Lab Assignment 2.1

Name: G.Hasini

Hallticket:2303A51099

Batch:02

Task 1: Statistical Summary for Survey Data

Scenario:

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

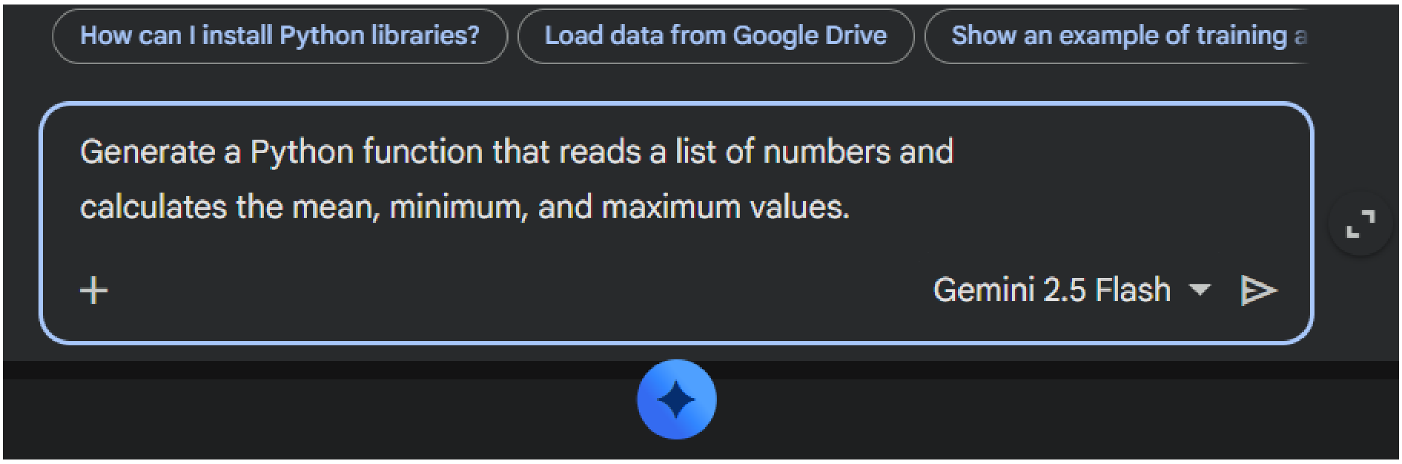
Task:

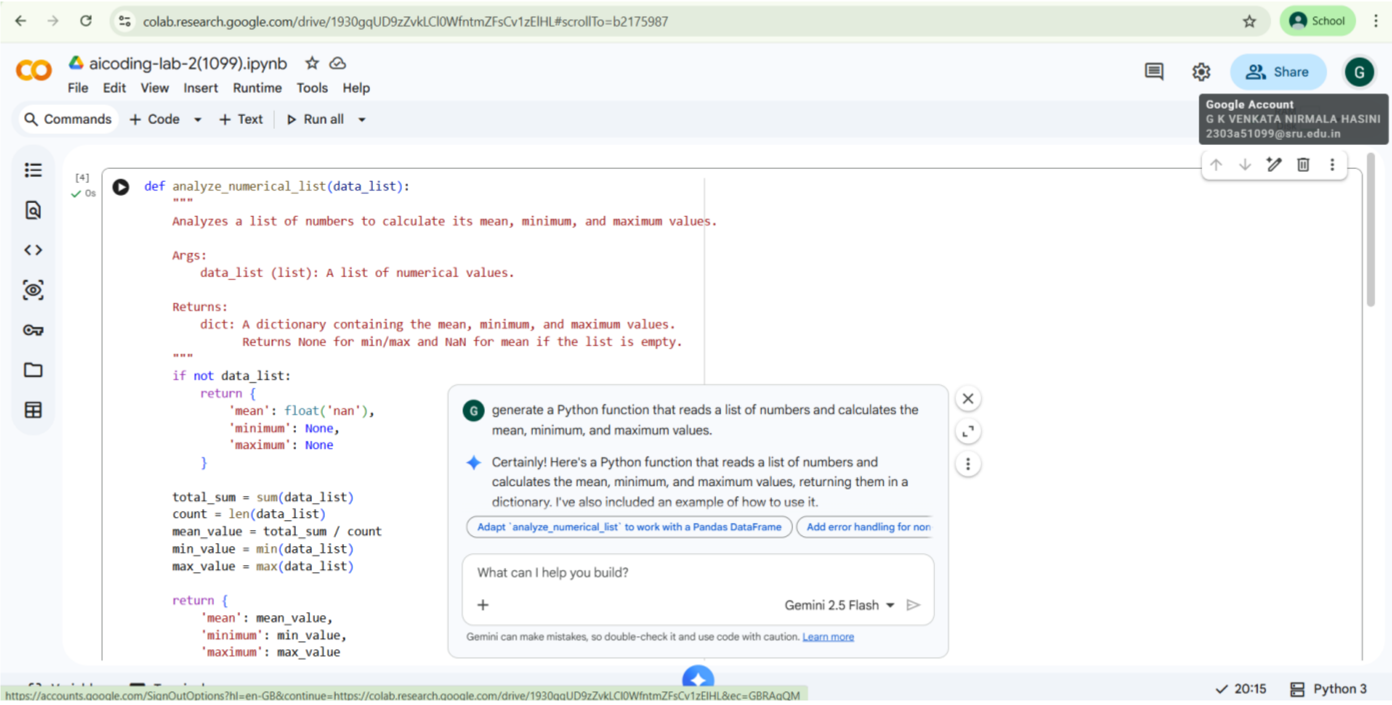
- Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

