5CS037 - Concepts and Technologies of AI. Worksheet - 0: Python Essentials for Machine Learning.

7 TO - DO - Task

Please complete all the problem listed below.

7.1 Warming Up Exercise:

In this exercise, you'll work with temperature data from Tribhuwan International Airport, Kathmandu. The data spans one month and represents typical early winter temperatures.

- Datasets: The temperatures list contains daily temperature readings in Celsius for one month in Kathmandu. Each day includes three readings representing night (00-08), evening (08-16), and day (16-24) temperatures.

Sample Code - List of temperature measured at Tribhuwan International Airport.

```
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]
```

Complete all the task below:

Task 1. Classify Temperatures:

1. Create empty lists for temperature classifications:

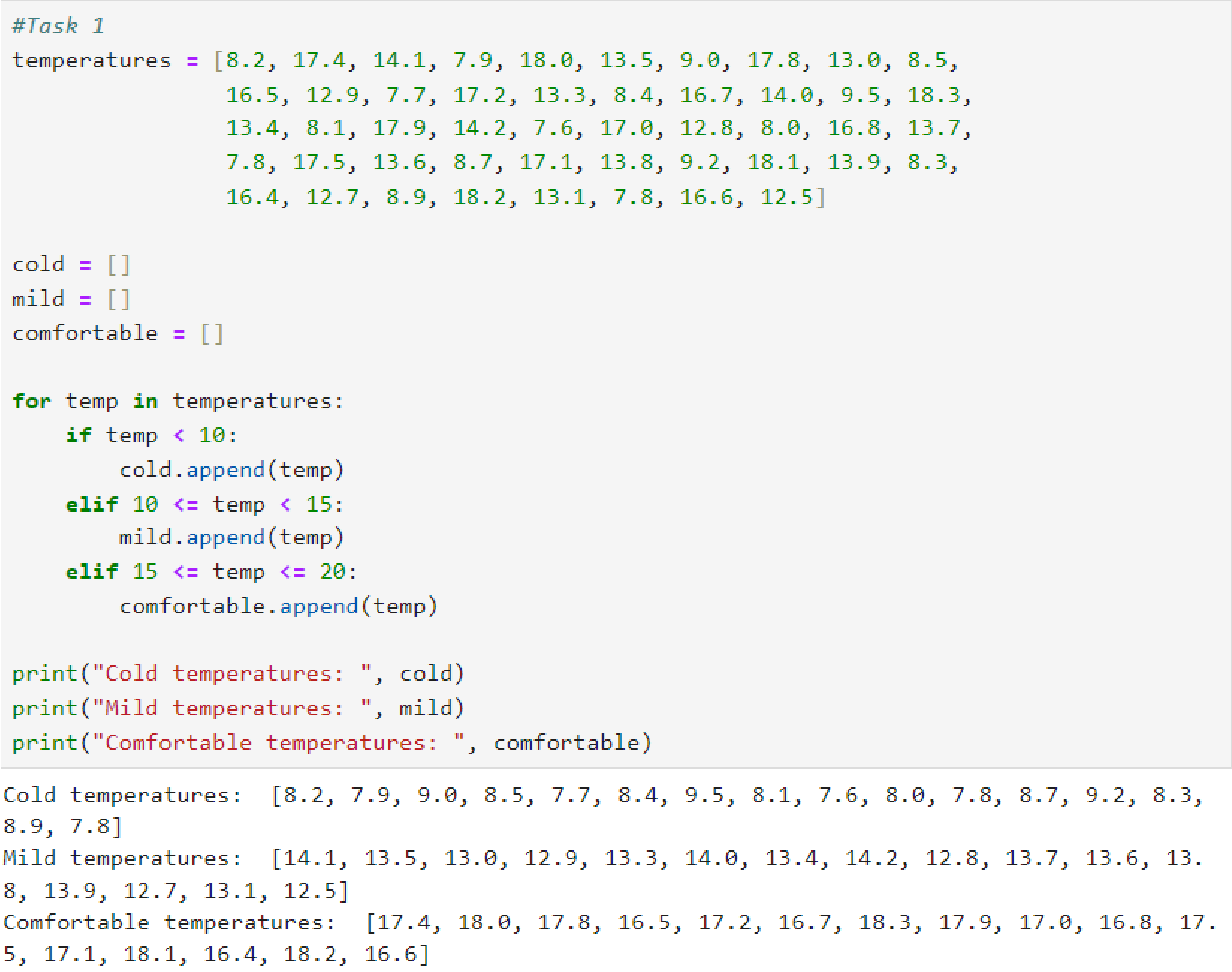
   (a) Cold: temperatures below 10°C.

   (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 cate gory.

3. Print the lists to verify the classifications.



```
#Task 1
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)

print("Cold temperatures: ", cold)
print("Mild temperatures: ", mild)
print("Comfortable temperatures: ", comfortable)
```

```
Cold 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]
Mild temperatures:  [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 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]
```

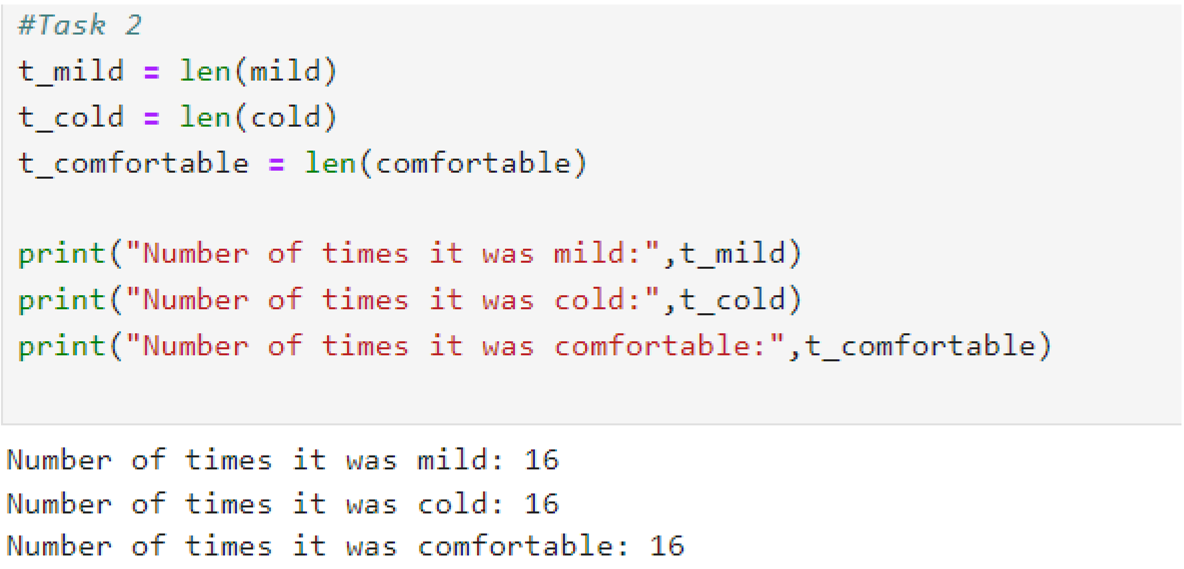
Task 2. Based on Data - Answer all the Questions:

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?



```
#Task 2
t_mild = len(mild)
t_cold = len(cold)
t_comfortable = len(comfortable)

print("Number of times it was mild:",t_mild)
print("Number of times it was cold:",t_cold)
print("Number of times it was comfortable:",t_comfortable)
```

```
Number of times it was mild: 16
Number of times it was cold: 16
Number of times it was comfortable: 16
```

Task 3. Convert Temperatures from Celsius to Fahrenheit

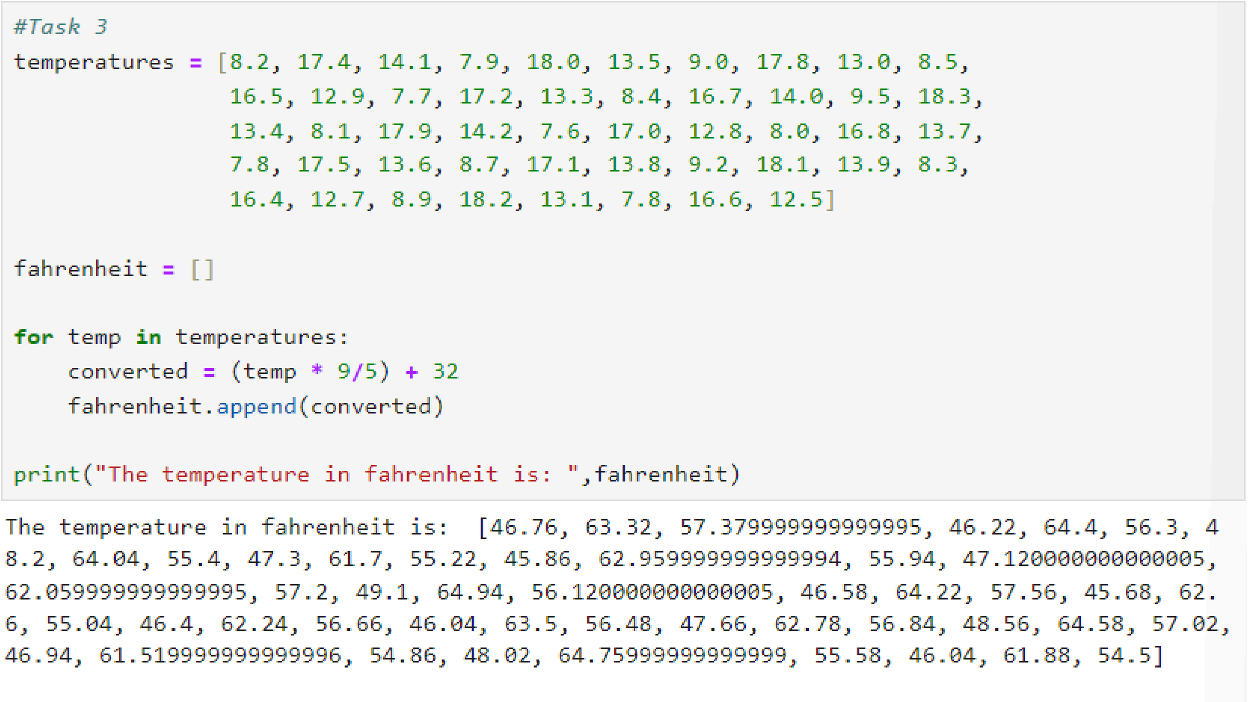
Using the formula for temperature conversion, convert each reading from Celsius to Fahren heit and store it in a new list called temperatures_fahrenheit.

