Lab Assignment 2.1

Name:Anand

Hallticket:2303A51090

Batch:02

Task 1: Statistical Summary for Survey Data

Scenario:

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

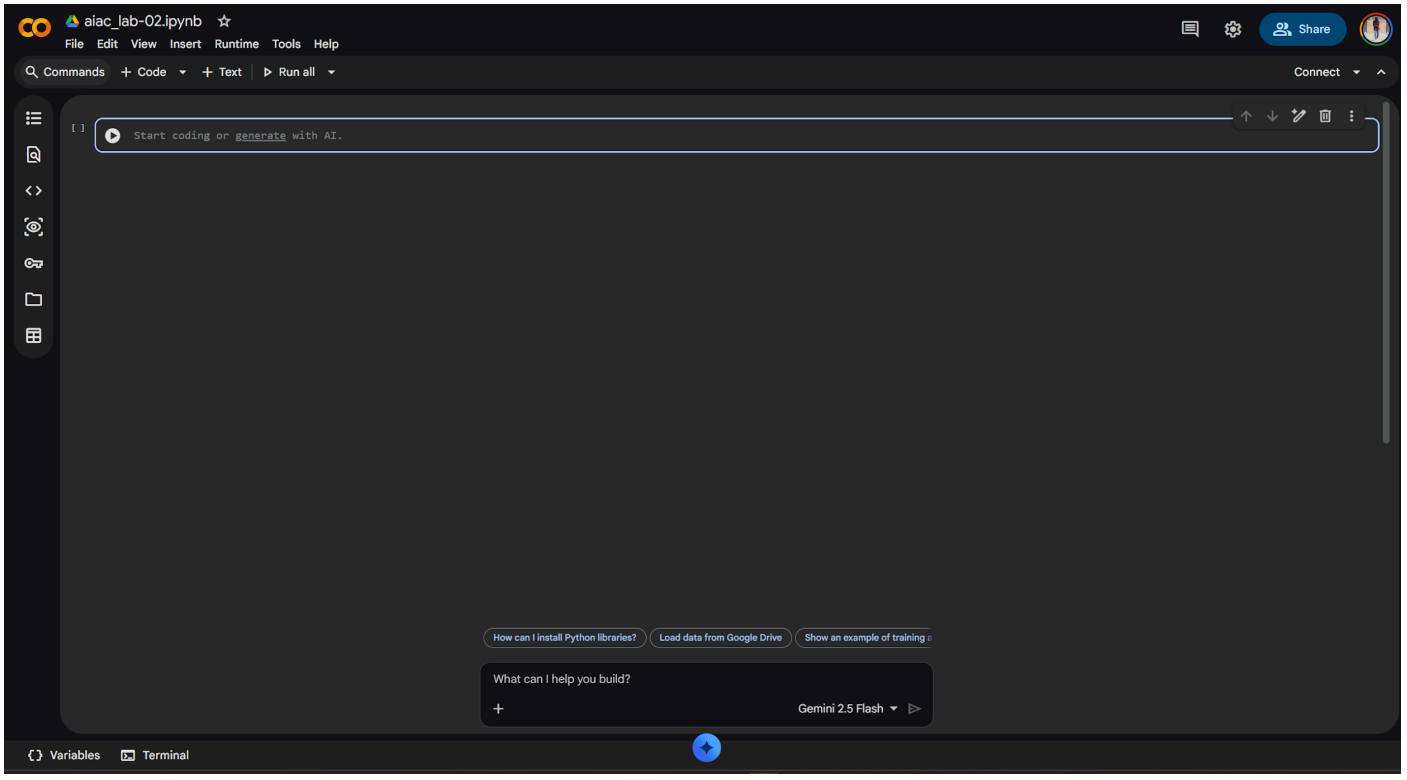
Task:

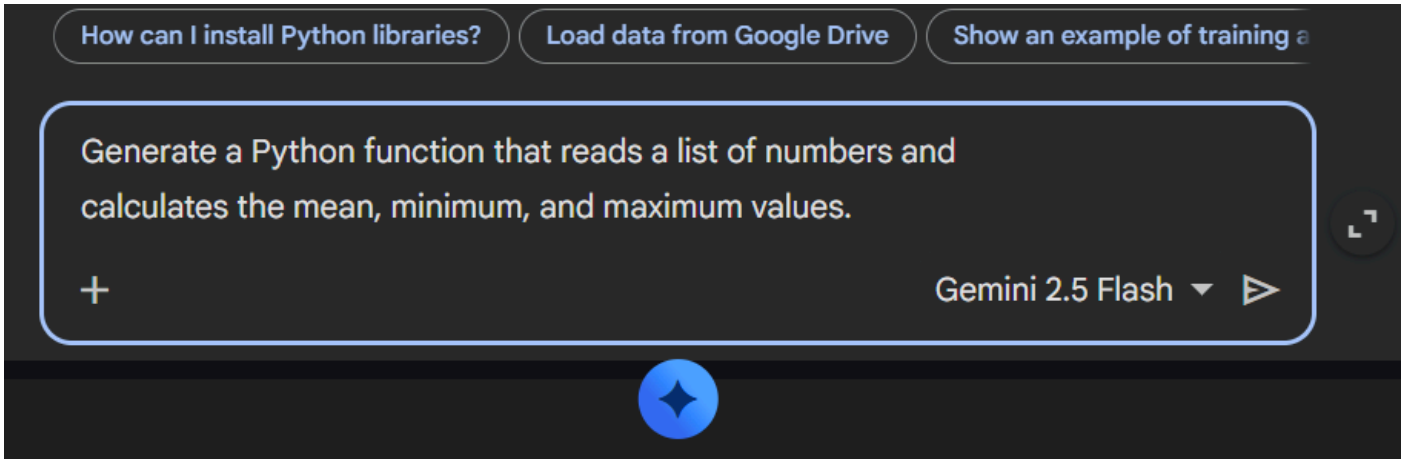
- Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

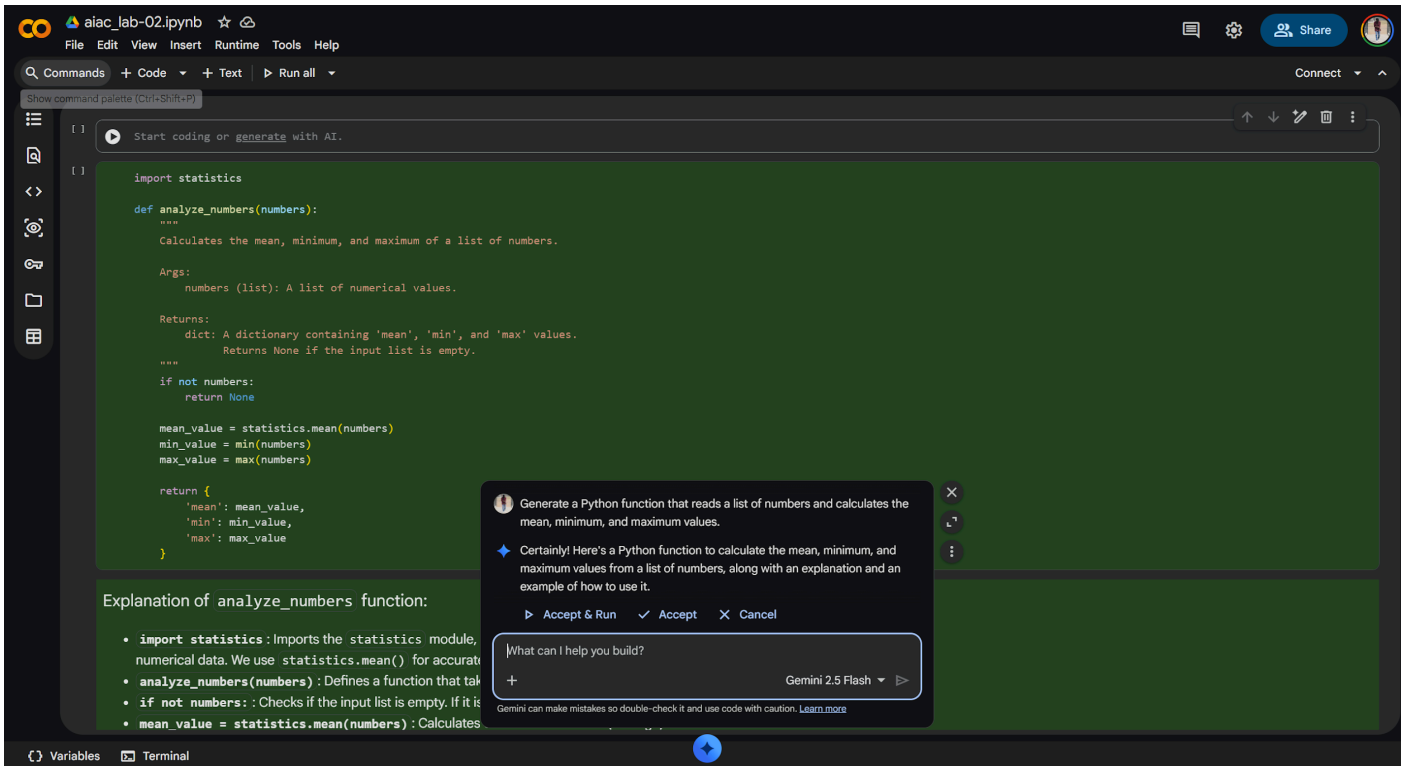
Expected Output:

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

Google Colab Workspace Screenshots:







Code:

```
import statistics

def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum of a list of numbers.

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

    Returns:
        dict: A dictionary containing 'mean', 'min', and 'max' values.
              Returns None if the input list is empty.
```

```
    """
    if not numbers:
        return None

    mean_value = statistics.mean(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return {
        'mean': mean_value,
        'min': min_value,
        'max': max_value
    }
data = [10, 20, 30, 40, 50, 5, 25]
analysis_results = analyze_numbers(data)

if analysis_results:
    print(f"Original List: {data}")
    print(f"Mean: {analysis_results['mean']}")
    print(f"Minimum: {analysis_results['min']}")
    print(f"Maximum: {analysis_results['max']}")
else:
    print("The list was empty.")

empty_data = []
empty_results = analyze_numbers(empty_data)

if empty_results:
    print(f"Original List: {empty_data}")
    print(f"Mean: {empty_results['mean']}")
    print(f"Minimum: {empty_results['min']}")
    print(f"Maximum: {empty_results['max']}")
else:
    print(f"Original List: {empty_data}")
    print("The list was empty.")
```

output:

```
Original List: [10, 20, 30, 40, 50, 5, 25]
Mean: 25.714285714285715
Minimum: 5
Maximum: 50
Original List: []
The list was empty.
```

Task 2: Armstrong Number – AI Comparison

Scenario:

- You are evaluating AI tools for numeric validation logic.

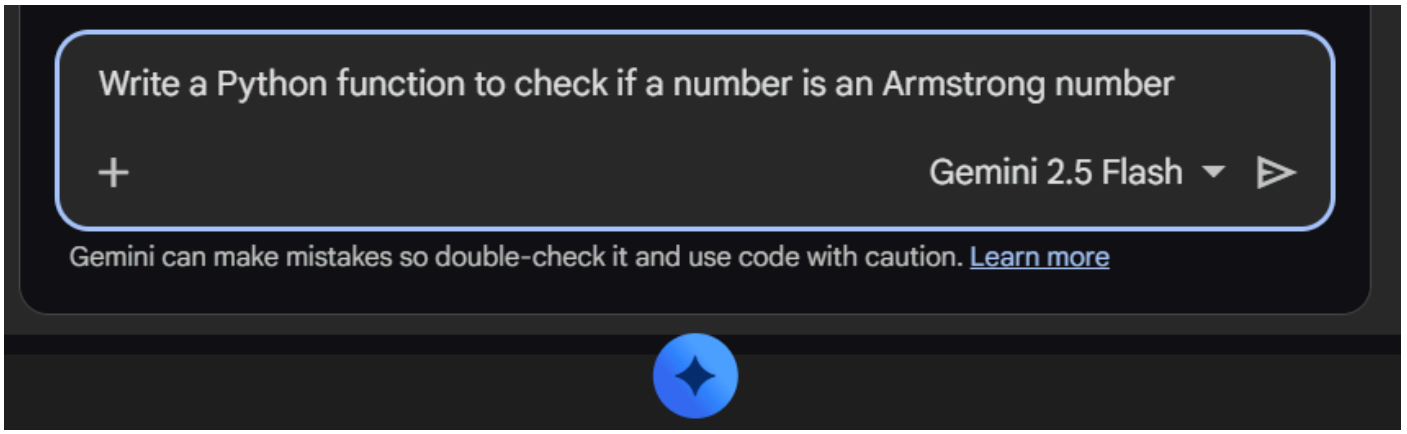
Task:

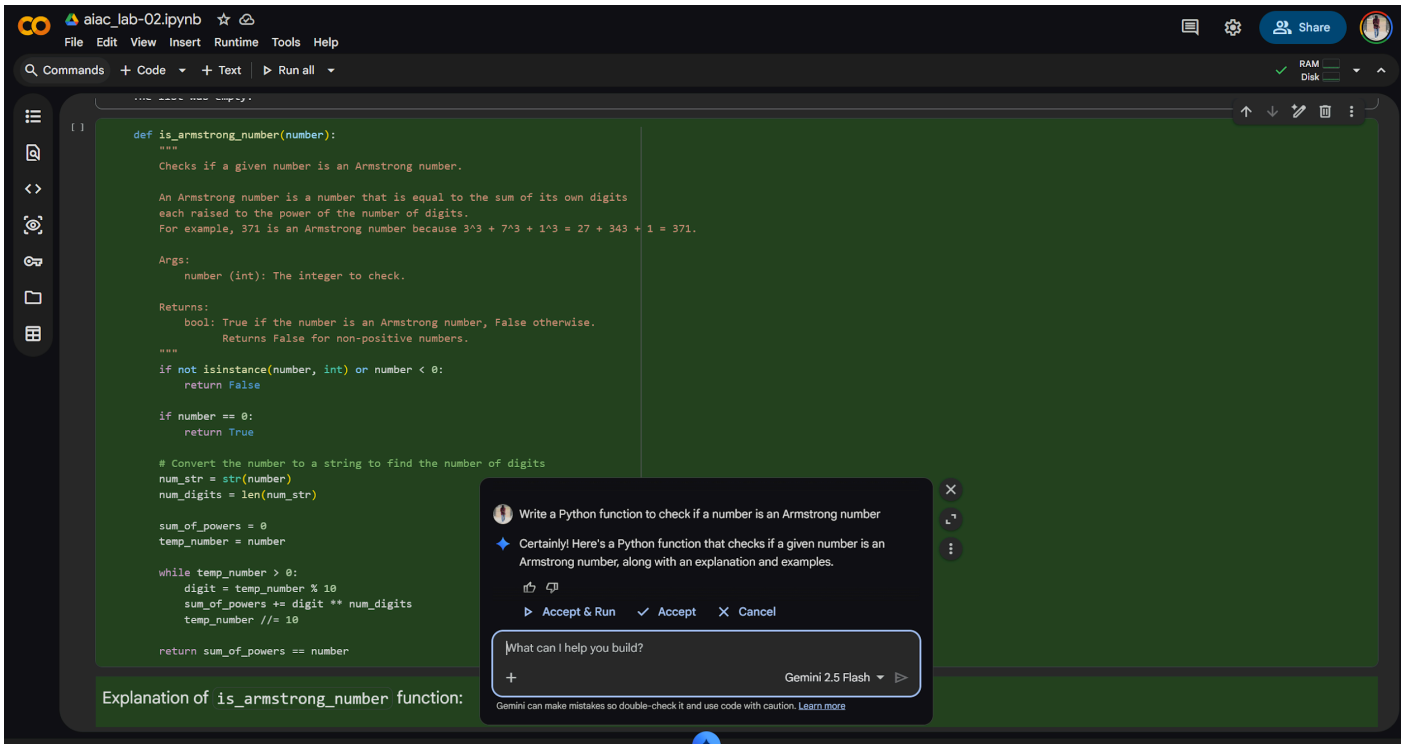
- Generate an Armstrong number checker using Gemini and GitHub Copilot.
- Compare their outputs, logic style, and clarity.

Expected Output:

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

Google Colab Workspace Screenshots:





Code:

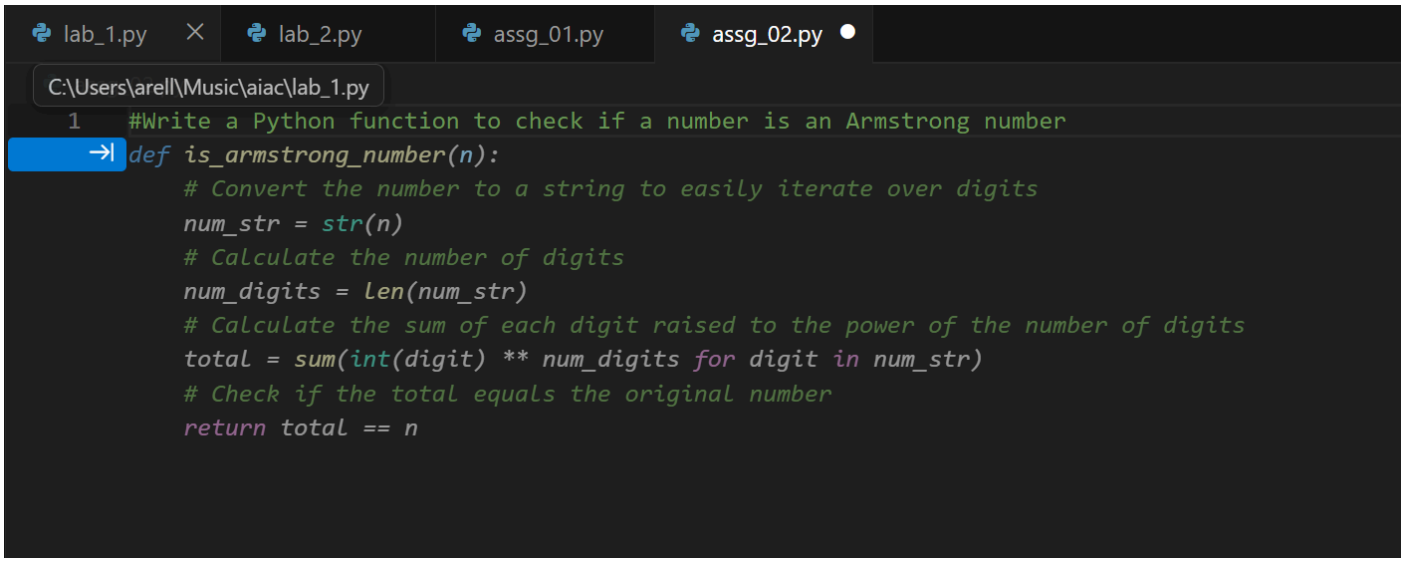
```
def is_armstrong_number(number):
```

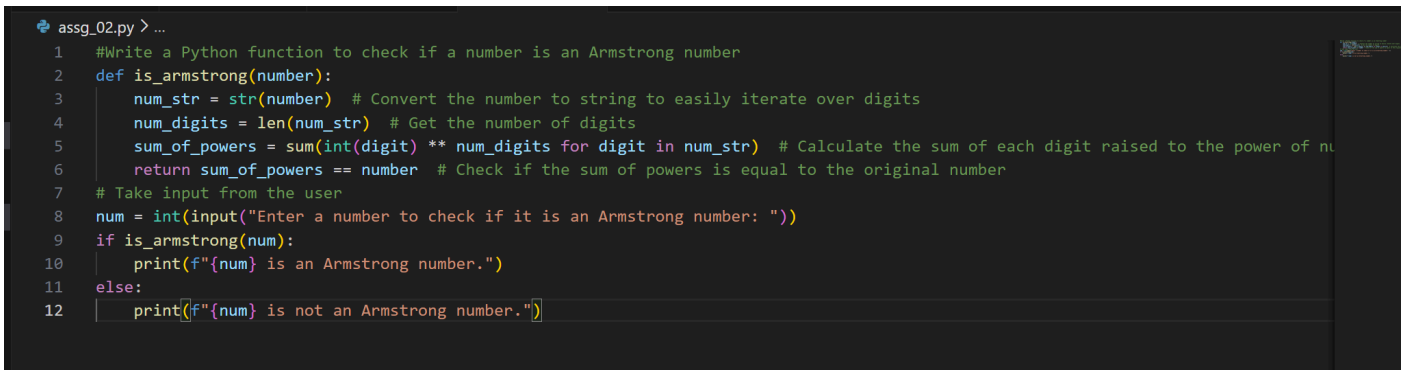
```
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number is a number that is equal to the sum of its own
digits
    each raised to the power of the number of digits.
    For example, 371 is an Armstrong number because 3^3 + 7^3 + 1^3 = 27 +
343 + 1 = 371.

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
              Returns False for non-positive numbers.
    """
    if not isinstance(number, int) or number < 0:
        return False

    if number == 0:
        return True

    # Convert the number to a string to find the number of digits
    num_str = str(number)
    num_digits = len(num_str)

    sum_of_powers = 0
    temp_number = number

    while temp_number > 0:
        digit = temp_number % 10
        sum_of_powers += digit ** num_digits
        temp_number //= 10

    return sum_of_powers == number
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 153 an Armstrong number? {is_armstrong_number(153)}")
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}")
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}")
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}")
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}")
print(f"Is 1000 an Armstrong number? {is_armstrong_number(1000)}")
print(f"Is -153 an Armstrong number? {is_armstrong_number(-153)}")
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")
```

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 407 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 1000 an Armstrong number? False
Is -153 an Armstrong number? False
Is 0 an Armstrong number? True
```

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:

```
PS C:\Users\arell\Music\aiac> & C:/Users/arell/AppData/Loca
Enter a number to check if it is an Armstrong number: 153
153 is an Armstrong number.
PS C:\Users\arell\Music\aiac>
```

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.

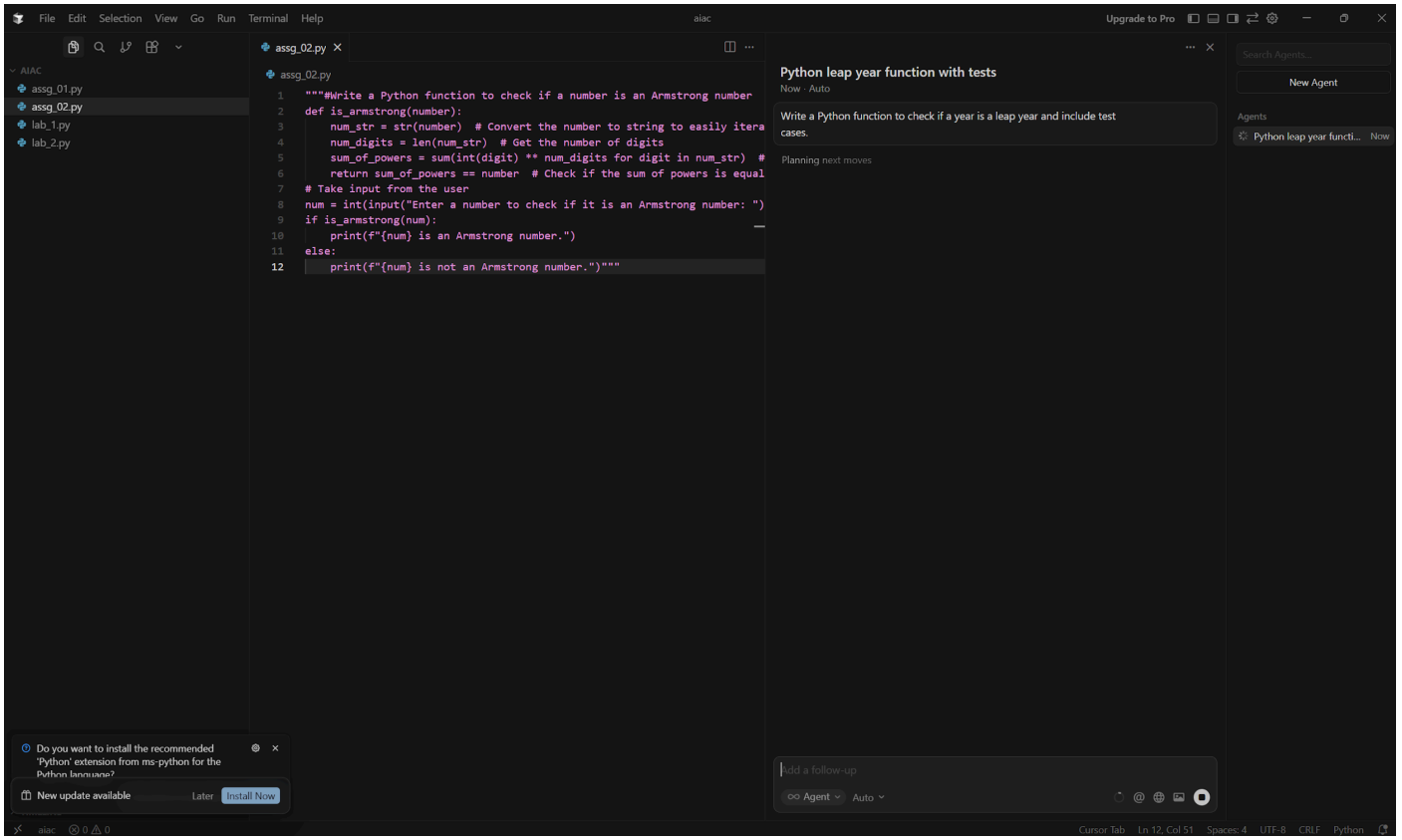
Expected Output:

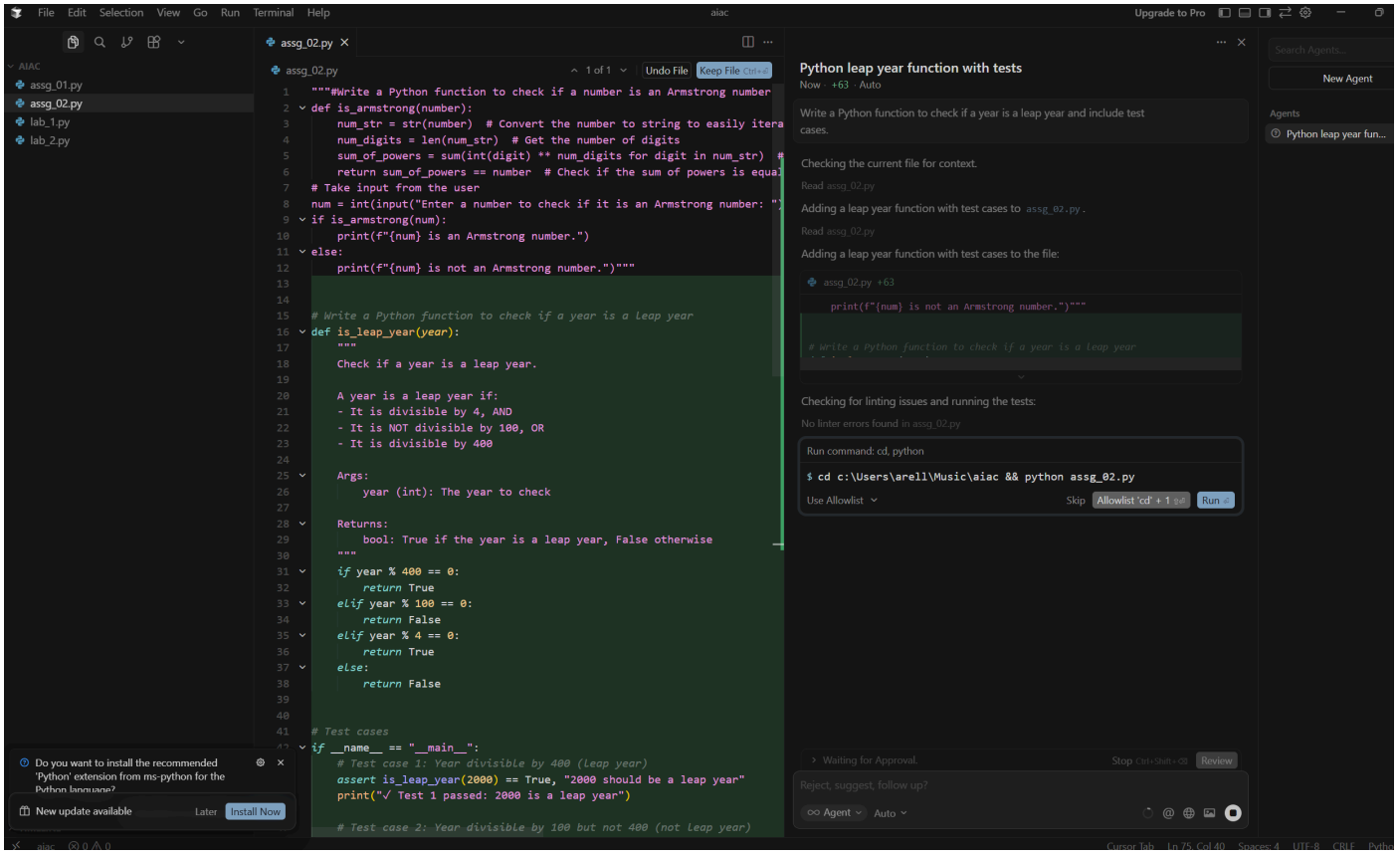
- Two versions of code
- Sample inputs/outputs
- Brief comparison

Version - 1:

Instruction Given:

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





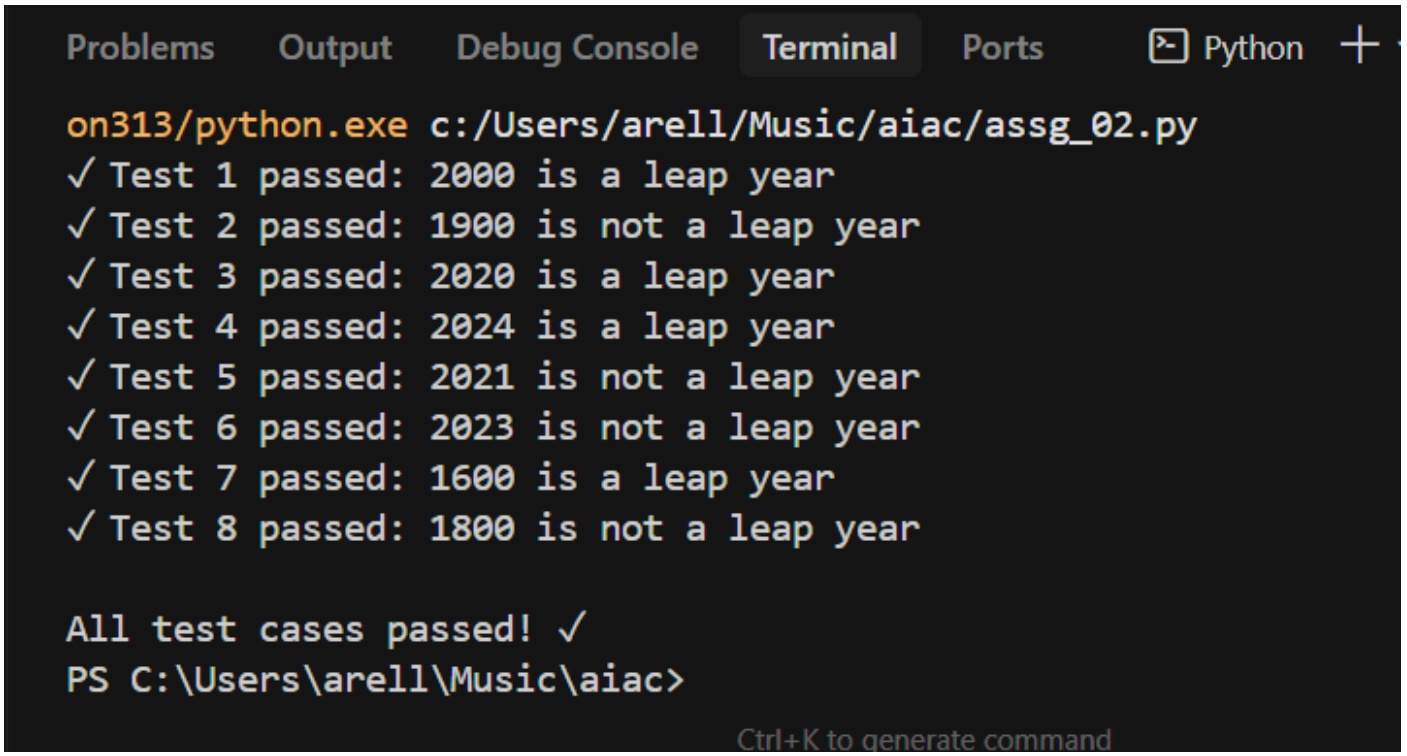
Code:

```
# 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.
```

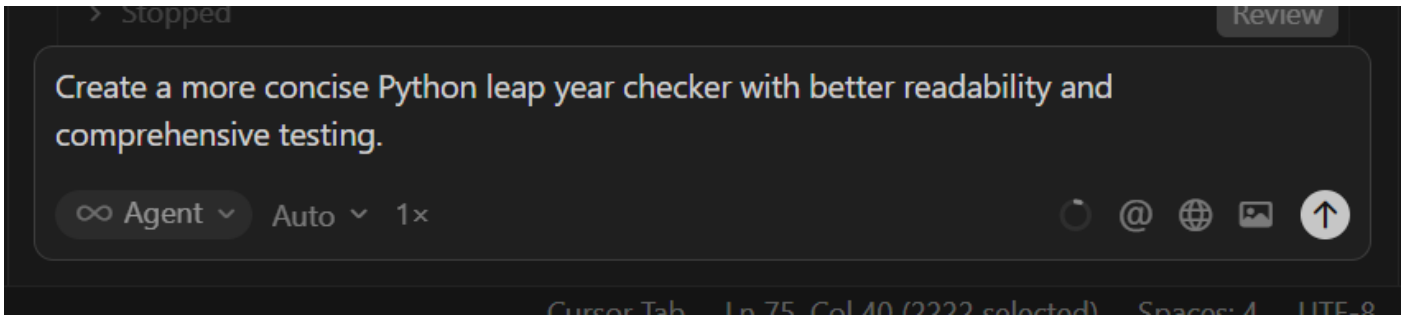
```
    A year is a leap year if:
    - It is divisible by 4, AND
    - It is NOT divisible by 100, OR
    - It is divisible by 400

    Args:
        year (int): The year to check

    Returns:
        bool: 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
if __name__ == "__main__":
    # Test case 1: Year divisible by 400 (leap year)
    assert is_leap_year(2000) == True, "2000 should be a leap year"
    print("✓ Test 1 passed: 2000 is a leap year")

    # Test case 2: Year divisible by 100 but not 400 (not leap year)
    assert is_leap_year(1900) == False, "1900 should not be a leap year"
    print("✓ Test 2 passed: 1900 is not a leap year")

    # Test case 3: Year divisible by 4 but not 100 (leap year)
    assert is_leap_year(2020) == True, "2020 should be a leap year"
    print("✓ Test 3 passed: 2020 is a leap year")

    # Test case 4: Year divisible by 4 but not 100 (leap year)
    assert is_leap_year(2024) == True, "2024 should be a leap year"
    print("✓ Test 4 passed: 2024 is a leap year")

    # Test case 5: Year not divisible by 4 (not leap year)
    assert is_leap_year(2021) == False, "2021 should not be a leap year"
    print("✓ Test 5 passed: 2021 is not a leap year")

    # Test case 6: Year not divisible by 4 (not leap year)
    assert is_leap_year(2023) == False, "2023 should not be a leap year"
    print("✓ Test 6 passed: 2023 is not a leap year")
```

```python
# Test case 7: Year divisible by 400 (leap year)
assert is_leap_year(1600) == True, "1600 should be a leap year"
print("✓ Test 7 passed: 1600 is a leap year")

# Test case 8: Year divisible by 100 but not 400 (not leap year)
assert is_leap_year(1800) == False, "1800 should not be a leap year"
print("✓ Test 8 passed: 1800 is not a leap year")

print("\nAll test cases passed! ✓")
```

output:



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
    test_cases = [
        # Centuries divisible by 400 (leap years)
        (2000, True, "Century divisible by 400"),
        (1600, True, "Century divisible by 400"),
        (2400, True, "Century divisible by 400"),

        # Centuries not divisible by 400 (not leap years)
        (1900, False, "Century not divisible by 400"),
        (1800, False, "Century not divisible by 400"),
        (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
            print(f"{status} {description}: {year} -> {result}")
        else:
            failed += 1
            print(f"{status} {description}: {year} -> {result} (expected
{expected})")

    # Summary
    print(f"\n{'='*50}")
    print(f"Tests passed: {passed}/{len(test_cases)}")
    if failed > 0:
        print(f"Tests failed: {failed}/{len(test_cases)}")
    else:
        print("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:

Aspect	Version 1 (Basic)	Version 2 (Concise)
Logic	Nested if-else	Single return with logical operators
Lines of code	More (≈12)	Fewer (≈5)
Readability	Step-by-step	Compact
Beginner friendly	Yes	No
Style	Traditional	Pythonic
Conditions	Separate checks	Combined logic
Testing	Basic tests	Comprehensive tests

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:

```
def even_odd_sum_tuple(num):
    even_sum = 0
    odd_sum = 0
    for i in range(1, num+1):
        if i%2 == 0:
            even_sum += i
        else:
            odd_sum += i
    return even_sum, odd_sum
user_input = int(input("Enter a number: "))
even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
```

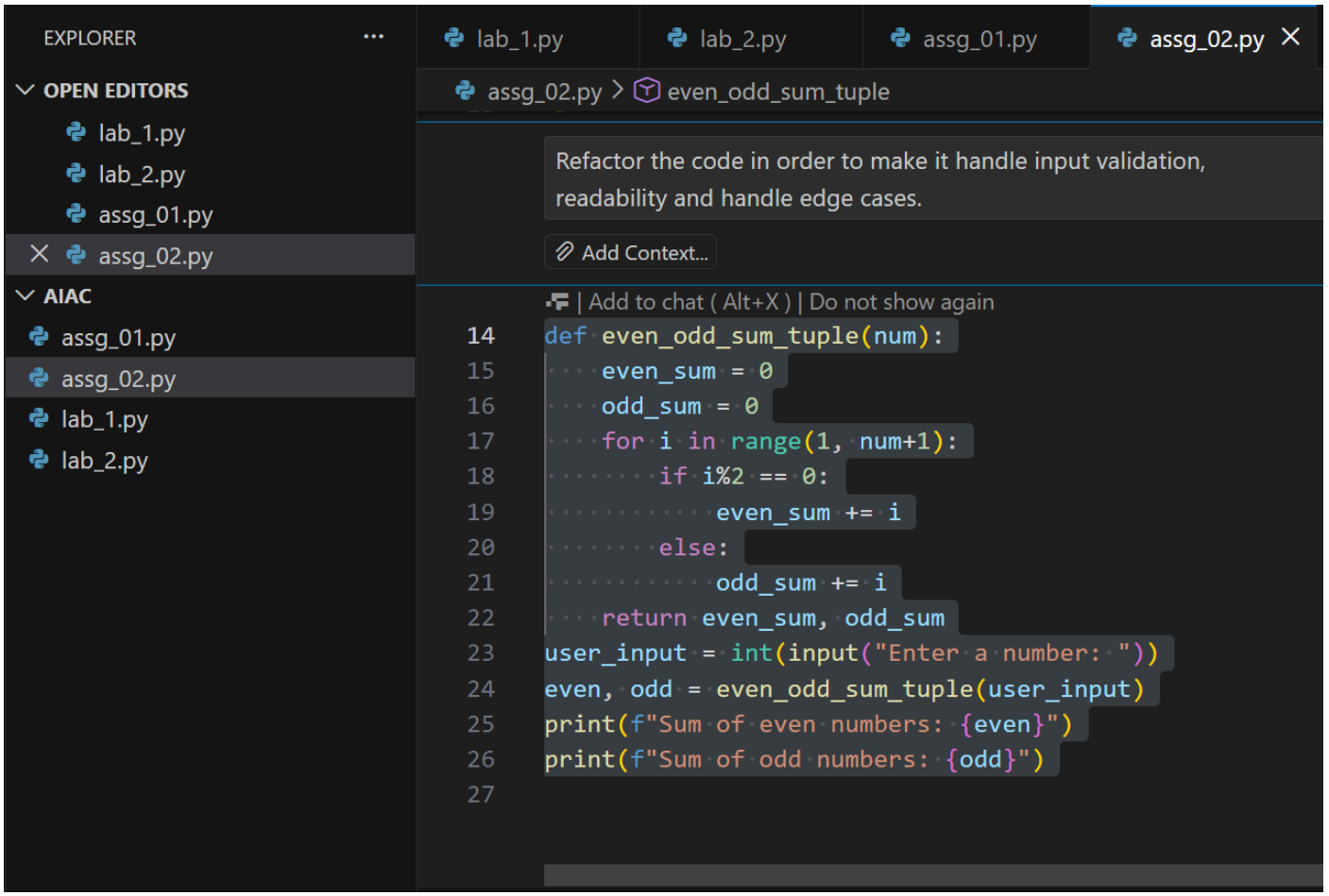
Code:

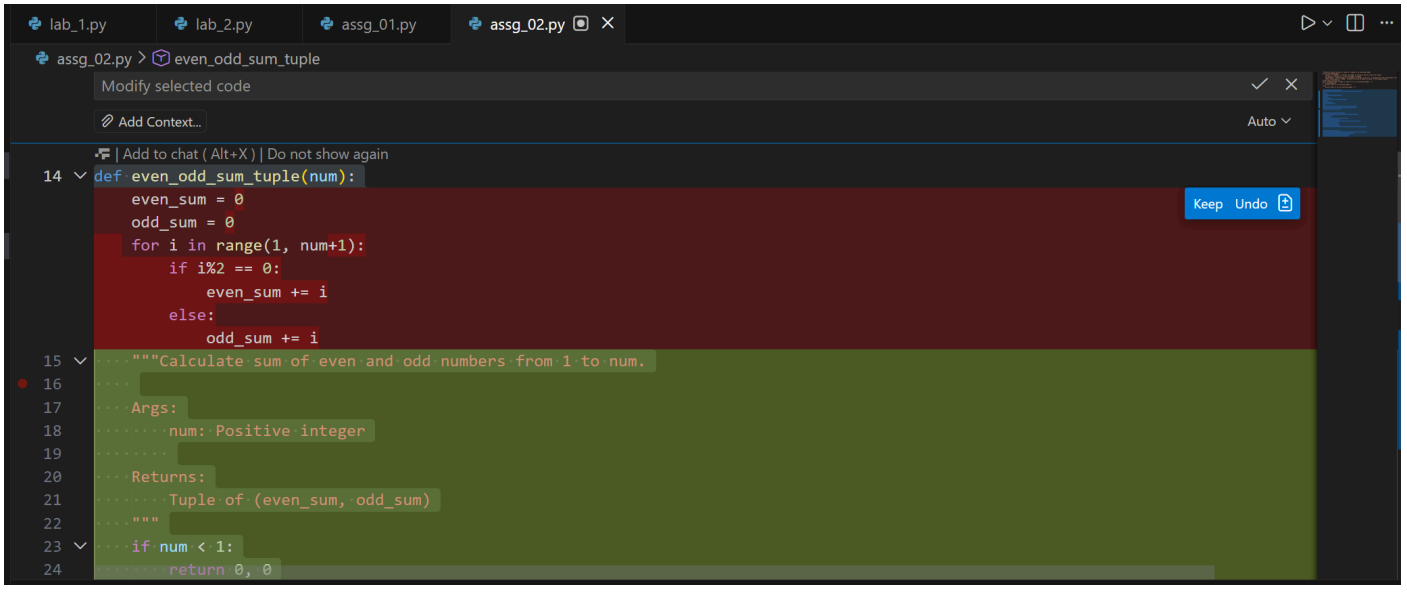
```
def even_odd_sum_tuple(num):
    even_sum = 0
    odd_sum = 0
    for i in range(1, num+1):
        if i%2 == 0:
            even_sum += i
        else:
            odd_sum += i
    return even_sum, odd_sum
user_input = int(input("Enter a number: "))
even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
```

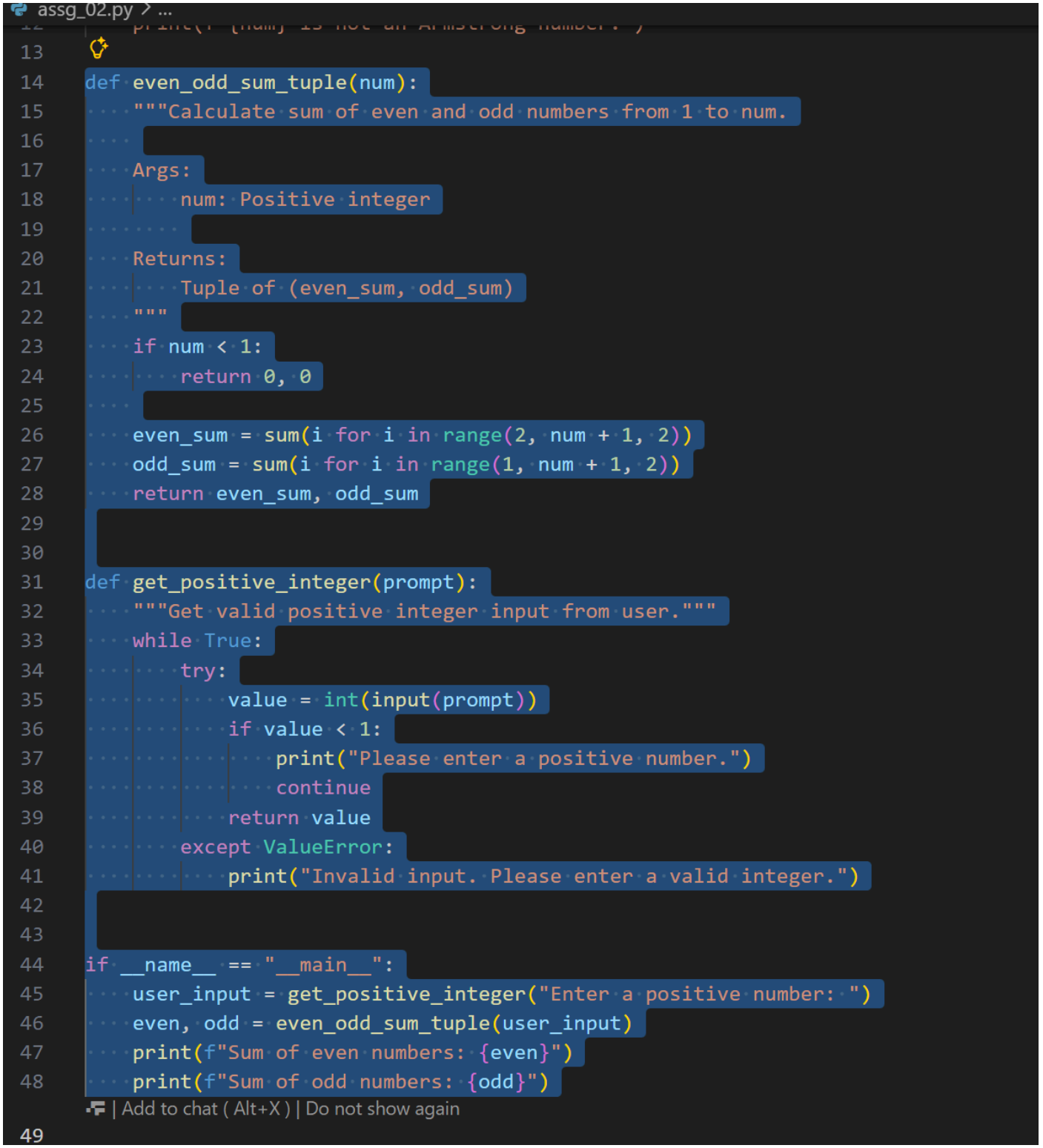
Output:

```
Enter a number: 50
Sum of even numbers: 650
Sum of odd numbers: 625
PS C:\Users\arell\Music\aiac>
```

Refactoring the code :







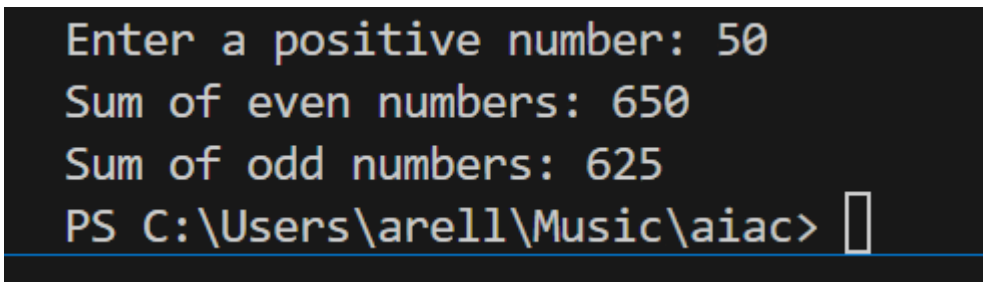
Code

d

```
ef 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}")
```

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).