ASSIGNMENT-2.3

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

Batch No: 05 Course: Al Assisted Coding

Task Description#1

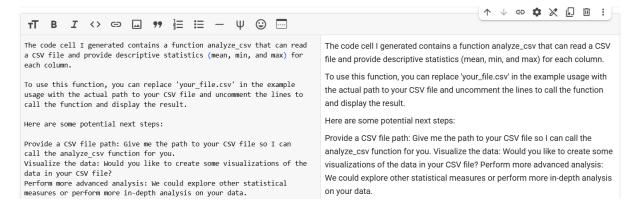
• Use Google Gemini in Colab to write a function that reads a CSV file and calculates mean, min, max.

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.
   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:



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

Expected Output#2

• Side-by-side comparison and observations

Gemini's code:



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.



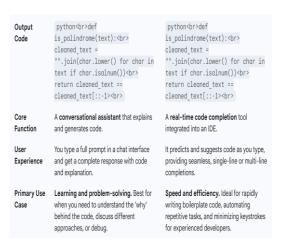
Simple Word:

```
# 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:



Provides detailed explanations of the Does not provide explanations. It Explanation code's logic, discussing how it handles assumes the developer understands the case sensitivity, punctuation, and code it generates. efficiency. Output Delivers code as part of a more extensive, In-line suggestions that you can accept structured text response, often with or ignore with a single keystroke (e.g., Tab **Format** code blocks and prose. key).

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

Expected Output#3

• Detailed explanation with code snippet

```
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'}),
        'retcangle': (lambda l, w: l * w, {'tength', 'width'}),
        'triangle': (lambda b, h: 8.5 * 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 "Frror: A {shape} requires the following arguments: {', '.join(missing_args = required_args.difference(kwargs.keys())
    return "Frror: A {shape} required arguments
# 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:
 - 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 shope not in formulos: :This checks if the user's requested shape is a valid key in our
 formulos dictionary. If not, it returns an error. This is a cleaner way to handle invalid shapes
 than a final lease statement.
- formula_func, required_orgs = formulas[shope]: This line unpacks the tuple stored in the dictionary, assigning the lambda function to formula_func and the set of required arguments to required_orgs.
- If not required_orgs.issubset(kworgs.keys()): This is a powerful and concise way to
 validate the input. It checks if the set of required_orgs is a subset of the keys provided
 in kworgs. If any required arguments are missing, it calculates which ones and returns a
 detailed error message.
- return formula_func(**{org: kworgs{org} for org in required_orgs}): 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 kworgs. 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.

• 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.

```
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([])
```

```
# 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 <code>sum_of_squores</code> 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 @
 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 <code>if __name__ == "__main__":</code> 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.

• 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

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.