Lab assignment-02

NAME: CH.VISHWANATH RAO

BATCH:02

HT NO:2303A51095

Task 1: Statistical Summary for Survey Data

Scenario:

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

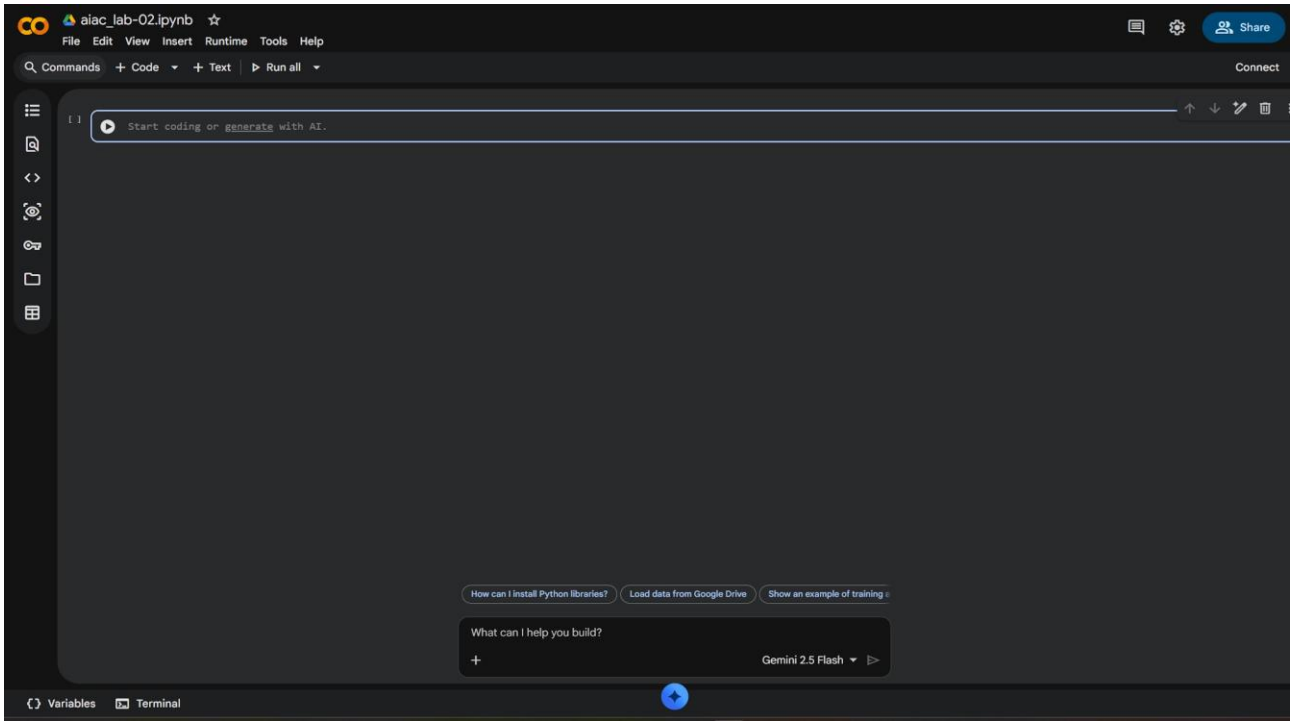
Task:

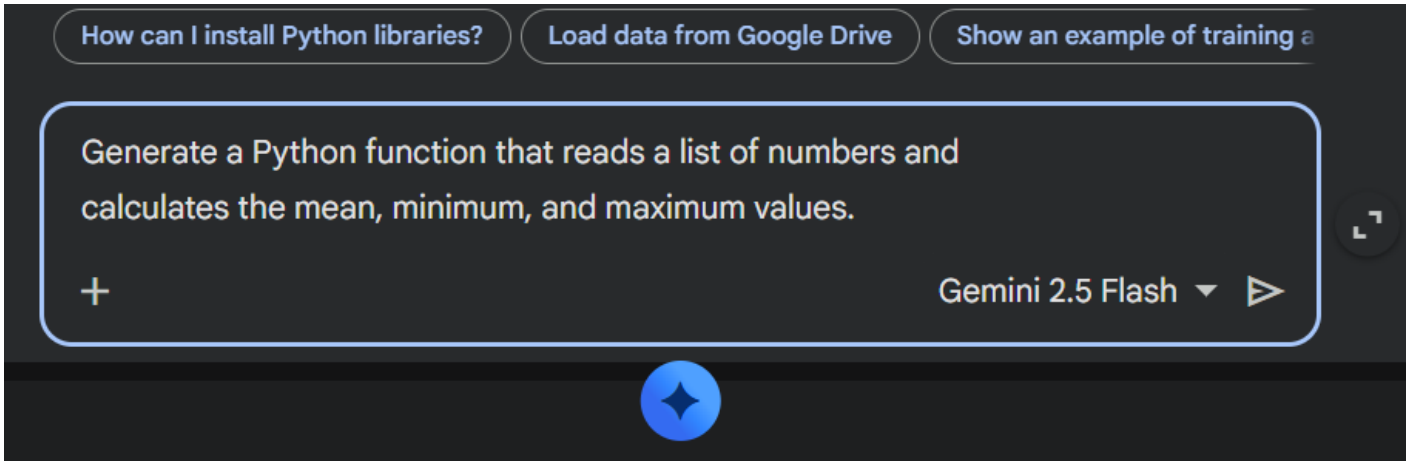
● Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

