ASSIGNMENT 4

RAMIDI.SAI CHARAN

2403A52124

TASK 1 :

```
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    return True
  else:
    return False

# Example usage:
year_to_check = 2024
if is_leap_year(year_to_check):
  print(f"{year_to_check} is a leap year.")
else:
  print(f"{year_to_check} is not a leap year.")

year_to_check = 1900
if is_leap_year(year_to_check):
  print(f"{year_to_check} is a leap year.")
else:
  print(f"{year_to_check} is not a leap year.")

year_to_check = 2000
if is_leap_year(year_to_check):
  print(f"{year_to_check} is a leap year.")
else:
  print(f"{year_to_check} is not a leap year.")
```

```
2024 is a leap year.
1900 is not a leap year.
2000 is a leap year.
```

EXPLANATION :

- `def is_leap_year(year):` : This line defines a function named `is_leap_year` that takes one argument, `year`.
- `""" ... """` : This is a docstring, which explains what the function does, its arguments (`Args`), and what it returns (`Returns`).
- `# A year is a leap year if it is divisible by 4, ...` : This is a comment explaining the rule for determining a leap year.
- `if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):` : This is the core logic. It checks if the `year` meets the criteria for a leap year:
  - `(year % 4 == 0 and year % 100 != 0)` : This part checks if the year is divisible by 4 but not by 100.
  - `(year % 400 == 0)` : This part checks if the year is divisible by 400.
  - The `or` operator means that if either of these conditions is true, the entire condition is true.
- `return True` : If the `if` condition is true (the year is a leap year), the function returns `True`.
- `else:` : This is the alternative case if the `if` condition is false.
- `return False` : If the `if` condition is false (the year is not a leap year), the function returns `False`.

Task 2:

```
def centimeters_to_inches(cm):
  """Converts centimeters to inches."""
  inches = cm * 0.393701
  return inches


cm_input = 10
inches_output = centimeters_to_inches(cm_input)
print(f"{cm_input} centimeters is equal to {inches_output} inches.")
```

```
10 centimeters is equal to 3.9370100000000003 inches.
```

Explanation :

- `def centimeters_to_inches(cm):` : This line defines a function named `centimeters_to_inches` that takes one argument, `cm` (representing centimeters).
- `"""Converts centimeters to inches."""` : This is a docstring, briefly explaining what the function does.
- `inches = cm * 0.393701` : This is the core of the conversion. It multiplies the input value in centimeters (`cm`) by the conversion factor `0.393701` to get the equivalent value in inches. This result is stored in the variable `inches`.
- `return inches` : This line returns the calculated value in inches.
- `# Example usage:` : This is a comment indicating the start of the example usage section.
- `# Input: 10 centimeters` : A comment showing the example input.
- `# Expected output: 3.93701 inches` : A comment showing the expected output for the example input.
- `cm_input = 10` : This line assigns the value `10` to the variable `cm_input`, representing the input in centimeters for the example.
- `inches_output = centimeters_to_inches(cm_input)` : This line calls the `centimeters_to_inches` function with `cm_input` (which is 10) as the argument. The returned value (the conversion result in inches) is stored in the variable `inches_output`.
- `print(f"{cm_input} centimeters is equal to {inches_output} inches.")` : This line prints the result in a user-friendly format using an f-string. It includes the original centimeter value and the calculated inch value.

Task 3 :

```python
def format_name_last_first(full_name):
   name_parts = full_name.split()
  if len(name_parts) >= 2:
    first_name = name_parts[0]
    last_name = name_parts[-1] # Handles middle names by taking the last part
    return f"{last_name}, {first_name}"
  else:
    return full_name # Return original name if not enough parts
# Example usage:
print(format_name_last_first("John Doe"))
print(format_name_last_first("Jane E. Smith"))
print(format_name_last_first("Peter Jones"))
```

```
Doe, John
Smith, Jane
Jones, Peter
```

Explanation :

- `def format_name_last_first(full_name):` : This line defines a function named `format_name_last_first` that takes one argument, `full_name`, which is expected to be a string containing the full name.
- `""" ... """` : This is a docstring explaining the function's purpose, arguments, and return value.
- `# Assuming the full name has at least a first and a last name` : This is a comment noting an assumption about the input format.
- `name_parts = full_name.split()` : This line splits the `full_name` string into a list of substrings based on whitespace. For example, "John Doe" becomes `['John', 'Doe']`, and "Jane E. Smith" becomes `['Jane', 'E.', 'Smith']`. This list is stored in the `name_parts` variable.
- `if len(name_parts) >= 2:` : This line checks if the `name_parts` list has two or more elements. This is to ensure there's at least a potential first and last name.
- `first_name = name_parts[0]` : If the condition in the `if` statement is true, this line takes the first element of the `name_parts` list (which is assumed to be the first name) and assigns it to the `first_name` variable.
- `last_name = name_parts[-1]` : This line takes the last element of the `name_parts` list (which is assumed to be the last name, even if there are middle names) and assigns it to the `last_name` variable.
- `return f"{last_name}, {first_name}"` : This line constructs the formatted string in the "Last, First" format using an f-string and returns it.
- `else:` : This is the alternative case if the `if` condition is false (meaning the `name_parts` list has fewer than two elements).
- `return full_name` : If there aren't at least two parts to the name, the function returns the original `full_name` string without any changes.

