

AI ASSISTED CODING

Name : G. Sanjay

Hall Ticket No: 2303A510B9

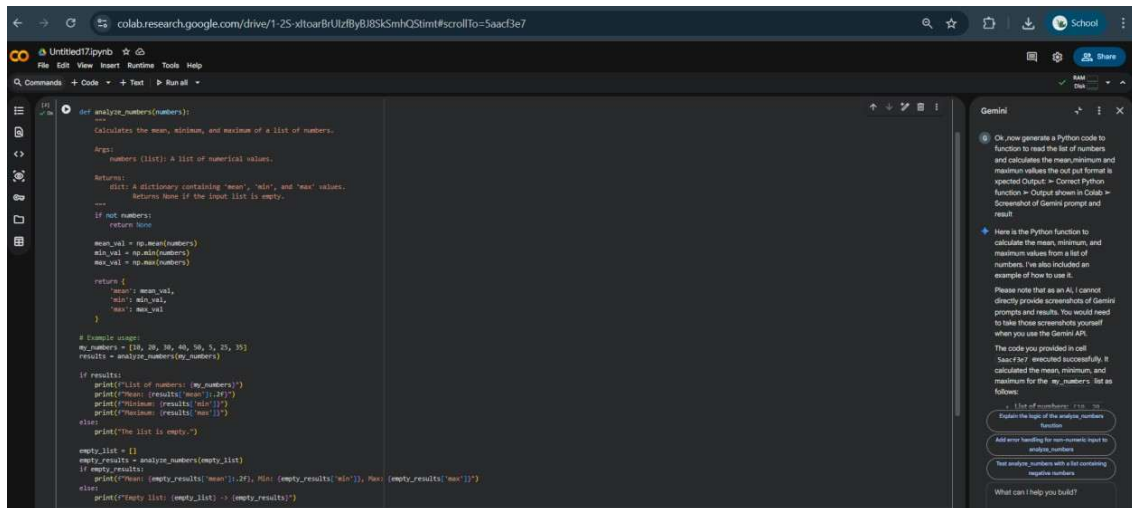
Batch:13

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



```
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)

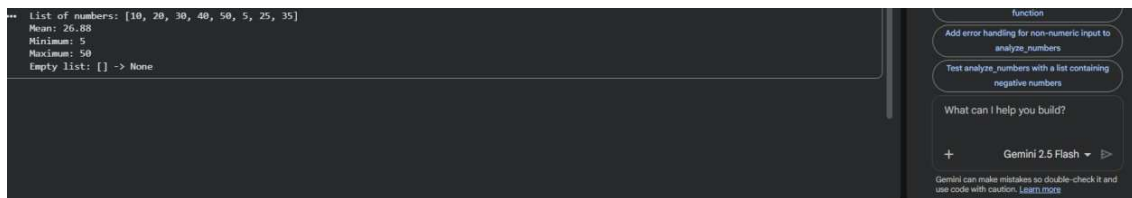
    return {
        'mean': mean_val,
        'min': min_val,
        'max': max_val
    }

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

if results:
    print(f"List of numbers: {my_numbers}")
    print(f"Mean: {results['mean']:.2f}")
    print(f"Minimum: {results['min']}")
    print(f"Maximum: {results['max']}")
else:
    print("The list is empty.")

empty_list = []
empty_results = analyze_numbers(empty_list)
if empty_results:
    print(f"Mean: {empty_results['mean']:.2f}, Min: {empty_results['min']}, Max: {empty_results['max']}")
else:
    print(f"Empty list: {empty_list} -> {empty_results}")
```

Output:



```
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.

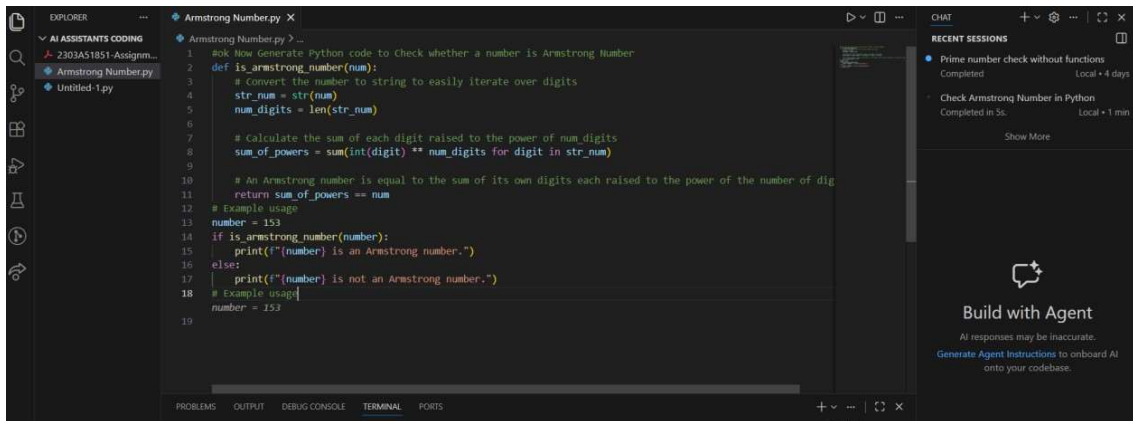
Task-2. Generate an Armstrong number checker using Gemini and GitHub Copilot.

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 a code editor with a file named 'Armstrong Number.py'. The code is a Python function 'is_armstrong_number(num)' that checks if a number is an Armstrong number. It converts the number to a string, iterates over its digits, calculates the sum of each digit raised to the power of the number of digits, and compares the result to the original number. Comments explain each step. The right sidebar shows 'RECENT SESSIONS' and a 'Build with Agent' button.

```
1 # Ask Now Generate Python code to Check whether a number is Armstrong Number
2 def is_armstrong_number(num):
3     # Convert the number to string to easily iterate over digits
4     str_num = str(num)
5     num_digits = len(str_num)
6
7     # Calculate the sum of each digit raised to the power of num_digits
8     sum_of_powers = sum(int(digit) ** num_digits for digit in str_num)
9
10    # An Armstrong number is equal to the sum of its own digits each raised to the power of the number of dig
11    return sum_of_powers == num
12
13    # Example usage
14    number = 153
15    if is_armstrong_number(number):
16        print(f"{number} is an Armstrong number.")
17    else:
18        print(f"{number} is not an Armstrong number.")
19
20    # Example usage
21    number = 153
```

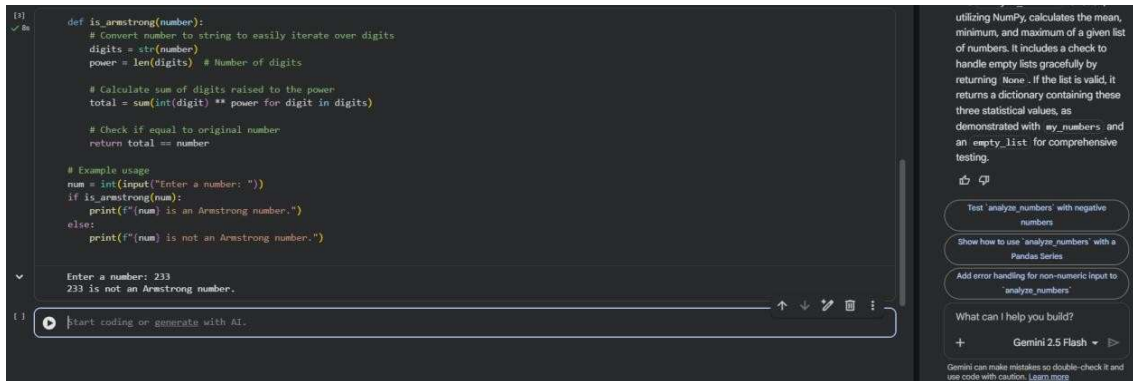
Output:



The screenshot shows a terminal window with the command 'python.exe "C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Armstrong Number.py"' and the output '53 is an Armstrong number.' and '53 is not an Armstrong number.'

```
$ C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding> python.exe "C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Armstrong Number.py"
53 is an Armstrong number.
$ C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding> python.exe "C:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Armstrong Number.py"
53 is not an Armstrong number.
```

Colab Code :



The screenshot shows a Google Colab notebook with a cell containing the same Python code as the previous image. The output of the code is shown below the cell: 'Enter a number: 233' and '233 is not an Armstrong number.' The right sidebar shows a 'What can I help you build?' section with a 'Gemini 2.5 Flash' button.

```
10 def is_armstrong(number):
11     # Convert number to string to easily iterate over digits
12     digits = str(number)
13     power = len(digits) # Number of digits
14
15     # Calculate sum of digits raised to the power
16     total = sum(int(digit) ** power for digit in digits)
17
18     # Check if equal to original number
19     return total == number
20
21 # Example usage
22 num = int(input("Enter a number: "))
23 if is_armstrong(num):
24     print(f"{num} is an Armstrong number.")
25 else:
26     print(f"{num} is not an Armstrong number.")
27
28 Enter a number: 233
29 233 is not an Armstrong number.
```

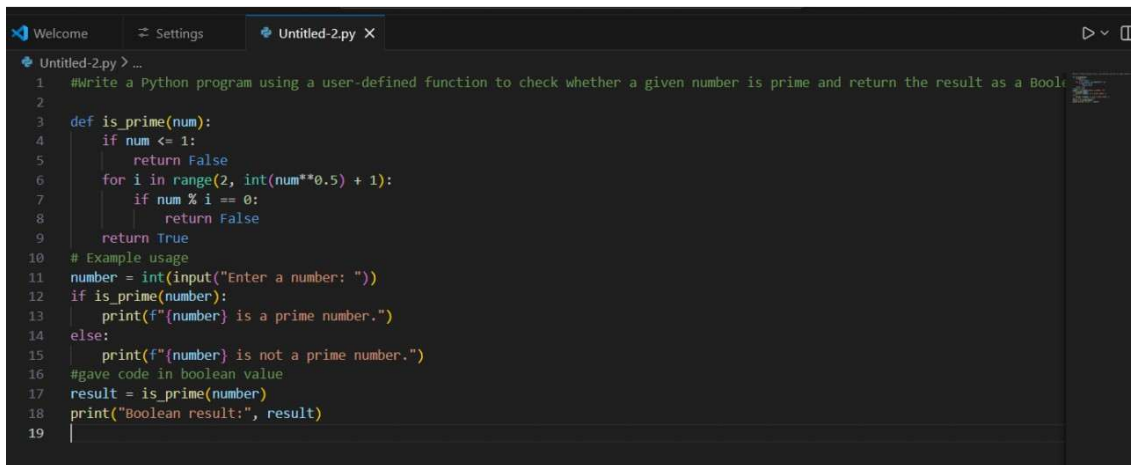
Justification:

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.

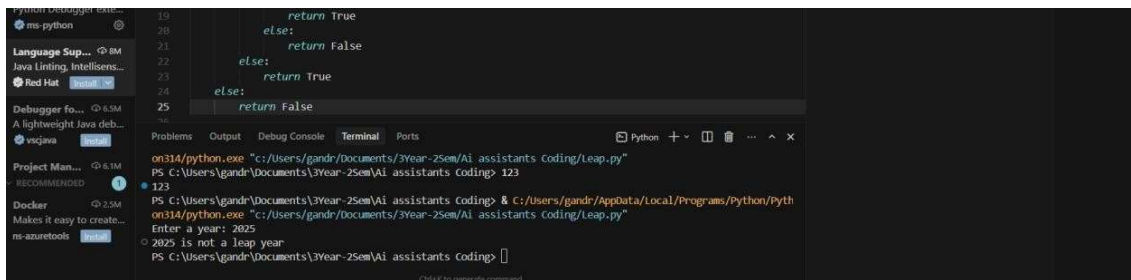
Task-3. Leap Year Validation Using Cursor AI Prompt:

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



```
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
11 # Example usage
12 number = int(input("Enter a number: "))
13 if is_prime(number):
14     print(f"{number} is a prime number.")
15 else:
16     print(f"{number} is not a prime number.")
17 #gave code in boolean value
18 result = is_prime(number)
19 print("Boolean result:", result)
```

Output:



```
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"
Enter a year: 2025
2025 is not a leap year
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants coding>
```

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.

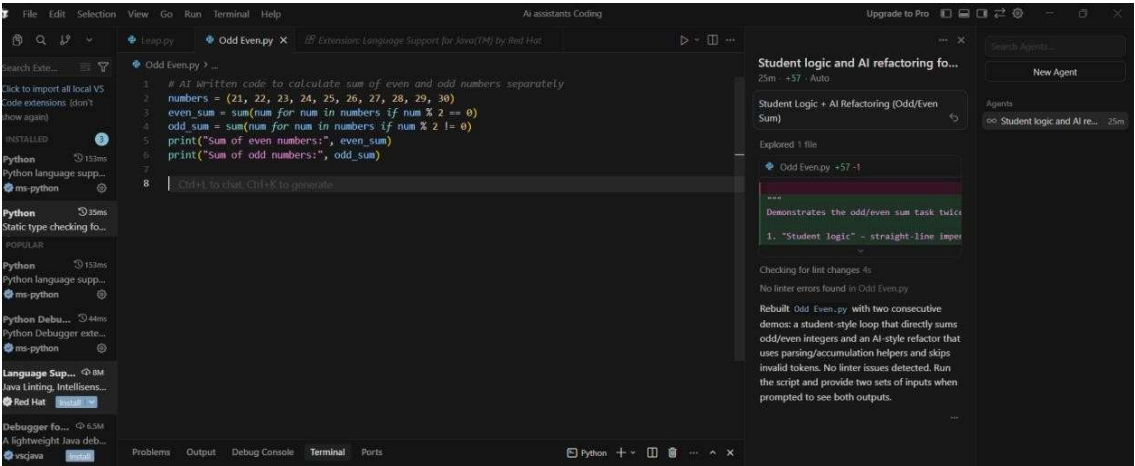
Code Student :

```
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:



Code AI :



Out Put :



```
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding> cd "C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding\Odd Even.py"
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding\Odd Even.py> python.exe
Sum of even numbers: 138
Sum of odd numbers: 125
```

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.