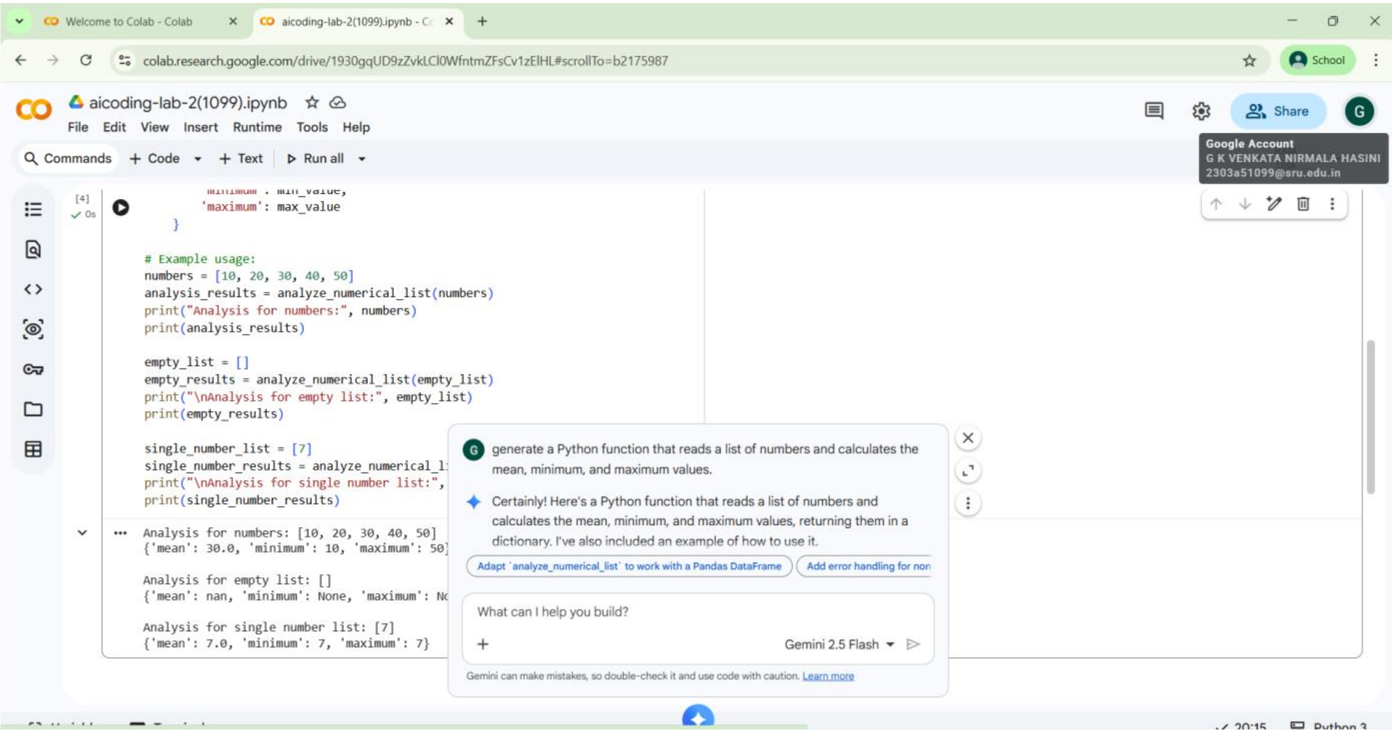
Expected Output:

- Correct Python function
- Output shown in Colab
- Screenshot of Gemini prompt and result

Google Colab Workspace Screenshots:







Code:

```
def analyze_numerical_list(data_list):
    """
    Analyzes a list of numbers to calculate its mean, minimum, and maximum values.

    Args:
        data_list (list): A list of numerical values.
```

```
    Returns:
        dict: A dictionary containing the mean, minimum, and maximum values.
              Returns None for min/max and NaN for mean if the list is empty.
    """
    if not data_list:
        return {
            'mean': float('nan'),
            'minimum': None,
            'maximum': None
        }

    total_sum = sum(data_list)
    count = len(data_list)
    mean_value = total_sum / count
    min_value = min(data_list)
    max_value = max(data_list)

    return {
        'mean': mean_value,
        'minimum': min_value,
        'maximum': max_value
    }
```

output:

```
# Example usage 1:
numbers = [10, 20, 30, 40, 50]
analysis_results = analyze_numerical_list(numbers)
print("Analysis for numbers:", numbers)
print(analysis_results)


Analysis for numbers: [10, 20, 30, 40, 50]
{'mean': 30.0, 'minimum': 10, 'maximum': 50}
```

```
# Example usage 2:

empty_list = []

empty_results = analyze_numerical_list(empty_list)

print("\nAnalysis for empty list:", empty_list)

print(empty_results)

Analysis for empty list: []

{'mean': nan, 'minimum': None, 'maximum': None}

# Example usage 3:

single_number_list = [7]

single_number_results = analyze_numerical_list(single_number_list)

print("\nAnalysis for single number list:", single_number_list)

print(single_number_results)

Analysis for single number list: [7]

{'mean': 7.0, 'minimum': 7, 'maximum': 7}
```

Task 2: Armstrong Number – AI Comparison

Scenario:

- You are evaluating AI tools for numeric validation logic.

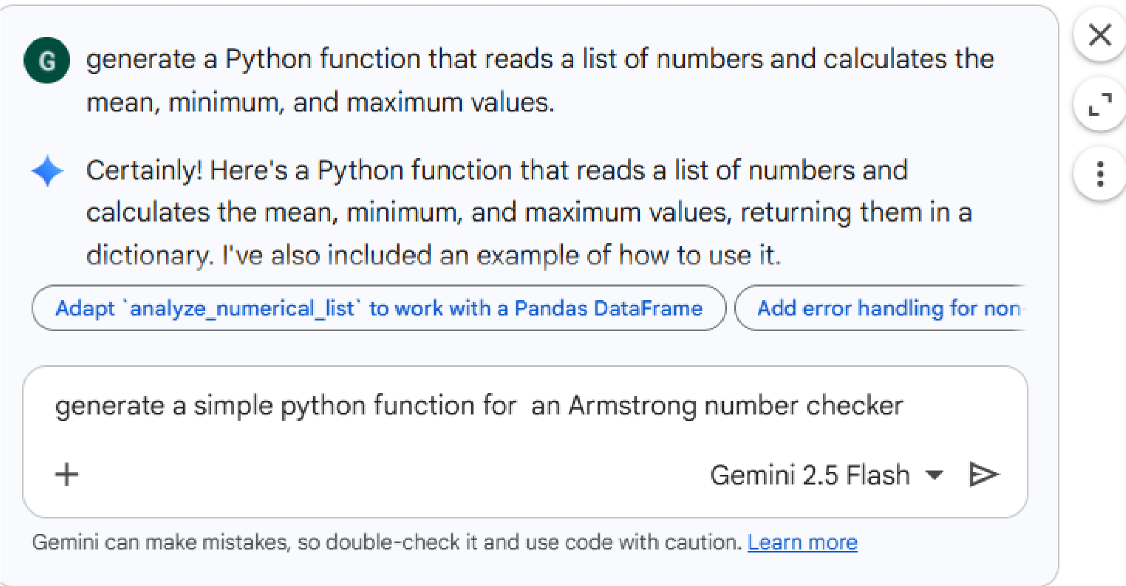
Task:

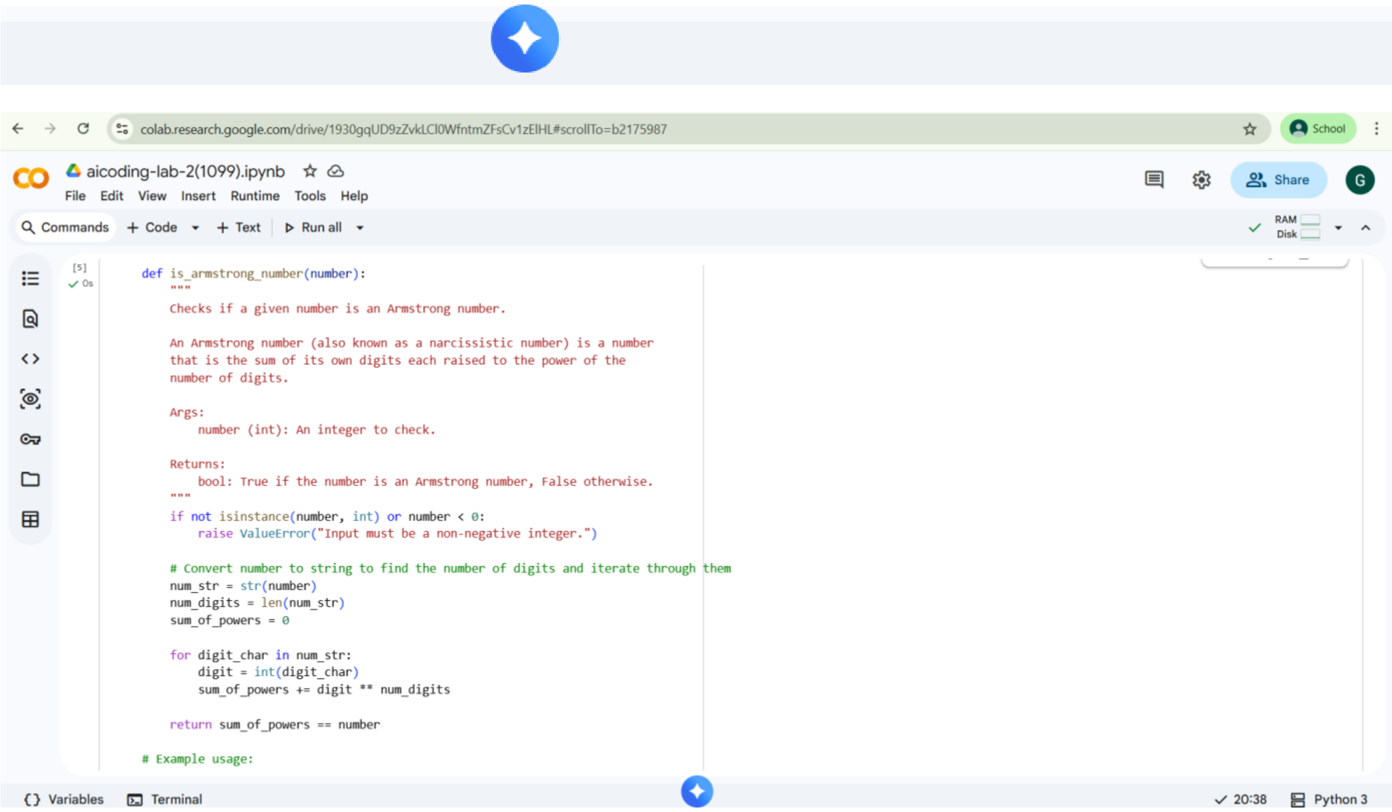
- Generate an Armstrong number checker using Gemini and GitHub Copilot.
- Compare their outputs, logic style, and clarity.

Expected Output:

- Side-by-side comparison table
- Screenshots of prompts and generated code

Google Colab (using gemini) Screenshots:





def is_armstrong_number(number):

"""

Checks if a given number is an Armstrong number.

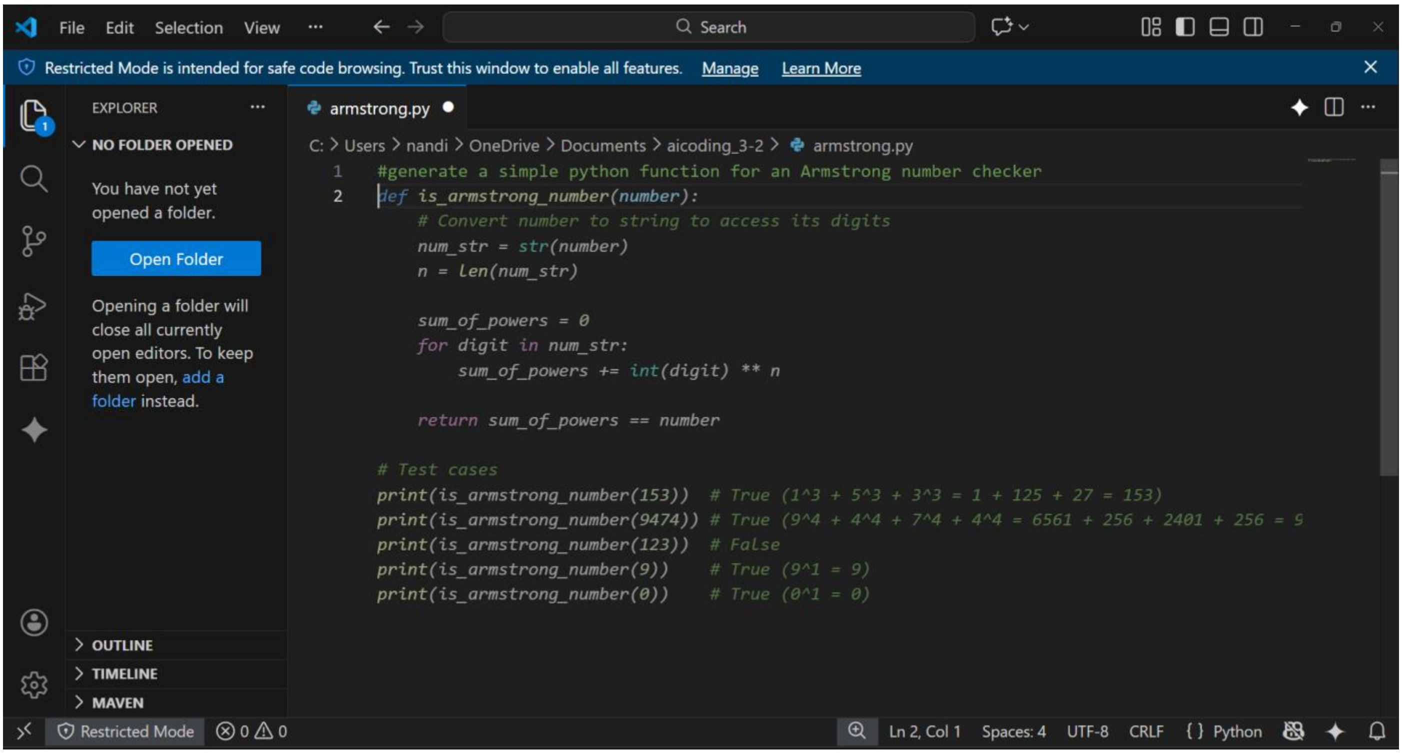
An Armstrong number (also known as a narcissistic number) is a number