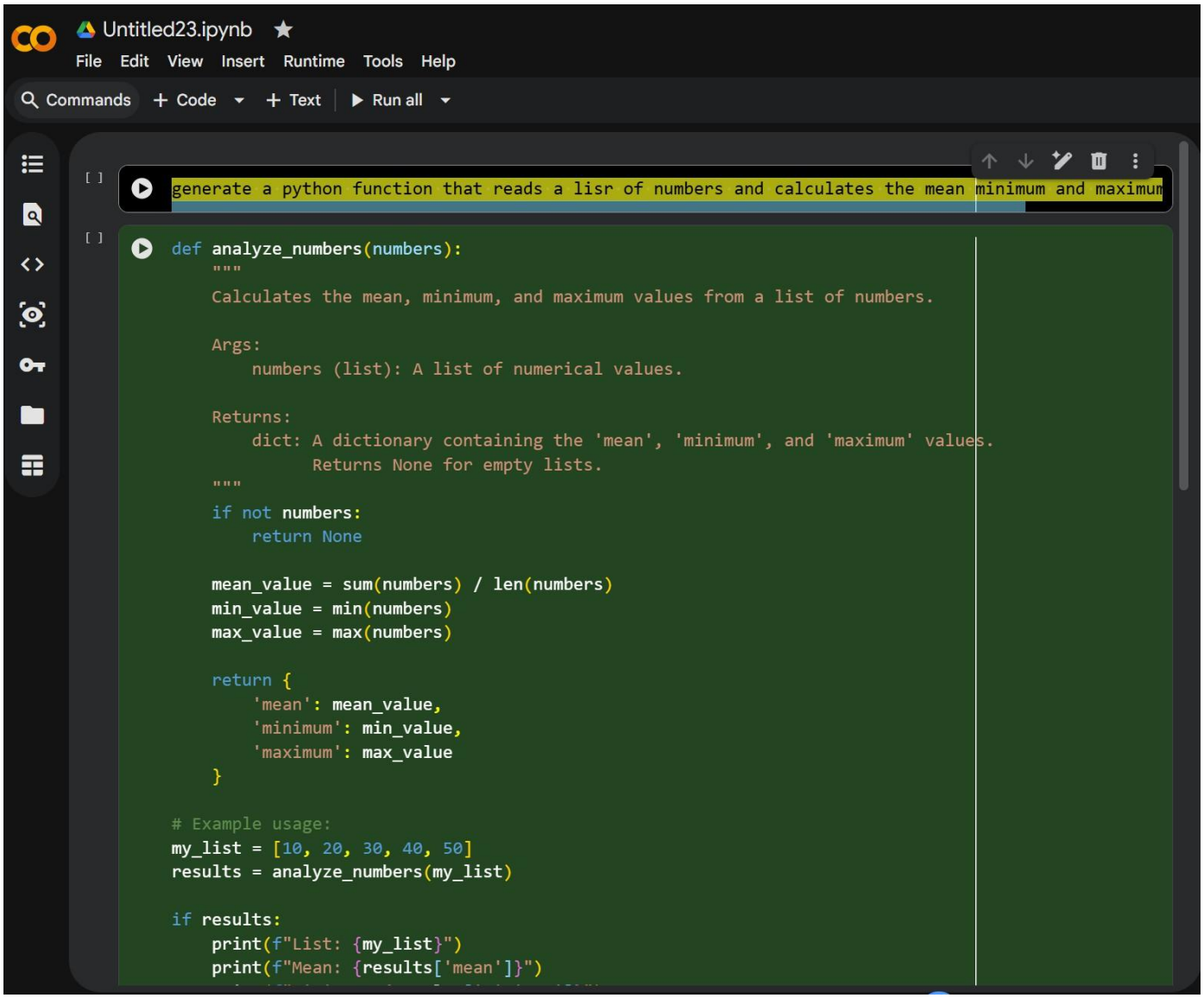
Expected Output:

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

Screenshots:




How can I install Python libraries?
Load data from Google Drive
Show an example of training a
Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.
Gemini 2.5 Flash


Untitled23.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
generate a python function that reads a lisr of numbers and calculates the mean minimum and maximum
def analyze_numbers(numbers):
"""
Calculates the mean, minimum, and maximum values from a list of numbers.
Args:
numbers (list): A list of numerical values.
Returns:
dict: A dictionary containing the 'mean', 'minimum', and 'maximum' values.
Returns None for empty lists.
"""
if not numbers:
return None
mean_value = sum(numbers) / len(numbers)
min_value = min(numbers)
max_value = max(numbers)
return {
'mean': mean_value,
'minimum': min_value,
'maximum': max_value
}
# Example usage:
my_list = [10, 20, 30, 40, 50]
results = analyze_numbers(my_list)
if results:
print(f"List: {my_list}")
print(f"Mean: {results['mean']}")

Code:

```
def analyze_numbers(numbers):
    """
```

```
    Calculates the mean, minimum, and maximum values from a list of
numbers.

    Args:
        numbers (list): A list of numerical values.

    Returns:
        dict: A dictionary containing the 'mean', 'minimum', and
'maximum' values.
              Returns None for empty lists.
    """
    if not numbers:
        return None

    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

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

# Example usage:
my_list = [10, 20, 30, 40, 50]
results = analyze_numbers(my_list)

if results:
    print(f"List: {my_list}")
    print(f"Mean: {results['mean']}")
```

```
        (f"Minimum: {        ['minimum']}")
        (f"Maximum: {        ['maximum']}")
else:
        ("The list is empty.")


            []
                                ( )


if              :
        (f"Empty List Results: {             }")
else:
```

```
    print(f"Empty List: {empty_list}")
    print("The list is empty.")
```

output:

```
List: [10, 20, 30, 40, 50]

Mean: 30.0

Minimum: 10

Maximum: 50
```

Task 2: Armstrong Number – AI Comparison

Scenario:

- You are evaluating AI tools for numeric validation logic.

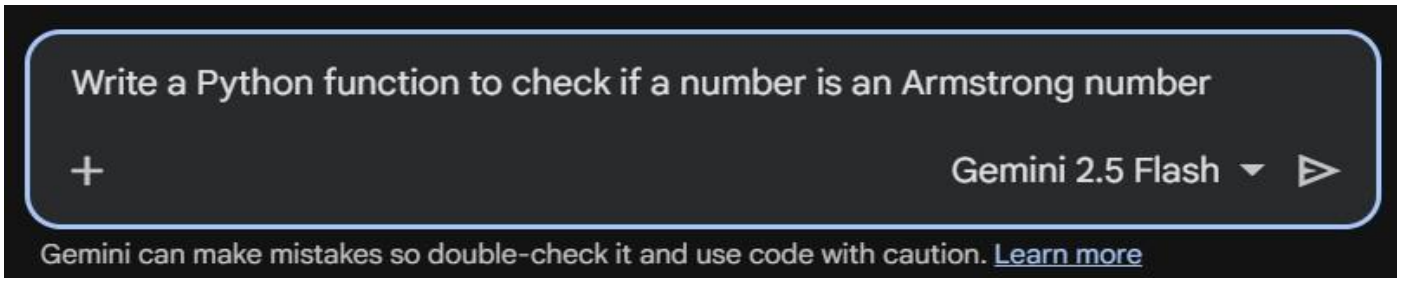
Task:

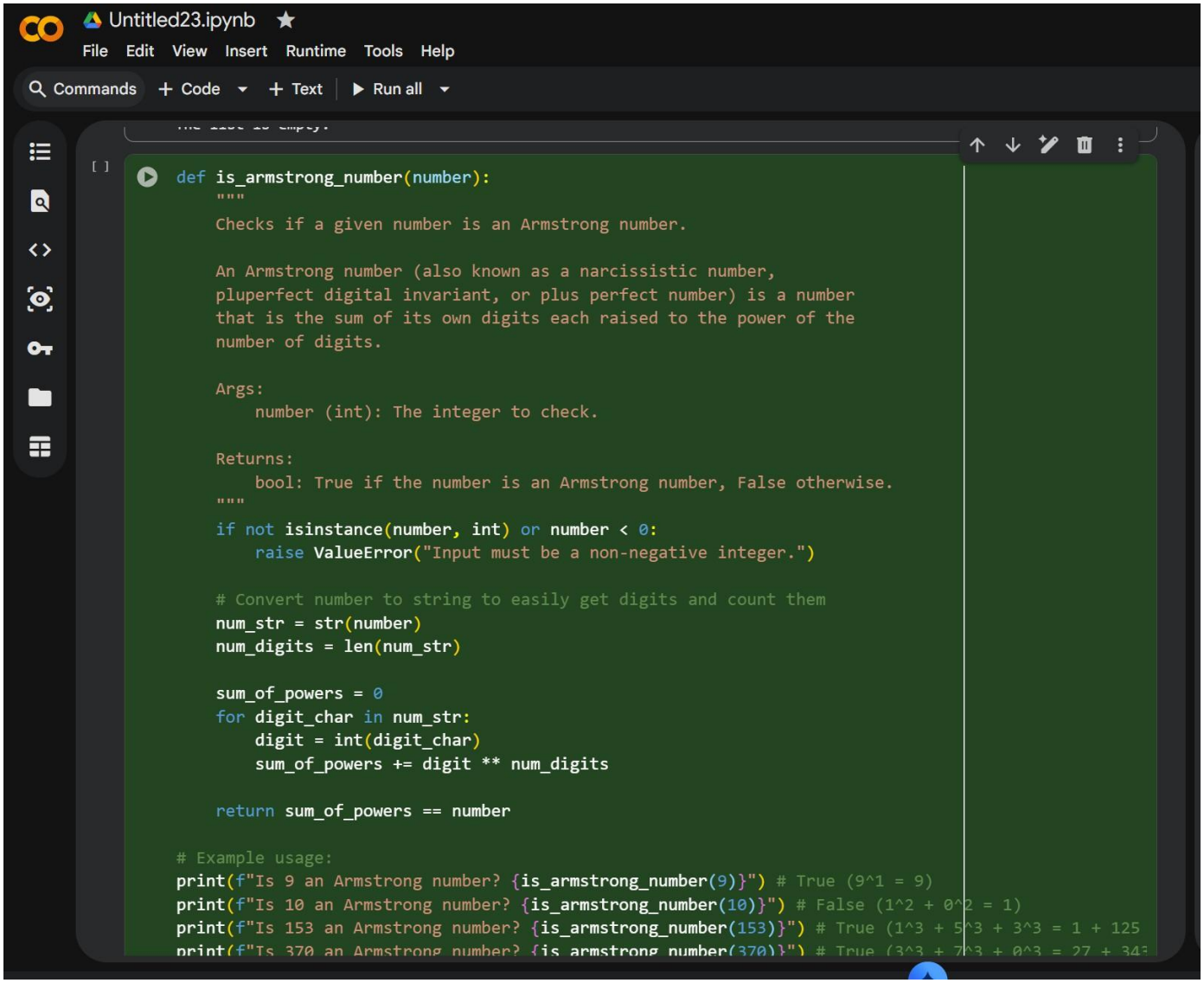
- Generate an Armstrong number checker using Gemini and GitHub Copilot.
- Compare their outputs, logic style, and clarity.

Expected Output:

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

Google Colab Workspace Screenshots:




Untitled23.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
def is_armstrong_number(number):
"""
Checks if a given number is an Armstrong number.
An Armstrong number (also known as a narcissistic number,
pluperfect digital invariant, or plus perfect 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): The 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 easily get digits and count 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 9 an Armstrong number? {is_armstrong_number(9)}") # True (9^1 = 9)
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}") # False (1^2 + 0^2 = 1)
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # True (1^3 + 5^3 + 3^3 = 1 + 125

Code:

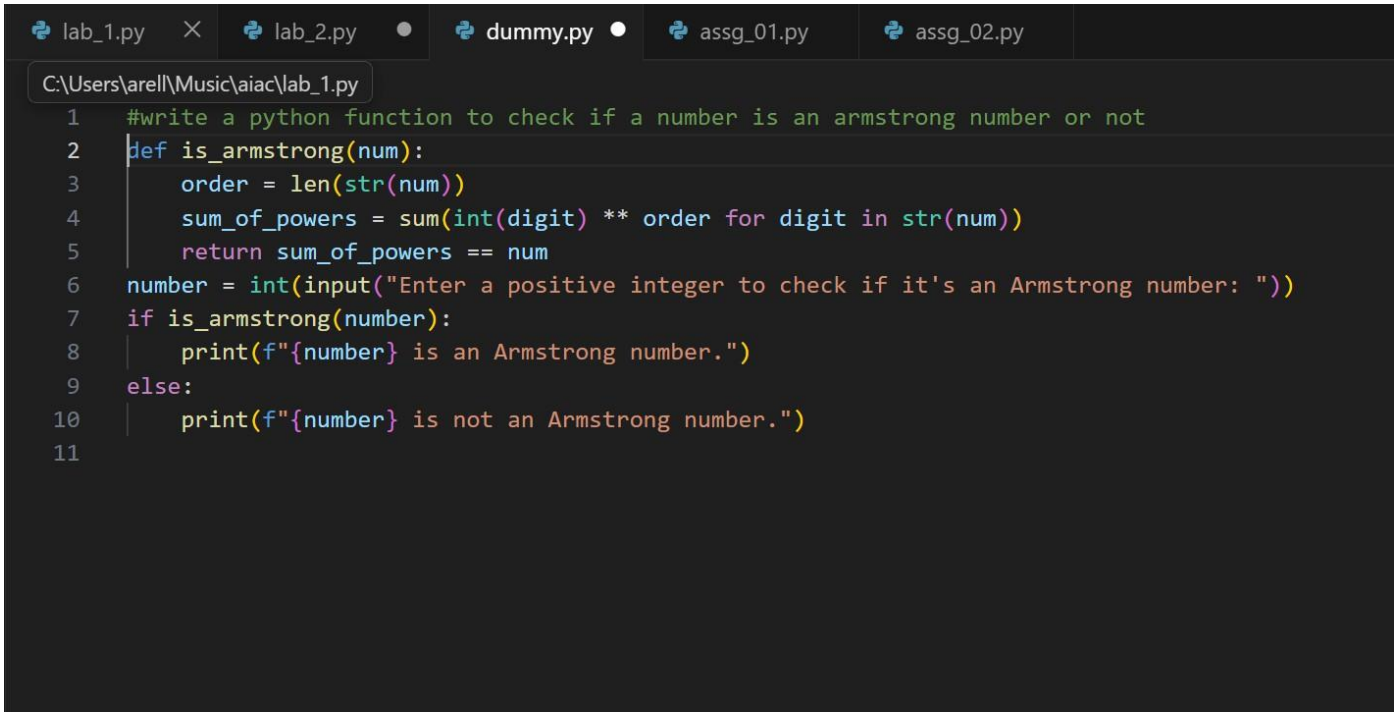
```
def is_armstrong_number(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number,
    pluperfect digital invariant, or plus perfect 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): The integer to check.
```

```
    """
    if not isinstance(number, int) or number < 0:
        raise ValueError("Input must be a non-negative integer.")

    # Convert number to string to easily get digits and count 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 9 an Armstrong number? {is_armstrong_number(9)}") # True (9^1 = 9)
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}") # False (1^2 + 0^2 = 1)
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # True (3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370)
    (f"Is 371 an Armstrong number? {          (371)}") # True (3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371)
    (f"Is 1634 an Armstrong number? {          (1634)}") # True (1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634)
    (f"Is 1635 an Armstrong number? {          (1635)}") # False
```

output:

```
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 1634 an Armstrong number? True

Is 1635 an Armstrong number? False
```

Vs code :



Code:

```
#write a python function to check if a number is an armstrong number or not def is_armstrong(num):

    order = len(str(num))

    sum_of_powers = sum(int(digit) ** order for digit in str(num)) return sum_of_powers ==
    num

number = int(input("Enter a positive integer to check if it's an Armstrong number: ")) if is_armstrong(number):

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

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

output:

```
Enter a positive integer to check if it's an Armstrong number: 153
153 is an Armstrong number.
```

Task 3: Leap Year Validation Using Cursor AI

Scenario:

● You are validating a calendar module for a backend system.

Task:

● Use Cursor AI to generate a Python program that checks whether a given year is

a leap year.

● Use at least two different prompts and observe changes in code.

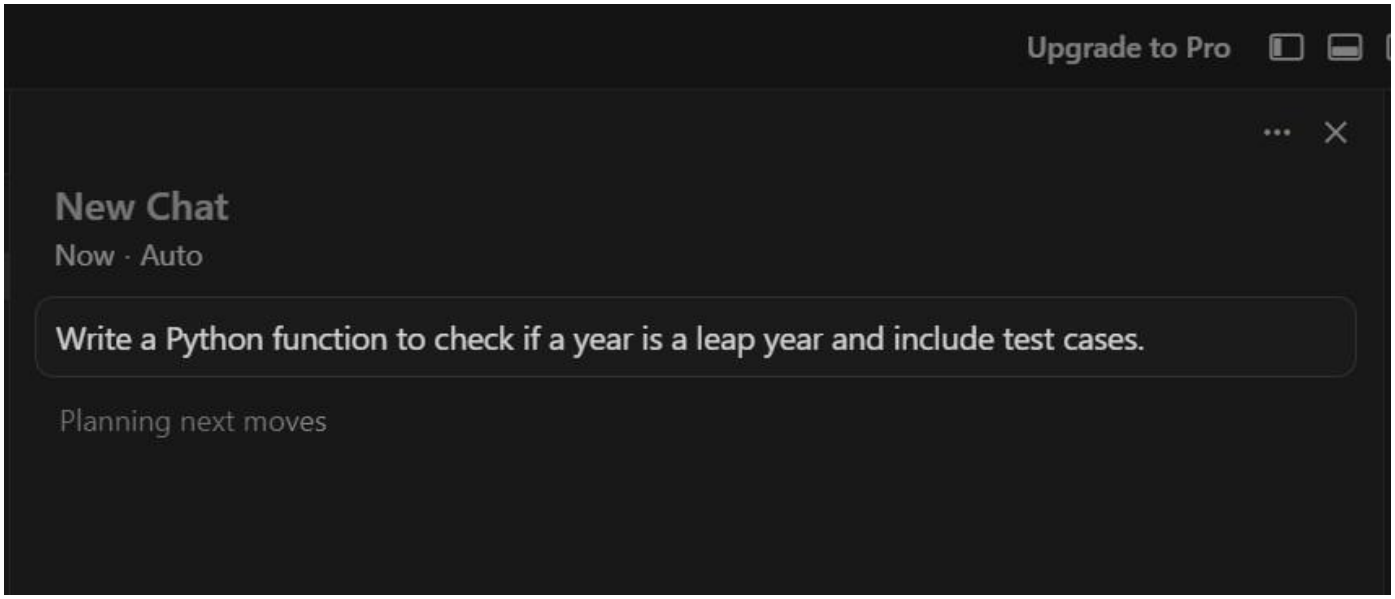
Expected Output:

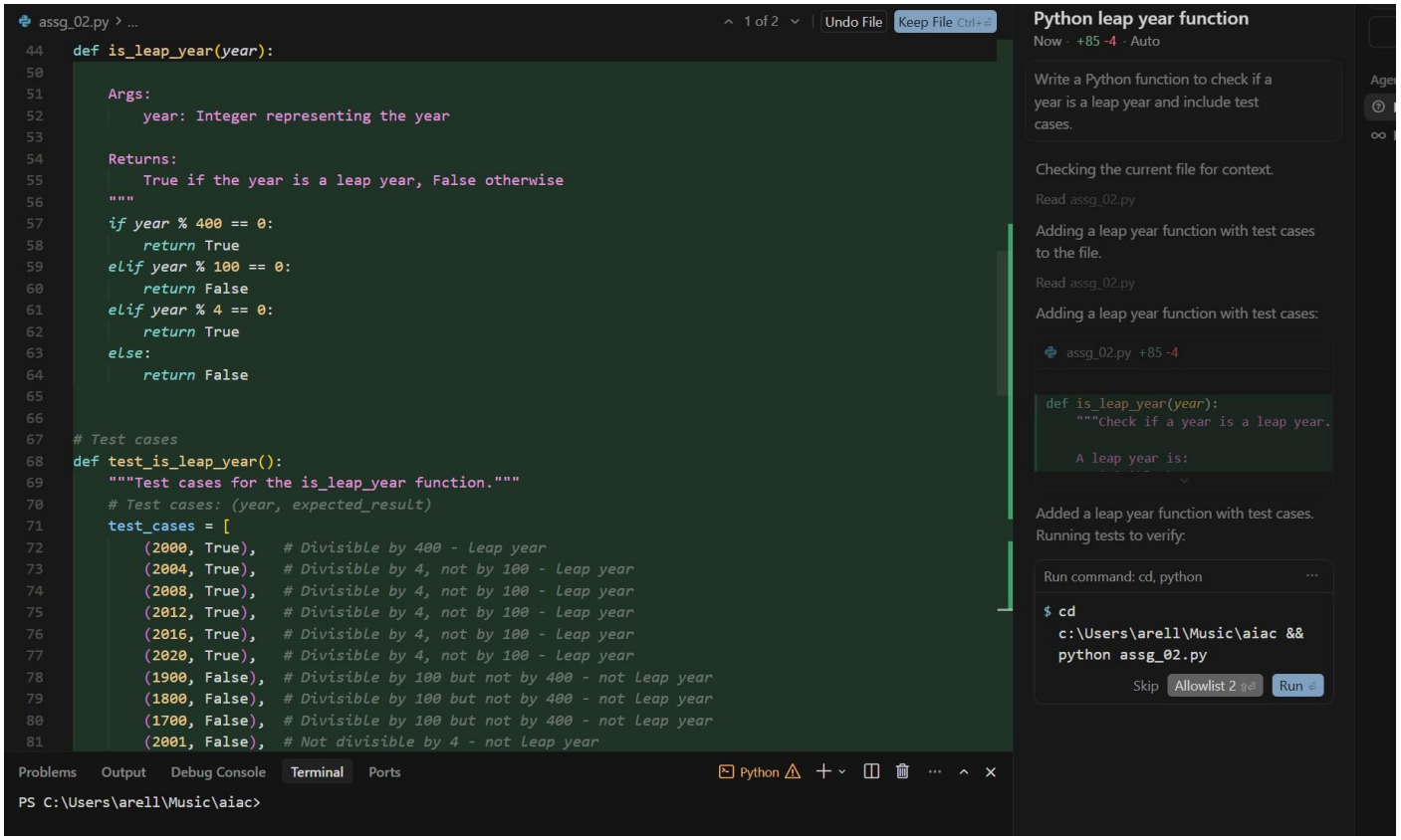
● Two versions of code
● Sample inputs/outputs
● Brief comparison

Version - 1:

Instruction Given:

Write a Python function to check if a year is a leap year and include test cases.




assg_02.py > ...
1 of 2
Undo File
Keep File Ctrl+⏎
def is_leap_year(year):
"""
Args:
year: Integer representing the year
Returns:
True if the year is a leap year, False otherwise
"""
if year % 400 == 0:
return True
elif year % 100 == 0:
return False
elif year % 4 == 0:
return True
else:
return False
# Test cases
def test_is_leap_year():
"""Test cases for the is_leap_year function."""
# Test cases: (year, expected_result)
test_cases = [
(2000, True), # Divisible by 400 - leap year
(2004, True), # Divisible by 4, not by 100 - leap year
(2008, True), # Divisible by 4, not by 100 - leap year
(2012, True), # Divisible by 4, not by 100 - leap year
(2016, True), # Divisible by 4, not by 100 - leap year
(2020, True), # Divisible by 4, not by 100 - leap year
(1900, False), # Divisible by 100 but not by 400 - not leap year
(1800, False), # Divisible by 100 but not by 400 - not leap year
(1700, False), # Divisible by 100 but not by 400 - not leap year
(2001, False), # Not divisible by 4 - not leap year
Problems
Output
Debug Console
Terminal
Ports
Python
PS C:\Users\arell\Music\aiac>
Python leap year function
Now · +85 -4 · Auto
Write a Python function to check if a year is a leap year and include test cases.
Checking the current file for context.
Read assg_02.py
Adding a leap year function with test cases to the file.
Read assg_02.py
Adding a leap year function with test cases:
assg_02.py +85 -4
def is_leap_year(year):
"""Check if a year is a leap year.
A leap year is:
Added a leap year function with test cases.
Running tests to verify:
Run command: cd, python
$ cd c:\Users\arell\Music\aiac && python assg_02.py
Skip
Allowlist 2 ⇧⏎
Run ⏎

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.")"""


def even_odd_sum_tuple(num):

    """Calculate sum of even and odd numbers from 1 to num.
```

```
        Tuple of (even_sum, odd_sum)
    """
    if num < 1:
        return 0, 0

    even_sum = sum(i for i in range(2, num + 1, 2)) odd_sum = sum(i
    for i in range(1, num + 1, 2)) return even_sum, odd_sum


def get_positive_integer(prompt):
    """Get valid positive integer input from user."""
    while True:
        try:
            value = int(input(prompt))
            if value < 1:
                print("Please enter a positive number.")
                continue
            return value
        except ValueError:
            print("Invalid input. Please enter a valid integer.")


def is_leap_year(year):
    """Check if a year is a leap year.

    A leap year is:
    - Divisible by 400, OR
    - Divisible by 4 but NOT divisible by 100

    Args:
        year: Integer representing the year
```

```
    Returns:
        True if the year is a leap year, False otherwise
    """
    if year % 400 == 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 4 == 0:
        return True
    else:
        return False
```

```
# Test cases

def test_is_leap_year():

    """Test cases for the is_leap_year function."""

    # Test cases: (year, expected_result)

    test_cases = [

        (2000, True),          # Divisible by 400 - leap year


        (2004, True),   # Divisible by 4, not by 100 - leap year

        (2008, True),   # Divisible by 4, not by 100 - leap year

        (2012, True),   # Divisible by 4, not by 100 - leap year

        (2016, True),   # Divisible by 4, not by 100 - leap year

        (2020, True),   # Divisible by 4, not by 100 - leap year

        (1900, False),  # Divisible by 100 but not by 400 - not leap year (1800, False),  # Divisible by 100 but not by 400 - not leap year (1700, False),  # Divisible by 100 but not by 400 - not leap year (2001, False),  # Not divisible by 4 - not leap year

        (2002, False),  # Not divisible by 4 - not leap year

        (2003, False),          # Not divisible by 4 - not leap year

        (2100, False),          # Divisible by 100 but not by 400 - not leap year

        (2400, True),          # Divisible by 400 - leap year

    ]


    print("Running test cases for is_leap_year function:") print("-" * 50)


    passed = 0

    failed = 0


    for year, expected in test_cases: result =
        is_leap_year(year)

        status = "PASS" if result == expected else "FAIL"


        if result == expected: passed += 1

        else:
```

```
                failed += 1

        print(f"{status}: is_leap_year({year}) = {result} (expected {expected})")

print("-" * 50)
print(f"Total tests: {len(test_cases)}") print(f"Passed: {passed}")
print(f"Failed: {failed}")

return failed == 0
```

```
if __name__ == "__main__":
    # Run test cases
    test_is_leap_year()

    print("\n" + "=" * 50)
    print("Testing with user input:") print("=" * 50)

    # Interactive test try:
        year = int(input("Enter a year to check if it's a leap year: "))
        if is_leap_year(year):
            print(f"{year} is a leap year.")
```

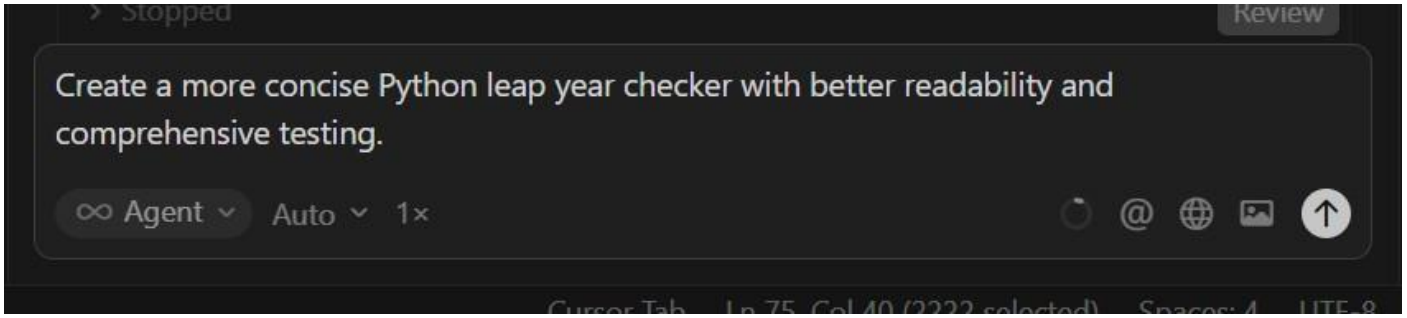
output:

```
--------------------------------------------------
Running test cases for is_leap_year function:

PASS: is_leap_year(2000) = True (expected True)
PASS: is_leap_year(2004) = True (expected True)
PASS: is_leap_year(2008) = True (expected True)
PASS: is_leap_year(2012) = True (expected True)
PASS: is_leap_year(2016) = True (expected True)
PASS: is_leap_year(2020) = True (expected True)

PASS: is_leap_year(1900) = False (expected False)
PASS: is_leap_year(1800) = False (expected False)
PASS: is_leap_year(1700) = False (expected False)
PASS: is_leap_year(2001) = False (expected False)
PASS: is_leap_year(2002) = False (expected False)
PASS: is_leap_year(2003) = False (expected False)
PASS: is_leap_year(2100) = False (expected False)
PASS: is_leap_year(2400) = True (expected True)

--------------------------------------------------

Total tests: 14
```

```
Passed: 14

Failed: 0


================================================
```

```
Testing with user input:

==========================================
```

Version—02



Code”:

```python
# Write a Python function to check if a year is a leap year
def is_leap_year(year):
    """
    Check if a year is a leap year.

    A year is a leap year if it is divisible by 4, except when it's divisible
    by 100 unless it's also divisible by 400.

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)


# Comprehensive test cases
if __name__ == "__main__":
    # Test cases organized by category
```

```
        (2100, False, "Century not divisible by 400"),


        # Regular years divisible by 4 (leap years)
        (2020, True, "Year divisible by 4"),
        (2024, True, "Year divisible by 4"),
        (2016, True, "Year divisible by 4"),
        (2004, True, "Year divisible by 4"),


        # Regular years not divisible by 4 (not leap years)
        (2021, False, "Year not divisible by 4"),
        (2023, False, "Year not divisible by 4"),
        (2019, False, "Year not divisible by 4"),
        (2022, False, "Year not divisible by 4"),


        # Edge cases
        (1, False, "Year 1"),
        (4, True, "Year 4 (first leap year)"),
        (100, False, "Year 100"),
        (400, True, "Year 400"),
    ]


    # Run all tests
    passed = 0
    failed = 0


    for year, expected, description in test_cases:
        result = is_leap_year(year)
        status = "✓" if result == expected else "✗"
        if result == expected:
            passed += 1
                f"{status} {description}: {year} -> {result}"
        else
                    1
```

```
                f"{status} {description}: {year} -> {result} (expected
{expected})"


    # Summary
          f"\n{'='*50}"
          f"Tests passed: {passed}/{len(test_cases)}"
    if          0
              f"Tests failed: {failed}/{len(test_cases)}"
    else
              "All tests passed! ✓"
```

output:

```
All test cases passed! ✓
PS C:\Users\arell\Music\aiac> & C:/Users/arell/AppData/Local/Programs/Python/Python313/python.exe c:/Users/arell/Music/aiac/assg_02.py
✓ Century divisible by 400: 2000 -> True
✓ Century divisible by 400: 1600 -> True
✓ Century divisible by 400: 2400 -> True
✓ Century not divisible by 400: 1900 -> False
✓ Century not divisible by 400: 1800 -> False
✓ Century not divisible by 400: 2100 -> False
✓ Year divisible by 4: 2020 -> True
✓ Year divisible by 4: 2024 -> True
✓ Year divisible by 4: 2016 -> True
✓ Year divisible by 4: 2004 -> True
✓ Year not divisible by 4: 2021 -> False
✓ Year not divisible by 4: 2023 -> False
✓ Year not divisible by 4: 2019 -> False
✓ Year not divisible by 4: 2022 -> False
✓ Year 1: 1 -> False
✓ Year 4 (first leap year): 4 -> True
✓ Year 100: 100 -> False
✓ Year 400: 400 -> True

================================================
Tests passed: 18/18
All tests passed! ✓
PS C:\Users\arell\Music\aiac>
```

Brief comparison:

The first approach follows a traditional, step-by-step style where the logic is clearly written using separate **if-else** conditions. Because each step is shown explicitly, the code is longer but very easy to read and understand. This makes it beginner-friendly and suitable for exams, as the flow of logic is clear and testing is usually basic.

The second approach uses a compact and Pythonic style by combining logic and reducing the number of lines of code. While it looks clean and efficient, the logic is not immediately obvious to beginners. This style focuses more on brevity and smart syntax, and it is generally preferred by experienced programmers with more comprehensive testing.

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 :

lab_1.py | lab_2.py | dummy.py

dummy.py > ...

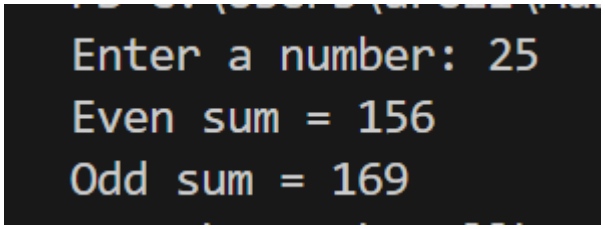
```python
def even_odd_sum(num):
    even = 0
    odd = 0
    i = 1

    while i <= num:
        if i % 2 == 0:
            even += i
        else:
            odd += i
        i += 1

    return even, odd


n = int(input("Enter a number: "))
e, o = even_odd_sum(n)

print("Even sum =", e)
print("Odd sum =", o)

```

Code:

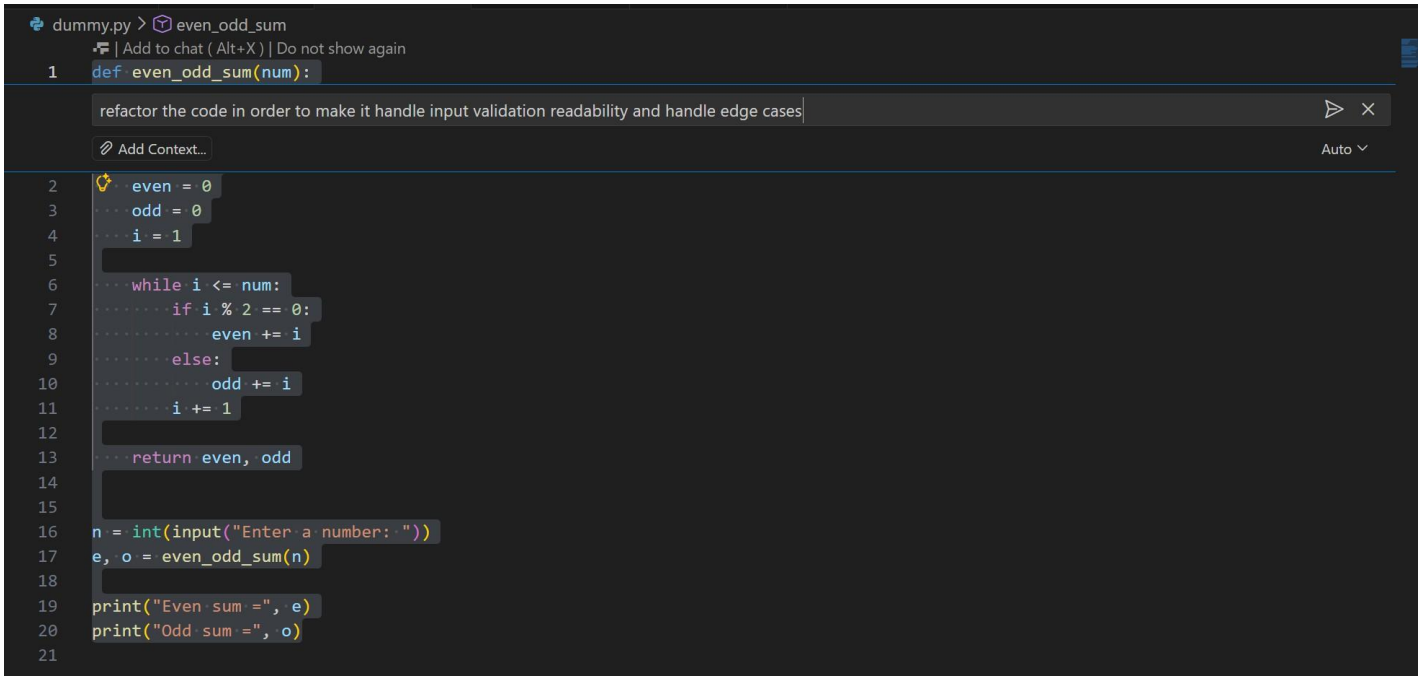
```
defeven_odd_sum(num): even =
    0

    odd = 0

    i = 1
```

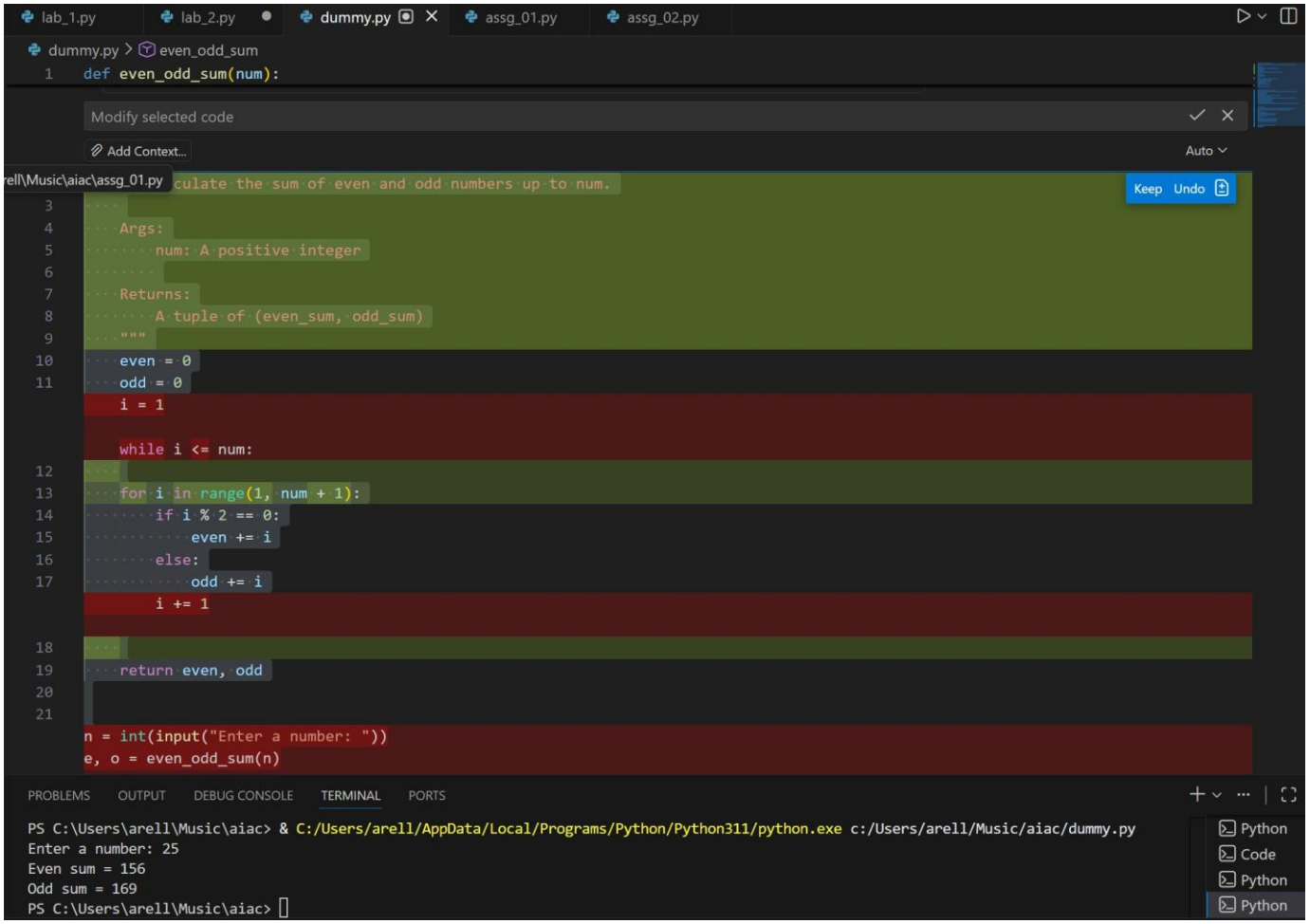
```
    while i <= num:
        if i % 2 == 0: even += i
        else:
            odd += i i
        += 1

    return even, odd

n = int(input("Enter a number: ")) e, o =
```

Output:



Refactoring the code :


dummy.py > even_odd_sum
| Add to chat (Alt+X) | Do not show again
def even_odd_sum(num):
refactor the code in order to make it handle input validation readability and handle edge cases
Add Context...
Auto
even = 0
odd = 0
i = 1
while i <= num:
if i % 2 == 0:
even += i
else:
odd += i
i += 1
return even, odd
n = int(input("Enter a number: "))
e, o = even_odd_sum(n)
print("Even sum =", e)
print("Odd sum =", o)


lab_1.py
lab_2.py
dummy.py
assg_01.py
assg_02.py
dummy.py > even_odd_sum
def even_odd_sum(num):
Modify selected code
Add Context...
Auto
rell\Music\aiac\assg_01.py
culate the sum of even and odd numbers up to num.
Keep
Undo
Args:
num: A positive integer
Returns:
A tuple of (even_sum, odd_sum)
"""
even = 0
odd = 0
i = 1
while i <= num:
for i in range(1, num + 1):
if i % 2 == 0:
even += i
else:
odd += i
i += 1
return even, odd
n = int(input("Enter a number: "))
e, o = even_odd_sum(n)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS C:\Users\arell\Music\aiac> & C:/Users/arell/AppData/Local/Programs/Python/Python311/python.exe c:/Users/arell/Music/aiac/dummy.py
Enter a number: 25
Even sum = 156
Odd sum = 169
PS C:\Users\arell\Music\aiac>
Python
Code
Python
Python

assg_02.py > ...

```python
def even_odd_sum_tuple(num):
    """Calculate sum of even and odd numbers from 1 to num.

    Args:
        num: Positive integer

    Returns:
        Tuple of (even_sum, odd_sum)
    """
    if num < 1:
        return 0, 0

    even_sum = sum(i for i in range(2, num + 1, 2))
    odd_sum = sum(i for i in range(1, num + 1, 2))
    return even_sum, odd_sum


def get_positive_integer(prompt):
    """Get valid positive integer input from user."""
    while True:
        try:
            value = int(input(prompt))
            if value < 1:
                print("Please enter a positive number.")
                continue
            return value
        except ValueError:
            print("Invalid input. Please enter a valid integer.")


if __name__ == "__main__":
    user_input = get_positive_integer("Enter a positive number: ")
    even, odd = even_odd_sum_tuple(user_input)
    print(f"Sum of even numbers: {even}")
    print(f"Sum of odd numbers: {odd}")
```

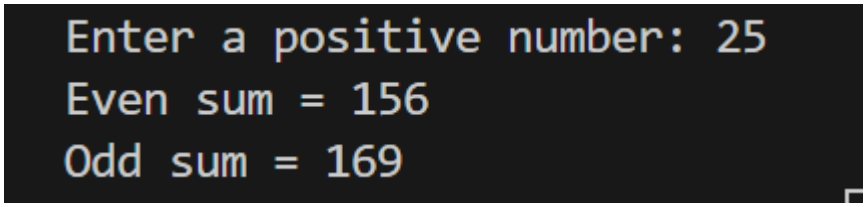

Code

```
def even_odd_sum(num):
    """Calculate the sum of even and odd numbers up to num.


    Args:
        num: A positive integer
```

```
        A tuple of (even_sum, odd_sum)
    """
    even = 0
    odd = 0

    for i in range(1, num + 1): if i % 2 == 0:
            even += i else:
            odd += i

    return even, odd


def get_valid_input(prompt):
    """Get and validate a positive integer from user input.""" while True:
        try:
            num = int(input(prompt)) if num <
            0:
                print("Please enter a non-negative number.") continue
            return num
        except ValueError:
                print("Invalid input. Please enter a valid integer.")


def main():
    """Main function to run the even/odd sum calculator.""" n =
    get_valid_input("Enter a positive number: ")

    if n == 0:
        print("Even sum = 0") print("Odd
        sum = 0") return
```

```
    even, odd = even_odd_sum(n)
    print(f"Even sum = {even}")
    print(f"Odd sum = {odd}")



if __name__ == "__main__":
    main()
```

output:



Explanation of improvements:

Input validation

> Making sure the user enters a valid integer.
> Handling negative numbers gracefully (return (0,0) or raise an error).

Readability

> Using clear variable names.
> Adding docstrings and comments.

Edge cases

> If input is 0, both sums should be 0.
> If input is negative, we can either reject it or compute sums up to that number (here I'll reject it for clarity).