5CS037 - Concepts and Technologies of AI.

Given By: Siman Giri {Module Leader - 5CS037}

Completed By: Kamal Dhital {Group - L5CG5}

1 Instructions

This is a Pre-requisite homework assignment to be completed on your own before your firstworkshop and is compulsory to submit.

{Cautions!!!:Failure to Submit this assignment might affect your future grades and abilityto receive highest grades}.Please answer the questions below using python in the Jupyter Notebook and follow theguidelines below:

> • This worksheet must be completed individually.
>
> • All the solutions must be written in Jupyter Notebook.
>
> • You are allowed to use basic packages liketime, collection etc.but do not use thepackages to solve the problem directly.

2 Getting Started with Python.

This is NOT a problem, but you are highly recommended to run the following code withsome of the input changed in order to understand the meaning of the operations.

> • Cautions!!!:
>
> > – This Guide doesnot contain sample output, as we expect you to re-write the codeand observe the output.
> >
> > – If found: any error or bugs, please report to your instructor and Module leader.
> >
> > {Will hugely appreciate your effort.}

2.1 About a Python:

Python is a high-level, general-purpose programming language created by Guido van Rossum,first released in 1991. Its design focuses on making code easy to read and understand, usingclear formatting and meaningful whitespace. Python's structure and support for object-oriented

programming help programmers write organized, logical code adaptable for bothsmall-scale and large-scale projects. Known for its flexibility and powerful libraries, Pythonhas become a popular language for machine learning research and is the main language usedin frameworks likeNumpy, Pandas, Matplotlib, sickit learnand many more.

Python Version Check

```
import sys
# Check Python version
if sys.version_info.major == 3 and sys.version_info.minor >= 6:
    print("Hello, World!")
    print(f"Python version: {sys.version}")
else:
    print("Please use Python 3.6 or higher.")
```

```
Hello, World!
Python version: 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0]
```

3 Data Types in Python:

Mutable Data Types:Mutable objects can be changed after they are created. This meansyou can modify their contents, such as adding or removing items, or changing their valueswithout creating a new object.

Immutable Data Types:Immutable objects cannot be changed once they are created.Any operation that seems to modify an immutable object will actually create a new objectinstead of changing the original.

3.1 Some Common Data Types in Python:

Numeric

These data types are used to represent numerical values.

> • int: Represents integer values (e.g., 10, -3).

> • float: Represents floating-point (decimal) values (e.g., 10.5, -2.7).

> • complex: Represents complex numbers (e.g., 3 + 4j)

Data Types - Numeric

```
age = 23 #int
pi = 3.14 #float
temperature = -5.5 #float
print("data type of variable age = ", type(age))
```

```
print("data type of variable pi = ", type(pi))
print("data type of variable temperature = ", type(temperature))
```

```
data type of variable age =  <class 'int'>
data type of variable pi =  <class 'float'>
data type of variable temperature =  <class 'float'>
```

Sequence

These data types are ordered collections of items. You can access elements by their position(index).

> • **str**: string (str) represents sequence of characters enclosed by double quotes or singlequotes. (e.g., “Hello, World!”).It is an immutable sequence.

Data Types - Sequence - String

```
name = "Alice"
greeting = 'Hello'
address = "123 Main St"
print("data type of the variable name = ",type(name))
print("data type of the variable greeting = ",type(greeting))
print("data type of the variable address = ",type(address))
# slice only one element
print("The first letter of the name is:", name[0])
print("The last letter of the name is:", name[-1])
# slice a range of elements
print("The second letter to the fourth of the name is:", name[1:4])
print("The first two letters of the name are:", name[:2])
print("Substring starting from the third letter is:", name[2:])
```

```
data type of the variable name =  <class 'str'>
data type of the variable greeting =  <class 'str'>
data type of the variable address =  <class 'str'>
The first letter of the name is: A
The last letter of the name is: e
The second letter to the fourth of the name is: lic
The first two letters of the name are: Al
Substring starting from the third letter is: ice
```

> • **list**: Represents lists, which can contain mixed data types and are mutable (e.g., ["apple", "banana"]).

