Name : **Daraniya Neel**

Python – Collections, functions and Modules

1. Accessing List

- Understanding how to create and access elements in a list.

➢ Creating a List

A list is a collection of items that are ordered and mutable.
You can create it using square brackets [].

```
# Example of list creation
my_list = [10, 20, 30, 40, 50]
print(my_list)
```

➢ Accessing Elements

You can access list elements using indexing or slicing.

a. Indexing

Each element in a list has an index starting from 0.
Use the index in square brackets [] to access elements.

```
print(my_list[0])  # Output: 10 (First element)
```

b. Slicing

Slicing allows you to access a range of elements using start:end.

```
print(my_list[1:4])  # Output: [20, 30, 40]
```

- Indexing in lists (positive and negative indexing).

➢ Positive Indexing

Positive indexing starts from 0 for the first element and increases by 1 for each subsequent element.

```
# A sample list
my_list = ['a', 'b', 'c', 'd', 'e']

# Accessing elements with positive indexing
print(my_list[0])  # Output: 'a' (First element)
print(my_list[2])  # Output: 'c' (Third element)
print(my_list[4])  # Output: 'e' (Fifth element)
```

- Negative Indexing

  Negative indexing starts from -1 for the last element and decreases as you move left.

```
# Accessing elements with negative indexing
print(my_list[-1])  # Output: 'e' (Last element)
print(my_list[-2])  # Output: 'd' (Second to last element)
print(my_list[-5])  # Output: 'a' (First element)
```

• Slicing a list: accessing a range of elements.

- General Syntax

```
list[start:stop:step]
```

start (optional): The index where the slice starts (inclusive). Default is 0 if omitted.
stop: The index where the slice ends (exclusive). Required unless only the step is specified.
step (optional): The increment between each index in the slice. Default is 1 if omitted.

- Basic Slicing

```
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Access elements from index 2 to 5 (excluding 5)
print(numbers[2:5])  # Output: [30, 40, 50]
```

- Omitting start or stop

```
# From the start to index 4
print(numbers[:4])  # Output: [10, 20, 30, 40]

# From index 6 to the end
print(numbers[6:])  # Output: [70, 80, 90, 100]
```

- Using a step

```
# Every second element from index 1 to 7
print(numbers[1:8:2])  # Output: [20, 40, 60, 80]

# Reverse the list
print(numbers[::-1])  # Output: [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

- Negative Indices

```
# Slice using negative indices
print(numbers[-5:-2])  # Output: [60, 70, 80]
```

2. List Operations

- Common list operations: concatenation, repetition, membership.

- Concatenation (+)

Combines two or more lists into a single list.

```
# Example lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Concatenating lists
result = list1 + list2
print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

- Repetition (*)

Repeats a list a specified number of times.

```
# Example list
list1 = [1, 2, 3]

# Repeating the list 3 times
result = list1 * 3
print(result)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- Membership (in and not in)

Tests whether an element exists in the list.

in

Checks if an element is present in the list.

```
# Example list
fruits = ["apple", "banana", "cherry"]

# Checking membership
print("apple" in fruits)  # Output: True
print("orange" in fruits)  # Output: False
```

- Understanding list methods like append(), insert(), remove(), pop().

➢ append()

Adds a single element to the end of the list.

Syntax:

```
list.append(element)
```

Example:

```
# Initial list
numbers = [1, 2, 3]

# Adding an element
numbers.append(4)
print(numbers)  # Output: [1, 2, 3, 4]
```

➢ insert()

Inserts an element at a specified position in the list.

Syntax:

```
list.insert(index, element)
```

Example:

```
# Initial list
numbers = [1, 2, 3]

# Inserting at index 1
numbers.insert(1, 10)
print(numbers)  # Output: [1, 10, 2, 3]
```

```
# Inserting at a position beyond the list's length
numbers.insert(10, 5)
print(numbers)  # Output: [1, 10, 2, 3, 5]
```

- remove()

  Removes the first occurrence of the specified element from the list.

  Syntax:

  ```
  list.remove(element)
  ```

  Example:

  ```
  # Initial list
  fruits = ["apple", "banana", "cherry", "apple"]

  # Removing the first "apple"
  fruits.remove("apple")
  print(fruits)  # Output: ["banana", "cherry", "apple"]
  ```

- pop()

  Removes and returns the element at a specified position. If no index is specified, it removes the last element.

  Syntax:

  ```
  list.pop(index)
  ```

  Example:

  ```
  # Initial list
  numbers = [1, 2, 3, 4]

  # Removing the last element
  last_item = numbers.pop()
  print(last_item)  # Output: 4
  print(numbers)    # Output: [1, 2, 3]

  # Removing an element by index
  second_item = numbers.pop(1)
  print(second_item)  # Output: 2
  print(numbers)      # Output: [1, 3]
  ```

3. Working with Lists

- Iterating over a list using loops.

➢ Using a for Loop

The most common way to iterate over a list is with a for loop.

Syntax:

```
for element in list:
 # Perform actions with element
```

Example:

```
numbers = [10, 20, 30, 40]

for num in numbers:
        print(num)
```

➢ Using a while Loop

A while loop can iterate over a list using an index counter.

Syntax:

```
index = 0
while index < len(list):
        # Access list[index]
        index += 1
```

Example:

```
numbers = [10, 20, 30, 40]

index = 0
while index < len(numbers):
        print(numbers[index])
        index += 1
```

➢ Iterating with a Condition

You can filter elements during iteration.

Example:

```
numbers = [10, 15, 20, 25, 30]
```

```
# Print only even numbers
for num in numbers:
    if num % 2 == 0:
        print(num)
```

- Sorting and reversing a list using sort(), sorted(), and reverse().

➢ sort()

The sort() method modifies the list in place and sorts it in ascending order by default.

Syntax:

```
list.sort(key=None, reverse=False)
```

key: A function that specifies a sort criterion. Default is None.
reverse: If True, sorts the list in descending order.

➢ sorted()

The sorted() function returns a new sorted list and does not modify the original list.

Syntax:

```
sorted(iterable, key=None, reverse=False)
```

Same parameters as sort(), but works on any iterable, not just lists.

➢ reverse()

The reverse() method reverses the list in place without sorting it.

Syntax:

```
list.reverse()
```

- Basic list manipulations: addition, deletion, updating, and slicing.

➢ Addition

You can add elements to a list using append(), extend(), insert(), or the + operator.

Appending an Element (append):

Adds a single element to the end of the list.

Adding Multiple Elements (extend):

Combines two lists by adding elements from another list to the end.

Inserting at a Specific Position (insert):

Inserts an element at a specified index.

Concatenating Lists (+):

Creates a new list by combining two lists.

- Deletion

You can delete elements using remove(), pop(), del, or clear().

Removing by Value (remove):

Removes the first occurrence of a specified element.

Removing by Index (pop):

Removes and returns the element at a specified index. If no index is provided, removes the last element.

Clearing the Entire List (clear): Removes all elements from the list.

- Updating

You can update individual elements or slices of a list.

Updating an Element by Index:

Assign a new value to a specific index.

Updating a Slice:

Replace multiple elements with new values.

- Slicing

Slicing allows you to access specific ranges of elements in a list.

Basic Slicing

Omitting Start or End

Using Step

4. Tuple

- ## Introduction to tuples, immutability.

- A tuple in Python is an ordered, immutable collection of elements.
- It is similar to a list but differs in one key aspect: tuples cannot be modified after they are created.
- Tuples are typically used to represent collections of related items that should not change, such as geographic coordinates, RGB color values, or fixed configurations.

- Creating a Tuple

  Tuples are created by placing elements inside parentheses () and separating them with commas.