Task 4 :

```python
def count_vowels_zero_shot(text):
  """Counts the number of vowels in a string (zero-shot example)."""
  vowels = "aeiouAEIOU"
  vowel_count = 0
  for char in text:
    if char in vowels:
      vowel_count += 1
  return vowel_count

# Example usage:
print(f"'hello' has {count_vowels_zero_shot('hello')} vowels.")
print(f"'programming' has {count_vowels_zero_shot('programming')} vowels.")
print(f"'Python' has {count_vowels_zero_shot('Python')} vowels.")
```

```
'hello' has 2 vowels.
'programming' has 3 vowels.
'Python' has 1 vowels.
```

```python
def count_vowels_few_shot(text):
  """Counts the number of vowels in a string (few-shot example)."""
  vowels = "aeiouAEIOU"
  vowel_count = 0
  for char in text:
    if char in vowels:
      vowel_count += 1
  return vowel_count

print(f"'hello' has {count_vowels_few_shot('hello')} vowels.")
print(f"'programming' has {count_vowels_few_shot('programming')} vowels.")
print(f"'Python' has {count_vowels_few_shot('Python')} vowels.")
```

```
'hello' has 2 vowels.
'programming' has 3 vowels.
'Python' has 1 vowels.
```

Explanation :

- `def count_vowels_zero_shot(text):` : This line defines a function named `count_vowels_zero_shot` that takes one argument, `text`, which is expected to be the input string.
- `"""Counts the number of vowels in a string (zero-shot example)."""` : This is a docstring that explains what the function does.
- `vowels = "aeiouAEIOU"` : This line creates a string containing all lowercase and uppercase vowels. This string is used to easily check if a character is a vowel.
- `vowel_count = 0` : This line initializes a variable named `vowel_count` to 0. This variable will keep track of the number of vowels found in the string.
- `for char in text:` : This line starts a `for` loop that iterates through each character in the input `text` string. In each iteration, the current character is assigned to the variable `char`.
- `if char in vowels:` : Inside the loop, this line checks if the current character (`char`) is present in the `vowels` string.
- `vowel_count += 1` : If the `if` condition is true (meaning the character is a vowel), this line increments the `vowel_count` by 1.
- `return vowel_count` : After the loop has finished iterating through all the characters in the string, this line returns the final value of `vowel_count`, which is the total number of vowels in the string.

e example usage code below the function demonstrates how to call the function
h different strings and prints the returned vowel count.

Task 5 :

```
    try:
        with open(file_path, 'r') as file:
            line_count = sum(1 for line in file)
            return line_count
    except FileNotFoundError:
        print(f"Error: The file at {file_path} was not found.")
        return -1
    except Exception as e:
        print(f"An error occurred: {e}")
        return -1
if __name__ == "__main__":
    file_name = input("Enter the file name (e.g., 'my_document.txt'): ")
    file_location = input("Enter the file location (e.g., 'C:/Users/Documents/'): ")
    full_path = file_location + file_name
```

Shell

```
>>> %Run -c $EDITOR_CONTENT
Enter the file name (e.g., 'my_document.txt'): lab
Enter the file location (e.g., 'C:/Users/Documents/'):
The file 'lab' has '6' lines.
```

Explanation :

This script is a robust tool for counting lines in a file, designed with error handling. The `count_lines_in_file` function is defined to accept a `file_path`. Inside, a `try` block is used to handle potential errors gracefully. It uses `with open(file_path, 'r') as file:` to open the file in read mode, which is a best practice that ensures the file is automatically closed. The core of the line counting is a compact **generator expression**, `sum(1 for line in file)`, which iterates through each line and sums a `1` for each one, providing the total count. This result is then returned.

If the specified file doesn't exist, the program doesn't crash. Instead, the specific `except FileNotFoundError` block catches this error, prints a user-friendly message, and returns `-1` as an error signal. A more general `except Exception as e` block catches any other unexpected errors, printing the error details before also returning `-1`.

The program's main execution starts within `if __name__ == "__main__":`. It prompts the user for the file name and its location. These two strings are then combined to create the complete file path. The `count_lines_in_file` function is then called with this full path, and its return value is stored. Finally, an `if` statement checks if the returned value is not `-1`. If it isn't, it means the line count was successful, and the program prints the result. If the value is `-1`, it means an error occurred and was already reported by the function, so no further output is generated.