Data Types - Sequence - List

```
list1 = [1, 2, 3, 4]
mixed_list = [12, "Hello", True]
# List is mutable
```

```
mixed_list[0] = False
mixed_list
```

```
[False, 'Hello', True]
```

> • **tuple**: Represents tuples, which are ordered and immutable collections (e.g.,(1, 2,3)).

Data Types - Sequence - Tuple

```
colors = ('red', 'green', 'yellow', 'blue')
print("First element:", colors[0])
print("Last two elements:", colors[2:])
print("Middle two elements:", colors[1:3])
# colors[0] = 'purple'
colors # will generate an error as tuple is immutable.
```

```
First element: red
Last two elements: ('yellow', 'blue')
Middle two elements: ('green', 'yellow')
('red', 'green', 'yellow', 'blue')
```

Mapping:

This category includes data types that store key-value pairs, allowing for efficient retrievalbased on keys.

> • dict: Represents dictionaries, which can store various data types as values associatedwith unique keys (e.g.,"name": "Alice", "age": 25).

Data Types - Sequence - Dict

```
person = {'name':'John','age':30,'city':'Pittsburgh'}
print(f"Hello my name is {person['name']}. I am {person['age']} years old andI live at {pers
print("All keys:", list(person.keys()))
print("All values:", list(person.values()))
```

```
Hello my name is John. I am 30 years old andI live at Pittsburgh.
All keys: ['name', 'age', 'city']
All values: ['John', 30, 'Pittsburgh']
```

Set:

Sets are unordered collections of unique elements. They are useful for membership testingand eliminating duplicate entries.

> • set: A mutable collection of unique items (e.g.,1, 2, 3).

> • frozenset: An immutable version of a set (e.g.,frozenset([1, 2, 3])).

Data Types - Set

```
unique_numbers = {1,2,3,3,3,3,4,5}
print(unique_numbers)
```

```
{1, 2, 3, 4, 5}
```

Boolean:

This category contains types that represent truth values.

> • bool: Represents boolean values (True or False).

Data Types - Boolean

```
is_student = True
has_license = False
print("data type of the variable is_student = ",type(is_student))
print("data type of the variable has_license = ",type(has_license))
```

```
data type of the variable is_student =  <class 'bool'>
data type of the variable has_license =  <class 'bool'>
```

Special:

This category is used for unique data types that do not fit into the other categories.

> • **NoneType**: Represents the absence of a value or a null value (e.g.,None).

Data Types - Boolean

```
result = None
```

4 Logical Statements and Loops.

Python code can be decomposed into packages, modules, statements, and expressions, as follows:

> 1. Expressions create and process objects:
>
> > • Expressions are part of statements that return a value, such as variables, opera-tors, or function calls.
>
> 2. Statements contain expressions:

> • Statements are sections of code that perform an action. The main groups ofPython statements are: assignment statements, print statements, conditionalstatements (if, break, continue, try), and looping statements (for, while).

3. Module Contain statements:

> • Modules are Python files that contain Python statements, and are also calledscripts.

4. Packages are composed of modules:

> • Packages are Python programs that collect related modules together within asingle directory hierarchy.

4.1 Logical Statements:

Logical statements are used to perform conditional operations. The primary logical state-ments in Python are:

• if: Executes a block of code if the condition is true.

```
if condition:
    # Code to execute if condition is true
```

• elif: Short for ”else if,” allows for multiple conditions to be checked sequentially.

```
if condition1:
    # Code for condition1
elif condition2:
    # Code for condition2
```

• else: Executes a block of code if none of the preceding conditions are true.

```
if condition:
    # Code if condition is true
else:
    # Code if condition is false
```

Sample Code - if-else statement

```
num = 10
if num > 0:
    print("Positive")
elif num == 0:
    print("Zero")
else:
    print("Non-positive")
```

```
Positive
```

- Comparison operators: Used to compare values.
  - ==: Equal to
  - !=: Not equal to
  - <: Less than
  - >: Greater than
  - <=: Less than or equal to
  - >=: Greater than or equal to

Sample Code - Comparision Operator

```
x = 5
y = 10
if x > 0 and y < 20:
    print("Both conditions are true")
if x > 0 or y > 20:
    print("At least one condition is true")
if not x == 0:
    print("x is not equal to 0")
```

```
Both conditions are true
At least one condition is true
x is not equal to 0
```

- Logical operators: Used to combine multiple conditions.
  - and: True if both conditions are true.
  - or: True if at least one condition is true.