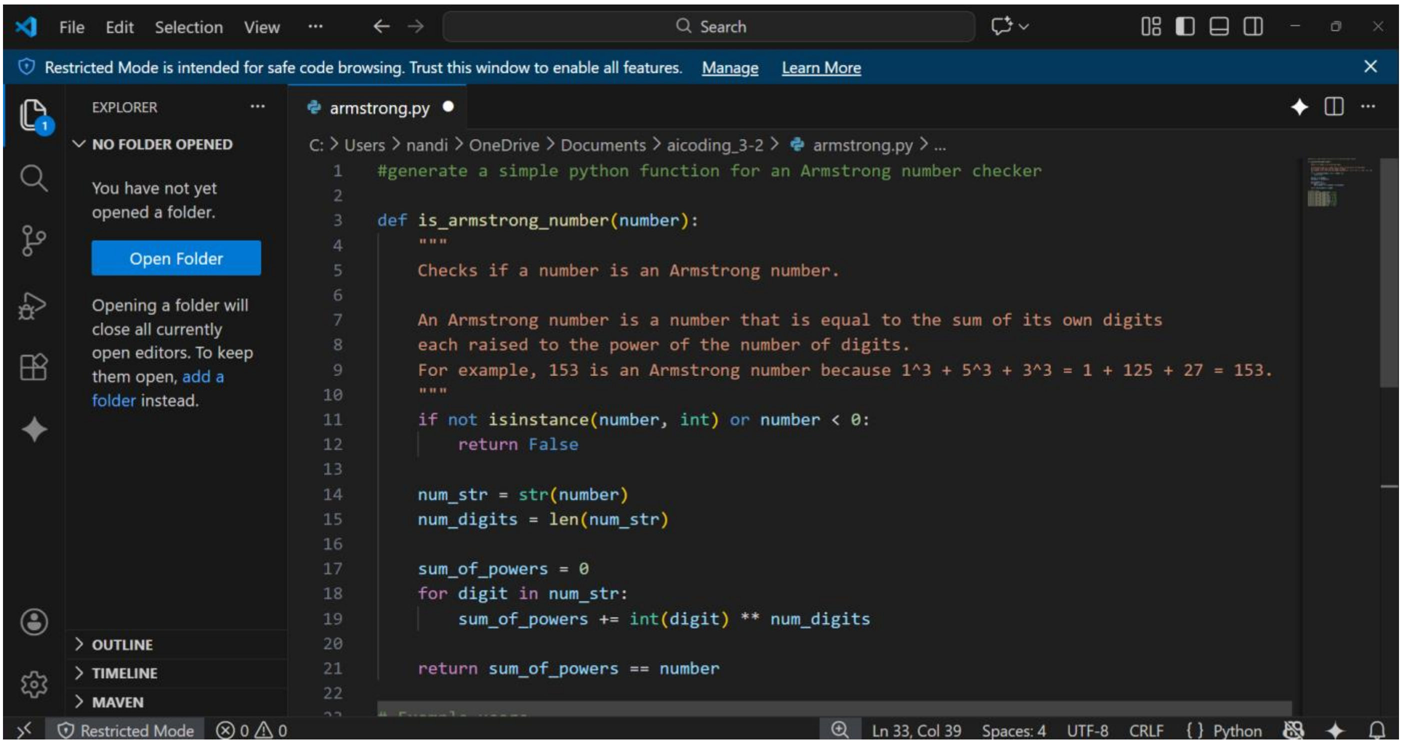
that is the sum of its own digits each raised to the power of the

number of digits.

```
    Args:
        number (int): An integer 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.")

    # Convert number to string to find the number of digits and iterate through them
    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)}")  # output: True
print(f"Is 9474 an Armstrong number? {is_armstrong_number(9474)}")  # output: True
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}")    # output: True
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}")   # output: False
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}")  # output : False
```

Vs code :





Code:

```
#Write a Python function to check if a number is an Armstrong number

def is_armstrong(number):

    num_str = str(number)  # Convert the number to string to easily iterate over digits

    num_digits = len(num_str)  # Get the number of digits
```

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

return sum_of_powers == number # Check if the sum of powers is equal to the original number

Take input from the user

num = int(input("Enter a number to check if it is an Armstrong number: "))

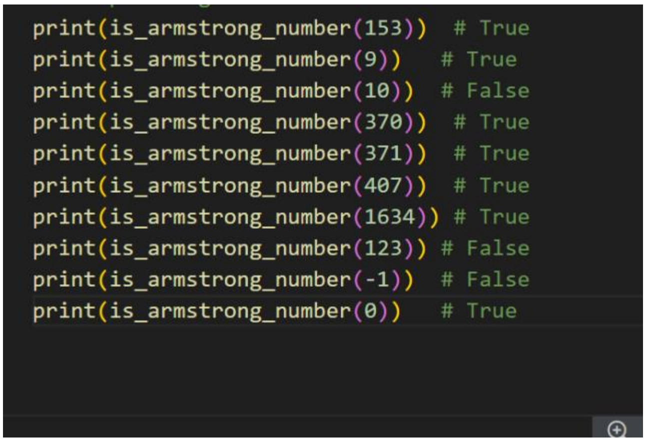
if is_armstrong(num):

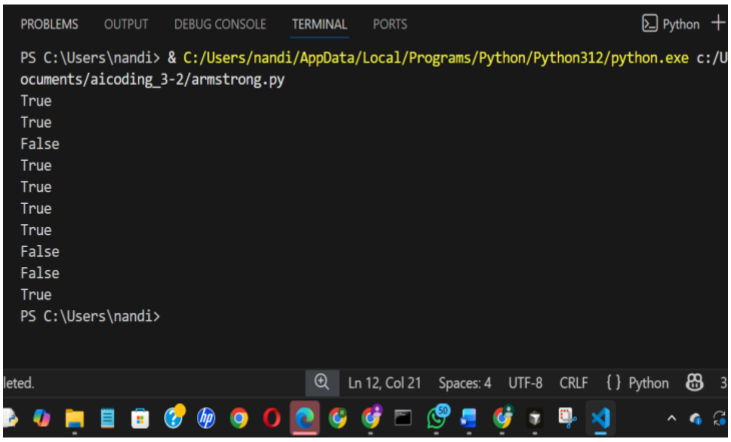
print(f"{num} is an Armstrong number.")

else:

print(f"{num} is not an Armstrong number.")

output:





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

Original Code written by me:

armstrong.py   # sumofEvenAndOdd.py

C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > # sumofEvenAndOdd.py > sum_of_even_and_odd

```python
# Python program that calculates the sum of odd and even numbers in a tuple
def sum_of_even_and_odd(numbers):
    if not isinstance(numbers, tuple):
        raise ValueError("Input must be a tuple.")

    sum_even = 0
    sum_odd = 0

    for num in numbers:
        if not isinstance(num, int):
            raise ValueError("All elements in the tuple must be integers.")
        if num % 2 == 0:
            sum_even += num
        else:
            sum_odd += num

    return {'even_sum': sum_even, 'odd_sum': sum_odd}

```

Code:

```python
def sum_of_even_and_odd(numbers):
    if not isinstance(numbers, tuple):
        raise ValueError("Input must be a tuple.")


    sum_even = 0
    sum_odd = 0


    for num in numbers:
        if not isinstance(num, int):
            raise ValueError("All elements in the tuple must be integers.")
        if num % 2 == 0:
            sum_even += num
        else:
            sum_odd += num


    return {'even_sum': sum_even, 'odd_sum': sum_odd}
```

Output:

Example usage 1

numbers_tuple = (1, 2, 3, 4, 5, 6)

result = sum_of_even_and_odd(numbers_tuple)

print(result)

numbers_tuple = (10, 15, 20, 25, 30)

