Week 0

Task 1 of 7.1

Task 1. Classify Temperatures:
1. Create empty lists for temperature classifications:
(a) Cold: temperatures below 10°C.
(b) Mild: temperatures between 10°C and 15°C.
(c) Comfortable: temperatures between 15°C and 20°C.

2. Iterate over the temperatures list and add each temperature to the appropriate category.

3. Print the lists to verify the classifications.

```
temperatures = [
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
    16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3,
    13.4, 8.1, 17.9, 14.2, 7.6, 17.0, 12.8, 8.0, 16.8, 13.7,
    7.8, 17.5, 13.6, 8.7, 17.1, 13.8, 9.2, 18.1, 13.9, 8.3,
    16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5
]

cold = []
mild = []
comfortable = []

for temp in temperatures:
    if temp < 10:
        cold.append(temp)
    elif 10 <= temp < 15:
        mild.append(temp)
    elif 15 <= temp <= 20:
        comfortable.append(temp)

classified_temperatures = {
    "Cold": cold,
    "Mild": mild,
    "Comfortable": comfortable
}
print(classified_temperatures);
```

```
{'Cold': [8.2, 7.9, 9.0, 8.5, 7.7, 8.4, 9.5, 8.1, 7.6, 8.0, 7.8, 8.7, 9.2, 8.3,
8.9, 7.8], 'Mild': [14.1, 13.5, 13.0, 12.9, 13.3, 14.0, 13.4, 14.2, 12.8, 13.7,
13.6, 13.8, 13.9, 12.7, 13.1, 12.5], 'Comfortable': [17.4, 18.0, 17.8, 16.5, 17.
2, 16.7, 18.3, 17.9, 17.0, 16.8, 17.5, 17.1, 18.1, 16.4, 18.2, 16.6]}
```

Task 2 of 7.1

Task 2. Based on Data - Answer all the Questions:
1. How many times was it mild?
(a) Hint: Count the number of items in the mild list and print the result.
2. How many times was it comfortable?
3. How many times was it cold?

```
print("Cold temperatures:", len(cold))

print("Mild temperatures:", len(mild))

print("Comfortable temperatures:", len(comfortable))
```

Cold temperatures: 16

Mild temperatures: 16

Comfortable temperature**s**: 16

```
Cold temperatures: 16

Mild temperatures: 16

Comfortable temperatures: 16
```

Task 3 of 7.1

Using the formula for temperature conversion, convert each reading from Celsius to Fahren-
heit and store it in a new list called temperatures_fahrenheit.

Formula: Fahrenheit = (Celsius ×9/5) + 32

1. Iterate over the temperatures list and apply the formula to convert each temperature.

2. Store the results in the new list.
3. Print the converted Fahrenheit values.

```
temperatures_fahrenheit = []


for temp in temperatures:
    fahrenheit = (temp * 9/5) + 32
    temperatures_fahrenheit.append(fahrenheit)


print("Temperatures in Fahrenheit:", temperatures_fahrenheit)
```

```
Temperatures in Fahrenheit: [46.76, 63.32, 57.379999999999995, 46.22, 64.4, 56.3
, 48.2, 64.04, 55.4, 47.3, 61.7, 55.22, 45.86, 62.959999999999994, 55.94, 47.120
000000000005, 62.059999999999995, 57.2, 49.1, 64.94, 56.120000000000005, 46.58,
64.22, 57.56, 45.68, 62.6, 55.04, 46.4, 62.24, 56.66, 46.04, 63.5, 56.48, 47.66,
 62.78, 56.84, 48.56, 64.58, 57.02, 46.94, 61.519999999999996, 54.86, 48.02, 64.
75999999999999, 55.58, 46.04, 61.88, 54.5]
```

Task 4 of 7.1

Task 4. Analyze Temperature Patterns by Time of Day:

Scenario: Each day's readings are grouped as:

• Night (00-08),

• Evening (08-16),

• Day (16-24).

1. Create empty lists for night, day, and evening temperatures.

2. Iterate over the temperatures list, assigning values to each time-of-day list based on their position.

3. Calculate and print the average day-time temperature.

4. (Optional) Plot "day vs. temperature" using matplotlib.

```
night_temps = []

evening_temps = []

day_temps = []


for i in range(0, len(temperatures), 3):

    night_temps.append(temperatures[i])

    evening_temps.append(temperatures[i + 1])

    day_temps.append(temperatures[i + 2])


average_day_temp = sum(day_temps) / len(day_temps)


print("Night temperatures:", night_temps,"\n")

print("Evening temperatures:", evening_temps,"\n")

print("Day temperatures:", day_temps,"\n")

print("Average day-time temperature:", average_day_temp,"\n")
```

```
Night temperatures: [8.2, 7.9, 9.0, 8.5, 7.7, 8.4, 9.5, 8.1, 7.6, 8.0, 7.8, 8.7,
 9.2, 8.3, 8.9, 7.8]

Evening temperatures: [17.4, 18.0, 17.8, 16.5, 17.2, 16.7, 18.3, 17.9, 17.0, 16.
8, 17.5, 17.1, 18.1, 16.4, 18.2, 16.6]

Day temperatures: [14.1, 13.5, 13.0, 12.9, 13.3, 14.0, 13.4, 14.2, 12.8, 13.7, 1
3.6, 13.8, 13.9, 12.7, 13.1, 12.5]

Average day-time temperature: 13.40625
```

Task 2 of 8.1

Task 2 - Generate All Permutations of a String:

Scenario: Given a string, generate all possible permutations of its characters. This is useful
for understanding backtracking and recursive depth-first search.
Task:
• Write a recursive function generate_permutations(s) that:
– Takes a string s as input and returns a list of all unique permutations.
• Test with strings like ”abc” and ”aab”.

print(generate_permutations("abc"))
 # Should return [’abc’, ’acb’, ’bac’, ’bca’, ’cab’, ’cba’]

```
def generate_permutations(s):
    """
    Generate all unique permutations of a string using recursion.

    Args:
        s (str): The input string.

    Returns:
        list: A list of all unique permutations of the string.

    Example:
        >>> generate_permutations("abc")
        ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
    """
    if len(s) == 1:
        return [s]

    permutations = []
```

```
    for i, char in enumerate(s):

        remaining = s[:i] + s[i+1:]
        for perm in generate_permutations(remaining):
            permutations.append(char + perm)

    return list(set(permutations))

print("Permutations of 'jkl':", generate_permutations("jkl"))

print("Permutations of 'def':", generate_permutations("def"))
```

```
Permutations of 'jkl': ['kjl', 'lkj', 'ljk', 'jkl', 'klj', 'jlk']
Permutations of 'def': ['def', 'dfe', 'fed', 'fde', 'efd', 'edf']
```

Task 3 of 8.1.1

1. Write a recursive function calculate_directory_size(directory) where:
• directory is a dictionary where keys represent file names (with values as sizes in

KB) or directory names (with values as another dictionary representing a subdirectory).

• The function should return the total size of the directory, including all nested subdirectories.
2. Test the function with a sample directory structure.

```
directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
```

```
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250,
            "file6.txt": 150
        }
    }
}

def calculate_directory_size(directory):
    total_size = 0
    for key, value in directory.items():
        if isinstance(value, dict):
            total_size += calculate_directory_size(value)
        else:
            total_size += value
    return total_size

total_size = calculate_directory_size(directory_structure)
print(f"Total directory size: {total_size} KB \n")
```

```
Total directory size: 1400 KB
```

Task 2 of 8.2.2

Task 2 - Longest Common Subsequence (LCS):
Scenario: Given two strings, find the length of their longest common subsequence (LCS). This is useful in text comparison.
Task:
1. Write a function longest_common_subsequence(s1, s2) that:
• Uses DP to find the length of the LCS of two strings s1 and s2.
2. Test with strings like "abcde" and "ace"; the LCS length should be 3 ("ace").

```
def longest_common_subsequence(s1, s2):
    """
    Finds the length of the longest common subsequence (LCS) between two strings using dynamic programming.

    Args:
        s1 (str): The first string.
        s2 (str): The second string.
```

```
    Returns:
        int: The length of the LCS.

    Example:
        >>> longest_common_subsequence("abcde", "ace")
        3
    """

    dp = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]

    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[len(s1)][len(s2)]

print("Length of LCS:", longest_common_subsequence("abcde", "abce"))
```

```
Length of LCS: 4
```

task 3 of 8.2.2

Task 3 - 0/1 Knapsack Problem:
Scenario: You have a list of items, each with a weight and a value. Given a weight capacity,
maximize the total value of items you can carry without exceeding the weight capacity.
Task:
1. Write a function knapsack(weights, values, capacity) that:
• Uses DP to determine the maximum value that can be achieved within the given weight capacity.

2. Test with weights [1, 3, 4, 5], values [1, 4, 5, 7], and capacity 7. The result should be 9.

```
def knapsack(weights, values, capacity):
    """
    Solves the 0/1 Knapsack problem using dynamic programming.
```

```
    Args:
        weights (list): List of item weights.
        values (list): List of item values.
        capacity (int): Maximum weight capacity of the knapsack.

    Returns:
        int: Maximum value achievable within the given weight capacity.

    Example:
        >>> knapsack([1, 3, 4, 5], [1, 4, 5, 7], 7)
        9
    """
    n = len(weights)

    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                include = values[i - 1] + dp[i - 1][w - weights[i - 1]]
                exclude = dp[i - 1][w]
                dp[i][w] = max(include, exclude)
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
capacity = 7

max_value = knapsack(weights, values, capacity)
print("Maximum Value:", max_value)
```

```
Maximum Value: 9
```