

AI ASSISTED CODING

ASSIGNMENT:2.1

M.KEERTHANA

2403A51259

BATCH-11

CSE-GEN

Task1:

Prompt:

- A write a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values

Response from gemini colab(code with output:

```
[1]
✓ Os

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

    Args:
        numbers: A list of numbers.

    Returns:
        A tuple containing the mean, minimum, and maximum values.
        Returns None for min/max if the list is empty, and 0 for mean.
    """
    if not numbers:
        return 0, None, None

    mean = sum(numbers) / len(numbers)
    minimum = min(numbers)
    maximum = max(numbers)

    return mean, minimum, maximum

# Example usage:
my_list = [10, 20, 30, 40, 50]
mean_value, min_value, max_value = calculate_stats(my_list)

print(f"List: {my_list}")
print(f"Mean: {mean_value}")
print(f"Minimum: {min_value}")
print(f"Maximum: {max_value}")

[1]
✓ Os

my_empty_list = []
mean_value_empty, min_value_empty, max_value_empty = calculate_stats(my_empty_list)

print(f"\nList: {my_empty_list}")
print(f"Mean: {mean_value_empty}")
print(f"Minimum: {min_value_empty}")
print(f"Maximum: {max_value_empty}")

→ List: [10, 20, 30, 40, 50]
Mean: 30.0
Minimum: 10
Maximum: 50

List: []
Mean: 0
Minimum: None
Maximum: None
```

Task2:

Prompt:

send Python function that checks whether a number is an Armstrong number

Response from copilot:

Code with output:

```
# 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)
    num_digits = len(num_str)
    total = sum(int(digit) ** num_digits for digit in num_str)
    return total == n

    # Example usage with output
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.")

[10] ✓ 4.1s
```

Python

Explanation:

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.")
```

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.

2. 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.

3. Sum of powered digits:

We compute the sum of each digit raised to the power of the total number of digits.

Example for 153:

$$(1^{3+5} + 3^3 = 1 + 125 + 27 = 153)$$

4. 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:

```

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

    An Armstrong number (also known as a narcissistic number,
    pluperfect digital invariant (PPDI), 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.

    For example:
    9 is an Armstrong number, because  $9 = 9^1 = 9$ 
    10 is not an Armstrong number, because  $1^2 + 0^2 = 1 \neq 10$ 
    153 is an Armstrong number, because  $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ 
    9474 is an Armstrong number, because  $9474 = 9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474$ 

    Args:
        number: An integer.

    Returns:
        True if the number is an Armstrong number, False otherwise.
    """
    # Convert the number to a string to easily get the number of digits
    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = 0

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

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

# Example usage:
print("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 9474 an Armstrong number? {is_armstrong_number(9474)}")
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}")

```

→ 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

Explanation:

The `is_armstrong_number(number)` function works as follows:

- 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`).
- 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.
- 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`.
- 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:

Comparison: Gemini vs Copilot (Armstrong Number Check in Python)

Aspect	Copilot	Gemini
Prompt/Task	User asked Copilot: " <i>Write a Python function to check if a number is an Armstrong number with examples.</i> "	User asked Gemini: " <i>Write a Python function to check if a number is an Armstrong number with examples.</i> "
Function Name	is_armstrong	is_armstrong_number
Docstring Style	Short and minimal: """Check whether a number is multiple worked examples. an Armstrong number."""	Long, detailed, with definition, alternate names, and examples. """Check whether a number is multiple worked examples. an Armstrong number."""
Logic	Uses generator expression with sum() (compact, Uses explicit for-loop with accumulation (clearer for efficient). beginners).	total = sum(int(digit) ** for digit in num_str: sum_of_powers += int(digit) ** num_digits for digit in num_digits num_str)
Example Usage	Interactive input: number = int(input("Enter a prints results for 9, 10, 153, 9474, 123 → good for number: ")) → checks one at a time.	Predefined multiple test cases: number = int(input("Enter a prints results for 9, 10, 153, 9474, 123 → good for number: ")) → demos/testing. number

	Friendly message:	Explicit test results:
Output Style		
	9474 is an Armstrong number. Is 153 an Armstrong number? True	
		Is 9 an Armstrong number? True Is 10 an Armstrong
Your Run	Enter a number: 9474	number? False Is 153 an Armstrong number?
True Is		
(Copilot)		an Armstrong number. 9474 an Armstrong number? True Is 123 an Armstrong number? False
	✓ Beginner-friendly	
	✓ Concise, efficient code	
Strengths		✓ Educational docstring
	✓ Interactive for user input	
	✓ Multiple auto test cases	
	Only checks one input at a	
		Verbose
Weaknesses	time	
		Slightly less efficient
	Minimal documentation	
	Quick script where user enters Teaching, debugging, or when you want multiple	
Best Use Case		
	any number.	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:

- A explain a Python function (e.g., `is_prime`) line by line, along with explanation step by step
- ◆ Okay, I can provide a Python function `is_prime` and then explain it line by line.

Code with output:

```

def is_prime(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.
    """
    if number <= 1:
        return False # Numbers less than or equal to 1 are not prime

    if number <= 3:
        return True # 2 and 3 are prime numbers

    if number % 2 == 0 or number % 3 == 0:
        return False # Eliminate multiples of 2 and 3

    i = 5
    while i * i <= number:
        if number % i == 0 or number % (i + 2) == 0:
            return False
        i += 6

    return True

# Example usage:
"""

print(f"Is 2 prime? {is_prime(2)}")
print(f"Is 10 prime? {is_prime(10)}")
print(f"Is 17 prime? {is_prime(17)}")
print(f"Is 21 prime? {is_prime(21)}")
print(f"Is 1 prime? {is_prime(1)}")
print(f"Is 0 prime? {is_prime(0)}")
"""

```

→ Is 2 prime? True
 Is 10 prime? False
 Is 17 prime? True
 Is 21 prime? False
 Is 1 prime? False
 Is 0 prime? False

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 $6k \pm 1$. 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 $6k \pm 1$ (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:

generate a Python function (e.g., sum of the first N natural numbers) and test its output.

Response from cursor ai:

```
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))

[2] ✓ 0.0s
... Sum of first 10 natural numbers: 55
```

Response from gemini:

```
# 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}")

# You can add more test cases here
# For example, test with n=1
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}")

# Test with n=0
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}")

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
```

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 $(n*(n+1)//2) \rightarrow$ very efficient ($O(1)$)	Loop (for i in range) \rightarrow 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 Both are correct .	Slower for very large N
	<ul style="list-style-type: none">• Cursor AI gave a mathematical shortcut.• Gemini gave a beginner-friendly approach with more test cases	

Task5:

Written code with output:

```
▶ 
numbers = (12, 3, 7, 8, 14, 21, 5, 10, 19)
sum_even = 0
sum_odd = 0
for num in numbers:
    if num % 2 == 0:
        sum_even += num
    else:
        sum_odd += num
print("Tuple:", numbers)
print("Sum of even numbers:", sum_even)
print("Sum of odd numbers:", sum_odd)

[11] ✓ 0.1s
...
Tuple: (12, 3, 7, 8, 14, 21, 5, 10, 19)
Sum of even numbers: 44
Sum of odd numbers: 55
Python
```

Prompt:

Refactor the code (numbers = (12, 3, 7, 8, 14, 21, 5, 10, 19))
sum_even = 0
sum_odd = 0
for num in numbers:
 if num % 2 == 0:
 sum_even += num
 else:
 sum_odd += num
print("Tuple:", numbers)
print("Sum of even numbers:", sum_even)
print("Sum of odd numbers:", sum_odd))to improve logic
and readability,

Response:

```
▶ 
# Refactored program to calculate the sum of odd and even numbers in a tuple

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)

# Display results
print(f"Tuple: {numbers}")
print(f"Sum of even numbers: {sum_even}")
print(f"Sum of odd numbers: {sum_odd}")

[12] ✓ 0.0s
...
Tuple: (12, 3, 7, 8, 14, 21, 5, 10, 19)
Sum of even numbers: 44
Sum of odd numbers: 55
Python
```

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.
-
2. **Finding Sum of Even Numbers**
 3. sum_even = sum(num for num in numbers if num % 2 == 0)

- o num for num in numbers if num % 2 == 0 → This is a **generator expression** that picks only even numbers from the tuple.
- o sum(...) → Adds them all together.
- o 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**.

3. Finding Sum of Odd Numbers

4. sum_odd = sum(num for num in numbers if num % 2 != 0)
- o Same idea, but checks if the number is **not divisible by 2** ($\neq 0$).
- o This picks out only the odd numbers.

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

4. Printing Results with f-strings

5. print(f"Tuple: {numbers}")
6. print(f"Sum of even numbers: {sum_even}")
7. print(f"Sum of odd numbers: {sum_odd}")
- o f"..." allows us to directly embed variables in the string (cleaner than concatenation).
- o 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.