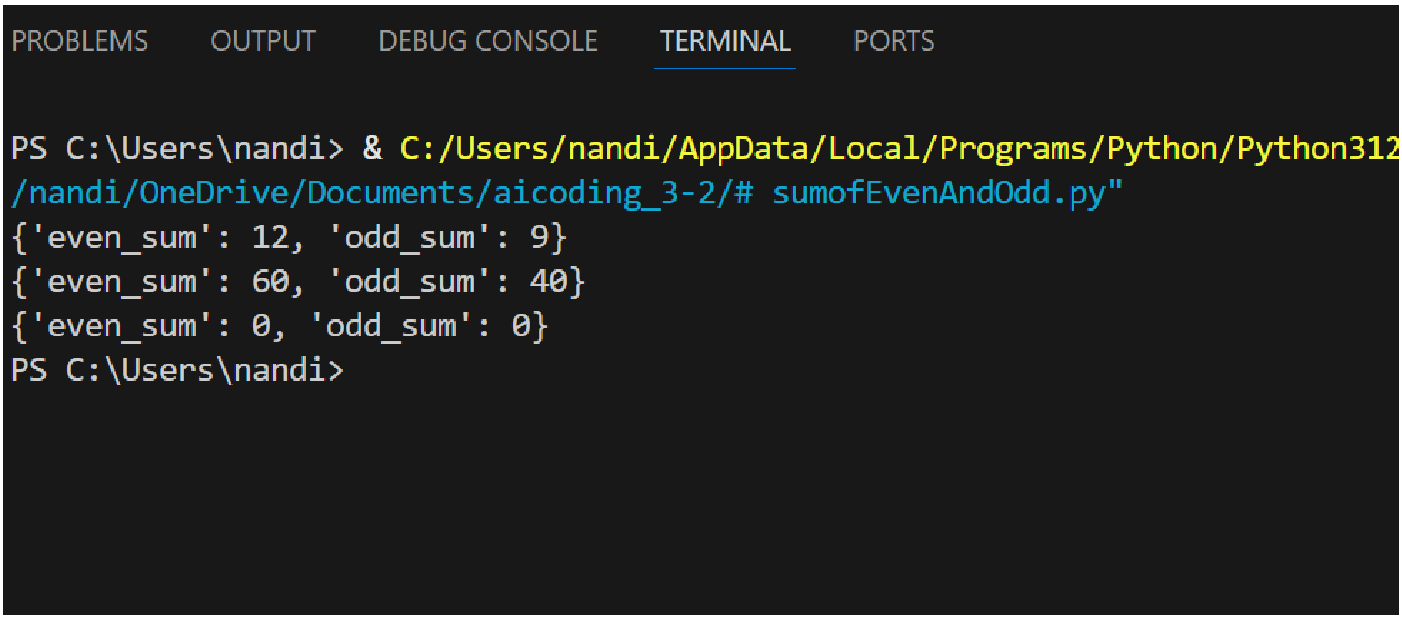
result = sum_of_even_and_odd(numbers_tuple)

print(result)

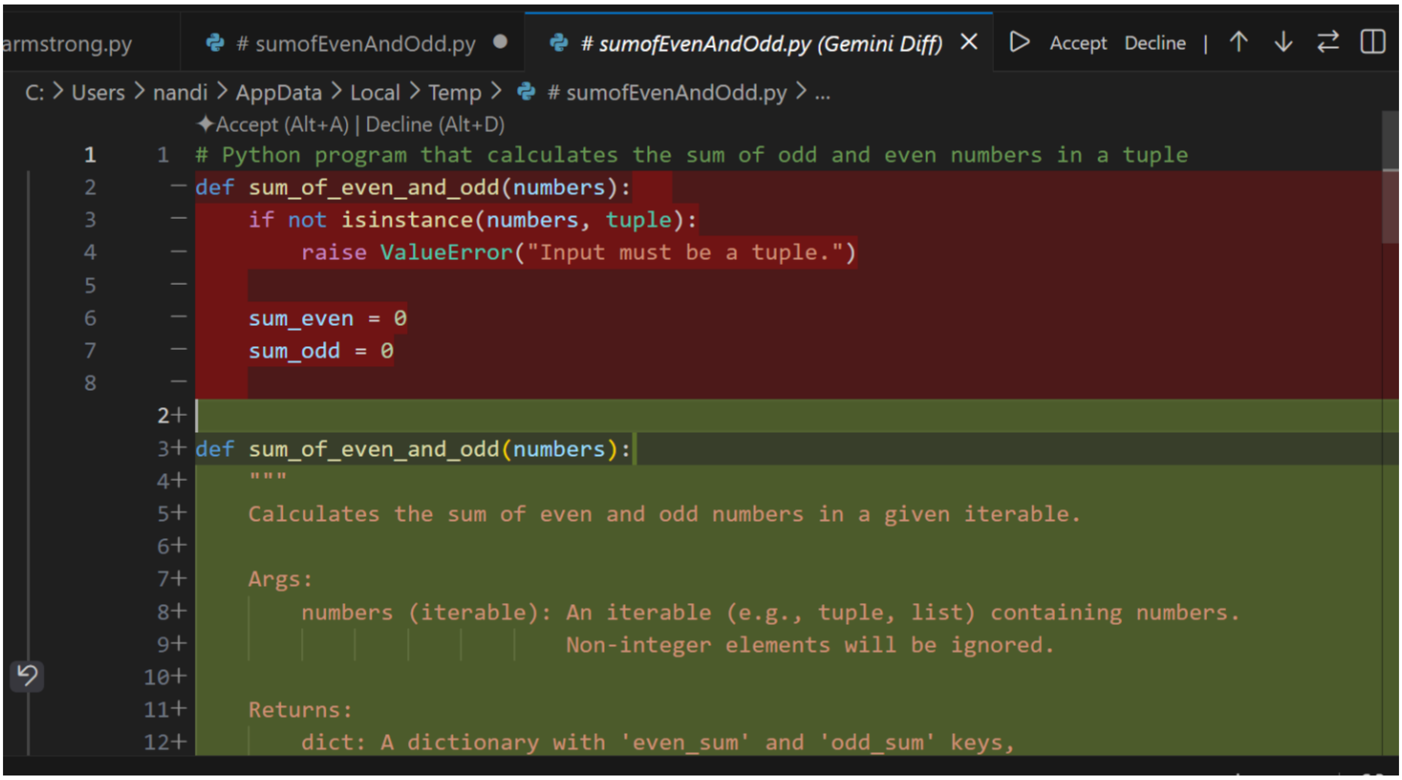
numbers_tuple = ()

