

Assignment-2

M.Paramesh

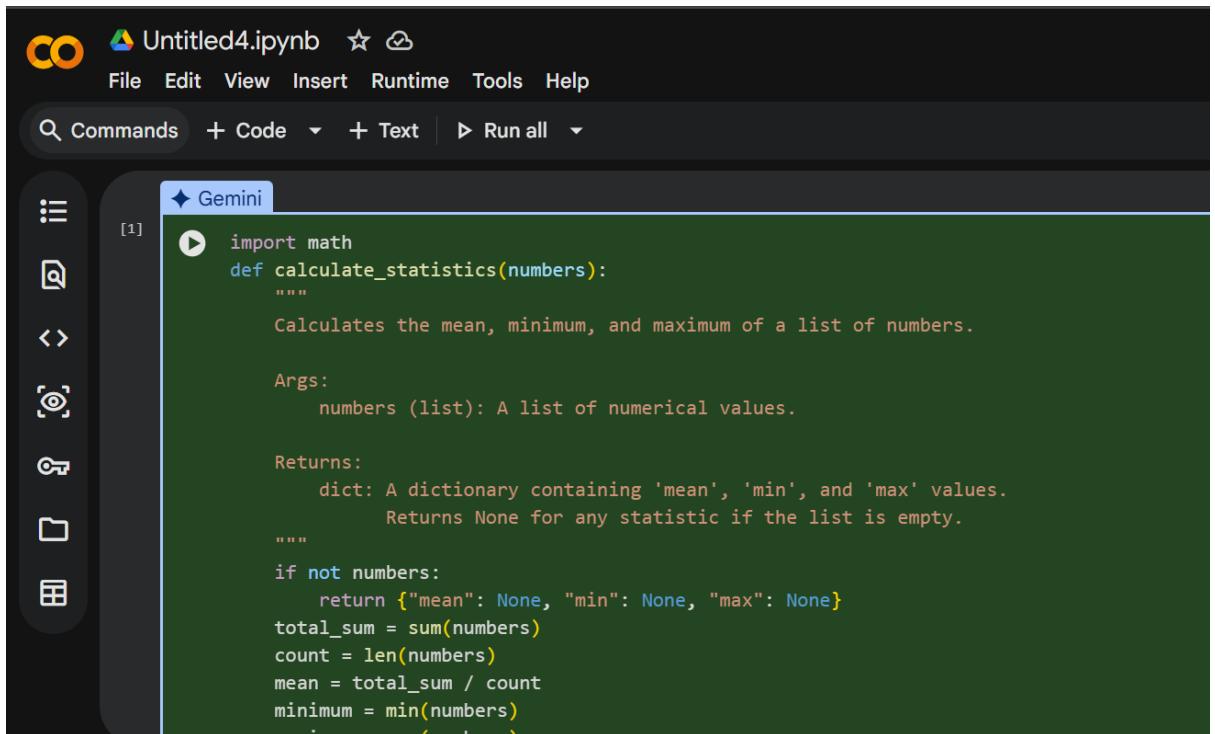
H.No. : 2303A53021

Batch : 46

Task 1 — Statistical Summary using Google Gemini (Colab)

Create a Python function that finds:

- Mean
 - Minimum
 - Maximum
- from a list of numbers using **Gemini in Colab**.



The screenshot shows a Google Colab interface with the following details:

- Title:** Untitled4.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help
- Search Bar:** Commands, + Code, + Text, Run all
- Sidebar:** A vertical sidebar with various icons for file operations.
- Code Cell:** [1] **Gemini**

```
import math
def calculate_statistics(numbers):
    """
    Calculates the mean, minimum, and maximum of a list of numbers.

    Args:
        numbers (list): A list of numerical values.

    Returns:
        dict: A dictionary containing 'mean', 'min', and 'max' values.
        Returns None for any statistic if the list is empty.
    """
    if not numbers:
        return {"mean": None, "min": None, "max": None}
    total_sum = sum(numbers)
    count = len(numbers)
    mean = total_sum / count
    minimum = min(numbers)
    maximum = max(numbers)
```

```
    if not numbers:
        return {"mean": None, "min": None, "max": None}
    total_sum = sum(numbers)
    count = len(numbers)
    mean = total_sum / count
    minimum = min(numbers)
    maximum = max(numbers)

    return {"mean": mean, "min": minimum, "max": maximum}

# Example usage:
my_list = [10, 20, 5, 15, 30, 25]
stats = calculate_statistics(my_list)
print(f"For the list {my_list}:")
print(f"  Mean: {stats['mean']}")  
print(f"  Minimum: {stats['min']}")  
print(f"  Maximum: {stats['max']}")

empty_list = []
empty_stats = calculate_statistics(empty_list)
print(f"\nFor an empty list:")
```