$$\text{Formula: Fahrenheit} = (\text{Celsius} \times \frac{9}{5}) + 32$$

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.



```
#Task 3
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]

fahrenheit = []

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

print("The temperature in fahrenheit is: ",fahrenheit)
```

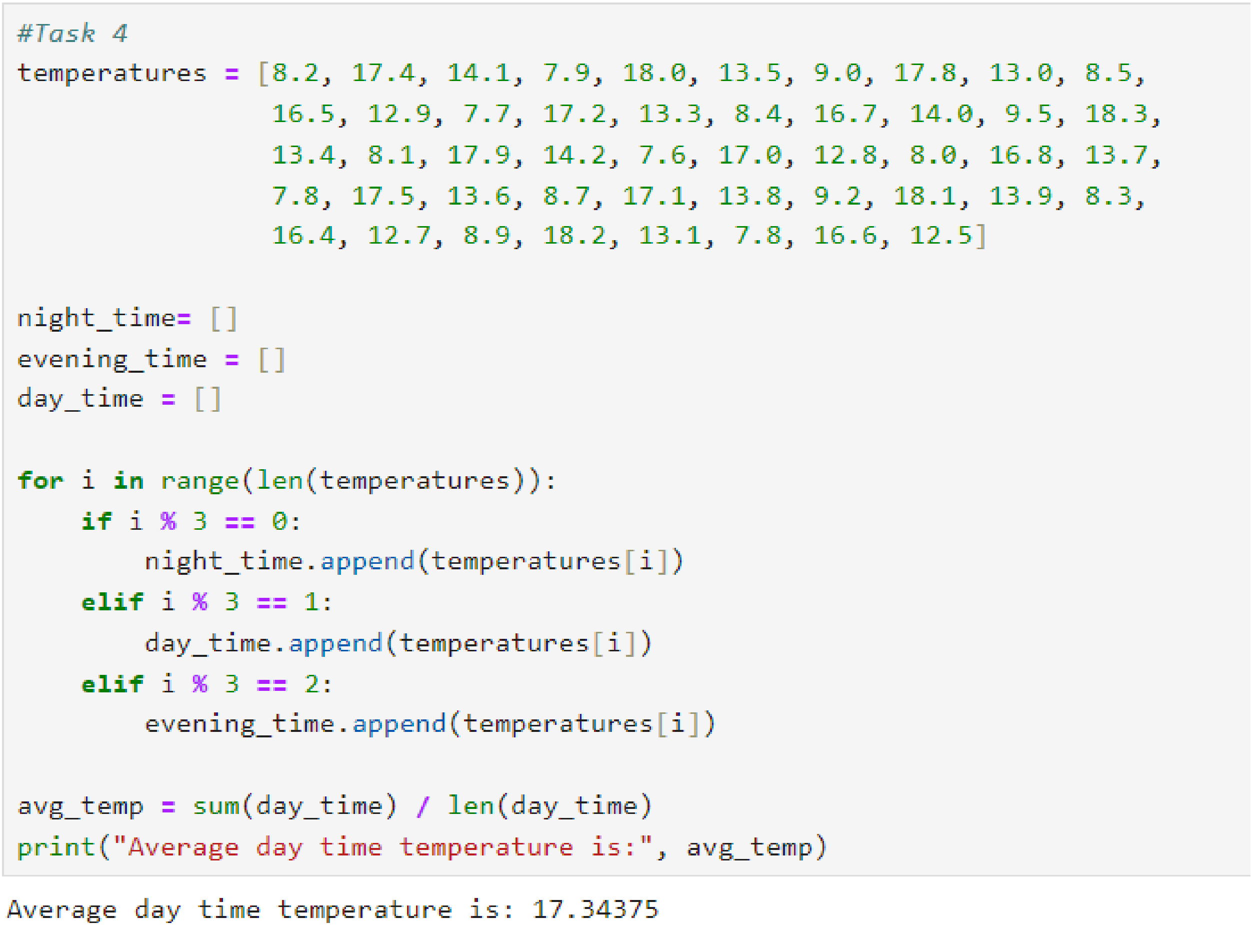
```
The temperature in fahrenheit is:  [46.76, 63.32, 57.379999999999995, 46.22, 64.4, 56.3, 4
8.2, 64.04, 55.4, 47.3, 61.7, 55.22, 45.86, 62.959999999999994, 55.94, 47.120000000000005,
62.059999999999995, 57.2, 49.1, 64.94, 56.120000000000005, 46.58, 64.22, 57.56, 45.68, 62.
6, 55.04, 46.4, 62.24, 56.66, 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. 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.



