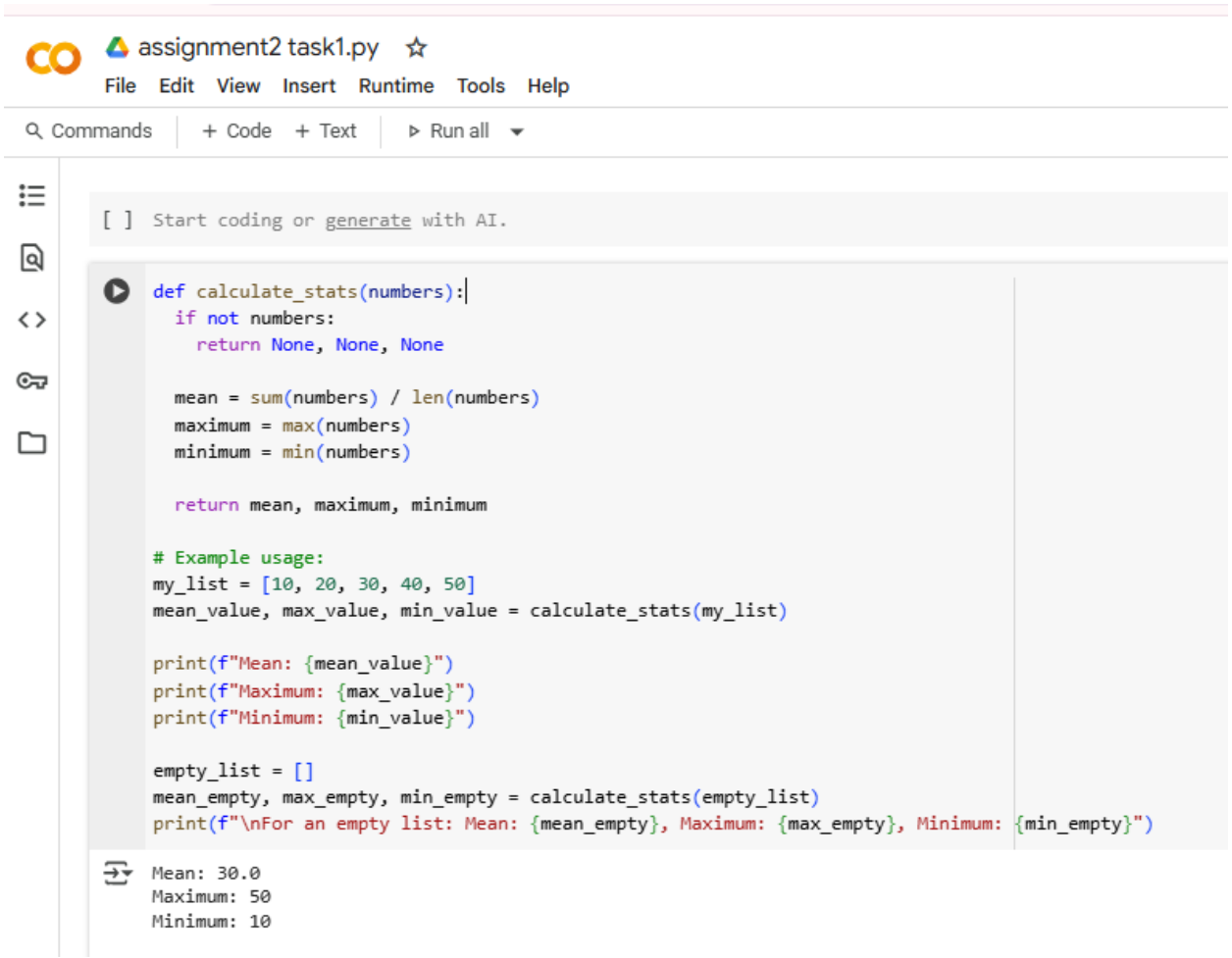
ASSIGNMENT-2.1

TASK-1:

