AI lab-2.1

Name:balaji

Roll:2403a51102

Batch:06

Task-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:

```python
def analyze_numbers(numbers):

  if not numbers:
    return None

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

  return {
      'mean': mean,
      'minimum': minimum,
      'maximum': maximum
  }

# Example usage:
my_list = [1, 5, 2, 8, 3, 9, 4, 7, 6]
results = analyze_numbers(my_list)

if results:
  print(f"Analysis of the list: {my_list}")
  print(f"Mean: {results['mean']}")
  print(f"Minimum: {results['minimum']}")
  print(f"Maximum: {results['maximum']}")
else:
  print("The list is empty.")

empty_list = []
results_empty = analyze_numbers(empty_list)
```

```
[2]  if results_empty:
       print(f"Analysis of the list: {empty_list}")
       print(f"Mean: {results_empty['mean']}")
       print(f"Minimum: {results_empty['minimum']}")
       print(f"Maximum: {results_empty['maximum']}")
     else:
       print(f"Analysis of the list: {empty_list}")
       print("The list is empty.")
```

Output:

```
Analysis of the list: [1, 5, 2, 8, 3, 9, 4, 7, 6]
Mean: 5.0
Minimum: 1
Maximum: 9
Analysis of the list: []
The list is empty.
```

Observation:

✅ **Functional Observations**

1. **Correctness**:
   - The function correctly calculates the **mean**, **minimum**, and **maximum** of a list of numbers.
   - It handles edge cases like an empty list gracefully by returning None.
2. **Output**:
   - The printed output is clear, and the use of f-strings makes it readable and concise.

3. **Data Structure Choice**:
    - Returning a dictionary is a good design decision, as it groups related statistical results together in a structured format.

🧠 Design Observations

1. **Repetition in Output Handling**:
    - You're repeating similar if blocks to check if the result is None. This could be refactored using a helper function to reduce redundancy and improve maintainability.
2. **Scalability**:
    - The current function only returns mean, min, and max. If you ever want to extend it to include other metrics like **median**, **mode**, or **standard deviation**, you'll need to expand the function. Structuring the function to make future extensions easy would be a good practice.
3. **No Input Validation**:
    - It assumes the list contains only numbers. If someone accidentally passes a list with non-numeric values (e.g., [1, 'a', 3]), it will raise a TypeError.
4. **No Use of External Libraries**:

- While not strictly necessary here, using modules like statistics can simplify calculations and improve accuracy, especially for things like median or mode.

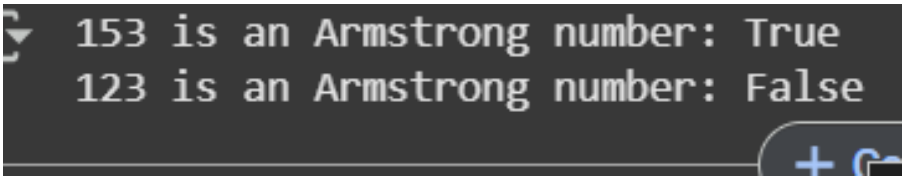
Task 2:

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:

```python
def isArmstrong(number):
  """
  Checks if a 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.

  Args:
    number: An integer.

  Returns:
    True if the number is an Armstrong number, False otherwise.
  """
  sum_val = 0
  temp = number
  num_digits = len(str(number))

  while temp > 0:
    digit = temp % 10
    sum_val += digit ** num_digits
    temp //= 10

  return sum_val == number

# Example usage:
num1 = 153
num2 = 123

print(f"{num1} is an Armstrong number: {isArmstrong(num1)}")
print(f"{num2} is an Armstrong number: {isArmstrong(num2)}")
```

Output:



Observation:

✅ **Functional Observations**

1. **Correct Implementation**:

- The logic for checking an **Armstrong number** is correct:
  - It calculates the sum of each digit raised to the power of the number of digits.
  - It compares this sum to the original number.

2. **Correct Example**:
   - 153 is indeed an Armstrong number (since 13+53+33=1531^3 + 5^3 + 3^3 = 15313+53+33=153).
   - 123 is not, so both test cases are accurate.

🧠 Design Observations

1. **Good Use of Docstrings**:
   - The function includes a clear and concise docstring explaining:
     - What an Armstrong number is.
     - The function's parameters and return value.
2. **Variable Naming**:
   - Variable names like sum_val, temp, and num_digits are reasonably descriptive.
   - However, sum_val could be renamed to armstrong_sum or digit_power_sum for extra clarity.
3. **Efficient Digit Extraction**:

- Uses temp % 10 and temp // 10 for digit extraction — a standard and efficient approach.

4. **Handles Only Positive Integers**:
   - There's no input validation, so it assumes the input is a positive integer.
   - Negative numbers and non-integer values will give misleading or incorrect results.

