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

1. Definition

- A list in Python is a collection that can store multiple items in a single variable.
- Lists are ordered, mutable, and allow duplicate values.
- They are written inside square brackets [ ].

2. Creating a List

Examples:

```
# List of numbers
numbers = [10, 20, 30, 40, 50]

# List of strings
fruits = ["apple", "banana", "cherry"]

# List with mixed data types
mixed = [25, "hello", 3.14, True]
```

3. Accessing Elements

- Using indexing:
  - Index starts from 0 for the first element.
  - Negative indexing (-1) is used for accessing from the end.

Example:

```
fruits = ["apple", "banana", "cherry", "date"]

print(fruits[0])
print(fruits[2])
print(fruits[-1])
```

2. Indexing in lists (positive and negative indexing).

1. Definition

- Indexing means accessing elements of a list using their position (index).
- In Python:
  - Indexing starts from 0 (first element).
  - Negative indexing starts from -1 (last element).

2. Positive Indexing

- Counts from **left to right** (starting at 0).

Example:

fruits = ["apple", "banana", "cherry", "date", "mango"]

print(fruits[0])
print(fruits[2])
print(fruits[4

3. Negative Indexing

- Counts from right to left (starting at -1).

Example:

fruits = ["apple", "banana", "cherry", "date", "mango"]

print(fruits[-1])
print(fruits[-2])
print(fruits[-5])

3. Slicing a list: accessing a range of elements.

1. Definition

- Slicing in Python allows us to extract a portion (sublist) of a list.
- Syntax:
- list[start:end]
    - start index → position where the slice begins (inclusive).
    - end index → position where the slice ends (exclusive).
    - If start or end is omitted, Python assumes beginning or end of the list.

2. Examples of Slicing

numbers = [10, 20, 30, 40, 50, 60, 70]

print(numbers[1:4])

print(numbers[:3])

print(numbers[2:])
print(numbers[:])

3. Negative Index Slicing

numbers = [10, 20, 30, 40, 50, 60, 70]

print(numbers[-3:])

print(numbers[-6:-2])

4. Common list operations: concatenation, repetition, membership.

1. Definition

Python provides several common operations that can be performed on lists:

- Concatenation (+) → joining two or more lists.
- Repetition (*) → repeating the elements of a list.
- Membership (in / not in) → checking if an element exists in a list.

2. Concatenation

- We can use the + operator to combine two or more lists into one.

Example:

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

result = list1 + list2
print(result)
```

3. Repetition

- We can use the * operator to repeat elements of a list multiple times.

Example:

```
list1 = ["A", "B"]

result = list1 * 3
print(result)
```

4. Membership

- We can use the keywords in and not in to check if an element exists in a list.

Example:

```
fruits = ["apple", "banana", "cherry"]

print("apple" in fruits)
print("mango" in fruits)
print("grape" not in fruits)
```

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

1. Definition

Python provides built-in list methods to modify and manage list elements.
Four commonly used methods are:

- append() → Adds an element to the end of the list.
- insert() → Inserts an element at a specific position.
- remove() → Removes the first occurrence of a specific element.
- pop() → Removes an element by index (default is last element).

2. append()

- Syntax: list.append(element)
- Used to add a single element at the end of the list.

Example:

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)
```

3. insert()

- Syntax: list.insert(index, element)
- Adds an element at the given index, shifting other elements to the right.

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "orange")
print(fruits)
```

4. remove()

- Syntax: list.remove(element)
- Removes the first occurrence of the specified element.

Example:

```
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")
print(fruits)
```

5. pop()

- Syntax: list.pop(index)
- Removes and returns the element at the given index.
- If index is not provided, it removes the **last element**.

Example:

fruits = ["apple", "banana", "cherry"]
fruits.pop(1)
print(fruits)
fruits.pop()
print(fruits)

6. Iterating over a list using loops.

1. Definition

- **Iteration** means going through each element of a list **one by one**.
- In Python, we can use **for loops** or **while loops** to iterate over lists.
- Iterating is useful when we want to **process, display, or manipulate** each element.

2. Using a `for` Loop

- The `for` loop is the most common way to iterate over a list.

Syntax:

```
for element in list_name:
    # do something with element
```

Example:

```
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    print(fruit)
```

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

1. sort() Method

- The sort() method sorts the elements of a list in place (it changes the original list).
- By default, it sorts in ascending order.

Example:

numbers = [5, 2, 9, 1, 7]
numbers.sort()
print("Ascending:", numbers)

numbers.sort(reverse=True)
print("Descending:", numbers)

2. sorted() Function

- The sorted() function returns a new sorted list without changing the original list.
- It also supports the reverse=True parameter.

Example:

```
numbers = [8, 3, 6, 1]
ascending = sorted(numbers)
descending = sorted(numbers, reverse=True)

print("Original:", numbers)
print("Ascending:", ascending)
print("Descending:", descending)
```

3. reverse() Method

- The reverse() method reverses the elements of the list in place, without sorting them.

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
print("Reversed:", fruits)
```

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

1. Addition of Elements

- You can add elements using:
  - append() → adds one item at the end
  - insert() → adds an item at a specific index
  - extend() → adds multiple items from another list

Example:

```
fruits = ["apple", "banana"]
fruits.append("cherry")
fruits.insert(1, "mango")
fruits.extend(["orange", "grape"])
print(fruits)
```

2. Deletion of Elements

- You can remove elements using:
  - remove(value) → removes by value
  - pop(index) → removes by position (returns the removed value)
  - del → deletes by index or the entire list
  - clear() → empties the list

Example:

```
numbers = [10, 20, 30, 40, 50]
numbers.remove(30)
numbers.pop(2)
del numbers[0]
print(numbers)
numbers.clear()
print(numbers)
```

3. Updating Elements

- You can update list items by directly assigning new values to specific indexes.

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits)
```

4. Slicing a List

- Slicing allows you to access a range of elements using [start:end].
- Negative indexing can also be used.

Example:

```
numbers = [10, 20, 30, 40, 50, 60]
print(numbers[1:4])
print(numbers[:3])
print(numbers[-3:])
```

9. Introduction to tuples, immutability.

1. Introduction to Tuples

- A tuple is an ordered collection of elements, similar to a list.
- But unlike lists, tuples are immutable, meaning their elements cannot be changed once assigned.
- Tuples are defined using parentheses ( ) instead of square brackets [ ].

Example:

```
my_tuple = (10, 20, 30, 40)
print(my_tuple)
```

2. Immutability of Tuples

- Tuples cannot be changed once created.
- You cannot add, update, or delete individual elements.

Example:

```
t = (1, 2, 3)
t[1] = 10
```

However, you can create a new tuple by combining existing ones:

```
t1 = (1, 2, 3)
t2 = (4, 5)
t3 = t1 + t2
print(t3)
```

10. Creating and accessing elements in a tuple.

1. Creating Tuples

a) Tuple with Multiple Elements

```
t2 = (10, 20, 30, 40)
print(t2)
```

b) Tuple with Mixed Data Types

A tuple can hold different types of data like integers, strings, floats, and booleans.

```
t3 = (1, "apple", 3.14, True)
print(t3)
```

c) Tuple without Parentheses (Tuple Packing)

You can create a tuple without using parentheses — this is known as tuple packing.

```
t4 = 5, 10, 15, 20
print(t4)
```

2. Accessing Elements in a Tuple

You can access elements in a tuple using:

- Indexing
- Negative Indexing
- Slicing

a) Indexing

Example:

```
t = (10, 20, 30, 40, 50)
print(t[0])
print(t[3])
```

b) Negative Indexing

Negative indexing starts from the end of the tuple.
-1 represents the last element, -2 the second last, and so on.

Example:

```
t = (100, 200, 300, 400)
print(t[-1])
print(t[-3])
```

c) Slicing

Slicing allows you to access a range of elements using the syntax:

```
tuple[start:end]
```

- The start index is inclusive
- The end index is exclusive

Example:

```
t = (10, 20, 30, 40, 50, 60)
print(t[1:4])
print(t[:3])
print(t[3:])
print(t[-3:])
```

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

1. Tuple Concatenation

- Concatenation means joining two or more tuples to form a new tuple.
- The + operator is used to concatenate tuples.
- It creates a new tuple

Example:

```
tuple1 = (10, 20, 30)
tuple2 = (40, 50, 60)
result = tuple1 + tuple2
print("Concatenated Tuple:", result)
```

Output:

```
Concatenated Tuple: (10, 20, 30, 40, 50, 60)
```

2. Tuple Repetition

- Repetition means repeating the elements of a tuple multiple times.
- The * operator is used for repetition.
- It returns a new tuple containing repeated elements.

Example:

```
tuple1 = ("A", "B")
result = tuple1 * 3
print("Repeated Tuple:", result)
```

Output:

```
Repeated Tuple: ('A', 'B', 'A', 'B', 'A', 'B')
```

3. Tuple Membership

- Membership operators in and not in are used to test whether an element exists in a tuple.
- It returns True if the element is found, otherwise False.

Example:

```
tuple1 = (10, 20, 30, 40, 50)
print(20 in tuple1)
print(100 not in tuple1)
```

12. Accessing tuple elements using positive and negative indexing.

1. Positive Indexing

- Positive indexing starts from 0 for the first element.
- The index increases by 1 for each element.

Example:

```
fruits = ("apple", "banana", "cherry", "mango", "orange")

print(fruits[0])
print(fruits[1])
print(fruits[2])
print(fruits[4])
```

2. Negative Indexing

- Negative indexing starts from -1 for the last element.
- The index decreases by 1 as you move toward the beginning.

Example:

```
fruits = ("apple", "banana", "cherry", "mango", "orange")

print(fruits[-1])
print(fruits[-2])
print(fruits[-3])
```

13. Slicing a tuple to access ranges of elements.

1. Syntax of Slicing

```
tuple_name[start : end : step]
```

Parameter	Description
start	Index from where the slice begins. Default = 0
end	Index where the slice ends. Default = length of tuple
step	The interval between elements. Default = 1

2. Basic Slicing Examples

Accessing a range of elements

```
numbers = (10, 20, 30, 40, 50, 60, 70)
print(numbers[1:4])
```

14. Introduction to dictionaries: key-value pairs.

1. Introduction

A dictionary in Python is an unordered collection of data stored in key–value pairs.
It allows you to store, access, and modify data efficiently using unique keys instead of numeric indexes (as in lists or tuples).

Dictionaries are defined using curly braces {}, where each key is paired with a value using a colon (:).

2. Syntax of a Dictionary

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

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

1. Accessing Dictionary Elements

You can access dictionary elements using their keys in two ways:

a) Using Square Brackets [ ]

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
print(student["name"])   # Output: Ravi
print(student["course"]) # Output: BCA
```

2. Adding Elements to a Dictionary

You can add a new key–value pair by simply assigning a value to a new key.

Example:

```
student = {"name": "Ravi", "age": 21}
student["course"] = "BCA"
student["grade"] = "A"

print(student)
```

3. Updating Dictionary Elements

You can modify existing values or update multiple key–value pairs at once.

a) Updating a Single Value

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
student["age"] = 22
print(student)
```

b) Updating Multiple Values using update()

```
student.update({"age": 23, "grade": "A+"})
print(student)
```

4. Deleting Dictionary Elements

There are several ways to remove items from a dictionary:

a) Using del Statement (specific key)

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
del student["age"]
print(student)
```

b) Using pop() Method

Removes the specified key and returns its value.

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
removed_value = student.pop("course")
print("Removed:", removed_value)
print(student)
```

c) Using popitem() Method

Removes and returns the last inserted key–value pair.

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
student.popitem()
print(student)
```

d) Using clear() Method

Removes all items from the dictionary.

```
student.clear()
print(student)
```

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

1. The keys() Method

Definition:

The keys() method returns a view object that displays a list of all the keys in the dictionary.

Syntax:
```
dictionary.keys()
```

Example:
```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
print(student.keys())
```

Use in Loop:
```
for key in student.keys():
    print(key)
```

2. The values() Method

Definition:

The values() method returns a view object that displays all the values in the dictionary.

Syntax:
```
dictionary.values()
```

Example:
```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
print(student.values())
```

Use in Loop:

```
for value in student.values():
    print(value)
```

3. The items() Method

Definition:

The items() method returns a view object that displays all key–value pairs as tuples.

Syntax:

```
dictionary.items()
```

Example:

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
print(student.items())
```

Use in Loop:

```
for key, value in student.items():
    print(key, ":", value)
```

17. Iterating over a dictionary using loops.

1. Iterating Over Keys

By default, when you use a for loop on a dictionary, it iterates over the keys.

Example:

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}
for key in student.keys():
    print(key)
```

2. Iterating Over Values

To access only the values, use the values() method.

Example:

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}

for value in student.values():
    print(value)
```

3. Iterating Over Key–Value Pairs

To access both key and value together, use the items() method.
Each element returned by items() is a tuple containing (key, value).

Example:

```
student = {"name": "Ravi", "age": 21, "course": "BCA"}

for key, value in student.items():
    print(key, ":", value)
```

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

1. Using a Loop to Merge Lists

You can use a for loop to iterate over both lists and add elements to a dictionary.

Example:

```
keys = ["name", "age", "course"]
values = ["Ravi", 21, "BCA"]

# Create an empty dictionary
student = {}

# Using a loop to merge
for i in range(len(keys)):
    student[keys[i]] = values[i]

print(student)
```

2. Using zip() to Merge Lists

The zip() function pairs elements from two lists into tuples.
Then, dict() converts these pairs into a dictionary.

Example:

```
keys = ["name", "age", "course"]
values = ["Ravi", 21, "BCA"]

# Merge using zip
student = dict(zip(keys, values))

print(student)
```

19. Counting occurrences of characters in a string using dictionaries.

- ➢ A dictionary in Python can be used to count how many times each character appears in a string.

- Key → Character
- Value → Number of occurrences

This technique is useful in text analysis, frequency counting, and data processing.

1.Count Characters

1. Create an empty dictionary.
2. Loop through each character in the string.
3. Check if the character is already a key in the dictionary:
    - If yes, increment its value by 1
    - If no, add it to the dictionary with value 1

2. Example Program

```
text = "hello world"

char_count = {}

# Count characters
for char in text:
    if char in char_count:
        char_count[char] += 1
    else:
        char_count[char] = 1

print(char_count)
```

20. Defining functions in Python.

- ➢ A function in Python is a block of reusable code that performs a specific task. Instead of writing the same code again and again, we can put it inside a function and call it whenever needed.

Functions help to make programs modular, readable, and efficient**.**

- ❖ **Syntax of a Function**

```
def function_name(parameters):
    # function body
    statement(s)
    return value
```

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

Type 1: Function Without Parameters and Without Return Value

Definition:
A function that doesn’t take any input (parameters) and doesn’t return any value.
It just performs a task.

Syntax:

```
def function_name():
    # statements
```

Example:

```
def greet():
    print("Hello, Welcome to Python Programming!")

# Function call
greet()
```

Type 2: Function With Parameters and Without Return Value

Definition:
A function that takes inputs (parameters) but does not return anything.
It usually performs an action using the given data.

Syntax:

```
def function_name(parameter1, parameter2):
    # statements
```

Example:

```
def greet(name):
    print("Hello,", name, "Welcome to Python!")

# Function call
greet("Dharmesh")
```

Type 3: Function Without Parameters and With Return Value

Definition:
A function that takes no input but returns a value.

Syntax:

```
def function_name():
    return value
```

Example:

```
def get_pi():
    return 3.14159

# Function call
pi_value = get_pi()
print("Value of Pi =", pi_value)
```

Type 4: Function With Parameters and With Return Value

Definition:
A function that accepts parameters and returns a value.
This is the most commonly used type.

Syntax:

```
def function_name(parameter1, parameter2):
    return value
```

Example:

```
def add(a, b):
    return a + b

# Function call
result = add(10, 20)
print("Sum =", result)
```

22. Anonymous functions (lambda functions).

1. Introduction

In Python**,** anonymous functions are functions without a name**.**
They are created using the lambda keyword and are also known as lambda functions**.**

A lambda function is a small, single-line function that can have any number of arguments, but only one expression**.**

2. Syntax

```
lambda arguments: expression
```

23. Introduction to Python modules and importing modules.

1. Introduction

A module in Python is a file that contains Python code**,** such as functions, variables, and classes.
Modules help to organize code**,** promote reusability**,** and avoid repetition in large programs.

Instead of writing all the code in a single file, we can divide it into multiple modules and use them when needed.

2. importing modules

Purpose

- To use built-in, user-defined, or external modules.
- To make code reusable and organized.
- To avoid rewriting the same code.

1. Ways to Import Modules

1. import module_name – Imports the whole module.
2. from module_name import function – Imports specific parts only.
3. import module_name as alias – Gives the module a short name.
4. from module_name import * – Imports everything (not recommended).

2. Types of Modules

- Built-in modules: e.g., math, random, os
- User-defined modules: created by the user
- External modules: installed using pip (e.g., numpy, pandas)

3. Benefits

✅ Code reusability
✅ Better organization
✅ Easy maintenance
✅ Access to powerful libraries

24. Standard library modules: math, random.

1. math Module

The math module provides mathematical functions and constants.

Commonly Used Functions:

Function	Description
math.sqrt(x)	Returns the square root of *x*
math.pow(x, y)	Returns *x* raised to the power *y*
math.factorial(x)	Returns factorial of *x*
math.ceil(x)	Rounds *x* upward to nearest integer
math.floor(x)	Rounds *x* downward to nearest integer

Function	Description
math.pi	Returns the value of π (3.14159...)
math.e	Returns Euler’s number (2.718...)

2. random Module

The random module is used to generate random numbers or select random elements.

Commonly Used Functions:

Function	Description
random.random()	Returns a random float between 0 and 1
random.randint(a, b)	Returns a random integer between *a* and *b*
random.choice(sequence)	Returns a random element from a list or string
random.shuffle(list)	Randomly rearranges elements in a list
random.uniform(a, b)	Returns a random float between *a* and *b*

25. Creating custom modules.

❖ Create a Custom Module

1. Create a Python file with a .py extension (for example, mymodule.py).
2. Write functions, variables, or classes inside it.
3. Save the file in the same directory as your main program.
4. Import and use the module in another Python file using the import statement.