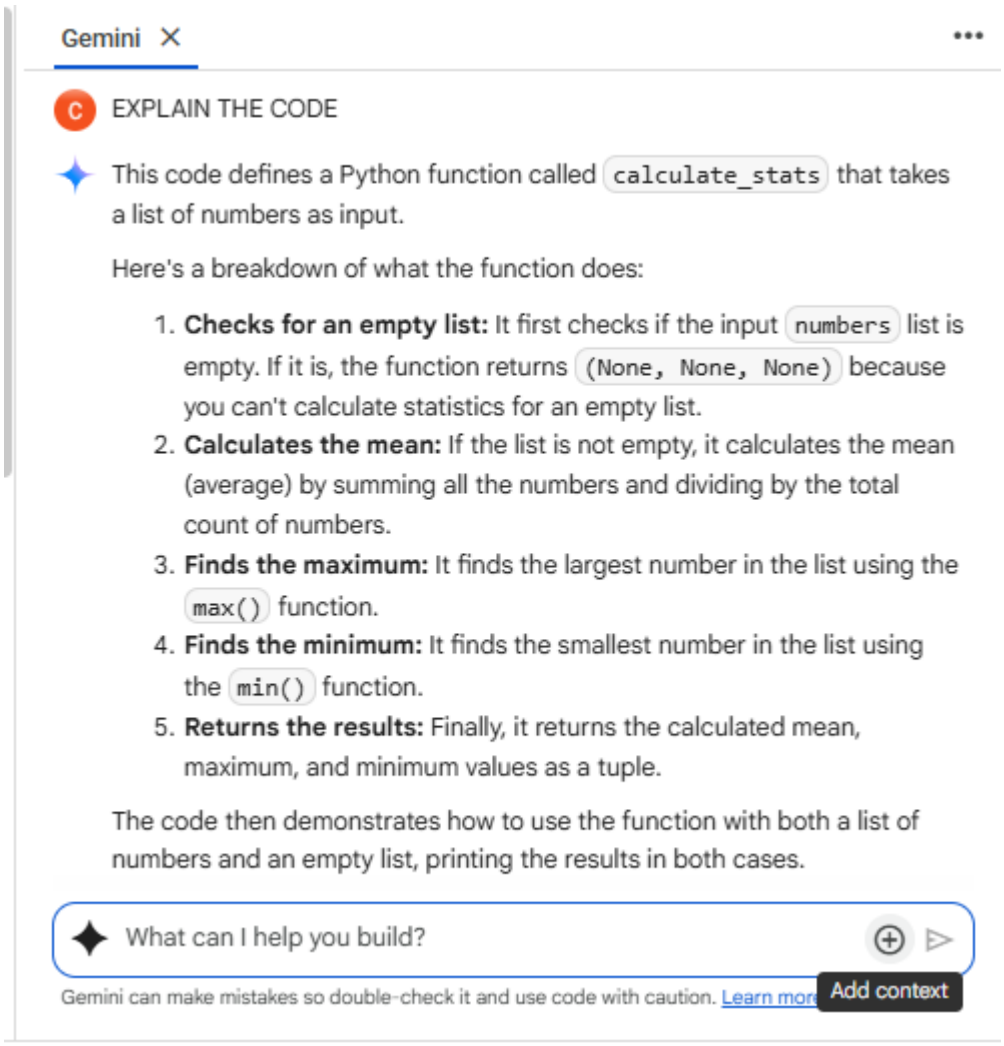
Prompt:

#write a python program to calculate mean, maximum, minimum using functions.

Code and Output:



Explanation:



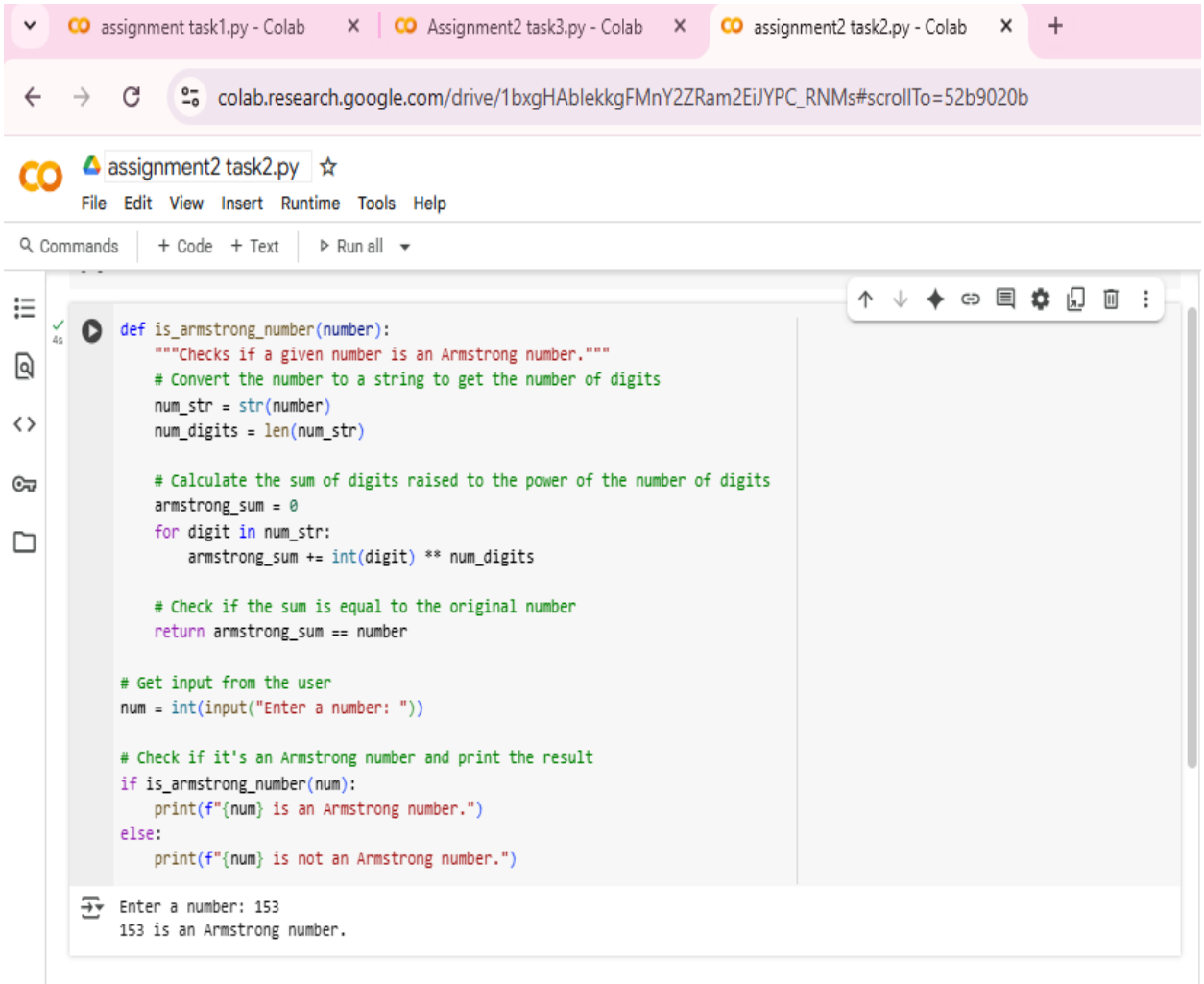
TASK-2:

Prompt:

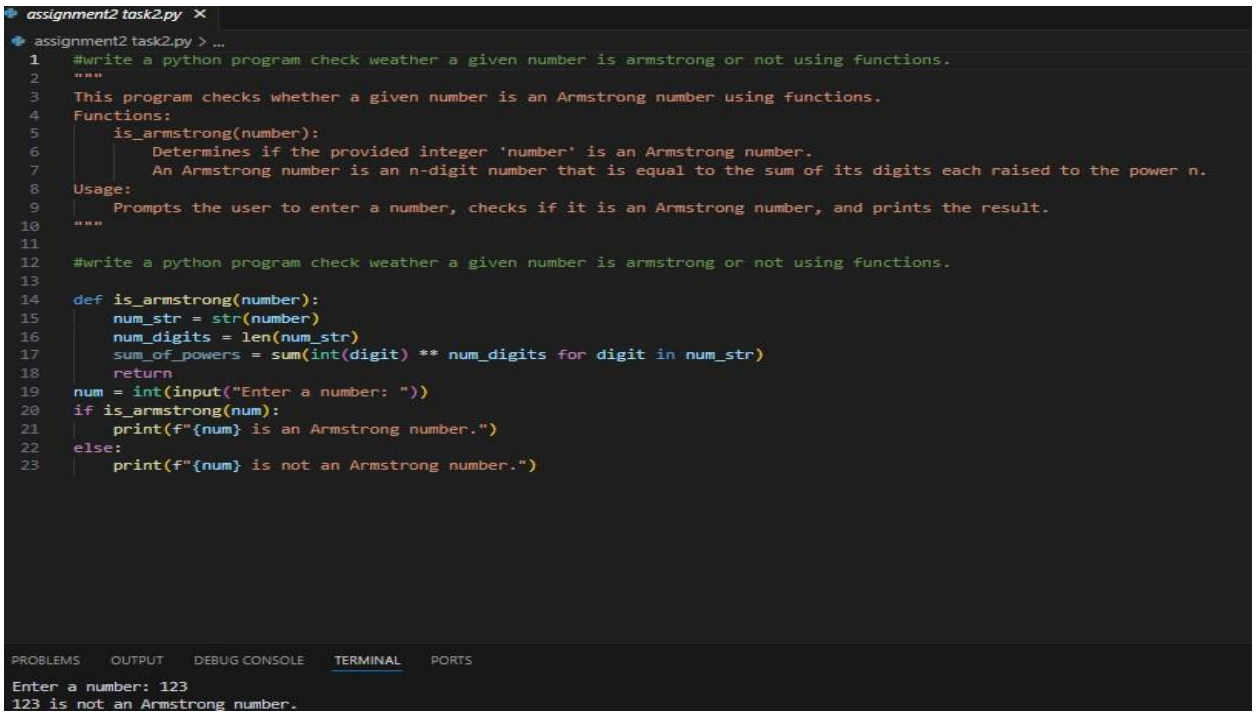
#write a python code to calculate Armstrong number using functions.

Code and Output:

Armstrong number code and output with Gemini AI:



Armstrong Number code and output with GITHUB Copilot:



```
assignment2 task2.py
#write a python program check weather a given number is armstrong or not using functions.
"""
This program checks whether a given number is an Armstrong number using functions.
Functions:
    is_armstrong(number):
        Determines if the provided integer 'number' is an Armstrong number.
        An Armstrong number is an n-digit number that is equal to the sum of its digits each raised to the power n.
Usage:
    Prompts the user to enter a number, checks if it is an Armstrong number, and prints the result.
"""

#write a python program check weather a given number is armstrong or not using functions.

def is_armstrong(number):
    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
    return
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.")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Enter a number: 123
123 is not an Armstrong number.
```

Explanation of both codes Gemini AI and GITHUB copoilt:

explain the code line by line

No problem! Here's a line-by-line explanation of the code:

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

This defines a function called `is_armstrong_number` that takes one argument, `number`. The docstring explains that this function checks if the input `number` is an Armstrong number.

```
    # Convert the number to a string to get the 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 using the `len()` function, storing the result in `num_digits`.

```
    # Calculate the sum of digits raised to the power of the number of digits
    armstrong_sum = 0
    for digit in num_str:
        armstrong_sum += int(digit) ** num_digits
```

An `armstrong_sum` variable is initialized to 0. Then, the code iterates through each `digit` in the `num_str`. For each digit, it's converted back to an integer (`int(digit)`) and raised to the power of `num_digits`. This result is added to `armstrong_sum`.

