Assignment-2.1

Hall.No:2303A510H1

Batch No:14

Name: K.Mahesh Reddy

Task 1

Statistical Summary for Survey Data

❖ **Scenario:**

You are a data analyst intern working with survey responses stored as numerical lists.

❖ **Task:**

Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

❖ **Expected Output:**

➢ Correct Python function

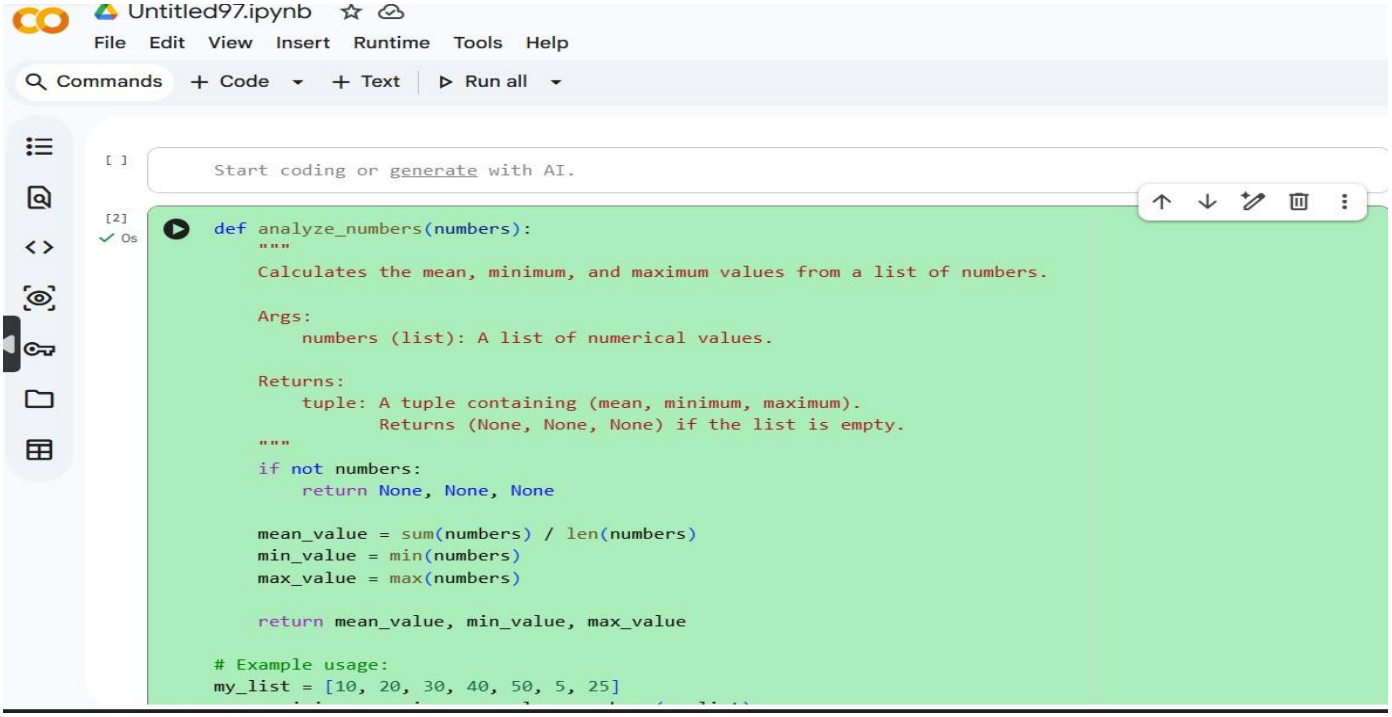
➢ Output shown in Colab

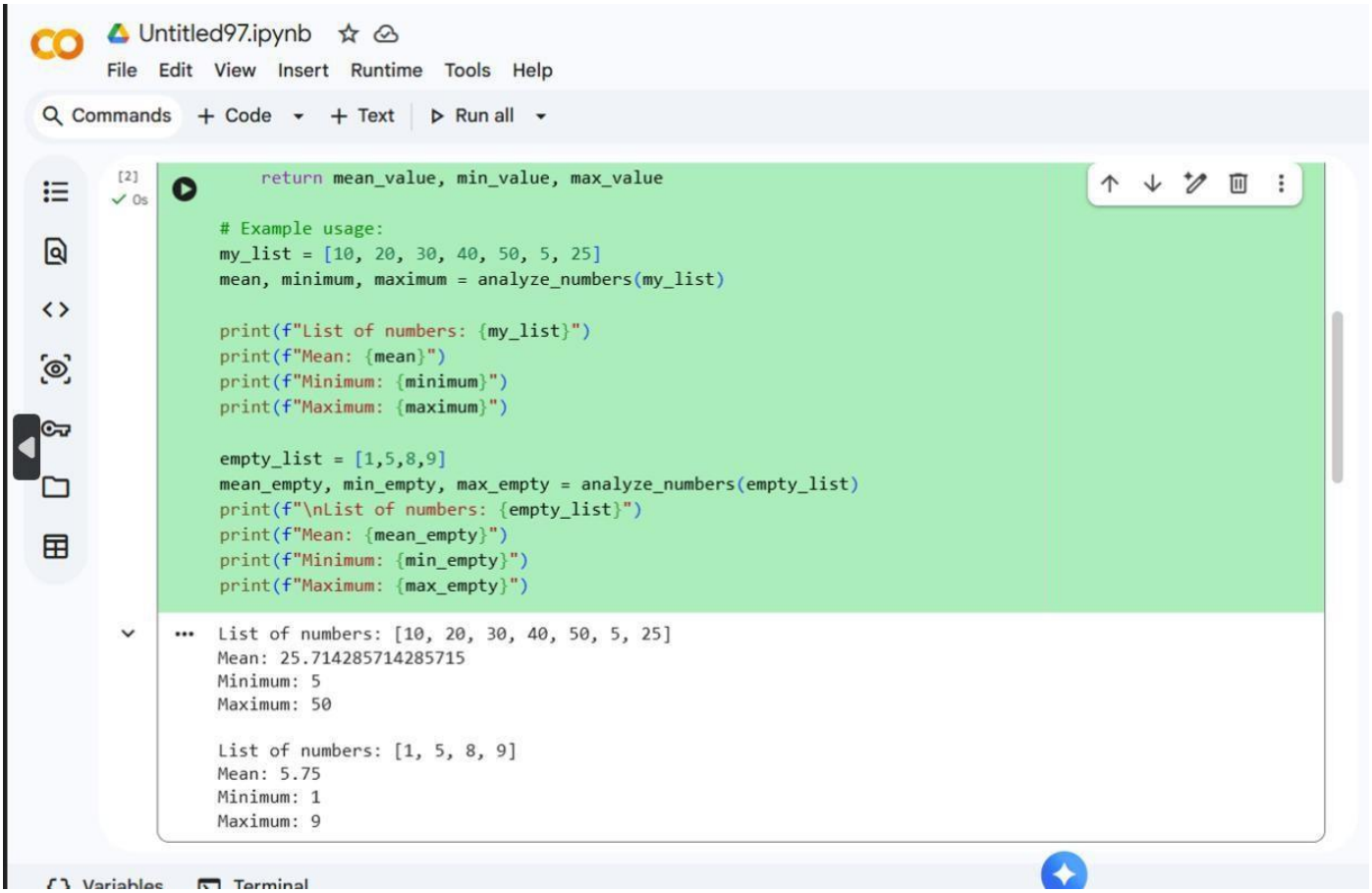
➢ Screenshot of Gemini prompt and result

