ASSIGNMENT-2.3

Name: D. Sravika Reddy   Hall-Ticket: 2403a510d0

Batch No: 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.

Expected 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

# Example usage (replace 'your_file.csv' with the actual file path)
# result = analyze_csv('your_file.csv')
# if result is not None:
#   display(result)
```

Explanation:

```
The code cell I generated contains a function analyze_csv that can read
a CSV file and provide descriptive statistics (mean, min, and max) for
each column.

To use this function, you can replace 'your_file.csv' in the example
usage with the actual path to your CSV file and uncomment the lines to
call the function and display the result.

Here are some potential next steps:

Provide a CSV file path: Give me the path to your CSV file so I can
call the analyze_csv function for you.
Visualize the data: Would you like to create some visualizations of the
data in your CSV file?
Perform more advanced analysis: We could explore other statistical
measures or perform more in-depth analysis on your data.
```

The code cell I generated contains a function analyze_csv that can read a CSV file and provide descriptive statistics (mean, min, and max) for each column.

To use this function, you can replace 'your_file.csv' in the example usage with the actual path to your CSV file and uncomment the lines to call the function and display the result.

Here are some potential next steps:

Provide a CSV file path: Give me the path to your CSV file so I can call the analyze_csv function for you. Visualize the data: Would you like to create some visualizations of the data in your CSV file? Perform more advanced analysis: We could explore other statistical measures or perform more in-depth analysis on your data.

Task Description#2

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

Expected Output#2

● Side-by-side comparison and observations

Gemini's code:

Python

```python
def is_palindrome(word):
    """
    Checks if a word is a palindrome.

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

    Returns:
        bool: True if the word is a palindrome, False otherwise.
    """
    # Reverse the word using slicing [::-1]
    reversed_word = word[::-1]

    # Compare the original word to the reversed word
    return word == reversed_word
```

Function Output and Examples

Below are a few examples demonstrating how the `is_palindrome` function works.

- `"racecar"` is a palindrome.

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

- `"hello"` is not a palindrome.

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

- `"Madam"` is not a palindrome because the function is case-sensitive.

```
is_palindrome("Madam")    # Output: False
```

Copilots code:

Given a prompt like `def is_palindrome(text):` , Copilot's most likely response would be a clean, concise, and robust function that handles common edge cases.

Python

```python
def is_palindrome(text):
    """
    Checks if a string is a palindrome, ignoring case and non-alphanumeric characte
    """
    cleaned_text = "".join(char.lower() for char in text if char.isalnum())
    return cleaned_text == cleaned_text[::-1]
```

- **Simple Word:**

```
# Input: "racecar"
# Expected Output: True

is_palindrome("racecar")
```

- **Simple Word (not a palindrome):**

```
# Input: "hello"
# Expected Output: False

is_palindrome("hello")
```

Side-by-side Comparisons and Observations:

Output Code	python def is_palindrome(text): cleaned_text = "".join(char.lower() for char in text if char.isalnum()) return cleaned_text == cleaned_text[::-1] 	python def is_palindrome(text): cleaned_text = "".join(char.lower() for char in text if char.isalnum()) return cleaned_text == cleaned_text[::-1]
Core Function	A **conversational assistant** that explains and generates code.	A **real-time code completion** tool integrated into an IDE.
User Experience	You type a full prompt in a chat interface and get a complete response with code and explanation.	It predicts and suggests code as you type, providing seamless, single-line or multi-line completions.
Primary Use Case	**Learning and problem-solving.** Best for when you need to understand the 'why' behind the code, discuss different approaches, or debug.	**Speed and efficiency.** Ideal for rapidly writing boilerplate code, automating repetitive tasks, and minimizing keystrokes for experienced developers.
Explanation	Provides **detailed explanations** of the code's logic, discussing how it handles case sensitivity, punctuation, and efficiency.	**Does not provide explanations.** It assumes the developer understands the code it generates.
Output Format	Delivers code as part of a more extensive, **structured text response**, often with code blocks and prose.	**In-line suggestions** that you can accept or ignore with a single keystroke (e.g., Tab key).

Task Description#3

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

Expected Output#3

● Detailed explanation with code snippet

Python

```python
import math

def get_area_formulas():
    """Maps shape names to a tuple of (function, required_args)."""
    return {
        'circle': (lambda r: math.pi * r**2, {'radius'}),
        'rectangle': (lambda l, w: l * w, {'length', 'width'}),
        'triangle': (lambda b, h: 0.5 * b * h, {'base', 'height'}),
    }

def calculate_area_v2(shape, **kwargs):
    """
    Calculates the area of various shapes using a dictionary of functions.

    Args:
        shape (str): The name of the shape.
        **kwargs: A dictionary of keyword arguments for the dimensions.

    Returns:
        float: The calculated area.
        str: An error message.
    """
    formulas = get_area_formulas()

    if shape not in formulas:
        return "Error: Invalid shape provided."

    formula_func, required_args = formulas[shape]

    # Check if all required arguments are present
    if not required_args.issubset(kwargs.keys()):
        missing_args = required_args.difference(kwargs.keys())
        return f"Error: A {shape} requires the following arguments: {', '.join(mis

    # Call the function with the required arguments
    # The `**` operator unpacks the dictionary into named arguments
    try:
        return formula_func(**{arg: kwargs[arg] for arg in required_args})
    except TypeError:
        return "Error: Invalid arguments provided."
```

Explanation:

Explanation

This version improves on the original by using a dictionary to store the logic for each shape. This makes the code **more scalable** and easier to manage if you need to add new shapes in the future.

- `get_area_formulas()` : This function returns a dictionary where each key is a shape name (e.g., `'circle'`). The value for each key is a tuple containing:
  1. A **lambda function** that calculates the area. Lambda functions are small, anonymous functions that are perfect for this simple calculation.
  2. A **set of strings** representing the required arguments for that shape.
- `if shape not in formulas:` : This checks if the user's requested shape is a valid key in our `formulas` dictionary. If not, it returns an error. This is a cleaner way to handle invalid shapes than a final `else` statement.
- `formula_func, required_args = formulas[shape]` : This line unpacks the tuple stored in the dictionary, assigning the lambda function to `formula_func` and the set of required arguments to `required_args` .
- `if not required_args.issubset(kwargs.keys()):` : This is a powerful and concise way to **validate the input**. It checks if the set of `required_args` is a subset of the keys provided in `kwargs` . If any required arguments are missing, it calculates which ones and returns a detailed error message.
- `return formula_func(**{arg: kwargs[arg] for arg in required_args})` : This is the core of the function. It uses a **dictionary comprehension** to create a new dictionary containing only the required arguments and their values from `kwargs` . The `**` operator then **unpacks** this new dictionary, passing its key-value pairs as named arguments to the lambda function, which then calculates and returns the final area. This is a highly flexible and powerful pattern in Python.

Task Description#4

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

Expected 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 empty list
    print(f"Sum of squares of empty list: {sum_of_squares([]

    # 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
```

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

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:

The provided Python code defines a function `sum_of_squares` that calculates the sum of the squares of all numbers in a given list. It also includes examples of how to use and test the function.

`sum_of_squares(numbers)` **Function**

This function takes a single 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 function immediately returns `0` because the sum of squares of an empty set of numbers is zero.
- `return sum(num ** 2 for num in numbers)` : This is the core of the function and uses a **generator expression** within the built-in `sum()` function.
    - `num ** 2` : For each number (`num`) in the `numbers` list, this expression calculates its square.
    - `for num in numbers` : This part iterates through each item in the input list.
    - `sum(...)` : The `sum()` function then efficiently adds up all the squared values generated by the expression, and the total is returned as the function's output.

Example Usage and Testing

The code block under `if __name__ == "__main__":` demonstrates how to use and test the function. This is a common Python practice to ensure that the code inside this block only runs when the script is executed directly (not when it's imported as a module).

- `test_numbers = [1, 2, 3, 4, 5]` : A list of integers is created and passed to the function, and the result is printed.
- `sum_of_squares([])` : This line tests the **empty list** edge case, which should return `0` .
- `decimal_numbers = [1.5, 2.5, 3.5]` : This test demonstrates that the function correctly handles lists containing **floating-point numbers**. The output will be a float.

Task Description#5

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

Expected Output#5

● Refactored code written by student with improved logic

Python

```python
def calculate_even_odd_sums(numbers):
    """
    Calculates the sum of even and odd numbers in a list.

    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 = 0
    odd_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num
    return even_sum, odd_sum

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

Key Improvements and Observations

- **Single Pass Efficiency**: The refactored code iterates through the list only once. A student's initial code might have two separate loops—one for even numbers and one for odd numbers. This single-pass approach is **more efficient** as it avoids redundant processing.
- **Clear Variable Naming**: The variable names `even_sum` and `odd_sum` are descriptive, making the code's purpose easy to understand at a glance.
- **Using a Single Function**: Returning both values from a single function is a cleaner design than having two separate functions (e.g., `sum_evens()` and `sum_odds()`). Python can easily return multiple values in a **tuple**, which can be unpacked directly into separate variables for convenience.
- **Modulus Operator (`%`)**: The logic `num % 2 == 0` is the standard and most reliable way to check if a number is even. The modulo operator gives the remainder of a division. If a number divided by 2 has a remainder of 0, it's even; otherwise, it's odd.
- **No Redundant Logic**: The `if/else` statement handles both conditions (`even` and `odd`) for each number in a single block, avoiding duplicated code.