AI ASSISTED CODING

Assignment-2

2403A52107

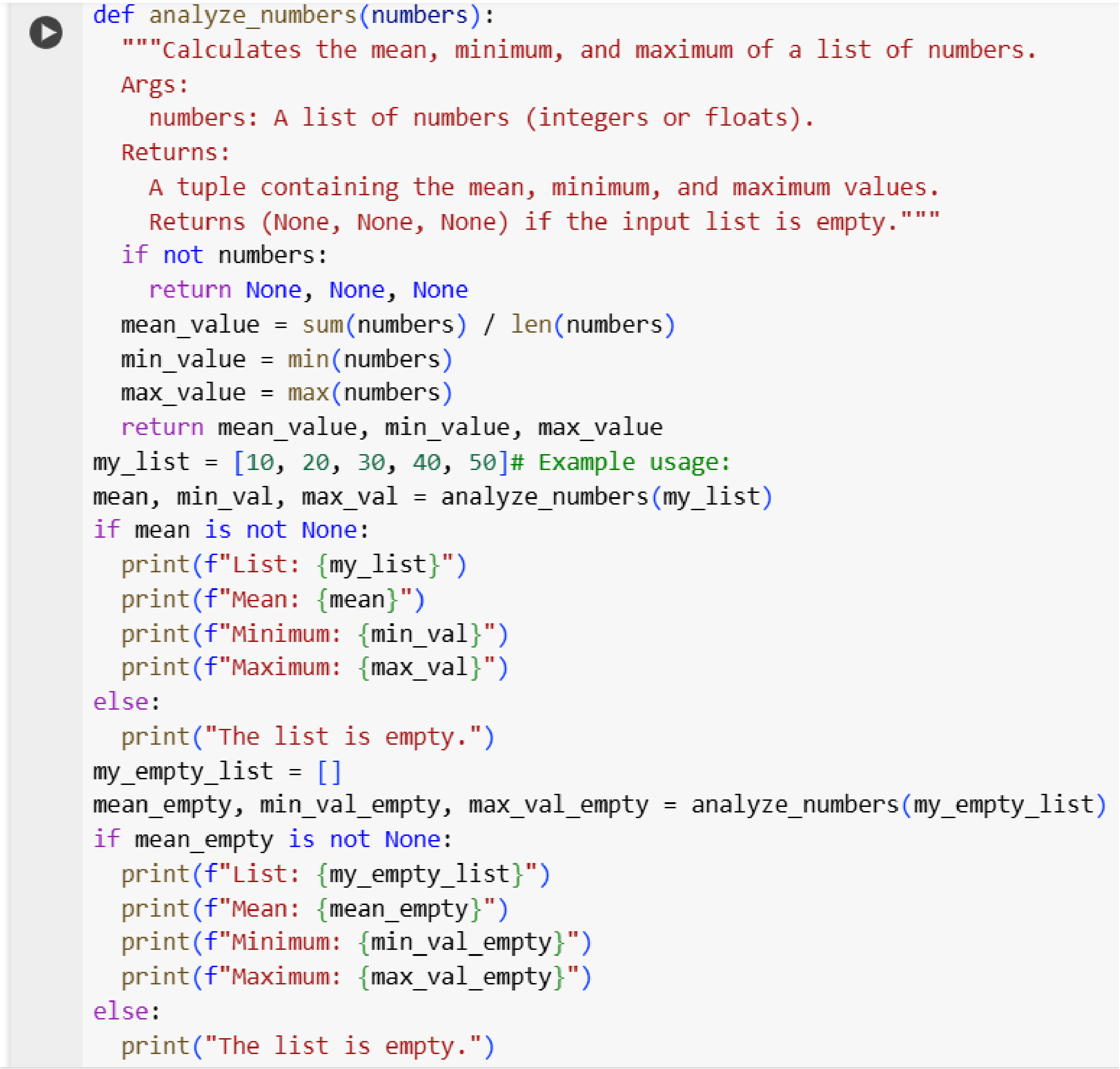
Pavani Voddepalli

Batch : 24BTCAIAI05

TASK 1: Use Google Gemini in Colab to write a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values .

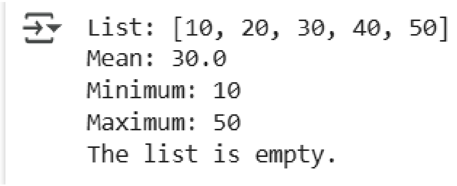
PROMPT : Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values

CODE :



```
def analyze_numbers(numbers):
  """Calculates the mean, minimum, and maximum of a list of numbers.
  Args:
    numbers: A list of numbers (integers or floats).
  Returns:
    A tuple containing the mean, minimum, and maximum values.
    Returns (None, None, None) if the input list is empty."""
  if not numbers:
    return None, None, None
  mean_value = sum(numbers) / len(numbers)
  min_value = min(numbers)
  max_value = max(numbers)
  return mean_value, min_value, max_value
my_list = [10, 20, 30, 40, 50]# Example usage:
mean, min_val, max_val = analyze_numbers(my_list)
if mean is not None:
  print(f"List: {my_list}")
  print(f"Mean: {mean}")
  print(f"Minimum: {min_val}")
  print(f"Maximum: {max_val}")
else:
  print("The list is empty.")
my_empty_list = []
mean_empty, min_val_empty, max_val_empty = analyze_numbers(my_empty_list)
if mean_empty is not None:
  print(f"List: {my_empty_list}")
  print(f"Mean: {mean_empty}")
  print(f"Minimum: {min_val_empty}")
  print(f"Maximum: {max_val_empty}")
else:
  print("The list is empty.")
```

