ASSIGNMENT-2.1

Name :D. Sravika Reddy　　　　　Hall Ticket No: 2403A510d0

Batch No:06　　　　　　　　　　Course: AI Assisted Coding

Task Description #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.

Expected Output #1

● Functional code with correct output and screenshot.

```python
def analyze_numbers(numbers):
  """
  Calculates the mean, minimum, and maximum of a list of numbers.

  Args:
    numbers: A list of numbers.

  Returns:
    A dictionary containing the mean, minimum, and maximum values.
  """
  if not numbers:
    return {"mean": None, "min": None, "max": None}

  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
my_list = [10, 20, 30, 40, 50]
results = analyze_numbers(my_list)
print(results)
```

```
{'mean': 30.0, 'min': 10, 'max': 50}
```

```
about the analyze_numbers function and the provided code:

Error Handling: The function includes a check for an empty list (if not
numbers:). If the input list is empty, it returns a dictionary with None values
for mean, min, and max, preventing a ZeroDivisionError when trying to calculate
the mean.
Efficiency: For calculating the mean, sum(numbers) and len(numbers) are
efficient built-in Python operations. Similarly, min(numbers) and max(numbers)
are efficient for finding the minimum and maximum values, especially for
moderately sized lists.
Readability: The code is well-structured with a clear function definition, a
docstring explaining its purpose, arguments, and return value, and a simple
example usage. This makes the code easy to understand and maintain.
Return Type: The function returns a dictionary, which is a convenient way to
group the three calculated values (mean, min, and max) and label them clearly.
Data Types: The mean is returned as a float (30.0), while the min and max are
returned as integers (10 and 50), which is consistent with the input data.
```

about the analyze_numbers function and the provided code:

Error Handling: The function includes a check for an empty list (if not numbers:). If the input list is empty, it returns a dictionary with None values for mean, min, and max, preventing a ZeroDivisionError when trying to calculate the mean. Efficiency: For calculating the mean, sum(numbers) and len(numbers) are efficient built-in Python operations. Similarly, min(numbers) and max(numbers) are efficient for finding the minimum and maximum values, especially for moderately sized lists. Readability: The code is well-structured with a clear function definition, a docstring explaining its purpose, arguments, and return value, and a simple example usage. This makes the code easy to understand and maintain. Return Type: The function returns a dictionary, which is a convenient way to group the three calculated values (mean, min, and max) and label them clearly. Data Types: The mean is returned as a float (30.0), while the min and max are returned as integers (10 and 50), which is consistent with the input data.

Task Description #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.

Expected Output #2

● Side-by-side comparison table with observations and screenshots

Gemini's Code:

```python
def is_armstrong(number):
  # Store the original number to compare later
  original_number = number

  # Calculate the number of digits
  num_digits = len(str(number))

  # Initialize the sum of powers
  sum_of_powers = 0

  # Loop through each digit
  while number > 0:
    # Get the last digit
    digit = number % 10

    # Add the digit raised to the power of the number of digits to the sum
    sum_of_powers += digit ** num_digits

    # Remove the last digit
    number //= 10

  # Return True if the sum equals the original number, otherwise False
  return sum_of_powers == original_number

# Example usage:
print(is_armstrong(153))  # Output: True
print(is_armstrong(123))  # Output: False
```

```
True
False
```

Copilots code:

```python
def is_armstrong(number):
    """Checks if a number is an Armstrong number."""
    # Convert number to string to find the number of digits
    num_str = str(number)
    num_digits = len(num_str)

    # Calculate the sum of the digits raised to the power of num_digits
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)

    # Return true if the sum equals the original number
    return armstrong_sum == number

# Example usage
print(is_armstrong(153))  # True
print(is_armstrong(9474)) # True
print(is_armstrong(123))  # False
```

```
True
True
False
```