```
    # Check if the sum is equal to the original number
    return armstrong_sum == number
```

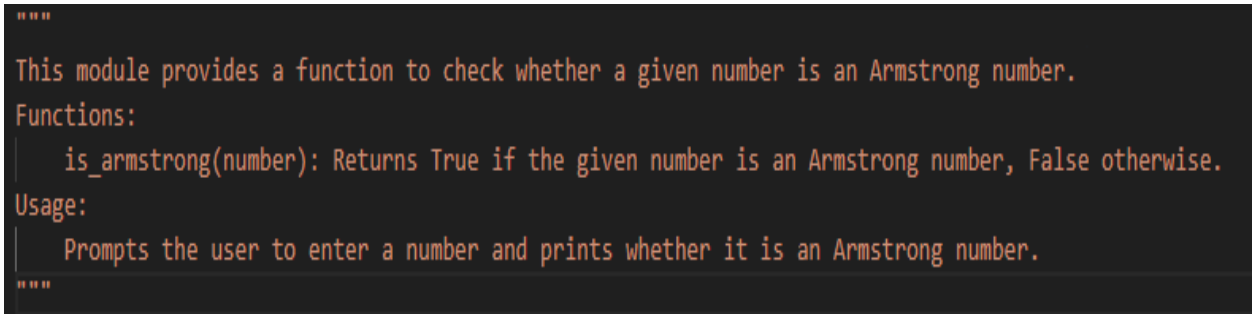
Finally, the function returns `True` if the calculated `armstrong_sum` is equal to the original `number`, indicating it's an Armstrong number, otherwise it returns `False`.

```
# Get input from the user
num = int(input("Enter a number: "))
```

This line prompts the user to "Enter a number:" and stores the input, converted to an integer, in the variable `num`.

```
# Check if it's an Armstrong number and print the result
if is_armstrong_number(num):
    print(f"{num} is an Armstrong number.")
else:
    print(f"{num} is not an Armstrong number.")
```

This `if` statement calls the `is_armstrong_number()` function with the user's input `num`. If the function returns `True`, it prints that `num` is an Armstrong number. Otherwise, it prints that `num` is not an Armstrong number.



```
"""
This module provides a function to check whether a given number is an Armstrong number.
Functions:
    is_armstrong(number): Returns True if the given number is an Armstrong number, False otherwise.
Usage:
    Prompts the user to enter a number and prints whether it is an Armstrong number.
"""
```

Comparing both the Code and Outputs:

The two codes shown aim to determine whether a given number is an Armstrong number, but only one of them functions correctly. In the first code (VS Code), although the logic to compute the sum of the digits raised to the power of their count is correctly written, the function lacks a proper `return` statement. It only has `return` without any value, so it returns `None` by default. This causes the program to always treat the result as `False`, printing that the number is not an Armstrong number—even when it is (like 153).

On the other hand, the second code (Colab) implements the logic correctly. It uses a loop to compute the sum and compares it to the original number, returning a Boolean value (`True` or `False`). The function in the second code includes clear comments, a docstring, and is structured in a more readable and understandable way.

Because of the logical mistake in the first code, the output is incorrect, while the second code produces the correct result. This highlights the importance of having a complete and functional `return` statement in Python functions. A small oversight like this can lead to entirely wrong outcomes, even if most of the logic is correct. Always test your function to ensure it returns expected results, especially for edge cases or known values like 153.

TASK-3:

Prompt:

#write a python program to check weather a given number is prime or not.

Code and Output:

Explanation:

explain code line by line

Let's break down the code line by line:

```
def is_prime(num):
```

This line defines a function named `is_prime` that takes one argument, `num`. This function is designed to check if `num` is a prime number.

```
  """Checks if a number is prime."""
```

This is a docstring, which is a brief explanation of what the `is_prime` function does. It's helpful for understanding the code.