OUTPUT :



```
List: [10, 20, 30, 40, 50]
Mean: 30.0
Minimum: 10
Maximum: 50
The list is empty.
```

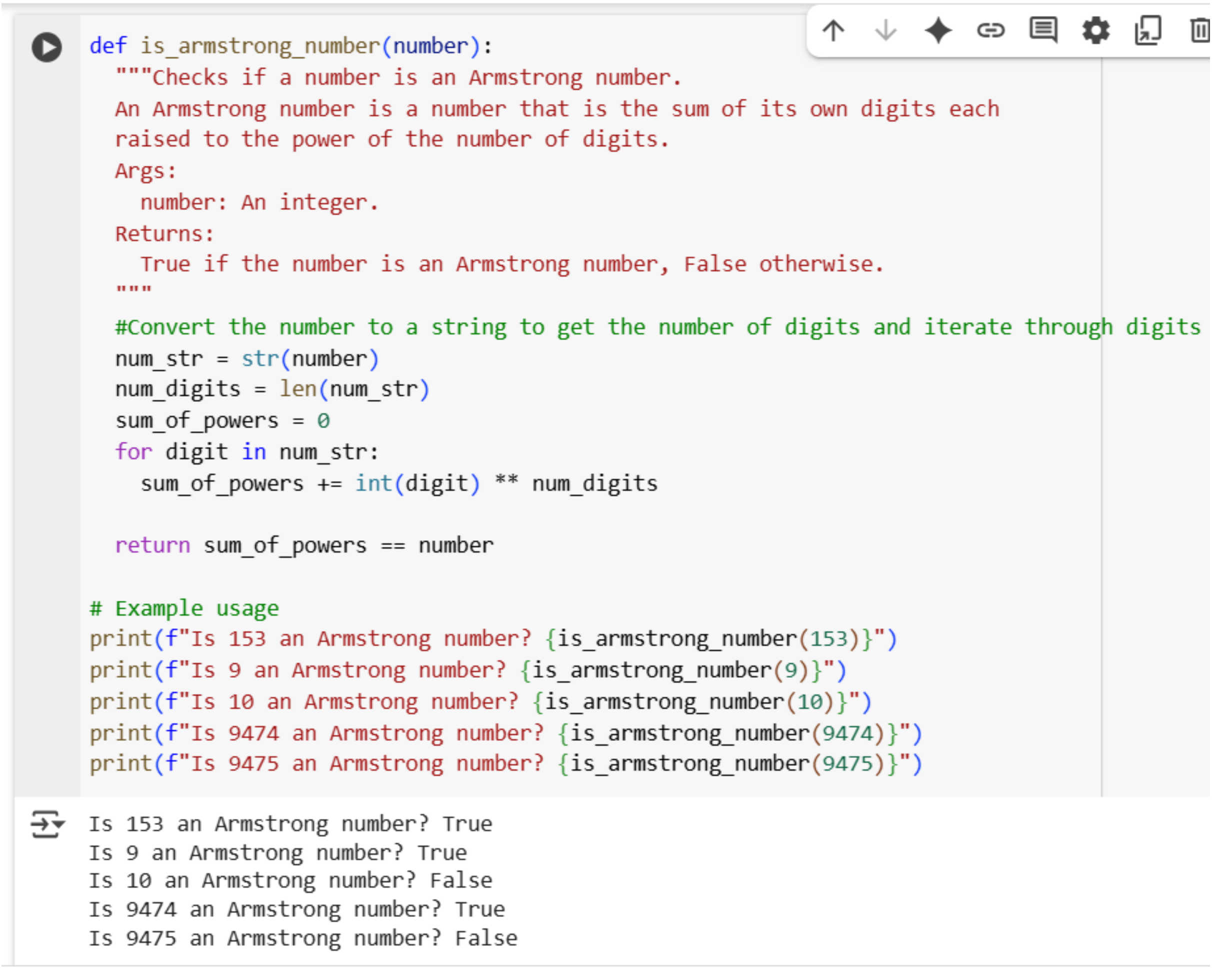
EXPLANATION:

- Test the function with a different list of numbers.
- Explore other statistical calculations (e.g., standard deviation, median).
- Visualize the data using charts or plots.
- Read numbers from a file instead of a list.

TASK 2 : Compare Gemini and Copilot outputs for a Python function that checks whether a number is an Armstrong number. Document the steps, prompts, and outputs

PROMPT FOR GEMINI : `Generate a python function that checks whether a number is an Armstrong number`

CODE :



```python
def is_armstrong_number(number):
  """Checks if a 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: An integer.
  Returns:
    True if the number is an Armstrong number, False otherwise.
  """
  #Convert the number to a string to get the number of digits and iterate through digits
  num_str = str(number)
  num_digits = len(num_str)
  sum_of_powers = 0
  for digit in num_str:
    sum_of_powers += int(digit) ** num_digits

  return sum_of_powers == number

# Example usage
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}")
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}")
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}")
print(f"Is 9474 an Armstrong number? {is_armstrong_number(9474)}")
print(f"Is 9475 an Armstrong number? {is_armstrong_number(9475)}")
```

```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 9474 an Armstrong number? True
Is 9475 an Armstrong number? False
```

STEPS :

The code defines a Python function called is_armstrong_number that checks if a given integer is an Armstrong number. An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the total number of digits in the number.

Here's a breakdown of the code:

Function Definition:

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

This defines the function is_armstrong_number which takes one argument, number. The docstring explains what the function does, its arguments, and what it returns.

2. **Convert to String and Get Number of Digits:**

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

The input number is converted to a string (num_str) to easily determine the number of digits (num_digits) using the len() function and to iterate through each digit.

