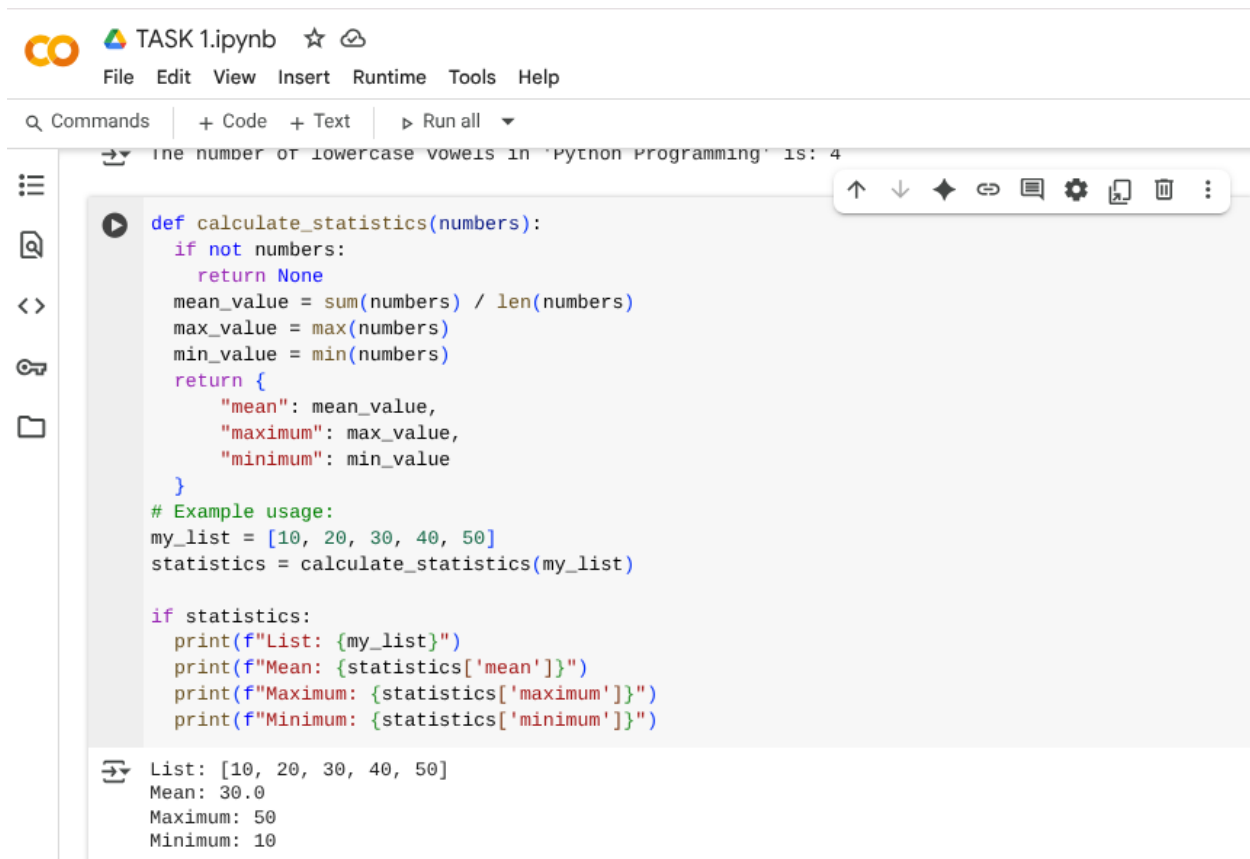


ASSIGNMENT-2

Task-1:

write a python program that reads a list of numbers and calculates mean, maximum and minimum values.

Code and Output:



The screenshot shows a Jupyter Notebook titled "TASK 1.ipynb". The code defines a function `calculate_statistics` that takes a list of numbers and returns a dictionary with the mean, maximum, and minimum values. An example usage is provided with the list `[10, 20, 30, 40, 50]`. The output of the code is displayed below the code cell.

```
def calculate_statistics(numbers):  
    if not numbers:  
        return None  
    mean_value = sum(numbers) / len(numbers)  
    max_value = max(numbers)  
    min_value = min(numbers)  
    return {  
        "mean": mean_value,  
        "maximum": max_value,  
        "minimum": min_value  
    }  
  
# Example usage:  
my_list = [10, 20, 30, 40, 50]  
statistics = calculate_statistics(my_list)  
  
if statistics:  
    print(f"List: {my_list}")  
    print(f"Mean: {statistics['mean']}")  
    print(f"Maximum: {statistics['maximum']}")  
    print(f"Minimum: {statistics['minimum']}")
```

Output:

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

Explanation:

This Python code defines a function `calculate_statistics` that takes a list of numbers as input. It checks if the list is empty and returns `None` if it is. Otherwise, it calculates the mean, maximum, and minimum values of the numbers in the list. The mean is calculated by summing the numbers and dividing by the count. The maximum and minimum are found using the built-in `max()` and `min()` functions. The function returns a dictionary containing the calculated mean, maximum, and minimum values. The example usage demonstrates how to call the function with different lists and print the results.

