AI ASSISTED CODING

**ASSIGNMENT:2.1**

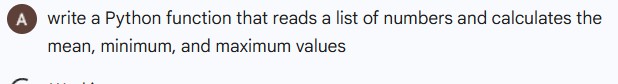
**M.Swapna**

**2403A51260**

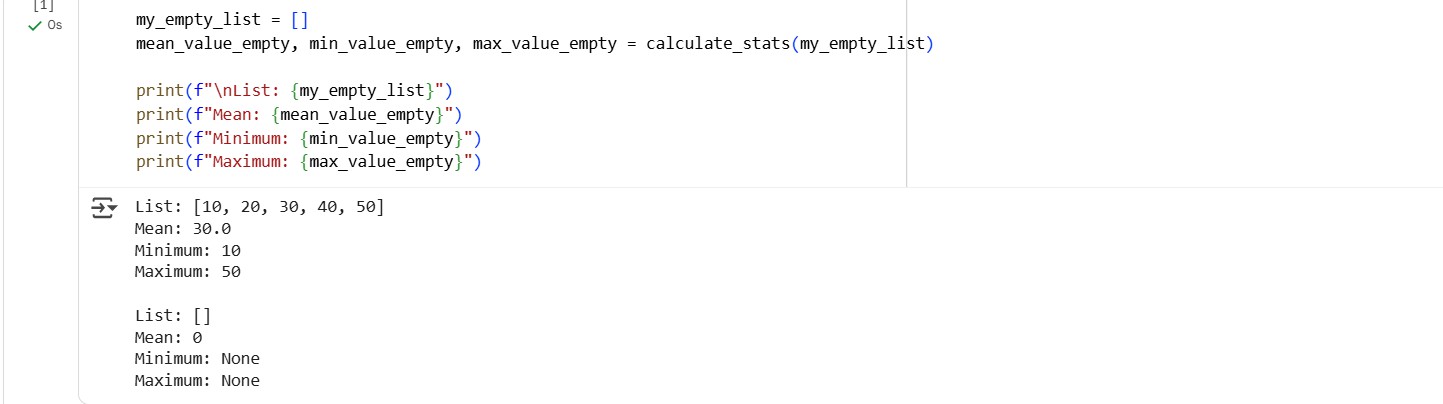
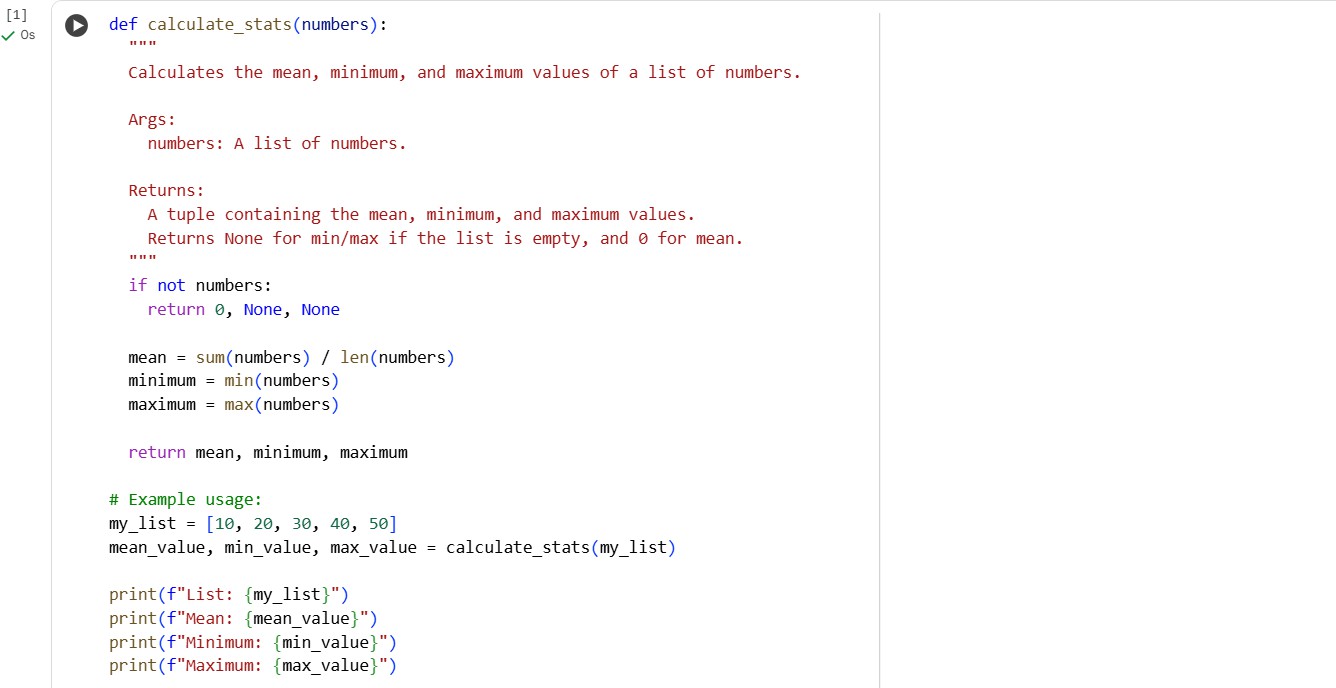
**BATCH-11**

**Task1:**

**Prompt:**

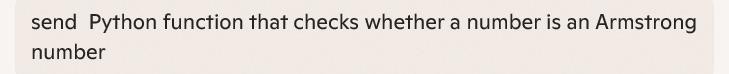
****

**Response from gemini colab(code with output:**

****

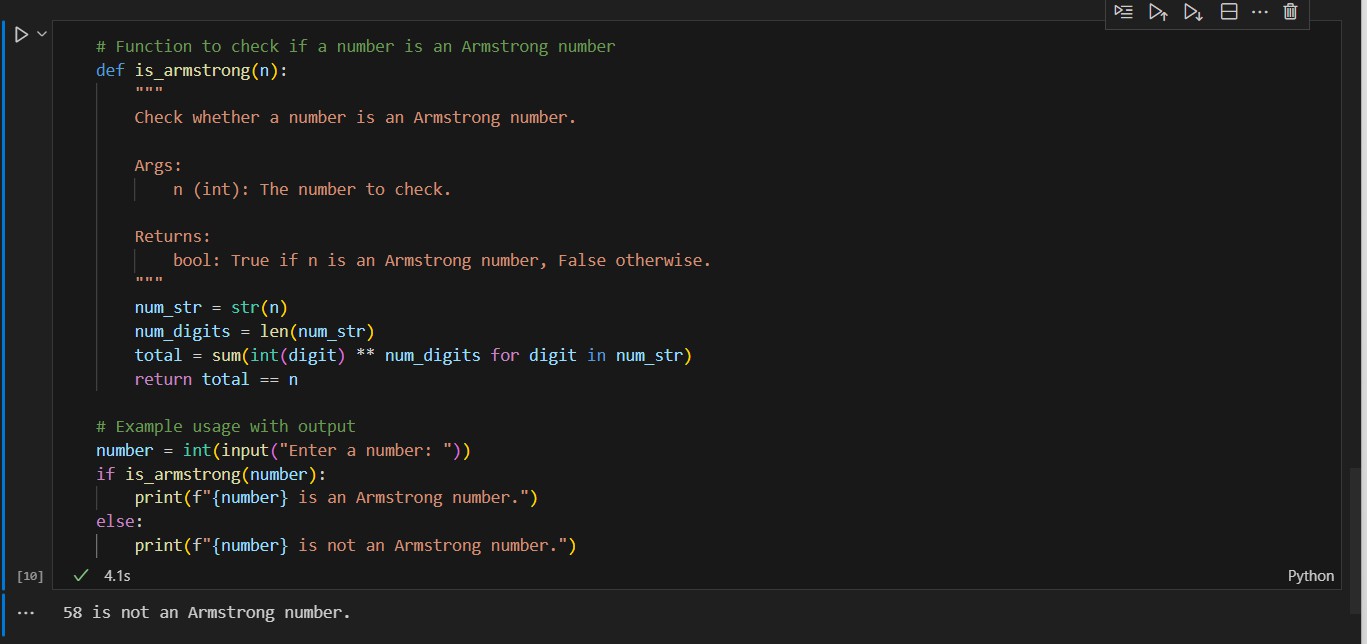
**Task2:**

**Prompt:**



**Response from copilot:**

**Code with output:**

****

**Explanation:**

˛C\* **Python Function: Armstrong Number Checker**

# Function to check if a number is an Armstrong number def is\_armstrong(n):

"""

Check whether a number is an Armstrong number.

Args:

n (int): The number to check.

Returns:

bool: True if n is an Armstrong number, False otherwise. """

num\_str = str(n) # Convert number to string to access digits num\_digits = len(num\_str) # Count how many digits the number has

total = sum(int(digit) \*\* num\_digits for digit in num\_str) # Sum of digits raised to power return total == n # Compare with original number

# Example usage

number = int(input("Enter a number: ")) if is\_armstrong(number):

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

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

\_] H **Explanation**

Let’s break it down step-by-step:

1. **Convert to string**:

We convert the number to a string so we can easily loop through each digit.

1. **Count digits**:

The number of digits determines the power to which each digit is raised. For example, 153 has 3 digits → each digit is raised to the power of 3.

1. **Sum of powered digits**:

We compute the sum of each digit raised to the power of the total number of digits. Example for 153:

( 13 + 53 + 3^3 = 1 + 125 + 27 = 153 )

1. **Compare with original**:

If the sum equals the original number, it's an Armstrong number.

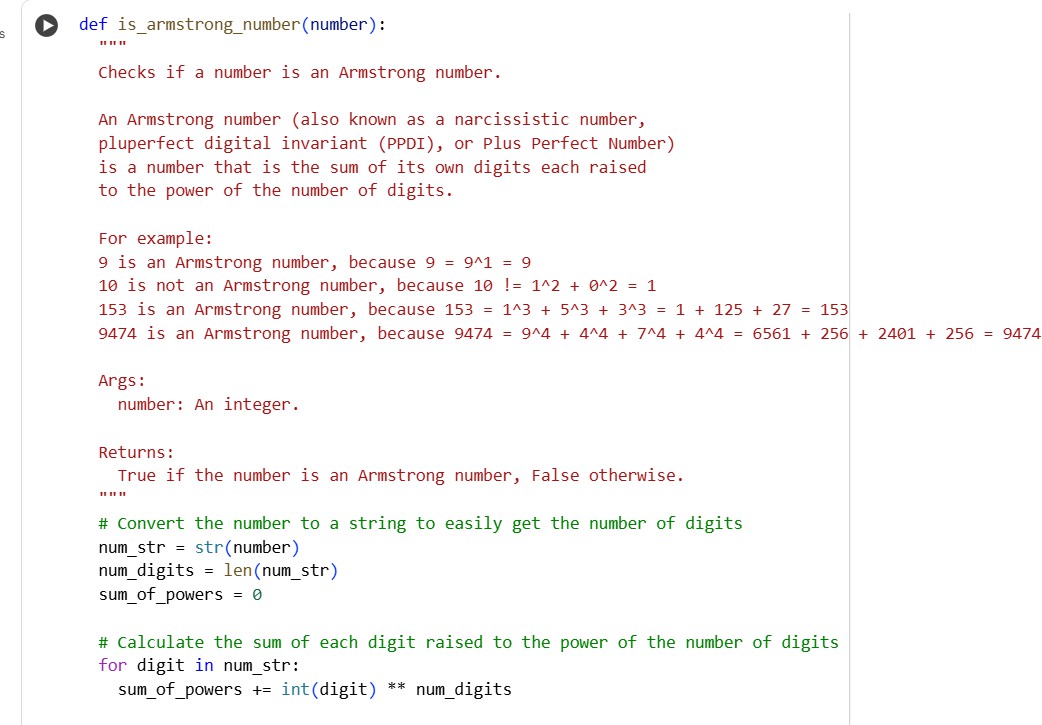
⬛ **Sample Run**

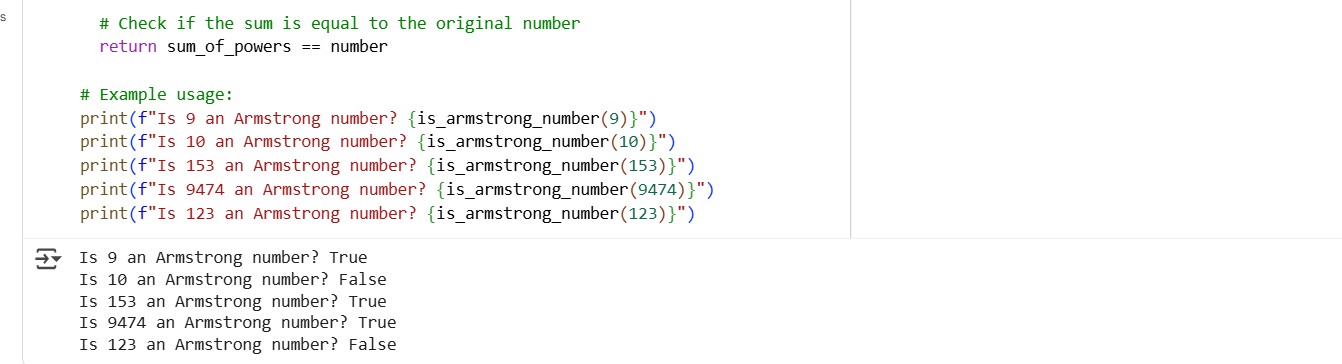
Enter a number: 9474

9474 is an Armstrong number.

Response from gemini:

Code with output:





Explanation:

The is\_armstrong\_number(number) function works as follows:

1. **Convert to String**: It first converts the input number to a string (num\_str). This is done to easily determine the number of digits in the number (num\_digits).
2. **Initialize Sum**: It initializes a variable sum\_of\_powers to 0. This variable will store the sum of each digit raised to the power of the total number of digits.
3. **Calculate Sum of Powers**: It then iterates through each digit in the num\_str. For each digit, it converts the digit back to an integer (int(digit)) and raises it to the power of num\_digits. This result is added to sum\_of\_powers.
4. **Check and Return**: Finally, it compares the calculated sum\_of\_powers with the

original number. If they are equal, it means the number is an Armstrong number, and the function returns True. Otherwise, it returns False.

The example usage demonstrates how to call the function with a few different numbers and prints whether each is an Armstrong number.

Comparision:

.˙•¸'7s **Comparison: Gemini vs Copilot (Armstrong Number Check in Python) Aspect Copilot Gemini**

**Prompt/Task**

User asked Copilot: *“Write a Python function to check if a number is an Armstrong number.”*

User asked Gemini: *“Write a Python function to check if a number is an Armstrong number with examples.”*

**Function Name**

is\_armstrong is\_armstrong\_number

**Docstring Style**

Short and minimal:

"""Check whether a number is an Armstrong number."""

Long, detailed, with definition, alternate names, and multiple worked examples. ⬛

**Logic**

**Example Usage**

Uses **generator expression with sum()** (compact, efficient).

total = sum(int(digit) \*\* num\_digits for digit in num\_str)

Interactive input:

number = int(input("Enter a number: ")) → checks one number at a time.

Uses **explicit for-loop with accumulation** (clearer for beginners).

for digit in num\_str: sum\_of\_powers += int(digit) \*\* num\_digits

Predefined multiple test cases:

prints results for 9, 10, 153, 9474, 123 → good for demos/testing. ⬛

**Output Style** Friendly message:

9474 is an Armstrong number.

Explicit test results:

Is 153 an Armstrong number? True

**Your Run (Copilot)**

**Strengths**

**Weaknesses**

Enter a number: 9474 9474 is an Armstrong number.

✔ Concise, efficient code

✔ Interactive for user input

+ Only checks one input at a time

+ Minimal documentation

Is 9 an Armstrong number? True Is 10 an Armstrong number? False Is 153 an Armstrong number? True Is 9474 an Armstrong number? True Is 123 an

Armstrong number? False

✔ Beginner-friendly

✔ Educational docstring

✔ Multiple auto test cases

+ Verbose

+ Slightly less efficient

**Best Use Case** Quick script where user enters

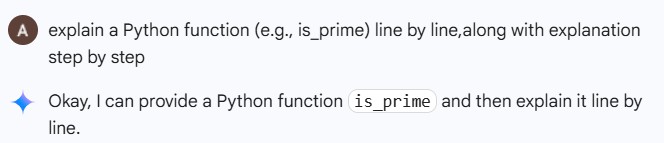
any number.

Teaching, debugging, or when you want multiple examples verified.

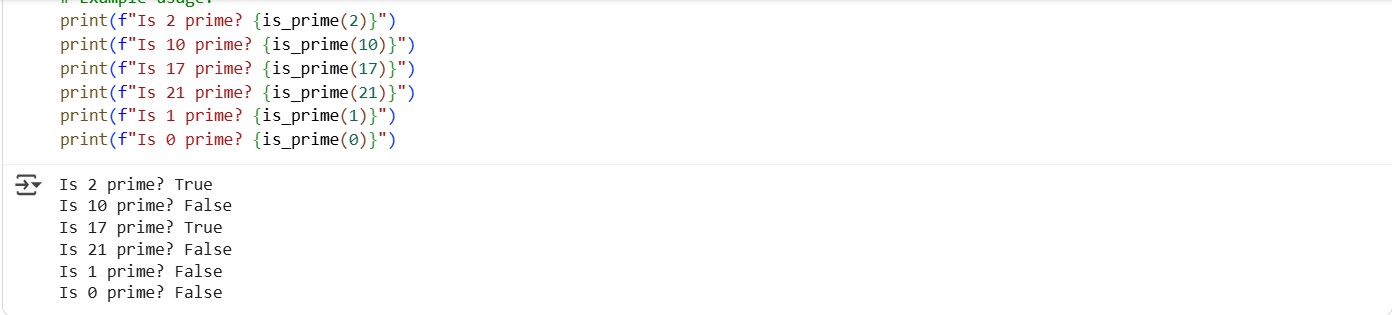
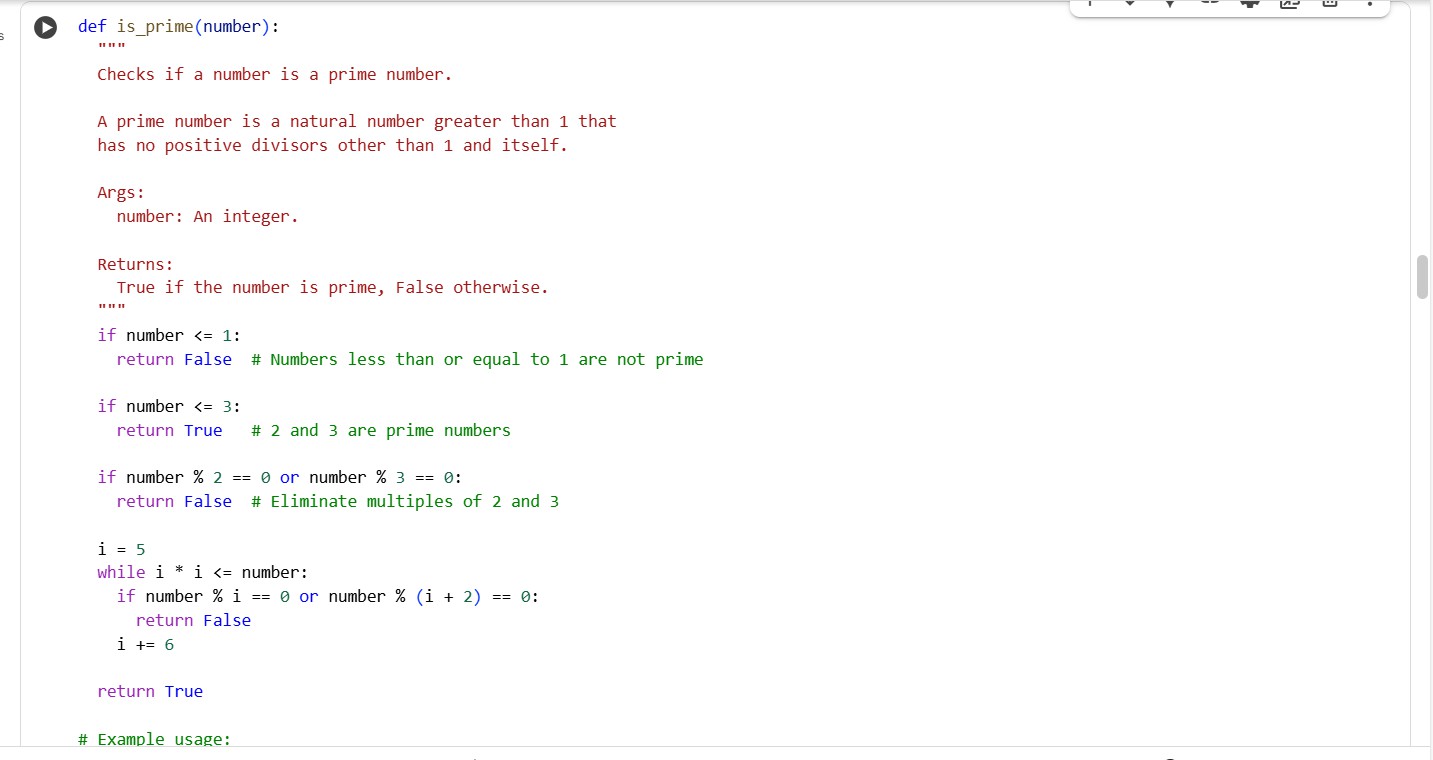
⬛ **Conclusion**:

* **Copilot** → Best for **concise, interactive programs**.
* **Gemini** → Best for **educational clarity & multiple test cases**.
* **Best Combo** → Use **Copilot’s efficient logic** + **Gemini’s rich documentation/examples**. **Task3:**

**Prompt:**

****

**Code with output:**

****

**Explanation:**

the explanation of the is\_prime function directly in our chat. Here is a line-by-line explanation:

def is\_prime(number):.

This line defines a function named is\_prime that takes one argument, number. """

Checks if a number is a prime number.

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Args:

number: An integer.

Returns:

True if the number is prime, False otherwise. """

This is a docstring, explaining what the function does, its arguments, and what it returns. if number <= 1:

return False # Numbers less than or equal to 1 are not prime

This checks if the input number is less than or equal to 1. Prime numbers are defined as being greater than 1, so if it's 1 or less, the function immediately returns False.

if number <= 3:

return True # 2 and 3 are prime numbers

This checks if the number is 2 or 3. These are prime numbers, so the function returns True. This check is done separately for optimization.

if number % 2 == 0 or number % 3 == 0:

return False # Eliminate multiples of 2 and 3

This checks if the number is divisible by 2 or 3. If it is, it's not prime (unless it's 2 or 3 themselves, which were handled in the previous step), so the function returns False. This eliminates a large number of non-prime numbers quickly.

i = 5

This initializes a variable i to 5. This variable will be used in the following loop to check for other divisors.

while i \* i <= number:

This is a while loop that continues as long as i \* i is less than or equal to the number. We only

need to check for divisors up to the square root of the number, because if a number has a divisor greater than its square root, it must also have a divisor less than its square root.

if number % i == 0 or number % (i + 2) == 0: return False

Inside the loop, this checks if the number is divisible by i or i + 2. This is an optimization based on the fact that all prime numbers greater than 3 can be expressed in the form 6k ± 1. So, we only need to check divisibility by numbers of this form (5, 7, 11, 13, 17, 19, and so on).

i += 6

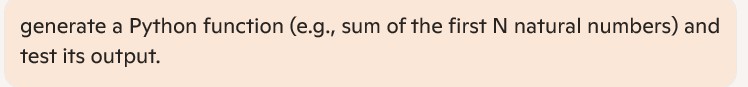
This increments i by 6 in each iteration of the loop. This moves to the next potential divisors of the form 6k ± 1 (e.g., if i was 5, the next i will be 11, checking 11 and 13).

return True

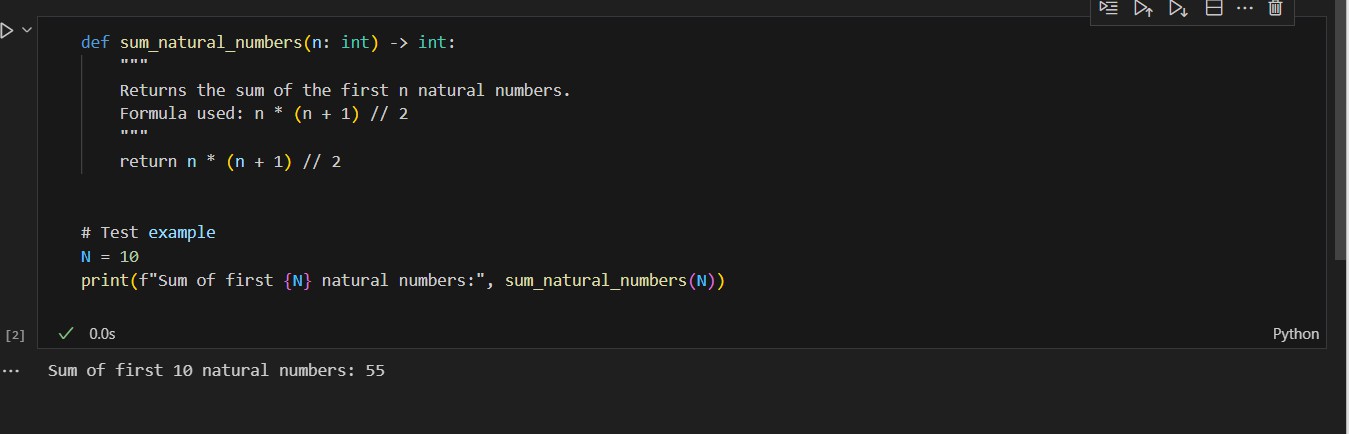
If the loop finishes without finding any divisors, it means the number is prime, and the function returns True.

**Task4:**

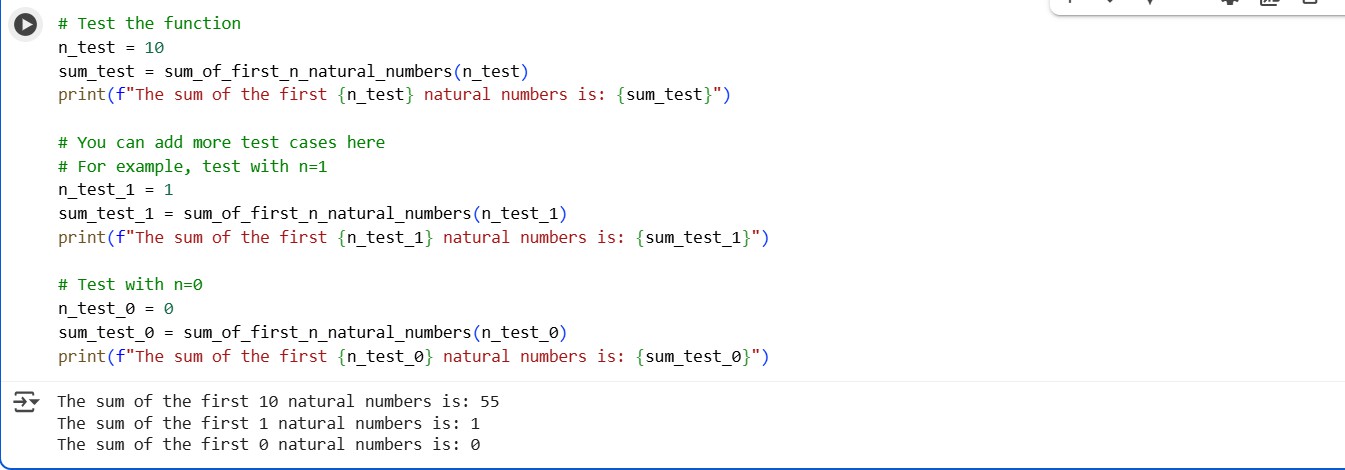
**Prompt:**

****

**Response from cursor ai:**

****

**Response from gemini:**

****

**Comparision of both outputs:**

Let’s clearly compare them:

* **Cursor AI Code (Formula Method)**

def sum\_natural\_numbers(n: int) -> int: """

Returns the sum of the first n natural numbers.

Formula used: n \* (n + 1) // 2 """

return n \* (n + 1) // 2

# Test example N = 10

print(f"Sum of first {N} natural numbers:", sum\_natural\_numbers(N))

⬛ **Output**

Sum of first 10 natural numbers: 55

* **Gemini Code (Loop + Multiple Tests)**

def sum\_of\_first\_n\_natural\_numbers(n: int) -> int: total = 0

for i in range(1, n + 1):

total += i return total

# Test the function n\_test = 10

sum\_test = sum\_of\_first\_n\_natural\_numbers(n\_test)

print(f"The sum of the first {n\_test} natural numbers is: {sum\_test}")

# More test cases n\_test\_1 = 1

sum\_test\_1 = sum\_of\_first\_n\_natural\_numbers(n\_test\_1)

print(f"The sum of the first {n\_test\_1} natural numbers is: {sum\_test\_1}")

n\_test\_0 = 0

sum\_test\_0 = sum\_of\_first\_n\_natural\_numbers(n\_test\_0)

print(f"The sum of the first {n\_test\_0} natural numbers is: {sum\_test\_0}")

⬛ **Output**

The sum of the first 10 natural numbers is: 55 The sum of the first 1 natural numbers is: 1

The sum of the first 0 natural numbers is: 0

* **Comparison**

**Feature Cursor AI Output Gemini Output**

**Method** Formula (n\*(n+1)//2) → very efficient

(O(1))

Loop (for i in range) → less efficient (O(n))

**Code Length** Short & optimized Longer, step-by-step

**Readability** May look tricky to beginners (uses math formula)

Easy to understand for beginners

**Tests** Only tested with N=10 Tested with N=10, N=1, and N=0

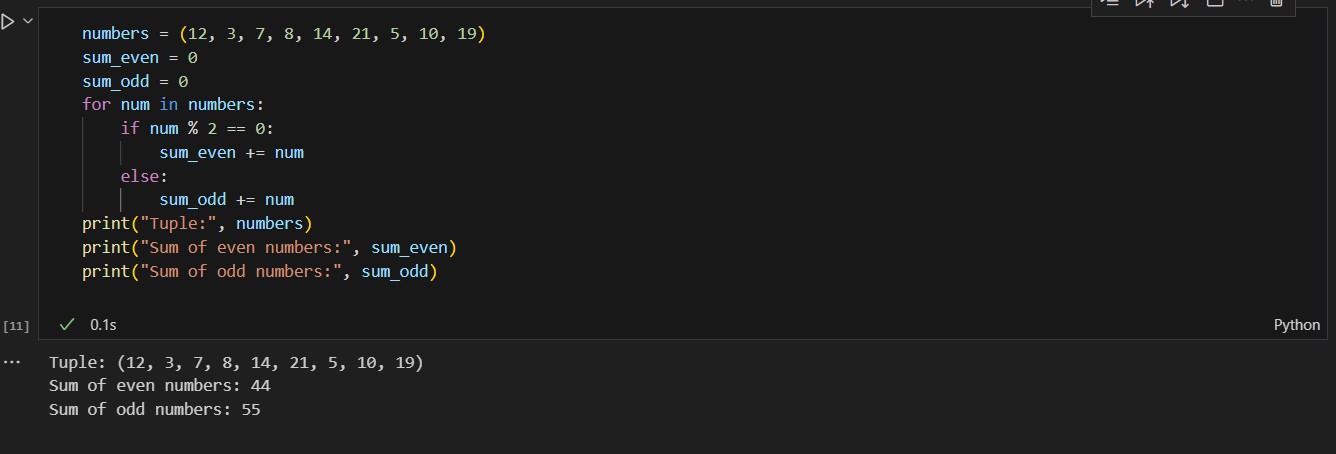
**Performance** Best for large N Slower for very large N

−– Both are **correct**.