3. **Calculate Sum of Powers:**

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

An accumulator variable sum_of_powers is initialized to 0. The code then iterates through each character (digit) in the num_str. Inside the loop, each digit (which is a string character) is converted back to an integer using int(digit), and then raised to the power of num_digits using the ** operator. This result is added to sum_of_powers.

4. **Check and Return:**

```
  return sum_of_powers == number
```

Finally, the function compares the calculated sum_of_powers with the original number. If they are equal, the number is an Armstrong number, and the function returns True; otherwise, it returns False.

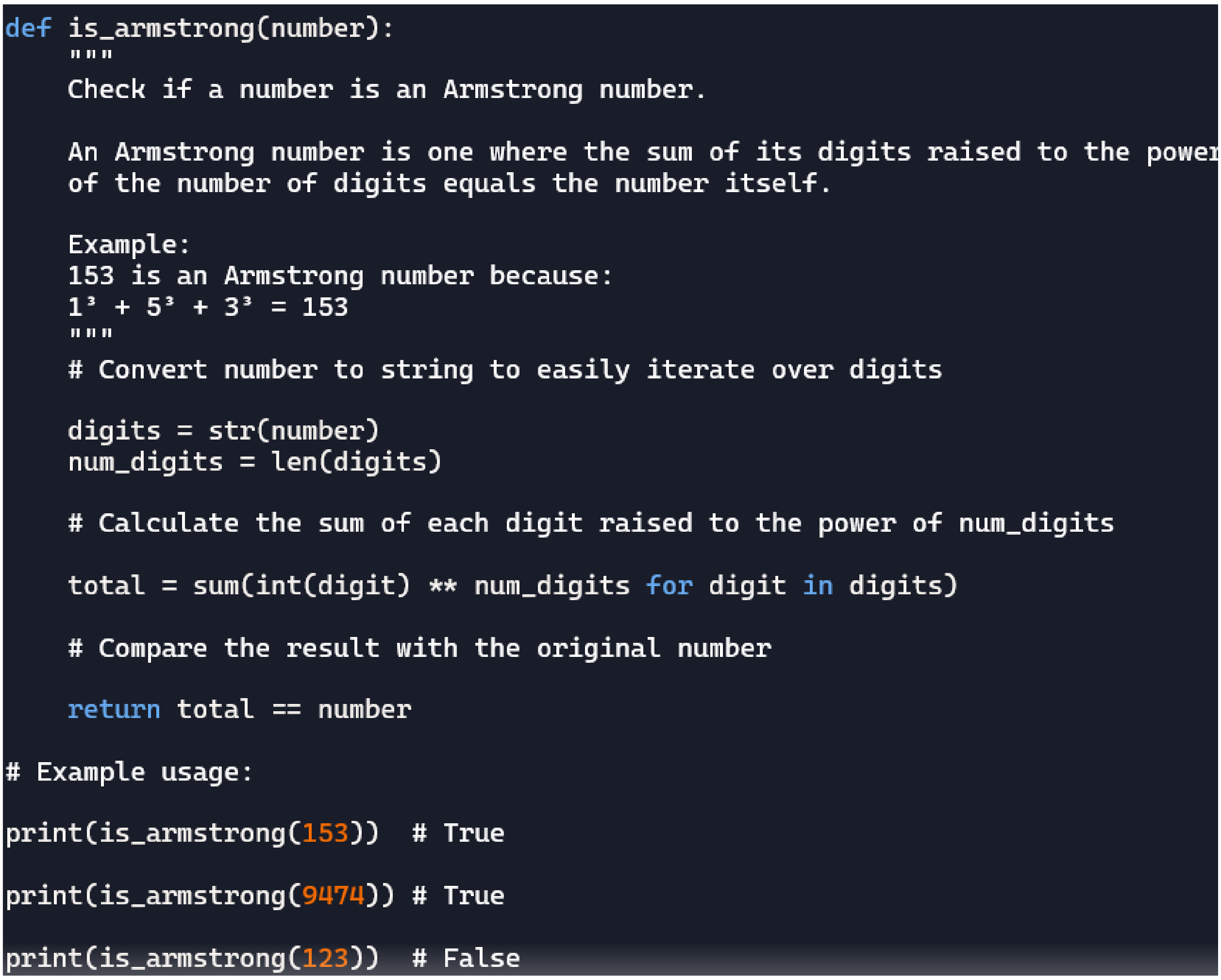
5. **Example Usage:**

```
# Example usage
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}")
# ... other print statements ...
```

This section demonstrates how to use the is_armstrong_number function with several example numbers (153, 9, 10, 9474, and 9475) and prints the results.

PROMPT FOR COPILOT : Generate a python function that checks whether a number is an Armstrong number

CODE :



```python
def is_armstrong(number):
    """
    Check if a number is an Armstrong number.

    An Armstrong number is one where the sum of its digits raised to the power
    of the number of digits equals the number itself.

    Example:
    153 is an Armstrong number because:
    1³ + 5³ + 3³ = 153
    """
    # Convert number to string to easily iterate over digits

    digits = str(number)
    num_digits = len(digits)

    # Calculate the sum of each digit raised to the power of num_digits

    total = sum(int(digit) ** num_digits for digit in digits)

    # Compare the result with the original number

    return total == number

# Example usage:

print(is_armstrong(153))  # True

print(is_armstrong(9474)) # True

print(is_armstrong(123))  # False
```

STEPS :

def is_armstrong(number):
- This defines a function named is_armstrong that takes one input: number.

digits = str(number)
- Converts the number to a string so we can loop through each digit easily.

num_digits = len(digits)
- Counts how many digits are in the number. This is the exponent we'll raise each digit to.

total = sum(int(digit) ** num_digits for digit in digits)
- This is the heart of the function:
- Loops through each digit.
- Converts it back to an integer.
- Raises it to the power of num_digits.
- Sums all those powered digits.