Prompt

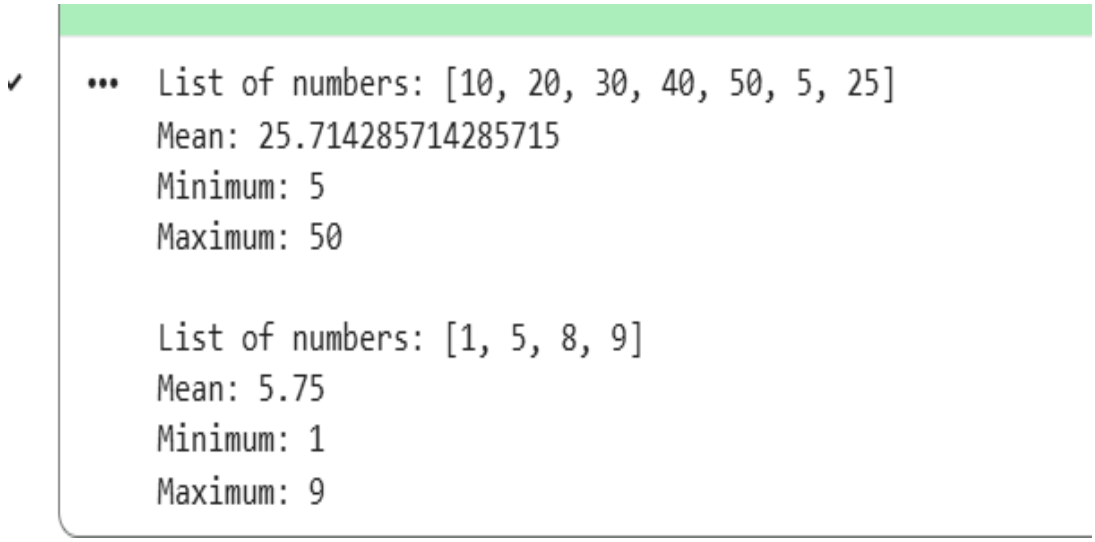
Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

Code




Untitled97.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[2]
0s
return mean_value, min_value, max_value
# Example usage:
my_list = [10, 20, 30, 40, 50, 5, 25]
mean, minimum, maximum = analyze_numbers(my_list)
print(f"List of numbers: {my_list}")
print(f"Mean: {mean}")
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")
empty_list = [1,5,8,9]
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print(f"\nList of numbers: {empty_list}")
print(f"Mean: {mean_empty}")
print(f"Minimum: {min_empty}")
print(f"Maximum: {max_empty}")
List of numbers: [10, 20, 30, 40, 50, 5, 25]
Mean: 25.714285714285715
Minimum: 5
Maximum: 50
List of numbers: [1, 5, 8, 9]
Mean: 5.75
Minimum: 1
Maximum: 9
Variables
Terminal

Output:


List of numbers: [10, 20, 30, 40, 50, 5, 25]
Mean: 25.714285714285715
Minimum: 5
Maximum: 50
List of numbers: [1, 5, 8, 9]
Mean: 5.75
Minimum: 1
Maximum: 9

Justification:

This task demonstrates how AI tools can assist in quickly generating reliable utility functions for basic data analysis. Survey datasets often require immediate preprocessing to understand value ranges and central tendencies. By using an AI-generated Python function, the analyst is able to obtain accurate mean, minimum, and maximum values with minimal effort. Running the code in Google Colab ensures correctness through instant execution and visible output. This task highlights how AI support improves productivity while maintaining correctness in simple yet essential analytical computations.

Task-2

Armstrong Number – AI Comparison

❖ **Scenario:**

You are evaluating AI tools for numeric validation logic.

❖ **Task:**

Generate an Armstrong number checker using Gemini and GitHub Copilot.

Compare their outputs, logic style, and clarity.

❖ **Expected Output:**

➢ Side-by-side comparison table

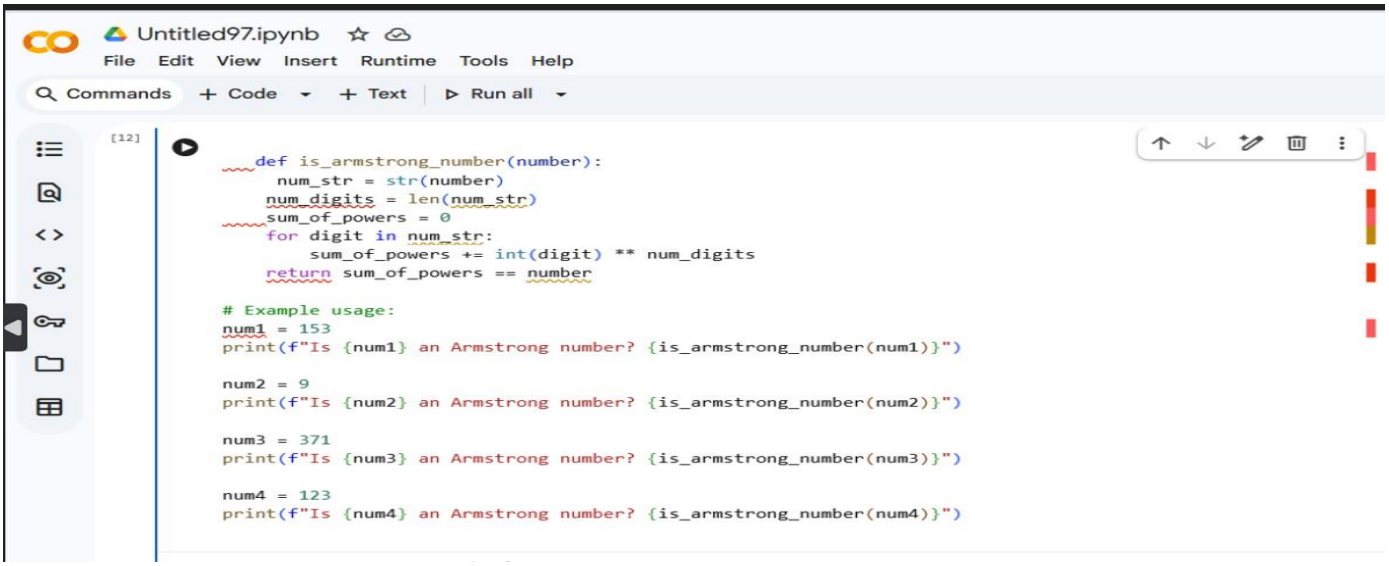
➢ Screenshots of prompts and generated code

Using Gemini

Prompt:

Write a Python program to check whether a given number is an Armstrong number.

Code:



Output:

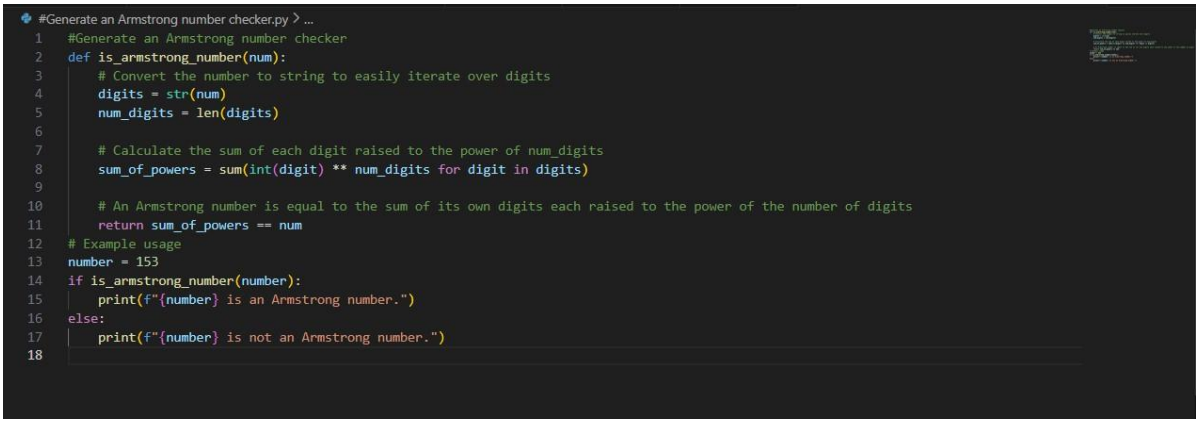
```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 371 an Armstrong number? True
Is 123 an Armstrong number? False
```

Using Github Copilot:

Prompt:

Generate a code which can check weather a number is a Armstrong number or not

Code:



```python
#Generate an Armstrong number checker
def is_armstrong_number(num):
    # Convert the number to string to easily iterate over digits
    digits = str(num)
    num_digits = len(digits)

    # Calculate the sum of each digit raised to the power of num_digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in digits)

    # An Armstrong number is equal to the sum of its own digits each raised to the power of the number of digits
    return sum_of_powers == num
# Example usage
number = 153
if is_armstrong_number(number):
    print(f"{number} is an Armstrong number.")
else:
    print(f"{number} is not an Armstrong number.")