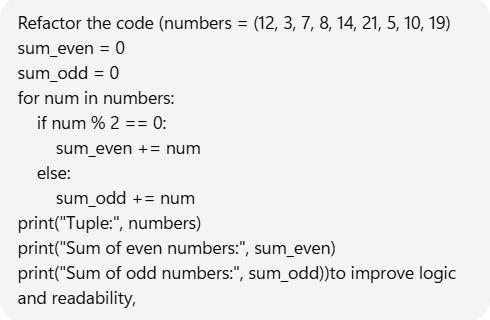
* **Cursor AI** gave a **mathematical shortcut**.
* **Gemini** gave a **beginner-friendly approach with more test cases**

**Task5:**

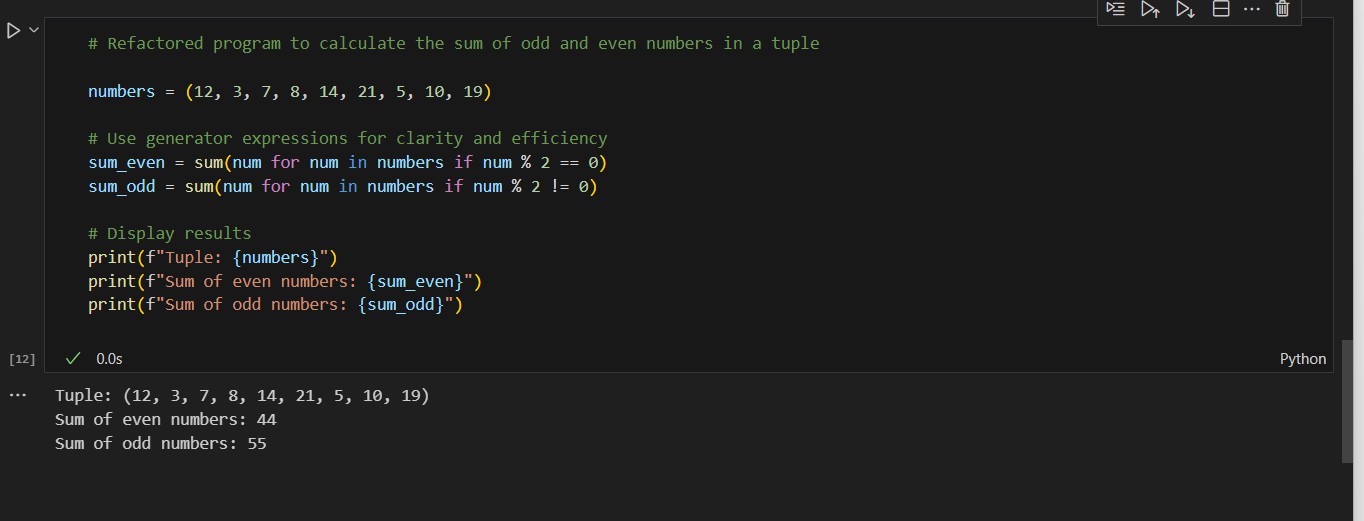
**Written code with output:**



Prompt:



Response:



x+˙ **Improvements Made**

1. **Removed manual loop** → replaced with **generator expressions inside sum()** (concise + readable).
2. **Used f-strings** for cleaner output formatting.
3. **Kept variable names meaningful** (sum\_even, sum\_odd).

⬛ **Example Output**

Tuple: (12, 3, 7, 8, 14, 21, 5, 10, 19)

Sum of even numbers: 44 Sum of odd numbers: 55

* + **Code Recap**

numbers = (12, 3, 7, 8, 14, 21, 5, 10, 19)

# Use generator expressions for clarity and efficiency

sum\_even = sum(num for num in numbers if num % 2 == 0) sum\_odd = sum(num for num in numbers if num % 2 != 0)

print(f"Tuple: {numbers}")

print(f"Sum of even numbers: {sum\_even}") print(f"Sum of odd numbers: {sum\_odd}")

* + **Explanation**

1. **The Tuple**
2. numbers = (12, 3, 7, 8, 14, 21, 5, 10, 19)
   * A tuple (like a list, but immutable) holds the numbers we want to analyze.
3. **Finding Sum of Even Numbers**
4. sum\_even = sum(num for num in numbers if num % 2 == 0)
   * num for num in numbers if num % 2 == 0 → This is a **generator expression** that picks only even numbers from the tuple.
   * sum(...) → Adds them all together.
   * So effectively, it’s summing only the **even numbers**.

⬛ Example: From (12, 3, 7, 8, 14, 21, 5, 10, 19) → it picks (12, 8, 14, 10) → sum = **44**.

1. **Finding Sum of Odd Numbers**
2. sum\_odd = sum(num for num in numbers if num % 2 != 0)
   * Same idea, but checks if the number is **not divisible by 2** (!= 0).
   * This picks out only the odd numbers.

⬛ Example: Picks (3, 7, 21, 5, 19) → sum = **55**.

1. **Printing Results with f-strings**
2. print(f"Tuple: {numbers}")
3. print(f"Sum of even numbers: {sum\_even}")
4. print(f"Sum of odd numbers: {sum\_odd}")
   * f"..." allows us to directly embed variables in the string (cleaner than concatenation).
   * Outputs the tuple and both sums in a nice format.

* **Why This Version is Better?**
* **Concise** → No manual loop, no extra variables.
* **Readable** → Each line clearly shows what it calculates.
* **Efficient** → Generator expressions avoid creating temporary lists.