**Python – Collections, functions and Modules**

**Accessing List:**

* Understanding how to create and access elements in a list.
* lists are a common way to store multiple items in a single variable.

Creating a list:

my\_list = [10, 20, 30, 40]

Accessing elements:

By index (starting from 0):

print(my\_list[0]) # Output: 10

print(my\_list[2]) # Output: 30

Negative indexing (accessing from the end):

print(my\_list[-1]) # Output: 40

print(my\_list[-3]) # Output: 20

* Indexing in lists (positive and negative indexing).
* ndexing in lists allows you to access individual elements using their position within the list. Indexing can be done using **positive** or **negative** indices.

1. Positive Indexing:

Starts from **0** for the first element and increases by 1.

The indices range from 0 to len(list) - 1.

2. Negative Indexing:

Starts from **-1** for the last element and decreases by 1 as you move left.

The indices range from -1 to -len(list).

* Slicing a list: accessing a range of elements.
* Slicing allows you to extract a subset of elements from a list by specifying a **start**, **stop**, and optionally a **step**.

Syntax:

list[start:stop:step]

2. List Operations

* Common list operations: concatenation, repetition, membership
* 1. Concatenation (+):Concatenation joins two or more lists into a single list using the + operator.
* 2. Repetition (\*):Repetition creates a new list by repeating the elements of an existing list a specified number of times using the \* operator.
* 3. Membership (in and not in): Membership checks if a specific element exists in a list using the in or not in operators.
* Understanding list methods like append(), insert(), remove(), pop().
* Python provides several useful methods for manipulating lists. Here’s an explanation of four common list methods: append(), insert(), remove(), and pop(). Each of these methods allows you to modify the list in different ways.

**1. append()**

* **Purpose**: Adds a single element to the end of the list.
* **Syntax**: list.append(element)
* **Returns**: None (it modifies the list in place).

### ****2.**** insert()

* **Purpose**: Inserts an element at a specific index in the list.
* **Syntax**: list.insert(index, element)
* **Returns**: None (it modifies the list in place).

### ****3.**** remove()

* **Purpose**: Removes the first occurrence of a specified element from the list.
* **Syntax**: list.remove(element)
* **Returns**: None (it modifies the list in place).
* **Raises**: ValueError if the element is not found in the list.

### ****4.**** pop()

* **Purpose**: Removes and returns the element at a specified index. If no index is provided, it removes and returns the last element.
* **Syntax**: list.pop(index) or list.pop()
* **Returns**: The removed element.
* **Raises**: IndexError if the list is empty or the index is out of range.

3. Working with Lists

* Iterating over a list using loops.
* Iterating over a list is a common task in programming and can be done in several ways depending on the language you're using.

Example :🡪

# Using a for loop

my\_list = [1, 2, 3, 4, 5]

for item in my\_list:

print(item)

* Sorting and reversing a list using sort(), sorted(), and reverse().
* Sorting and reversing lists are common operations in programming. Here's how you can use methods like sort(), sorted(), and reverse() in Python.

### ****1. Using**** sort()

* sort() is an **in-place** method. It modifies the list itself.
* By default, it sorts in **ascending order**.

### ****2. Using**** sorted()

* sorted() creates a **new list** and leaves the original list unmodified.
* It can sort in **ascending** or **descending** order using the reverse argument.

### ****3. Using**** reverse()

* reverse() is an **in-place** method. It reverses the elements of the list.
* It does not sort the list, only reverses the current order.
* Basic list manipulations: addition, deletion, updating, and slicing.
* Addition: Adding elements to a list can be done using various methods: append(), extend(),

insert()

* Deletion: Removing elements from a list can be done using the following methods: remove(), pop(), clear(),del Statement
* Updating: Modifying the value of an existing element
* Slicing : Extracting a subset of elements from the list

4. Tuple

* Introduction to tuples, immutability.
* A **tuple** is a built-in data structure in Python used to store a collection of elements. It is similar to a list but has an important property: **immutability**.

**Immutability:**  Once a tuple is created, its elements cannot be changed, added, or removed, This makes tuples ideal for data that should not change throughout a program.

* Creating and accessing elements in a tuple
* A **tuple** is a Python data structure that is immutable and ordered. Here's a guide to creating and accessing elements in a tuple.

Accessing Elements in a Tuple

Access by Index: Elements in a tuple can be accessed using their index (starting at 0).

Access Using Slicing: Slicing allows extracting a subset of the tuple.

* Basic operations with tuples: concatenation, repetition, membership
* Tuples in Python support several operations, including **concatenation**, **repetition**, and **membership testing**.

Concatenation: Concatenation combines two or more tuples to create a new tuple.

**Note**: Since tuples are immutable, a new tuple is created during concatenation.

Repetition: Repetition creates a new tuple by repeating the elements of an existing tuple a specified number of times.

**Note**: A new tuple is created; the original tuple remains unchanged.

Membership Testing: Membership testing checks whether an element is present in a tuple using the in or not in operators.

5. Accessing Tuples

* Accessing tuple elements using positive and negative indexing
* tuples are ordered collections, and you can access their elements using both **positive** and **negative** indexing.

Positive Indexing: Positive indexing starts from 0 for the first element and increments by 1 for each subsequent element.

Syntax:

tuple[index]

Negative Indexing: Negative indexing allows you to access the elements of the tuple starting from the last element, with -1 representing the last element, -2 representing the second last element, and so on.

Syntax:

tuple[-index]

* Slicing a tuple to access ranges of elements.
* tuples can be sliced to access a range of elements. Slicing is a powerful feature that allows you to extract a portion (or subtuple) from the original tuple.

Syntax:

tuple[start:stop:step]

6. Dictionaries

* Introduction to dictionaries: key-value pairs.
* A **dictionary** is a collection of items that are unordered, changeable, and indexed. Each item in a dictionary is made up of two parts: a **key** and a **value**, which together form a **key-value pair**. Dictionaries are widely used for storing and managing data in a way that allows fast access through keys.
* Accessing, adding, updating, and deleting dictionary elements
* Dictionaries in Python provide a flexible way to store and manipulate data using key-value pairs.

Accessing Dictionary Elements: You can access the value associated with a specific key using square brackets [] or the get() method.

Adding Dictionary Elements: You can add a new key-value pair to the dictionary by assigning a value to a new key.

Updating Dictionary Elements: To update an existing key, assign a new value to the key.

Deleting Dictionary Elements: Dictionaries allow you to remove elements using the del keyword or the pop() method.

* Dictionary methods like keys(), values(), and items().
* dictionaries come with several built-in methods that allow you to interact with and manipulate the keys and values in a dictionary.

keys() Method: The keys() method returns a view object that displays a list of all the keys in the dictionary. You can iterate over this view or convert it to a list if needed.

Syntax:

dictionary.keys()

values() Method: The values() method returns a view object that displays all the values in the dictionary. Similar to keys(), you can iterate over this view or convert it to a list.

Syntax:

dictionary.values()

items() Method: The items() method returns a view object that displays a list of key-value tuple pairs in the dictionary. Each item is a tuple where the first element is the key and the second is the value.

Syntax:

dictionary.items()

7. Working with Dictionaries

* Iterating over a dictionary using loops
* We can iterate over a dictionary using loops, which allows you to access both the keys and values. There are several ways to iterate over a dictionary: using a for loop, keys(), values(), or items() methods.
* Merging two lists into a dictionary using loops or zip().
* You can merge two lists into a dictionary in Python by pairing elements from both lists using loops or the zip() function.

Merging Two Lists into a Dictionary Using a Loop: To merge two lists into a dictionary using a loop, you can iterate over both lists simultaneously using the range() function, which will let you access both list elements by index.

Merging Two Lists into a Dictionary Using zip():The zip() function pairs elements from two or more iterables (like lists) together and returns an iterator of tuples. You can then pass these pairs to the dict() constructor to create a dictionary.

* Counting occurrences of characters in a string using dictionaries
* You can count the occurrences of characters in a string using a dictionary in Python. The idea is to use the characters as keys and their frequencies as values.

Counting Occurrences of Characters in a String Using a Dictionary:-

#### ****Steps****:

1. **Initialize an empty dictionary** to store character counts.
2. **Iterate through each character** in the string.
3. For each character:
   * If it's already in the dictionary, **increment its count**.
   * If it's not in the dictionary, **add it with a count of 1**.
4. Print or return the dictionary with character counts.

8. Functions

* Defining functions in Python.
* Functions in Python are blocks of reusable code designed to perform a specific task. They help make programs modular, organized, and easier to maintain.
* Different types of functions: with/without parameters, with/without return values
* Functions can be classified based on whether they take parameters and whether they return values.

Functions Without Parameters and Without Return Values:

**Description**: These functions neither accept arguments nor return any values. They perform a task and exit.

Functions Without Parameters and With Return Values:

**Description**: These functions don’t take any arguments but return a value after execution.

Functions With Parameters and Without Return Values:

**Description**: These functions accept arguments to customize their behavior but don’t return any value.

Functions With Parameters and With Return Values:

**Description**: These functions accept arguments and also return a value.

* Anonymous functions (lambda functions).
* **Anonymous functions**, often referred to as **lambda functions**, are small, single-expression functions that are defined without a name. They are commonly used for short tasks where defining a full function would be overkill.

9. Modules

* Introduction to Python modules and importing modules
* a **module** is a file containing Python definitions and statements. Modules allow you to logically organize your Python code into separate files for better maintainability, reusability, and readability.

Importing Modules:

Python provides several ways to import and use modules:

Importing the Entire Module :-

import math

print(math.sqrt(16)) # Outputs: 4.0

Importing Specific Functions or Variables:

from math import sqrt, pi

print(sqrt(16)) # Outputs: 4.0

print(pi) # Outputs: 3.141592653589793

* Standard library modules: math, random.
* Python’s **standard library** includes many built-in modules for a variety of tasks. Two commonly used modules are **math** for mathematical operations and **random** for generating random numbers.

math Module: The math module provides mathematical functions like trigonometry, logarithms, and constants.

random Module: The random module provides tools for generating random numbers and performing random operations.

* Creating custom modules
* Creating custom modules in Python is a straightforward process. A **module** is simply a file containing Python code (functions, classes, or variables) that you can reuse in other Python programs.

#### ****Create a Python File (Your Module)****

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

Example: Create a file named math\_utils.py with the following content:

# math\_utils.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

def multiply(a, b):

return a \* b

def divide(a, b):

if b == 0:

return "Cannot divide by zero"

return a / b