# Lab-2.1

Name :Kuraku.Rajavardhan Reddy

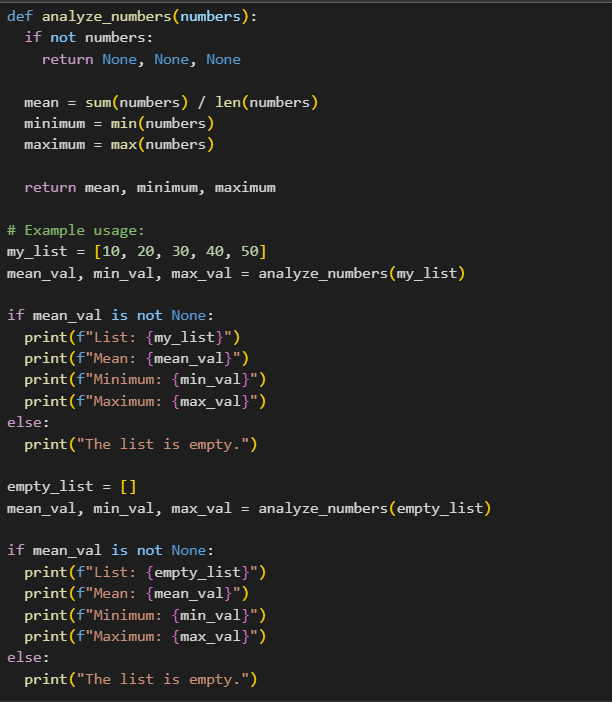
Roll no:2403A51108

Batch:06

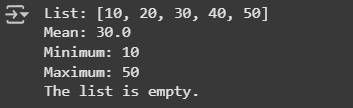
Task-1

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

Code:



Output:



Task-2:

2. Compare Gemini and Copilot outputs for a Python function that checks whether a number is an Armstrong number. Document the steps, prompts, and outputs

Code Snippet 1 (Original):

def analyze\_numbers(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, None, None) if the list is empty.

"""

if not numbers:

return None, 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\_val, min\_val, max\_val = analyze\_numbers(my\_list)

if mean\_val is not None:

print(f"List: {my\_list}")

print(f"Mean: {mean\_val}")

print(f"Minimum: {min\_val}")

print(f"Maximum: {max\_val}")

else:

print("The list is empty.")

empty\_list = []

mean\_val, min\_val, max\_val = analyze\_numbers(empty\_list)

if mean\_val is not None:

print(f"List: {empty\_list}")

print(f"Mean: {mean\_val}")

print(f"Minimum: {min\_val}")

print(f"Maximum: {max\_val}")

else:

print("The list is empty.")

Key Differences in Code Snippet 1:

Empty List Handling: If the input list numbers is empty, the function returns a tuple of three None values (None, None, None).

Return Type: For a non-empty list, the function returns a tuple containing the mean, minimum, and maximum values in that order (mean, minimum, maximum).

Example Usage: The example usage checks if the returned mean value is None to determine if the list was empty and prints the results accordingly. It unpacks the returned tuple into separate variables.

Code Snippet 2 (Provided by User):

def analyze\_numbers(numbers):

if not numbers:

return "The list is empty. Please provide valid numbers."

mean\_value = sum(numbers) / len(numbers)

min\_value = min(numbers)

max\_value = max(numbers)

return {

"mean": mean\_value,

"min": min\_value,

"max": max\_value

}

# Example usage:

num\_list = [10, 25, 7, 33, 19]

result = analyze\_numbers(num\_list)

print("Mean:", result["mean"])

print("Minimum:", result["min"])

print("Maximum:", result["max"])

Key Differences in Code Snippet 2:

Empty List Handling: If the input list numbers is empty, the function returns a string message: "The list is empty. Please provide valid numbers."

Return Type: For a non-empty list, the function returns a dictionary containing the mean, minimum, and maximum values with descriptive keys ("mean", "min", "max").

Example Usage: The example usage directly accesses the values from the returned dictionary using their keys (result["mean"], result["min"], result["max"]). It does not explicitly handle the case of an empty list in the printing part of the example, as the function itself returns a string in that case.

In summary:

Code Snippet 1 uses None values and a tuple to indicate an empty list and return values, respectively, requiring a check for None in the calling code.

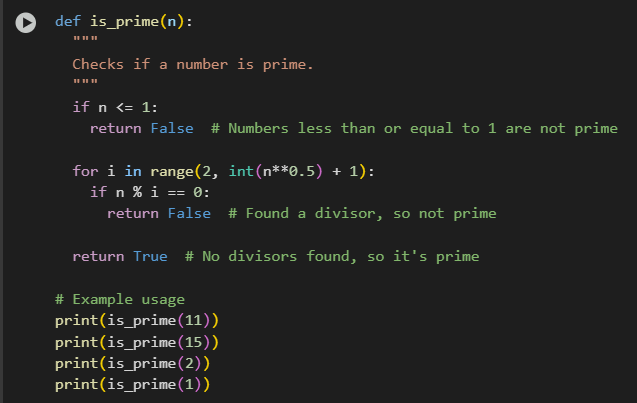
Code Snippet 2 uses a string message for an empty list and a dictionary for return values, providing more descriptive access to the results.

Both functions achieve the goal of calculating the mean, minimum, and maximum, but they differ in how they signal an empty list and the structure of their output for non-empty lists.

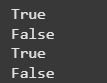
Task-3:

Ask Gemini to explain a Python function (e.g., is\_prime(n) or  
is\_palindrome(s)) line by line

Code:



Output:



Explanation:

Code Snippet:

def is\_prime(n):

"""

Checks if a number is prime.

"""

if n <= 1:

return False # Numbers less than or equal to 1 are not prime

for i in range(2, int(n\*\*0.5) + 1):

if n % i == 0:

return False # Found a divisor, so not prime

return True # No divisors found, so it's prime

# Example usage

print(is\_prime(11))

print(is\_prime(15))

print(is\_prime(2))

print(is\_prime(1))

Detailed Explanation with Gemini's Responses:

Let's break down the is\_prime(n) function line by line, incorporating the explanations provided:

1. Function Definition:

def is\_prime(n):

Gemini's explanation: This line is like giving a name to a set of instructions. We're creating a function called is\_prime that takes one piece of information, which we'll call n. Think of n as the number we want to check.

2. Docstring:

"""

Checks if a number is prime.

"""

Gemini's explanation: These lines in the triple quotes are a description of what the function does. It's like a little note for anyone reading the code to understand its purpose.

3. Handling Numbers Less Than or Equal to 1:

if n <= 1:

return False # Numbers less than or equal to 1 are not prime

Gemini's explanation: This line checks if the number n is less than or equal to 1. Prime numbers are defined as being greater than 1, so if n is 1 or less, we immediately know it's not prime, and the function stops and says "False".

4. Looping to Check for Divisors:

for i in range(2, int(n\*\*0.5) + 1):

Gemini's explanation: This line starts a loop. We're going to check for possible divisors of n starting from the number 2 up to the square root of n. We only need to check up to the square root because if a number has a divisor larger than its square root, it must also have a divisor smaller than its square root.

5. Checking for Divisibility:

if n % i == 0:

return False # Found a divisor, so not prime

Gemini's explanation: Inside the loop, this line checks if n can be perfectly divided by the current number i (the modulo operator % gives us the remainder). If the remainder is 0, it means i is a divisor of n. If we find any divisor other than 1 and itself, we know n is not prime, so the function stops and says "False".

6. Returning True if No Divisors Found:

return True # No divisors found, so it's prime

Gemini's explanation: If the loop finishes without finding any divisors (meaning the if n % i == 0: condition was never true), it means the number n is only divisible by 1 and itself. In this case, the function says "True", indicating that the number is prime.

