

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

Name:S.Hasitha varshini

Hall ticket no:2303A54038

Lab Objectives

- To explore and evaluate the functionality of **Google Gemini** for AI-assisted coding within **Google Colab**
 - To understand and use **Cursor AI** for code generation, explanation, and refactoring
 - To compare outputs and usability between **Gemini**, **GitHub Copilot**, and **Cursor AI**
 - To perform code optimization and documentation using AI tools
-

Lab Outcomes (LOs)

After completing this lab, students are able to:

- Generate Python code using Google Gemini in Google Colab
 - Analyze the effectiveness of Gemini's code explanations and suggestions
 - Set up and use Cursor AI for AI-powered coding assistance
 - Evaluate and refactor code using Cursor AI features
 - Compare AI tool behavior and code quality across different platforms
-

Task 1: Statistical Summary for Survey Data

Scenario

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

Prompt Used in Google Gemini (Colab)

"Write a Python function that takes a list of numbers and calculates the mean, minimum, and maximum values. Display the output."

Python Code Generated by Gemini

The screenshot shows a Google Colab interface with a dark theme. On the left, a code cell contains Python code for calculating statistics (mean, minimum, maximum) from a list of numbers. The code includes examples for an empty list and a non-empty list. On the right, the Gemini AI panel is open, displaying generated code and its output. Gemini provides results for three lists: [10, 20, 30, 40, 50, 60, 70], [5, 12, 8, 20, 3, 15], and the empty list [].

```
[1] Os
print("List of numbers: (data2)")
print("Mean: (mean2)")
print("Minimum: (minimum2)")
print("Maximum: (maximum2)")

# Example with an empty list
data_empty = []
mean_empty, min_empty, max_empty = calculate_stats(data_empty)

print("List of numbers: (data_empty)")
print("Mean: (mean_empty)")
print("Minimum: (min_empty)")
print("Maximum: (max_empty)")

...
List of numbers: [10, 20, 30, 40, 50, 60, 70]
Mean: 40
Minimum: 10
Maximum: 70

List of numbers: [5, 12, 8, 20, 3, 15]
Mean: 10.5
Minimum: 3
Maximum: 20

List of numbers: []
Mean: None
Minimum: None
Maximum: None
```

Gemini

- Certainly! Here's a Python function that calculates the mean, minimum, and maximum values from a list of numbers, along with an example of how to use it.

The code you provided executed successfully! Here are the results:

- For the list [10, 20, 30, 40, 50, 60, 70]:
 - Mean: 40
 - Minimum: 10
 - Maximum: 70
- For the list [5, 12, 8, 20, 3, 15]:
 - Mean: 10.5
 - Minimum: 3
 - Maximum: 20
- For the empty list []:
 - Mean: None
 - Minimum: None
 - Maximum: None

The function correctly handles the empty list by returning `None` for all statistics, as designed.

Explain the "calculate_stats" function logic | Add median calculation to "calculate_stats" | Test "calculate_stats" with a list of floats

What can I help you build? + Gemini 2.5 Flash ▶

Gemini can make mistakes so double-check it and use code with caution. Learn more

Output (Shown in Colab)

Mean: 36.2

Minimum: 12

Maximum: 67

Observation

- Gemini generated clean and readable code.
- Logic is concise and efficient.
- Explanation provided by Gemini was clear and beginner-friendly.

Task 2: Armstrong Number – AI Comparison

Scenario

You are evaluating AI tools for numeric validation logic.

Gemini-Generated Code

The screenshot shows a Jupyter Notebook in Google Colab. The main code cell contains Python code to check if a number is an Armstrong number. The sidebar features a Gemini AI interface with a card about Armstrong numbers and a message from Gemini.

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

    An Armstrong number is a number that is the sum of its own digits each raised to the power of the number of digits.

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

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.

    If not isinstance(number, int) or number < 0:
        return False # Armstrong numbers are typically defined for positive integers

    # Converts the number to a string to easily get its digits and count
    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("Is 153 an Armstrong number? ", is_armstrong_number(153)) # Expected: True (1^3 + 5^3 + 3^3 = 1+125+27 = 153)
print("Is 370 an Armstrong number? ", is_armstrong_number(370)) # Expected: True (3^3 + 7^3 + 0^3 = 27+343+0 = 370)
print("Is 9474 an Armstrong number? ", is_armstrong_number(9474)) # Expected: True (9^4 + 4^4 + 7^4 + 4^4 = 6561+256+2401+256 = 9474)
print("Is 123 an Armstrong number? ", is_armstrong_number(123)) # Expected: False
print("Is 9 an Armstrong number? ", is_armstrong_number(9)) # Expected: True (9^3 = 9 - single-digit numbers are Armstrong number)
print("Is 10 an Armstrong number? ", is_armstrong_number(10)) # Expected: True (0^3 = 0)
print("Is -153 an Armstrong number? ", is_armstrong_number(-153)) # Expected: False (due to input validation)
```

Gemini

B **Armstrong Number**

An Armstrong number (also known as a narcissistic number, a pluperfect digital invariant, or a plus perfect number) is a number that is the sum of its own digits each raised to the power of the number of digits.

For example, 153 is an Armstrong number because it has three digits, and $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

Here's a Python function to check for Armstrong numbers, with some examples:

Accept Run Accept Cancel

What can I help you build?

Gemini 2.5 Flash ▾

Gemini can make mistakes so double-check it and use code with caution. Learn more

GitHub Copilot-Generated Code

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows two open files: `FIBONACI.py` and `armstrong.py`. The `armstrong.py` file is the active editor.
- Terminal:** Shows the output of running the `armstrong.py` script. It prompts for the number of terms (6) and then lists the first 6 Fibonacci series numbers. It then asks for a non-negative integer and prints the Armstrong numbers up to 78.
- Status Bar:** Shows the path as `C:\AI assisted & C:\Users\Meghana\AppData\Local\Microsoft\WindowsApps\python 3.11.exe "C:\AI assisted\LAB1\armstrong.py"`.

Comparison

Table

Criteria	Gemini	GitHub Copilot
Logic Style	Pythonic, uses comprehension	Traditional loop-based
Readability	Very high	Moderate
Code Length	Short	Slightly longer
Beginner Friendly	Yes	Yes
Performance	Efficient	Efficient

Conclusion

- Gemini emphasizes modern Python practices.
 - Copilot follows classical algorithmic style.
 - Both produce correct and reliable results.
-

Task 3: Leap Year Validation Using Cursor AI

Scenario

You are validating a calendar module for a backend system.

Prompt 1 Used in Cursor AI

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

Version 1 Code

The screenshot shows the Cursor AI interface with the following details:

- File Explorer:** Shows a project structure with files: `leap_year.py`, `is_leap_year`, and `main`.
- Code Editor:** Displays the Python code for a leap year checker. The code includes a function `is_leap_year` and a `main` function for testing.
- Code Preview:** A right-hand panel shows the generated code for "Python leap year check". It includes a docstring, features, and usage instructions.
- Bottom Status Bar:** Shows file paths, cursor position (Ln 1, Col 1), encoding (UTF-8), and Python version (3.13.9 64-bit (Microsoft Store)).

Prompt 2 Used in Cursor AI

"Optimize the leap year program and convert it into a reusable function."

Version 2 Code

The screenshot shows a code editor interface with an AI-assisted toolbar at the top. On the left, there's a sidebar with 'AI ASSISTED' sections for LAB1 and LAB2, and a file list containing 'leap_year.py'. The main editor area displays the following Python code:

```

1 def is_leap_year(year):
2     """It is divisible by 400, or
3         - It is divisible by 4 but NOT by 100
4
5     Args:
6         year (int): The year to check
7
8     Returns:
9         bool: True if the year is a leap year, False otherwise
10
11 Examples:
12     >>> is_leap_year(2000)
13     True
14     >>> is_leap_year(1900)
15     False
16     >>> is_leap_year(2024)
17     True
18
19     """
20     return (year % 400 == 0) or (year % 4 == 0 and year % 100 != 0)
21
22
23
24
25
26 def main():
27     """Main function to test the leap year checker."""
28     try:
29         year = int(input("Enter a year: "))
30         if is_leap_year(year):
31             print(f"{year} is a leap year.")
32         else:
33             print(f"{year} is not a leap year.")
34     except ValueError:
35         print("Invalid input. Please enter a valid year (integer).")
36
37
38 if __name__ == "__main__":
39     main()

```

To the right of the code, a 'Python leap year check' panel provides feedback and suggestions:

- Now +40 - Auto**
- Optimize the leap year program and convert it into a reusable function.**
- statement:**

```
return (year % 400 == 0) or (year % 4 == 0 and
```

Below this, numbered steps explain the changes:

- More efficient:** Uses a single boolean expression instead of multiple conditional checks
- Better documentation:** Added examples to the docstring showing usage
- Reusable function:** The `is_leap_year()` function can now be easily imported and used in other modules:

```
from leap_year import is_leap_year
if is_leap_year(2024):
    print("It's a leap year!")
```

A note at the bottom explains the refactored code's logic:

The function maintains the same logic (divisible by 400, or divisible by 4 but not by 100) but is more concise and efficient. The `main()` function remains for testing purposes when running the script directly, but the core function is now optimized and ready for reuse in other projects.

At the bottom of the interface, status bars show 'Cursor Tab', 'Screen Reader Optimized', 'Ln 23 Col 1', 'Spaces: 4', 'UTF-8', 'CR LF', 'Python 3.13.9 64-bit (Microsoft Store)', and a review button.

Sample Input / Output

Input

2024

Output

Leap Year

Comparison

- Version 2 is reusable and modular.
- Cursor AI effectively refactored logic without altering correctness.
- Conditional expression improves readability.

Task 4: Student Logic + AI Refactoring (Odd/Even Sum)

Scenario

Company policy requires developers to write logic before using AI.

Original Student Code

`t = (1, 2, 3, 4, 5, 6)`

`even_sum = 0`

`odd_sum = 0`

```
for i in t:  
    if i % 2 == 0:  
        even_sum += i  
    else:  
        odd_sum += i  
  
print("Even Sum:", even_sum)  
print("Odd Sum:", odd_sum)
```

Refactored Code Using AI

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

```
t = (1, 2, 3, 4, 5, 6)  
even, odd = sum_even_odd(t)  
  
print("Even Sum:", even)  
print("Odd Sum:", odd)
```

Improvements After Refactoring

- Introduced function for reusability
 - Reduced number of lines
 - Used Python generator expressions
 - Improved readability and maintainability
-

Overall Tool Comparison Summary

Tool	Strengths	Best Use Case
Google Gemini	Clean code, strong explanations	Learning, data analysis
GitHub Copilot	Fast inline suggestions	Continuous development
Cursor AI	Refactoring, optimization	Code quality improvement

Final Conclusion

This lab demonstrated that different AI coding tools excel in different aspects of software development. Gemini is excellent for understanding and learning, Copilot enhances development speed, and Cursor AI significantly improves code structure and maintainability.