  ```
  empty_tuple = ()
  ```

- Why Tuples Are Immutable

  Immutability means that the tuple's content cannot be changed after it is created.

  This ensures:

  Data Safety: Protects data integrity by preventing accidental modification.
  Hashability: Tuples can be used as dictionary keys or elements in a set because their content is fixed and hashable.

- ## Creating and accessing elements in a tuple.

- Creating a Tuple

  To create a tuple, you place the elements inside parentheses () and separate them with commas.
  Tuples can contain any type of data: integers, strings, booleans, lists, and even other tuples.

  ```
  my_tuple = (10, 20, 30)
  print(my_tuple)
  ```

- Accessing Elements in a Tuple

  You can access elements in a tuple using indexing and slicing, similar to lists.

  Accessing by Index:

  Indexes in Python start at 0.

So, to access the first element of the tuple, you use index 0, the second element is accessed with index 1, and so on.

Slicing a Tuple:

You can extract a range of elements using slicing. The syntax is tuple[start:end], where:

start is the index to begin from (inclusive).

end is the index to end at (exclusive).

- Basic operations with tuples: concatenation, repetition, membership.

➢ Concatenation

Concatenation allows you to combine two or more tuples into a single tuple using the + operator.

Example:

```
# Two tuples
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenate the tuples
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)
```

➢ Repetition

Repetition allows you to repeat a tuple multiple times using the * operator. This is useful for creating a tuple with repeated elements.

Example:

```
# A single tuple
tuple1 = (1, 2, 3)

# Repeating the tuple 3 times
repeated_tuple = tuple1 * 3
print(repeated_tuple)  # Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

➢ Membership Testing

You can test if an element exists in a tuple using the in operator, which returns True if the element is found in the tuple and False otherwise.

Example:

```
# A tuple
tuple1 = (1, 2, 3, 4, 5)

# Check if an element is in the tuple
print(3 in tuple1)  # Output: True
print(6 in tuple1)  # Output: False
```

5. Accessing Tuples

- Accessing tuple elements using positive and negative indexing.

➢ Positive Indexing:

Positive indexing starts from 0 for the first element, 1 for the second, and so on.

Example:

```
# Defining a tuple
my_tuple = (10, 20, 30, 40, 50)

# Accessing elements using positive indexing
print(my_tuple[0])  # Output: 10
print(my_tuple[1])  # Output: 20
print(my_tuple[2])  # Output: 30
```

➢ Negative Indexing:

Negative indexing starts from -1 for the last element, -2 for the second to last, and so on.

Example:

```
# Accessing elements using negative indexing
print(my_tuple[-1])  # Output: 50
print(my_tuple[-2])  # Output: 40
print(my_tuple[-3])  # Output: 30
```

- Slicing a tuple to access ranges of elements.

➢ Slicing allows you to extract a portion of the tuple by specifying a start index, end index, and an optional step value.

➢ Syntax:

```
tuple[start:end:step]

start: The index from which the slice begins (inclusive).
end: The index where the slice ends (exclusive).
step: The step size or interval between elements (optional).
```

➢ Example:

```
# Defining a tuple
my_tuple = (10, 20, 30, 40, 50, 60, 70)

# Basic slicing (start: end)
print(my_tuple[1:5])  # Output: (20, 30, 40, 50)
# Explanation: Starts from index 1 (inclusive) and ends at index 5 (exclusive).

# Slicing with step
print(my_tuple[::2])  # Output: (10, 30, 50, 70)
# Explanation: Starts from the beginning, takes every second element (step = 2).

# Slicing with negative indexing
print(my_tuple[-5:-2])  # Output: (30, 40, 50)
# Explanation: Starts from index -5 (inclusive) and ends at index -2 (exclusive).

# Slicing with start, end, and step
print(my_tuple[1:6:2])  # Output: (20, 40, 60)
# Explanation: Starts from index 1, ends before index 6, taking every second
element.
```

6. Dictionaries

- Introduction to dictionaries: key-value pairs.

- A dictionary in Python is a collection of key-value pairs, where each key is unique, and the key maps to a specific value.
- Dictionaries are unordered, mutable, and allow for fast lookups of values based on keys.
- They are one of the most important data structures in Python.

