

# AI-ASSISTED CODING

## LAB ASSIGNMENT-2.3

Roll no:2403A510C6

Name: Nadhiya Seelam

Batch: 05

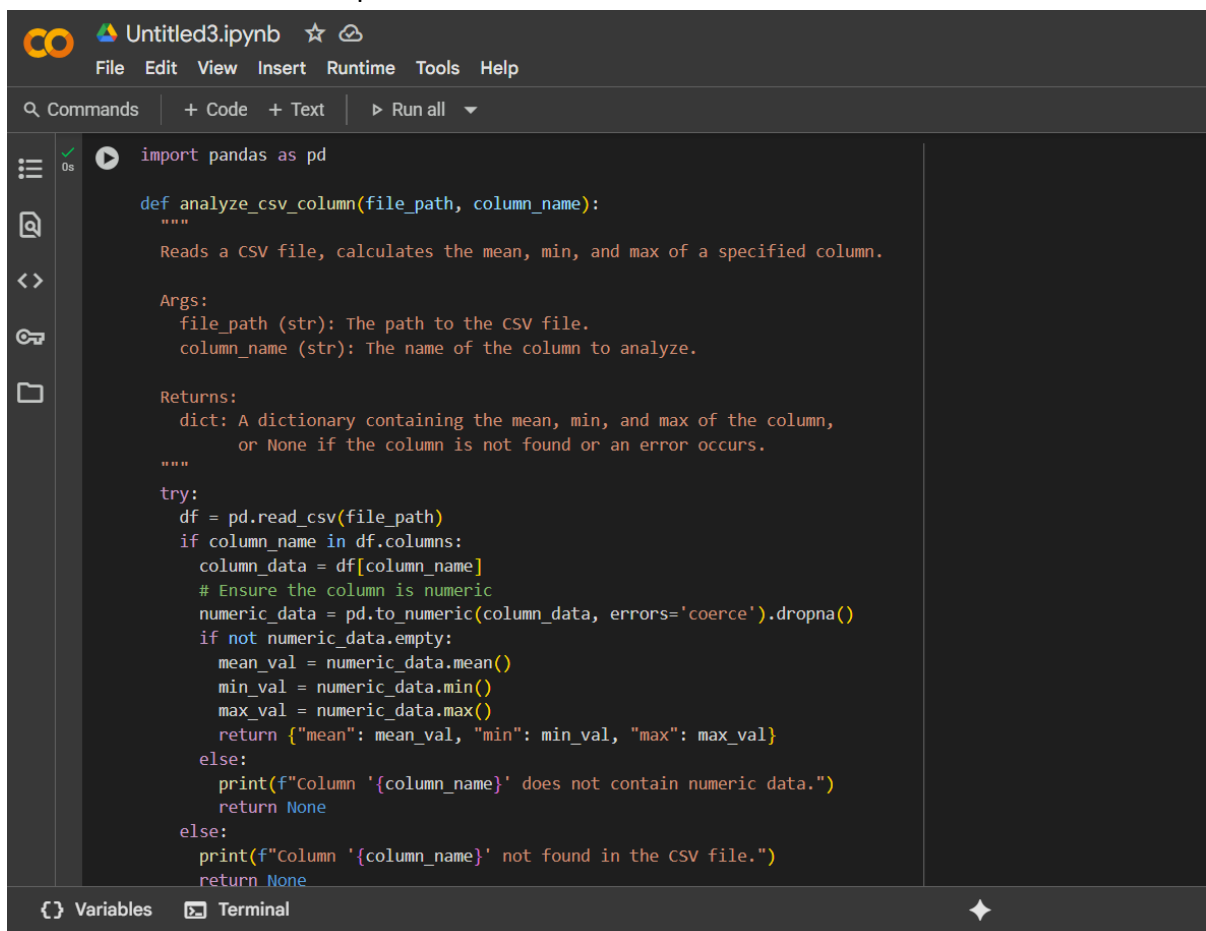
CSE 2<sup>nd</sup> year

### 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

The image shows a Google Colab notebook interface. At the top, it says 'Untitled3.ipynb' with a star icon and a share icon. Below the title bar is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Underneath the menu bar is a toolbar with 'Commands', '+ Code', '+ Text', and a 'Run all' button with a dropdown arrow. The main area of the notebook contains a Python code cell. The code defines a function 'analyze\_csv\_column' that takes 'file\_path' and 'column\_name' as arguments. It reads a CSV file, calculates the mean, min, and max of the specified column, and returns a dictionary with these values. If the column is not found or an error occurs, it returns None. The code is as follows:

```
import pandas as pd

def analyze_csv_column(file_path, column_name):
    """
    Reads a CSV file, calculates the mean, min, and max of a specified column.

    Args:
        file_path (str): The path to the CSV file.
        column_name (str): The name of the column to analyze.

    Returns:
        dict: A dictionary containing the mean, min, and max of the column,
        or None if the column is not found or an error occurs.
    """
    try:
        df = pd.read_csv(file_path)
        if column_name in df.columns:
            column_data = df[column_name]
            # Ensure the column is numeric
            numeric_data = pd.to_numeric(column_data, errors='coerce').dropna()
            if not numeric_data.empty:
                mean_val = numeric_data.mean()
                min_val = numeric_data.min()
                max_val = numeric_data.max()
                return {"mean": mean_val, "min": min_val, "max": max_val}
            else:
                print(f"Column '{column_name}' does not contain numeric data.")
                return None
        else:
            print(f"Column '{column_name}' not found in the CSV file.")
            return None
```

At the bottom of the notebook, there are tabs for 'Variables' and 'Terminal', and a star icon on the right.

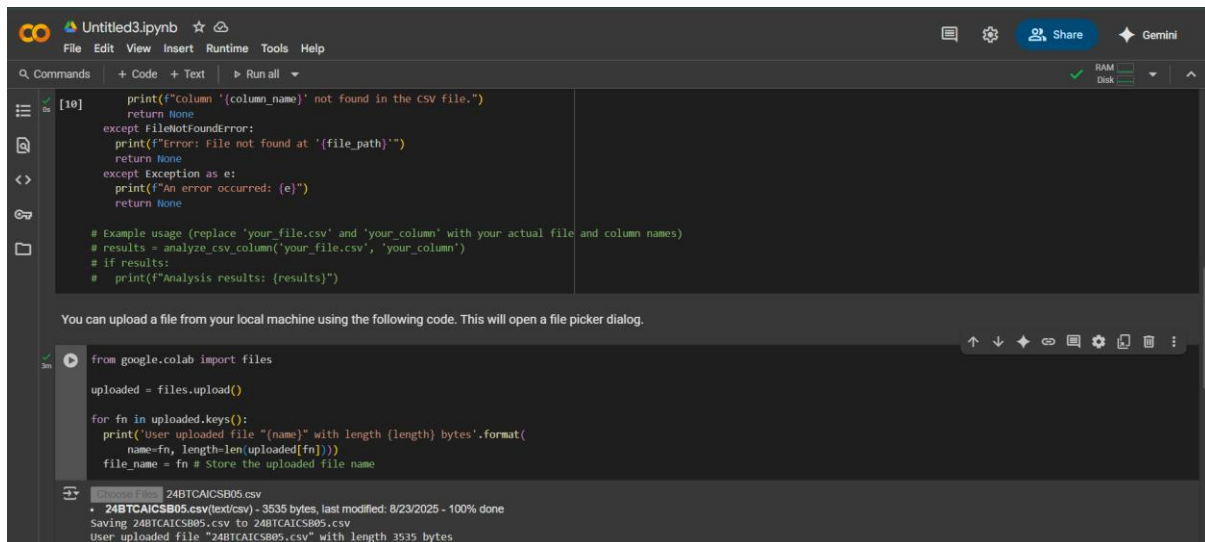
### Explanation:

Here's a breakdown of what it does:

- **import pandas as pd:** This line imports the pandas library, which is used for data manipulation and analysis, and gives it the alias pd.

- **def analyze\_csv\_column(file\_path, column\_name)::** This defines the function named `analyze_csv_column` that takes two arguments: `file_path` (the path to the CSV file) and `column_name` (the name of the column to analyze).
- **Docstring:** The text within triple quotes explains what the function does, its arguments, and what it returns.
- **try...except block:** This block is used for error handling. It attempts to execute the code within the try block and catches potential errors (like `FileNotFoundError` if the file doesn't exist or other exceptions) in the except blocks.
- **df = pd.read\_csv(file\_path):** This line reads the CSV file specified by `file_path` into a pandas DataFrame called `df`.
- **if column\_name in df.columns::** This checks if the `column_name` provided by the user exists in the DataFrame's columns.
- **column\_data = df[column\_name]:** If the column exists, this line selects the data from that column.
- **numeric\_data = pd.to\_numeric(column\_data, errors='coerce').dropna():** This is a crucial step. It attempts to convert the data in the selected column to numeric type. `errors='coerce'` will turn any values that cannot be converted into numbers into `NaN` (Not a Number), and `.dropna()` removes these `NaN` values. This ensures that calculations are only performed on valid numbers.
- **if not numeric\_data.empty::** This checks if there is any valid numeric data left after the conversion and dropping of non-numeric values.
- **mean\_val = numeric\_data.mean(), min\_val = numeric\_data.min(), max\_val = numeric\_data.max():** These lines calculate the mean, minimum, and maximum of the numeric data in the column using built-in pandas methods.
- **return {"mean": mean\_val, "min": min\_val, "max": max\_val}:** If calculations are successful, the function returns a dictionary containing the calculated mean, min, and max values.
- **else blocks and print statements:** These handle cases where the column is not found, the column does not contain numeric data, or an error occurs during file processing, printing informative messages to the user and returning `None`.
- **Example Usage (commented out):** The lines at the end show how to call the function and print the results.

In summary, this function provides a robust way to read a CSV, specifically target a column, handle non-numeric data within that column, and calculate basic descriptive statistics (mean, min, max) if valid numeric data is present.



The screenshot shows a Google Colab notebook titled 'Untitled3.ipynb'. The top bar includes the Colab logo, file name, and icons for settings, share, and Gemini. Below the top bar is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A search bar labeled 'Q. Commands' is on the left. The main code area contains two code cells. The first cell, labeled '[10]', contains Python code for file analysis with error handling. The second cell contains code for uploading a file from the local machine. Below the code cells, a message states: 'You can upload a file from your local machine using the following code. This will open a file picker dialog.' Below this message is a code block for file upload. At the bottom, a file upload progress bar shows '24BTCaICS805.csv' being uploaded, with details: '24BTCaICS805.csv(text/csv) - 3535 bytes, last modified: 8/23/2025 - 100% done'. Below the progress bar, it says 'Saving 24BTCaICS805.csv to 24BTCaICS805.csv' and 'User uploaded file "24BTCaICS805.csv" with length 3535 bytes'.

```
print(f"Column '{column_name}' not found in the csv file.")
return None
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' and 'your_column' with your actual file and column names)
# results = analyze_csv_column('your_file.csv', 'your_column')
# if results:
#     print(f"Analysis results: {results}")
```

```
from google.colab import files
uploaded = files.upload()

for fn in uploaded.keys():
    print('User uploaded file "{name}" with length {length} bytes'.format(
        name=fn, length=len(uploaded[fn])))
    file_name = fn # Store the uploaded file name
```

You can upload a file from your local machine using the following code. This will open a file picker dialog.

```
from google.colab import files
uploaded = files.upload()

for fn in uploaded.keys():
    print('User uploaded file "{name}" with length {length} bytes'.format(
        name=fn, length=len(uploaded[fn])))
    file_name = fn # Store the uploaded file name
```

24BTCaICS805.csv

• 24BTCaICS805.csv(text/csv) - 3535 bytes, last modified: 8/23/2025 - 100% done

Saving 24BTCaICS805.csv to 24BTCaICS805.csv

User uploaded file "24BTCaICS805.csv" with length 3535 bytes

## Explanation:

Here's a breakdown:

- **from google.colab import files:** This line imports the files object from the google.colab library, which provides utilities for working with files in Colab.
- **uploaded = files.upload():** This is the main command that triggers the file upload process. When you run this line, a file picker dialog will appear in your browser, allowing you to select one or more files from your local machine. Once you've selected and confirmed the files, they are uploaded to the Colab runtime's temporary storage. The files.upload() function returns a dictionary where the keys are the filenames and the values are the file contents as bytes.
- **for fn in uploaded.keys():** This loop iterates through the keys of the uploaded dictionary, which are the names of the files that were uploaded.
- **print('User uploaded file "{name}" with length {length} bytes'.format(name=fn, length=len(uploaded[fn]))):** Inside the loop, this line prints a confirmation message for each uploaded file, showing its name and size in bytes.
- **file\_name = fn # Store the uploaded file name:** This line assigns the name of the *last* uploaded file to the variable file\_name. This is useful if you only expect to upload one file and want to easily reference its name later in your code.

In short, this code provides a simple way to get files from your computer into your Colab notebook so you can work with them.

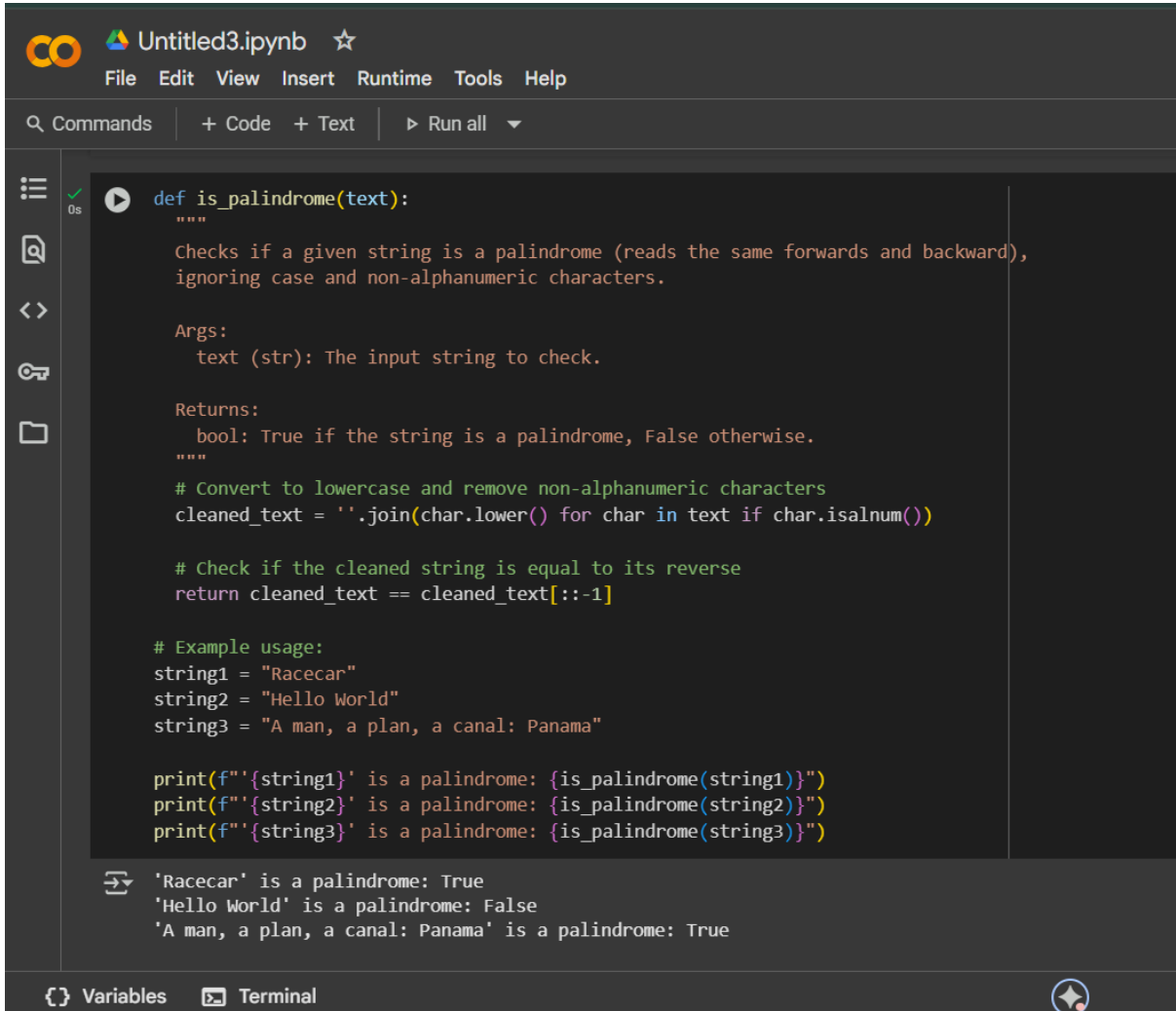
## TASK DESCRIPTION#2

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

## Expected Output#2

- Side-by-side comparison and observations

## Gemini Code and Output:



The screenshot shows a Jupyter Notebook titled 'Untitled3.ipynb'. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a search bar, and buttons for '+ Code', '+ Text', and 'Run all'. The left sidebar contains icons for file management and execution. The main area displays a Python function `def is_palindrome(text):` with a docstring explaining its purpose: 'Checks if a given string is a palindrome (reads the same forwards and backward), ignoring case and non-alphanumeric characters.' The function's arguments are defined as `text (str): The input string to check.` and its return value is a `bool: True if the string is a palindrome, False otherwise.` The function's logic involves converting the text to lowercase and removing non-alphanumeric characters using `cleaned_text = ''.join(char.lower() for char in text if char.isalnum())`, then checking if the cleaned text is equal to its reverse using `return cleaned_text == cleaned_text[::-1]`. Example usage is provided with three strings: 'Racecar', 'Hello World', and 'A man, a plan, a canal: Panama'. The output at the bottom shows the function's results: 'Racecar' is a palindrome: True, 'Hello World' is a palindrome: False, and 'A man, a plan, a canal: Panama' is a palindrome: True.

```
def is_palindrome(text):
    """
    Checks if a given string is a palindrome (reads the same forwards and backward),
    ignoring case and non-alphanumeric characters.

    Args:
        text (str): The input string to check.

    Returns:
        bool: True if the string is a palindrome, False otherwise.
    """
    # Convert to lowercase and remove non-alphanumeric characters
    cleaned_text = ''.join(char.lower() for char in text if char.isalnum())

    # Check if the cleaned string is equal to its reverse
    return cleaned_text == cleaned_text[::-1]

# Example usage:
string1 = "Racecar"
string2 = "Hello World"
string3 = "A man, a plan, a canal: Panama"

print(f'{string1} is a palindrome: {is_palindrome(string1)}')
print(f'{string2} is a palindrome: {is_palindrome(string2)}')
print(f'{string3} is a palindrome: {is_palindrome(string3)}')
```

```
'Racecar' is a palindrome: True
'Hello World' is a palindrome: False
'A man, a plan, a canal: Panama' is a palindrome: True
```

## Explanation:

This function checks if a given string is a palindrome, ignoring case and non-alphanumeric characters. Here's how it works:


- **def is\_palindrome(text)::** This line defines the function named `is_palindrome` that takes one argument, `text`, which is the string you want to check.
- **Docstring:** The text within the triple quotes explains the purpose of the function, its arguments, and what it returns.
- **cleaned\_text = ''.join(char.lower() for char in text if char.isalnum()):** This is the core of the cleaning process:
  - **for char in text:** It iterates through each character in the input text.
  - **if char.isalnum():** It checks if the current character is alphanumeric (a letter or a number). If it's not alphanumeric (like spaces, punctuation, etc.), it's skipped.

- **char.lower()**: If the character is alphanumeric, it's converted to lowercase. This makes the check case-insensitive (so "Racecar" and "racecar" are treated the same).
  - **".join(...)**: Finally, all the processed characters are joined back together into a new string called `cleaned_text`.
- **return cleaned\_text == cleaned\_text[::-1]**: This line checks if the `cleaned_text` is equal to its reverse.
  - **cleaned\_text[::-1]**: This is a Python slicing trick that creates a reversed copy of the `cleaned_text` string.
  - **==**: This compares the original `cleaned_text` with its reversed version.
  - **return**: The function returns `True` if they are the same (meaning it's a palindrome) and `False` otherwise.
- **Example Usage**: The lines after the function definition show how to call the function with different strings and print the results to the console.

In essence, the function cleans the input string by removing irrelevant characters and making it lowercase, and then it simply checks if the cleaned string reads the same forwards and backward.

#### **Copilot Code:**

Python ^


 Copy

```
def is_palindrome(s):
    # Remove spaces and convert to lowercase for uniformity
    cleaned = ''.join(c.lower() for c in s if c.isalnum())
    return cleaned == cleaned[::-1]

# Example usage
test_strings = ["Madam", "Racecar", "Hello", "12321", "Was it a car or a cat"]

for text in test_strings:
    result = is_palindrome(text)
    print(f"'{text}' is a palindrome? → {result}")
```

Python ^

 Copy


```
def is_palindrome(s):
    # Remove spaces and convert to lowercase for uniformity
    cleaned = ''.join(c.lower() for c in s if c.isalnum())
    return cleaned == cleaned[::-1]

# Example usage
test_strings = ["Madam", "Racecar", "Hello", "12321", "Was it a car or a cat I saw"]

for text in test_strings:
    result = is_palindrome(text)
    print(f"'{text}' is a palindrome? → {result}")
```

Copilot Output:

Code ^

 Copy

```
'Madam' is a palindrome? → True
'Racecar' is a palindrome? → True
'Hello' is a palindrome? → False
'12321' is a palindrome? → True
'Was it a car or a cat I saw' is a palindrome? → True
```

**Explanation:**

- ❓ `c.lower()`: Converts each character to lowercase so that "Madam" and "madam" are treated the same.
- ❓ `if c.isalnum()`: Keeps only letters and numbers, ignoring spaces, punctuation, etc.
- ❓ `"".join(...)`: Combines the cleaned characters back into a single string.

### 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

Code and Output:

```
import math

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

    Args:
        shape (str): The name of the shape (e.g., 'circle', 'rectangle', 'triangle').
        **kwargs: Keyword arguments for the dimensions of the shape.

    Returns:
        float: The area of the shape, or None if the shape is not supported or dimensions are missing.
    """
    if shape.lower() == 'circle':
        radius = kwargs.get('radius')
        if radius is not None:
            return math.pi * (radius ** 2)
        else:
            print("Error: Radius is required for a circle.")
            return None
    elif shape.lower() == 'rectangle':
        length = kwargs.get('length')
        width = kwargs.get('width')
        if length is not None and width is not None:
            return length * width
        else:
            print("Error: Length and width are required for a rectangle.")
            return None
    elif shape.lower() == 'triangle':
        base = kwargs.get('base')
        height = kwargs.get('height')
        if base is not None and height is not None:
            return 0.5 * base * height
        else:
            print("Error: Base and height are required for a triangle.")
            return None
    else:
        print(f"Error: Shape '{shape}' is not supported.")
        return None

# Example usage:
circle_area = calculate_area('circle', radius=5)
rectangle_area = calculate_area('rectangle', length=4, width=6)
triangle_area = calculate_area('triangle', base=3, height=7)

print(f"Area of circle with radius 5: {circle_area}")
print(f"Area of rectangle with length 4 and width 6: {rectangle_area}")
print(f"Area of triangle with base 3 and height 7: {triangle_area}")
```

Area of circle with radius 5: 78.53981633974483  
Area of rectangle with length 4 and width 6: 24  
Area of triangle with base 3 and height 7: 10.5

## Explanation:

Here is a line-by-line explanation of the `calculate_area` function:

- **import math:** This line imports the `math` module, which is necessary to use `math.pi` for circle area calculations.
- **def calculate\_area(shape, \*\*kwargs):** This defines the function named `calculate_area`.



- `shape` is a string argument specifying the type of shape (e.g., 'circle', 'rectangle').
  - `**kwargs` allows you to pass variable keyword arguments (like `radius=5`, `length=4`, `width=6`, etc.) for the dimensions.
- **Docstring:** The text in triple quotes explains the function's purpose, arguments, and return value.
- **`if shape.lower() == 'circle':`** This checks if the shape is 'circle' (case-insensitive).
- **`radius = kwargs.get('radius')`:** It safely retrieves the 'radius' value from the keyword arguments.
- **`if radius is not None:`** Checks if a radius was provided.
- **`return math.pi * (radius ** 2)`:** Calculates and returns the circle's area.
- **`else:`** If no radius was given for a circle:
- **`print("Error: Radius is required for a circle.")`:** Prints an error message.
- **`return None`:** Returns None.
- **`elif shape.lower() == 'rectangle':`** Checks if the shape is 'rectangle'.
- **`length = kwargs.get('length')` and **`width = kwargs.get('width')`:** Safely retrieve 'length' and 'width'.**
- **`if length is not None and width is not None:`** Checks if both length and width were provided.
- **`return length * width`:** Calculates and returns the rectangle's area.
- **`else:`** If length or width is missing:
- **`print("Error: Length and width are required for a rectangle.")`:** Prints an error.
- **`return None`:** Returns None.
- **`elif shape.lower() == 'triangle':`** Checks if the shape is 'triangle'.
- **`base = kwargs.get('base')` and **`height = kwargs.get('height')`:** Safely retrieve 'base' and 'height'.**
- **`if base is not None and height is not None:`** Checks if both base and height were provided.
- **`return 0.5 * base * height`:** Calculates and returns the triangle's area.
- **`else:`** If base or height is missing:
- **`print("Error: Base and height are required for a triangle.")`:** Prints an error.
- **`return None`:** Returns None.
- **`else:`** If the shape is not supported:
- **`print(f"Error: Shape '{shape}' is not supported.")`:** Prints an error message with the unsupported shape name.
- **`return None`:** Returns None.
- **Example Usage:** The lines outside the function show how to call the function with different shapes and dimensions and print the results.

#### 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

Code:

```
1 def sum_of_squares_builtin(numbers):
2     """
3     Calculate sum of squares using built-in sum() and list comprehension.
4
5     Args:
6         numbers (list): List of numbers
7
8     Returns:
9         float: Sum of squares of all numbers
10    """
11    return sum(x**2 for x in numbers)
12
13 def sum_of_squares_loop(numbers):
14     """
15     Calculate sum of squares using a traditional for loop.
16
17     Args:
18         numbers (list): List of numbers
19
20     Returns:
21         float: Sum of squares of all numbers
22    """
23    total = 0
24    for num in numbers:
25        total += num ** 2
26    return total
27
28 def sum_of_squares_map(numbers):
29     """
30     Calculate sum of squares using map() function.
31
32     Args:
33         numbers (list): List of numbers
```

```
27
28 def sum_of_squares_map(numbers):
29     """
30     Calculate sum of squares using map() function.
31
32     Args:
33         numbers (list): List of numbers
34
35     Returns:
36         float: Sum of squares of all numbers
37     """
38     return sum(map(lambda x: x**2, numbers))
39
40 def sum_of_squares_range(n):
41     """
42     Calculate sum of squares from 1 to n.
43
44     Args:
45         n (int): Upper limit
46
47     Returns:
48         int: Sum of squares from 1^2 to n^2
49     """
50     return sum(i**2 for i in range(1, n + 1))
51
52 def sum_of_squares_formula(n):
53     """ Calculate sum of squares from 1 to n using mathematical formula. TAB to jump here
54     Formula: n(n+1)(2n+1)/6
55
56     Args:
57         n (int): Upper limit
58
59     Returns:
```

```
File Edit Selection View Go Run Terminal Help
sum_of_squares.py - cursor - Cursor

sum_of_squares.py x
sum_of_squares.py > sum_of_squares_range
50 return sum(i**2 for i in range(1, n + 1))
51
52 def sum_of_squares_formula(n):
53     """ Calculate sum of squares from 1 to n using mathematical formula.
54     Formula: n(n+1)(2n+1)/6
55
56     Args:
57         n (int): Upper limit
58
59     Returns:
60         int: Sum of squares from 1² to n²
61     """
62     return n * (n + 1) * (2 * n + 1) // 6
63
64 # Example usage and testing
65 if __name__ == "__main__":
66     # Test with a list of numbers
67     test_numbers = [1, 2, 3, 4, 5]
68     print("Test numbers:", test_numbers)
69
70     print("\n--- Different Methods ---")
71     print(f"Using built-in sum(): {sum_of_squares_builtin(test_numbers)}")
72     print(f"Using for loop: {sum_of_squares_loop(test_numbers)}")
73     print(f"Using map(): {sum_of_squares_map(test_numbers)}")
74
75     # Test with range
76     n = 5
77     print(f"\n--- Sum of squares from 1 to {n} ---")
78     print(f"Using range: {sum_of_squares_range(n)}")
79     print(f"Using formula: {sum_of_squares_formula(n)}")
80
81     # Interactive example
82     print("\n--- Interactive Example ---")
83     ...
```

```
File Edit Selection View Go Run Terminal Help
sum_of_squares.py - cursor - Cursor

sum_of_squares.py x
sum_of_squares.py > sum_of_squares_range
81 # Interactive example
82 print("\n--- Interactive Example ---")
83 try:
84     user_input = input("Enter numbers separated by spaces (e.g., 1 2 3 4 5): ")
85     user_numbers = [float(x) for x in user_input.split()]
86     result = sum_of_squares_builtin(user_numbers)
87     print(f"Sum of squares: {result}")
88 except ValueError:
89     print("Invalid input. Please enter valid numbers separated by spaces.")
90 except KeyboardInterrupt:
91     print("\nProgram interrupted by user.")
92
93 # Performance comparison
94 print("\n--- Performance Comparison ---")
95 import time
96
97 large_list = list(range(1, 10001))
98
99 # Test built-in method
100 start_time = time.time()
101 result1 = sum_of_squares_builtin(large_list)
102 time1 = time.time() - start_time
103
104 # Test loop method
105 start_time = time.time()
106 result2 = sum_of_squares_loop(large_list)
107 time2 = time.time() - start_time
108
109 # Test map method
110 start_time = time.time()
111 result3 = sum_of_squares_map(large_list)
112 time3 = time.time() - start_time
113
```

```
sum_of_squares.py x
sum_of_squares.py > sum_of_squares_range
95 import time
96
97 large_list = list(range(1, 10001))
98
99 # Test built-in method
100 start_time = time.time()
101 result1 = sum_of_squares_builtin(large_list)
102 time1 = time.time() - start_time
103
104 # Test loop method
105 start_time = time.time()
106 result2 = sum_of_squares_loop(large_list)
107 time2 = time.time() - start_time
108
109 # Test map method
110 start_time = time.time()
111 result3 = sum_of_squares_map(large_list)
112 time3 = time.time() - start_time
113
114 print(f"Built-in method: {time1:.6f} seconds")
115 print(f"Loop method: {time2:.6f} seconds")
116 print(f"Map method: {time3:.6f} seconds")
117 print(f"All results match: {result1 == result2 == result3}")
118
```

Output:

```
Test numbers: [1, 2, 3, 4, 5]

--- Different Methods ---
Using built-in sum(): 55
Using for loop: 55
Using map(): 55

--- Sum of squares from 1 to 5 ---
Using range: 55
Using formula: 55

--- Interactive Example ---
Enter numbers separated by spaces (e.g., 1 2 3 4 5): 1 3
Sum of squares: 10.0

--- Performance Comparison ---
Built-in method: 0.001897 seconds
Loop method: 0.001707 seconds
Map method: 0.002927 seconds
All results match: True

Ctrl+K to generate a command
```

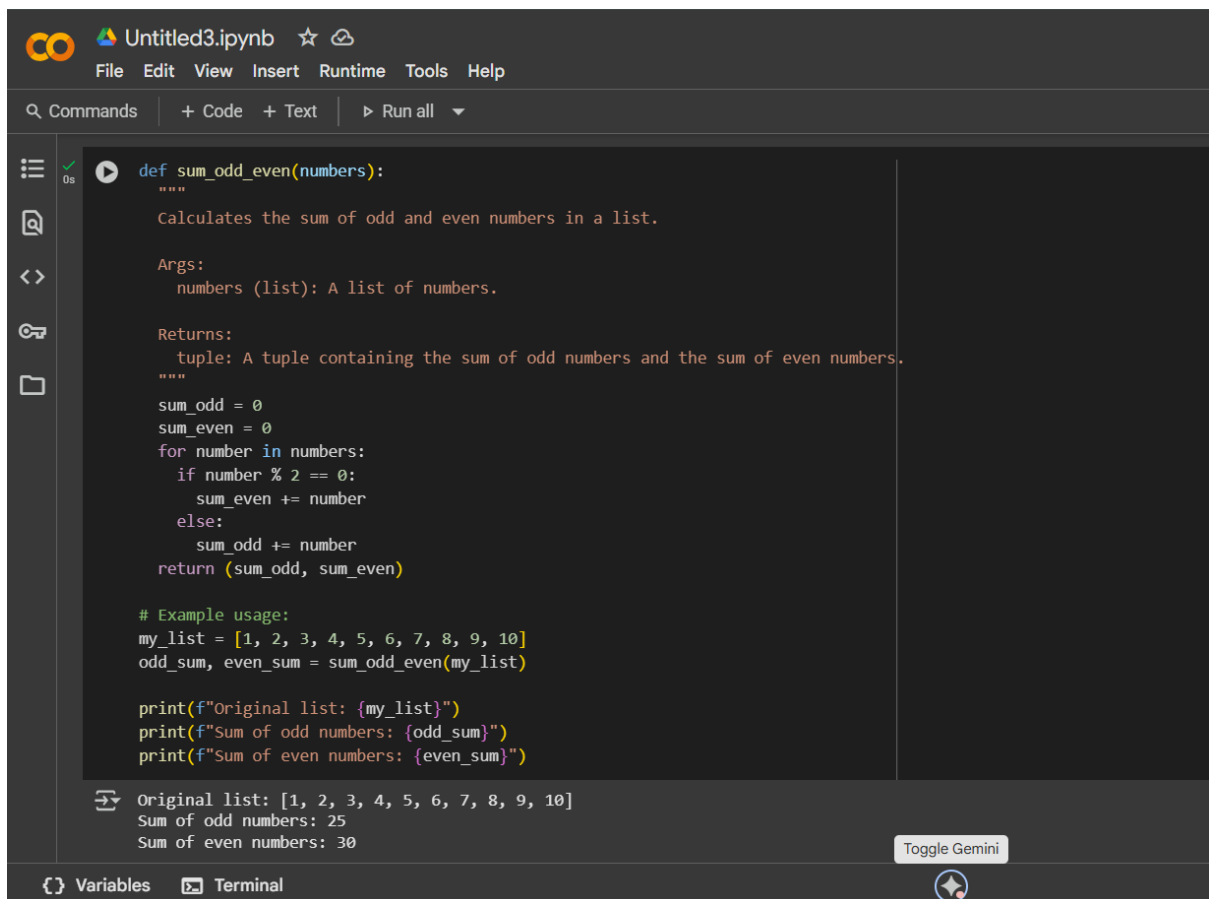
## TASK DESCRIPTION#5

- 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.

**Code and Output:**



The screenshot shows a Jupyter Notebook titled 'Untitled3.ipynb'. The interface includes a top menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. The main area contains a Python code cell with the following content:

```
def sum_odd_even(numbers):  
    """  
    Calculates the sum of odd and even numbers in a list.  
  
    Args:  
        numbers (list): A list of numbers.  
  
    Returns:  
        tuple: A tuple containing the sum of odd numbers and the sum of even numbers.  
    """  
    sum_odd = 0  
    sum_even = 0  
    for number in numbers:  
        if number % 2 == 0:  
            sum_even += number  
        else:  
            sum_odd += number  
    return (sum_odd, sum_even)  
  
# Example usage:  
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
odd_sum, even_sum = sum_odd_even(my_list)  
  
print(f"Original list: {my_list}")  
print(f"Sum of odd numbers: {odd_sum}")  
print(f"Sum of even numbers: {even_sum}")
```

Below the code cell, the output is displayed:

```
Original list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
Sum of odd numbers: 25  
Sum of even numbers: 30
```

At the bottom of the interface, there are tabs for 'Variables' and 'Terminal', and a 'Toggle Gemini' button.

## Explanation:

Here's a line-by-line explanation:

- **def sum\_odd\_even(numbers):**: This line defines the function named `sum_odd_even` that takes one argument, `numbers`, which is expected to be a list of numbers.
- **Docstring**: The text within triple quotes explains what the function does, its arguments, and what it returns.
- **sum\_odd = 0**: This initializes a variable `sum_odd` to zero. This variable will store the cumulative sum of the odd numbers found in the list.
- **sum\_even = 0**: This initializes a variable `sum_even` to zero. This variable will store the cumulative sum of the even numbers found in the list.
- **for number in numbers:**: This starts a loop that will iterate through each element in the input list `numbers`. In each iteration, the current element is assigned to the variable `number`.
- **if number % 2 == 0:**: Inside the loop, this line checks if the current number is even. The modulo operator (%) calculates the remainder when `number` is divided by 2. If the remainder is 0, the number is even.

- **sum\_even += number:** If the condition in the if statement is true (the number is even), this line adds the current number to the sum\_even variable. The += is a shorthand for sum\_even = sum\_even + number.
- **else::** If the condition in the if statement is false (the number is not even, meaning it's odd):
- **sum\_odd += number:** This line adds the current number to the sum\_odd variable.
- **return (sum\_odd, sum\_even):** After the loop has finished iterating through all the numbers in the list, this line returns a tuple containing two values: the final sum\_odd and the final sum\_even.
- **Example Usage:** The lines following the function definition demonstrate how to create a list (my\_list), call the sum\_odd\_even function with this list, and then print the original list and the calculated sums of odd and even numbers.

In essence, the function goes through the list number by number, checks if each number is odd or even, and adds it to the corresponding sum variable. Finally, it provides the total sums for both odd and even numbers.