ASSIGNMENT-4.3

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

Batch No: 05 Course: AI Assisted Coding

Task Description#1

• Zero-shot: Prompt AI to write a function that checks whether a given year is a leap year.

Expected Output#1

• AI-generated function with no examples provided

```
def is_leap_year(year):
    """
    Checks if a given year is a leap year.

    A year is a leap year if it is divisible by 4,
    except for end-of-century years which must be
    divisible by 400.

    Args:
        year (int): The year to check.

    Returns:
        bool: True if the year is a leap year, False otherwise.
    """
    if (year % 4) == 0:
        if (year % 100) == 0:
            if (year % 400) == 0:
                return True
            else:
                return False
        else:
            return True
    else:
        return False
```

Explanation:

The function, is_leap_year(year), uses a series of nested if/else statements to check the conditions in a specific order.

1. **First Check: Is it divisible by 4?**
    - The outermost if (year % 4) == 0: checks if the year is evenly divisible by 4.
    - Most of the time, this is the only check needed. If a year is divisible by 4 (and not a century year), like 2024, the code will jump to the else statement within the first if and return True. If it's not divisible by 4, like 2023, the code immediately goes to the final else statement and returns False.
2. **Second Check: Is it a century year?**
    - If the first check passes (the year is divisible by 4), the code then checks if it's a century year with if (year % 100) == 0: Century years are years like 1900, 2000, and 2100.
3. **Third Check: Is it a century leap year?**

- If a year is both divisible by 4 AND divisible by 100, the code performs the final check: if (year % 400) == 0:
- Only century years that are also divisible by 400 are leap years (e.g., 2000). If a century year is not divisible by 400 (e.g., 1900), the function returns False.

This nested structure ensures that all three rules are applied correctly and in the right order to get an accurate result. The example usage you see at the bottom of the code demonstrates each of these cases.

Task Description#2

• One-shot: Give one input-output example to guide AI in writing a function that converts centimeters to inches.

Expected Output#2

• Function with correct conversion logic

```
makefile

Input:  10
Output: 3.937
```

(Because 1 cm = 0.3937 inches)

Python Function

```python
def cm_to_inches(cm):
    return cm * 0.3937

# Example usage
print(cm_to_inches(10))  # Output: 3.937
```

Explanation:

Function Definition

def cm_to_inches(cm):

- def starts a function definition in Python.
- cm_to_inches is the function's name.
- (cm) is the parameter that takes the length in centimeters.

Conversion Logic

return cm * 0.3937

- 1 centimeter = 0.3937 inches.

- The function multiplies the input cm by 0.3937 to convert it to inches.
- return sends this value back as the output of the function.

Function Call & Output

print(cm_to_inches(10))

- Calls the function with 10 centimeters.
- Calculates: 10 * 0.3937 = 3.937 inches.
- print() displays the result.

Task Description#3

• Few-shot: Provide 2–3 examples to generate a function that formats full names as "Last, First"

Expected Output#3

• Well-structured function respecting the examples

```
def format_name_last_first(full_name):
    """
    Formats a full name string to "Last, First".
    This function splits a name into parts, assumes the last part is the last name,
    and the remaining parts constitute the first name. It then returns the formatted
    string as "Last, First".
    Args:
        full_name (str): A string containing the full name.
    Returns:
        str: The formatted name string.
    """
    # Split the full name into a list of words.
    name_parts = full_name.split()
    # The last element in the list is the last name.
    last_name = name_parts[-1]
    # All elements before the last one are the first name.
    # Join them with a space in case of a middle name.
    first_name = " ".join(name_parts[:-1])
    # Return the formatted string using an f-string.
    return f"{last_name}, {first_name}"
# --- Examples ---
# Example 1: John Doe
name1 = "John Doe"
formatted_name1 = format_name_last_first(name1)
# print(f"'{name1}' becomes '{formatted_name1}'")  # Expected: 'Doe, John'
# Example 2: Jane Smith
```

```
name2 = "Jane Smith"
formatted_name2 = format_name_last_first(name2)
# print(f"'{name2}' becomes '{formatted_name2}'")  # Expected: 'Smith, Jane'
# Example 3: Mary Anne Smith
name3 = "Mary Anne Smith"
formatted_name3 = format_name_last_first(name3)
# print(f"'{name3}' becomes '{formatted_name3}'") # Expected: 'Smith, Mary Anne'
```

Explanation:

The Python function format_name_last_first(full_name) is designed to reorder a person's name from "First Name Last Name" to "Last Name, First Name".

How It Works

1. **Splitting the Name**: The line name_parts = full_name.split() takes the input string (e.g., "John Doe") and breaks it into a list of separate words. For "John Doe", this creates the list ['John', 'Doe']. For a name like "Mary Anne Smith", it creates ['Mary', 'Anne', 'Smith'].
2. **Identifying the Last Name**: The line last_name = name_parts[-1] uses a negative index to grab the very last item from the list. This is a common and efficient Python trick. It correctly identifies 'Doe' in the first example and 'Smith' in the second.
3. **Identifying the First Name**: The line first_name = " ".join(name_parts[:-1]) handles the first name.
    - name_parts[:-1] creates a new list containing all elements **except** the last one.
    - " ".join(...) then combines these elements back into a single string, with a space between each word. This is crucial for multi-part first names like "Mary Anne", ensuring the output is 'Mary Anne' and not 'MaryAnne'.
4. **Returning the Formatted String**: Finally, the line return f"{last_name}, {first_name}" uses an f-string to build the final output. It takes the values stored in the last_name and first_name variables and combines them with a comma and a space in the desired format.
5. This method is robust for simple names and even for names with middle names, as shown in the examples provided in the Canvas

Task Description#4

• Compare zero-shot and few-shot prompts for writing a function that counts the number of vowels in a string.

Expected Output#4

• Functional output and comparative reflection

1. Zero-Shot Prompt

> *Prompt*:
> "Write a Python function that counts the number of vowels in a string."

AI Output (Example)

```python
def count_vowels(s):
    vowels = "aeiouAEIOU"
    return sum(1 for char in s if char in vowels)

print(count_vowels("Hello World"))  # Output: 3
```

- **Explanation**:
  The AI was asked to perform the task without any prior examples.
  It inferred that vowels include both uppercase and lowercase letters.
  Uses a generator expression to count matching characters.

2. Few-Shot Prompt

> *Prompt*:
> "Here are examples:
> Input: 'apple' → Output: 2
> Input: 'Sky' → Output: 0
> Now write a Python function that counts the number of vowels in a string."

AI Output (Example)

```python
def count_vowels(s):
    vowels = "aeiouAEIOU"
    count = 0
    for char in s:
        if char in vowels:
            count += 1
    return count

print(count_vowels("Hello World"))  # Output:
```

Comparative Reflection

- **Zero-Shot**:
  - Relies on AI's internal knowledge of vowel definition and counting.
  - May produce compact code but risks missing edge cases (e.g., uppercase vowels).
- **Few-Shot**:
  - Provides concrete examples to guide reasoning.
  - Produces code aligned with example patterns and less ambiguity.
  - Usually more accurate and reliable, especially for tricky cases.

Task Description#5

• Use few-shot prompting to generate a function that reads a .txt file and returns the number of lines.

Expected Output#5

• Working file-processing function with AI-guided logic.

```python
def count_lines_in_file(filepath):
    """
    Counts the number of lines in a text file efficiently.
    This function opens a file specified by its path, iterates over its lines
    without loading the entire file into memory, and counts them. It uses a
    'with' statement to ensure the file is properly closed, even if errors occur.
    Args:
        filepath (str): The path to the text file.
    Returns:
        int: The total number of lines in the file. Returns -1 if the file
             does not exist or cannot be read.
    """
    try:
        with open(filepath, 'r', encoding='utf-8') as file:
            line_count = sum(1 for line in file)
            return line_count
    except FileNotFoundError:
        print(f"Error: The file at '{filepath}' was not found.")
        return -1
    except Exception as e:
        print(f"An error occurred: {e}")
        return -1
```

```
def count_lines_in_file(filepath):
    except Exception as e:

# --- Example Usage ---

# 1. Create a dummy file for the example.
# You can uncomment and run this section once.
# with open('test_file.txt', 'w', encoding='utf-8') as f:
#     f.write("This is the first line.\n")
#     f.write("This is the second line.\n")
#     f.write("This is the third line.")

# 2. Call the function with the file path.
# file_path = 'test_file.txt'
# num_lines = count_lines_in_file(file_path)

# 3. Print the result.
# if num_lines != -1:
#     print(f"The file '{file_path}' has {num_lines} lines.")
# else:
#     print("Could not count lines due to an error.")

# --- Example for a non-existent file ---
# non_existent_file = 'non_existent_file.txt'
# num_lines_non_existent = count_lines_in_file(non_existent_file)
```

Explanation:

The function count_lines_in_file(filepath) takes a single argument, which is the path to the file you want to analyze. Here's a breakdown of the key parts:

- **Error Handling (try...except)**: The entire file-reading process is wrapped in a try block. This is a best practice that allows the code to handle potential errors gracefully.
    - If the file specified by filepath doesn't exist, a FileNotFoundError is raised. The except block catches this specific error, prints a message, and returns -1.
    - A more general except Exception as e: is included to catch any other unforeseen errors that might occur during the process.
- **Safe File Handling (with open(...))**: The with open(...) statement is a very common pattern in Python for working with files. It ensures that the file is automatically and properly closed once the code block is finished, even if an error occurs. This prevents resource leaks. The arguments r (read mode) and encoding='utf-8' specify how the file should be opened.
- **Efficient Line Counting (sum(1 for line in file))**: This is the most clever part of the code. Instead of reading the entire file into a list of strings (which could use a lot of memory for a very large file), it uses a **generator expression**.
    - for line in file iterates over the file object one line at a time.
    - 1 for line in file generates the number 1 for each line it encounters.
    - sum(...) simply adds up all the 1s, giving you a total count of the lines. This is extremely memory-efficient as it only needs to keep track of a single number (the count) at any given time.