```

Output:

```
PS C:\Users\neera\OneDrive\Desktop\AI AC> & "C:/Users/neera/OneDrive/Desktop/AI AC/.venv/Scripts/python.exe" "c:/Users/neera/OneDrive/Desktop/AI AC/#write.py"
153 is an Armstrong number.
```

Logic Style Comparison:

Aspect	Google Gemini	Github Copilot
Code Length	Short and concise	slightly longer
Logic Style	pythonic	Traditional
Readability	clear and compact	Step by step
Beginner friendly	moderate	High
Output Accuracy	correct	Correct

Justification:

This task focuses on understanding how different AI coding assistants approach the same computational problem. By generating Armstrong number validation logic using both Google Gemini and GitHub Copilot, variations in implementation style, structure, and readability become evident. While both solutions achieve correct results, the differences help evaluate which tool better aligns with developer preferences and coding standards. This task justifies the need to compare AI tools before adopting them in real projects, as logic clarity and maintainability are as important as correctness.

Task-3

Leap Year Validation Using Cursor AI

❖ **Scenario:**

You are validating a calendar module for a backend system.

❖ **Task:**

Use Cursor AI to generate a Python program that checks whether a given year is a leap year.

Use at least two different prompts and observe changes in code.

❖ **Expected Output:**

➢ Two versions of code

➢ Sample inputs/outputs

➢ Brief comparison

Prompt1:

Write a Python program to check whether a given year is a leap year.

Code:

```
year = int(input("Enter a year: "))

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print(year, "is a leap year")
else:
    print(year, "is not a leap year")
    Ctrl+L to chat, Ctrl+K to generate
```

Output:

```
Problems  Output  Debug Console  Terminal  Ports                                   powershell

Enter a year: 24
24 is a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 2024
2024 is a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 2026
2026 is not a leap year
PS C:\Program Files\cursor>
```

Prompt2:

Create a Python function that returns true if a year is a leap year, otherwise false.

Code:

```
> Users > neera > OneDrive > Desktop > AI AC > python.py > ...
1   def is_leap_year(year):
2       return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
3   year = int(input("Enter a year: "))
4   if is_leap_year(year):
5       print(year, "is a leap year")
6   else:
7       print(year, "is not a leap year")
8
```

Output:

```
Problems  Output  Debug Console  Terminal  Ports                                    powershell
Enter a year: 2026
2026 is not a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 2025
2025 is not a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 1994
1994 is not a leap year
PS C:\Program Files\cursor>
```

Justification:

This task emphasizes the role of prompt specificity in AI-generated code quality. Using two different prompts with Cursor AI resulted in distinct implementations of leap year validation logic. The variation between outputs illustrates how clearer and more detailed prompts lead to more accurate and reusable solutions. This is especially important in backend systems where incorrect date logic can cause serious errors. The task validates that developers must actively refine prompts to obtain reliable AI-assisted solutions rather than relying on generic instructions.

Task-4

Student Logic + AI Refactoring (Odd/Even Sum)

❖ **Scenario:**

Company policy requires developers to write logic before using AI.

❖ **Task:**

Write a Python program that calculates the sum of odd and even numbers in a tuple, then refactor it using any AI tool.

❖ **Expected Output:**

➢ Original code

➢ Refactored code

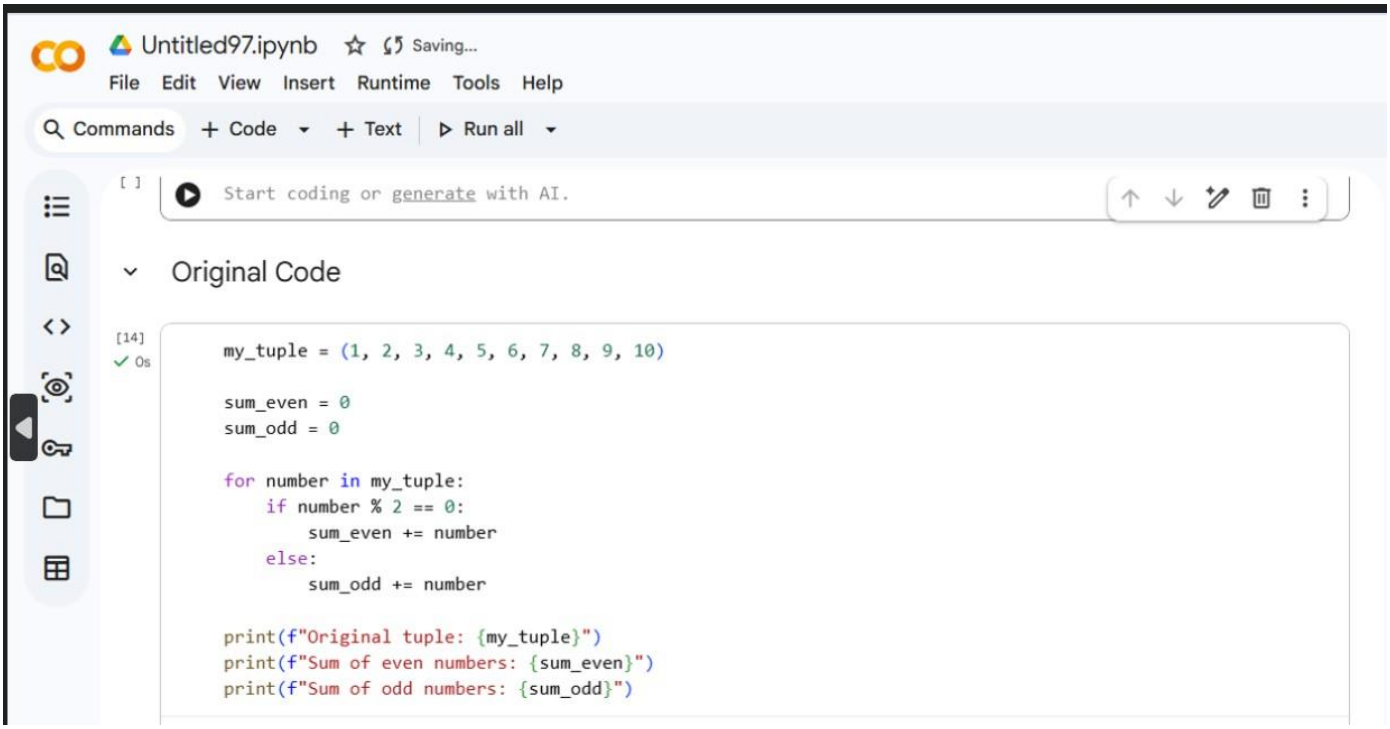
➢ Explanation of improvements

Prompt:

Write a Python program that calculates the sum of odd and even numbers in a tuple.

Code:

1)Original code:



Output:

```
Original tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Refactored - Sum of even numbers: 30
Refactored - Sum of odd numbers: 25
```

2) Refactored code

Code:

Refactored Code (using AI assistance)

```
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Refactored using generator expressions for conciseness and efficiency
sum_even_refactored = sum(number for number in my_tuple if number % 2 == 0)
sum_odd_refactored = sum(number for number in my_tuple if number % 2 != 0)

print(f"Original tuple: {my_tuple}")
print(f"Refactored - Sum of even numbers: {sum_even_refactored}")
print(f"Refactored - Sum of odd numbers: {sum_odd_refactored}")
```

```
Original tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Refactored - Sum of even numbers: 30
Refactored - Sum of odd numbers: 25
```

Output:

```
Original tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Refactored - Sum of even numbers: 30
Refactored - Sum of odd numbers: 25
```

Explanation of Improvements

The refactored code leverages Python's built-in sum() function along with generator expressions. This approach offers several benefits.

First, conciseness: the refactored code significantly reduces the number of lines, making the logic more compact and clean.

Second, readability: for developers familiar with Pythonic constructs, generator expressions clearly express the intent of the code, such as calculating the sum of numbers based on a condition, which makes the program easier to understand at a glance..

Justification:

This task assesses Copilot's ability to generate different logical strategies for the same problem. Comparing basic and optimized approaches highlights differences in execution flow, time complexity, and performance, helping determine when each approach is appropriate.