ASSIGNMENT-2.3

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Batch : 05

Course : AI Assisted Coding

Task Description#1

● Use Google Gemini in Colab to write a function that reads a CSV file and calculates
mean, min, max.

Output#1

● Functional code with output and screenshot.

```
import pandas as pd

def analyze_csv(file_path):
  """
  Reads a CSV file and calculates the mean, min, and max for each column.

  Args:
    file_path: The path to the CSV file.

  Returns:
    A pandas DataFrame containing the mean, min, and max for each column,
    or None if an error occurs.
  """
  try:
    df = pd.read_csv(file_path)
    description = df.describe().loc[['mean', 'min', 'max']]
    return description
  except FileNotFoundError:
    print(f"Error: File not found at {file_path}")
    return None
  except Exception as e:
    print(f"An error occurred: {e}")
    return None
```

Explanation:

```
This code defines a Python function called analyze_csv that takes one
argument, file_path, which is expected to be the path to a CSV file.

Here's a breakdown of what the code does:

Function Definition: Defines a function analyze_csv that accepts
file_path as input.
Docstring: The triple-quoted string is a docstring explaining the
function's purpose, arguments (Args), and what it returns (Returns).
Error Handling: The try...except block is used for error handling. It
attempts to execute the code within the try block.
Read CSV: pd.read_csv(file_path) attempts to read the CSV file
specified by file_path into a pandas DataFrame.
Describe and Select: df.describe() generates descriptive statistics for
the DataFrame (including mean, min, max, etc.). .loc[['mean', 'min',
'max']] then selects only the rows corresponding to 'mean', 'min', and
'max' from the descriptive statistics.
Return Result: If the file is read successfully and statistics are
calculated, the resulting DataFrame with mean, min, and max is returned.
Handle File Not Found: If a FileNotFoundError occurs (meaning the
specified file doesn't exist), it prints an error message and returns
None.
Handle Other Exceptions: If any other exception occurs during the
process, it prints a generic error message along with the exception
details and returns None.
This function is a useful tool for quickly getting a summary of the
central tendency and range of values in a CSV file.
```

This code defines a Python function called analyze_csv that takes one argument, file_path, which is expected to be the path to a CSV file.

Here's a breakdown of what the code does:

Function Definition: Defines a function analyze_csv that accepts file_path as input. Docstring: The triple-quoted string is a docstring explaining the function's purpose, arguments (Args), and what it returns (Returns). Error Handling: The try...except block is used for error handling. It attempts to execute the code within the try block. Read CSV: pd.read_csv(file_path) attempts to read the CSV file specified by file_path into a pandas DataFrame. Describe and Select: df.describe() generates descriptive statistics for the DataFrame (including mean, min, max, etc.). .loc[['mean', 'min', 'max']] then selects only the rows corresponding to 'mean', 'min', and 'max' from the descriptive statistics. Return Result: If the file is read successfully and statistics are calculated, the resulting DataFrame with mean, min, and max is returned. Handle File Not Found: If a FileNotFoundError occurs (meaning the specified file doesn't exist), it prints an error message and returns None. Handle Other Exceptions: If any other exception occurs during the process, it prints a generic error message along with the exception details and returns None. This function is a useful tool for quickly getting a summary of the central tendency and range of values in a CSV file.

Task Description#2

● Compare Gemini and Copilot outputs for a palindrome check function.

Output#2

● Side-by-side comparison and observations

Gemini's Code:

Python

```python
def is_palindrome_two_pointers(word):
    """
    Checks if a word is a palindrome using a two-pointer approach.

    Args:
        word (str): The string to check.

    Returns:
        bool: True if the word is a palindrome, False otherwise.
    """
    left = 0
    right = len(word) - 1

    while left < right:
        if word[left] != word[right]:
            return False
        left += 1
        right -= 1

    return True
```

- `"racecar"` is a palindrome.

```
is_palindrome_two_pointers("racecar")  # Output: True
```

- `"hello"` is not a palindrome.

```
is_palindrome_two_pointers("hello")    # Output: False
```

- `"madam"` is a palindrome.

```
is_palindrome_two_pointers("madam")    # Output: True
```

- `"12321"` is a palindrome.

```
is_palindrome_two_pointers("12321")    # Output: True
```

Copilots Code:

Python

```python
def check_palindrome(word):
    return word == word[::-1]
```

Function Output and Examples

This version of the function is concise but doesn't handle spaces, punctuation, or different casing.