s

Terminal



Untitled4.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[1] print(f" Statistics: {empty_stats}")

single_number_list = [42]
single_stats = calculate_statistics(single_number_list)
print(f"\nFor a list with a single number {single_number_list}:")
print(f" Mean: {single_stats['mean']}")
print(f" Minimum: {single_stats['min']}")
print(f" Maximum: {single_stats['max']}")

... For the list [10, 20, 5, 15, 30, 25]:
Mean: 17.5
Minimum: 5
Maximum: 30

For an empty list:
Statistics: {'mean': None, 'min': None, 'max': None}

For a list with a single number [42]:
Mean: 42.0
Minimum: 42
Maximum: 42

Variables Terminal

Task 2 — Armstrong Number (Gemini vs Copilot)

Generate Armstrong number checker using:

- Gemini
- GitHub Copilot ,Then compare them.

The screenshot shows a Jupyter Notebook interface with two code cells. The left cell contains the following Python code:

```
def is_armstrong_number(number):
    """
    Checks if a number is an Armstrong number.

    An Armstrong number is a number that is equal to the sum of its own digit each raised to the power of the number of digits.
    For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 + 125

    Args:
        number (int): The number to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    if not isinstance(number, int) or number < 0:
        raise ValueError("Input must be a non-negative integer.")
```

The right side of the interface shows the Gemini AI interface. A user prompt asks for a Python program to check for Armstrong numbers. Gemini responds with the provided code and offers options to accept or run it.

The screenshot shows a Jupyter Notebook interface with two code cells. The left cell contains the following Python code:

```
num_str = str(number)
num_digits = len(num_str)

sum_of_powers = 0
for digit_char in num_str:
    digit = int(digit_char)
    sum_of_powers += digit ** num_digits

return sum_of_powers == number

# Example usage:
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # Expected: True
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") # Expected: True
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # Expected: False
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") # Expected: True
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # Expected: False
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}") # Expected: False
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}") # Expected: True
```

The right side of the interface shows the Gemini AI interface. A user prompt asks for a Python program to check for Armstrong numbers. Gemini responds with the provided code and offers options to explain the logic, add standard deviation, or test edge cases. It also provides a 'What can I help you build?' section and a note about potential mistakes.

The screenshot shows a GitHub Copilot interface. On the left, there's a terminal window titled "Terminal" containing Python code to check if numbers are Armstrong numbers. The code uses a function named `is_armstrong_number` and prints out results for various numbers. On the right, a sidebar labeled "Gemini" shows a list of suggestions or completions. At the bottom, there are two buttons: "+ Code" and "+ Text".

```

print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}")
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # Expected:
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # Expected:
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}") # Expected:
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}") # Expected:

...
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 407 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
  
```

Feature	Gemini	Copilot
Logic	Uses step-by-step digit extraction and power calculation, easy to follow	Uses the same core logic but often written in a more compact style
Code Clarity	Well-structured code with meaningful variable names and comments	Shorter code, fewer comments, more developer-style
Readability	Very beginner-friendly and easy to understand	Slightly less verbose but clean and professional

Task 3 — Leap Year using Cursor AI

Generate leap year program using **two different prompts**

The screenshot shows the Cursor AI interface with two separate code completion sessions for generating a Python leap year program.

Prompt 1: "Write a Python program to check if a year is leap year."

Code Generated:

```
Python leap year program
3m · +449 · Auto

Creating a Python program to check if a year is a leap year.

leap_year.py (new) +46 -1

def is_leap_year(year):
    """
    Check if a given year is a leap year.

    A year is a leap year if:
    - It is divisible by 4, AND
    - It is NOT divisible by 100, OR
    - It IS divisible by 400

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    if year % 400 == 0:
```

Prompt 2: "Creating a Python program to check if a year is a leap year."

Code Generated:

```
leap_year.py (new) +46 -1

def is_leap_year(year):
    """
    Check if a given year is a leap year.

    A year is a leap year if:
    - It is divisible by 4, AND
    - It is NOT divisible by 100, OR
    - It IS divisible by 400

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    if year % 400 == 0:
```

The interface includes a sidebar with navigation links like "New Agent", "Show Terminal", "Show Files", "Search Files", "Open Browser", and "Maximize Chat". At the bottom, there are buttons for "Plan", "@", "File", and a microphone icon, along with dropdown menus for "Agent" and "Auto".

Python leap year program
5m · +449 · Auto

give me optimized code with comparison

OPTIMIZATION_COMPARISON.md (new) +98

```
# Leap Year Implementation: Original vs Optimized

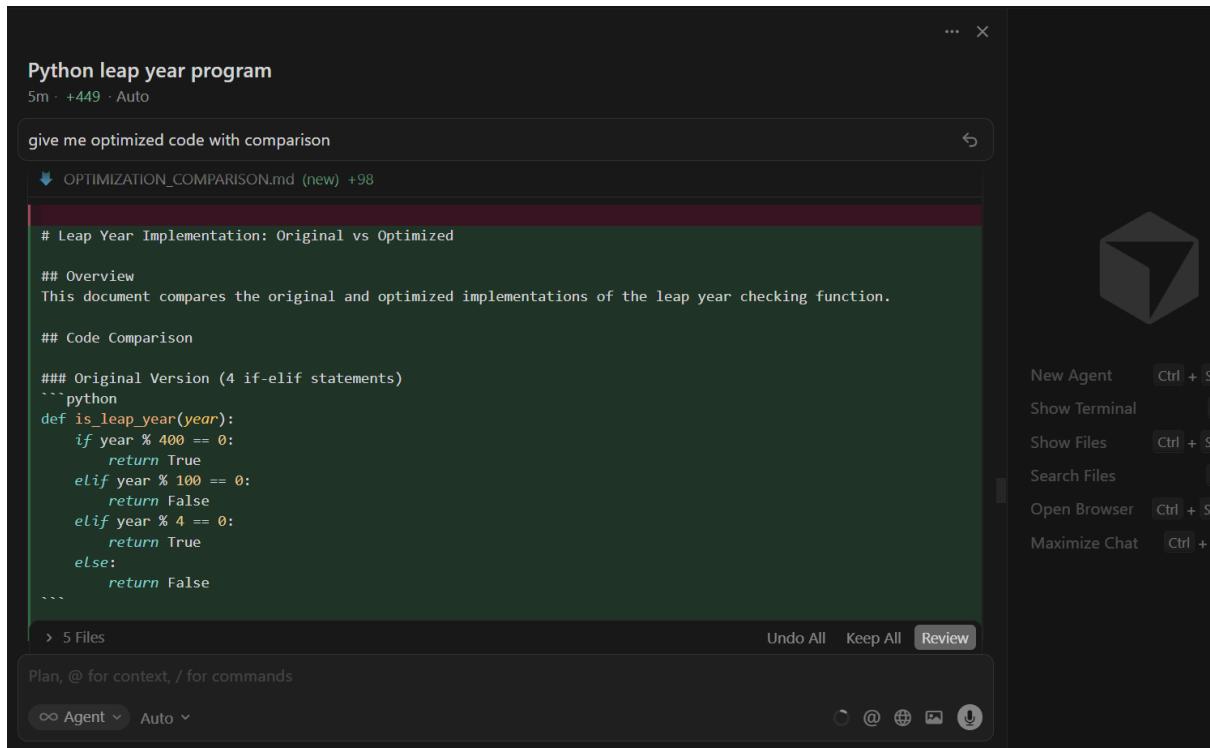
## Overview
This document compares the original and optimized implementations of the leap year checking function.

## Code Comparison

### Original Version (4 if-elif statements)
```python
def is_leap_year(year):
 if year % 400 == 0:
 return True
 elif year % 100 == 0:
 return False
 elif year % 4 == 0:
 return True
 else:
 return False
```
... 5 Files Undo All Keep All Review
```

Plan, @ for context, / for commands

Agent Auto



Python leap year program
6m · +449 · Auto

give me optimized code with comparison

Benefits of Optimized Version:

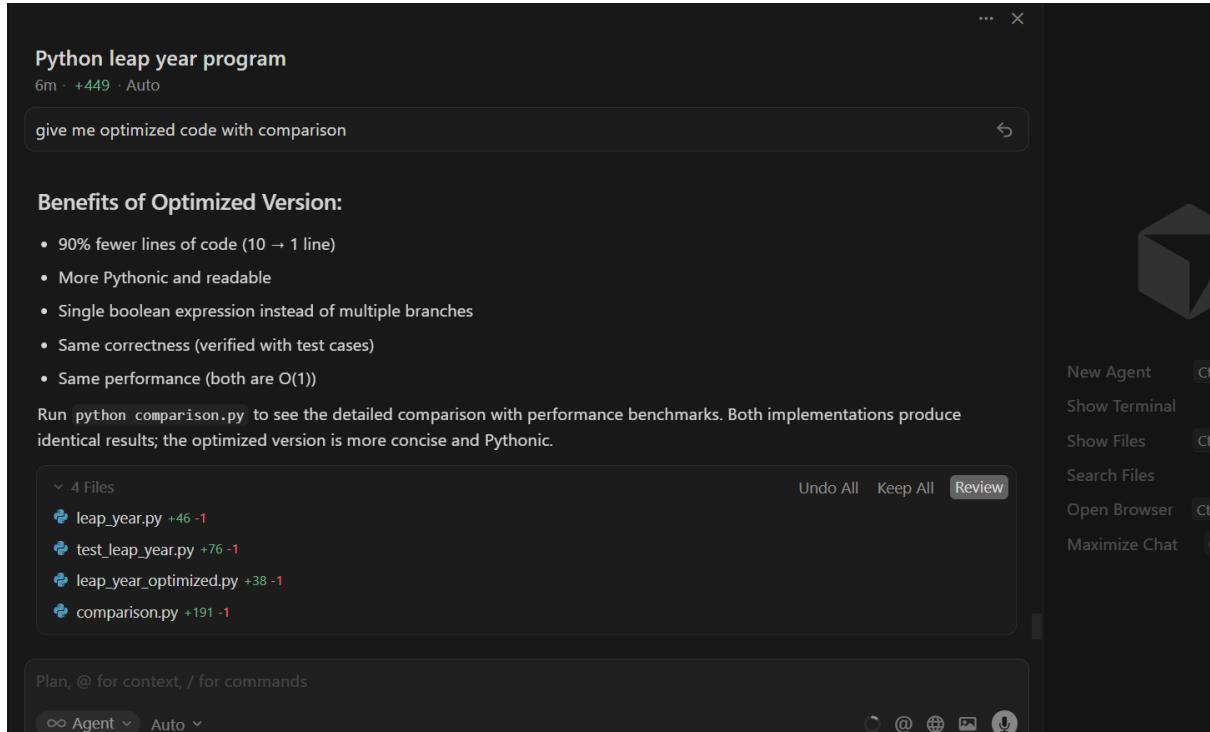
- 90% fewer lines of code (10 → 1 line)
- More Pythonic and readable
- Single boolean expression instead of multiple branches
- Same correctness (verified with test cases)
- Same performance (both are O(1))

Run `python comparison.py` to see the detailed comparison with performance benchmarks. Both implementations produce identical results; the optimized version is more concise and Pythonic.

4 Files leap_year.py +46 -1 test_leap_year.py +76 -1 leap_year_optimized.py +38 -1 comparison.py +191 -1 Undo All Keep All Review

Plan, @ for context, / for commands

Agent Auto



Task 4 — Odd/Even Sum (Manual + AI Refactor)

Manual Code

```

# Tuple of numbers
t = (10, 15, 20, 25, 30, 35)

even_sum = 0
odd_sum = 0

# Loop through each number
for num in t:
    if num % 2 == 0:
        even_sum += num
    else:
        odd_sum += num

print("Sum of Even Numbers:", even_sum)
print("Sum of Odd Numbers:", odd_sum)

```

Output:

```

Sum of Even Numbers: 60
Sum of Odd Numbers: 75

```

AI Refactored Code (Optimized Version)

```

def calculate_even_odd_sum(numbers):
    """Returns the sum of even and odd numbers from a tuple."""
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum

numbers = (10, 15, 20, 25, 30, 35)
even, odd = calculate_even_odd_sum(numbers)

print(f"Sum of Even Numbers: {even}")
print(f"Sum of Odd Numbers: {odd}")

```

Output :

```

Sum of Even Numbers: 60
Sum of Odd Numbers: 75

```