```
  if num <= 1:
    return False
```

This is an `if` statement that checks if the input number `num` is less than or equal to 1. By definition, prime numbers are greater than 1, so if `num` is 1 or less, the function immediately returns `False`, indicating it's not prime.

```
  for i in range(2, int(num**0.5) + 1):
```

This line starts a `for` loop. It iterates through a sequence of numbers starting from 2 up to the integer part of the square root of `num`, plus 1. We only need to check for divisors up to the square root of `num` because if a number has a divisor larger than its square root, it must also have a divisor smaller than its square root.

```
    if num % i == 0:
      return False
```

Inside the loop, this `if` statement checks if `num` is perfectly divisible by the current number `i` (using the modulo operator `%`). If `num` is divisible by `i`, it means `num` has a divisor other than 1 and itself, so it's not prime. In this case, the function immediately returns `False`.

```
  return True
```

If the `for` loop completes without finding any divisors (meaning the `if num % i == 0:` condition was never met), it means `num` is not divisible by any number from 2 up to its square root. Therefore, `num` must be prime, and the function returns `True`.

```
def find_primes_up_to(limit):
```

This line defines another function named `find_primes_up_to` that takes one argument, `limit`. This function will find all prime numbers up to and including this `limit`.

```
  """Finds all prime numbers up to a given limit."""
```

This is the docstring for the `find_primes_up_to` function, explaining its purpose.

```
    prime_numbers = []
```

This line initializes an empty list called `prime_numbers`. This list will store the prime numbers found within the specified `limit`.

```
    for num in range(2, limit + 1):
```

This line starts a `for` loop that iterates through numbers starting from 2 up to `limit` (inclusive).

```
        if is_prime(num):
```

Inside this loop, for each number `num`, this line calls the `is_prime` function we defined earlier. It checks if the current number `num` is prime.

```
            prime_numbers.append(num)
```

If the `is_prime(num)` function returns `True` (meaning `num` is prime), this line adds the current number `num` to the `prime_numbers` list.

```
    return prime_numbers
```

After the loop finishes checking all numbers up to `limit`, this line returns the `prime_numbers` list, which now contains all the prime numbers found.

```
# Example usage: Find prime numbers up to 50
```

This is a comment explaining that the following lines demonstrate how to use the functions.

```
limit = 50
```

This line sets a variable `limit` to the value 50. This is the upper bound for finding prime numbers in this example.

```
prime_list = find_primes_up_to(limit)
```

This line calls the `find_primes_up_to` function with the `limit` of 50. The list of prime numbers returned by the function is stored in the variable `prime_list`.

