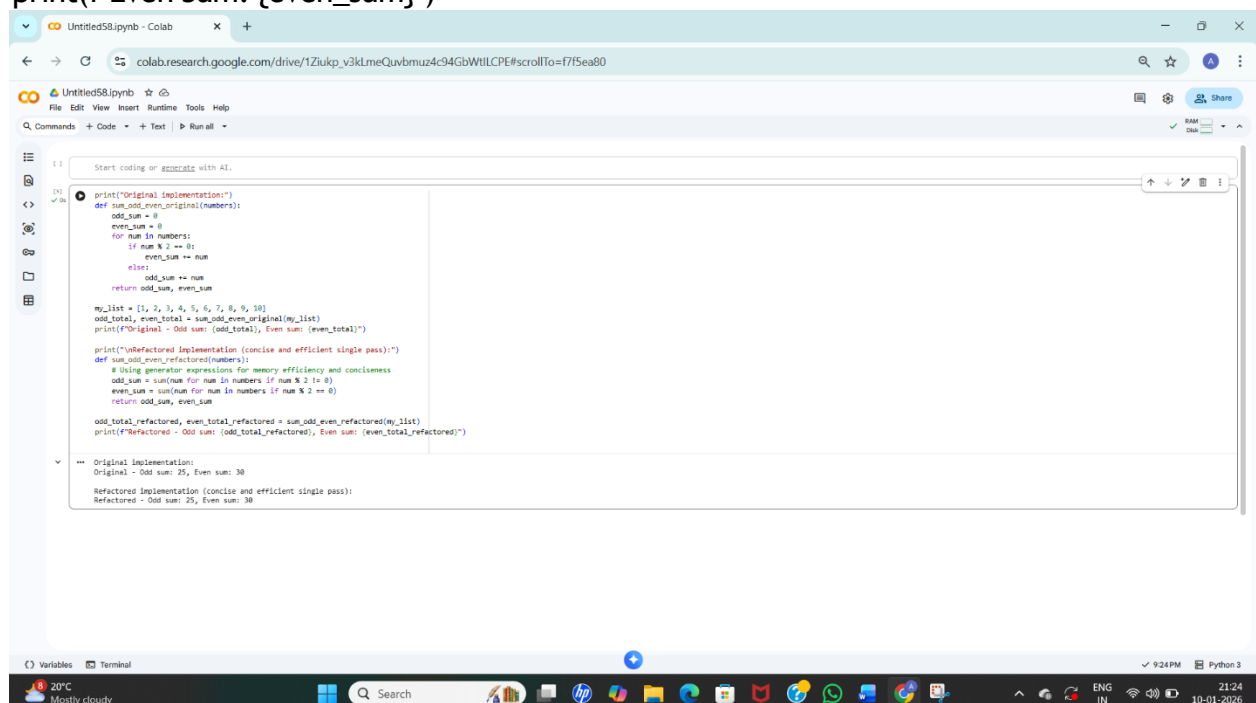
```
numbers = [10, 15, 22, 33, 40]
```

```
odd_sum = sum(num for num in numbers if num % 2 != 0)
```

```
even_sum = sum(num for num in numbers if num % 2 == 0)
```

```
print(f"Odd Sum: {odd_sum}")
```

```
print(f"Even Sum: {even_sum}")
```



The screenshot shows a Google Colab notebook titled 'Untitled58.ipynb'. The code is as follows:

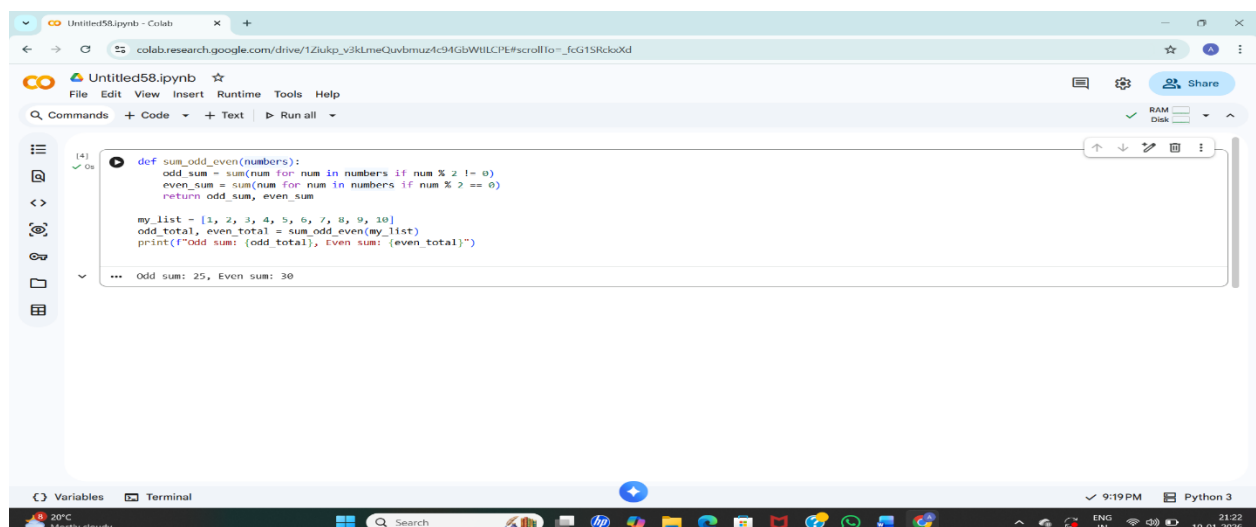
```
def sum_odd_even_original(numbers):
    odd_sum = 0
    even_sum = 0
    for num in numbers:
        if num % 2 != 0:
            odd_sum += num
        else:
            even_sum += num
    return odd_sum, even_sum

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_total, even_total = sum_odd_even_original(my_list)
print(f"Original - Odd sum: {odd_total}, Even sum: {even_total}")

print("Unrefactored Implementation (concise and efficient single pass):")
def sum_odd_even_refactored(numbers):
    # Using generator expressions for memory efficiency and conciseness
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    even_sum = sum(num for num in numbers if num % 2 == 0)
    return odd_sum, even_sum

odd_total_refactored, even_total_refactored = sum_odd_even_refactored(my_list)
print(f"Refactored - Odd sum: {odd_total_refactored}, Even sum: {even_total_refactored}")
```

The output shows the original implementation results: Original - Odd sum: 25, Even sum: 30. The refactored implementation results are also shown: Refactored - Odd sum: 25, Even sum: 30.



The screenshot shows a Google Colab notebook titled 'Untitled58.ipynb'. The code is as follows:

```
def sum_odd_even(numbers):
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    even_sum = sum(num for num in numbers if num % 2 == 0)
    return odd_sum, even_sum

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_total, even_total = sum_odd_even(my_list)
print(f"Odd sum: {odd_total}, Even sum: {even_total}")
```

The output shows the results of the refactored code: Odd sum: 25, Even sum: 30.

Explanation:

The refactored version improves readability by using Python's built-in `sum()` function and list comprehensions. It reduces the number of lines and avoids manual looping, making the code cleaner and more efficient.

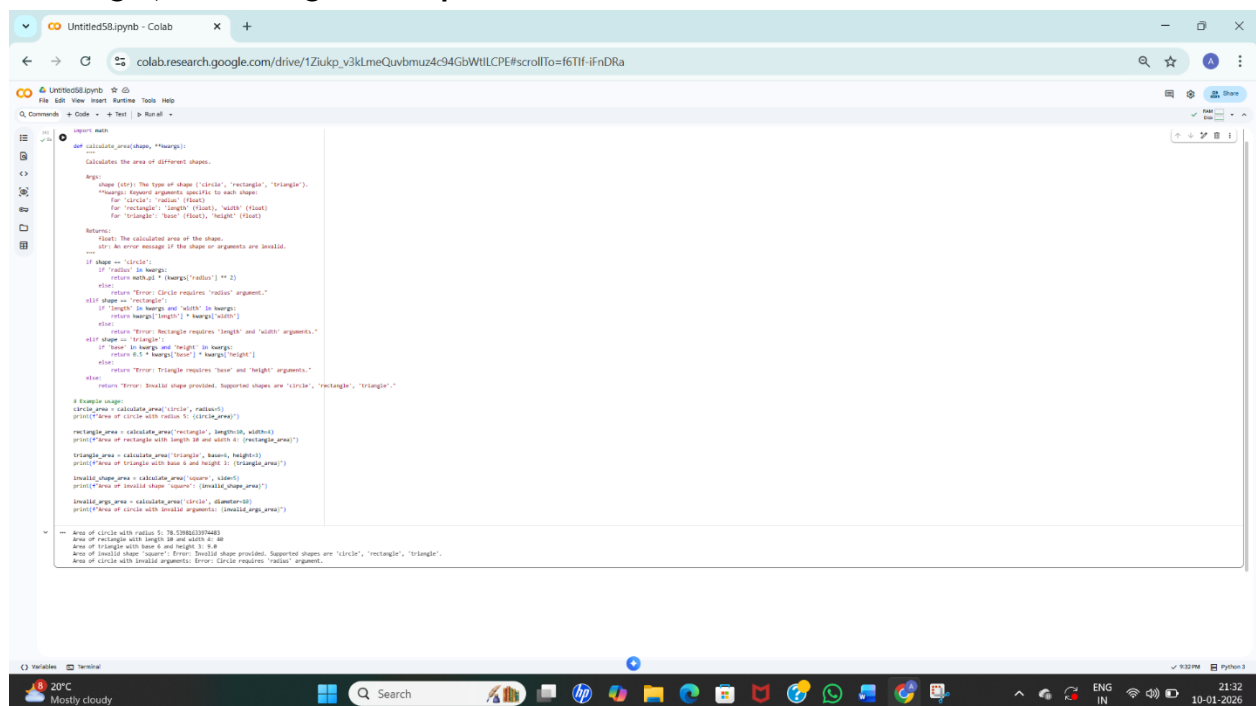
Task 2: Area Calculation Explanation

Scenario

You are onboarding a junior developer and want a clear explanation of a function that calculates the area of different shapes.

Prompt Used (Google Gemini):

"Explain a Python function that calculates the area of different shapes like circle, rectangle, and triangle in simple terms."



The screenshot shows a Google Colab notebook titled 'Untitled58.ipynb'. The code defines a function `calculate_area(shape, *kwargs)` that calculates the area of different shapes. The function uses a dictionary to map shapes to their respective area calculation functions. It includes error handling for invalid shapes and arguments. The output shows the area of a circle with radius 5, a rectangle with length 10 and width 4, a triangle with base 6 and height 3, and an invalid shape 'square'.

```
def calculate_area(shape, *kwargs):
    """
    Calculates the area of different shapes.

    Args:
        shape (str): The type of shape ('circle', 'rectangle', 'triangle').
        **kwargs: Keyword arguments specific to each shape:
            - For 'circle': 'radius' (float)
            - For 'rectangle': 'length' (float), 'width' (float)
            - For 'triangle': 'base' (float), 'height' (float)

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape or arguments are invalid.
    """
    if shape == 'circle':
        if 'radius' in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: Circle requires 'radius' argument."
    elif shape == 'rectangle':
        if 'length' in kwargs and 'width' in kwargs:
            return kwargs['length'] * kwargs['width']
        else:
            return "Error: Rectangle requires 'length' and 'width' arguments."
    elif shape == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
            return 0.5 * kwargs['base'] * kwargs['height']
        else:
            return "Error: Triangle requires 'base' and 'height' arguments."
    else:
        return "Error: Invalid shape provided. Supported shapes are 'circle', 'rectangle', 'triangle'."

# Example usage:
circle_area = calculate_area('circle', radius=5)
print(f"Area of circle with radius 5: {circle_area}")

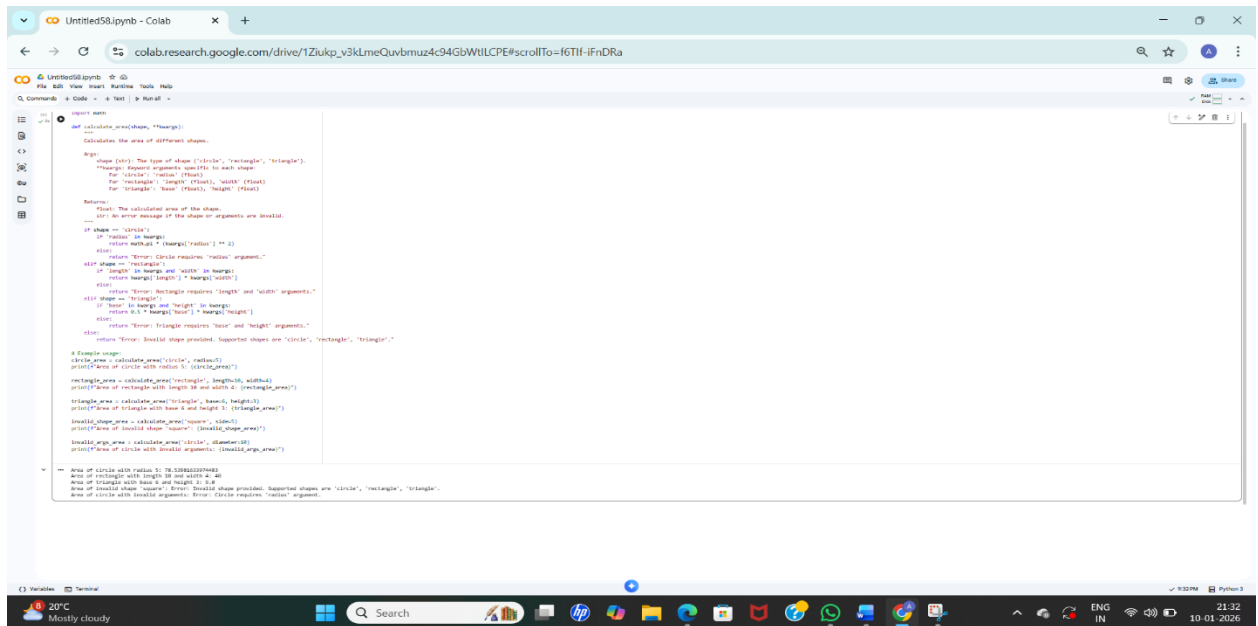
rectangle_area = calculate_area('rectangle', length=10, width=4)
print(f"Area of rectangle with length 10 and width 4: {rectangle_area}")

triangle_area = calculate_area('triangle', base=6, height=3)
print(f"Area of triangle with base 6 and height 3: {triangle_area}")

invalid_shape_area = calculate_area('square', side=5)
print(f"Area of invalid shape 'square': {invalid_shape_area}")

invalid_arg_area = calculate_area('circle', diameter=10)
print(f"Area of circle with invalid arguments: {invalid_arg_area}")
```

Area of circle with radius 5: 78.53981633974483
Area of rectangle with length 10 and width 4: 40
Area of triangle with base 6 and height 3: 9.0
Area of invalid shape 'square': Error: Invalid shape provided. Supported shapes are 'circle', 'rectangle', 'triangle'.
Area of circle with invalid arguments: Error: Circle requires 'radius' argument.



Code

```

import math

def calculate_area(shape, value1, value2=None):

if shape == "circle":

return math.pi * value1 * value1

elif shape == "rectangle":

return value1 * value2

elif shape == "triangle":

return 0.5 * value1 * value2

else:

return "Invalid shape"

```

Explanation

This function calculates the area based on the shape name passed as input. For a circle, it uses the formula πr^2 where value1 is the radius. For a rectangle, it multiplies length and breadth. For a triangle, it applies the formula $\frac{1}{2} \times \text{base} \times \text{height}$. The function checks the shape type using conditional statements and returns the appropriate area.

Task 3: Prompt Sensitivity Experiment

Scenario

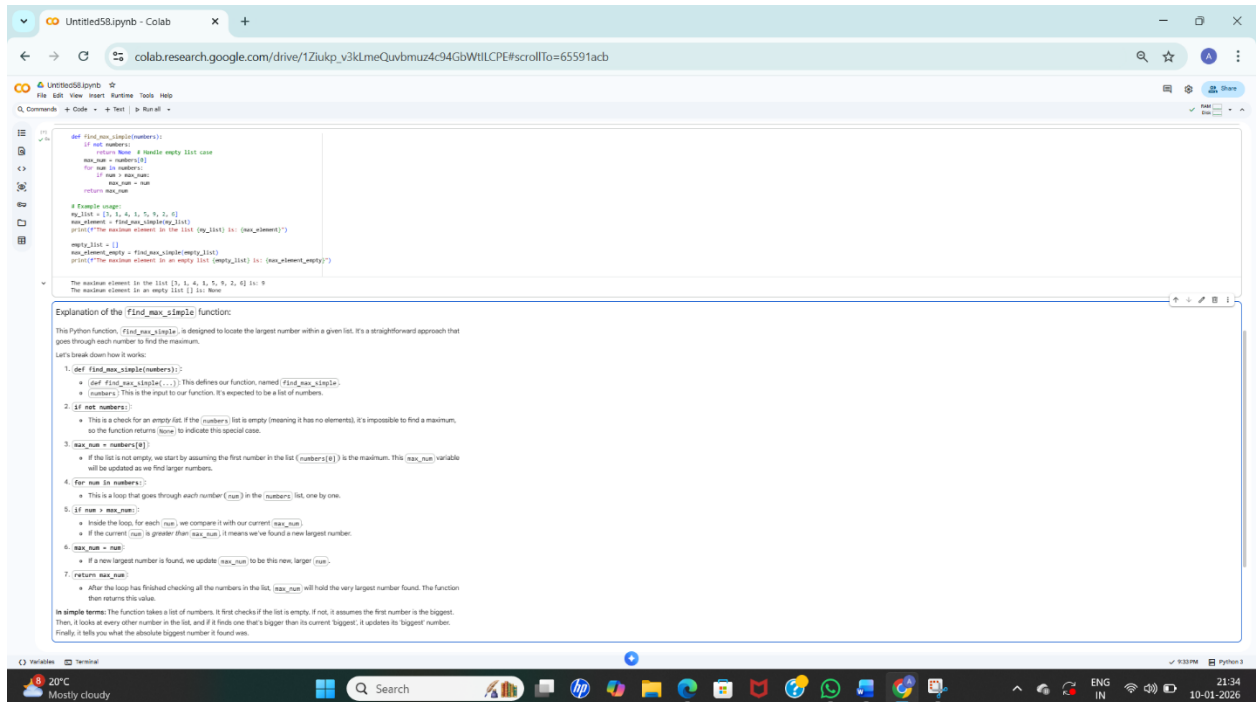
You are testing how Cursor AI responds to different prompts for the same problem.

Problem Statement

"Write a Python function to find the maximum number in a lists

Prompt 1

"Write a simple Python function to find the maximum element in a list."



The screenshot shows a Google Colab notebook titled "Untitled58.ipynb". The code defines a function `find_max_simple(numbers)` that finds the maximum element in a list. It includes a docstring, a type hint, and a return statement. The function is tested with a list of numbers and an empty list. Below the code, there is an "Explanation of the find_max_simple function:" section that breaks down the code into steps: 1. Define the function, 2. Check for an empty list, 3. Initialize the maximum value, 4. Iterate through the list, 5. Compare each element with the current maximum, 6. Update the maximum if a larger element is found, and 7. Return the maximum value.

```
def find_max_simple(numbers):  
    """  
    Find the maximum element in a list.  
    """  
    if not numbers:  
        return None  # Handle empty list case  
    max_num = numbers[0]  
    for num in numbers:  
        if num > max_num:  
            max_num = num  
    return max_num  
  
# Example usage:  
nums_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
max_element = find_max_simple(nums_list)  
print(f"The maximum element in the list {nums_list} is: {max_element}")  
  
empty_list = []  
max_element_empty = find_max_simple(empty_list)  
print(f"The maximum element in an empty list {empty_list} is: {max_element_empty}")
```

The maximum element in the list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] is: 10
The maximum element in an empty list [] is: None

Explanation of the find_max_simple function:

This Python function, `find_max_simple`, is designed to locate the largest number within a given list. It's a straightforward approach that goes through each number to find the maximum.

Let's break down how it works:

- 1. def find_max_simple(numbers):**
 - `def find_max_simple(...)`: This defines our function, named `find_max_simple`.
 - `numbers`: This is the input to our function. It's expected to be a list of numbers.
- 2. if not numbers:**
 - This is a check for an empty list. If the `numbers` list is empty (meaning it has no elements), it's impossible to find a maximum, so the function returns `None` to indicate this special case.
- 3. max_num = numbers[0]**
 - If the list is not empty, we start by assuming the first number in the list (`numbers[0]`) is the maximum. This `max_num` variable will be updated as we find larger numbers.
- 4. for num in numbers:**
 - This is a loop that goes through each number (`num`) in the `numbers` list, one by one.
- 5. if num > max_num:**
 - Inside the loop, for each `num`, we compare it with our current `max_num`.
 - If the current `num` is greater than `max_num`, it means we've found a new largest number.
- 6. max_num = num**
 - If a new largest number is found, we update `max_num` to be this new, larger `num`.
- 7. return max_num**
 - After the loop has finished checking all the numbers in the list, `max_num` will hold the very largest number found. The function then returns this value.

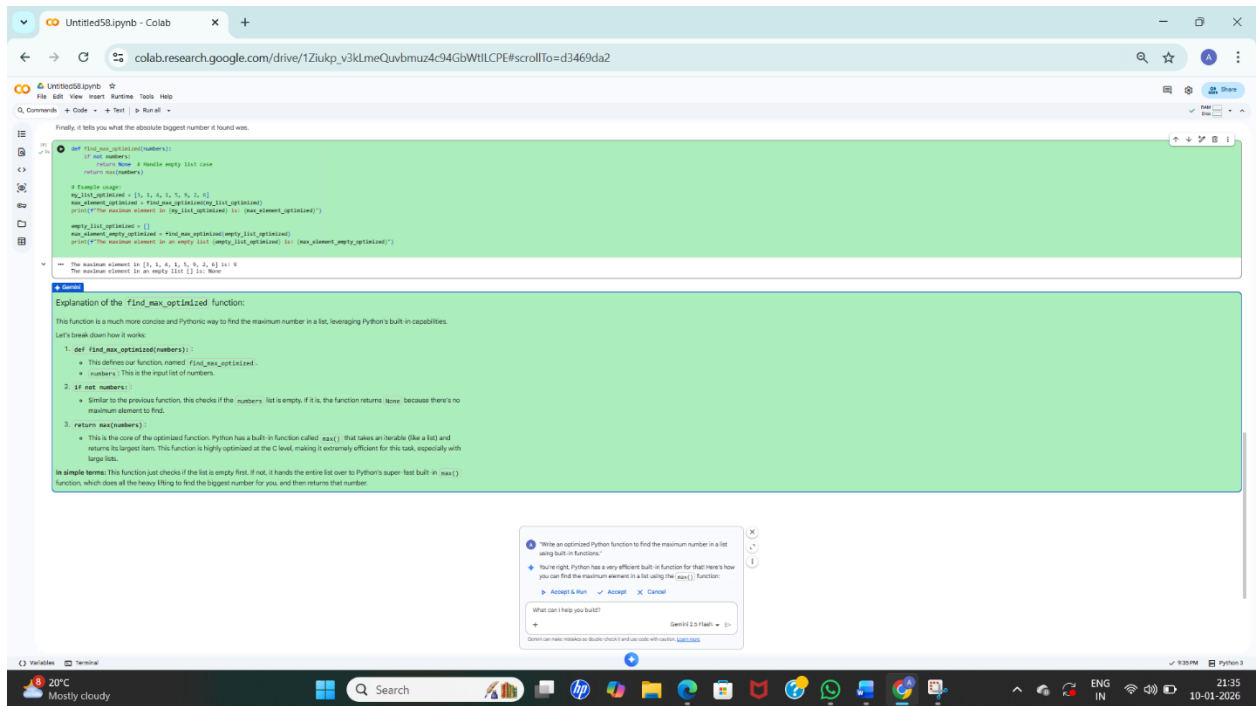
In simple terms: The function takes a list of numbers. It first checks if the list is empty. If not, it assumes the first number is the biggest. Then, it looks at every other number in the list, and if it finds one that's bigger than its current 'biggest', it updates its 'biggest' number. Finally, it tells you what the absolute biggest number it found was.

Code:

```
def find_max(nums):  
    max_value = nums[0]  
    for num in nums:  
        if num > max_value:  
            max_value = num  
    return max_value
```

Prompt 2

"Write an optimized Python function to find the maximum number in a list using built-in functions."

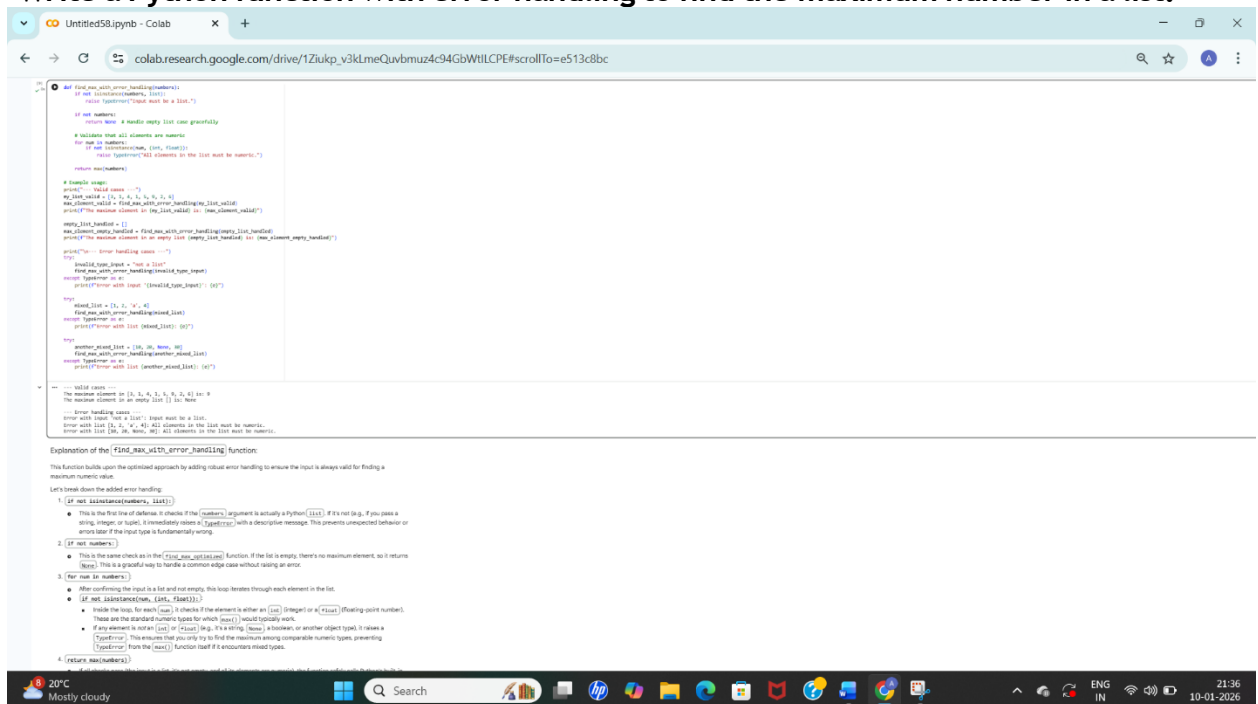


Output Code

```
def find_max(nums): return max(nums)
```

Prompt 3

"Write a Python function with error handling to find the maximum number in a list."



Output Code

```
def find_max_robust(numbers):
```

```
    if not isinstance(numbers, list):
```

```
        raise TypeError("Input must be a list.")
```

```
    if not numbers:
```

```
        return None # Return None for an empty list
```

```

try:

    return max(numbers)
except TypeError as e:
    if "not supported between instances of" in str(e) or "unorderable types" in str(e):
        raise TypeError("List elements must be numbers for comparison.")
    else:
        raise e

# --- Example Usage ---
print("--- Valid Cases ---")
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
print(f"The list: {my_list}\nMaximum element: {find_max_robust(my_list)}\n")

another_list = [-10, -5.5, -2, -20.1]
print(f"The list: {another_list}\nMaximum element: {find_max_robust(another_list)}\n")
empty_list = []
print(f"The list: {empty_list}\nMaximum element: {find_max_robust(empty_list)}\n")
single_element_list = [42]
print(f"The list: {single_element_list}\nMaximum element: {find_max_robust(single_element_list)}\n")

print("--- Error Handling Cases ---")
# Case 1: Input is not a list
try:
    find_max_robust("this is not a list")
except TypeError as e:
    print(f"Error caught (not a list): {e}\n")

# Case 2: List contains non-numeric elements
try:

```

```
find_max_robust([1, 2, 'a', 4])  
  
except TypeError as e:  
    print(f"Error caught (non-numeric element): {e}\n")  
  
try:  
    find_max_robust(['x', 2, 3, 4])  
  
except TypeError as e:  
    print(f"Error caught (non-numeric first element): {e}\n")
```

Observation

Cursor AI clearly demonstrates prompt sensitivity, meaning the quality, structure, and safety of the generated code change based on how the prompt is written.

- In Prompt 1, the instruction is very basic, so Cursor AI produces a traditional loop-based solution. This version is easy to understand for beginners and shows the step-by-step logic of finding a maximum value manually.
- In Prompt 2, the word *optimized* guides Cursor AI to use Python's built-in `max()` function. This results in shorter, cleaner, and more efficient code with better readability and performance.
- In Prompt 3, the phrase *with error handling* makes Cursor AI consider edge cases. The generated code checks whether the list is empty before applying `max()`, preventing runtime errors and improving reliability.

Task 4: Tool Comparison Reflection

Reflection

Google Gemini is best for learning and understanding code because it explains concepts clearly in simple language. It is very helpful for students and beginners who want to understand logic step by step. GitHub Copilot is useful for fast coding and productivity, especially for experienced developers working on real projects. However, it focuses more on code completion than explanation. Cursor AI is strong in refactoring and experimenting with different prompts, helping improve code quality, readability, and error handling. Each tool serves a different purpose based on learning, speed, or code improvement needs, making them suitable for different stages of software development.