- **Simple Palindrome:**

```python
check_palindrome("level")  # Output: True
```

- **Not a Palindrome:**

```python
check_palindrome("hello")  # Output: False
```

- **Case-Sensitive Failure:**

```python
check_palindrome("Madam")  # Output: False
```

Side-by-Side by Comparisons and Observations:

Side-by-Side Code Comparison

String Slicing Method (Gemini/Copilot's Default)	Two-Pointer Method (Gemini's Alternative)

python
def is_palindrome(word):
 return word == word[::-1>
 	
	python
	def is_palindrome(word):
	 left = 0
	 right = len(word) - 1
	 while left < right:
	 if word[left] != word[right]:
	 return False
	 left += 1
	 right -= 1
	 return True

Export to Sheets

Observations and Comparison Table

Aspect	String Slicing Method	Two-Pointer Method
Readability	**High.** It's a single, highly intuitive line of code that is easy for any Python developer to understand.	**Medium.** It's a more traditional algorithm that requires reading through the loop logic to understand the process.
Conciseness	**Extremely high.** It is often referred to as the "Pythonic" one-liner solution.	**Low.** Requires multiple lines of code to set up and manage pointers and the loop.
Performance (Time)	**Very fast** for most inputs. The string slicing and comparison are implemented in **highly optimized C code** under the hood, making them exceptionally efficient for most common use cases.	**Fast.** The time complexity is $O(n)$ where n is the length of the string because it only checks up to half of the characters. However, because it's interpreted Python code, it's often slightly slower than the C-optimized slicing.
Memory Usage	**Low for small strings, high for large ones.** Creating a reversed copy of the string requires $O(n)$ **extra space** in memory, where n is the length of the string.	**Extremely low.** This method operates in-place with $O(1)$ **constant extra space** because it only stores two integer variables (`left` and `right`). This makes it ideal for very large strings.
Use Case	**General purpose.** It's the best choice for everyday coding, simple scripts, and most interview questions due to its simplicity and readability.	**Performance-critical applications.** The ideal choice when working with extremely long strings where memory usage is a significant concern. It demonstrates a deeper understanding of algorithms.

Task Description#3

● Ask Gemini to explain a Python function (to calculate area of various shapes) line by

line..

Output#3

● Detailed explanation with code snippet

```python
import math

def calculate_area(shape, **kwargs):
    """
    Calculates the area of various shapes.

    Args:
        shape (str): The name of the shape ('circle', 'rectangle', 'triangle').
        **kwargs: A dictionary of keyword arguments for the dimensions.

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape or arguments are invalid.
    """
    if shape == 'circle':
        if 'radius' in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: A circle requires a radius."

    elif shape == 'rectangle':
        if 'length' in kwargs and 'width' in kwargs:
            return kwargs['length'] * kwargs['width']
        else:
            return "Error: A rectangle requires both length and width."

    elif shape == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
            return 0.5 * kwargs['base'] * kwargs['height']
        else:
            return "Error: A triangle requires both base and height."

    else:
        return "Error: Invalid shape provided."
```

Explanation :

The provided function, `calculate_area`, is designed to compute the area of a circle, rectangle, or triangle. It's a versatile function because it uses `**kwargs` to accept different dimensional arguments for each shape.

Key Aspects:

- `import math` : Imports the `math` library to use the constant `math.pi` .
- `def calculate_area(shape, **kwargs)` : Defines the function that takes a `shape` string and a flexible number of keyword arguments (`**kwargs`).
- `if-elif-else` : The code uses a series of conditional statements to determine the shape and execute the correct area formula.
- **Keyword Argument Check**: For each shape, it checks if the necessary arguments (e.g., `'radius'` , `'length'` and `'width'`) are present in the `kwargs` dictionary.
- **Area Calculation**: Based on the shape, it applies the appropriate formula:
  - **Circle**: `π * radius²`
  - **Rectangle**: `length * width`
  - **Triangle**: `0.5 * base * height`
- **Error Handling**: If a shape or a required argument is missing, the function returns a clear error message. This makes the function robust and user-friendly.

Task Description#4

● Install and configure Cursor AI. Use it to generate a Python function (e.g., sum of squares).

Output#4

● Screenshots of working environments with few prompts to generate python code.

```python
def sum_of_squares(numbers):
    """
    Calculate the sum of squares for a list of numbers.

    Args:
        numbers (list): A list of numbers (integers or float

    Returns:
        float: The sum of squares of all numbers in the list

    Examples:
        >>> sum_of_squares([1, 2, 3])
        14
        >>> sum_of_squares([2.5, 3.5])
        18.5
    """
    if not numbers:
        return 0

    return sum(num ** 2 for num in numbers)


# Example usage and testing
if __name__ == "__main__":
    # Test cases
    test_numbers = [1, 2, 3, 4, 5]
    result = sum_of_squares(test_numbers)
    print(f"Sum of squares of {test_numbers}: {result}")

    # Test with empty list
    print(f"Sum of squares of empty list: {sum_of_squares([]
```

```python
    # Test with decimal numbers
    decimal_numbers = [1.5, 2.5, 3.5]
    decimal_result = sum_of_squares(decimal_numbers)
    print(f"Sum of squares of {decimal_numbers}: {decimal_re

    # Test with single number
    single_number = [7]
    single_result = sum_of_squares(single_number)
    print(f"Sum of squares of {single_number}: {single_resul
```

This function:

- Takes a list of numbers as input
- Calculates the sum of squares (each number squared, then summed)
- Includes proper documentation with docstring
- Has error handling for empty lists
- Includes example usage and test cases
- Can handle both integers and floating-point numbers
- Uses a generator expression for memory efficiency

You can run this file directly to see the test results, or import the `sum_of_squares` function to use it in other Python scripts.

Explanation:

Function Explanation

The `sum_of_squares(numbers)` function takes one argument: a list of numbers.

- `if not numbers:` : This line is an **edge case** check. It handles the scenario where an empty list is passed to the function. If the list is empty, the expression `not numbers` evaluates to `True` , and the function immediately returns `0` since the sum of squares of an empty list is zero.
- `return sum(num ** 2 for num in numbers)` : This is the core logic of the function. It's a highly efficient and concise way to perform the calculation using a **generator expression** and Python's built-in `sum()` function.
  - `num ** 2` : For each number (`num`) in the input `numbers` list, this part squares it.
  - `for num in numbers` : This part iterates over each element of the input list.
  - `sum(...)` : The `sum()` function then adds up all the squared values as they are generated, returning the total sum.

Example Usage and Testing

The `if __name__ == "__main__":` block is a standard Python convention. The code inside this block will only run when the script is executed directly, not when it's imported as a module into another script. This is used to demonstrate the function's usage and test its correctness with different types of input.

- `test_numbers = [1, 2, 3, 4, 5]` : This test checks the function's behavior with a typical list of positive integers. The expected output is $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$.
- `sum_of_squares([])` : This test specifically checks the **empty list** edge case, ensuring the function correctly returns `0` as intended by the first line of the function.
- `decimal_numbers = [1.5, 2.5, 3.5]` : This test demonstrates that the function correctly handles **floating-point numbers**. The output will be a float ($1.5^2 + 2.5^2 + 3.5^2 = 2.25 + 6.25 + 12.25 = 20.75$).
- `single_number = [7]` : This test ensures the function works correctly with a list containing a **single element**. The output will be $7^2 = 49$.

Task Description#5

● Student need to write code to calculate sum of add number and even numbers in the
list

Output#5

● Refactored code written by student with improved logic

Python

```python
def calculate_sums_with_generators(numbers):
    """
    Calculates the sum of even and odd numbers using generator expressions.

    Args:
        numbers (list): A list of integers.

    Returns:
        tuple: A tuple containing the sum of even numbers and the sum of odd numbe
    """
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum

# Example Usage
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum, odd_sum = calculate_sums_with_generators(my_list)

print(f"Original list: {my_list}")
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

Explanation:

This approach leverages the power of Python's built-in functions and syntactic sugar.

- `sum(...)` : This is a powerful built-in function that takes an iterable (like a list or a generator) and returns the sum of its elements. It handles the accumulation for you, eliminating the need for a separate accumulator variable (`even_sum = 0`).
- `num for num in numbers if num % 2 == 0` : This is a **generator expression**. Think of it as a blueprint for a sequence of numbers. It tells Python: "For every number in the `numbers` list, if that number is divisible by 2 with no remainder (`num % 2 == 0`), yield that number." It generates the numbers one by one and passes them directly to the `sum()` function, which adds them up. This is very memory-efficient as it doesn't create a new list in memory.
- `num for num in numbers if num % 2 != 0` : This works identically to the previous expression but uses a different condition (`!= 0`) to filter for all the **odd numbers**.

While this approach is praised for its **concise syntax**, it does require **two passes** through the list —one to calculate the even sum and another for the odd sum. For most applications, this performance difference is negligible, but for extremely large lists or in performance-critical scenarios, the single-loop method is technically more efficient.