7. Example Usage:

# Example usage

print(is\_prime(11))

print(is\_prime(15))

print(is\_prime(2))

print(is\_prime(1))

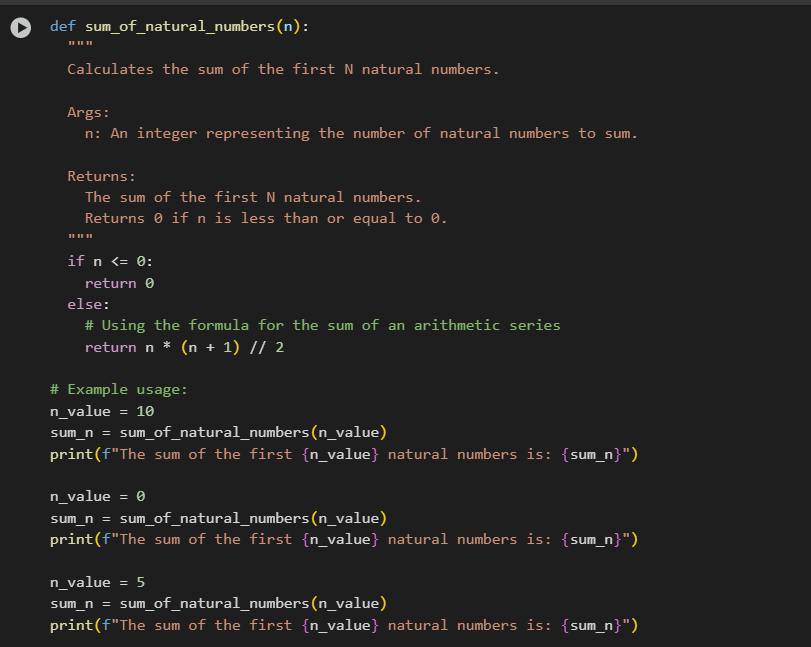
Gemini's explanation: These lines show how to use the is\_prime function with different numbers and print the results to see if they are prime or not.

This detailed breakdown combines the code with the easy-to-understand explanations provided earlier. Let me know if you would like me to explain anything else!

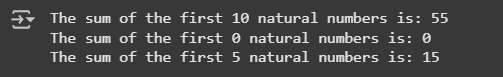
Task-4:

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

Code:



Output:



Explanation:

def sum\_of\_natural\_numbers(n):

"""

Calculates the sum of the first N natural numbers.

Args:

n: An integer representing the number of natural numbers to sum.

Returns:

The sum of the first N natural numbers.

Returns 0 if n is less than or equal to 0.

"""

This block defines the function sum\_of\_natural\_numbers that takes one argument n. The triple-quoted string is a docstring explaining the function's purpose, arguments, and return value.

if n <= 0:

return 0

This is a conditional statement that checks if the input n is less than or equal to 0. Natural numbers are typically considered positive integers (1, 2, 3, ...). If n is not a positive integer, the function returns 0, as there are no natural numbers to sum in this case.

else:

# Using the formula for the sum of an arithmetic series

return n \* (n + 1) // 2

If n is a positive integer, this block is executed. It calculates the sum of the first n natural numbers using the formula for the sum of an arithmetic series: n \* (n + 1) / 2. The // 2 ensures integer division, which is appropriate here since the sum of integers will always be an integer.

# Example usage:

n\_value = 10

sum\_n = sum\_of\_natural\_numbers(n\_value)

print(f"The sum of the first {n\_value} natural numbers is: {sum\_n}")

This block demonstrates how to use the function. It sets n\_value to 10, calls sum\_of\_natural\_numbers with this value, and stores the result in sum\_n. Finally, it prints a formatted string showing the input n\_value and the calculated sum\_n.

n\_value = 0

sum\_n = sum\_of\_natural\_numbers(n\_value)

print(f"The sum of the first {n\_value} natural numbers is: {sum\_n}")

This block shows another example with n\_value set to 0 to demonstrate the handling of non-positive input.

n\_value = 5

sum\_n = sum\_of\_natural\_numbers(n\_value)

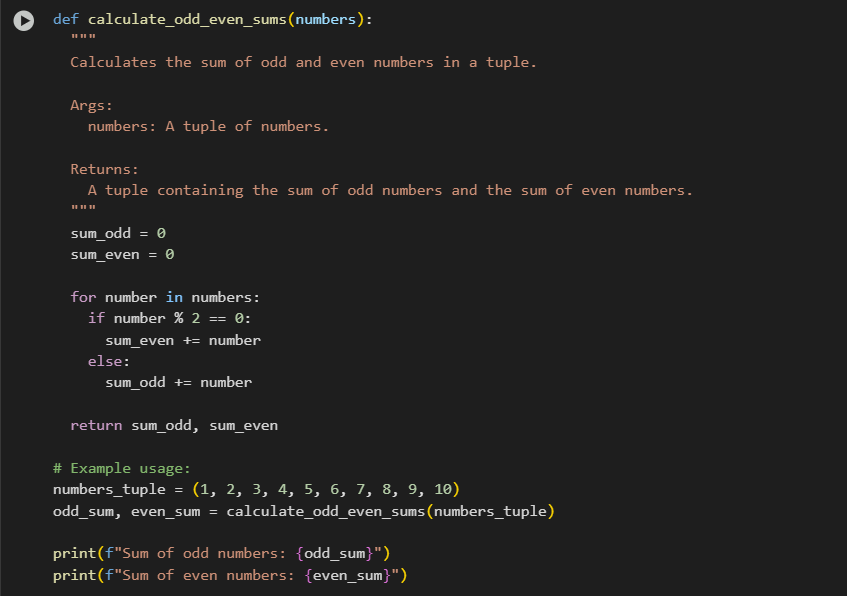
print(f"The sum of the first {n\_value} natural numbers is: {sum\_n}")

This block provides a third example with n\_value set to 5.

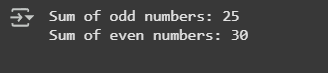
Task-5:

5.Generate a Python program to calculate the sum of odd and even numbers in a given tuple, refactor the code to improve logic and readability, and provide explanations and output.

Code:



Output:



Explanation:

Detailed Explanation of the Refactored Code

Function Definition and Docstring:

def calculate\_odd\_even\_sums(numbers):

Calculates the sum of odd and even numbers in a tuple.

Args:

numbers: A tuple of numbers.

Returns:

A tuple containing the sum of odd numbers and the sum of even numbers.

This section defines a function named `calculate\_odd\_even\_sums` that accepts one argument, `numbers`, which is expected to be a tuple.

The triple-quoted string below the function definition is a docstring. It explains the purpose of the function,

describes its arguments (`Args`), and specifies what it returns (`Returns`). This is crucial for code documentation and understanding.

Initialization of Variables:

sum\_odd = 0

sum\_even = 0

These lines initialize two variables, `sum\_odd` and `sum\_even`, to 0.

These variables will store the cumulative sum of odd and even numbers found in the input tuple, respectively.

Starting them at 0 ensures that the sums are calculated correctly from the beginning.

Loop for Iterating Through the Tuple:

for number in numbers:

This line starts a `for` loop that iterates through each element in the `numbers` tuple.

In each iteration, the current element from the tuple is assigned to the variable `number`.

Conditional Check for Odd/Even Numbers:

if number % 2 == 0:

sum\_even += number

else:

sum\_odd += number

This is a conditional statement that checks if the current `number` is even or odd.

The modulo operator (`%`) calculates the remainder when `number` is divided by 2.

If the remainder is 0 (`number % 2 == 0`), it means the number is even, and its value is added to the `sum\_even` variable using the `+=` operator (which is shorthand for `sum\_even = sum\_even + number`).

If the remainder is not 0 (i.e., it's 1 for integers), it means the number is odd, and its value is added to the `sum\_odd` variable using the `+=` operator.

Return Statement:

return sum\_odd, sum\_even

After the loop has finished iterating through all the numbers in the tuple, this line returns the final calculated sums of odd and even numbers.

The sums are returned as a tuple `(sum\_odd, sum\_even)`.

Example usage:

numbers\_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

odd\_sum, even\_sum = calculate\_odd\_even\_sums(numbers\_tuple)

print(f"Sum of odd numbers: {odd\_sum}")

print(f"Sum of even numbers: {even\_sum}")

This part demonstrates how to use the `calculate\_odd\_even\_sums` function.

It defines a sample tuple `numbers\_tuple`.

It then calls the function with this tuple and unpacks the returned tuple of sums into two separate variables, `odd\_sum` and `even\_sum`.

Finally, it prints the calculated sums using f-strings for formatted output.