```
print(f"Prime numbers up to {limit}: {prime_list}")
```

This line uses an f-string to print the result to the console. It displays the text "Prime numbers up to", followed by the value of `limit` (which is 50), a colon, and then the contents of the `prime_list`.

TASK-4:

Prompt:

#write a python program to find sum of N natural numbers using functions.

Code and Output:

Explanation:

What the program does

- **Goal:** Read a positive integer N, then print the sum of the first N natural numbers (1 + 2 + ... + N).

Key parts

- `sum_natural_numbers(n)`:
  - Returns 0 for non-positive `n`.
  - Uses the formula `n * (n + 1) // 2` to compute the sum in O(1) time.
  - `//` is integer division, ensuring an integer result.
- `main()`:
  - Prompts the user: "Enter a positive integer N:".
  - Converts input to `int`; on failure, shows "Invalid input!".
  - If `n > 0`, calls `sum_natural_numbers(n)` and prints the result.
  - Otherwise asks for a positive integer.
- **Entry point:**
  - `if __name__ == "__main__": main()` runs `main()` only when you execute this file directly.

Example

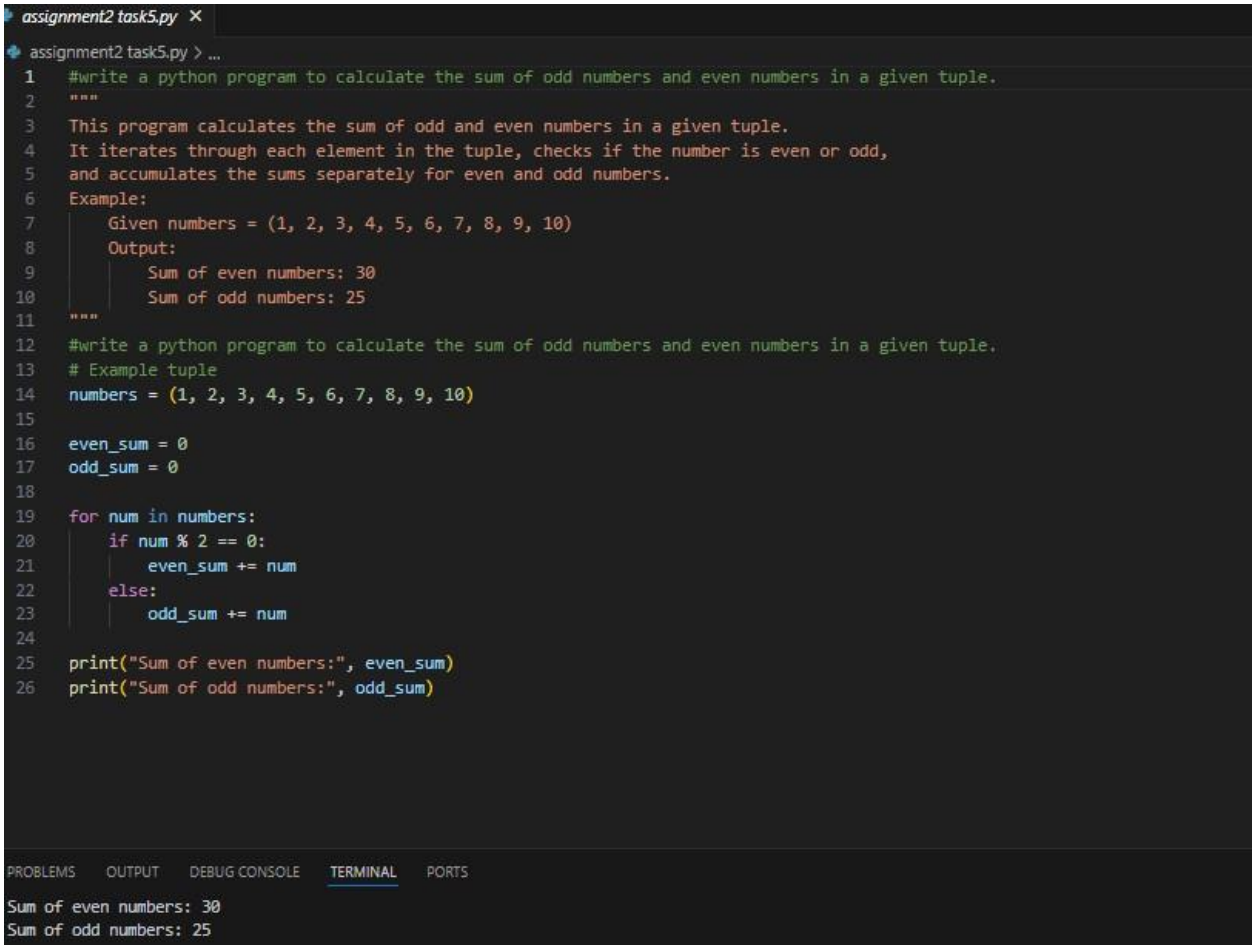
- Input: 5 → Output: "The sum of first 5 natural numbers is: 15" (since 1+2+3+4+5 = 15)
- The program validates input and handles errors gracefully while using an efficient formula.
- **Summary:**
  - Function-based solution with fast O(1) formula.
  - Robust input parsing and validation.
  - Clear separation of logic (`sum_natural_numbers`) and I/O (`main`).

TASK-5

Prompt:

#write a python program to calculate the sum of odd numbers and even numbers in a given tuple.

Code and Output:



Explanation: