

## Lab Assignment 2.1

Name: G.Hasini

Hallticket:2303A51099

Batch:02

### Task 1: Statistical Summary for Survey Data

Scenario:

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

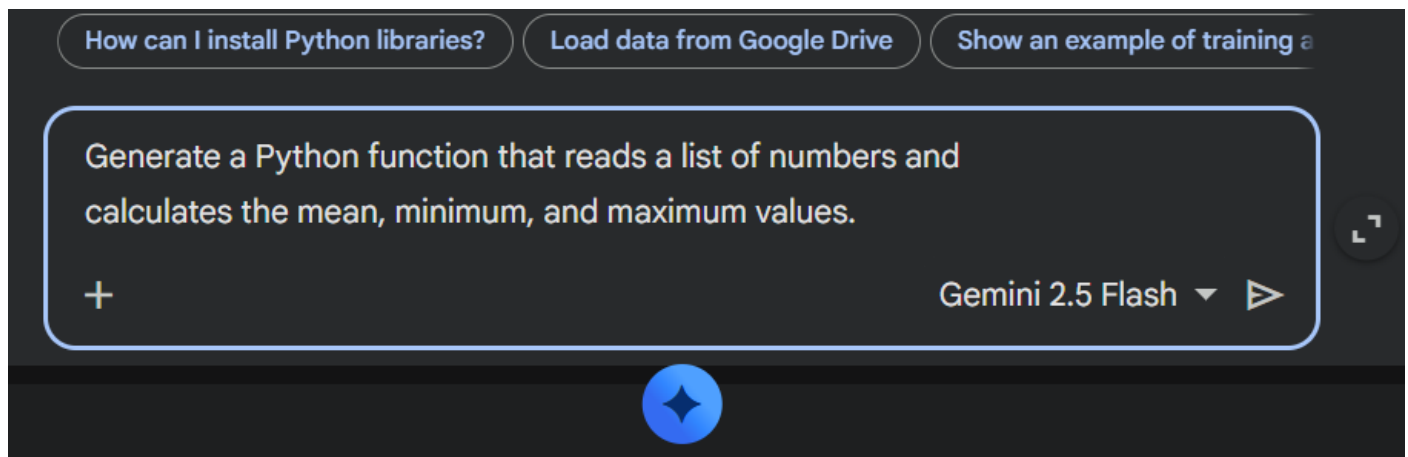
Task:

- 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:



The screenshot shows a Google Colab notebook with the following code in the first cell:

```
[4] def analyze_numerical_list(data_list):  
    """  
    Analyzes a list of numbers to calculate its mean, minimum, and maximum values.  
  
    Args:  
        data_list (list): A list of numerical values.  
  
    Returns:  
        dict: A dictionary containing the mean, minimum, and maximum values.  
        Returns None for min/max and NaN for mean if the list is empty.  
    """  
    if not data_list:  
        return {  
            'mean': float('nan'),  
            'minimum': None,  
            'maximum': None  
        }  
  
    total_sum = sum(data_list)  
    count = len(data_list)  
    mean_value = total_sum / count  
    min_value = min(data_list)  
    max_value = max(data_list)  
  
    return {  
        'mean': mean_value,  
        'minimum': min_value,  
        'maximum': max_value  
    }
```

A Gemini AI chat interface is open on the right side of the notebook. The prompt is: "generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values." The response is: "Certainly! Here's a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values, returning them in a dictionary. I've also included an example of how to use it." Below the response are two buttons: "Adapt 'analyze\_numerical\_list' to work with a Pandas DataFrame" and "Add error handling for non". At the bottom of the chat interface is a text input field with the placeholder "What can I help you build?" and a "Gemini 2.5 Flash" button.

The screenshot shows the same Google Colab notebook with the second cell executed. The output of the function is displayed in the left margin:

```
minimum = min_value,  
'maximum': max_value  
  
# Example usage:  
numbers = [10, 20, 30, 40, 50]  
analysis_results = analyze_numerical_list(numbers)  
print("Analysis for numbers:", numbers)  
print(analysis_results)  
  
empty_list = []  
empty_results = analyze_numerical_list(empty_list)  
print("\nAnalysis for empty list:", empty_list)  
print(empty_results)  
  
single_number_list = [7]  
single_number_results = analyze_numerical_list(single_number_list)  
print("\nAnalysis for single number list:", single_number_list)  
print(single_number_results)
```

The Gemini AI chat interface is still open on the right side of the notebook. The prompt is: "generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values." The response is: "Certainly! Here's a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values, returning them in a dictionary. I've also included an example of how to use it." Below the response are two buttons: "Adapt 'analyze\_numerical\_list' to work with a Pandas DataFrame" and "Add error handling for non". At the bottom of the chat interface is a text input field with the placeholder "What can I help you build?" and a "Gemini 2.5 Flash" button.

Code:

```
def analyze_numerical_list(data_list):
```

```
    """
```

Analyzes a list of numbers to calculate its mean, minimum, and maximum values.

Args:

`data_list (list):` A list of numerical values.

Returns:

dict: A dictionary containing the mean, minimum, and maximum values.

Returns None for min/max and NaN for mean if the list is empty.

```
"""
```

```
if not data_list:
```

```
    return {  
        'mean': float('nan'),  
        'minimum': None,  
        'maximum': None  
    }
```

```
total_sum = sum(data_list)  
count = len(data_list)  
mean_value = total_sum / count  
min_value = min(data_list)  
max_value = max(data_list)
```

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

### **output:**

# Example usage 1:

```
numbers = [10, 20, 30, 40, 50]  
analysis_results = analyze_numerical_list(numbers)  
print("Analysis for numbers:", numbers)  
print(analysis_results)
```

Analysis for numbers: [10, 20, 30, 40, 50]

{'mean': 30.0, 'minimum': 10, 'maximum': 50}

# Example usage 2:

```
empty_list = []
```

```
empty_results = analyze_numerical_list(empty_list)
```

```
print("\nAnalysis for empty list:", empty_list)
```

```
print(empty_results)
```

Analysis for empty list: []

```
{'mean': nan, 'minimum': None, 'maximum': None}
```

# Example usage 3:

```
single_number_list = [7]
```

```
single_number_results = analyze_numerical_list(single_number_list)
```

```
print("\nAnalysis for single number list:", single_number_list)
```

```
print(single_number_results)
```

Analysis for single number list: [7]

```
{'mean': 7.0, 'minimum': 7, 'maximum': 7}
```

## 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 (using gemini) Screenshots:

**G** generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

✦ Certainly! Here's a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values, returning them in a dictionary. I've also included an example of how to use it.

[Adapt 'analyze\\_numerical\\_list' to work with a Pandas DataFrame](#)

[Add error handling for non](#)

generate a simple python function for an Armstrong number checker

+

Gemini 2.5 Flash ▾ ▶

Gemini can make mistakes, so double-check it and use code with caution. [Learn more](#)



The screenshot shows a Google Colab notebook interface. The browser address bar at the top displays the URL: colab.research.google.com/drive/1930gqUD9zZvklCIOWfntmZFsCv1zEIHL#scrollTo=b2175987. The notebook's title bar indicates it is an 'aicoding-lab-2(1099).ipynb' file. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu, there are tabs for 'Commands', '+ Code', '+ Text', and 'Run all'. On the right side, there are icons for 'Share' and 'G' (Gemini), along with a status bar showing 'RAM' and 'Disk' usage. The main code editor area contains a Python function definition for 'is\_armstrong\_number'. The function includes a docstring with a description of an Armstrong number, its arguments, and its return value. The code logic involves converting the number to a string to find the number of digits, then iterating through each digit, raising it to the power of the number of digits, and summing these values to check if they equal the original number. An example usage is provided at the bottom of the code block.

```
def is_armstrong_number(number):  
    """  
    Checks if a given number is an Armstrong number.  
  
    An Armstrong number (also known as a narcissistic 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): An 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 find the number of digits and iterate through 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:
```

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

```
    """
```

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

```
    An Armstrong number (also known as a narcissistic 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): An 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 find the number of digits and iterate through 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 153 an Armstrong number? {is\_armstrong\_number(153)}") # output: True

print(f"Is 9474 an Armstrong number? {is\_armstrong\_number(9474)}") # output: True

print(f"Is 9 an Armstrong number? {is\_armstrong\_number(9)}") # output: True

print(f"Is 10 an Armstrong number? {is\_armstrong\_number(10)}") # output: False

print(f"Is 123 an Armstrong number? {is\_armstrong\_number(123)}") # output : False

Vs code :

The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left indicates 'NO FOLDER OPENED' and provides instructions on how to open a folder. The main editor window displays a file named 'armstrong.py' with the following Python code:

```
C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > armstrong.py
1 #generate a simple python function for an Armstrong number checker
2 def is_armstrong_number(number):
    # Convert number to string to access its digits
    num_str = str(number)
    n = len(num_str)

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

    return sum_of_powers == number

# Test cases
print(is_armstrong_number(153)) # True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(is_armstrong_number(9474)) # True (9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9
print(is_armstrong_number(123)) # False
print(is_armstrong_number(9)) # True (9^1 = 9)
print(is_armstrong_number(0)) # True (0^1 = 0)
```

The status bar at the bottom shows 'Ln 2, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Python'.

The screenshot shows the Visual Studio Code editor interface with a more detailed version of the 'armstrong.py' file. The Explorer sidebar remains the same. The main editor window displays the following Python code:

```
C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > armstrong.py > ...
1 #generate a simple python function for an Armstrong number checker
2
3 def is_armstrong_number(number):
4     """
5     Checks if a number is an Armstrong number.
6
7     An Armstrong number is a number that is equal to the sum of its own digits
8     each raised to the power of the number of digits.
9     For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
10    """
11    if not isinstance(number, int) or number < 0:
12        return False
13
14    num_str = str(number)
15    num_digits = len(num_str)
16
17    sum_of_powers = 0
18    for digit in num_str:
19        sum_of_powers += int(digit) ** num_digits
20
21    return sum_of_powers == number
22
```

The status bar at the bottom shows 'Ln 33, Col 39', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Python'.

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: "))
```

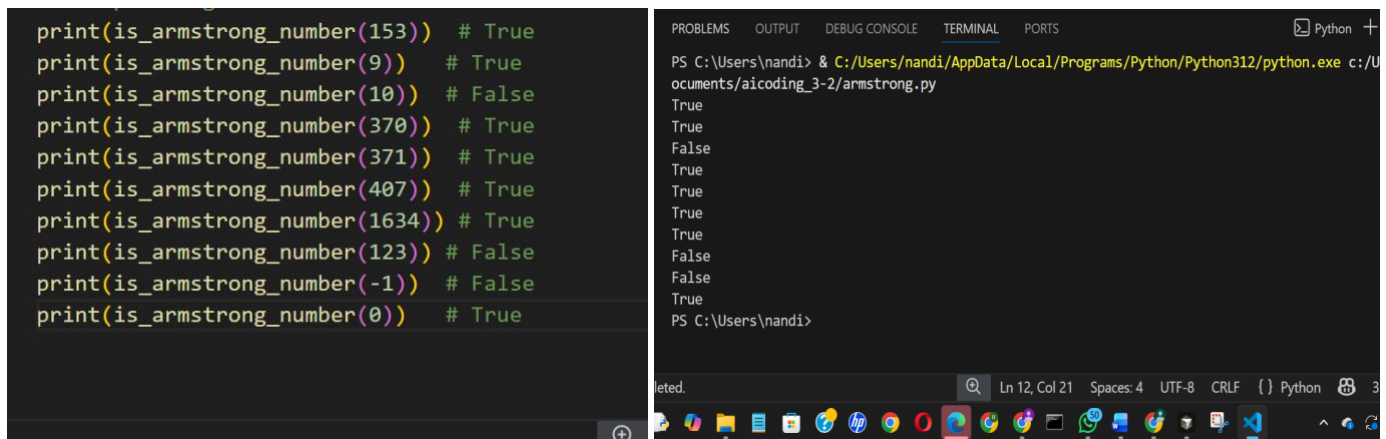
```
if is_armstrong(num):
```

```
    print(f"{num} is an Armstrong number.")
```

```
else:
```

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

output:



The image shows a code editor on the left and a terminal window on the right. The code editor contains the following Python code:

```
print(is_armstrong_number(153)) # True
print(is_armstrong_number(9)) # True
print(is_armstrong_number(10)) # False
print(is_armstrong_number(370)) # True
print(is_armstrong_number(371)) # True
print(is_armstrong_number(407)) # True
print(is_armstrong_number(1634)) # True
print(is_armstrong_number(123)) # False
print(is_armstrong_number(-1)) # False
print(is_armstrong_number(0)) # True
```

The terminal window shows the output of the program, which matches the comments in the code editor:

```
PS C:\Users\nandi> & C:/Users/nandi/AppData/Local/Programs/Python/Python312/python.exe c:/U
ocuments/aicoding_3-2/armstrong.py
True
True
False
True
True
True
True
False
False
True
PS C:\Users\nandi>
```

#### 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:



```
armstrong.py # sumofEvenAndOdd.py
C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > # sumofEvenAndOdd.py > sum_of_even_and_odd
1 # Python program that calculates the sum of odd and even numbers in a tuple
2 def sum_of_even_and_odd(numbers):
3     if not isinstance(numbers, tuple):
4         raise ValueError("Input must be a tuple.")
5
6     sum_even = 0
7     sum_odd = 0
8
9     for num in numbers:
10        if not isinstance(num, int):
11            raise ValueError("All elements in the tuple must be integers.")
12        if num % 2 == 0:
13            sum_even += num
14        else:
15            sum_odd += num
16
17    return {'even_sum': sum_even, 'odd_sum': sum_odd}
18
```

Code:

```
def sum_of_even_and_odd(numbers):
```

```
    if not isinstance(numbers, tuple):
```

```
        raise ValueError("Input must be a tuple.")
```

```
    sum_even = 0
```

```
    sum_odd = 0
```

```
    for num in numbers:
```

```
        if not isinstance(num, int):
```

```
            raise ValueError("All elements in the tuple must be integers.")
```

```
        if num % 2 == 0:
```

```
            sum_even += num
```

```
        else:
```

```
            sum_odd += num
```

```
    return {'even_sum': sum_even, 'odd_sum': sum_odd}
```

Output:

# Example usage 1

```
numbers_tuple = (1, 2, 3, 4, 5, 6)
```

```
result = sum_of_even_and_odd(numbers_tuple)
```

```
print(result)
```

```
numbers_tuple = (10, 15, 20, 25, 30)
```

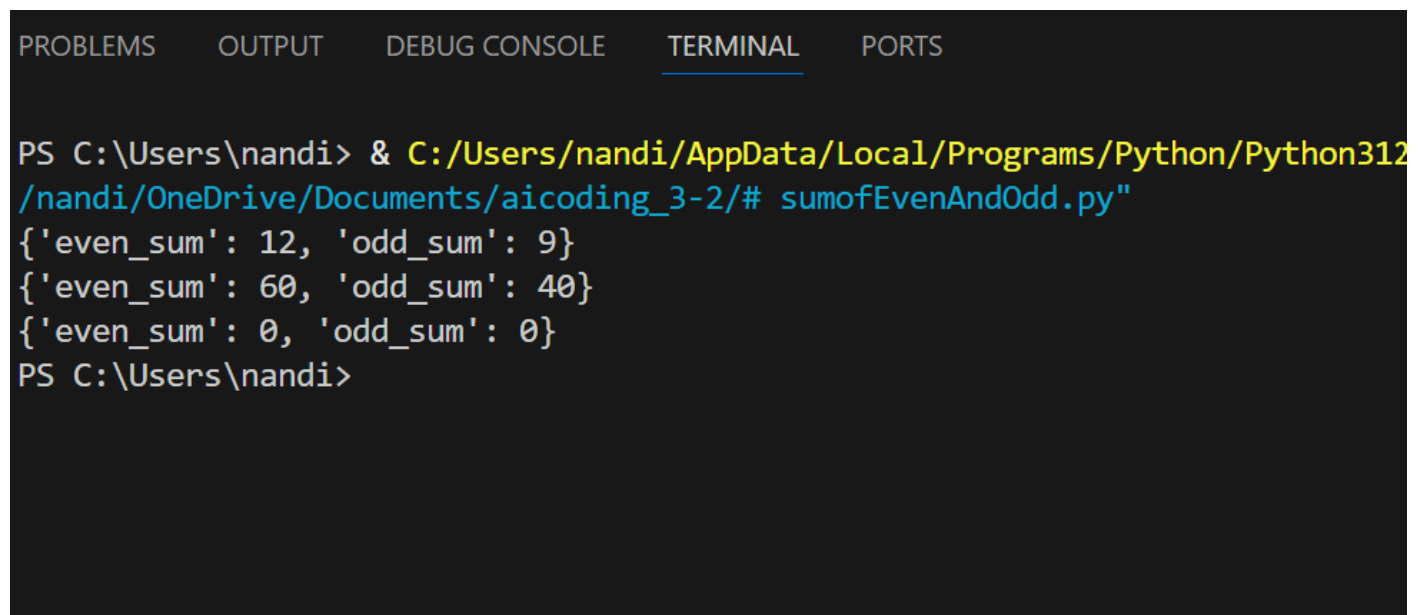
```
result = sum_of_even_and_odd(numbers_tuple)
```

```
print(result)
```

```
numbers_tuple = ()
```

```
result = sum_of_even_and_odd(numbers_tuple)
```

```
print(result)
```

A screenshot of a Windows command prompt window. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected and underlined), and 'PORTS'. The command prompt shows the following text: 

```
PS C:\Users\nandi> & C:/Users/nandi/AppData/Local/Programs/Python/Python312/nandi/OneDrive/Documents/aicoding_3-2/# sumofEvenAndOdd.py"
```

 The output of the script is displayed on the next three lines: 

```
{'even_sum': 12, 'odd_sum': 9}
{'even_sum': 60, 'odd_sum': 40}
{'even_sum': 0, 'odd_sum': 0}
```

 The prompt then returns to 

```
PS C:\Users\nandi>
```

Refactoring the code :

```
armstrong.py # sumofEvenAndOdd.py # sumofEvenAndOdd.py (Gemini Diff) X Accept Decline | ↑ ↓ ↺ □
C: > Users > nandi > AppData > Local > Temp > # sumofEvenAndOdd.py > ...
◆ Accept (Alt+A) | Decline (Alt+D)
1 1 # Python program that calculates the sum of odd and even numbers in a tuple
2 - def sum_of_even_and_odd(numbers):
3 -     if not isinstance(numbers, tuple):
4 -         raise ValueError("Input must be a tuple.")
5 -
6 -     sum_even = 0
7 -     sum_odd = 0
8 -
2+
3+ def sum_of_even_and_odd(numbers):
4+ """
5+     Calculates the sum of even and odd numbers in a given iterable.
6+
7+     Args:
8+         numbers (iterable): An iterable (e.g., tuple, list) containing numbers.
9+                             Non-integer elements will be ignored.
10+
11+     Returns:
12+         dict: A dictionary with 'even_sum' and 'odd_sum' keys,
```

## Code

```
def sum_of_even_and_odd(numbers):
```

```
    """
```

Calculates the sum of even and odd numbers in a given iterable.

Args:

numbers (iterable): An iterable (e.g., tuple, list) containing numbers.

Non-integer elements will be ignored.

Returns:

dict: A dictionary with 'even\_sum' and 'odd\_sum' keys,

representing the sum of even and odd numbers respectively.

Returns {'even\_sum': 0, 'odd\_sum': 0} for empty or invalid inputs.

```
    """
```

```
# Initialize sums
```

```
even_sum = 0
```

```
odd_sum = 0
```

```

# Handle non-iterable input gracefully
if not hasattr(numbers, '__iter__'):
    print("Warning: Input is not an iterable. Returning sums as 0.")
    return {'even_sum': 0, 'odd_sum': 0}

# Iterate through the numbers and categorize them
for num in numbers:
    # Ensure the element is an integer or can be converted to one
    if isinstance(num, (int, float)):
        # Convert floats to integers for parity check, or handle as needed
        # For this problem, we'll treat floats as their integer part for parity
        num_int = int(num)
        if num_int % 2 == 0:
            even_sum += num_int
        else:
            odd_sum += num_int
    # Optionally, you can log or handle non-numeric types differently
    # else:
    # print(f"Skipping non-numeric element: {num}")

return {'even_sum': even_sum, 'odd_sum': odd_sum}

```

```

# Example usage
numbers_tuple = (1, 2, 3, 4, 5, 6)
result = sum_of_even_and_odd(numbers_tuple)
print(result) # Output: {'even_sum': 12, 'odd_sum': 9}

numbers_tuple = (10, 15, 20, 25, 30)
result = sum_of_even_and_odd(numbers_tuple)
print(result) # Output: {'even_sum': 60, 'odd_sum': 40}

```

```
result = sum_of_even_and_odd(())
```

```
print(result) # Output: {'even_sum': 0, 'odd_sum': 0}
```

```
result = sum_of_even_and_odd([1, 2, 3, 'a', 4.5]) # Handles list and non-integers
```

```
print(result) # Output: {'even_sum': 6, 'odd_sum': 4} (4.5 becomes 4, 'a' is ignored)
```

```
result = sum_of_even_and_odd(None) # Handles non-iterable input
```

```
print(result) # Output: {'even_sum': 0, 'odd_sum': 0}
```

```
C: > Users > nandi > OneDrive > Documents > aicoding_3-2 > # sumofEvenAndOdd.py > ...
41
42 # Example usage
43 numbers_tuple = (1, 2, 3, 4, 5, 6)
44 result = sum_of_even_and_odd(numbers_tuple)
45 print(result) # Output: {'even_sum': 12, 'odd_sum': 9}
46
47 numbers_tuple = (10, 15, 20, 25, 30)
48 result = sum_of_even_and_odd(numbers_tuple)
49 print(result) # Output: {'even_sum': 60, 'odd_sum': 40}
50
51 result = sum_of_even_and_odd(())
52 print(result) # Output: {'even_sum': 0, 'odd_sum': 0}
53
54 result = sum_of_even_and_odd([1, 2, 3, 'a', 4.5]) # Handles list and non-integers
55 print(result) # Output: {'even_sum': 6, 'odd_sum': 4} (4.5 becomes 4, 'a' is ignored)
56
```

```
PS C:\Users\nandi> & C:/Users/nandi/AppData/Local/Programs/Python/Python312/python.exe "C:/Users/nandi/OneDrive/Documents/aicoding_3-2/# sumofEvenAndOdd.py"
{'even_sum': 12, 'odd_sum': 9}
{'even_sum': 60, 'odd_sum': 40}
{'even_sum': 0, 'odd_sum': 0}
{'even_sum': 6, 'odd_sum': 4}
Warning: Input is not an iterable. Returning sums as 0.
{'even_sum': 0, 'odd_sum': 0}
PS C:\Users\nandi>
```

Explanation of improvements:

Input validation

- Making sure the user enters a valid integer.
- Handling even in case we enter an invalid integer in a tuple and gives valid input only for integers

## Readability

- Using clear variable names.
- Adding comments and handling even edge and improper cases

## Edge cases

- If input is 0, both sums should be 0.
- If input contains any non-integers then that part of non integer is ignored and the rest of the tuple is processed in the effective way and produces valid output.