- Syntax:

  A dictionary is defined using curly braces {} with key-value pairs, separated by a colon :.

```
my_dict = {
   'key1': 'value1',
   'key2': 'value2',
   'key3': 'value3'
}
```

- Key-Value Pairs:

  Key: A unique identifier used to access a value. Keys must be immutable (e.g., strings, numbers, tuples).

  Value: The data associated with a key. Values can be of any data type (strings, lists, tuples, numbers, other dictionaries, etc.).

  Example:

```
person = {
        'name': 'Alice',
        'age': 30,
        'city': 'New York'
}
```

• Accessing, adding, updating, and deleting dictionary elements.

- Accessing Dictionary Elements

  You can access the values in a dictionary using their corresponding keys.

  Using Square Brackets: You can access a value by referencing its key inside square brackets [].

  Using get() Method: The get() method is another way to access values. It is safer than using square brackets because it does not raise an error if the key does not exist. Instead, it returns None (or a default value if provided).

- Adding or Updating Dictionary Elements

  You can add new key-value pairs or update the value of an existing key.
  Adding or Updating Using Square Brackets: If the key does not exist, a new key-value pair is added. If the key exists, the value is updated.

  Using update() Method: The update() method allows you to add multiple key-value pairs or update existing ones at once. You can pass a dictionary or key-value pairs.

- Deleting Dictionary Elements

  You can remove key-value pairs using different methods.

  Using del Statement: The del statement removes a key-value pair by specifying the key. It will raise a KeyError if the key does not exist.

  Using pop() Method: The pop() method removes a key-value pair and returns the value associated with the key. If the key does not exist, you can provide a default value to avoid an error.

  Using popitem() Method: The popitem() method removes and returns the last key-value pair as a tuple. If the dictionary is empty, it raises a KeyError.

  Using clear() Method: The clear() method removes all the key-value pairs from the dictionary.

• Dictionary methods like keys(), values(), and items().

- keys()

  The keys() method returns a view object containing all the keys in the dictionary. The keys are unique and can be iterated over.

  Syntax:

  ```
  dictionary.keys()
  ```

  Example:

  ```
  my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

  # Get all keys
  keys = my_dict.keys()
  print(keys)  # Output: dict_keys(['name', 'age', 'city'])
  ```

```
# Iterate over keys
for key in keys:
    print(key)
```

- values()

The values() method returns a view object containing all the values in the dictionary. The values can be of any data type and are not necessarily unique.

Syntax:

```
dictionary.values()
```

Example:

```
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Get all values
values = my_dict.values()
print(values)  # Output: dict_values(['Alice', 30, 'New York'])

# Iterate over values
for value in values:
    print(value)
```

- items()

The items() method returns a view object containing all the key-value pairs as tuples. Each tuple is in the format (key, value).

Syntax:

```
dictionary.items()
```

Example:

```
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Get all key-value pairs
items = my_dict.items()
print(items)  # Output: dict_items([('name', 'Alice'), ('age', 30), ('city',
'NewYork')])

# Iterate over key-value pairs
for key, value in items:
    print(f"{key}: {value}")
```

7. Working with Dictionaries

- Iterating over a dictionary using loops.

- Dictionaries in Python allow you to iterate through their keys, values, or key-value pairs using loops.
- This is a common operation when processing the data stored in a dictionary.

- Iterating Over Keys:
  You can iterate through the keys of a dictionary using a for loop.
  By default, looping over a dictionary directly gives you its keys.

- Iterating Over Values:
  To loop through the values of a dictionary, use the values() method.

  Example:

```
# Loop through values
for value in my_dict.values():
    print(value)
```

- Iterating Over Key-Value Pairs:
  To loop through both keys and values together, use the items() method.
  This returns each key-value pair as a tuple.

  Example:

```
# Loop through key-value pairs
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

- Using Dictionary Comprehensions:
  You can also iterate over a dictionary while performing transformations using dictionary comprehensions.

  Example:

```
# Example: Doubling the values in a dictionary (if numeric)
numbers = {'a': 1, 'b': 2, 'c': 3}
doubled = {key: value * 2 for key, value in numbers.items()}
print(doubled)
# Output: {'a': 2, 'b': 4, 'c': 6}
```

- ## Merging two lists into a dictionary using loops or zip().

- Two lists can be combined into a dictionary where one list serves as keys and the other as values.
- Below are methods to accomplish this using loops or the zip() function.

- Using a Loop:

  You can use a loop to pair each element from one list (keys) with the corresponding element from another list (values).

  Example:

```
keys = ['name', 'age', 'city']
values = ['Alice', 30, 'New York']

# Merging using a loop
merged_dict = {}
for i in range(len(keys)):
    merged_dict[keys[i]] = values[i]

print(merged_dict)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

- Using zip():

  The zip() function is a more Pythonic way to merge two lists into a dictionary.
  It pairs elements from both lists and creates an iterable of tuples.

  Example:

```
keys = ['name', 'age', 'city']
values = ['Alice', 30, 'New York']

# Merging using zip()
merged_dict = dict(zip(keys, values))

print(merged_dict)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

- ## Counting occurrences of characters in a string using dictionaries.

- Dictionaries are an ideal data structure for counting the occurrences of characters in a string, as they allow you to store each character as a key and its count as the corresponding value.

- Using a Loop:

  You can iterate through each character in the string and update the dictionary accordingly.

  Example:

```
# String to analyze
text = "hello world"

# Dictionary to store counts
char_count = {}

# Loop through each character in the string
for char in text:
    if char in char_count:
        char_count[char] += 1  # Increment the count if char exists
    else:
        char_count[char] = 1   # Initialize count for new char

print(char_count)
# Output: {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

- Using get() Method:

  The get() method simplifies handling the case where a key might not exist in the dictionary.

  Example:

```
text = "hello world"
char_count = {}

for char in text:
    char_count[char] = char_count.get(char, 0) + 1

print(char_count)
# Output: {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

- Using collections.Counter:

The collections.Counter class from Python's collections module is designed specifically for counting occurrences.

Example:

```
from collections import Counter

text = "hello world"
char_count = Counter(text)

print(char_count)
# Output: Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

8. Functions

- Defining functions in Python.

- Functions in Python are reusable blocks of code that perform a specific task.
- They help organize code, avoid repetition, and improve readability.

- Syntax:

```
def function_name(parameters):
    """docstring (optional): Describe the function."""
    # Function body
    return value  # Optional: Specifies the value to return
```

`def`: The keyword to define a function.
`function_name`: The name of the function. It should be descriptive and follow naming conventions (e.g., snake_case).
`parameters`: Optional inputs for the function, specified in parentheses. Multiple parameters are separated by commas.
`return`: Optional keyword to specify the output of the function.

- Example:

```
# Function with no parameters
def greet():
    """This function prints a greeting message."""
    print("Hello, World!")
greet()
# Output: Hello, World!
```

- Different types of functions: with/without parameters, with/without return values.

- Python functions can be categorized based on whether they accept parameters and/or return values.

- Functions Without Parameters and Without Return Values:

  These functions neither take input parameters nor return any value.
  They perform their task and exit.

  Example:

```
def greet():
    """This function prints a greeting message."""
    print("Hello, World!")

greet()
```

- Functions With Parameters and Without Return Values:

  These functions accept parameters as input but do not return any value.
  They only perform an action.

  Example:

```
def greet(name):
    """This function greets the user by name."""
    print(f"Hello, {name}!")

greet("Alice")
```

- Functions Without Parameters and With Return Values:

  These functions do not require input but return a value after performing their task.

  Example:

```
def get_greeting():
    """This function returns a greeting message."""
    return "Hello, World!"

message = get_greeting()
print(message)
```

- Functions With Parameters and With Return Values:

These functions accept input parameters and return a value as output after performing their task.

Example:

```
def add(a, b):
    """This function adds two numbers and returns the result."""
    return a + b

result = add(5, 3)
print(result)
```

- Anonymous functions (lambda functions).

- Anonymous functions in Python are defined using the lambda keyword.
- These are also known as lambda functions and are typically used for small, simple operations where defining a full function is unnecessary.

- Syntax:

```
lambda arguments: expression
```

`lambda`: The keyword used to define the function.
`arguments`: The input(s) to the function (can be zero or more).
`expression`: A single expression that is evaluated and returned.

- Examples:

```
add = lambda x, y: x + y

print(add(5, 3))
```

Here:

x and y are parameters.
x + y is the expression being evaluated and returned.

9. Modules

- Introduction to Python modules and importing modules.

- In Python, a module is a file containing Python code that can include functions, classes, and variables.
- Modules help organize and reuse code across multiple programs, making them an essential part of Python programming.

- What is a Module?

  A module is simply a Python file (.py) containing code that can be imported and used in other Python scripts.

  It can include:

  Functions
  Classes
  Variables
  Executable code

- Importing Modules:

  You can import a module using the import keyword.

  Syntax:

  ```
  import module_name
  ```

  Examples:

  ```
  import math

  # Using a function from the math module
  print(math.sqrt(16))  # Output: 4.0
  ```

• Standard library modules: math, random.

- math Module

  The math module provides mathematical functions and constants.

  Importing the Module:

  ```
  import math
  ```

  Mathematical Constants:

Constant	Description	Example
math.pi	The value of π (pi).	math.pi → 3.14159
math.e	The base of the natural logarithm.	math.e → 2.71828

Common Functions in math:

Function	Description	Example
math.sqrt(x)	Returns the square root of x.	math.sqrt(16) → 4.0
math.pow(x, y)	Returns x raised to the power of y.	math.pow(2, 3) → 8.0
math.floor(x)	Returns the largest integer ≤ x.	math.floor(4.7) → 4
math.ceil(x)	Returns the smallest integer ≥ x.	math.ceil(4.3) → 5
math.factorial(x)	Returns the factorial of x (non-negative only).	math.factorial(5) → 120

- random Module

  The random module is used to generate random numbers and perform random operations.

  Importing the Module:

  ```
  import random
  ```

  Common Functions in random:

Function	Description	Example
random.random()	Returns a random float between 0.0 and 1.0.	random.random() → 0.5728
random.randint(a, b)	Returns a random integer between a and b (inclusive).	random.randint(1, 10) → 7
random.uniform(a, b)	Returns a random float between a and b.	random.uniform(1.5, 10.5) → 4.75
random.choice(seq)	Returns a random element from a sequence.	random.choice([1, 2, 3]) → 2
random.choices(pop, k)	Returns a list of k random elements from a population.	random.choices([1, 2, 3+, k=2) → *3, 1+

• Creating custom modules.

- In Python, you can create your own custom modules to organize reusable code into a separate file.
- This allows you to maintain a clean structure and promotes code reuse across multiple programs.

- Create a Python File (Module):

A custom module is simply a Python file (.py) containing code such as functions, classes, or variables.

Example: Create a file named my_module.py with the following code:

```
# my_module.py

# A function to greet a user
def greet(name):
    return f"Hello, {name}!"

# A function to calculate the square of a number
def square(num):
    return num ** 2

# A variable
PI = 3.14159
```

- Import the Custom Module:

  You can import your custom module into another Python script using the import statement.

  Example: Create a Python script named main.py:

```
# main.py
import my_module  # Importing the custom module

# Using the module's functions and variables
print(my_module.greet("Alice"))  # Output: Hello, Alice!
print(my_module.square(5))       # Output: 25
print("Value of PI:", my_module.PI)  # Output: Value of PI: 3.14159
```

- Rename the Module or Components (Using as):

  You can use aliases to rename a module or its components for convenience.

```
# Renaming the module
import my_module as mm

print(mm.greet("Charlie"))  # Output: Hello, Charlie!

# Renaming a function
from my_module import square as sq
print(sq(4))  # Output: 16
```