– not: Inverts the truth value of the condition.

Sample Code - Logical Operator

```
# Define two boolean variables
a = True
b = False
# Using the'and' operator
if a and b:
    print("Both a and b are True")
else:
    print("Either a or b is False") # This will be printed
# Using the ’or’ operator
if a or b:
    print("At least one of a or b is True") # This will be printed
else:
    print("Both a and b are False")
# Using the ’not’ operator
if not a:
    print("a is False") # This will not be printed
else:
    print("a is True") # This will be printed
```

```
Either a or b is False
At least one of a or b is True
a is True
```

4.2 Loops:

Loops are used to execute a block of code multiple times. The primary loop types in Python are:

• forloop: Iterates over a sequence (like a list, tuple, or string).

```
for item in iterable:
    # Code to execute for each item
```

• whileloop: Repeats as long as a condition is true.

```
while condition:
    # Code to execute while condition is true
```

• break: Exits the loop immediately.

```
for item in iterable:
    if some_condition:
```

```
        break # Exit loop
```

•continue: Skips the current iteration and continues with the next iteration of the loop.

```
    for item in iterable:
        if some_condition:
            continue # Skip to the next iteration
```

Sample Code - Various Loops

```
# for loop:
fruits = ["apple", "banana","cherry"]
for fruit in fruits:
    print(fruit)
# While loop:
count = 0
while count < 5:
    print("Count is:", count)
    count += 1
# Break
fruits = ["apple", "banana", "orange"]
print("loop 1")
for fruit in fruits:
    print(fruit)
    if fruit == "apple":
        break
# Continiue
print("loop 2")
for fruit in fruits:
    if fruit == "apple":
        continue
    else:print(fruit)
```

```
apple
banana
cherry
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
loop 1
apple
loop 2
banana
orange
```

5 Functions.

How to write a Function Correctly? - Docstring are Most.

Correct way to Write Function

```
def add_binary(a, b):
    '''Returns the sum of two decimal numbers in binary digits.

    Parameters:
        a (int): A decimal integer
        b (int): Another decimal integer

    Returns:
    binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

5.2 Built - in - Functions:

Python provides a wide range of built-in functions that are ready to use. These func-tions perform common tasks and are available without the need to import additional mod-ules.Presented below is an example showcase; for more details, refer to the Python Referenceon W3Schools.

Example on Built - in - Function

```
print("Hello, World!") # Output: Hello, World!
length = len("Hello") # Output: 5
data_type = type(42) # Output: <class 'int'>
total = sum([1, 2, 3]) # Output: 6
```

```
Hello, World!
```

5.3 Built - in - Methods:

Methods are similar to functions but are associated with objects. They act specificallyon data types (e.g.,strings, lists, dictionariesand are called using dot notationobject.method(). Presented below is an example showcase; for more details, refer to thePython Reference on W3Schools

Example on Built - in - Methods

```
# str.upper(): Converts all characters in a string to uppercase.
message = "hello".upper() # Output: "HELLO"
# list.append(): Adds an element to the end of a list.
fruits = ["apple", "banana"]
fruits.append("cherry") # Output: ["apple", "banana", "cherry"]
# dict.get(): Returns the value associated with a key in a dictionary.
```

```
info = {"name": "Alice", "age": 25}
age = info.get("age") # Output: 25
```

5.4 User - Defined - Function:

User-defined functions are created by the user to perform specific tasks. These functions aredefined using thedefkeyword and may include parameters and return values.Here is an example function python that converts Celsius to Fahrenheit and vice versa.

Example of well-structured user-defined function

```
def temperature_converter():
    """Converts temperature between Celsius and Fahrenheit.
    This function prompts the user to specify the conversion type (Celsius to Fahrenheitor F

    Returns:
        float: The converted temperature value.
    """
    print("Choose conversion type:")
    print("1. Celsius to Fahrenheit")
    print("2. Fahrenheit to Celsius")
    # Get conversion choice from user
    choice = input("Enter 1 or 2: ")
    if choice == "1":
        # Celsius to Fahrenheit conversion
        celsius = float(input("Enter temperature in Celsius: "))
        fahrenheit = (celsius * 9/5) + 32
        print(f"{celsius}C is equal to {fahrenheit}F")
        return fahrenheit
    elif choice == "2":
        # Fahrenheit to Celsius conversion
        fahrenheit = float(input("Enter temperature in Fahrenheit: "))
        celsius = (fahrenheit - 32) * 5/9
        print(f"{fahrenheit}F is equal to {celsius}C")
        return celsius
    else:
        print("Invalid choice. Please enter 1 or 2.")
        return None

# Call the function
temperature_converter()
```

```
Choose conversion type:
1. Celsius to Fahrenheit
2. Fahrenheit to Celsius
Enter 1 or 2: 1
Enter temperature in Celsius: 20
20.0C is equal to 68.0F
68.0
```

5.5 Global and Local Variables:

1. Gloabl Variables:

A **global variable** is defined outside of any function and is accessible from any part ofthe program, including within functions. Global variables maintain their values throughoutthe program's execution and can be accessed or modified by any function unless explicitlydeclared asnonlocalor redefined within the function.

Global Varaible

```
x = 10 # Global Variable.
def print_global():
    print(x) # Accessing global variable
print_global() # Output: 10
```

```
10
```

In this example, the variablexis global and can be accessed inside the print_global() function.

2. Local Variables:

A local variableis defined within a function and can only be accessed within that function.It exists only during the function's execution, and once the function completes, the localvariable is removed from memory.

Global Varaible

```
def print_local():
    y = 5 # Local variable
    print(y)

print_local() # Output: 5
print(y) # This would cause an error because y is not accessible outside the function
```

```
5
10
```

Here,y is a local variable within print_local()and cannot be accessed outside the function.

3. Accessing and Modifying Global Variables Inside a Function

To modify a global variable within a function, we must use theglobalkeyword. Otherwise,assigning a new value to a global variable within a function will create a new local variablewith the same name,

leaving the global variable unchanged.

Global Varaible

```
x = 10 # Global variable
def modify_global():
    global x # Declaring x as global
    x = 20 # Modifying global x

modify_global()
print(x) # Output: 20
```

```
20
```

Using global here allows the function to modify the global variablexdirectly.

6 Exception and Error Handling.

6.1 To - Summarize:

> • Syntax Errors:occur due to incorrect code structure and are identified before execution.

> • Exceptions:occur at runtime and can be handled usingtry-exceptblocks.

> • Thefinallyblock ensures that essential cleanup code is run, regardless of exceptions.

> • Custom exceptions can be created withraiseto handle specific scenarios.

Understanding and implementing exception handling is essential for writing resilient codethat can gracefully handle unexpected situations.

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 onemonth 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
```

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.

```
cold = [temp for temp in temperatures if temp < 10]
mild = [temp for temp in temperatures if temp > 10 and temp < 15]
comfortable = [temp for temp in temperatures if temp > 15 and temp < 20]
print("Cold: ", cold,"\nMild: ", mild, "\nComfortable: ", comfortable)
```

```
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
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, 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?

```
mild_count = 0
for m_count in mild:
    mild_count += 1
print("Mild Count: ", mild_count)

comfortable_count = 0
for c_count in comfortable:
    comfortable_count += 1
print("Comfortable Count: ", comfortable_count)

cold_count = 0
for co_count in cold:
    cold_count +=  1
print("Cold Count: ", cold_count)
```

```
Mild Count:  16
Comfortable Count:  16
Cold Count:  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 calledtemperatures_fahrenheit. Formula: Fahrenheit= (Celsius×95) + 321 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.

```
fahrenheit_list = []
for temp in temperatures:
    fahrenheit = (temp * (9/5)) + 32
    formated_fahrenheit = round(fahrenheit, 2)
    fahrenheit_list.append(formated_fahrenheit)

print("Fahrenheit: ", fahrenheit_list)
```

```
Fahrenheit:  [46.76, 63.32, 57.38, 46.22, 64.4, 56.3, 48.2, 64.04, 55.4, 47.3, 61.7, 55.
```

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 ontheir position.
3. Calculate and print the average day-time temperature.
4. (Optional)Plot ”day vs. temperature” using matplotlib.

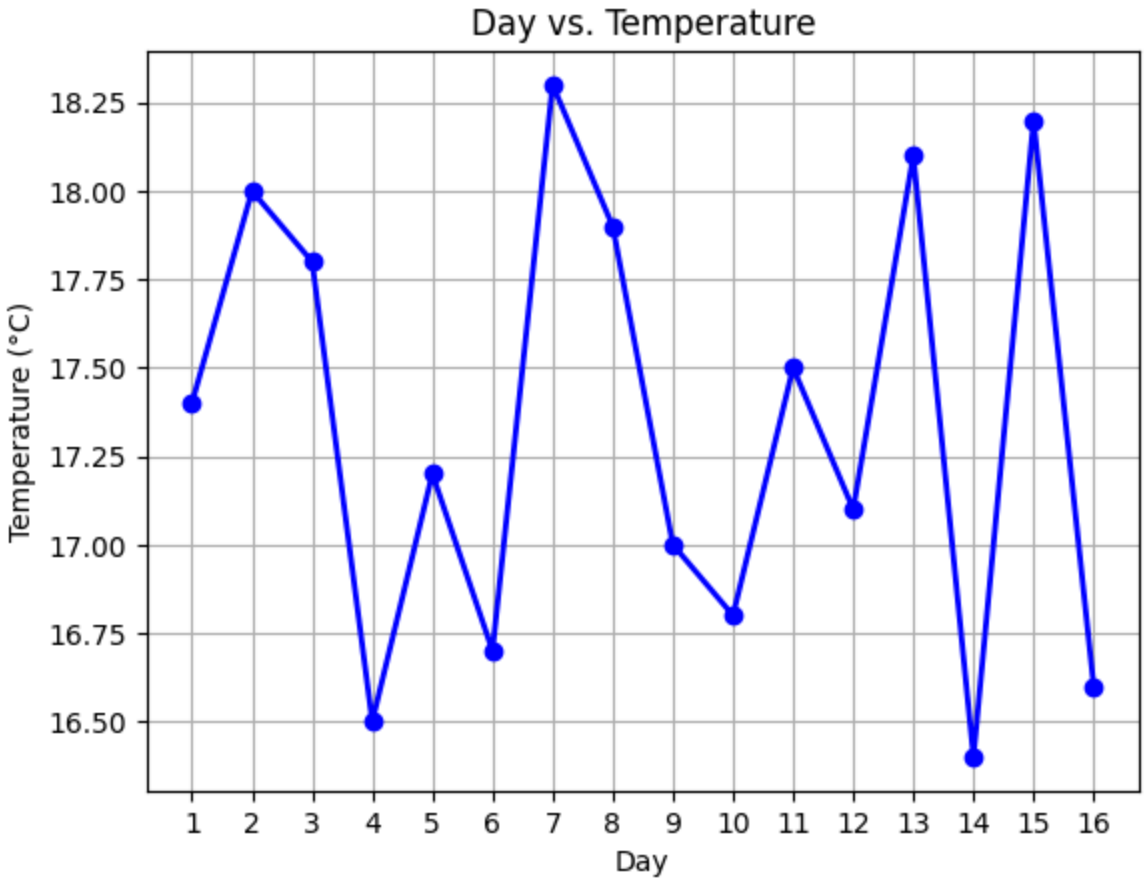
```
night = [night for night in temperatures if night > 0 and night < 8]
evening = [evening for evening in temperatures if evening > 8 and evening < 16]
day = [day for day in temperatures if day > 16 and day < 24]
print("Night: ", night ,"\nEvening: ", evening, "\nDay: ", day)
```

```
Night:  [7.9, 7.7, 7.6, 7.8, 7.8]
Evening:  [8.2, 14.1, 13.5, 9.0, 13.0, 8.5, 12.9, 13.3, 8.4, 14.0, 9.5, 13.4, 8.1, 14.2,
Day:  [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.
```

```
day_temp = 0
count = 0
for temp in day:
    day_temp += temp
    count += 1
average = day_temp/count
print(f"Average day temperature: {average:.2f}")
```

```
Average day temperature: 17.34
```

```
import matplotlib.pyplot as plt

# Provided data: temperatures for each day
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

# Create a list for the days, assuming these are sequential days (e.g., Day 1, Day 2, ..., D
days = list(range(1, 17))  # 16 days in total

# Plotting the data
plt.plot(days, temperatures, marker='o', color='b', linestyle='-', linewidth=2, markersize=6

# Adding title and labels
plt.title('Day vs. Temperature')
plt.xlabel('Day')
plt.ylabel('Temperature (°C)')

# Show the plot
plt.grid(True)
plt.xticks(days)  # Set x-ticks to show each day number
plt.show()
```



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 functionsum_nested_list(nested_list)that:

1. Takes a nested list (a list that can contain numbers or other lists of numbers) asinput.
2. Sums all numbers at every depth level of the list, regardless of how deeply nestedthe 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.

```
def sum_nested_list(nested_list):
    """Calculate the sum of all numbers in a nested list.This function takes a list that may
    int: The total sum of all integers in the nested list, including those in sublists.Examp
    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

    nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
    print(sum_nested_list(nested_list))
```

```
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”.

```
def generate_permutations(s):
    if len(s) == 1:
        return [s]

    result = []
    for i in range(len(s)):
        remaining = s[:i] + s[i+1:]
        for perm in generate_permutations(remaining):
            result.append(s[i] + perm)

    return result

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

```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

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.

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

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

calculate_directory_size(directory_structure)
```

```
1400
```

```
test_directory = {
    "document1.pdf": 500,
    "document2.docx": 750,
    "project": {
        "report.txt": 300,
        "data": {
            "data1.csv": 400,
            "data2.csv": 600,
            "raw": {
                "raw1.log": 100,
                "raw2.log": 200
            }
        },
        "summary.pdf": 700
```

```
        },
        "photos": {
            "photo1.jpg": 1200,
            "photo2.jpg": 950,
            "albums": {
                "vacation": {
                    "beach.png": 800,
                    "mountains.png": 900
                },
                "family": {
                    "birthday.jpg": 700,
                    "wedding.jpg": 1100
                }
            }
        },
        "readme.txt": 100
    }
    calculate_directory_size(test_directory)
```

```
9300
```

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]).

```
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

min_coins([1,2,5], 11)
```

```
3
```

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):
    new_list = []
    s1_list = [s for s in s1]
    s2_list = [s for s in s2]
    for i in s1_list:
        for j in s2_list:
            if i == j:
                new_list.append(i)
            else:
                continue

    length = len(new_list)
    string = ''
    print(f"The LCS of two string {s1} and {s2} is: ({string.join(new_list)}) with length {]

longest_common_subsequence("abcde", "ace");
```

```
The LCS of two string abcde and ace is: (ace) with length 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 result should be 9.

```
def knapsack(weights, values, capacity):
    sum_of_weight = 0
    answer = 0
    for i in weights:
        for j in weights:
            i_index = weights.index(i)
            j_index = weights.index(j)
            if i + j == capacity:
                return values[i_index]+values[j_index]
            elif i + j <= capacity:
                if i + j > sum_of_weight:
                    answer = values[i_index] + values[j_index]
                else:
                    continue
            else:
                continue
    return answer
```

```
weight = [1, 3, 4, 5]
value = [1, 4, 5, 7]
capacity = 7
print(knapsack(weight, value, capacity))
```