result = sum_of_even_and_odd(numbers_tuple)

print(result)



Refactoring the code :



Code

```
def sum_of_even_and_odd(numbers):
    """
    Calculates the sum of even and odd numbers in a given iterable.

    Args:
        numbers (iterable): An iterable (e.g., tuple, list) containing numbers.
                    Non-integer elements will be ignored.

    Returns:
        dict: A dictionary with 'even_sum' and 'odd_sum' keys,
            representing the sum of even and odd numbers respectively.
            Returns {'even_sum': 0, 'odd_sum': 0} for empty or invalid inputs.
    """
    # Initialize sums
    even_sum = 0
    odd_sum = 0
```

```python
    # Handle non-iterable input gracefully
    if not hasattr(numbers, '__iter__'):
        print("Warning: Input is not an iterable. Returning sums as 0.")
        return {'even_sum': 0, 'odd_sum': 0}

    # Iterate through the numbers and categorize them
    for num in numbers:
        # Ensure the element is an integer or can be converted to one
        if isinstance(num, (int, float)):
            # Convert floats to integers for parity check, or handle as needed
            # For this problem, we'll treat floats as their integer part for parity
            num_int = int(num)
            if num_int % 2 == 0:
                even_sum += num_int
            else:
                odd_sum += num_int
        # Optionally, you can log or handle non-numeric types differently
        # else:
        # print(f"Skipping non-numeric element: {num}")

    return {'even_sum': even_sum, 'odd_sum': odd_sum}

# Example usage
numbers_tuple = (1, 2, 3, 4, 5, 6)
result = sum_of_even_and_odd(numbers_tuple)
print(result)  # Output: {'even_sum': 12, 'odd_sum': 9}

numbers_tuple = (10, 15, 20, 25, 30)
result = sum_of_even_and_odd(numbers_tuple)
print(result)  # Output: {'even_sum': 60, 'odd_sum': 40}
```

result = sum_of_even_and_odd(())

print(result) # Output: {'even_sum': 0, 'odd_sum': 0}

result = sum_of_even_and_odd([1, 2, 3, 'a', 4.5]) # Handles list and non-integers

print(result) # Output: {'even_sum': 6, 'odd_sum': 4} (4.5 becomes 4, 'a' is ignored)

result = sum_of_even_and_odd(None) # Handles non-iterable input

print(result) # Output: {'even_sum': 0, 'odd_sum': 0}

C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > # sumofEvenAndOdd.py > ...

```python

# Example usage
numbers_tuple = (1, 2, 3, 4, 5, 6)
result = sum_of_even_and_odd(numbers_tuple)
print(result)  # Output: {'even_sum': 12, 'odd_sum': 9}

numbers_tuple = (10, 15, 20, 25, 30)
result = sum_of_even_and_odd(numbers_tuple)
print(result)  # Output: {'even_sum': 60, 'odd_sum': 40}

result = sum_of_even_and_odd(())
print(result)  # Output: {'even_sum': 0, 'odd_sum': 0}

result = sum_of_even_and_odd([1, 2, 3, 'a', 4.5]) # Handles list and non-integers
print(result) # Output: {'even_sum': 6, 'odd_sum': 4} (4.5 becomes 4, 'a' is ignored)

```

```
PS C:\Users\nandi> & C:/Users/nandi/AppData/Local/Programs/Python/Python312/python.exe "c
/nandi/OneDrive/Documents/aicoding_3-2/# sumofEvenAndOdd.py"
{'even_sum': 12, 'odd_sum': 9}
{'even_sum': 60, 'odd_sum': 40}
{'even_sum': 0, 'odd_sum': 0}
{'even_sum': 6, 'odd_sum': 4}
Warning: Input is not an iterable. Returning sums as 0.
{'even_sum': 0, 'odd_sum': 0}
PS C:\Users\nandi>
```

Explanation of improvements:

Input validation

- Making sure the user enters a valid integer.
- Handling even in case we enter an invalid integer in a tupel and gives valid input only for integers

Readability

- Using clear variable names.
- Adding comments and handling even edge and improper cases

Edge cases

- If input is 0, both sums should be 0.
- If input contains any non-integers then that part of non integer is ignored and the rest of the tuple Is processed in the effective way and produces valid output.