```
#Task 4
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]

night_time= []
evening_time = []
day_time = []

for i in range(len(temperatures)):
    if i % 3 == 0:
        night_time.append(temperatures[i])
    elif i % 3 == 1:
        day_time.append(temperatures[i])
    elif i % 3 == 2:
        evening_time.append(temperatures[i])

avg_temp = sum(day_time) / len(day_time)
print("Average day time temperature is:", avg_temp)
```

```
Average day time temperature is: 17.34375
```

8 Problem based on Popular Algorithm.

8.1.1 Exercise - Recursion:

Task 1 - Sum of Nested Lists:

Scenario: You have a list that contains numbers and other lists of numbers (nested lists). You want to find the total sum of all the numbers in this structure.
Task:

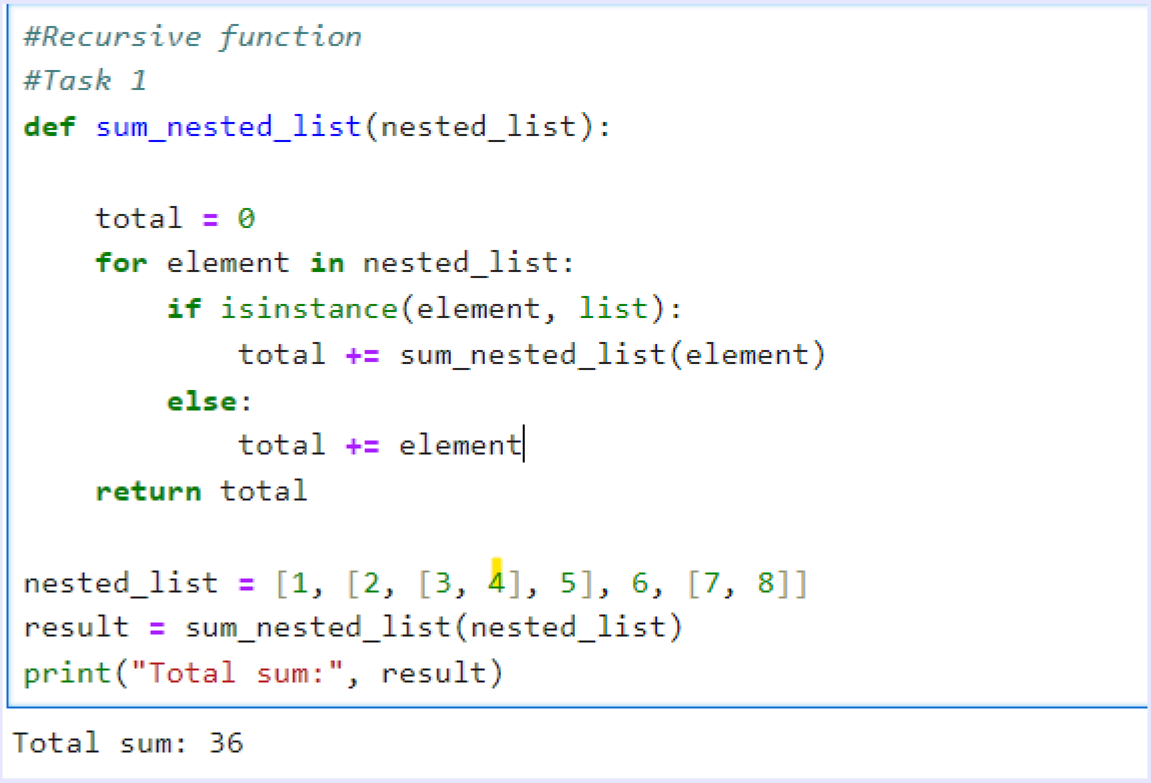
- Write a recursive function sum_nested_list(nested_list) that:
  1. Takes a nested list (a list that can contain numbers or other lists of numbers) as input.
  2. Sums all numbers at every depth level of the list, regardless of how deeply nested the numbers are.
- Test the function with a sample nested list, such as
  nested_list = [1, [2, [3, 4], 5], 6, [7, 8]].
  The result should be the total sum of all the numbers.

Sample Code - Sum of Nested lists.

```
def sum_nested_list(nested_list):
```

```
"""
Calculate the sum of all numbers in a nested list.
This function takes a list that may contain integers and other nested lists. It recursively traverses
the list and sums all the integers, no matter how deeply nested they are.
Args:
    nested_list (list): A list that may contain integers or other lists of integers. Returns:
```

```
    int: The total sum of all integers in the nested list, including those in sublists .
Example:
    >>> sum_nested_list([1, [2, [3, 4], 5], 6, [7, 8]])
    36
    >>> sum_nested_list([1, [2, 3], [4, [5]]])
    15
"""
total = 0
for element in nested_list:
    if isinstance(element, list): # Check if the element is a list
        total += sum_nested_list(element) # Recursively sum the nested list
    else:
        total += element # Add the number to the total
return total
```



```
#Recursive function
#Task 1
def sum_nested_list(nested_list):

    total = 0
    for element in nested_list:
        if isinstance(element, list):
            total += sum_nested_list(element)
        else:
            total += element
    return total

nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
result = sum_nested_list(nested_list)
print("Total sum:", result)
```

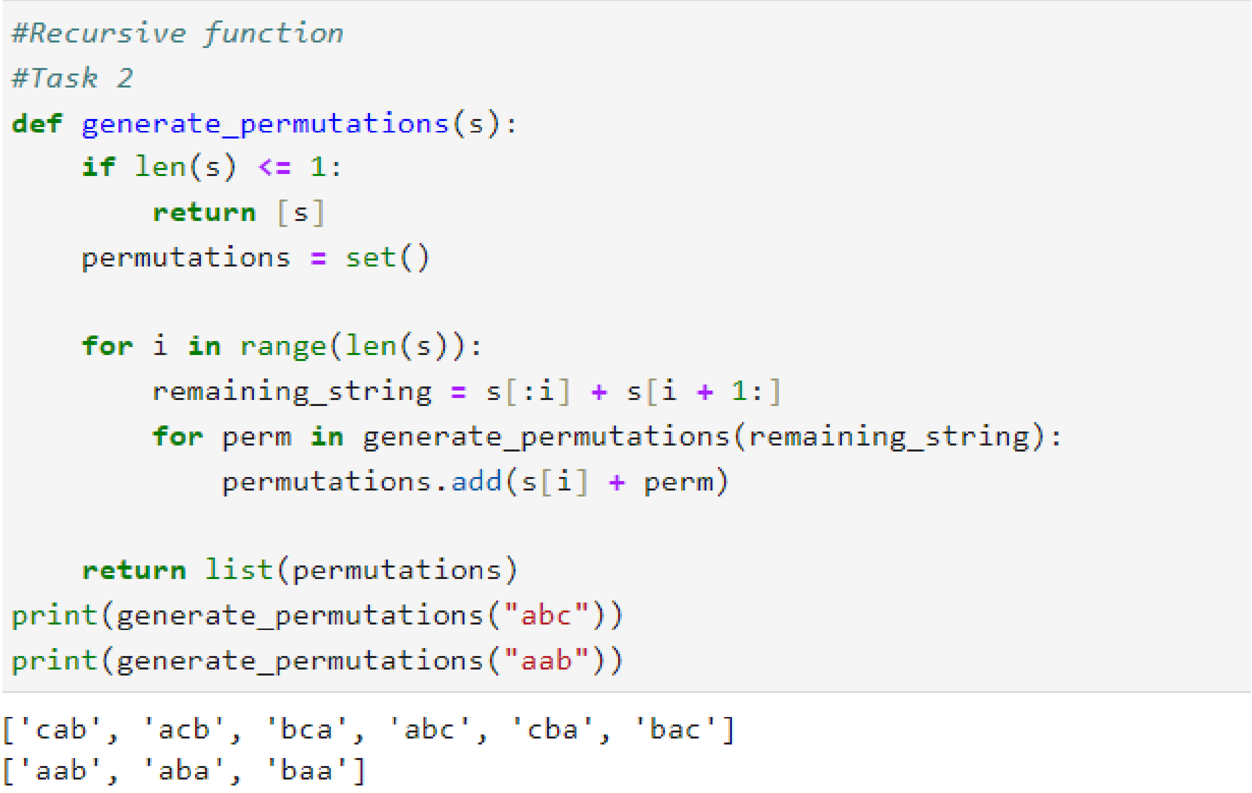
```
Total sum: 36
```

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’]
```



```
#Recursive function
#Task 2
def generate_permutations(s):
    if len(s) <= 1:
        return [s]
    permutations = set()

    for i in range(len(s)):
        remaining_string = s[:i] + s[i + 1:]
        for perm in generate_permutations(remaining_string):
            permutations.add(s[i] + perm)

    return list(permutations)
print(generate_permutations("abc"))
print(generate_permutations("aab"))
```

```
['cab', 'acb', 'bca', 'abc', 'cba', 'bac']
['aab', 'aba', 'baa']
```

Task 3 - Directory Size Calculation:

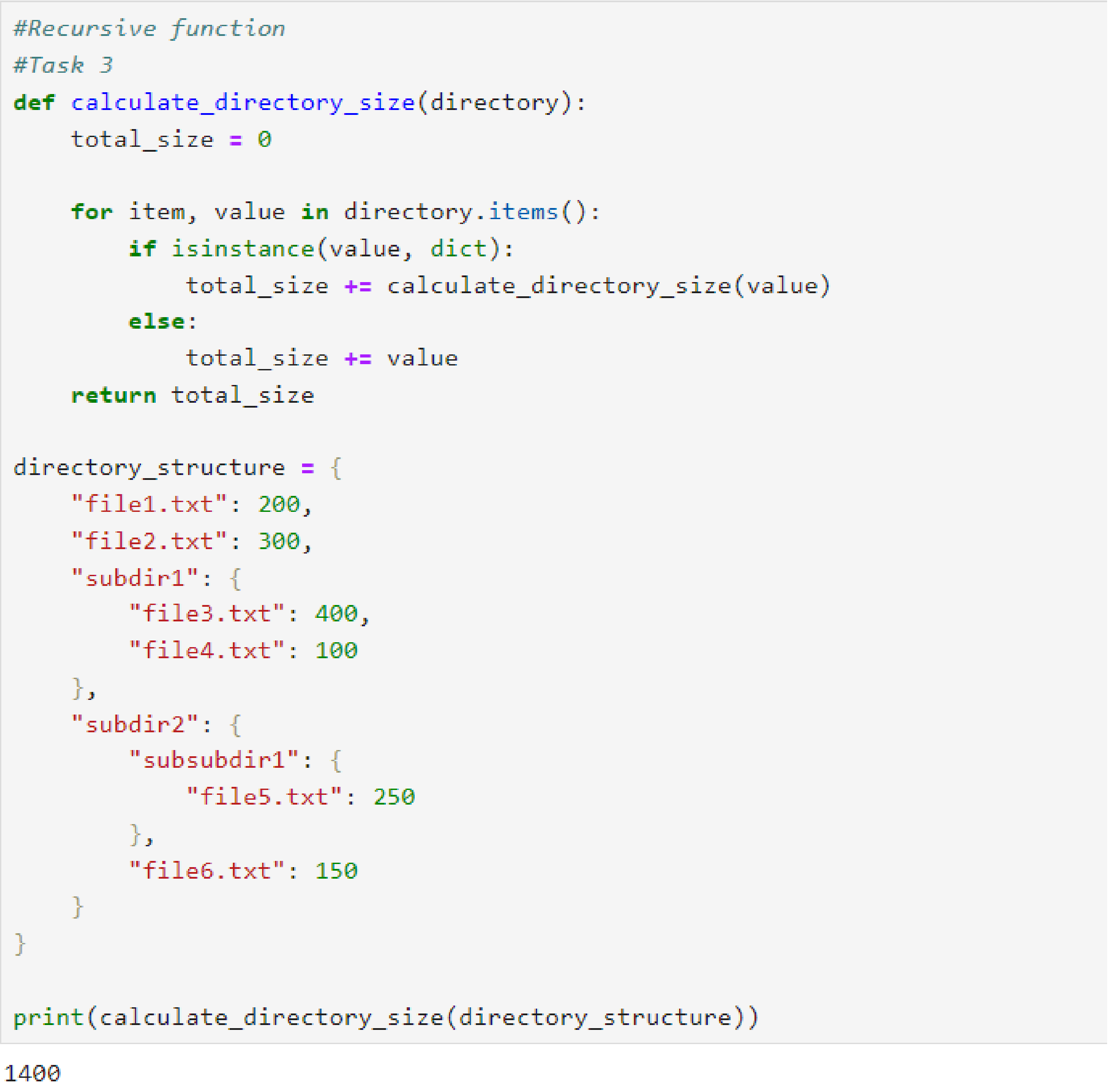
Directory Size Calculation Scenario: Imagine a file system where directories can contain files (with sizes in KB) and other directories. You want to calculate the total size of a directory, including all nested files and subdirectories.

Sample directory structure.

```
# 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
    }
}
```

Task:

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



```python
#Recursive function
#Task 3
def calculate_directory_size(directory):
    total_size = 0

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

directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250
        },
        "file6.txt": 150
    }
}

print(calculate_directory_size(directory_structure))
```

```
1400
```

8.2.2 Exercises - Dynamic Programming:

Task 1 - Coin Change Problem:

Scenario: Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the amount. If it’s not possible, return - 1.
Task:

1. Write a function min_coins(coins, amount) that:

   - Uses DP to calculate the minimum number of coins needed to make up the amount.

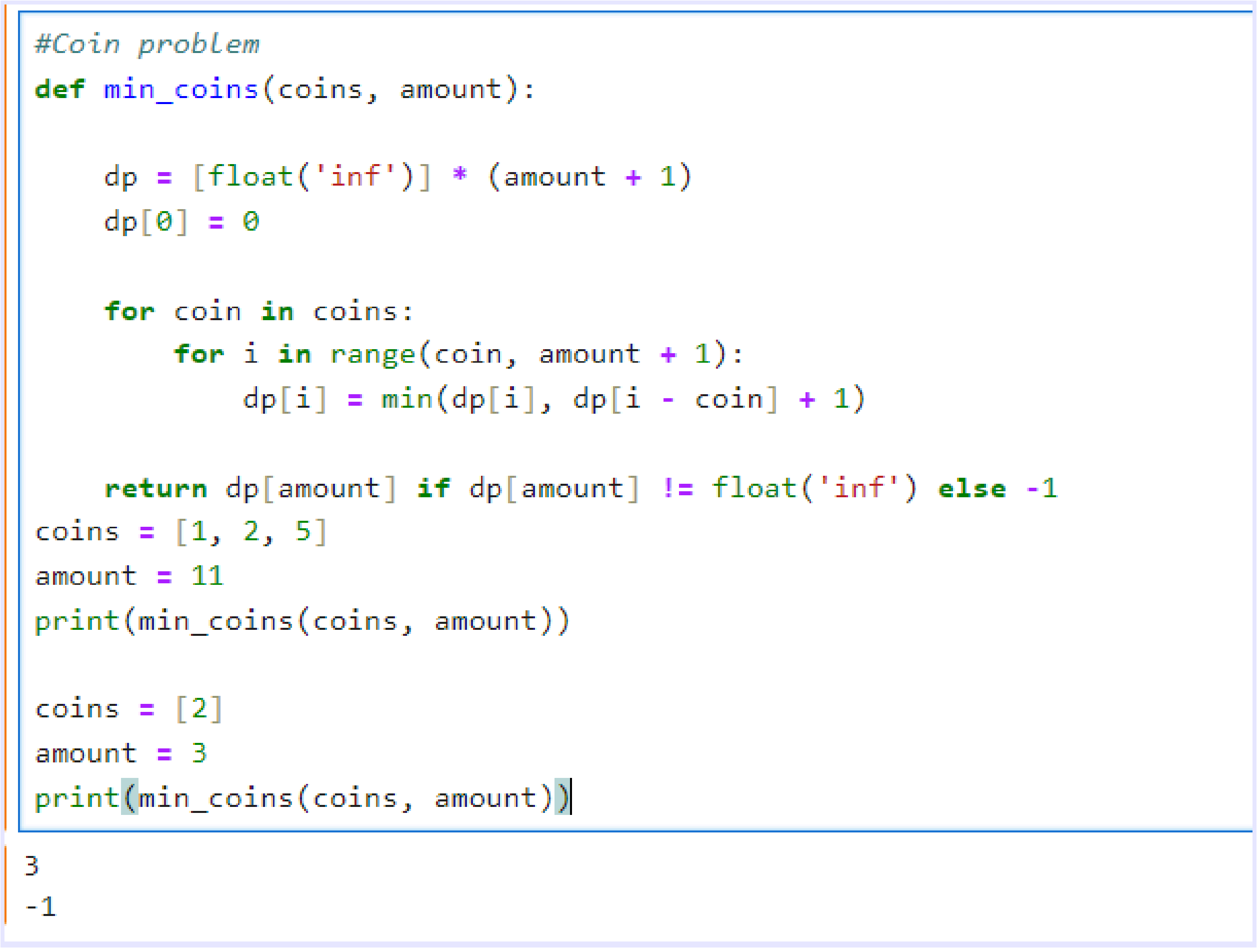
2. Test with coins = [1, 2, 5] and amount = 11. The result should be 3 (using coins [5, 5, 1]).

Sample Code - Coin Change Problem.

```
def min_coins(coins, amount):
"""
Finds the minimum number of coins needed to make up a given amount using dynamic programming.
This function solves the coin change problem by determining the fewest number of coins from a given set
of coin denominations that sum up to a target amount. The solution uses dynamic
programming(tabulation) to iteratively build up the minimum number of coins required for each amount.
Parameters:
coins (list of int): A list of coin denominations available for making change. Each coin denomination is a
positive integer.
amount (int): The target amount for which we need to find the minimum number of coins . It must be a
non-negative integer.
Returns:
int: The minimum number of coins required to make the given amount.
If it is not possible to make the amount with the given coins, returns -1. Example:
>>> min_coins([1, 2, 5], 11)
3
>>> min_coins([2], 3)
-1
"""
dp = [float(’inf’)] * (amount + 1)
dp[0] = 0

for coin in coins:
for i in range(coin, amount + 1):
dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] != float(’inf’) else -1
```



```
#Coin problem
def min_coins(coins, amount):

    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1
coins = [1, 2, 5]
amount = 11
print(min_coins(coins, amount))

coins = [2]
amount = 3
print(min_coins(coins, amount))
```

```
3
-1
```

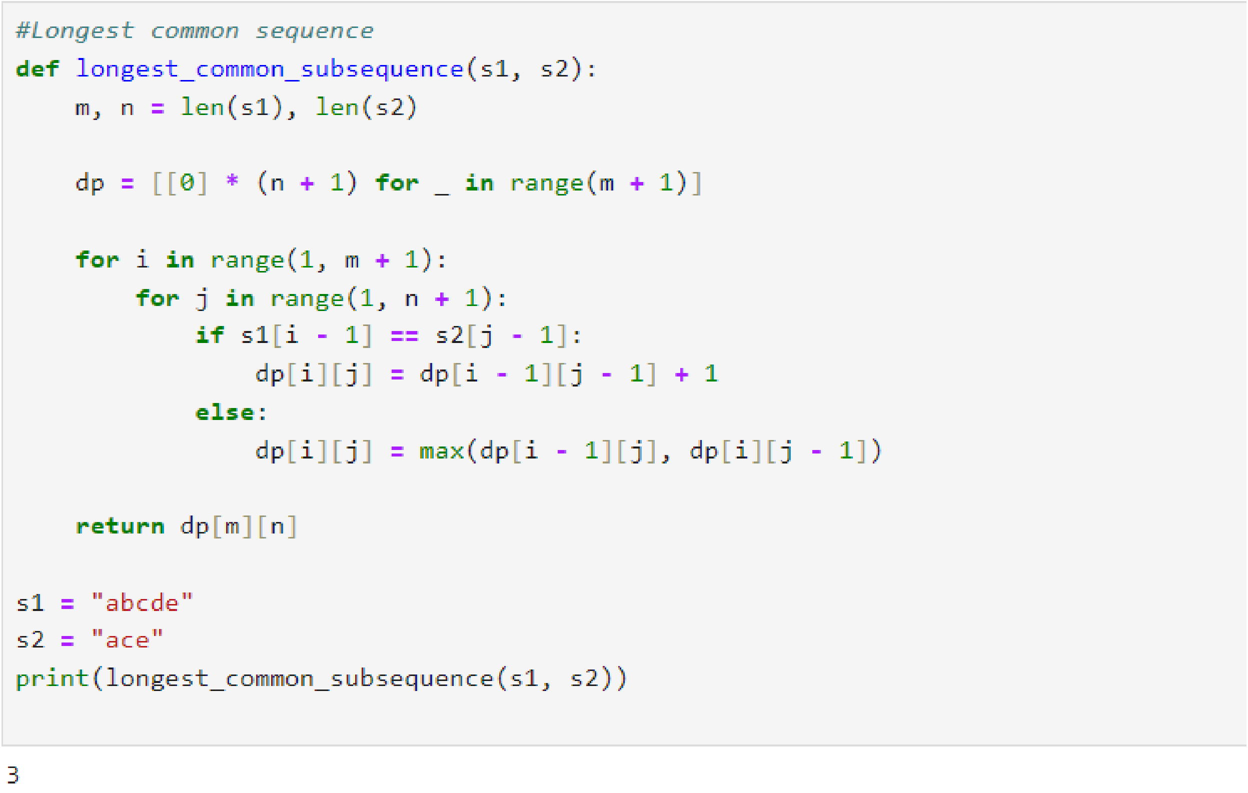
Task 2 - Longest Common Subsequence (LCS):

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"). 23



```python
#Longest common sequence
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)

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

    for i in range(1, m + 1):
        for j in range(1, n + 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[m][n]

s1 = "abcde"
s2 = "ace"
print(longest_common_subsequence(s1, s2))
```

```
3
```

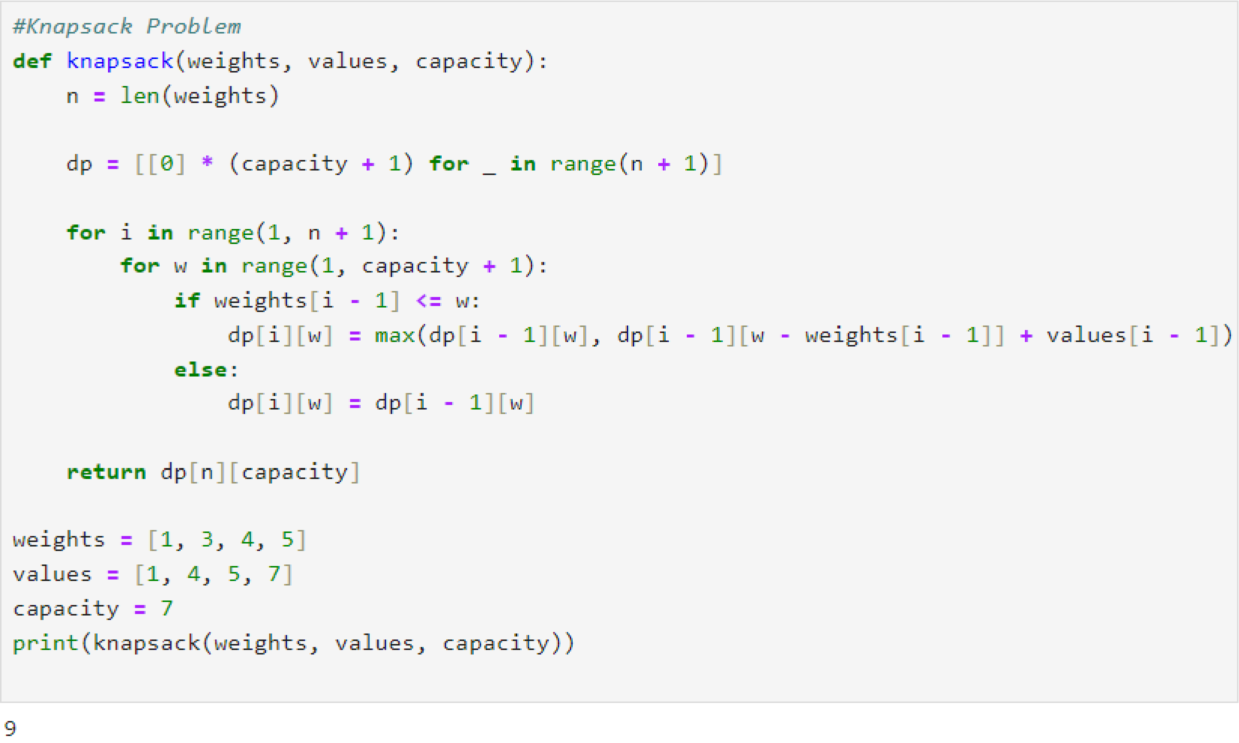
Task 3 - 0/1 Knapsack Problem:

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 re sult should be 9.



```
#Knapsack Problem
def knapsack(weights, values, capacity):
    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:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
capacity = 7
print(knapsack(weights, values, capacity))
```

```
9
```