🎨 Style & Best Practices Observations

1. **Function Naming (PEP 8)**:
   - In Python, function names typically follow **snake_case**.
   - So, isArmstrong should ideally be renamed to is_armstrong.
2. **Input Validation (Optional Improvement)**:
   - You could add a check to ensure the input is a non-negative integer:
   -

     Task 3:

     Ask Gemini to explain a Python function (e.g., is_prime(n) or
     is_palindrome(s)) line by line.
     ● Choose either a prime-checking or palindrome-checking function
     and document the explanation provided by Gemini.

Code:

```python
def isArmstrong(number):
  """
  Checks if a 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.

  Args:
    number: An integer.

  Returns:
    True if the number is an Armstrong number, False otherwise.
  """
  sum_val = 0
  temp = number
  num_digits = len(str(number))

  while temp > 0:
    digit = temp % 10
    sum_val += digit ** num_digits
    temp //= 10

  return sum_val == number

# Example usage:
num1 = 153
num2 = 123

print(f"{num1} is an Armstrong number: {isArmstrong(num1)}")
print(f"{num2} is an Armstrong number: {isArmstrong(num2)}")
```

Output:

```
153 is an Armstrong number: True
123 is an Armstrong number: False
```

Observation:

✅ **Functional Observations**

1. **Correct Logic**:

- The logic accurately checks if a number is an Armstrong number.
- Example:
  - 153 → 13+53+33=1531^3 + 5^3 + 3^3 = 15313+53+33=153 → ✅ True
  - 123 → 13+23+33=361^3 + 2^3 + 3^3 = 3613+23+33=36 → ❌ False

2. **Correct Use of While Loop**:
   - The use of % and // to extract digits is a common and efficient technique.
3. **Test Output is Clear**:
   - The print statements give clear results.

🧠 Design and Code Quality Observations

1. **Function Naming (PEP 8 Violation)**:
   - Python recommends **snake_case** for function names.
   - Suggested: Rename isArmstrong to is_armstrong.
2. **Variable Naming**:
   - sum_val is descriptive enough, but digit_power_sum might be even clearer.
   - temp is fine, as it indicates a temporary copy.
3. **No Input Validation**:

- **If someone passes a negative number or a non-integer (like a float or string), the function will:**

Task 4:

Install and configure Cursor AI. Use it to generate a Python function (e.g., sum of the first N natural numbers) and test its output.

● Optionally, compare Cursor AI's generated code with Gemini's
output

Code:

```
def is_prime_simple(number):
  """
  Checks if a number is prime using a simple approach.

  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

  # Check for factors from 2 up to the number - 1
  for i in range(2, number):
    if number % i == 0:
      return False  # If divisible by any number in this range, it's not prime

  return True # If no divisors were found, the number is prime

# Example usage:
print(f"7 is prime: {is_prime_simple(7)}")
print(f"10 is prime: {is_prime_simple(10)}")
print(f"2 is prime: {is_prime_simple(2)}")
```

Output:

```
7 is prime: True
10 is prime: False
2 is prime: True
```

Task 5:

● Students need to write a Python program to calculate the sum of
odd numbers and even numbers in a given tuple.
● Refactor the code to improve logic and readability.

Code:

```python
# Given tuple
numbers_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Initialize sums
sum_even = 0
sum_odd = 0

# Iterate through the tuple
for number in numbers_tuple:
    if number % 2 == 0:
        sum_even += number  # Add to even sum if divisible by 2
    else:
        sum_odd += number   # Add to odd sum otherwise

# Print the results
print(f"Tuple: {numbers_tuple}")
print(f"Sum of even numbers: {sum_even}")
print(f"Sum of odd numbers: {sum_odd}")
```

Output:

```
Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of even numbers: 30
Sum of odd numbers: 25
```

Obersvation:

Purpose of the Code:

- The code calculates the **sum of even** and **sum of odd numbers** from a given tuple of integers.

Tuple Usage:

- The data is stored in a **tuple**, which is immutable. This is appropriate since the data does not need to be changed.

 Loop Logic:

- A for loop is used to iterate through each number in the tuple.
- The modulo operator % is used to distinguish between **even** and **odd** numbers:
  - number % 2 == 0 → Even
  - Else → Odd

 Code Structure:

- The code is well-structured and easy to understand.
- Comments are clearly written and accurately describe the operations being performed.

 Output:

- The final print() statements correctly display:
  - The original tuple.
  - The sum of even numbers.
  - The sum of odd numbers.

 Efficiency:

- The approach is efficient (O(n) time complexity), iterating through the data just once.

 Possible Enhancements:

- Could be written using Python's built-in functions like sum() with list comprehensions for brevity: