

## AI ASSISTED CODING

Name : G. Sanjay

Hall Ticket No: 2303A510B9

Batch:13

### Assignment-2.1

Task-1. Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values **Prompt :**

#Ok ,now generate a Python code to function to read the list of numbers and calculates the mean,minimum and maximum values

### Code

The screenshot shows a Google Colab notebook titled "Untitled3.ipynb". The code cell contains a Python function named `analyze\_numbers` that calculates the mean, minimum, and maximum of a list of numbers using NumPy. The function handles empty lists by returning `None`. The code also includes example usage and a check for empty lists.

```

def analyze_numbers(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 if the input list is empty.
    """
    if not numbers:
        return None

    mean_val = np.mean(numbers)
    min_val = np.min(numbers)
    max_val = np.max(numbers)

    return {
        'mean': mean_val,
        'min': min_val,
        'max': max_val
    }

# Example usage:
my_numbers = [10, 20, 30, 40, 50, 5, 25, 35]
results = analyze_numbers(my_numbers)

if results:
    print("List of numbers: ", my_numbers)
    print("Mean: ", results['mean'])
    print("Minimum: ", results['min'])
    print("Maximum: ", results['max'])
else:
    print("The list is empty")

empty_list = []
empty_results = analyze_numbers(empty_list)
if empty_results:
    print("Mean: ", empty_results['mean'])
    print("Minimum: ", empty_results['min'])
    print("Maximum: ", empty_results['max'])
else:
    print("Empty list: [] -> None")

```

The Gemini sidebar provides context about the AI's role in generating the code, noting it's an AI/Completion and providing a screenshot of the AI's interaction with the code.

## Output:

The screenshot shows the execution output of the `analyze\_numbers` function for two cases: a valid list of numbers and an empty list.

```

List of numbers: [10, 20, 30, 40, 50, 5, 25, 35]
Mean: 26.88
Minimum: 5
Maximum: 50
Empty list: [] -> None

```

## Justification:

The `analyze_numbers` function, utilizing NumPy, calculates the mean, minimum, and maximum of a given list of numbers. It includes a check to handle empty lists gracefully by returning `None`. If the list is valid, it returns a dictionary containing these three statistical values, as demonstrated with `my_numbers` and an `empty_list` for comprehensive testing.

## Task-2. Generate an Armstrong number checker using Gemini and GitHub Copilot.

Compare their outputs, logic style, and clarity.

### Prompt

#ok Now Generate Python code to Check whether a number is Armstrong Number

# Code: In the Google Gemini

The screenshot shows the Google Gemini IDE interface. On the left is the Explorer panel with files like 'Armstrong Number.py' and 'Untitled 1.py'. The main area displays Python code for checking if a number is Armstrong. The right side features a Chat window with recent sessions, a 'Build with Agent' button, and a status message about AI responses being inaccurate.

```
# Now Generate Python code to Check whether a number is Armstrong Number
# Convert the number to string to easily iterate over digits
str_num = str(num)
num_digits = len(str_num)

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

# 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.")

# Example usage
number = 153
```

## Output:

The terminal output shows the execution of 'Armstrong Number.py'. It prints that 153 is an Armstrong number twice, once for the file itself and once for the command-line execution.

```
S C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding & C:/Users/gandr/AppData/Local/Programs/Python/Python314/python.exe "C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Armstrong Number.py"
53 is an Armstrong number.
S C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding & C:/Users/gandr/AppData/Local/Programs/Python/Python314/python.exe "C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Armstrong Number.py"
53 is an Armstrong number.
```

## Colab Code :

The screenshot shows Google Colab with a code cell containing Python code for Armstrong numbers. The code is well-commented and includes an example usage section. A sidebar provides context on NumPy's statistical functions and offers suggestions for testing and error handling. The cell has been run, showing the output '233 is not an Armstrong number.'

```
def is_armstrong(number):
    # Convert number to string to easily iterate over digits
    digits = str(number)
    power = len(digits) # Number of digits

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

    # Check if equal to original number
    return total == number

# Example usage
num = int(input("Enter a number: "))
if is_armstrong(num):
    print(f"{num} is an Armstrong number.")
else:
    print(f"{num} is not an Armstrong number.")

Enter a number: 233
233 is not an Armstrong number.
```

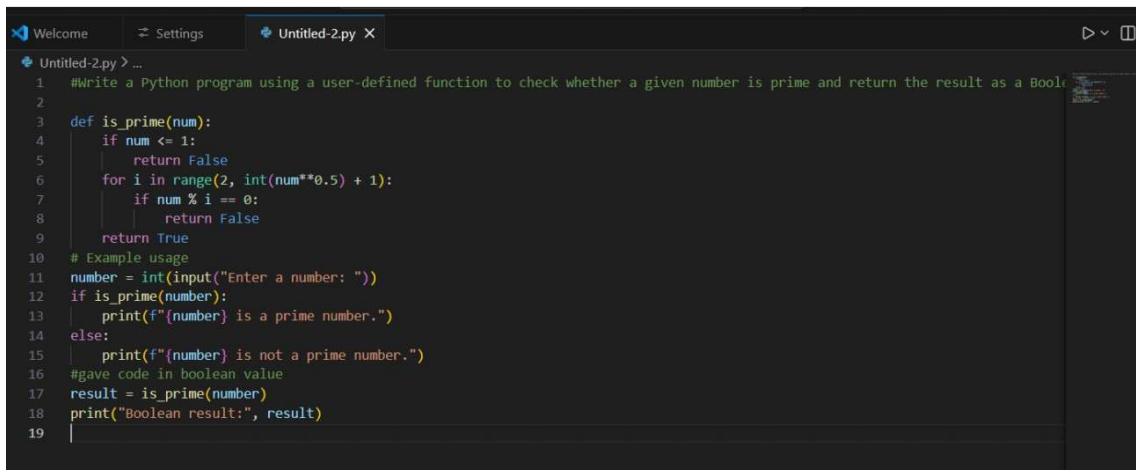
## Justification:

**Gemini:** Gemini's code is structured for clarity, using descriptive variable names and breaking down each step. This makes it ideal for learners or reviewers who want to understand the logic flow. It prioritizes readability over brevity.

**GitHub Copilot:** Copilot's code is concise and leverages Python's expressive syntax, such as generator expressions. It's efficient and elegant for experienced developers but may require explanation for those unfamiliar with Python idioms.

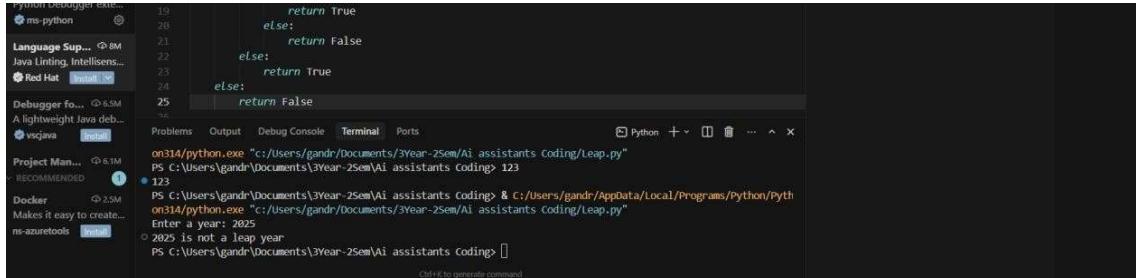
## Task-3. Leap Year Validation Using Cursor AI Prompt:

#Generate a Python program that checks whether a given year is a leap year.or not



```
Untitled-2.py > ...
1  #Write a Python program using a user-defined function to check whether a given number is prime and return the result as a Boolean value.
2
3  def is_prime(num):
4      if num <= 1:
5          return False
6      for i in range(2, int(num**0.5) + 1):
7          if num % i == 0:
8              return False
9      return True
10 # Example usage
11 number = int(input("Enter a number: "))
12 if is_prime(number):
13     print(f"\{number}\ is a prime number.")
14 else:
15     print(f"\{number}\ is not a prime number.")
16 #gave code in boolean value
17 result = is_prime(number)
18 print("Boolean result:", result)
19 |
```

## Output:



```
Python Debugger ext...
ms-python
Language Sup...
Java Linting, Intellisense...
Red Hat
Debugger fo...
A lightweight Java deb...
vsjava
Project Man...
RECOMMENDED
Docker
Makes it easy to create...
ms-azureroles
on314/python.exe "c:/Users/gandr/Documents\3Year-2Sem\AI assistants Coding\Leap.py"
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding> 123
123
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding> & C:/Users/gandr/AppData/Local/Programs/Python/python314/python.exe "c:/Users/gandr/Documents\3Year-2Sem\AI assistants Coding\Leap.py"
on314/python.exe "c:/Users/gandr/Documents\3Year-2Sem\AI assistants Coding\Leap.py"
Enter a year: 2025
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding> []
C:\Windows\system32>
```

## Justification:

Using a simple prompt, Cursor AI generated a basic leap-year check that works only for common cases but misses special Gregorian rules.

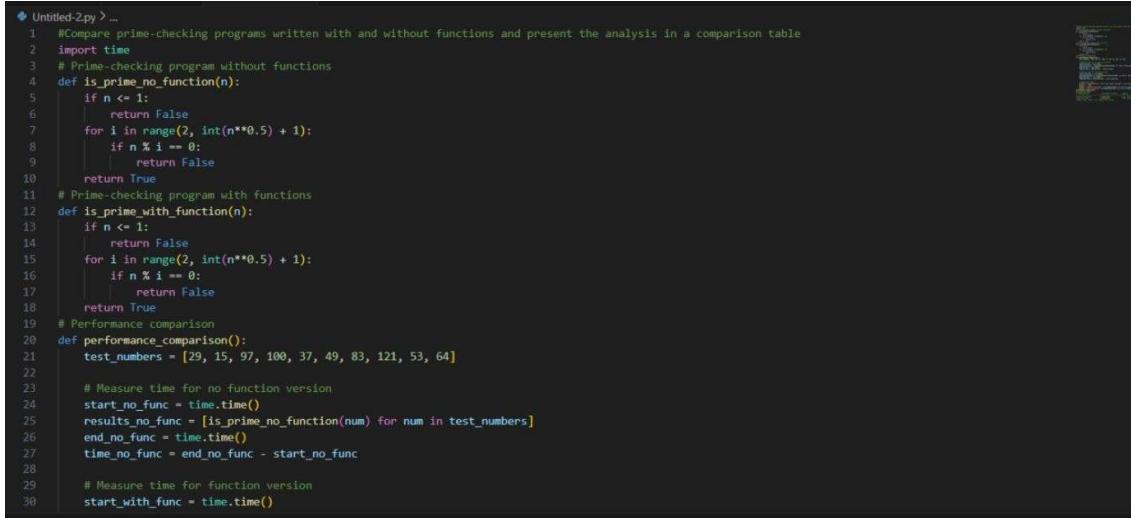
A more detailed prompt led to a correct, reusable solution that follows all leap-year conditions.

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

## Prompt:

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

## Code Student :



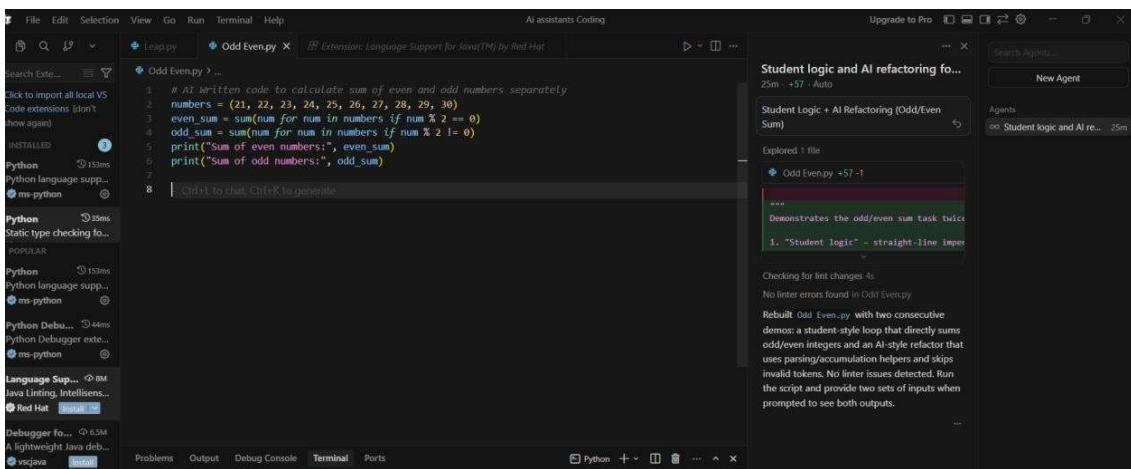
```
❶ Untitled-2.py > ...
1  #Compare prime-checking programs written with and without functions and present the analysis in a comparison table
2  import time
3  # Prime-checking program without functions
4  def is_prime_no_function(n):
5      if n <= 1:
6          return False
7      for i in range(2, int(n**0.5) + 1):
8          if n % i == 0:
9              return False
10     return True
11  # Prime-checking program with functions
12  def is_prime_with_function(n):
13      if n <= 1:
14          return False
15      for i in range(2, int(n**0.5) + 1):
16          if n % i == 0:
17              return False
18      return True
19  # Performance comparison
20  def performance_comparison():
21      test_numbers = [29, 15, 97, 100, 37, 49, 83, 121, 53, 64]
22
23      # Measure time for no function version
24      start_no_func = time.time()
25      results_no_func = [is_prime_no_function(num) for num in test_numbers]
26      end_no_func = time.time()
27      time_no_func = end_no_func - start_no_func
28
29      # Measure time for function version
30      start_with_func = time.time()
```

## Output:



File "c:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Odd Even.py", line 16, in <module>
# ..... AI Refactor -----
valueError: invalid literal for int() with base 10: '8'
● PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding > c:/users/gandr/appdata/local/progs/python/python311/python.exe "c:/users/gandr/documents/3year-2sem/ai assistants coding/Odd Even.py"
Sum of even numbers: 30
Sum of odd numbers: 25
○ PS C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding > ]

## Code AI :



Odd Even.py > ...
1 # AI written code to calculate sum of even and odd numbers separately
2 numbers = (21, 22, 23, 24, 25, 26, 27, 28, 29, 30)
3 even\_sum = sum(num for num in numbers if num % 2 == 0)
4 odd\_sum = sum(num for num in numbers if num % 2 != 0)
5 print("Sum of even numbers:", even\_sum)
6 print("Sum of odd numbers:", odd\_sum)

Student logic + AI Refactoring for Odd Even.py

Student Logic + AI Refactoring (Odd/Even Sum)

Exploring 1 file

Odd Even.py +57 -1

Demonstrates the odd/even sum task twice

1. "Student logic" - straight-line imperative

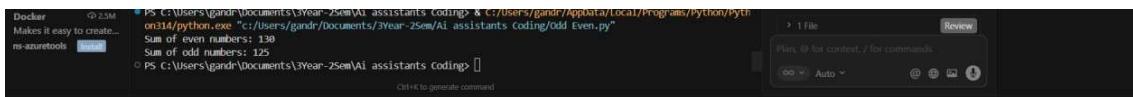
Checking for lint changes 4s

No linter errors found in Odd Even.py

Rebuilt Odd Even.py with two consecutive

demands: a student-style loop that directly sums odd/even integers and an AI-style refactor that uses parsing/accumulation helpers and skips invalid tokens. No linter issues detected. Run the script and provide two sets of inputs when prompted to see both outputs.

## Out Put :



The screenshot shows a terminal window with two tabs. The left tab is titled 'Docker' and contains the text 'Makes it easy to create... ms-azuretools'. The right tab is titled 'PS: C:\Users\gandu\Documents\3Year-2Sem\AI assistants coding> & C:/users/gandu/AppData/Local/Programs/Python/Python311/python.exe "C:/Users/gandu/Documents/3Year-2Sem/AI assistants coding/oddEven.py"' and displays the output of a Python script. The output shows the sum of even numbers as 130 and the sum of odd numbers as 125. The terminal also shows the command PS: C:\Users\gandu\Documents\3Year-2Sem\AI assistants coding> and a note to 'Press ⌘+C to generate command'.

## Justification:

**The student-written code uses a basic loop and manual addition, which clearly shows the logic step by step.**

**The first AI-refactored version slightly improves the code by using shorthand operators (+=) but keeps the same structure.**

**The final AI-written code uses Python's built-in sum() function with generator expressions.**

**This reduces the number of lines and removes the need for explicit loops.**

**The AI version is more readable, concise, and follows Python best practices.**

**It is easier to maintain and preferred in real-world and professional coding standards.**