Task-2:

write a python function that checks the given number is an Armstrong or not.

Comparision between gemini and copilot-

Gemini:

Code and Output:

```

def is_armstrong_number(number):
    # Convert the number to a string to easily get the number of digits and individual
    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_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    # Check if the sum of powers equals the original number
    return sum_of_powers == number

# Example usage:
num1 = 153
if is_armstrong_number(num1):
    print(f"{num1} is an Armstrong number.")
else:
    print(f"{num1} is not an Armstrong number.") |

```

153 is an Armstrong number.
 123 is not an Armstrong number.
 9 is an Armstrong number.

Explanation:

The provided Python code defines a function called `is_armstrong_number` that determines if a given integer is an Armstrong number.

An Armstrong number is a number that equals the sum of its own digits, each raised to the power of the total number of digits in the number.

The function begins by converting the input `number` into a string to easily access its individual digits and determine the total count of digits (`num_digits`).

It then initializes a variable `sum_of_powers` to zero.

The code iterates through each character (digit) in the number's string representation. For each digit character, it converts it back to an integer.

This integer digit is then raised to the power of `num_digits`, and the result is added to `sum_of_powers`.

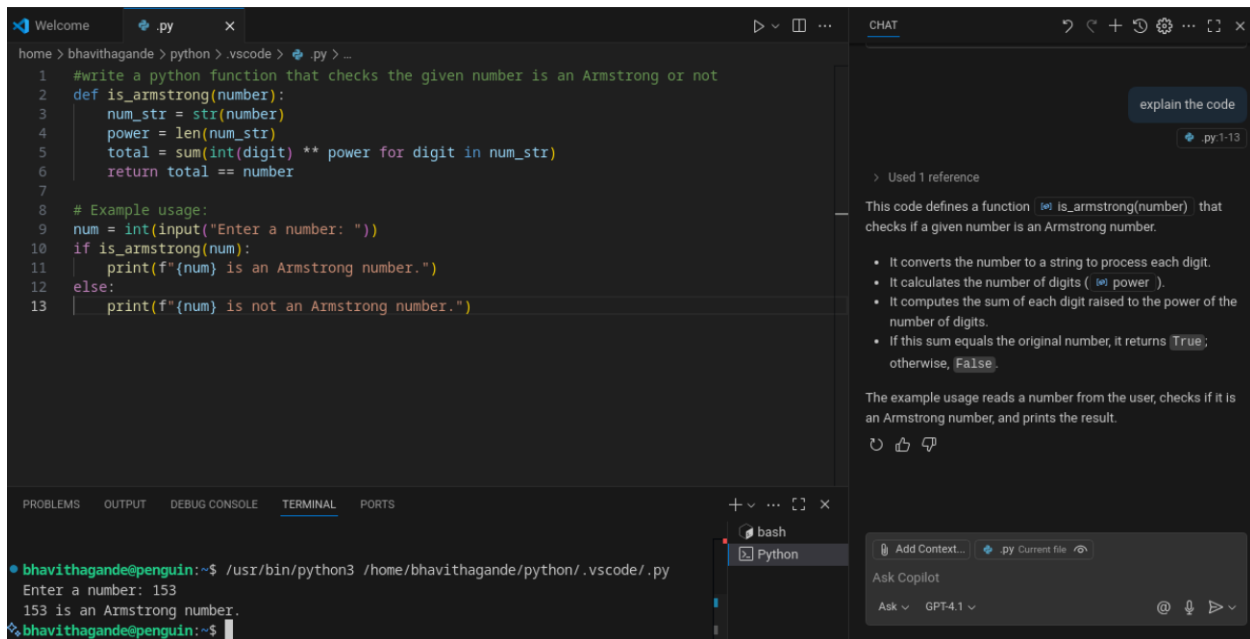
After processing all the digits, the function compares the calculated `sum_of_powers` with the original `number`.

If they are equal, the function returns `True`, indicating that the number is an Armstrong number.

If they are not equal, the function returns `False`.

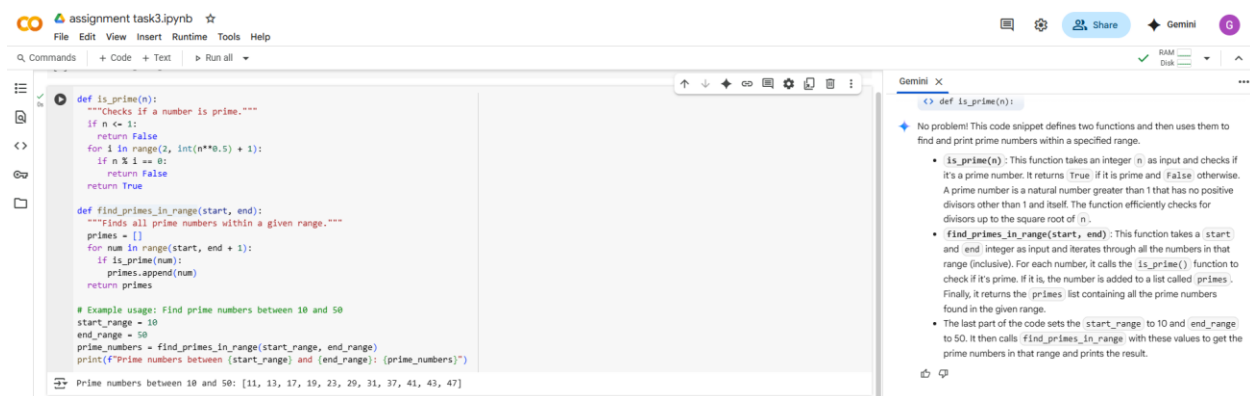
The example usage demonstrates how to call the `is_armstrong_number` function with different numbers (153, 123, and 9) and prints whether each number is an Armstrong number or not based on the function's return value.

Copilot:



Task-3:

Code, Output and Explanation:



Task-4:

Code and Output:

```

1  #write a python function that calculates the sum of the first N Natural Numbers.
2
3  def sum_natural_numbers_iterative(n):
4      """
5      Calculate the sum of first N natural numbers using iteration.
6
7      Args:
8          n (int): A positive integer
9
10     Returns:
11         int: Sum of first N natural numbers (1 + 2 + 3 + ... + n)
12     """
13     if n <= 0:
14         return 0
15
16     total = 0
17     for i in range(1, n + 1):
18         total += i
19     return total
20
21
22 def sum_natural_numbers_formula(n):
23     """
24     Calculate the sum of first N natural numbers using mathematical formula.
25     Formula: sum = n * (n + 1) / 2
26
27     Args:
28         n (int): A positive integer
29
30     Returns:
31         int: Sum of first N natural numbers
32     """
33     if n <= 0:
34         return 0
35
36     return n * (n + 1) // 2
37

```

```

37
38
39 def sum_natural_numbers_recursive(n):
40     """
41     Calculate the sum of first N natural numbers using recursion.
42
43     Args:
44         n (int): A positive integer
45
46     Returns:
47         int: Sum of first N natural numbers
48     """
49     if n <= 0:
50         return 0
51     elif n == 1:
52         return 1
53     else:
54         return n + sum_natural_numbers_recursive(n - 1)
55
56
57 # Example usage and testing
58 if __name__ == "__main__":
59     # Test cases
60     test_numbers = [5, 10, 100]
61
62     for num in test_numbers:
63         print(f"Sum of first {num} natural numbers:")
64         print(f"    Iterative: {sum_natural_numbers_iterative(num)}")
65         print(f"    Formula: {sum_natural_numbers_formula(num)}")
66         print(f"    Recursive: {sum_natural_numbers_recursive(num)}")
67         print()

```

Explanation:

Functions Implemented:

1. `sum_natural_numbers_iterative(n)` - Uses a simple loop to add numbers from 1 to n
2. `sum_natural_numbers_formula(n)` - Uses the mathematical formula: $n * (n + 1) / 2$ (most efficient)
3. `sum_natural_numbers_recursive(n)` - Uses recursion to calculate the sum

Key Features:

- **Input validation:** All functions handle non-positive numbers by returning 0
- **Documentation:** Each function has clear docstrings explaining parameters and return values
- **Test cases:** The code includes example usage with test cases for N = 5, 10, and 100
- **Expected results:** Comments show what the results should be

Performance Comparison:

- **Formula method** is the most efficient (O(1) time complexity)
- **Iterative method** has O(n) time complexity
- **Recursive method** has O(n) time complexity but uses more memory due to call stack

You can run the file to see all three methods in action with the test cases. The formula method is generally recommended for production use due to its efficiency, especially for large values of N.

Task-5:

Code, Output and Explanation:

The screenshot displays a Jupyter Notebook environment with the following components:

- File Explorer:** Shows a file named `assignment task5.ipynb`.
- Code Cell:** Contains a Python function `sum_odd_even(limit)` that calculates the sum of odd and even numbers up to a given limit. The function uses a `for` loop and conditional logic to separate odd and even numbers. Example usage is provided with `limit = 10`, resulting in odd sum 25 and even sum 30.
- Output Cell:** Displays the results of the example usage: "Sum of odd numbers up to 10: 25" and "Sum of even numbers up to 10: 30".
- Gemini AI Explanation:** A sidebar on the right provides a detailed breakdown of the code logic, explaining the function definition, docstring, variable initialization, loop structure, conditional checks for odd/even numbers, and the final return statement.