📊 Side-by-Side Comparison

Feature	Gemini's Output	Copilot's Output	Observations
Code Style	**Verbose and procedural**. Uses a `while` loop with intermediate variables.	**Concise and Pythonic**. Uses a list comprehension and the `sum()` function.	Copilot's approach is more compact, while Gemini's is easier to follow for someone new to Python.
Explanation	**Detailed**. Includes a docstring and inline comments explaining each step of the algorithm.	**Minimal**. Provides a short docstring and no inline comments.	Gemini acts more like a teacher, providing a thorough explanation. Copilot assumes the user is familiar with the concept.
Readability	**High**. The explicit steps and comments make the logic very clear for all skill levels.	**High (for experienced Python users)**. The one-liner is elegant but may be less intuitive for beginners.	Both are readable, but for different audiences. Gemini is more beginner-friendly.
Output Format	A complete code block with an explanation of the logic and a single example.	A complete code block with a one-line explanation and multiple examples.	Gemini's output is designed for learning, while Copilot's is designed for productivity.

Gemini's Observation:

Observations:

- Gemini's code is highly readable due to descriptive variable names and a well-structured docstring.
- The function uses a `while` loop, which is a common and efficient approach for this problem.
- It provides a clear, detailed explanation of the logic, making it suitable for both beginners and experienced developers.
- The code includes comments that explain each step, which is an excellent practice for maintainability.

Copilots Observation:

Observations:

- Copilot's code is more **concise**, using a list comprehension and the `sum()` function to calculate the result in a single line. This is a very "Pythonic" way to solve the problem.
- The docstring is much shorter, focusing only on the function's purpose.
- It provides multiple examples, which is a good practice for demonstrating functionality.
- The output is minimal and straight to the point, which is useful for experienced programmers who prefer a cleaner, less verbose response.

Task Description #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.

Expected Output #3

● Detailed explanation with the code snippet and Gemini's response.

Gemini's code:

Python

````python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```"
````

🤖 Gemini's Detailed Explanation

Gemini provides a clear, line-by-line breakdown of the function's logic. It starts with a summary and then goes through the code sequentially, explaining what each line or block of code does and why it's there. This is a very helpful approach for someone trying to understand the code's mechanics.

Code Snippet:

Here's the code snippet:

Python

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```
```

Here is Gemini's response:

"This Python function, `is_prime(n)`, is an efficient way to determine if a given number `n` is prime. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Here's the line-by-line explanation:

1. `def is_prime(n):`
    - This line **defines a function** named `is_prime`.
    - It takes one argument, `n`, which is the number we want to check.
    - All the code indented beneath this line belongs to this function.
2. `if n <= 1:`
    - This is the first line of code that executes inside the function.
    - It checks if the number `n` is less than or equal to 1.
    - By definition, prime numbers must be greater than 1. So, any number that is 1 or less is not prime.
3. `return False`
    - If the condition `n <= 1` is true, this line executes immediately.
    - It **returns** `False`, which means the function stops executing and gives the result `False`. The rest of the code in the function is skipped.
4. `for i in range(2, int(n**0.5) + 1):`
    - This line starts a `for` **loop**. This loop is the core of the primality test.
    - It iterates through a range of numbers.
    - The `range` starts at `2` because we've already handled numbers less than or equal to 1, and every number is divisible by 1.
    - The loop's end condition is `int(n**0.5) + 1`. This is a key optimization. If a number `n` has a divisor, at least one of its divisors must be less than or equal to its square root. So, we only need to check for divisors up to the square root of `n`. The `+ 1` ensures that the range includes the square root itself if it's an integer.
5. `if n % i == 0:`
    - Inside the loop, this line checks if there is **any remainder** when `n` is divided by the current number `i`.
    - The modulus operator (`%`) returns the remainder of the division.
    - If the remainder is `0`, it means `i` is a **divisor** of `n`.
6. `return False`
    - If the condition `n % i == 0` is true, this line executes.
    - It **returns** `False` because if we find even one divisor (other than 1 and itself), the number is not prime.
    - The function stops here, and we don't need to check any further numbers.
7. `return True`
    - This line is only reached if the `for` loop **completes without finding any divisors**.
    - If the loop finishes, it means `n` was not divisible by any number in the range from 2 up to its square root.

Task Description #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.

Expected Output #4

● Screenshots of Cursor AI setup, prompts used, and generated code with output.

```python
def sum_first_n(n: int) -> int:
    """
    Return the sum of the first n natural numbers (0 through n) using Gauss's formula.
    Raises:
        TypeError: if n is not an integer
        ValueError: if n is negative
    """
    if not isinstance(n, int):
        raise TypeError("n must be an int")
    if n < 0:
        raise ValueError("n must be >= 0")
    return n * (n + 1) // 2


if __name__ == "__main__":
    # Demonstration
    for n in (0, 1, 5, 10):
        print(f"sum_first_n({n}) = {sum_first_n(n)}")

    # Simple correctness checks
    assert sum_first_n(0) == 0
    assert sum_first_n(1) == 1
    assert sum_first_n(5) == 15
    assert sum_first_n(10) == 55

    print("All tests passed.")
```

Comparison: Cursor AI vs. Gemini

While a direct, side-by-side comparison with screenshots of Cursor AI is not possible, I can provide a conceptual comparison of how both tools typically handle such a request.

- **Gemini's Output (as demonstrated above):** Gemini often provides a complete, well-documented function with a clear docstring, an explanation of the logic (like using a formula instead of a loop), and a test case. Its responses are comprehensive and educational, explaining not just the *what* but also the *why*. It focuses on providing a full solution and context around it.
- **Cursor AI's Output:** Cursor AI, as an in-editor AI, would likely be more integrated and responsive. When you type `def sum_first_n_naturals(n):`, Cursor would likely offer a code completion suggestion for the entire function body. Its primary value is in real-time assistance and a seamless coding experience. The output would be a concise, correct function, possibly without the extensive comments or explanations that Gemini provides. It's designed to be a coding copilot, focused on speed and efficiency within the development environment, rather than a standalone educational tool.

```python
def sum_first_n(n: int) -> int:
    if not isinstance(n, int):
        raise TypeError("n must be an int")
    if n < 0:
        raise ValueError("n must be >= 0")
    return n * (n + 1) // 2


if __name__ == "__main__":
    # Demonstration
    for n in (0, 1, 5, 10):
        print(f"sum_first_n({n}) = {sum_first_n(n)}")

    # Simple correctness checks
    assert sum_first_n(0) == 0
    assert sum_first_n(1) == 1
    assert sum_first_n(5) == 15
    assert sum_first_n(10) == 55

    print("All tests passed.")
```

```
sum_first_n(0) = 0
sum_first_n(1) = 1
sum_first_n(5) = 15
sum_first_n(10) = 55
All tests passed.
```

Task Description #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.

Expected Output #5

● Student-written refactored code with explanations and output screenshots.

```
def sum_even_odd_numbers(numbers_tuple):
    # Use a dictionary to store the sums for better readability.
    sums = {'even_sum': 0, 'odd_sum': 0}

    # Iterate through each number in the tuple directly.
    # This is more readable than using `range(len(numbers_tuple))`.
    for number in numbers_tuple:
        if number % 2 == 0:
            # If the number is even, add it to the even_sum.
            sums['even_sum'] += number
        else:
            # If the number is odd, add it to the odd_sum.
            sums['odd_sum'] += number

    return sums

# Given tuple of numbers
my_numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

# Call the function and get the result
result = sum_even_odd_numbers(my_numbers)

# Print the results
print(f"The original tuple is: {my_numbers}")
print(f"Sum of even numbers: {result['even_sum']}")
print(f"Sum of odd numbers: {result['odd_sum']}")
```

```
The original tuple is: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
Sum of even numbers: 30
Sum of odd numbers: 36
```

🖼️ Explanation and Output

This refactored code improves on the original student-written version in a few key ways:

1. **Readability**: Instead of using a traditional `for i in range(len(tuple))` loop, we use a more "Pythonic" `for number in numbers_tuple` loop. This makes the code easier to read and understand, as you are directly working with the number itself rather than an index.
2. **Clarity**: By using a dictionary (`sums`) to hold the results, the code is much clearer. You can access the sums using descriptive keys like `'even_sum'` and `'odd_sum'`, which is more intuitive than using separate variables.
3. **Efficiency**: The logic remains a single pass through the tuple, making it efficient for tuples of any size.

The output will clearly show the sums of both the even and odd numbers in the given tuple.