

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 two versions of the same Python code in a Google Colab notebook. The left pane contains the original implementation, which uses a for loop and two separate sum calculations. The right pane shows the AI-improved refactored version, which uses generator expressions to achieve the same result in a more concise and efficient single pass.

```

Original implementation:
def sum_odd_even(numbers):
    odd_sum = 0
    even_sum = 0
    for num in numbers:
        if num % 2 != 0:
            even_sum += num
        else:
            odd_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(my_list)
print(f"Original - Odd sum: {odd_total}, Even sum: {even_total}")

*** Original implementation:
def sum_odd_even(numbers):
    # Using generator expression 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}")

*** Refactored Implementation (concise and efficient single pass):
Refactored - Odd sum: 25, Even sum: 30

```

The screenshot shows the refactored code running with a different list of numbers. The output indicates that the odd sum is 25 and the even sum is 30.

```

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}")

*** 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.

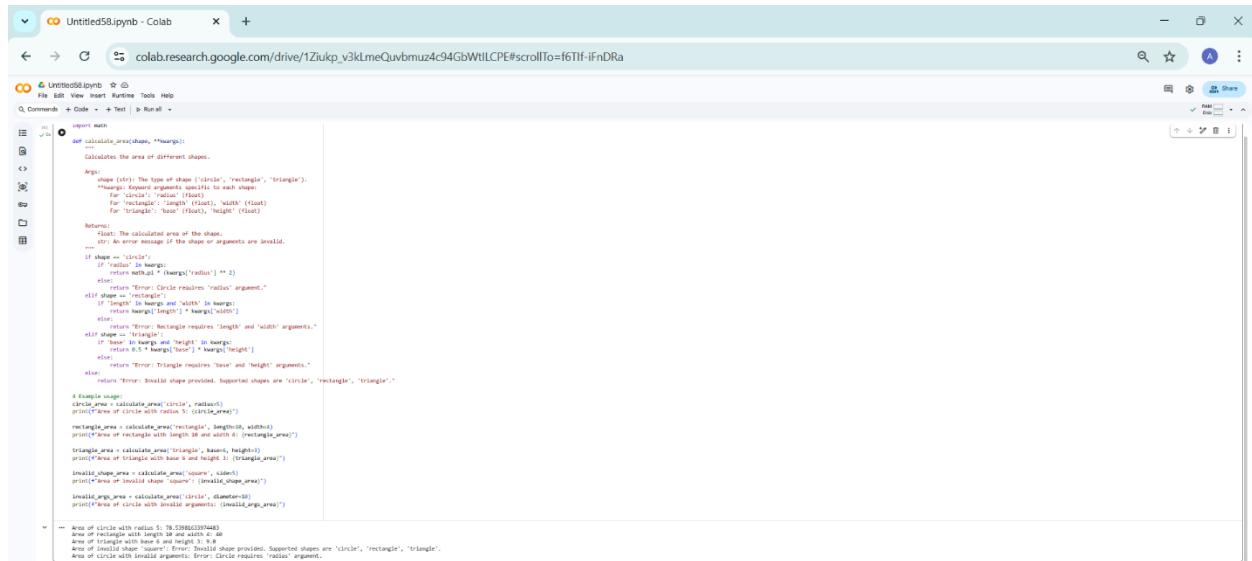
Task 2: Area Calculation Explanation

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."



```
import math

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" not in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: Circle requires 'radius' argument."
    elif shape == "rectangle":
        if "length" not in kwargs or "width" not in kwargs:
            return "Error: Rectangle requires 'length' and 'width' arguments."
        else:
            return kwargs['length'] * kwargs['width']
    else:
        return "Error: Triangle requires 'base' and 'height' arguments."
    else:
        return "Error: Invalid shape provided. Supported shapes are 'circle', 'rectangle', 'triangle'."

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

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

triangle_area = calculate_area('triangle', base=5, height=10)
print("Area of triangle with base 5 and height 10: ", triangle_area)

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

invalid_arg_area = calculate_area('circle', diameter=10)
print("Area of circle with invalid argument: ", invalid_arg_area)

-- Area of circle with radius 5: 78.53982094488
Area of rectangle with length 10 and width 4: 40
Area of triangle with base 5 and height 10: 25
Area of invalid shape 'square': Error: Invalid shape provided. Supported shapes are 'circle', 'rectangle', 'triangle'.
Area of circle with invalid argument: Error: Circle requires 'radius' argument.
```



```

import math

def calculate_area(shape, *args):
    """
    Calculates the area of different shapes.
    Args:
        shape (str): The type of shape ('circle', 'rectangle', 'triangle').
        *args: Required 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.
    """
    if len(args) == 0:
        raise ValueError("An error message of the shape or arguments are invalid.")

    if shape == "circle":
        if len(args) != 1:
            return math.pi * args[0]**2
        else:
            return f"Error: Circle requires 'radius' argument."
    elif shape == "rectangle":
        if len(args) != 2:
            return f"Error: Rectangle requires 'length' and 'width' arguments."
        else:
            return args[0] * args[1]
    elif shape == "triangle":
        if len(args) != 2:
            return f"Error: Triangle requires 'base' and 'height' arguments."
        else:
            return 0.5 * args[0] * args[1]
    else:
        return f"Error: Invalid shape requires 'circle', 'rectangle', 'triangle'."

# Example usage:
if __name__ == "__main__":
    print(f"Area of circle with radius 5: {calculate_area('circle', 5)}")
    rectangle_area = calculate_area('rectangle', length=4, width=3)
    print(f"Area of rectangle with length 4 and width 3: {rectangle_area}")
    triangle_area = calculate_area('triangle', base=4, height=3)
    print(f"Area of triangle with base 4 and height 3: {triangle_area}")
    invalid_shape_area = calculate_area('square', 5)
    print(f"Area of square with side 5: {invalid_shape_area}")
    invalid_args_area = calculate_area('circle', diameter=10)
    print(f"Area of circle with invalid arguments: {invalid_args_area}")

--> Area of circle with radius 5: 78.54000000000001
Area of rectangle with length 4 and width 3: 12.0
Area of triangle with base 4 and height 3: 6.0
Area of square with side 5: 0.0
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."

```
def find_max_simple(numbers):
    if not numbers:
        return None # Handle empty list case
    max_num = numbers[0]
    for num in numbers[1:]:
        if num > max_num:
            max_num = num
    return max_num

# Example usage:
og_list = [1, 2, 3, 4, 5, 6, 7, 8]
max_element = find_max_simple(og_list)
print(f"The maximum element in the list {og_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] is: 8
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 the code:

1. `def find_max_simple(numbers):`
 - 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."

```

File Edit View Insert Tools Help
File Comment Code Test Run all
Finally, it tells you what the absolute biggest number it found was.

def find_max_optimized(numbers):
    if not numbers:
        return None
    else:
        max_element = numbers[0]
        for num in numbers[1:]:
            if num > max_element:
                max_element = num
        return max_element

# Example usage:
my_list_optimized = [1, 2, 3, 4, 5, 6, 7, 8]
max_element_optimized = find_max_optimized(my_list_optimized)
print(f"The maximum element in {my_list_optimized} is {max_element_optimized}")

empty_list_optimized = []
max_element_empty_optimized = find_max_optimized(empty_list_optimized)
print(f"The maximum element in an empty list {empty_list_optimized} is {max_element_empty_optimized}")

```

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

Explanation of the `find_max_optimized` function:

This function is a much more concise and Pythonic way to find the maximum number in a list, leveraging Python's built-in capabilities.

Let's break down how it works:

1. `def find_max_optimized(numbers):`: This defines our function, named `find_max_optimized`.
2. `if not numbers:`: Similar to the previous function, this checks if the `numbers` list is empty. If it is, the function returns `None`, because there's no maximum element to find.
3. `else:` `max_element = numbers[0]`: This is the core of the optimized function. Python has a built-in function called `max()` that takes an iterable (like a list) and returns its largest item. This function is highly optimized at the C level, making it extremely efficient for this task, especially with large lists.
4. `for num in numbers[1:]:`: In simple terms, this function just checks if the list is empty first, if not, it iterates over the list to Python's super fast built-in `max()` function, which does all the heavy lifting to find the biggest number for you, and then returns that number.

What can I help you build? Accept & Run Cancel



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."

```

File Edit View Insert Tools Help
File Comment Code Test Run all
def find_max_with_error_handling(numbers):
    if not numbers:
        raise TypeError("Input must be a list.")

    if not numbers:
        raise TypeError("Input must be a list, and must contain at least one element.")

    # Validate that all elements are numeric.
    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError("All elements in the list must be numeric.")

    return max(numbers)

# Example usage:
my_list_valid = [1, 2, 3, 4, 5, 6, 7, 8]
find_max_with_error_handling(my_list_valid)
print(f"The maximum element in {my_list_valid} is {max_element_valid}")

empty_list_valid = []
find_max_with_error_handling(empty_list_valid)
print(f"The maximum element in an empty list {empty_list_valid} is {max_element_empty_valid}")

print("----- Error handling cases -----")
invalid_type_list = [1, "a", 3]
find_max_with_error_handling(invalid_type_list)
print(f"Error with list {invalid_type_list}: {error}")

another_invalid_list = [1, 20, None]
find_max_with_error_handling(another_invalid_list)
print(f"Error with list {another_invalid_list}: {error}")

# Valid cases ...
the_maximum_element_is_in_a_list = [1, 2, 3, 4, 5, 6, 7, 8]
find_max_with_error_handling(the_maximum_element_is_in_a_list)
print(f"The maximum element in [1, 2, 3, 4, 5, 6, 7, 8] is {9}")

# Error handling cases
error_with_type_error = [1, 2, 3]
find_max_with_error_handling(error_with_type_error)
print(f"Error with list {error_with_type_error}: {error}")

error_with_value_error = [1, 2, 3, 4, 5, 6, 7, 8]
find_max_with_error_handling(error_with_value_error)
print(f"Error with list {error_with_value_error}: {error}")

error_with_index_error = [1, 2, 3, 4, 5, 6, 7, 8]
find_max_with_error_handling(error_with_index_error)
print(f"Error with list {error_with_index_error}: {error}")

```

Explanation of the `find_max_with_error_handling` function:

This function builds upon the optimized approach by adding robust error handling to ensure the input is always valid for finding a maximum numeric value.

Let's break down the added error handling:

1. `if not numbers:`: This is the first line of defense to check if the `numbers` argument is actually a Python `list`. If it's not (e.g., if you pass a string, integer, or tuple), it immediately raises a `TypeError` with a descriptive message. This prevents unexpected behavior or errors later if the input type is fundamentally wrong.
2. `if not numbers:`: This is the same check as the `if not numbers:` in the `find_max_optimized` function. If the list is empty, there's no maximum element, so it returns `None`. This is a graceful way to handle a common edge case without raising an error.
3. `for num in numbers:`:
 - After confirming the input is a list and not empty, this loop iterates through each element in the list.
 - 4. `if not isinstance(num, (int, float)):`: Inside the loop, for each `num`, it checks if the element is either an `int` (integer) or a `float` (floating-point number). These are the standard numeric types for which `max()` would typically work.
 - 5. `raise TypeError("All elements in the list must be numeric.")`: If the element is neither an `int` nor a `float`, or another object type, it raises a `TypeError`. This ensures that you only try to find the maximum of comparable numeric types, preventing `TypeError` from the `max()` function itself if it encounters mixed types.

What can I help you build? Accept & Run Cancel



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:\n{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.