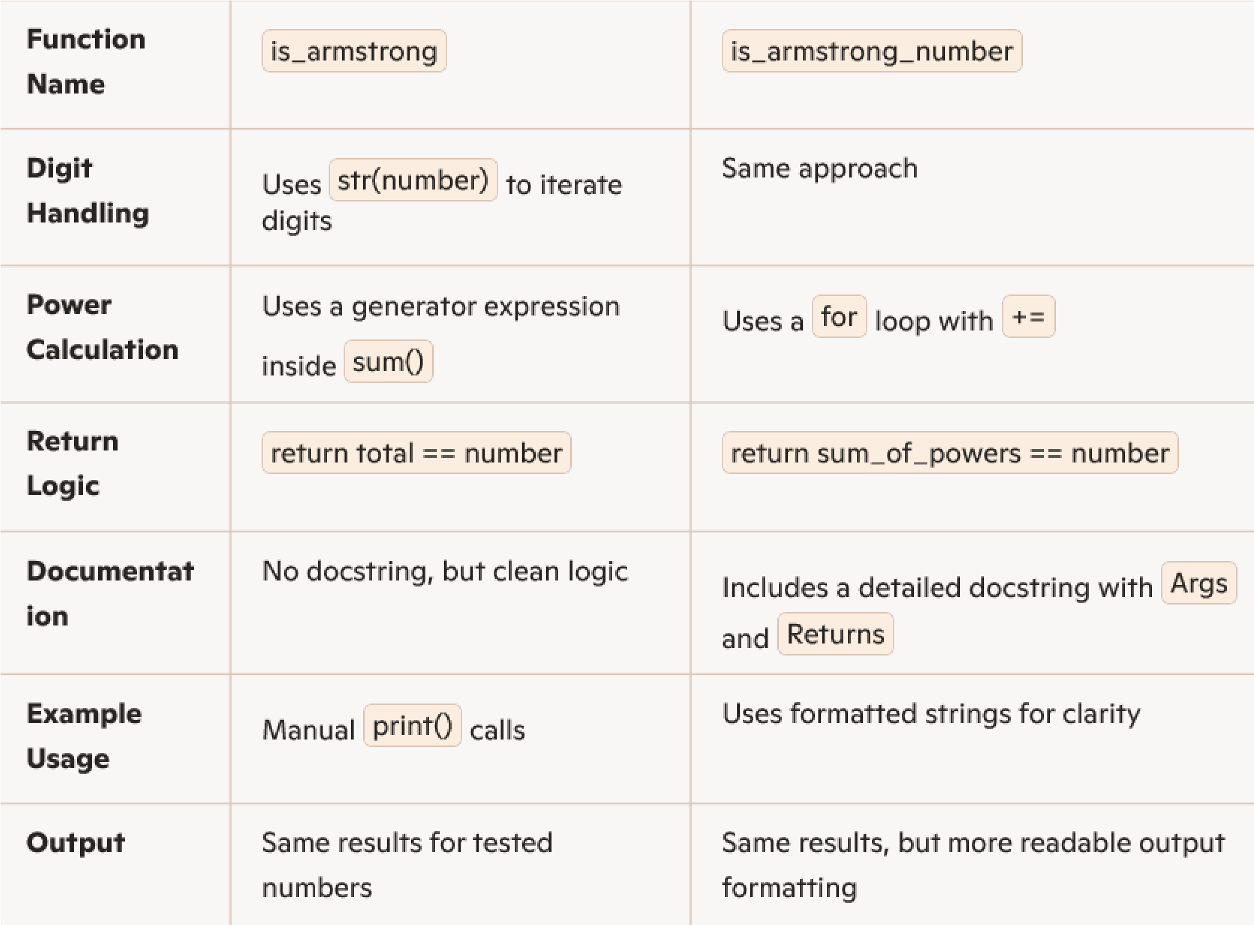
return total == number
- Finally, it checks if the sum equals the original number.

- If yes, it returns True → it's an Armstrong number.
- Otherwise, False.

Sample Output

print(is_armstrong(153)) # True
print(is_armstrong(9474)) # True
print(is_armstrong(123)) # False

COMPARISION :



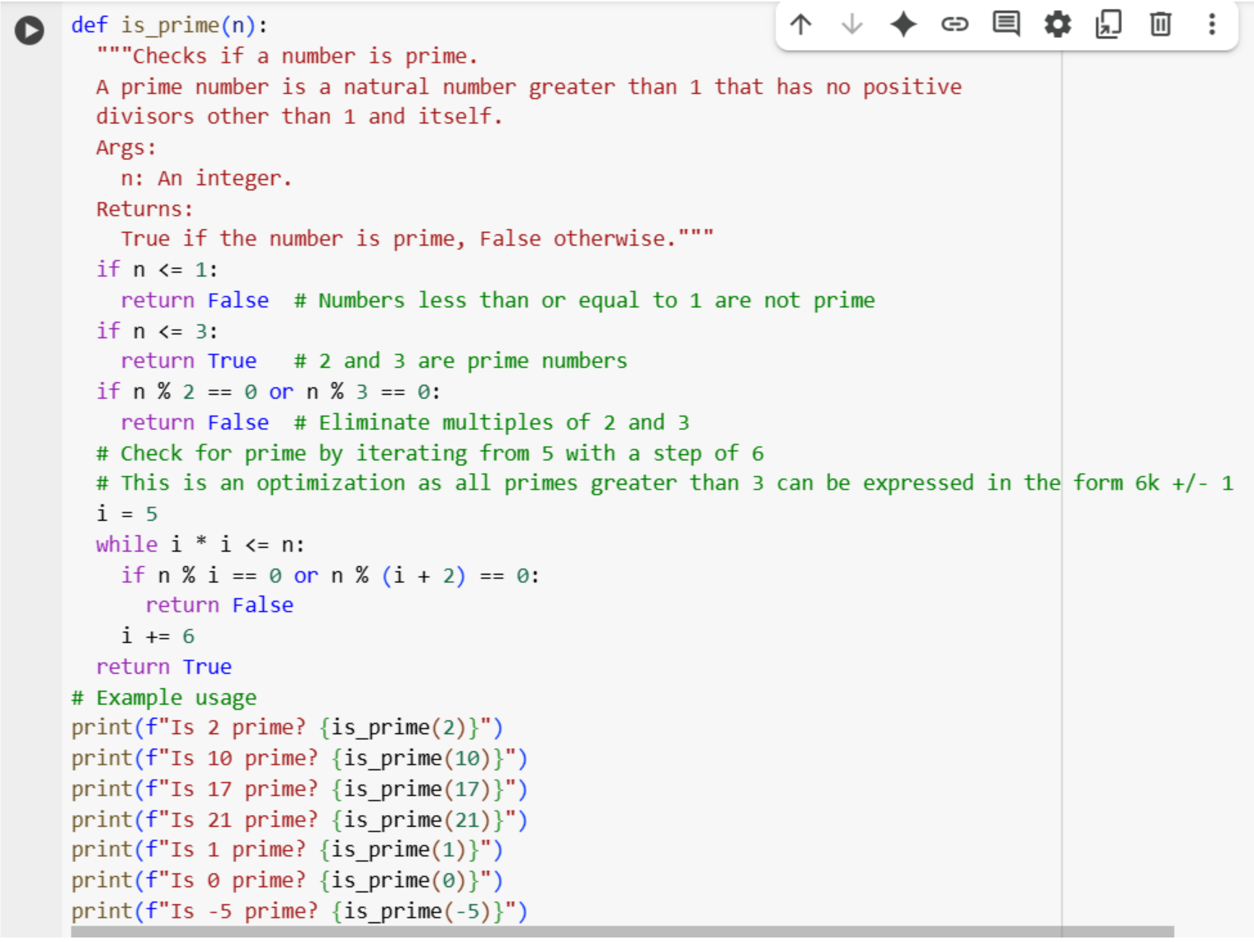
Function Name	`is_armstrong`	`is_armstrong_number`
Digit Handling	Uses `str(number)` to iterate digits	Same approach
Power Calculation	Uses a generator expression inside `sum()`	Uses a `for` loop with `+=`
Return Logic	`return total == number`	`return sum_of_powers == number`
Documentation	No docstring, but clean logic	Includes a detailed docstring with `Args` and `Returns`
Example Usage	Manual `print()` calls	Uses formatted strings for clarity
Output	Same results for tested numbers	Same results, but more readable output formatting

TASK 3 : Ask Gemini to explain a Python function (e.g., is_prime(n) or is_palindrome(s)) line by line.

Choose either a prime-checking or palindrome-checking function and document the explanation provided by Gemini

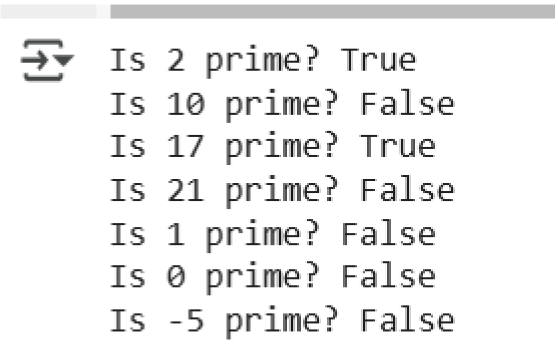
PROMPT : Generate a python function is_prime(n) to check whether a number is prime

CODE :



```python
def is_prime(n):
  """Checks if a number is prime.
  A prime number is a natural number greater than 1 that has no positive
  divisors other than 1 and itself.
  Args:
    n: An integer.
  Returns:
    True if the number is prime, False otherwise."""
  if n <= 1:
    return False  # Numbers less than or equal to 1 are not prime
  if n <= 3:
    return True   # 2 and 3 are prime numbers
  if n % 2 == 0 or n % 3 == 0:
    return False  # Eliminate multiples of 2 and 3
  # Check for prime by iterating from 5 with a step of 6
  # This is an optimization as all primes greater than 3 can be expressed in the form 6k +/- 1
  i = 5
  while i * i <= n:
    if n % i == 0 or n % (i + 2) == 0:
      return False
    i += 6
  return True
# Example usage
print(f"Is 2 prime? {is_prime(2)}")
print(f"Is 10 prime? {is_prime(10)}")
print(f"Is 17 prime? {is_prime(17)}")
print(f"Is 21 prime? {is_prime(21)}")
print(f"Is 1 prime? {is_prime(1)}")
print(f"Is 0 prime? {is_prime(0)}")
print(f"Is -5 prime? {is_prime(-5)}")
```

OUTPUT :



```
Is 2 prime? True
Is 10 prime? False
Is 17 prime? True
Is 21 prime? False
Is 1 prime? False
Is 0 prime? False
Is -5 prime? False
```

LINE BY LINE EXPLANATION :

def is_prime(n):
"""Checks if a number is prime.
# ... docstring ...
"""

This line defines a function named is_prime that takes one argument, n. The docstring provides a brief explanation of what the function does, what a prime number is, the argument it takes, and what it returns.

if n <= 1:
return False # Numbers less than or equal to 1 are not prime .This is the first conditional check. Prime numbers are defined as natural numbers greater than 1. So, if the input number n is less than or equal to 1, the function immediately returns False. The comment clarifies why this check is performed.

```
    if n <= 3:
        return True   # 2 and 3 are prime numbers
```

This check handles the numbers 2 and 3. Both 2 and 3 are prime numbers. If n is either 2 or 3 (and not less than or equal to 1 based on the previous check), the function returns True. The comment explains that 2 and 3 are prime.

```
    if n % 2 == 0 or n % 3 == 0:
        return False  # Eliminate multiples of 2 and 3
```

This is an optimization. Any number greater than 3 that is divisible by 2 or 3 is not prime. This check quickly eliminates multiples of 2 and 3. The modulo operator (%) checks for the remainder of a division. If n divided by 2 or 3 has a remainder of 0, it means n is divisible by 2 or 3, and thus not prime, so the function returns False. The comment explains that this step eliminates multiples of 2 and 3.

```
    # Check for prime by iterating from 5 with a step of 6
    # This is an optimization as all primes greater than 3 can be expressed in the form 6k +/- 1
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
```

*This is the core of the primality test for numbers greater than 3. It uses an optimization: all prime numbers greater than 3 can be expressed in the form 6k + 1 or 6k - 1 (where k is an integer). The code starts checking for divisibility from i = 5. The while i * i <= n: condition is an optimization because if a number n has a divisor greater than its square root, it must also have a divisor smaller than its square root, which would have already been found. Inside the loop, it checks if n is divisible by i (which represents 6k - 1) or i + 2 (which represents 6k + 1). If n is divisible by either, it's not prime, and the function returns False. The variable i is then incremented by 6 in each iteration (i += 6) to efficiently check only numbers of the form 6k +/- 1. The comments explain the optimization and the logic of the loop.*

```
    return True
```

If the loop completes without finding any divisors, it means the number n is not divisible by any number up to its square root (following the 6k +/- 1 pattern), and therefore it is a prime number. In this case, the function returns True.These lines demonstrate how to call the is_prime function with different integer inputs and print the results to the console, showing whether each number is prime or not.

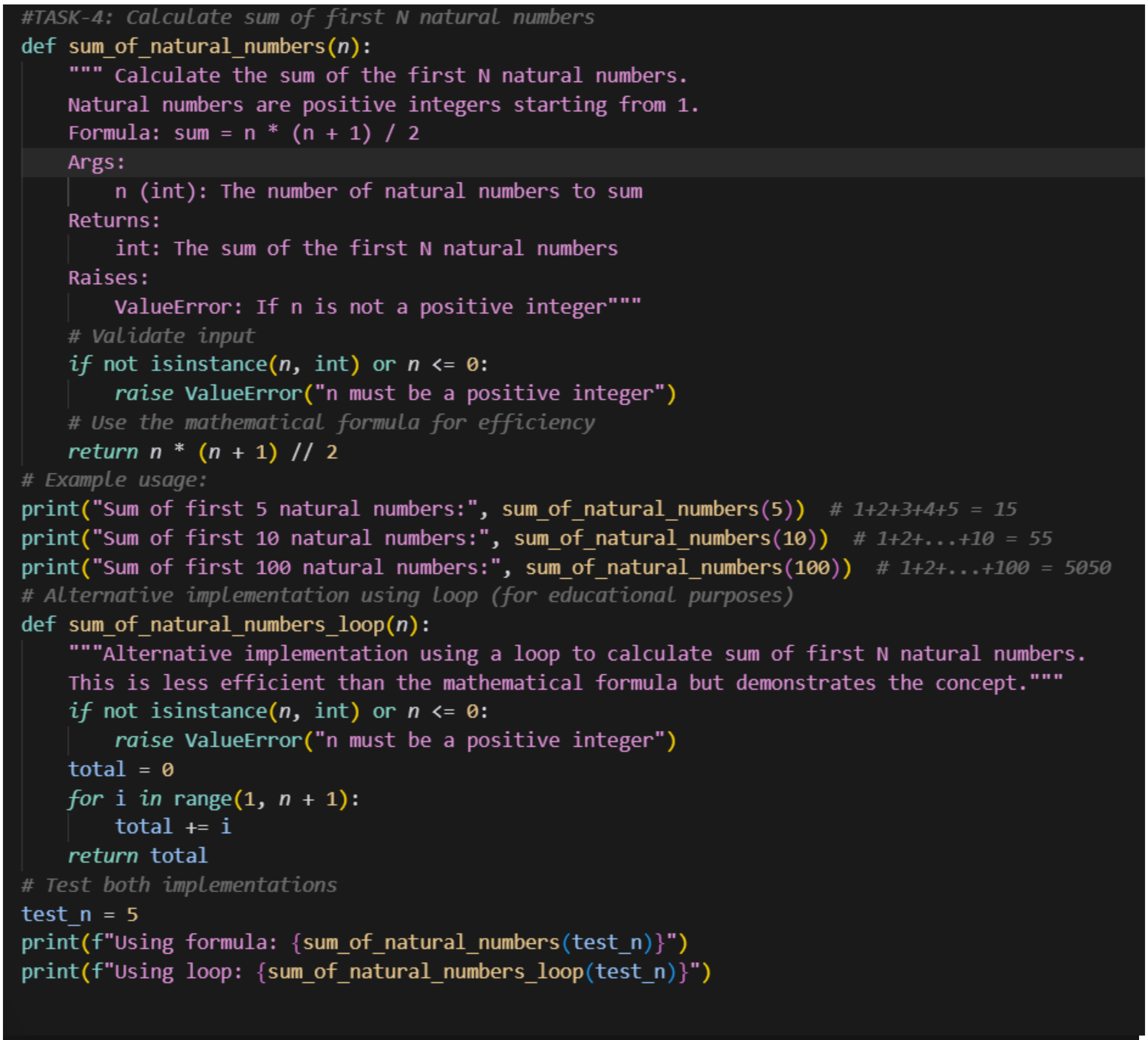
TASK 4 : Install and configure Cursor AI. Use it to generate a Python function (e.g., sum of the first N natural numbers) and test its output.

PROMPT : Generate a Python function that calculates sum of the first N natural numbers

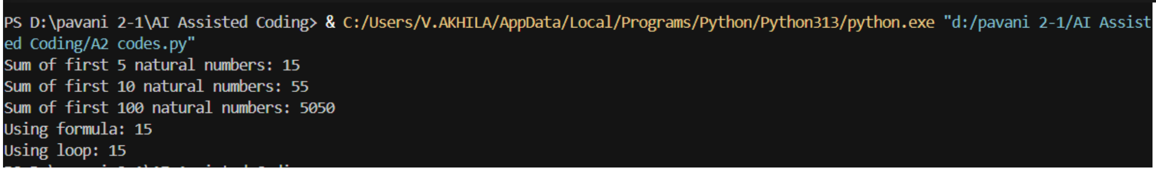
CODE :

&

OUTPUT:



```python
#TASK-4: Calculate sum of first N natural numbers
def sum_of_natural_numbers(n):
    """ Calculate the sum of the first N natural numbers.
    Natural numbers are positive integers starting from 1.
    Formula: sum = n * (n + 1) / 2
    Args:
        n (int): The number of natural numbers to sum
    Returns:
        int: The sum of the first N natural numbers
    Raises:
        ValueError: If n is not a positive integer"""
    # Validate input
    if not isinstance(n, int) or n <= 0:
        raise ValueError("n must be a positive integer")
    # Use the mathematical formula for efficiency
    return n * (n + 1) // 2
# Example usage:
print("Sum of first 5 natural numbers:", sum_of_natural_numbers(5))  # 1+2+3+4+5 = 15
print("Sum of first 10 natural numbers:", sum_of_natural_numbers(10))  # 1+2+...+10 = 55
print("Sum of first 100 natural numbers:", sum_of_natural_numbers(100))  # 1+2+...+100 = 5050
# Alternative implementation using loop (for educational purposes)
def sum_of_natural_numbers_loop(n):
    """Alternative implementation using a loop to calculate sum of first N natural numbers.
    This is less efficient than the mathematical formula but demonstrates the concept."""
    if not isinstance(n, int) or n <= 0:
        raise ValueError("n must be a positive integer")
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
# Test both implementations
test_n = 5
print(f"Using formula: {sum_of_natural_numbers(test_n)}")
print(f"Using loop: {sum_of_natural_numbers_loop(test_n)}")
```



```
PS D:\pavani 2-1\AI Assisted Coding> & C:/Users/V.AKHILA/AppData/Local/Programs/Python/Python313/python.exe "d:/pavani 2-1/AI Assist
ed Coding/A2 codes.py"
Sum of first 5 natural numbers: 15
Sum of first 10 natural numbers: 55
Sum of first 100 natural numbers: 5050
Using formula: 15
Using loop: 15
```

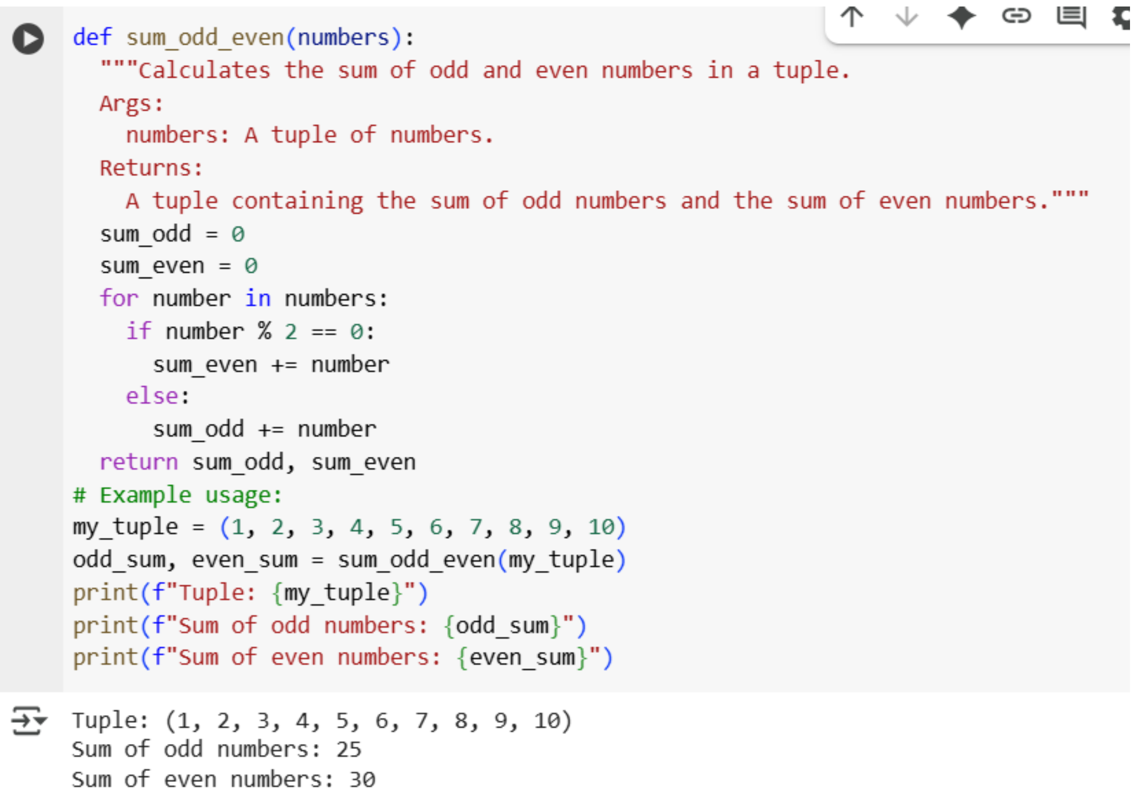
<u>TASK 5 :</u> Write a Python program to calculate the sum of odd numbers and even numbers in a given tuple. Refactor the code to improve logic and readability

PROMPT : <u>Generate a Python program to calculate the sum of odd numbers and even numbers in a given tuple.</u>

CODE :

&

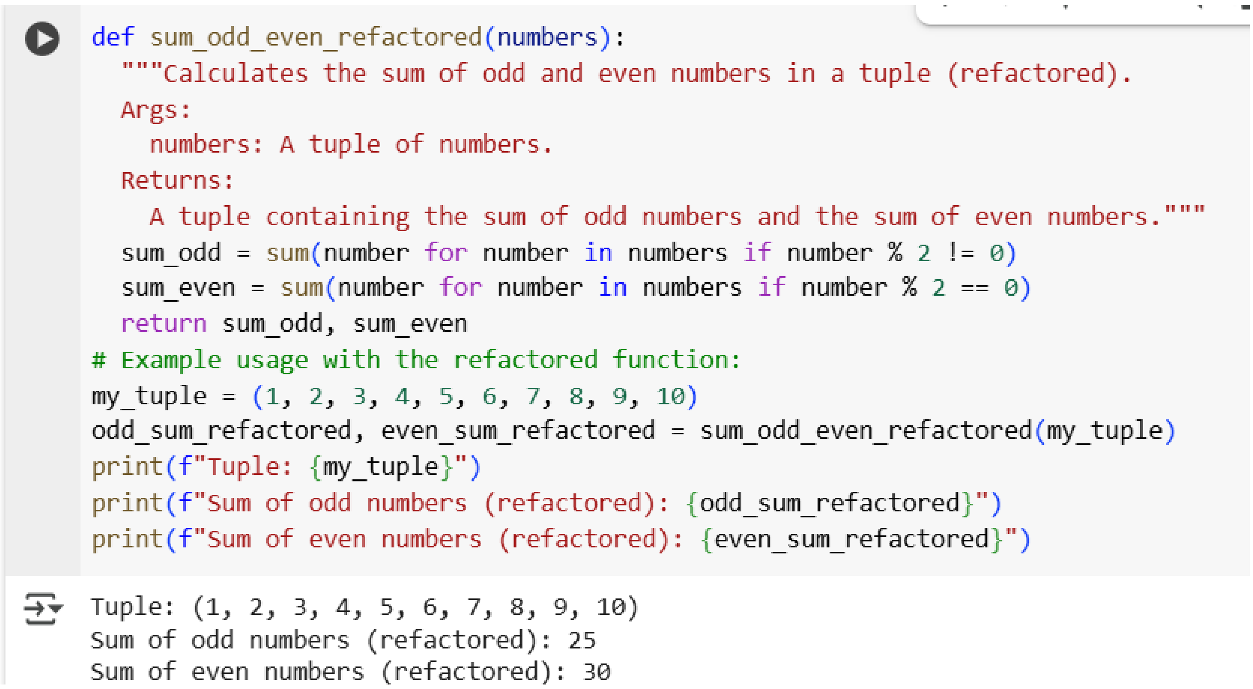
OUTPUT :



```python
def sum_odd_even(numbers):
  """Calculates the sum of odd and even numbers in a tuple.
  Args:
    numbers: A tuple of numbers.
  Returns:
    A tuple containing the sum of odd numbers and the sum of even numbers."""
  sum_odd = 0
  sum_even = 0
  for number in numbers:
    if number % 2 == 0:
      sum_even += number
    else:
      sum_odd += number
  return sum_odd, sum_even
# Example usage:
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
odd_sum, even_sum = sum_odd_even(my_tuple)
print(f"Tuple: {my_tuple}")
print(f"Sum of odd numbers: {odd_sum}")
print(f"Sum of even numbers: {even_sum}")
```

```
Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of odd numbers: 25
Sum of even numbers: 30
```

Refactoring the code to improve logic and readability



```python
def sum_odd_even_refactored(numbers):
  """Calculates the sum of odd and even numbers in a tuple (refactored).
  Args:
    numbers: A tuple of numbers.
  Returns:
    A tuple containing the sum of odd numbers and the sum of even numbers."""
  sum_odd = sum(number for number in numbers if number % 2 != 0)
  sum_even = sum(number for number in numbers if number % 2 == 0)
  return sum_odd, sum_even
# Example usage with the refactored function:
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
odd_sum_refactored, even_sum_refactored = sum_odd_even_refactored(my_tuple)
print(f"Tuple: {my_tuple}")
print(f"Sum of odd numbers (refactored): {odd_sum_refactored}")
print(f"Sum of even numbers (refactored): {even_sum_refactored}")
```

```
Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of odd numbers (refactored): 25
Sum of even numbers (refactored): 30
```