AI Assistant Coding

Assignment-2.5

Name: **A.Anjali**

HT.NO.: **2303A51319**

Batch:**19**

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."

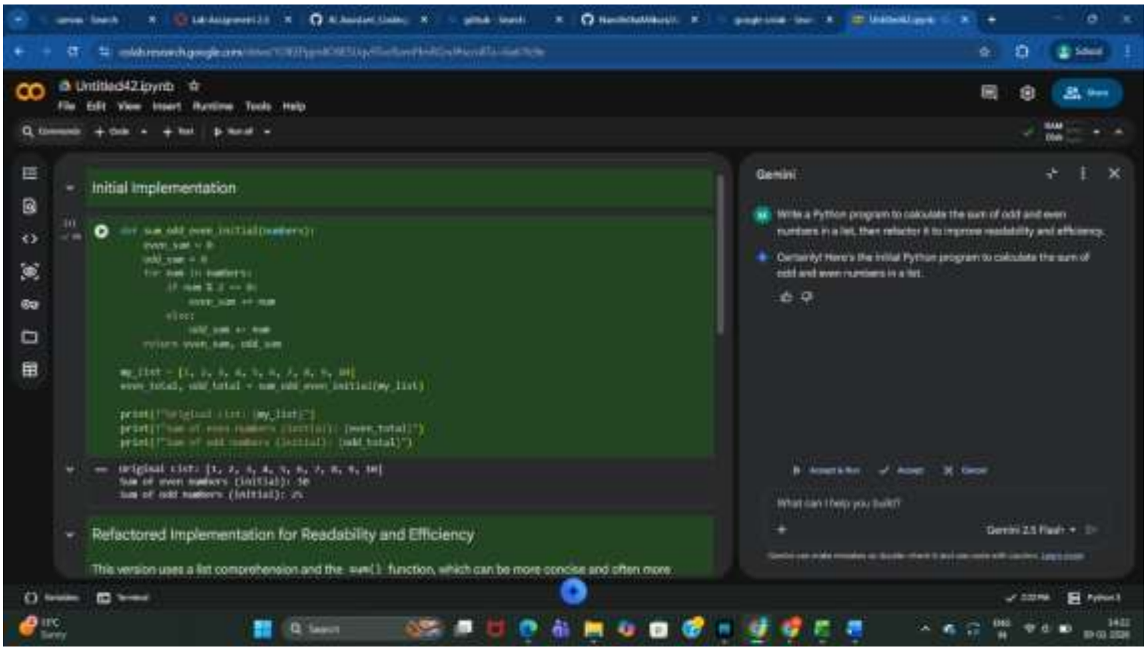
Original Code (Legacy Version)

```
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}")
```



Initial Implementation

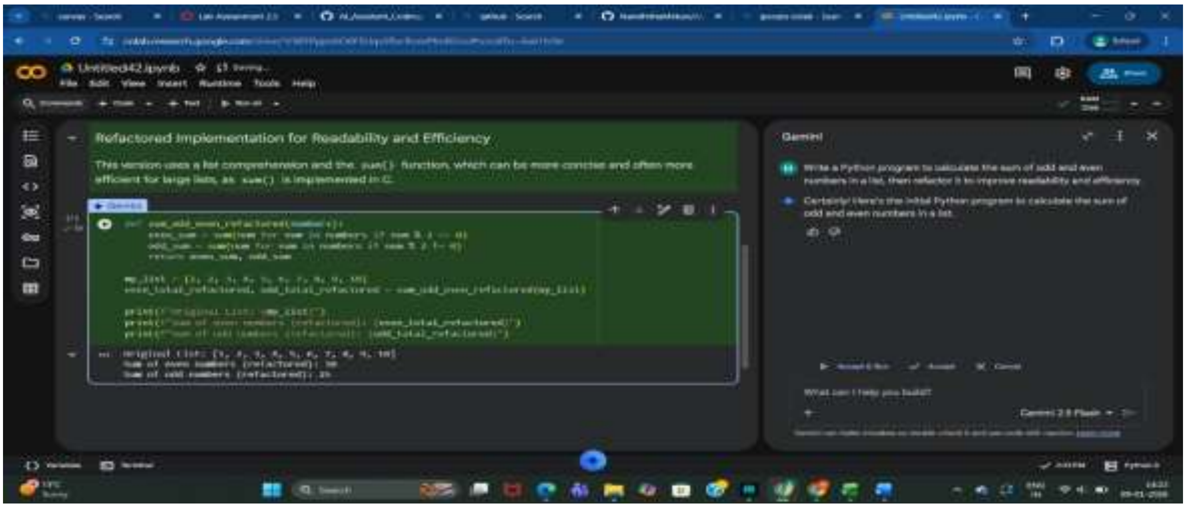
Refactored Implementation for Readability and Efficiency

This version uses a list comprehension and the sum() function, which can be more concise and often more

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

Certainly! Here's the initial Python program to calculate the sum of odd and even numbers in a list.

What can I help you build?
Gemini 2.5 Flash



Untitled42.ipynb
File Edit View Insert Runtime Tools Help
Gemini

Refactored Implementation for Readability and Efficiency

This version uses a list comprehension and the sum() function, which can be more concise and often more efficient for large lists, as sum() is implemented in C.

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

Certainly! Here's the initial Python program to calculate the sum of odd and even numbers in a list.

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.

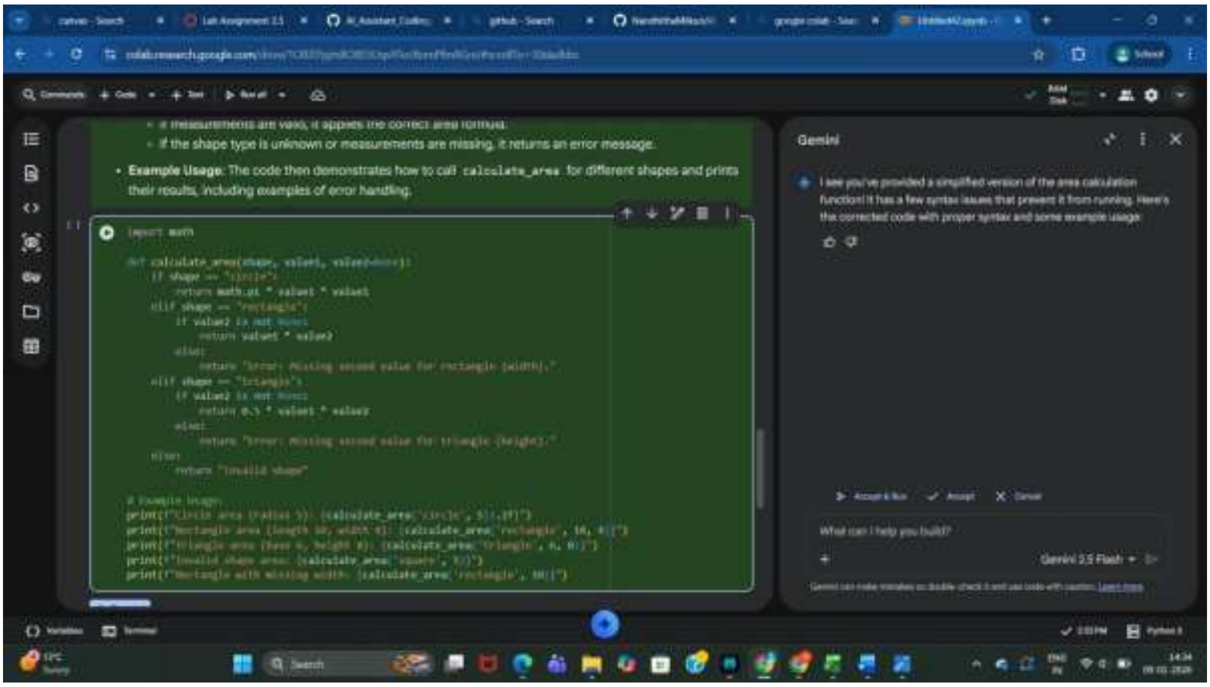
Task 2: Area Calculation Explanation

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."



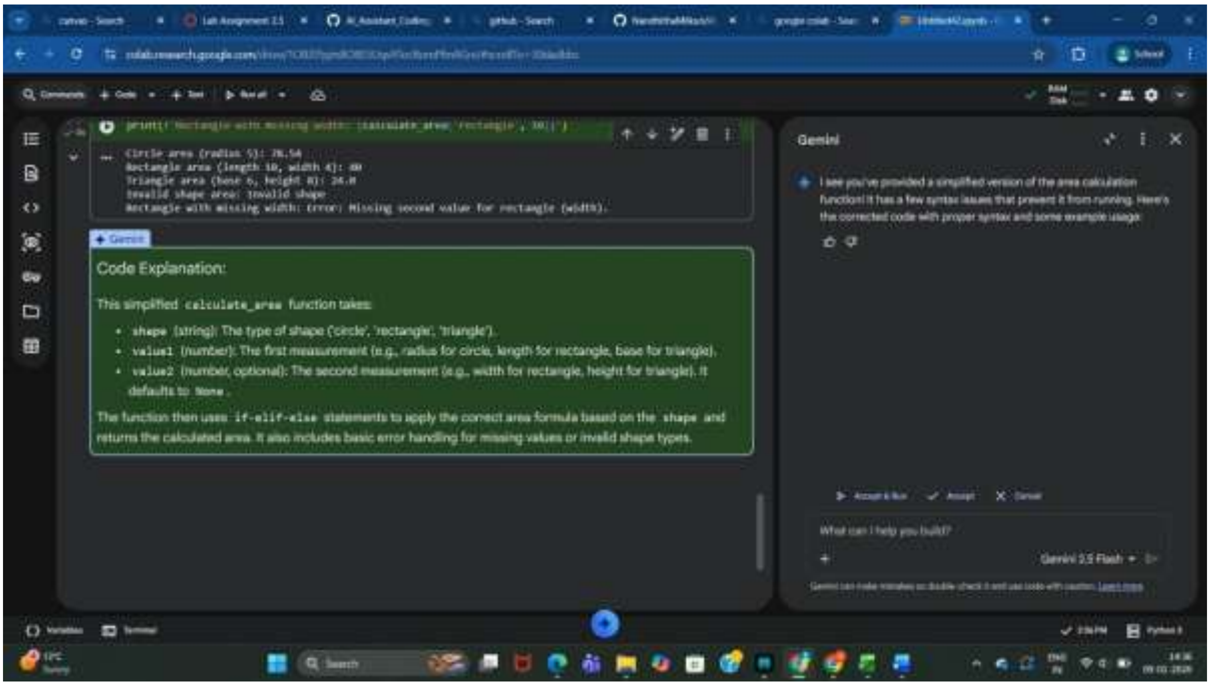
- If measurements are valid, it applies the correct area formula.
- If the shape type is unknown or measurements are missing, it returns an error message.

- **Example Usage**: The code then demonstrates how to call `calculate_area` for different shapes and prints their results, including examples of error handling.

```
import math

def calculate_area(shape, value1, value2=None):
    if shape == "circle":
        return math.pi * value1 * value1
    elif shape == "rectangle":
        if value2 is not None:
            return value1 * value2
        else:
            return "Error: Missing second value for rectangle (width)."
    elif shape == "triangle":
        if value2 is not None:
            return 0.5 * value1 * value2
        else:
            return "Error: Missing second value for triangle (height)."
    else:
        return "Invalid shape"

# Example Usage:
```

Gemini

I see you've provided a simplified version of the area calculation function! It has a few syntax issues that prevent it from running. Here's the corrected code with proper syntax and some example usage:



```
Circle area (radius 5): 78.54
Rectangle area (length 10, width 4): 40
Triangle area (base 6, height 8): 24.0
Invalid shape area: Invalid shape
Rectangle with missing width: Error: Missing second value for rectangle (width).
```

Code Explanation:

This simplified `calculate_area` function takes:

- `shape` (string): The type of shape ('circle', 'rectangle', 'triangle').
- `value1` (number): The first measurement (e.g., radius for circle, length for rectangle, base for triangle).
- `value2` (number, optional): The second measurement (e.g., width for rectangle, height for triangle). It defaults to `None`.

The function then uses `if-elif-else` statements to apply the correct area formula based on the `shape` and returns the calculated area. It also includes basic error handling for missing values or invalid shape types.

Gemini

I see you've provided a simplified version of the area calculation function! It has a few syntax issues that prevent it from running. Here's the corrected code with proper syntax and some example usage:

What can I help you build?

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 πr² where value1 is the radius. For a rectangle, it multiplies length and breadth. For a triangle, it applies the formula ½ × base × height. The function checks the shape type using conditional statements and returns the appropriate area.

Task 3: Prompt Sensitivity Experiment

Scenario

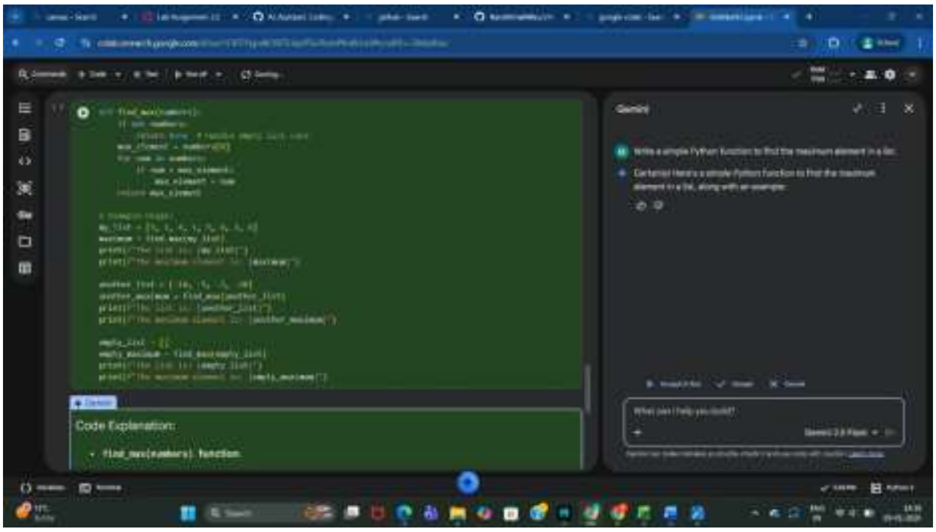
You are testing how Cursor AI responds to different prompts for the same problem.

Problem Statement

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

Prompt 1

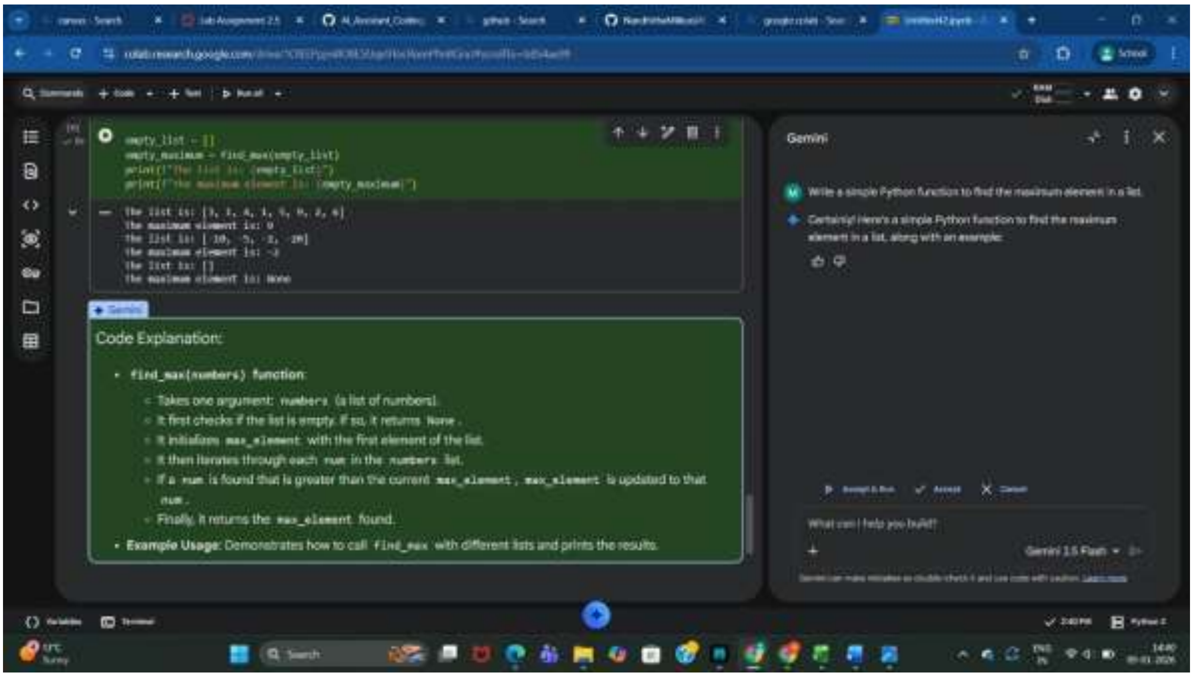
"Write a simple Python function to find the maximum element in a list."



```python
def find_max(numbers):
    if not numbers:
        return None  # Handle empty list case
    max_element = numbers[0]
    for num in numbers:
        if num > max_element:
            max_element = num
    return max_element

# Example Usage:
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
maximum = find_max(my_list)
print(f"The list is: {my_list}")
print(f"The maximum element is: {maximum}")

another_list = [-10, -5, -2, -20]
another_maximum = find_max(another_list)
print(f"The list is: {another_list}")
print(f"The maximum element is: {another_maximum}")

empty_list = []
empty_maximum = find_max(empty_list)
print(f"The list is: {empty_list}")
print(f"The maximum element is: {empty_maximum}")
```

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

Certainly! Here's a simple Python function to find the maximum element in a list, along with an example:



```python
empty_list = []
empty_maximum = find_max(empty_list)
print(f"The list is: {empty_list}")
print(f"The maximum element is: {empty_maximum}")
```

```
The list is: [3, 1, 4, 1, 5, 9, 2, 6]
The maximum element is: 9
The list is: [-10, -5, -2, -20]
The maximum element is: -2
The list is: []
The maximum element is: None
```

Code Explanation:

- `find_max(numbers)` function:
  - Takes one argument: `numbers` (a list of numbers).
  - It first checks if the list is empty. If so, it returns `None`.
  - It initializes `max_element` with the first element of the list.
  - It then iterates through each `num` in the `numbers` list.
  - If a `num` is found that is greater than the current `max_element`, `max_element` is updated to that `num`.
  - Finally, it returns the `max_element` found.
- **Example Usage:** Demonstrates how to call `find_max` with different lists and prints the results.

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

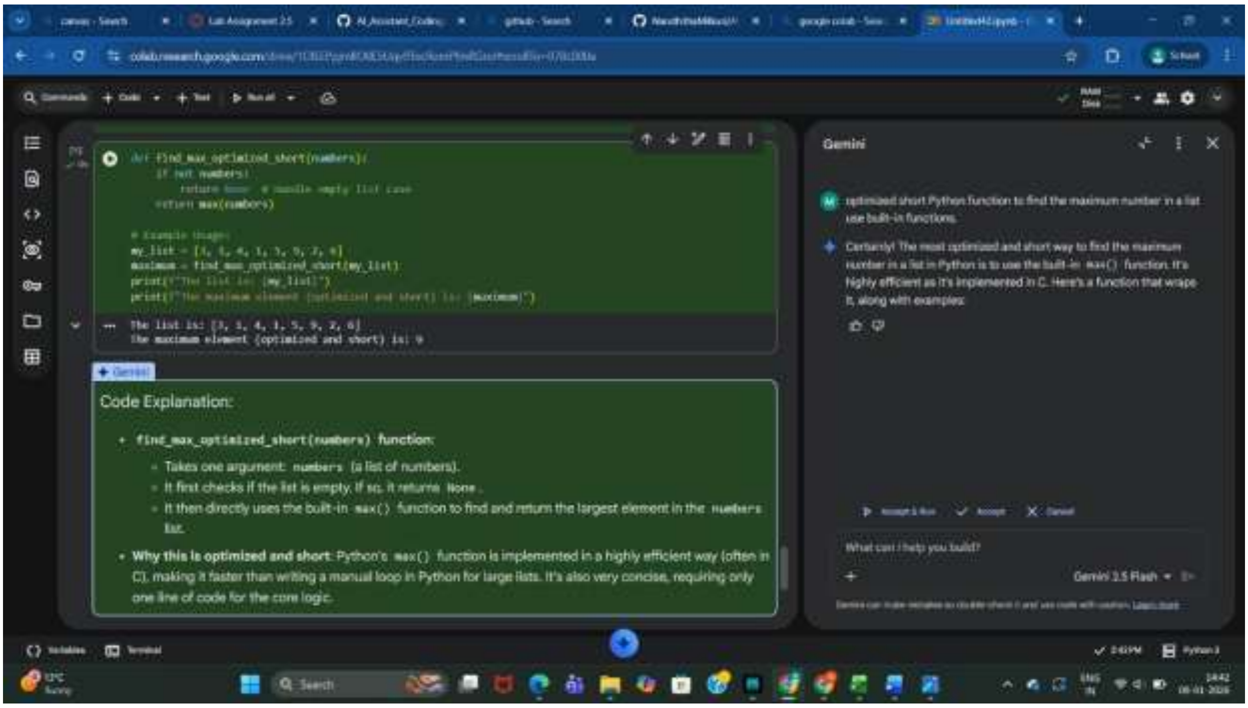
Certainly! Here's a simple Python function to find the maximum element in a list, along with an example:

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."



```
def find_max_optimized_short(numbers):
    if not numbers:
        return None  # Handle empty list case
    return max(numbers)

# Example usage:
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
maximum = find_max_optimized_short(my_list)
print(f"The list is: {my_list}")
print(f"The maximum element (optimized and short) is: {maximum}")
```

```
The list is: [3, 1, 4, 1, 5, 9, 2, 6]
The maximum element (optimized and short) is: 9
```

Code Explanation:

- `find_max_optimized_short(numbers)` function:
    - Takes one argument: `numbers` (a list of numbers).
    - It first checks if the list is empty. If so, it returns `None`.
    - It then directly uses the built-in `max()` function to find and return the largest element in the `numbers` list.
- **Why this is optimized and short**: Python's `max()` function is implemented in a highly efficient way (often in C), making it faster than writing a manual loop in Python for large lists. It's also very concise, requiring only one line of code for the core logic.

optimized short Python function to find the maximum number in a list use built-in functions.

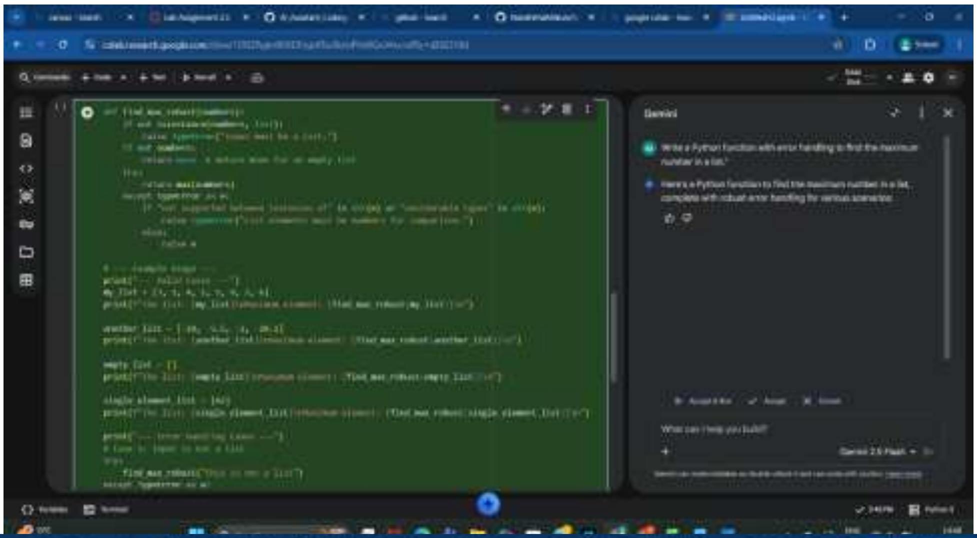
Certainly! The most optimized and short way to find the maximum number in a list in Python is to use the built-in `max()` function. It's highly efficient as it's implemented in C. Here's a function that wraps it, along with examples:

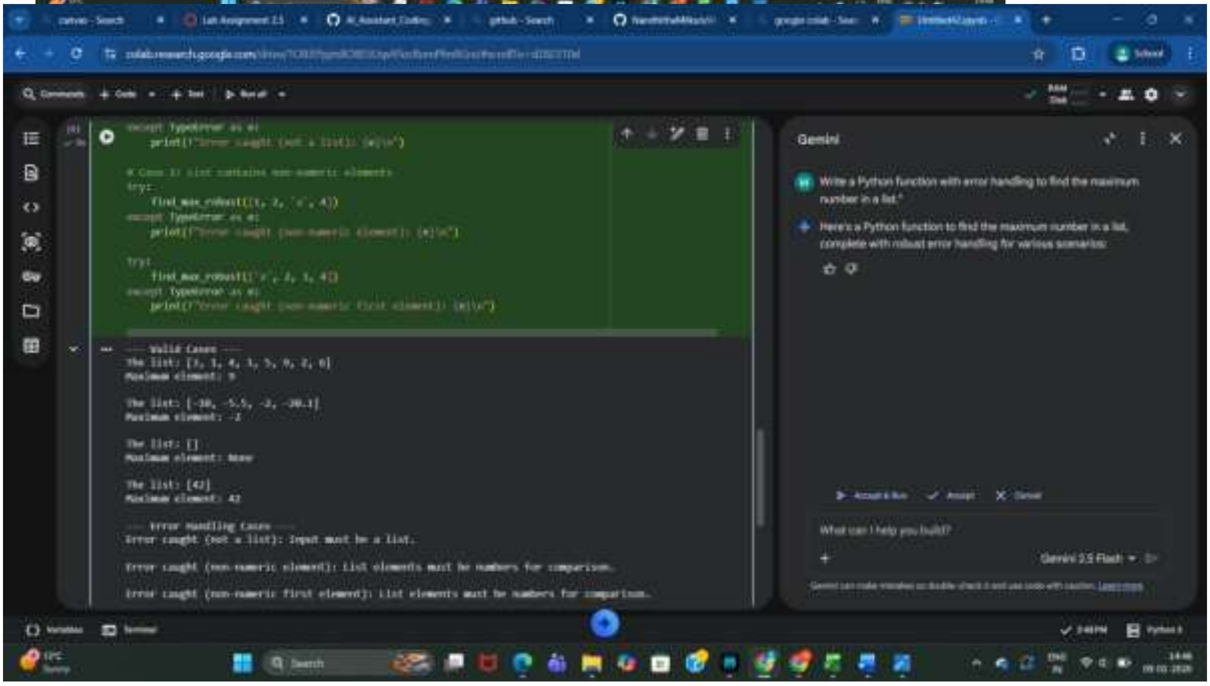
Output Code

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

Prompt 3

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





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

Here's a Python function to find the maximum number in a list, complete with robust error handling for various scenarios:

```
--- Valid Cases ---
The list: [3, 1, 4, 1, 5, 9, 2, 6]
Maximum element: 9

The list: [-10, -5.5, -2, -20.1]
Maximum element: -2

The list: []
Maximum element: None

The list: [42]
Maximum element: 42

--- Error Handling Cases ---
Error caught (not a list): Input must be a list.

Error caught (non-numeric element): List elements must be numbers for comparison.

Error caught (non-numeric first element): List elements must be numbers for comparison.
```

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.